

Using Help

About online Help

Adobe Systems, Inc. provides complete documentation in the Adobe PDF Help system. The Help system includes information on all the tools, commands, and features for both Windows and Mac OS. The PDF format is designed for easy navigation online, and support for third-party screen readers compatible with Windows. The Help can also be printed as a desktop reference.

Navigating in Help

The Help will open in an Acrobat window with the bookmark pane open. If the bookmark pane is not open choose Window > Bookmarks. You can also navigate using the navigation bar, the index, or search the document.

At the top and bottom of each page is a navigation bar. Click Using Help to return to this introduction. Clicking Contents, or Index will take you to that section.

The Next Page ▶ and the Previous Page ◀ arrows let you move through the pages sequentially. Click Back to return to the last page you viewed. You can also use the navigation arrows in the Acrobat toolbar.

Using bookmarks, the table of contents, the index, and Find

The contents of Help are shown as bookmarks in the bookmark pane. To view subtopics, click the plus sign next to a bookmark. Each bookmark is a hyperlink to the associated section of the Help document.

To go to the information, click its bookmark. As the information is displayed in the document pane, its bookmark is highlighted.

You can turn highlighting on or off by selecting the Highlight Current Bookmark option from the bookmark pane menu.

To find a topic using the table of contents:

- 1 Click Contents in the navigation bar at the top or bottom of any page.
- 2 Click a topic on the Contents page to move to the first page of that topic.
- 3 In the bookmark pane, expand the topic to see its subtopics.

To find a topic using the index:

- 1 Click Index in the navigation bar at the top or bottom of any page.
- 2 Click the appropriate letter at the top of the page.

You can also expand the Index bookmark, and click the letter in the bookmark pane.

- 3 Locate your entry, and click the page number link to view the information.
- 4 To view multiple entries, click Back to return to the same place in the index.



To find a topic using the Find command:

- 1 Choose Edit > Find.
- 2 Enter a word or a phrase in the text box, and click OK.

Acrobat will search the document, starting from the current page, and display the first occurrence of the word or phrase you are searching for.

- 3 To find the next occurrence, choose Edit > Find Again.

Printing the Help file

Although the Help has been optimized for on-screen viewing, you can print pages you select, or the entire file.

To print, choose Print from the File menu, or click the printer icon in the Acrobat toolbar.

Contents

[Introduction 4](#)

[Overview 7](#)

[Writing Scripts 14](#)

[Behaviors 26](#)

[Movie Clips 42](#)

[Movie Clip Events and Event Handlers 64](#)

[Dynamic Data 77](#)

[Script Editor 85](#)

[Debugger 94](#)

[Reference 105](#)

[Glossary 248](#)

[Legal Notices 250](#)



Introduction

Overview of this guide

The *Adobe® LiveMotion™ 2.0 Scripting Guide* is your guide to enhancing compositions created with the LiveMotion user interface. By incorporating JavaScript code into your compositions, you can control animations and responses to user events in ways that would be impossible or extremely tedious to do with the user interface tools and menus alone. If you have created behaviors in LiveMotion 1.0, you will soon recognize the power of scripting in LiveMotion 2.0. With some practice and working with scripting language, you are bound to be a convert.

Early sections of this guide start with some simple examples to get you started with scripting right away. Just understanding how to create a simple composition that uses scripts may be all you need to know. Later sections take you through more advanced examples and cover the highlights of scripting LiveMotion 2.0 compositions.

What you should know

This guide assumes that you have an understanding of JavaScript syntax. If you do, the transition to writing scripts should be easy. The scripts that you write are JavaScript with a few differences to support exporting your `.liv` file to the SWF file format. “JavaScript in LiveMotion” on page 8 points out some of these differences.

If you need to learn JavaScript language fundamentals, such as what operators, variables, and looping mechanisms are, you will find a wealth of publications available online and at your local bookstore. “Where to go for more information” on page 6 lists several publications and some helpful Web sites.

Organization of this guide

This guide is organized as follows:

- “Introduction” on page 4 acquaints you with the *LiveMotion 2.0 Scripting Guide*, tells you what you should know before you start reading, summarizes section contents and organization, lists all the hands-on examples and where they are located in this guide, and provides references for additional information.
- “Overview” on page 7 introduces LiveMotion’s authoring environment, provides a high-level description of objects and movie clips, and points out the advantages of using scripting in LiveMotion compositions. In addition, this section compares JavaScript in LiveMotion with ActionScript and ECMA-standard JavaScript.
- “Writing Scripts” on page 14 gets you up and running. It describes basic ways you can manipulate objects through scripting. In the process, you learn where and how to add scripts to your compositions. The chapter uses very simple scripting examples. It is



meant to reach everyone who will be writing scripts, including those who are very new to scripting.

- “Behaviors” on page 26 provides procedures for creating scripts for each of the LiveMotion 1.0 behaviors.
- “Movie Clips” on page 42 describes how to create movie clips manually and programmatically, how to use the built-in movie clip methods and properties, how to create your own movie clip methods and properties, how to reference movie clips in the object hierarchy, and finally how to load and unload SWF files.
- “Movie Clip Events and Event Handlers” on page 64 describes how to write [system-based and user-generated] event handlers. This section provides several hands-on examples showing ways to create these handlers.
- “Dynamic Data” on page 77 describes how to create LiveMotion applications that dynamically accept user input and respond with the results of user queries within the LiveMotion movie clip or browser window.
- “Script Editor” on page 85 introduces and explains in detail how to use the Script Editor features to help you with writing scripts
- “Debugger” on page 94 describes the Debugger and Console window in detail.
- “Reference” on page 105 is the detailed reference to writing scripts. The chapter describes each global variable and function, each object and its associated methods and properties in the JavaScript extensions, and all the JavaScript core functions that are supported when writing scripts.
- “Glossary” on page 248 defines terms used in this guide.

Hands-on examples in this guide

This guide provides hands-on examples to get you involved in writing scripts that exercise pertinent concepts. You are encouraged to save your examples, but this is optional. A few of them are used again, but in those cases, the examples let you know if you should save results.

Here is a list of all the hands-on examples and their locations in this guide:

“Writing Scripts”

- “Hands-on example 2_1: Writing a keyframe script to the composition timeline” on page 17
- “Hands-on example 2_2: Writing a keyframe script to a movie clip timeline” on page 19
- “Hands-on example 2_3: Creating a simple event handler” on page 22
- “Hands-on example 2_4: Initializing a movie clip property” on page 22
- “Hands-on example 2_5: Creating a bounds check” on page 23
- “Hands-on example 2_6: Creating a state script” on page 24

“Behaviors”

- “Hands-on example 3_1: Changing movie clip states” on page 32
- “Hands-on example 3_2: Creating a preloader” on page 37

“Movie Clips”

- “Hands-on example 4_1: Mouse trailer” on page 51

“Events and Event Handlers”

- “Hands-on example 5_1: Using system-based event handlers to rotate a movie clip” on page 65
- “Hands-on example 5_2: Programmatic bounce” on page 66
- “Hands-on example 5_3: Creating an onKeyDown event handler” on page 69
- “Hands-on example 5_4: Creating a simple button event handler” on page 72
- “Hands-on example 5_5: Creating a toggle button” on page 73
- “Hands on example 5_6: Experimenting with automatically generated button event handlers” on page 74

Where to go for more information

For more information on LiveMotion

See the *LiveMotion 2.0 User Guide* for detailed information on using Adobe Online to access a resources that will help you with using LiveMotion.

For information on JavaScript

Flanagan, David, *JavaScript The Definitive Guide, Third Edition*, O'Reilly & Associates, 1998 (ISBN: 1-56592-392-8)

Moncur, Michael, *Teach Yourself JavaScript in 24 Hours, Second Edition*, Sams, 2000

Goodman, Danny, *JavaScript Bible, Fourth Edition*, IDG Books, 2000

Smith, Dori and Tom Negrino, *JavaScript For the World-Wide Web*

Wyke, Gilliam, and Ting, *Pure JavaScript*, Sams, 1999

Web sites

Check <http://www.adobe.com> for updated lists of reference sites.

See <http://www.moock.org> for ActionScript help to assist you in learning about LiveMotion scripting.

Overview

Script authoring

LiveMotion 2.0 is a script authoring tool. It makes use of a JavaScript editor, interpreter, and debugger, which enable you to create, preview, troubleshoot, and export the scripted contents of your *composition* (.liv file).

Through the Script Editor you can write scripts to the composition and movie clip timelines. In addition, you can write scripts that respond to events such as pressing a key or loading a movie clip. The Script Editor provides guidance in using the JavaScript core syntax and extensions. It lists all the current movie clips, labels, and states defined in your composition, provides you with the ability to set breakpoints, and assists you in locating all the scripts that are currently written.

LiveMotion 2.0 also includes a Debugger that you can use in Preview mode to troubleshoot your compositions before they are exported. The Debugger not only locates and identifies errors but provides you with a number of significant debugging features including the ability to view variable values, set script breakpoints, and step through lines of a script as they are executed. When you are satisfied with the way a composition is working, you can export it to the SWF file format for viewing in the standalone Flash Player or in the Flash Player plug-in installed in your Netscape or Microsoft® Internet Explorer browser. Exporting the .liv file causes the JavaScript it contains to be converted to ActionScript and embedded in the exported SWF file.

LiveMotion objects

As you recall from the *LiveMotion 2.0 User Guide*, objects are the basic element of a composition, and they have a hierarchical organization. Movie clips, the focus of this guide, are also objects. And they can be manipulated manually in all the ways you have already learned about in the User Guide, plus new ways.

Writing scripts to objects

You can manipulate objects through the JavaScript scripting language. This opens up all sorts of new possibilities for handling objects. However, you can only write scripts to a certain type of object, namely, the movie clip.

A movie clip starts out as a “regular” (unscriptable) object. To access it through scripting, you must convert the object into a movie clip. A movie clip has its own timeline so that it can play independently of the composition timeline and independently of any parent timeline (in the case of nested movie clips). When you add states to an object, LiveMotion automatically converts the object into a movie clip for you. Movie clips are equivalent to the time-independent objects and time-independent groups in LiveMotion 1.0.



Extending functionality

By writing scripts, you can perform many functions on a movie clip that are equivalent to those you can perform without using scripting. You can, for example, set a movie clip's vertical and horizontal position properties. This capability is equivalent to setting the position stopwatch and creating animation keyframes. By setting properties through scripts, you can perform functions such as changing an object's opacity, rotation, and scale—to name a few. However, this is just the beginning of what you can do through scripting.

Scripting enables you to control how your composition responds to events, use logic to compare values and make decisions based on those values, easily repeat long processes using a variety of looping mechanisms, respond to user events such as mouse and keyboard changes, and encapsulate tasks into functions that can be called by any number of movie clips anywhere in a composition. Not only can you write scripts that interact with the user, you can write scripts that interact with servers. Through scripting, you can get data from a server and post data to a server. The information obtained from a server can be used to dynamically update your composition. You will find it difficult, if not, impossible, to perform most these tasks through the use of keyframes (and basic LiveMotion 1.0 behaviors). These programmatic controls, available through the JavaScript language, *extend* what you can create with keyframes and enable you to fine tune your composition.

Script locations

You can attach scripts at different locations in your composition to achieve the result that you are after, whether that be animation, user interaction, or interaction with a server. These locations are:

- On keyframes
- In event handlers
- In state change handlers

Although using labels is not a script writing technique in and of itself, you typically use labels in combination with scripts to redirect the flow of execution of a timeline to a frame with the identifying label. For example, this script sends the playhead of `myClip`'s timeline to the frame labeled "Start":

```
myClip.gotoAndPlay("Start");
```

For more information on writing scripts to various locations in your composition, see "Writing Scripts" on page 14. That section introduces you to script writing and provides short exercises that you can work through.

JavaScript in LiveMotion

The LiveMotion scripting environment is based on JavaScript, but it also is compatible with ActionScript and ECMA-standard JavaScript (with a few caveats). Table 1 describes these caveats.

Table 1 JavaScript as Implemented in LiveMotion, Compared to ActionScript and ECMA-Standard JavaScript

Characteristic	ActionScript vs. the JavaScript implementation in LiveMotion	ECMA-standard JavaScript vs. the JavaScript implementation in LiveMotion
Case	In ActionScript, keywords are case sensitive, but variables and other identifiers are not. JavaScript as implemented in LiveMotion behaves the same way.	ECMA-standard JavaScript is entirely case sensitive.
switch/case construct	ActionScript does not support the switch/case construct. JavaScript and the LiveMotion scripting environment do.	ECMA-standard JavaScript and JavaScript as implemented in LiveMotion both support the switch/case syntax.
States	With the <code>movieClip.lmSetCurrentState()</code> method, LiveMotion supports the setting of states of movie clips using scripting code. ActionScript does not support this.	ECMA-standard JavaScript has no language facilities to deal with states of objects in this sense.
<code>eval()</code> global function	The ActionScript and the LiveMotion scripting environments implement the <code>eval()</code> global function in the same way. (See “Reference” on page 105.)	ECMA-standard JavaScript implements an expanded <code>eval()</code> function.
Support for Unicode	ActionScript and JavaScript as implemented in LiveMotion do not support Unicode.	ECMA-standard JavaScript supports Unicode.
Maximum number of nested <code>with</code> statements.	ActionScript and JavaScript as implemented in LiveMotion support a maximum of 8 levels of nested <code>with</code> statements.	ECMA-standard JavaScript supports any number of levels of nested <code>with</code> statements.
Exception handling	ActionScript and LiveMotion do not support exception handling.	ECMA-standard JavaScript supports error objects and exception classes.
Function constructor	ActionScript and LiveMotion do not support the <code>Function</code> constructor. However, object-based functions can be created. For example: <code>this.myFunction = function() {}</code>	ECMA-standard JavaScript supports the <code>Function</code> constructor.

Characteristic	ActionScript vs. the JavaScript implementation in LiveMotion	ECMA-standard JavaScript vs. the JavaScript implementation in LiveMotion
Frame numbers	<p>In ActionScript, the following global functions and movie clip methods accept either frames or labels as arguments. In LiveMotion, only labels are used.</p> <p><code>gotoAndPlay()</code> global function</p> <p><code>gotoAndStop()</code> global function</p> <p><code>movieClip.gotoAndPlay()</code> method</p> <p><code>movieClip.gotoAndPlay()</code> method</p> <p>In addition, the <code>lmFrameofLabel()</code> global function is available in LiveMotion but not in ActionScript. In LiveMotion, it is used to return the frame number of the label that is passed in as an argument to the call. <code>lmFrameofLabel()</code> only works for labels on the <code>_root</code> timeline.</p>	ECMA-standard JavaScript has no language facilities to deal with frames or labels in this sense.

Characteristic	ActionScript vs. the JavaScript implementation in LiveMotion	ECMA-standard JavaScript vs. the JavaScript implementation in LiveMotion
Syntax	<p>JavaScript as implemented in LiveMotion supports most ActionScript syntax. For a complete listing, see “Reference” on page 105. The following ActionScript syntax is not supported, either because it was deprecated in Flash 5, or for other reasons.</p> <p><code>call()</code> function</p> <p><code>chr()</code> function</p> <p><code>getProperty()</code> function</p> <p><code>_highquality</code> property</p> <p><code>ifFrameLoaded()</code> function</p> <p><code>int()</code> function</p> <p><code>nextScene()</code> function</p> <p><code>prevScene()</code> function</p> <p><code>print()</code> function</p> <p><code>printAsBitmap()</code> function</p> <p><code>printAsBitmapNum()</code> function</p> <p><code>printNum()</code> function</p> <p><code>random()</code> function</p> <p><code>setProperty()</code> function</p> <p><code>set</code> statement</p> <p><code>setVariable()</code> function</p> <p><code>substring()</code> function</p> <p><code>tellTarget()</code> function</p> <p><code>toggleHighQuality()</code> function</p> <p><code>\$version()</code> function</p> <p>Most common string operators (e.g., <code>add</code> and <code>and</code>)</p> <p>Note that some deprecated Flash 5 calls can be duplicated using JavaScript syntax. For example, the following code shows how you can mimic <code>getProperty()</code> and <code>setProperty()</code>:</p> <pre>movieClip.property = value; var value = movieClip.property</pre>	<p>ECMA-standard JavaScript and JavaScript as implemented in LiveMotion share the same basic objects, properties, and methods, as described in “Reference” on page 105.</p> <p>Note that in LiveMotion a <code>Date()</code> object cannot be constructed using a text string to provide the current date.</p>

Characteristic	ActionScript vs. the JavaScript implementation in LiveMotion	ECMA-standard JavaScript vs. the JavaScript implementation in LiveMotion
<code>onClipEvent()</code> movie clip event handlers	<p>ActionScript supports the <code>onClipEvent()</code> movie clip event handlers:</p> <ul style="list-style-type: none"><code>load</code><code>unload</code><code>enterFrame</code><code>mouseMove</code><code>mouseDown</code><code>mouseUp</code><code>keyDown</code><code>keyUp</code><code>data</code> <p>LiveMotion supports the equivalents of the ActionScript <code>onClipEvent()</code> movie clip event handlers:</p> <ul style="list-style-type: none"><code>onLoad</code><code>onUnload</code><code>onEnterFrame</code><code>onMouseMove</code><code>onMouseDown</code><code>onMouseUp</code><code>onKeyDown</code><code>onKeyUp</code><code>onData</code> <p>Note: The <code>onData</code> event handler is not available from <code>_root</code>.</p>	<p>ECMA-standard JavaScript doesn't support movie clip events.</p>

Characteristic	ActionScript vs. the JavaScript implementation in LiveMotion	ECMA-standard JavaScript vs. the JavaScript implementation in LiveMotion
on() button event handlers	<p>ActionScript supports the on() button event handlers for the button object:</p> <pre>press release releaseOutside rollOver rollOut dragOver dragOut</pre> <p>LiveMotion supports the equivalents of the ActionScript on() button event handlers for all movie clips (in LiveMotion, a button is simply another movie clip—there is no separate button object):</p> <pre>onButtonPress onButtonRelease onButtonReleaseOutside onButtonRollOver onButtonRollOut onButtonDragOver onButtonDragOut</pre>	ECMA-standard JavaScript doesn't support movie clip events.
Evaluating undefined as a number	In ActionScript, evaluating <code>undefined</code> as a number returns <code>0</code> . LiveMotion does the same.	In ECMA-standard JavaScript, evaluating <code>undefined</code> as a number returns <code>undefined</code> .
Evaluating undefined as a string	In ActionScript, evaluating <code>undefined</code> as a string returns <code>" "</code> . LiveMotion does the same.	In ECMA-standard JavaScript, evaluating <code>undefined</code> as a string returns <code>NaN</code> .
Boolean value of non-empty strings	In ActionScript, only strings that can be converted to valid non-zero numbers convert to <code>true</code> .	In ECMA-standard JavaScript, all non-empty strings convert to <code>true</code> .

Writing Scripts

Introduction to script writing

This section introduces you to some simple examples of writing movie clip scripts. It emphasizes where you place scripts, as script placement determines when a script gets called. Scripts are placed at three locations. These are:

- Script keyframes
- Event handlers
- State change handlers

In addition, this section discusses labels, which are frequently used in conjunction with scripting.

The section begins with a brief overview of the Script Editor user interface. To acquaint you with the functionality provided by the Script Editor, each example is presented as an exercise that you can work through yourself. You are also introduced to movie clip referencing and *some* basic JavaScript syntax, although a tutorial on JavaScript basics is beyond the scope of this guide. Understanding JavaScript is a prerequisite if you want to do any serious LiveMotion scripting.

Script Editor overview

You will be using the Script Editor to write your scripts and to locate information. Figure 1 shows the Script Editor window. The callouts identify its main functionality.

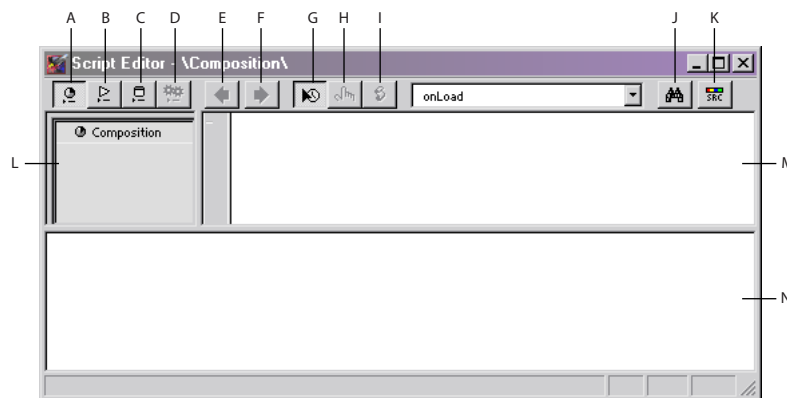


Figure 1 Script Editor main window

- A.** Movie clip navigator **B.** Scripting syntax helper **C.** Composition browser
D. Automation syntax helper **E.** Go to previous script **F.** Go to next script
G. Handler scripts **H.** State scripts **I.** Keyframe scripts **J.** Find **K.** Syntax highlighting
L. Scripting helper window **M.** Script window **N.** Description window



Table 2 briefly describes each of the control buttons and windows shown in the Script Editor window.

Table 2 Script editor buttons and windows

Button or window	Description
Movie clip navigator	Lists all the movie clips in a composition in hierarchical order. Selecting a movie clip in this window allows you to see and edit scripts on that movie clip.
Scripting syntax helper	Lists the LiveMotion 1.0 Behaviors, ActionScript syntax, and JavaScript syntax. Selecting an item in the list displays a brief description of the argument in the Description window. Double-clicking a syntax entry adds the item's syntax to the current script.
Composition browser	Lists all the movie clips, labels, and states in the composition. Selecting an item in the list displays the reference text that will be entered in the Script window. Double-clicking a movie clip, label, or state adds the respective movie clip reference, label name, or state name to the current script.
Automation syntax helper	Lists and describes all the global objects and properties in the JavaScript core that are supported by automation scripting and all predefined objects, their methods, and properties in the Automation scripting DOM. This button is available when the export format is Live Tab when you are editing an automation script. For details on automation scripts and Live Tabs, see the LiveMotion 2.0 SDK.
Go to previous script	Switches the script view to the previously edited script. This button works like the Back button in a Web browser.
Go to next script	Switches the script view to the more recently edited script. This button works like the Forward button in a Web browser.
Handler scripts	Lists all the event handlers in the drop-down menu for which you can write scripts. This button, as well as the State scripts and Keyframe scripts buttons (described below), display a blue triangle when they contain scripts.
State scripts	Lists all states in the drop-down menu that are defined for the current movie clip (movie clip selected in the Movie clip navigator). The list contains the normal state, and it can include the predefined states over, down, and out, plus any custom states defined for the movie clip.
Keyframe scripts	Lists all script keyframes in the drop-down menu for the current movie clip.
Drop-down menu	Displays the keyframes, event handlers, or states for the current movie clip. The contents displayed depend on which of the previous three buttons is selected. Items in this menu will display an asterisk if scripts exist on them.
Find	Opens a dialog for finding and replacing text strings in the current script.

Button or window	Description
Syntax highlighting	Turns syntax highlighting on and off.
Script window	Displays existing scripts and new scripts that you write to the current movie clip.
Description window	Displays brief descriptions of the syntax listed in the Scripting syntax helper.
Scripting helper window	Displays contents of the Scripting Editor's Movie clip navigator, syntax helper, and browser buttons. The contents displayed are dependant on which of the buttons is selected.

Using labels

What is a label?

A *label* is a string identifier, or name, that references a frame in a timeline. You can use labels as arguments in scripts that you write. You could, for example, create a label called "right here" on a particular frame. With the label in place, you can write a script that sets the current frame of a timeline to the frame marked with the label "right here." Labels don't have to be used in scripts; they can be used simply to annotate a timeline. For example, you could apply the label "Accelerate" to a frame to identify where an object appears to pick up speed.

Guidelines for creating label names

To create a label name, follow these guidelines:

- The first character of a label name must be in this set [a-z, A-Z, _, \$]. It must not be a number.
- The remaining characters include the characters in the above set plus the numbers 0 through 9.

Note: Labels names that start with invalid characters will automatically have an underscore (_) character added to the beginning of the name.

How to create labels

To create a label:

- 1 Display the timeline to which you want to add a label.
- 2 Move the current-time marker to the frame to which you want to add a label.
- 3 Click the Labels button in the timeline. See Figure 2.
- 4 Enter a name for the label in the text box and click OK.

The label name and icon appear on the timeline at that frame.

You can duplicate, rename, move, or delete labels. See the *LiveMotion 2.0 User Guide* for details.

Using a label in a script

For examples of using labels in scripts, see “Hands-on example 2_1: Writing a keyframe script to the composition timeline” on page 17 and “Hands-on example 2_2: Writing a keyframe script to a movie clip timeline” on page 19.

Using script keyframes

What are script keyframes?

A *script keyframe* is a frame in a timeline to which a script is added. When the player head enters that frame during playback, the script executes.

How to create script keyframes

To add a script to a keyframe:

- 1 Navigate to the timeline where you want to add the script keyframe.
- 2 In the Timeline window, move the current-time marker to the specified frame.

Note: Optionally, click the Labels button, and enter a name for the point in time where the script will be added to the timeline.

- 3 Click the Scripts button to the left of the timeline to create a script keyframe at the current-time marker. This also opens the Script Editor.

Hands-on example 2_1: Writing a keyframe script to the composition timeline

This example uses script keyframes and a label. A script written to the composition timeline moves a movie clip horizontally across the Composition window.

To use script keyframes on the composition timeline:

- 1 Create a new document in LiveMotion. Save the file as `Ex2_1.liv`.
- 2 Bring up the Timeline window by choosing Timeline > Composition Window from the main menu. Alternately, you can use Ctrl+T (Windows®) or Command+T (Mac OS).
- 3 Create an ellipse in the Composition window, and select it.

Note: By default, the object is selected after you create it.

- 4 Choose Object > Movie Clip from the main menu to convert the object into a movie clip. Alternately, you can click the “Make selected objects movie clips” button located at the bottom of the Timeline window.

A movie clip icon appears to the left of the object name in the Timeline window.

Note: To be scriptable, an object must be converted into a movie clip!

5 Select the object name in the timeline, press Enter, and enter in the new name “Ball” into the text box. Press OK.

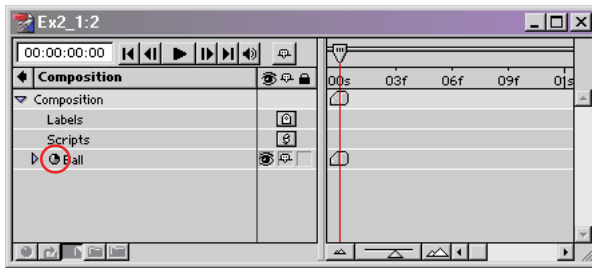


Figure 2 Timeline window showing the movie clip icon to the left of Ball

6 In the Timeline window, be sure the current-time marker is set to frame 0.

7 Click the Scripts button to add a script keyframe at frame 0.

This also brings up the Script Editor. With the Script Editor window displayed, you can add scripts to the script keyframe you just created.

8 Write a script to the script keyframe at frame 0 that will move Ball 5 pixels to the right. Here is a script that does this:

```
_root.Ball._x += 5;
```

In the script, `_root.Ball` is the absolute reference to the movie clip named `Ball`. `_root` represents the composition timeline. All movie clips placed on `_root`'s timeline can be accessed by name as properties of `_root`. Thus we can access `Ball` by saying `_root.Ball`. (For details on `_root` and absolute references, see “Movie clip addressing” on page 45.) `_x` is the horizontal position property of `Ball`. It is one of several built-in movie clip properties. (For details, see “Movie clip properties and methods” on page 48.) The operator `(+=)` is just a shorthand way to write the code:

```
_root.Ball._x = _root.Ball._x + 5;
```

9 With the current-time marker still at frame 0, click the Labels button in composition timeline. Enter Start in the text box, and click OK to create a label named Start at frame 0.

Note: When you create the label on the timeline frame, do not enclose the label name in quotation marks. However, when you provide the value for label (which is of type string) as a method argument, you must enclose the name in quotation marks to specify it as a string literal. This is done in step 13 of this example.

10 Move the current-time marker to frame 1.

11 Drag the endpoint of the composition timeline so that it ends at frame 1.

This also extends the endpoint of Ball’s duration bar so that it ends at frame 1.

12 Click the Scripts button to create a script keyframe at frame 1. See Figure 3. This also opens the Script Editor window (if it is not already open).

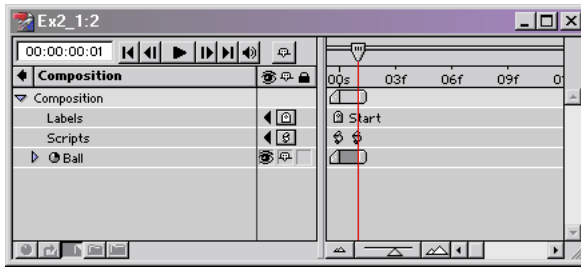


Figure 3 Timeline window showing label and script keyframe at frame 1 and script keyframe at frame 2

13 Enter the following code in the Script window:

```
_root.gotoAndPlay("Start");
```

`gotoAndPlay()` is a movie clip method that jumps a movie clip's timeline to a specific label and plays the timeline from the frame associated with the label. In this case, it jumps to the label "Start" on the composition timeline (`_root`).

Note: When you created the label on the timeline (step 9), you did not enclose the label name in quotation marks. However, when you provide the string value for label to `gotoAndPlay()`, you must enclose the name in quotation marks.

14 Preview the movie clip by switching to Preview mode or by exporting your composition to the Flash Player.

When the composition is previewed, the script you added at frame 0 moves Ball 5 pixels to the right on the screen. When execution reaches frame 1, the `gotoAndPlay()` statement moves the current-time marker to the frame labeled "Start" (in this case frame 0) and plays the timeline. At this point the script on frame 0 executes again.

You can adjust the speed of Ball by changing the value added to `_x` in the script to a new value.

This concludes your first scripted composition!

Hands-on example 2_2: Writing a keyframe script to a movie clip timeline

This example writes a script to the movie clip's own timeline rather than to the composition timeline. The results are the same as before. The difference is that, in the previous example, `_root` moved the Ball movie clip. In this example, the movie clip moves itself.

To write a keyframe script to the timeline:

1 Repeat steps 1 through 5 of "Hands-on example 2_1: Writing a keyframe script to the composition timeline" on page 17 to create a movie clip named Ball. Save this file as

Ex2_2.liv.

2 Double click Ball in the composition timeline to open its own timeline. In the movie clip's timeline, be sure the current-time marker is set to frame 0. See Figure 4.

- 3** Click the Scripts button in the Timeline window to insert a script keyframe at frame 0. This also brings up the Script Editor.

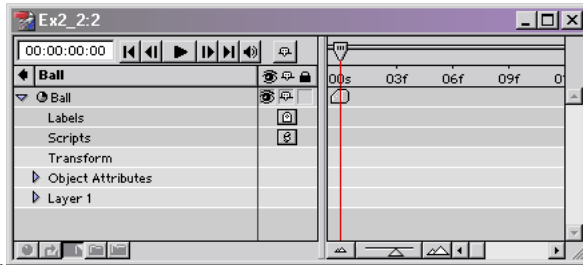


Figure 4 Ball movie clip timeline

- 4** Write this script in the Script window to move Ball 5 pixels to the right.

```
this._x += 5;
```

The following keyword in the above statement refers to the movie clip to which the script is added—in this case, the movie clip Ball:

```
this
```

Thus, the statement is incrementing Ball's horizontal position property.

You can also use the absolute reference as you did in the previous example in “Hands-on example 2_1: Writing a keyframe script to the composition timeline” on page 17. The absolute reference would appear as:

```
_root.Ball._x += 5;
```

If, however, the object hierarchy for Ball changes (that is, Ball is placed in a movie clip *group*), the absolute reference would no longer be valid. (For details on how movie clip groups change the object hierarchy, see “Effect of creating a movie clip and a movie clip group” on page 43.)

- 5** With the current-time marker still at frame 0 in the Timeline window, click the Labels button. Enter Start in the text box, and click OK to add the label to frame 0.
- 6** Move the current-time marker to frame 1, and drag the end point of Ball's timeline so that it ends at frame 1.
- 7** Create a script keyframe at frame 1, and enter the following code in the Script window:
- ```
this.gotoAndPlay("Start");
```

- 8** Preview the movie clip.

Ball moves across the screen just as it did in the previous example. The movie clip advances its horizontal position with each successive execution of the script.

## Using event handlers

### What are event handlers?

An event handler is script that is run as a result of a user action or a system-based event. For example, you can write an event handler that executes every time the user presses the mouse button or passes the mouse cursor over the movie clip. System-based events such as `onLoad` and `onData` occur as a result of composition playback or loading variables into a movie clip.

Table 3 lists all the event handlers and describes the events they handle.

Table 3 Movie clip events

| Event handler                       | Event                                                                                                                                             |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>onLoad</code>                 | First appearance of a movie clip in the composition. You can write scripts here to initialize and declare variables and functions.                |
| <code>onUnload</code>               | The first frame after the movie clip is removed from the composition.                                                                             |
| <code>onEnterFrame</code>           | Each time the playhead enters a frame, before the frame is rendered, while the movie clip is in the composition.                                  |
| <code>onMouseMove</code>            | Any movement of the mouse cursor while the movie clip is in the composition.                                                                      |
| <code>onMouseDown</code>            | Pressing the mouse button while the movie clip is in the composition.                                                                             |
| <code>onMouseUp</code>              | Releasing the mouse button while the movie clip is in the composition.                                                                            |
| <code>onKeyDown</code>              | Pressing a key while the movie clip is in the composition.                                                                                        |
| <code>onKeyUp</code>                | Releasing a key while the movie clip is in the composition.                                                                                       |
| <code>onData</code>                 | When the loading of variables into a movie clip is complete or a portion of a loaded movie completes loading into a movie clip.                   |
| <code>onButtonPress</code>          | Clicking the mouse button while the mouse cursor is on the movie clip.                                                                            |
| <code>onButtonRelease</code>        | Releasing the mouse button while the mouse cursor is on the movie clip.                                                                           |
| <code>onButtonReleaseOutside</code> | After pressing the mouse button while the mouse cursor is on the movie clip, moving the mouse cursor off the movie clip and releasing the button. |
| <code>onButtonRollOver</code>       | Moving the mouse cursor on the movie clip.                                                                                                        |
| <code>onButtonRollOut</code>        | Moving the mouse cursor off the movie clip.                                                                                                       |
| <code>onButtonDragOver</code>       | After pressing the mouse button while the mouse cursor is on the movie clip, moving the cursor off and then back on the movie clip.               |
| <code>onButtonDragOut</code>        | After pressing the mouse button while the mouse cursor is on the movie clip, moving the mouse cursor off the movie clip.                          |

## How to add a script to an event handler

### To add a script to an event handler:

- 1 Select a movie clip in the timeline or in the composition.
- 2 Choose Scripts > Script Editor to open the Script Editor. Alternately, you can use Ctrl+J (Windows) or Command+J (Mac OS).
- 3 In the Script Editor, click the Handler scripts button to display the drop-down menu of events.
- 4 Select the handler name from the list for which you want to write a handler.
- 5 Write the script in the Script window.

## Hands-on example 2\_3: Creating a simple event handler

This hands-on example adds the same movement to the movie clip Ball as the previous keyframe script examples did. See “Hands-on example 2\_1: Writing a keyframe script to the composition timeline” on page 17 and “Hands-on example 2\_2: Writing a keyframe script to a movie clip timeline” on page 19. However, it uses an event handler to call the script that moves Ball.

### To create an event handler:

- 1 Repeat steps 1 through 5 of “Hands-on example 2\_1: Writing a keyframe script to the composition timeline” on page 17 to create a movie clip named Ball. Save this file as Ex2\_3.liv.
- 2 Choose Scripts > Script Editor to open the Script Editor.
- 3 In the Script Editor, click the Handler scripts button to display the drop-down menu of event handler names.
- 4 Select the onEnterFrame handler, and enter this script to move Ball horizontally.

```
this._x += 5;
```

This onEnterFrame event handler script causes Ball to move itself each time the playhead enters a frame.

- 5 Preview the composition. The ball moves horizontally across the Composition window.
- 6 Save this file for the next two hands-on exercises.

## Hands-on example 2\_4: Initializing a movie clip property

This example builds on the previous one. It uses Ball's onLoad event handler to explicitly set the horizontal starting position of Ball and to initialize a property containing the speed that Ball will move. For this example, open Ex2\_3.liv.

### To initialize a property:

- 1 Select Ball, and choose Scripts > Script Editor to open the Script Editor.
- 2 Click the Handler scripts button, and select the onLoad event. Enter this script:

```
this._x = 100; //sets the initial position of Ball
this.speed = 5;
```

The first statement in this onLoad event handler script sets the initial horizontal position of Ball to 100. The second creates a new property of Ball called `speed` and assigns it the value 5.

**3** With the Handler scripts button still toggled on, select the onEnterFrame handler from the drop-down menu. This brings up the event handling script that moves Ball.

```
this._x += 5;
```

Change the script to:

```
this._x += speed;
```

**4** Preview the results.

**5** Save this file as `Ex2_3.liv` for use in the next hands-on exercise.

Except for setting Ball's initial position, the behavior is the same as in the previous exercise. Ball moves horizontally across the Composition window.

## Hands-on example 2\_5: Creating a bounds check

As another variation on the previous example, you can modify the onEnterFrame event handler to do a bounds check to be sure Ball doesn't move out of the Composition window.

**To create a bounds check:**

**1** Open the file `Ex2_3.liv` that you created in a previous exercise.

**2** Select Ball in the Timeline window, and choose Scripts > Script Editor to open the Script Editor.

**3** Click the Handler scripts button, and select onEnterFrame from the drop-down menu of event handlers. This brings up Ball's event handling script:

```
this._x += speed;
```

**4** To this script, add these `if` statements.

```
if(this._x > 550)
 this.speed = -5;
if(this._x < 0)
 this.speed = 5;
```

**5** Preview.

Ball moves back and forth horizontally across the Composition window. You should adjust the value 550 to reflect your Composition window's actual width. Check Composition Settings to determine the width.

## Using state scripts

### What are state scripts?

Thus far, the examples in this section have illustrated adding scripts to:

- The composition timeline using its Labels and Scripts buttons
- Movie clip timelines using its Labels and Scripts buttons
- Event handlers

From working with the LiveMotion 1.0 user interface, recall that you can create rollover states for an object. Scripts also can be added to these states. The state script is executed each time the object changes to the state to which the script is added.

## How to add scripts to states

### To add a script to a state:

- 1 Select the object.
- 2 Open the States palette to view the movie clip states.
- 3 In the States palette, select the movie clip state to which you want to add a script.
- 4 Click the Scripts button in the palette.

This opens the Script Editor with the correct state script displayed.

- 5 Write the script in the Script window.

## Hands-on example 2\_6: Creating a state script

This example is similar to the keyframe examples you have created so far. Using the States palette, you create an over state, which, for effect, you can change to a different color. Then you write a script that moves the Ball one direction in the normal state and another, in the over state.

### To create the state script:

- 1 Repeat steps 1 through 5 of “Hands-on example 2\_1: Writing a keyframe script to the composition timeline” on page 17 to create a movie clip named Ball.
- 2 Using the States palette, create an over state for the movie clip. Give it a different fill color, so you can more easily recognize the movement in the over state during playback.
- 3 In the States palette, select the over state.

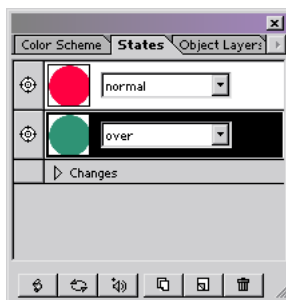


Figure 5 States palette with over state selected

- 4 Click the Scripts button at the bottom of the palette.

This opens the Script Editor at the location where you can add a script for the over state.

- 5 Enter the following code to move the movie clip 5 pixels to the right:

```
this._x += 5;
```



**6** Select normal from the Script Editor's drop-down menu of states, and enter the following code:

```
this._y += 20;
```

This moves the movie clip vertically.

**7** Preview the composition.

Ball first appears in its normal state. It does not move until you first pass the mouse over it. Try this a few times. Each time the mouse is moved over Ball, it moves five pixels to the right. Moving the mouse off Ball causes the movie clip to return to its normal state. Each time Ball enters its normal state, it moves vertically downward 20 pixels.

# Behaviors

---

## Introduction to behaviors

This section describes how you can create LiveMotion 1.0 behaviors in LiveMotion 2.0. It is meant to help you move on to a new way of looking at what behaviors really are.

In LiveMotion 1.0, behaviors did everything from playing and stopping compositions to entertaining the viewer with a looping movie clip while a lengthy, complex animation is loading. Traditionally, behaviors executed when either a movie clip reached a certain point on its timeline or when a movie clip entered a certain state. In LiveMotion 2.0, behaviors have evolved into JavaScript code. To assist you in your transition to writing scripts, this section explains where you can add scripts and the implications of adding the scripts in these locations. It provides an overview of how to add, open, and remove scripts. Then for each LiveMotion 1.0 behavior, the section provides a procedure for implementing that behavior in LiveMotion 2.0. As additional help, you are provided guidance using the Scripting syntax helper to access the LiveMotion 1.0 behaviors and the LiveMotion 2.0 code to which each behavior maps.

Even if you are new to LiveMotion, it will benefit you to read this section to learn how LiveMotion 1.0 behaviors are implemented in JavaScript, because you can incorporate their functionality into any scripts that you write. You are not required to know anything about LiveMotion 1.0 behaviors to create the examples in this chapter, which can instead serve as simple examples to start you down the road to scripting.

## Working with scripts that replace behaviors

This section provides procedures for adding, opening, and deleting scripts from keyframes and states.

**Note:** In LiveMotion 2.0, you also can write scripts to handle events. Event handling is made possible in LiveMotion 2.0 because of its support for scripting. For details on creating event handlers, see “Movie Clip Events and Event Handlers” on page 64.

## The effect of writing scripts to movie clip timelines versus movie clip states

You can write scripts to movie clip timelines or to movie clip states, depending on the effect that you are after. To prepare you for working with scripts, you should understand these concepts:

- Timelines have *script keyframes* (that is, script icons on timeline frames)
- States have timelines



When you write a script to a movie clip timeline, you write that script to a specific timeline frame. The frame is called a script keyframe. During execution of the `.liv` file in Preview mode or on export of the SWF file, the script keyframe executes at a specific frame in the lifetime of the movie clip—that is, when the playhead reaches that script keyframe. A timeline can have multiple script keyframes.

All objects have a normal state by default. You also can add any of the *predefined* states (over, down, or out) to a movie clip in the States palette, or you can define custom states with their own names. Each movie clip state contains its own independent timeline, and each of these timelines can contain keyframes scripts.

When you write a script to a state, the script executes only when the movie clip enters that state, not at a preset point in the movie clip's lifetime. Say, for example, the user presses the mouse button on a movie clip for which you have defined a down state. This would execute any script you may have written for that state. You can write scripts to any or all states that you define for a movie clip. You also can write multiple scripts to the timeline of a single defined state by adding script keyframes.

## Accessing scripts

You can access scripts from:

- Script keyframes in a timeline. Clicking the script keyframe opens the Script Editor and displays the script added to that frame on the timeline.
- The Scripts button towards the bottom of the States palette. Clicking the scripts button opens the Script Editor on the state currently selected in the States palette.

In LiveMotion 1.0, the Scripts button was called the Behaviors button. For your general reference, the following four figures show you the LiveMotion 1.0 and LiveMotion 2.0 Timeline windows and States palettes.

Figure 6 shows the LiveMotion 1.0 Timeline window with a behavior added to a keyframe in a timeline.

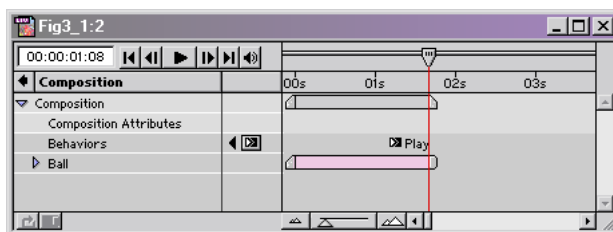


Figure 6 LiveMotion 1.0 Timeline window

Figure 7 shows the LiveMotion 2.0 Timeline window. In place of the Behaviors button, the Scripts button is used to create new scripts on timeline script keyframes. A separate Labels button is used to create labels on a timeline. The figure shows a label on a script keyframe.

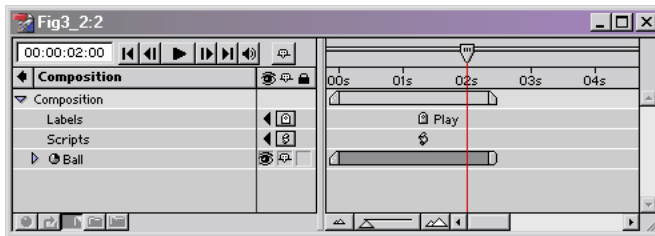


Figure 7 LiveMotion 2.0 Timeline window

Figure 8 shows the LiveMotion 1.0 Rollovers palette. The Behaviors button adds behaviors to object states and allows the user to access the behaviors. In the figure, the behaviors icon on the over state indicates that a behavior has been added to that state.

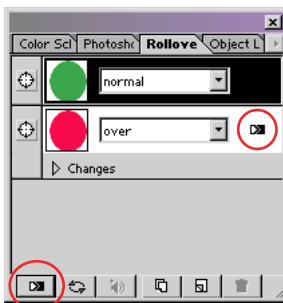


Figure 8 LiveMotion 1.0 Rollovers palette

Figure 9 shows the LiveMotion 2.0 States palette. This is very similar to the LiveMotion 1.0 Rollovers palette. However, you use a Scripts button to add new scripts to, and to access existing scripts on, object states. Like the LiveMotion 1.0 Rollovers palette, the script icon on the over state in the figure indicates that a custom script has been added to that state.

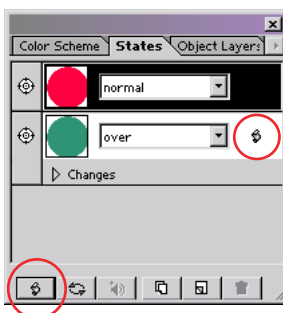


Figure 9 LiveMotion 2.0 States palette

**Advanced users:** You can access scripts by selecting Scripts > Script Editor from the main menu. Alternately, you can use the keyboard shortcut Ctrl + J (Windows) or Command + J (Mac OS). Then, select the movie clip whose script you want to access in the Script Editor's Movie clip navigator. This takes you to that movie clip's scripts, but not necessarily to the script that you want. You must then navigate to the event handler, state, or script keyframe containing the script you want to access.

## Adding Scripts

### To add a script to a movie clip state:

**Note:** The first three steps of this procedure also open a script on a state, as shown in the procedure in "To open a script from a movie clip state:" on page 30.

- 1 In the Timeline window, select the movie clip to which you want to add a state script.
- 2 Open the States palette to view that movie clip's states.
- 3 In the States palette, select the movie clip state to which you want to add a script.
- 4 Click on the Scripts button at the bottom of the States palette. See Figure 9.

This opens the Script Editor and displays the state's Script window.

- 5 Click the Scripting syntax helper button to open the list of LM 1.0 behaviors. Select the desired script by its LM 1.0 behavior name, and press Enter (or double click the name).

The script for the behavior is added to the Script window, as shown in Figure 10. For details on the Scripting syntax helper, see "Script Editor" on page 85.

- 6 Replace any parameters the script requires with their values.

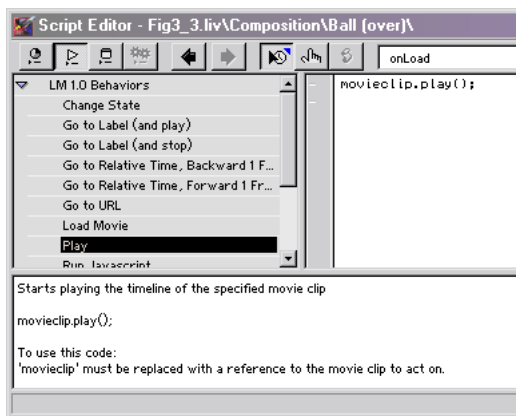


Figure 10 Scripting syntax helper open to LM 1.0 behaviors with the play behavior selected

### To add a script to a movie clip timeline:

- 1 Navigate to the timeline where you want to add the script keyframe.
- 2 In the Timeline window, move the current-time marker to the frame to which you want to add a script. Optionally, click the Labels button (see Figure 7), and enter a name for the point in time where the script will be added to the timeline.
- 3 Click the Scripts button on the timeline to create a script keyframe at the current-time marker.

This also opens the Script Editor.

**Note:** *If a script keyframe already exists on the specified frame, clicking the Scripts button simply opens the Script Editor and displays the scripts on that keyframe.*

4 Click the Scripting syntax helper button to open the list of LM 1.0 behaviors. See Figure 10. Select the desired behavior by its LM 1.0 name, and press Enter (or double click the name).

The script for the behavior is added to the Script window.

5 Replace any parameters the script requires with their values.

## Opening scripts

### To open a script from a movie clip state:

- 1 Open the States palette to view movie clip states.
- 2 In the States palette, select the movie clip state with the script you want to open.
- 3 Click the Scripts button in the palette.

This brings up the Script Editor and displays the script for that movie clip state in the Script window.

### To open a script from the timeline:

Locate the script icon for the script you want to view, and double-click.

## Deleting scripts

### To delete a script from a movie clip state:

- 1 Open the States palette to view movie clip states.
- 2 In the States palette, select the movie clip state with the script you want to delete.
- 3 Click the Scripts button in the palette.

This brings up the Script Editor and displays the script for that movie clip state in the Script window.

- 4 Select the script implementing the behavior you want to delete, and press Delete.

### To delete a script from the timeline:

- 1 Locate the script icon for the script you want to view, and double-click.
- 2 Select the script implementing the behavior you want to delete, and press Delete.

## Creating LiveMotion 1.0 behaviors using LiveMotion 2.0 scripts

This section provides details on how you create scripts that duplicate LiveMotion 1.0 behaviors. For your reference, Table 4 lists the LiveMotion 1.0 behaviors supported and the LiveMotion 2.0 scripts to which they map.

Table 4 LiveMotion 1.0 Behaviors and their corresponding scripts

| LM 1.0 Behavior                       | script                                                                                                                     | Description                                                                                                                           |
|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| Change State                          | <code>movieClip.lmSetCurrentState(<i>state</i>);</code>                                                                    | Change the state of the specified movie                                                                                               |
| Go to Relative Time, Backward 1 Frame | <code>movieClip.prevFrame();</code>                                                                                        | Go to the movie clip's relative time backward 1 frame                                                                                 |
| Go to Relative Time, Forward 1 Frame  | <code>movieClip.nextFrame();</code>                                                                                        | Go to the movie clip's relative time forward 1 frame                                                                                  |
| Go to URL                             | <code>getURL(<i>url</i>, <i>window</i>);</code>                                                                            | Open a URL in the specified browser window or frame                                                                                   |
| Go to Label (and stop)                | <code>movieClip.gotoAndStop(<i>label</i>);</code>                                                                          | Go to the specified label and stop                                                                                                    |
| Go to Label (and play)                | <code>movieClip.gotoAndPlay(<i>label</i>);</code>                                                                          | Go to the specified label and play                                                                                                    |
| Load Movie                            | <code>loadMovieNum(<i>url</i>, <i>levelNum</i>);</code>                                                                    | Load the specified URL into the specified SWF file level                                                                              |
| Run JavaScript                        | <code>getURL("javascript:code")</code>                                                                                     | Run the javascript specified                                                                                                          |
| Stop All Sounds                       | <code>stopAllSounds();</code>                                                                                              | Stop all sounds from playing, but do not stop the movie                                                                               |
| Unload Movie                          | <code>unloadMovieNum(<i>levelNum</i>);</code>                                                                              | Unload the specified movie                                                                                                            |
| Wait For Download                     | <pre> if (this._framesloaded &lt; lmFrameOfLabel(<i>finishLabel</i>)) {     this.gotoAndPlay(<i>repeatLabel</i>); } </pre> | Loop the composition timeline to a certain label until all the frames up to a specified label on the composition timeline have loaded |
| Play                                  | <code>movieClip.play();</code>                                                                                             | Start playing the specified movie                                                                                                     |
| Stop                                  | <code>movieClip.stop();</code>                                                                                             | Stop playing the specified movie                                                                                                      |

## Creating Change State scripts

The Change State script changes the state of the specified movie clip.

### To change the state of a movie clip:

- 1 Navigate to the location where you want to add the state change. See “Adding Scripts” on page 29.

**2** In the Script Editor, click the Scripting syntax helper button. Select Change State from the LM 1.0 behaviors list, and press Enter (or double click the name).

The appropriate script appears in the Script window:

```
movieClip.lmSetCurrentState(state);
```

**3** Replace the arguments described below with the appropriate values.

*movieClip* is a reference to the movie clip whose state you want to change.

*state* is a string containing the name of the state you want to set.

You can use the Script Editor's Scripting syntax helper (Description window), to obtain brief definitions of the script contents, and the Composition browser, to help fill in the values. For details on using the Script Editor features, see "Script Editor" on page 85.

### Hands-on example 3\_1: Changing movie clip states

In this exercise, you will create two movie clip (buttons) that control the state of a third movie clip.

#### To create this example:

- 1** Create a new composition. Save the file as `Ex3_1.liv`.
- 2** In the Composition window, create two ellipses. Give one a red fill color and the other, a blue fill.
- 3** Create a down state for each ellipse in the States palette.  
This converts each ellipse to a movie clip.
- 4** In the Composition window, create a rectangle. Give the rectangle a fill color, such as yellow (not red or blue).

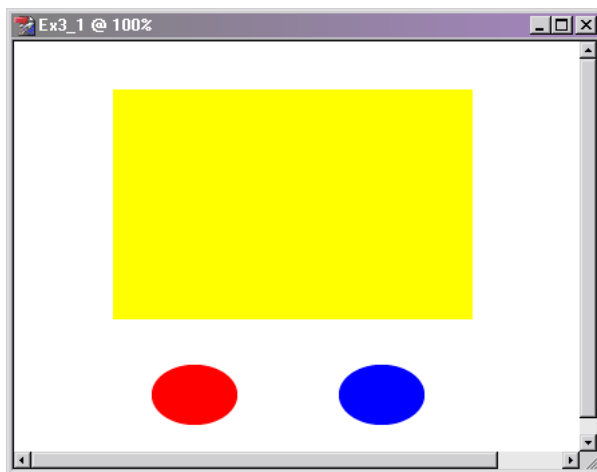


Figure 11 Composition window with two ellipses and a rectangle

- 5** In the States palette, give the rectangle two custom states: red and blue.  
This converts the rectangle into a movie clip.



- 6 For each of the custom states, give the rectangle the appropriate fill color: provide a blue fill for the blue state and red fill, for the red.
- 7 Open the Timeline window.
- 8 Select the rectangle, press Enter, and give it the new name Box. Press OK.
- 9 Select the red ellipse.
- 10 In the States palette, select the down state, and click the Scripts button to open the Script Editor.
- 11 Click the Scripting syntax helper button, and expand the list of LiveMotion 1.0 behaviors.
- 12 Select the Change State behavior, and press Enter (or double click the behavior name). The following script is generated in the Script window:

```
movieClip.lmSetCurrentState(stateName);
```

Replace *movieClip* with the absolute reference to Box, and replace *stateName* with the custom state "red". You can use the Composition browser in the Script Editor to help fill in the values for *movieClip*, and *stateName*. For details on using the Script Editor features, see "Script Editor" on page 85.

With these two parameters replaced, the script should appear as:

```
_root.Box.lmSetCurrentState("red");
```

- 13 Close the Script Editor.
- 14 In the Composition window, select the blue ellipse. Repeat steps 8 through 12, opening the script for the down state of the blue ellipse, but set the state of Box to blue instead of to red. With the parameters replaced, the script should appear as:

```
_root.Box.lmSetCurrentState("blue");
```

- 15 Preview.

Clicking the red ellipse changes the color of box to red. Clicking the blue ellipse changes the color of box to blue.

## Creating scripts to manipulate a movie clip timeline

These scripts can be used to manipulate a timeline:

- Play
- Stop
- Go To Relative Time, Backward 1 Frame
- Go To Relative Time, Forward 1 Frame
- Go To Label (and stop)
- Go to Label (and play)

The Play and Stop scripts play or stop a specified timeline. You can, for example, add scripts to the first frame of a composition timeline to stop the timelines of all the movie clips it contains. Although the movie clip timelines will be stopped, the composition timeline will continue playing, enabling you to run individual movie clips as needed using the script for Play.

In LiveMotion 2.0, the Go To Relative Time scripts only support going forward or backward one frame; whereas, the LiveMotion 1.0 behavior supported going forward or backward a specified number of frames. To achieve the same result as Go To Relative Time in LiveMotion 1.0, you can use the Go To Label script.

The Go to Label (and stop) script moves the animation to a specific label in a timeline and stops the timeline.

The Go to Label (and play) script sends the playhead of a movie clip's timeline to the specified frame or label to play the timeline at that frame.

### To add a Play or Stop script:

- 1 Navigate to the location where you want to add the script. See “Adding Scripts” on page 29.
- 2 In the Script Editor, click the Scripting syntax helper button. Select Stop or Play from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

The appropriate script appears in the Script window:

```
movieClip.stop();
```

or

```
movieClip.play();
```

- 3 Replace the *movieClip* argument described below with the appropriate value.

*movieClip* is a reference to the movie clip you want to start or stop at it's current frame. If the movie clip is stopping or playing itself, use `this` for the movie clip, for example, `this.stop()`;

or

```
this.play();
```

`play()` and `stop()` are movie clip methods that are equivalent in functionality to the respective LiveMotion 1.0 Play and Stop behaviors.

You can use the Script Editor's Scripting syntax helper (Description window), to obtain brief definitions of the script contents, and the Composition browser, to help fill in the values. For details on using the Script Editor features, see “Script Editor” on page 85.

### To add a Go to Relative Time script:

- 1 Navigate to the location where you want to add the script. See “Adding Scripts” on page 29.

2 Click the Scripting syntax helper button. Select Go to Relative Time, Backward 1 Frame or Go to Relative Time, Forward 1 Frame from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

The appropriate script appears in the Script window:

```
movieClip.prevFrame();
```

or

```
movieClip.nextFrame();
```

3 Replace the *movieClip* argument described below with the appropriate value.

*movieClip* is a reference to the movie clip you want to move backward or forward 1 frame.

You can use the Script Editor's Scripting syntax helper (Description window), to obtain brief definitions of the script contents, and the Composition browser, to help fill in the values. For details on using the Script Editor features, see "Script Editor" on page 85.

### To add a Go to Label (and stop) script:

1 Navigate to the location where you want to add the script. See "Adding Scripts" on page 29.

2 Click the Scripting syntax helper button. Select Go to Label (and stop) from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

The script appears in the Script window:

```
movieClip.gotoAndStop(label);
```

Replace the *movieClip* and *label* arguments described below with the appropriate values. You can use the Scripting syntax helper and the Composition browser in the Script Editor to help fill in these values. For details on using the Script Editor features, see "Script Editor" on page 85.

*movieClip* is a string containing the label name associated with the frame on the movie clip's timeline to which the playhead will be sent and stopped.

*label* is a string associated with the frame on the movie clip's timeline to which the playhead will be sent and stopped.

Here is an example script with the values filled in:

```
_root.gotoAndStop("end");
```

**Note:** When you create the label on a timeline frame, do not enclose the label name in quotation marks. However, when you fill in the value for label (which is of type string) in the script, you must enclose the label name in quotation marks, as shown in this example script.

### To add a Go to Label (and play) script:

1 Navigate to the location where you want to add the script. See "Adding Scripts" on page 29.

2 Click the Scripting syntax helper button. Select Go to Label (and play) from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

The script appears in the Script window:

```
movieClip.gotoAndPlay(label)
```

**3** Replace the `movieClip` and `label` arguments described below with the appropriate values.

`movieClip` is the name of the movie clip that you want to go to `label` and play.

`label` is a string containing the label name associated with the frame on the movie clip's timeline to which the playhead will be sent and played.

You can use the Script Editor's Scripting syntax helper (Description window), to obtain brief definitions of the script contents, and the Composition browser, to help fill in the values. For details on using the Script Editor features, see "Script Editor" on page 85.

## Creating Wait For Download scripts

The Wait For Download script is a special case of timeline manipulation. It is used to loop part of the composition timeline until all the items placed on the timeline up to a specified frame have been downloaded. A Wait For Download script can be used to prevent poor performance for compositions that include large objects, or for lengthy and complex movie clips.

The script only works in a script keyframe on the composition timeline and is useful only in compositions that are later loaded with the `loadMovie()` movie clip method or global function. The first SWF file in the Flash Player is always downloaded completely before playback begins.

Wait For Download consists of three items on the main timeline: two labels with a script in between. These items work together to loop the timeline until all the content up to a certain frame has been downloaded.

The first label on the timeline identifies the first timeline frame that is part of the waiting loop. The second label, and last item on the timeline, identifies the timeline frame that is being waited upon to finish downloading. Situated in between the labels on the timeline is a script keyframe to which the Wait For Download script is added. The script keyframe marks the last frame of the waiting loop. Upon execution, the script tests to see if the frame on the timeline containing the second label has loaded. If it has, the composition timeline plays forward; otherwise, the playhead of the composition timeline is placed back at the location of the first label where it repeats playing the frames between the first label and the script keyframe.

This looping pattern continues until all the content on the composition timeline—up to and including the location of the second label—has been loaded. All the objects to download must be placed on the timeline after the script keyframe containing the Wait For Download code and before the second label.

### To add a Wait For Download script:

- 1** Move the current-time marker to the location on the composition timeline where you want your waiting loop to begin. Create the first label here.
- 2** Move the current-time marker to the location on the composition timeline after all the large objects that you want to wait to download have appeared on the timeline. Create the second label here.
- 3** Move the current-time marker to a location between the two labels where you want your waiting loop to end.

This point must be before the large objects waiting to be downloaded have appeared on the timeline.

**4** Create a script keyframe here. This also opens the Script Editor.

**5** In the Script Editor, click the Scripting syntax helper button.

**6** Select Wait For Download from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

This script appears in the Script window:

```
if (this._framesloaded < lmFrameOfLabel(finishLabel))
{
 this.gotoAndPlay(repeatLabel);
}
```

`lmFrameOfLabel()` is a global function that converts a label on the composition timeline into the corresponding frame number on export.

**7** Replace the *finishLabel* and *repeatLabel* arguments described below with the appropriate values.

*repeatLabel* is a string containing the name of the first label, created in step 1.

*finishLabel* is a string containing the name of the second label, created in step 2.

You can use the Composition browser in the Script Editor to help fill in the values for *repeatLabel* and *finishLabel*. For details on using the Script Editor features, see "Script Editor" on page 85.

## Hands-on example 3\_2: Creating a preloader

This example creates a preloader that loops a piece of the main timeline until sufficient frames (containing large items) of the main timeline have loaded.

A preloader of this style consists of three parts:

- Two labels and a keyframe script that implement Wait for Download
- The large item to download
- The content that to be displayed during the waiting loop

This example uses an image from the Library palette as the large item to be downloaded and a text object that reads "Loading" as the content displayed during the waiting loop. However, you can do whatever you want during the wait for download "pause." For example, you could create a small animation to entertain the viewer or a status bar showing the progress of the download.

### To create the Wait for Download:

- 1** Create a new composition in LiveMotion, and save it as `Ex3_2.liv`.
- 2** Open the Timeline window
- 3** Move the current-time marker to frame 0 on the composition timeline, and create a label. Name it "loading."
- 4** Drag the endpoint of the composition timeline to frame 10.
- 5** Move the current-time marker to frame 10, create a label, and name it "end."

**6** Move the current-time marker to frame 5 on the composition timeline, and click the Scripts button to create a script keyframe.

**7** In the Script Editor, click the Scripting syntax helper button, and expand the LM 1.0 behaviors list.

**8** Double click the behavior, Wait for Download. The script for this behavior appears in the Script window:

```
if (this._framesloaded < lmFrameOfLabel(finishLabel))
{
 this.gotoAndPlay(repeatLabel);
}
```

8) Replace *finishLabel* with the string "end" as shown in Figure 12.

9) Replace *repeatLabel* with the string "loading" as shown in Figure 12.

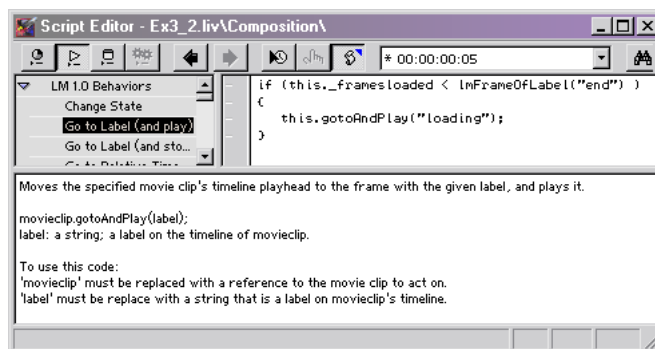


Figure 12 Script with label strings filled in

### To place the rocket image:

- 1** Move the current-time marker to frame 6.
- 2** Open the Library palette, select the rocket image, and place it in the Composition window.
- 3** Adjust the duration bar of the rocket image so that it starts at frame 6 and ends at frame 10.

### To create the waiting content:

- 1** Move the current-time marker to frame 0.
- 2** Choose the text field tool from the Tools palette, and create a rectangle in the Composition window.
- 3** Enter "Loading..." as the text.

- 4 Adjust the duration bar of the text object so that it starts at frame 0 and ends at frame 5.

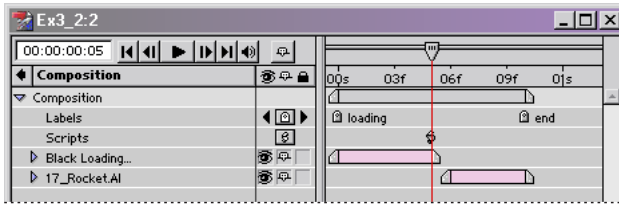


Figure 13 Timeline window showing text and rocket image duration bars

- 5 Preview.

## Creating scripts to command the Flash Player

Three scripts create commands to the Flash Player. These are:

- Load Movie
- Unload Movie
- Stop All Sounds

Load Movie loads and plays a SWF file that can either replace the existing SWF file, or play in another level of the Flash Player. Unload Movie removes an already-loaded SWF file from the player. Stop All Sounds stops all sounds in the player, including event sounds.

### To load a SWF file:

- 1 Navigate to the location where you want to add the script to load a SWF file. See "Adding Scripts" on page 29.
- 2 Click the Scripting syntax helper button. Select Load Movie from the LM 1.0 Behaviors list, and press Enter (or double click the behavior name).

The behavior script appears in the Script window:

```
loadMovieNum(url, number);
```

- 3 Replace the arguments described below with the appropriate values.

*url* is a string containing an absolute or relative reference to the external SWF file.

These are examples:

```
"http://www.mydomain.com/loadedMovie.swf"
```

or

```
"loadedMovie.swf"
```

*number* is a non-negative integer specifying the player level into which the SWF file will be loaded. Your default composition is considered to be level number 0. If the level already contains a SWF file, it is replaced by the one being loaded. For details on player levels, see "Levels of the Flash Player" on page 62.

### To unload a SWF file:

- 1 Navigate to the location where you want to add the script to unload a SWF file. See "Adding Scripts" on page 29.
- 2 Click the Scripting syntax helper button. Select Unload Movie from the LM 1.0 Behaviors list, and press Enter (or double click the behavior name).

The behavior script appears in the Script window:

```
unloadMovieNum(number);
```

**3** Replace the argument described below with the appropriate value.

*number* is a non-negative integer specifying the document level of the SWF file to be unloaded. For details on document levels, see “Levels of the Flash Player” on page 62.

### To stop all sounds:

**1** Navigate to the location where you want to add the script to stop all sounds. See “Adding Scripts” on page 29.

**2** Click the Scripting syntax helper button. Select Stop All Sounds from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

The script appears in the Script window:

```
stopAllSounds();
```

## Creating scripts to control the Web browser

There are two browser command scripts. These are:

- Run JavaScript
- Go to URL

Run JavaScript executes JavaScript code in the user's browser. The Go to URL script opens a specified URL in the user's browser and loads it into the browser at the specified target.

### To run JavaScript:

**1** Navigate to the location where you want to add the script to execute JavaScript. See “Adding Scripts” on page 29.

**2** Click the Scripting syntax helper button. Select Run JavaScript from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

The script appears in the Script window:

```
getURL("javascript:code");
```

**3** Replace the *code* argument with your code, as illustrated by the example below:

```
getURL("javascript: window.alert('hello world');");
```

This code displays the string ‘hello world’ in the browser window.

### To add a Go to URL script:

**1** Navigate to the location where you want to add the Go to URL script. See “Adding Scripts” on page 29.

**2** Click the Scripting syntax helper button. Select Go to URL from the LM 1.0 behaviors list, and press Enter (or double click the behavior name).

The script appears in the Script window:

```
getURL(url, window);
```

**3** Replace the *url* and *window* arguments described below with the appropriate values.

*url* is a string containing the URL you want to load.



*window* is a string specifying the browser location to load the URL into—either a custom frame name or one of the four standard values: `_blank`, `_parent`, `_self`, or `_top`.

Here is an example:

```
getURL("http://www.adobe.com", "_blank");
```

# Movie Clips

## Introduction to movie clips

A movie clip is a LiveMotion object that you can manipulate programmatically through scripting. Movie clips are JavaScript objects. Like other JavaScript objects, movie clips have properties and methods, and they can be assigned to variables and placed in arrays.

**Note:** *The composition is a movie clip that you reference as `_root`. Composition and `_root` are synonymous. `_root` is a special movie clip in that you do not create or name it. It is there by default when you create a composition, and it functions like other movie clips with the exception of a few built-in properties and methods, which do not apply. For details, see “Movie clip properties and methods” on page 48.*

Movie clips have a set of built-in properties and methods that are defined by the Flash Player. A movie clip's built-in properties describe the physical features of a movie clip, for example its height, width, position, and the number of frames on its timeline. You can set the values of these built-in properties to programmatically control the appearance and behavior of a movie clip throughout its lifetime. A movie clip's built-in methods include functionality that you can perform on movie clips such as creating copies, loading and unloading movie clips, and playing and stopping movie clips. In addition, you can use built-in methods to obtain information about a movie clip such as its size, the number of bytes loaded, and whether it intersects with other movie clips at specified points. You can also define your own methods and properties for movie clips, as described in “Creating movie clip properties and methods” on page 54.

In addition to having the characteristics of standard JavaScript objects, movie clips have the ability to handle user- and system-generated events such as pressing a key or loading a movie clip. For a movie clip to respond to an event, you must write an *event handler* for the event on that movie clip. The handler then executes whenever the event occurs. For details on movie clip event handling, see “Movie Clip Events and Event Handlers” on page 64.

Unlike other JavaScript objects, movie clip objects cannot be instantiated: that is, you cannot create a new, original movie clip programmatically. A movie clip has no constructor, and cannot be instantiated using the `new` operator.

So, you might ask, how do I create a movie clip instance? The simplest method, and the one to work with first, is to create the movie clip manually in the Composition window. Later, this section describes two other methods that programmatically create copies of existing movie clips.



## How to create a movie clip using LiveMotion

LiveMotion objects start out as “regular” (unscriptable) objects. To write scripts to an object, you must convert the object into a movie clip or a movie clip group. The exception is objects for which you have defined additional states (besides the normal state, which all objects have by default). In such a case, LiveMotion automatically converts the object into a movie clip. As an indication that an object or a group of objects has been turned into a movie clip, the movie clip icon is displayed to the left of the movie clip or the movie clip group name on the timeline. Conversion gives the movie clip its own timeline so that it can play independently of the main composition timeline and independently of any parent timeline, if the movie clip is nested. Movie clips are equivalent to the time-independent objects and time-independent groups in LiveMotion 1.0.

### Basic methods

You can manually create movie clips in two basic ways: by converting an object to a movie clip and by creating movie clip groups. Movie clip groups differ from movie clip objects in that a movie clip group contains one or more child objects (movie clips or regular objects). A movie clip in itself is not a group and, as such, cannot contain a child object.

#### To convert an object to a movie clip:

Select one object in the timeline, and click the “Make selected objects movie clips” button at the bottom of the Timeline window, or choose Object > Movie Clip from LiveMotion’s main menu.

#### To create a movie clip group:

Select one or more objects in the timeline, and click the “Group objects and make movie clip” button at the bottom of the Timeline window, or choose Object > Make Movie Clip Group from LiveMotion’s main menu. Make Movie Clip Group first groups the selected objects. Then it turns the group into a movie clip with its own independent timeline. Movie clip groups can contain regular (unscriptable) objects, and movie clips, as well as other movie clip groups.

You also can create a movie clip group using this two-step approach:

- 1 Select one or more objects, and choose Select Object > Group from the main menu. Alternately, you can press Ctrl+ G (Windows) or Command+G (Mac OS).
- 2 Click the “Make selected objects movie clips” button at the bottom of the timeline, or choose Object > Make Movie Clip from LiveMotion’s main menu.

#### Effect of creating a movie clip and a movie clip group

When you create a movie clip group, you add an extra timeline between the objects in the movie clip group and the main composition timeline. This is the timeline of the movie clip group object. Figure 14 compares what happens before and after making a movie clip to what happens before and after making a movie clip group.

Immediately after creating a movie clip group, the movie clip group name is displayed in the Timeline window. To view the group's contents, you must expand the movie clip group's timeline.

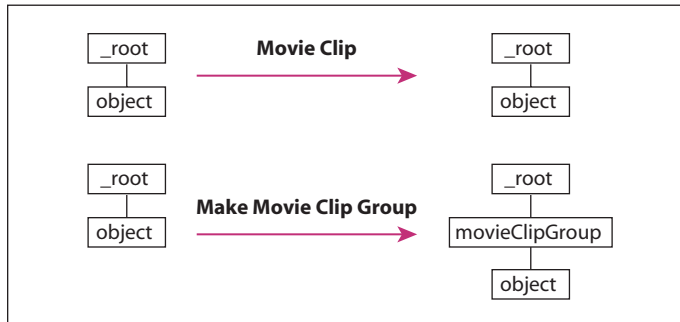


Figure 14 Before and after creating a movie clip and creating a movie clip group

## Movie clip hierarchy

All movie clips are arranged in a hierarchy. At the top of the hierarchy is the composition (also referred to as the `_root` movie clip or, simply, `_root`).

In Figure 15, `movieClipGroupA` is a child of `_root`. `_root` also has a second child, `movieClipE`. Because `movieClipGroupA` and `movieClipE` share the same *parent*, they are referred to as *siblings*. `movieClipB` and `movieClipC` are children of `movieClipGroupA`.

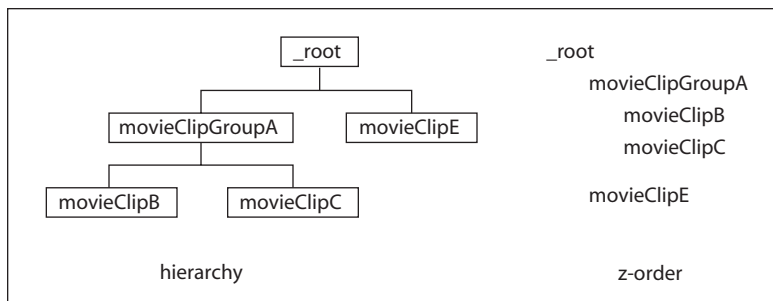


Figure 15 Movie clip hierarchy and z-order

In LiveMotion, you create a parent-child relationship any time you place (or create) a movie clip or movie clip group on the timeline of another movie clip group or `_root`. The movie clip group becomes the parent of the movie clips it contains. For details on creating movie clip groups, see "How to create a movie clip using LiveMotion" on page 43.

### Relationship of movie clip hierarchy to z-order

In the movie clip hierarchy shown in Figure 15, a parent appears above its children. This hierarchy fails to demonstrate the z-order that you see reflected in the Timeline window, however. (Recall that *z-order* is the order in which objects overlap. For details, see the *LiveMotion 2.0 User Guide*.) To see the z-order of a movie clip group's children, you open the group's timeline.

Ignoring programmatically generated movie clips for the moment, the visual result in the Composition window of the Timeline z-order window is determined by the order of the movie clip groups *and* the order of the movie clips within them. This is still true when programmatically generated movie clips are added to a composition, as described in “What the programmatic stack does to the movie clip hierarchy” on page 58. The order just takes on some more detail.

If, for example, you were to open the Timeline window for the composition shown in Figure 15, z-order would show the composition timeline at the top and `movieClipGroupA`, above `movieClipE`. But because `movieClipGroupA` is just a movie clip group containing `movieClipB` and `movieClipC`, the movie clips would appear from front to back in this order in the Composition window: `movieClipB`, `movieClipC`, `movieClipE`.

**Note:** To be able to refer to child movie clips in scripting, siblings must have unique names. Otherwise, you will only be able to access the redundant child name that is topmost in z-order.

## How to access movie clips in the hierarchy

In scripting language, children are accessed as properties of their parent using dot (.) notation. For example, `movieClipGroupA` can access its child `movieClipB` as:

```
_this._movieClipB
```

A child can access its parent using the movie clip `_parent` property. For example, this is how `movieClipGroupA` can access `_root`:

```
this._parent
```

The keyword `this` refers to the movie clip to which a script is added. The above script is interpreted to mean: “From this movie clip’s position in the object hierarchy, go up one level in the hierarchy to access the parent of `this`, which happens to be `_root`.”

In Figure 15 `movieClipB` is a grandchild of `_root`. Here is how `_root` is accessed from `movieClipB` using the `_parent` property:

```
this._parent._parent
```

## Movie clip addressing

You most likely will be changing the object hierarchy as you develop your composition. It is important that you understand movie clip addressing, so you can make the appropriate changes to movie clip references in your scripts as a result of object hierarchy changes. This section describes movie clip addressing and makes suggestions on addressing choices, depending on your situation.

There are two types of movie clip addresses:

- Absolute reference
- Relative reference

This section uses the movie clip object hierarchy shown in Figure 16 to illustrate the addressing types.

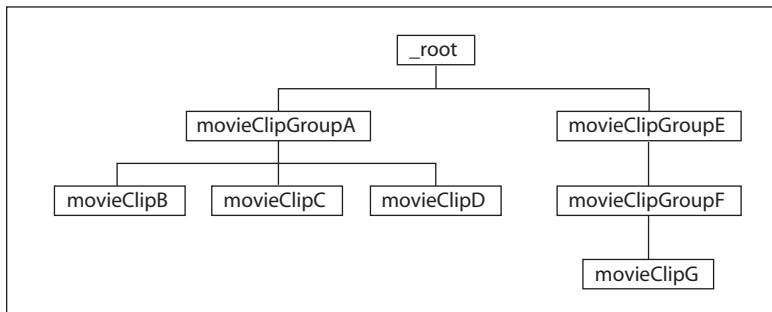


Figure 16 Movie clip addressing

### What is an absolute reference?

An absolute reference is a reference to a movie clip that begins at the top of the composition, and walks down through the object hierarchy—parent to child—until reaching the movie clip of interest. An absolute reference always starts with `_root`, and uses dot (`.`) notation to access the children of `_root`, and the children's children, and so on until it obtains the movie clip being referenced. The absolute reference is the same regardless of where in the movie clip hierarchy the *source* movie clip (movie clip making the reference) is located.

### Absolute reference example

For example, the absolute reference to `movieClipB` is:

```
_root.movieClipGroupA.movieClipB
```

`_root` is always at the top of the hierarchy and starts in the absolute reference. In this example, `movieClipGroupA` is at the level just above `movieClipB`. The reference ends with `movieClipB`, the movie clip being referenced.

### What is a relative reference?

A relative reference is a reference that begins with the source movie clip (movie clip making the reference) and walks through the movie clip hierarchy, each step being parent-to-child or child-to-parent until it reaches the movie clip of interest. Relative references always start with `this`, and access the next movie clip in the reference—either as a child, or through the `_parent` property—until it obtains a reference to the desired movie clip. A relative reference is dependent on the relationship between the source movie clip and the movie clip it is referencing and varies from source to source.

**Note:** Although using `'this'` is optional in the relative reference, this scripting guide begins all relative references with `'this'` so you can more easily distinguish between absolute references and relative references.

### Relative reference examples

Here is an example of the relative reference from `movieClipGroupA` (shown in Figure 16) to `movieClipGroupE`:

```
this._parent.movieClipGroupE
```

this refers to `movieClipGroupA`.

`_parent` is `movieClipGroupA`'s parent (in this case, `_root`) which is up one level in the object hierarchy from `movieClipGroupA`. From `_root` the reference leads down one level to `movieClipGroupE`.

This is the relative reference from `movieClipC` to `_root`:

```
this._parent._parent
```

In this example, `_root` is `movieClipC`'s grandparent.

## When to use an absolute or a relative reference

You can access all the movie clips in a composition using either type of reference for movie clip addressing. However, in most cases one reference style makes more sense than the other.

Here are two general rules:

- Choose the reference style that you believe is least likely to change during your editing process.
- The simpler reference is usually the better one.

If, for example, you know that the location of the movie clip that you want to access is not going to change in the object hierarchy, but you are not sure where the source movie clip that is accessing it is going to be, it is probably better to use an absolute reference. Then, regardless of where the source movie clip is located in the hierarchy, the reference to the target will be correct. If you know that the relationship between two movie clips in the hierarchy is not going to change, but you are not sure where these movie clips will be located relative to `_root`, it is probably better to use a relative reference. If you're still uncertain about the relationship of the movie clips, choose the simpler reference. For example, it makes more sense for `movieClipG` to refer to `movieClipF` as `this._parent` than as `_root.movieClipGroupE.movieClipF`.

## More examples of movie clip addressing

This section provides additional examples of movie clip addresses. It identifies all the references from the objects in Figure 17 to `movieClipD`.

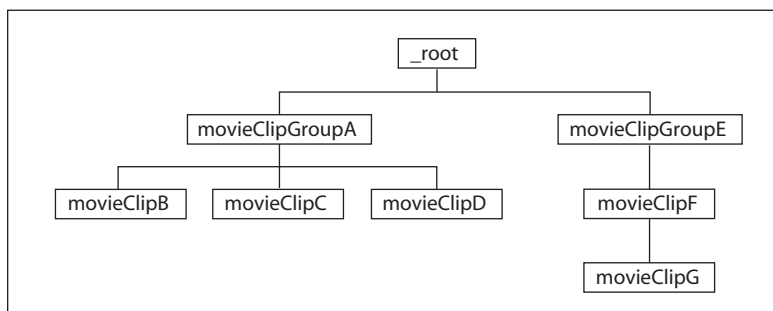


Figure 17 Object hierarchy for examples

There is only one absolute reference to `movieClipD`:

```
_root.movieClipGroupA.movieClipD
```

Table 5 shows all the relative references to `movieClipD` from each of the other movie clips in Figure 17.

Table 5 Relative references to `movieClipD`

| Source                       | Relative reference to <code>movieClipD</code>                        |
|------------------------------|----------------------------------------------------------------------|
| <code>movieClipGroupA</code> | <code>this.movieClipD</code>                                         |
| <code>movieClipB</code>      | <code>this._parent.movieClipD</code>                                 |
| <code>movieClipC</code>      | <code>this._parent.movieClipD</code>                                 |
| <code>movieClipD</code>      | <code>this</code>                                                    |
| <code>movieClipGroupE</code> | <code>this._parent.movieClipGroupA.movieClipD</code>                 |
| <code>movieClipF</code>      | <code>this._parent._parent.movieClipGroupA.movieClipD</code>         |
| <code>movieClipG</code>      | <code>this._parent._parent._parent.movieClipGroupA.movieClipD</code> |

## Movie clip properties and methods

### Built-in movie clip properties

As illustrated in the previous example, you can manipulate a movie clip's properties to create effects such as animation. Movie clips come with a large number of built-in properties. You can use these properties to modify the physical features of a movie clip, such as changing its size or opacity or changing its location.

Table 6 lists all the built-in movie clip properties. The built-in property names start with the underscore (`_`) character to distinguish them from properties that you might define yourself.

**Note:** The `_root` movie clip works with all of these properties except `_name` and `_parent`.

Table 6 Built-in movie clip properties

| Property                   | Description                                                                                                                         |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <code>_alpha</code>        | Opacity of the movie clip on a scale of 0 (transparent) to 100 (opaque).                                                            |
| <code>_currentframe</code> | Position of the playhead in the movie clip's timeline.                                                                              |
| <code>_droptarget</code>   | Absolute reference (in slash notation) of a movie clip over which a movie clip passes during drag operations performed by the user. |
| <code>_framesloaded</code> | Number of the movie clip frames that have been loaded.                                                                              |
| <code>_height</code>       | Height of the movie clip in pixels.                                                                                                 |
| <code>_name</code>         | Name of the movie clip. This property does not work with <code>_root</code> .                                                       |
| <code>_parent</code>       | Movie clip containing this movie clip. This property does not work with <code>_root</code> .                                        |
| <code>_rotation</code>     | Rotation angle of the movie clip in degrees.                                                                                        |



| Property                  | Description                                                                                              |
|---------------------------|----------------------------------------------------------------------------------------------------------|
| <code>_target</code>      | Absolute reference of the movie clip in slash notation.                                                  |
| <code>_totalframes</code> | Number of frames in the movie clip.                                                                      |
| <code>_url</code>         | URL of the SWF file that this movie clip is a part of.                                                   |
| <code>_visible</code>     | Boolean indicating whether the movie clip is visible.                                                    |
| <code>_width</code>       | Width of the movie clip in pixels.                                                                       |
| <code>_x</code>           | Horizontal location of the movie clip in pixels relative to the anchor point of the movie clip's parent. |
| <code>_xmouse</code>      | Horizontal location of mouse pointer in pixels relative to the anchor point of the movie clip.           |
| <code>_xscale</code>      | Horizontal percentage scale factor of the movie clip (100% is full size).                                |
| <code>_y</code>           | Vertical location of the movie clip in pixels relative to the anchor point of the movie clip's parent.   |
| <code>_ymouse</code>      | Vertical location of mouse pointer in pixels relative to the anchor point of the movie clip.             |
| <code>_yscale</code>      | The vertical percentage scale factor of the movie clip (100% is full size).                              |

## Built-in movie clip methods

Movie clip methods are functions attached to the movie clip object and are called using `()`. Scripting provides a set of built-in movie clip methods that you can use to control a movie clip in various ways. Included are methods with which you can affect the behavior of a movie clip, change or find out about a movie clip's characteristics, load additional SWF files, and programmatically create duplicates of a movie clip. (Programmatically creating movie clips is described at length in "Creating movie clips programmatically" on page 55.) Table 7 lists the built-in movie clip methods and describes their functions. See "Reference" on page 105 for details on the arguments to each of these methods.

**Note:** The `_root` movie clip works with all of these methods except `duplicateMovieClip()`, `removeMovieClip()`, and `swapDepths()`.

Table 7 Built-in movie clip methods

| Method                            | Description                                                                                                                                                                           |
|-----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>attachMovie()</code>        | Attach the named movie clip (passed in as an argument) to the movie clip. For details see "Static and programmatic stacks" on page 56.                                                |
| <code>duplicateMovieClip()</code> | Duplicate this movie clip. For details see "Movie clip global functions that use <code>_levelN</code> as an argument" on page 63. This method does not work with <code>_root</code> . |
| <code>getBounds()</code>          | Return bounds of the movie clip. The returned object contains the values in the properties <code>xMin</code> , <code>xMax</code> , <code>yMin</code> , and <code>yMax</code> .        |

| Method                           | Description                                                                                                                                                                                                                                                   |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getBytesLoaded()</code>    | Return the number of bytes already loaded if the movie clip is external (loaded with <code>MovieClip.loadMovie()</code> ). If the movie clip is internal, the number returned is always the same as that returned by <code>MovieClip.getBytesTotal()</code> . |
| <code>getBytesTotal()</code>     | Return the size of the movie clip in bytes. When running under the preview tool in LiveMotion, this number is always 1000.                                                                                                                                    |
| <code>getUrl()</code>            | Load the URL into the browser.                                                                                                                                                                                                                                |
| <code>globalToLocal()</code>     | Convert the given global point to the movie clip's coordinate space.                                                                                                                                                                                          |
| <code>gotoAndPlay()</code>       | Go to the specified label and play. Also a global movie clip method.                                                                                                                                                                                          |
| <code>gotoAndStop()</code>       | Go to the specified label and stop. Also a global movie clip method.                                                                                                                                                                                          |
| <code>hitTest()</code>           | Return a Boolean value indicating whether the movie clip intersects with a given clip (passed in as an argument) or given <i>x,y</i> coordinates.                                                                                                             |
| <code>lmSetCurrentState()</code> | Change the state of the movie. The LiveMotion state of the movie must already be defined and appear in the state browser.                                                                                                                                     |
| <code>loadMovie()</code>         | Load an external SWF file into the movie clip. The contents of the movie clip are replaced with the contents of the SWF file.                                                                                                                                 |
| <code>loadVariables()</code>     | Load variables into the movie clip fetched from the specified URL. The movie clip's <code>onData</code> handler is called when the variables have been loaded.                                                                                                |
| <code>localToGlobal()</code>     | Convert a point in the movie's coordinate space to global coordinates.                                                                                                                                                                                        |
| <code>nextFrame()</code>         | Go to the next frame and stop playing. Also a global movie clip method.                                                                                                                                                                                       |
| <code>play()</code>              | Start playing.                                                                                                                                                                                                                                                |
| <code>prevFrame()</code>         | Go to the previous frame and stop playing.                                                                                                                                                                                                                    |
| <code>removeMovieClip()</code>   | Delete a duplicated or attached instance. This method does not work with <code>_root</code> .                                                                                                                                                                 |
| <code>startDrag()</code>         | Start dragging a movie clip. Also a global movie clip method.                                                                                                                                                                                                 |
| <code>stop()</code>              | Stop playing.                                                                                                                                                                                                                                                 |
| <code>stopDrag()</code>          | Stop any drag operation in progress.                                                                                                                                                                                                                          |
| <code>swapDepths()</code>        | Swap the movie clips's depth with that of another movie clip. For details on depth, see "Movie clip global functions that use <code>_leveln</code> as an argument" on page 63. This method does not work with <code>_root</code> .                            |
| <code>unloadMovie()</code>       | Unload a movie that was previously loaded with <code>loadMovie()</code> .                                                                                                                                                                                     |
| <code>valueOf()</code>           | Returns the absolute reference to the movie in absolute terms using dot (as opposed to slash) notation.                                                                                                                                                       |

## Hands-on example 4\_1: Mouse trailer

This example creates a mouse trailer. It uses the following movie clip properties and methods:

```
_x
_y
_xmouse
_ymouse
_xscale
_yscale
duplicateMovieClip()
gotoAndPlay()
```

The `_xmouse` and `_ymouse` movie clip properties establish the position of the mouse relative to the movie clip position. Each mouse movement causes the manually created movie clip and several programmatically generated and scaled duplicates to follow the mouse. The `_xscale` and `_yscale` movie clip properties progressively scale the duplicates from smallest to largest as they are generated in the Composition window.

### To create a mouse trailer:

**1** Create a new composition. Save the file as `Ex4_1.liv`.

**2** Create an object in the Composition window.

The object will be the base of your mouse trailer. The size of this object will be the size of the largest object in your trailer. After completing the code for this example, you can go back later and edit the object to change the appearance of your mouse trailer.

**3** Select the object in the Timeline window, convert it into a movie clip, and name it `Base0`.

**4** Select `Base0`, and make it a movie clip group by choosing `Object > Make Movie Clip Group` from the main menu.

With `Base0` inside of a movie clip group, the timeline object hierarchy is:

```
_root
 (Movie clip group) Group of 1 objects
 (Movie clip) Base0
```

**5** Select the newly created Group of 1 objects, and name it `MouseTrailer`. The timeline object hierarchy changes to:

```
_root
 (Movie clip group) MouseTrailer
 (Movie clip) Base0
```

**6** Expand `MouseTrailer`'s timeline. Drag the end marker of `MouseTrailer`'s duration bar to frame 2. Be sure that the endpoint of `Base0`'s duration bar also is at frame 2.

Both duration bars should be three frames long, as shown in Figure 18.

**7** Place the current-time marker at frame 0.

**8** Click the Scripts button to create a script keyframe at frame 0. This also opens the Script Editor. In the Script window, enter the code:

```
this.trailers = new Array(); //an array of objects that trail the mouse
```

```
//create 9 more objects for the trailer
```

```
var i;
```

```
for (i = 1; i < 10 ; i++)
```

```
{
```

```
 // create the new object, give it a unique name, and
```

```
 // place it at a unique depth
```

```
 this.Base0.duplicateMovieClip("Base" + i, i);
```

```
 // put the new object in the array
```

```
 this.trailers[i] = this["Base" + i];
```

```
 // change the scale of the new object
```

```
 this.trailers[i]._xscale = 100 - i*10;
```

```
 this.trailers[i]._yscale = 100 - i*10;
```

```
}
```

```
// put the original in the array
```

```
this.trailers[0] = this.Base0;
```

This code sets up the mouse trailer. It creates a series of duplicates of Base0, places each duplicate in the array, and scales the objects such that the topmost is the smallest, and the bottommost is the largest.

**9** Close the Script Editor window.

**10** In the Timeline window, move the current-time marker to frame 1, and create a label. Name the label "repeat."

This example uses labels so that, if you change the frame rate of the composition, the mouse trailer still works.

**11** At frame 1, create a script keyframe. In the Script Editor, enter the code:

```

/* update the position of the trailers
place the topmost trailer at the position of the mouse
*/

this.trailers[9]._x = this._xmouse;
this.trailers[9]._y = this._ymouse;

/*
update the position of the rest of the objects, placing the object
halfway between its current position and the position of the object
in front of it.
*/

var i = 0;
for(i = 0; i < 9 ; i++)
{
 this.trailers[i]._x += (this.trailers[i+1]._x -
this.trailers[i]._x)/2;
 this.trailers[i]._y += (this.trailers[i+1]._y -
this.trailers[i]._y)/2;
}

```

Each time this code is called, it updates the position of Base0 and each of the duplicates of Base0.

**12** Close the Script Editor window.

**13** Move the current-time marker to frame 2, and click the Scripts button to create a script keyframe.

This also opens the Script Editor. Figure 18 shows how the MouseTrailer timeline should appear at this point.

**14** In the Script window, enter the code:

```
this.gotoAndPlay("repeat");
```

Each time this code is called, it resets the current-time marker to the frame labeled “repeat,” which is where the code to update the positions is located.

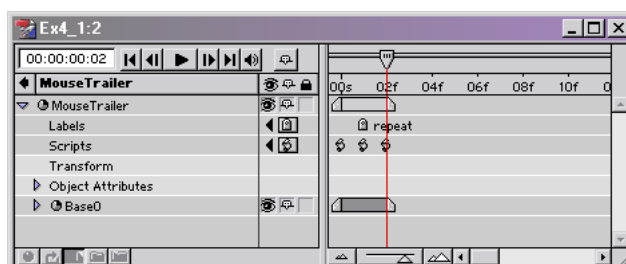


Figure 18 Mouse trailer timeline with script keyframes and repeat label

**15** Preview.

**16** Export this file, and save it as `Ex4_1.swf`.

## Creating movie clip properties and methods

You can create your own movie clip properties and methods. To do so, navigate to the timeline of the movie clip for which you want to create a property or method, open the Script Editor. You can enter the code for the definitions in the movie clip's `onLoad` handler.

This example creates the movie clip property `toggle`, which returns a boolean value. The example uses `toggle` in the `blink()` method to change the movie clip's opacity:

```
// define the toggle property

this.toggle = true;

// define the blink method

this.blink = function()
{
 if(this.toggle == true)
 {
 this._alpha = 50; // change opacity value to 50
 this.toggle = false;
 }
 else
 {
 this._alpha = 100; // change opacity value to 100
 this.toggle = true;
 }
}
```

You can call the methods that you created in the same way that you call a method on any object. Provide the name of the movie clip and the method name. The call to `blink()` appears as:

```
this.blink(); // calling the blink method
```

## Creating movie clips programmatically

You can create movie clips manually or programmatically. As previously described, you can manually create movie clips or movie clip groups by creating regular objects using LiveMotion's tools in the Composition window and then converting those objects to movie clips or to movie clip groups. Besides creating a movie clip manually in the Composition window, you can create a movie clip programmatically using the built-in movie clip methods: `attachMovie()` and `duplicateMovieClip()`.

**Note:** Simple movie clips cannot have children: this includes static and programmatic children.

### Using `attachMovie()` to create movie clip copies

The `attachMovie()` movie clip method creates a new copy of an attachable movie clip. The movie clip copy is attached as a child of `movieClip` at the specified `depth` in `movieClip`'s programmatic stack. The syntax of this method is:

```
movieClip.attachMovie(exportName, newName, depth);
```

|                   |                                                                                                                                                                                                                                |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>exportName</i> | Sharing name assigned to the movie clip in the Export palette. For details on creating sharing names for use with the <code>attachMovie()</code> method, see "Making shareable movie clips (and shareable sounds)" on page 60. |
| <i>newName</i>    | New name given to the attached movie clip to differentiate it from other movie clips in the SWF file.                                                                                                                          |
| <i>depth</i>      | Integer that tells where in <code>movieClip</code> 's programmatic stack to place the movie clip copy.                                                                                                                         |

### Using `duplicateMovieClip()` to create movie clip copies

The movie clip method `duplicateMovieClip()` instructs a movie clip to create a copy of itself. The copy becomes a sibling of the original. The syntax of this method is:

```
movieclip.duplicateMovieClip(newName, depth);
```

|                |                                                                                                           |
|----------------|-----------------------------------------------------------------------------------------------------------|
| <i>newName</i> | String indicating the name of the movie clip copy.                                                        |
| <i>depth</i>   | Integer that tells where in the programmatic stack of the original's parent to place the movie clip copy. |

You can also call `duplicateMovieClip()` as a global function. Instead of copying itself, the global function copies a movie clip passed as an argument. The syntax of this function is:

```
duplicateMovieClip(target, newName, depth);
```

|               |                                                                                                  |
|---------------|--------------------------------------------------------------------------------------------------|
| <i>target</i> | Path or reference to a movie clip or a string indicating the location of the movie clip to copy. |
|---------------|--------------------------------------------------------------------------------------------------|

|                |                                                                                                           |
|----------------|-----------------------------------------------------------------------------------------------------------|
| <i>newName</i> | String indicating the name of the movie clip copy.                                                        |
| <i>depth</i>   | Integer that tells where in the programmatic stack of the original's parent to place the movie clip copy. |

## Static and programmatic stacks

**Note:** *Because the children of a movie clip group also can themselves be parents (that is, movie clip groups) with their own children, this guide uses the term 'movie clip' for simplicity in most cases. If the movie clip has children, by definition it really is a movie clip group.*

Movie clips have two stacks: a static stack and a programmatic stack. A movie clip's static stack contains its *manually created* children. A manually created movie clip starts as a regular object that you create in the Composition window and then convert into a movie clip. A movie clip has a programmatic stack that contains its programmatically generated children.

Figure 19 illustrates the static and programmatic stacks of manually created movie clip A.

Movie clip A's static stack contains its manually created children. Manually created movie clips become the children of a manually created parent when you create a movie clip group that contains them. In Figure 19, "A" is the name of the movie clip group that contains manually created movie clip X and movie clip Y in its static stack.

Immediately above movie clip A's static stack is its programmatic stack. The programmatic stack is where programmatically generated movie clips are placed. Although there can be many levels to the programmatic stack, for simplicity Figure 19 depicts four, with depth values: 0, 1, 2, and 3. Each level of movie clip A's programmatic stack can contain a programmatically generated movie clip that is a child of movie clip A. In the programmatic stack, the movie clip with the highest numeric depth value is the topmost movie clip overlapping all others when the movie clip executes in the Composition window in Preview mode or in the exported SWF file. The movie clip with the next highest numeric depth value overlaps the movie clip with next highest numeric depth value, and so on.

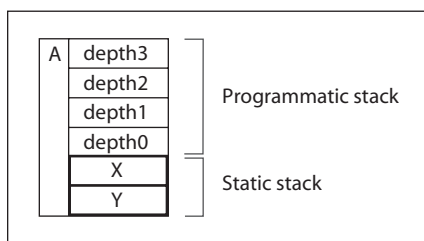


Figure 19 A's programmatic and static stacks

Every movie clip—even those that are created programmatically—makes space for a programmatic stack.



## Manipulating the stack depth with `attachMovie()` and `duplicateMovieClip()`

When you create a movie clip programmatically with `attachMovie()` or `duplicateMovieClip()`, you assign it a `depth` value. `depth` can be any integer value that is 0 or higher. You are not required to assign the depth values to movie clips generated in any particular order.

Assume for this example that movie clip A has no programmatic children. You can attach movie clip instances to movie clip A to create, say, movie clips E, B, and C by making calls to the `attachMovie()` method as shown here:

```
A.attachMovie(exportName, "E", 3);
A.attachMovie(exportName, "B", 0);
A.attachMovie(exportName, "C", 1);
```

Figure 20 (1) depicts the placement of the programmatically generated movie clips in movie clip A's stack after these three calls.

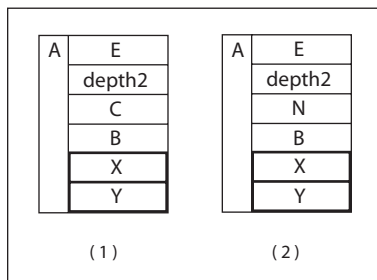


Figure 20 Using `attachMovie()`

A subsequent call to `attachMovie()` specifying a `depth` already occupied just replaces the current movie clip with a new one. So if you call `attachMovie()` again as shown here:

```
A.attachMovie(exportName, "N", 1);
```

Movie clip `N` will replace movie clip `C`, as shown in Figure 20 (2).

The `duplicateMovieClip()` method also creates movie clip copies. However the copies are placed in the programmatic stack of the caller's parent. The new movie becomes a sibling of the movie from which it was duplicated.

Here is an example of the manually created movie clip X creating a duplicate movie clip D:

```
X.duplicateMovieClip("D", 2);
```

Movie clip **D** is placed in movie clip **A**'s programmatic stack, because it is a sibling of movie clip **X**, as shown in Figure 21

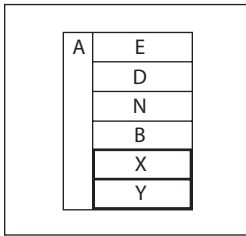


Figure 21 Using `duplicateMovieClip()`

### Using `swapDepths()` to swap movie clip positions in the programmatic stack

You can use the `swapDepths()` method to swap the positions of two movie clips. For this method to work, both movie clips must be siblings. The syntax is either of two forms:

```
movieClip.swapDepths(target);
```

```
movieClip.swapDepths(depth);
```

**target** Path or reference to a movie clip or a string indicating the name of the movie clip to swap depths with *movieClip*.

**depth** Integer that tells where to place *movieClip* in the programmatic stack of *movieClip*'s parent.

When called with the *target* argument, the method swaps depths of *movieClip* and *target*, provided that the movie clips share the same parent.

When called with the *depth* argument, the method places *movieClip* in a new position in the programmatic stack of its parent. If that position is occupied, the movie clip occupying it is moved to *movieClip*'s old position.

### What the programmatic stack does to the movie clip hierarchy

So far you have viewed a composition from the perspective of its movie clip hierarchy and its relationship to z-order for movie clips that are created manually. For details, see "Relationship of movie clip hierarchy to z-order" on page 44. Figure 22 illustrates the effect on the hierarchy when you add programmatically generated movie clips. You can't view this order in the Composition window, however, unless you are in Preview mode or you export the composition to a SWF file. The programmatically generated movie clips appear during the course of execution at the time they are generated.

Figure 22 represents the order of manually and programmatically created movie clips. The dashed lines separate the parent and children movie clips. Movie clips A and B are manually created. Movie clip A has two manually created children, W and X. Like A, movie clip B has two manually created children, Y and Z. Figure 22 (left) shows the manually created movie clips. Figure 22 (right) shows the location of the programmatic stack for `_root`, movie clip A, and movie clip B.

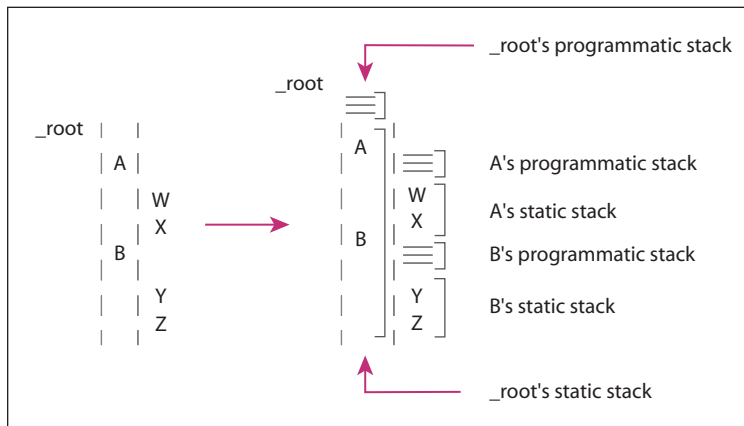


Figure 22 Manually and programmatically created movie clips

Here are two examples that show how attaching and duplicating a movie clip compare. Say that you create movie clip P with `attachMovie()` as shown here:

```
A.attachMovie(exportName, "P", depth);
```

Movie clip P is placed at the specified `depth` in A's programmatic stack. Movie clip P is a programmatic child of movie clip A.

Now, you create a movie clip L with `duplicateMovieClip()`, as shown here:

```
A.duplicateMovieClip("L", depth);
```

Movie clip L is placed at the specified `depth` in `_root`'s programmatic stack, because it is a sibling of movie clip A.

Table 8 illustrates some more examples of programmatically generated movie clips and indicates the stack in which the movie clips are placed.

Table 8 Placement of programmatically generated movie clips

| Method call                                     | Stack and depth where movie clip is placed                                |
|-------------------------------------------------|---------------------------------------------------------------------------|
| <code>A.attachMovie(exportName, "R", 1);</code> | R is placed in A's programmatic stack at depth 1.                         |
| <code>B.duplicateMovieClip("M", 0);</code>      | M is placed in <code>_root</code> 's programmatic stack at depth 0.       |
| <code>B.attachMovie(exportName, "N", 4);</code> | N is placed in B's programmatic stack at depth 4.                         |
| <code>Y.duplicateMovieClip("P", 4);</code>      | P is placed in B's programmatic stack at depth 4, replacing movie clip N. |

| Method call                                            | Stack and depth where movie clip is placed                                 |
|--------------------------------------------------------|----------------------------------------------------------------------------|
| <code>z.attachMovie(<i>exportName</i>, "Q", 2);</code> | Q is placed in Z's programmatic stack at depth 2 (not shown in Figure 22); |

In Table 8, Z's programmatic stack *would be* represented as a fourth view of the composition shown in Figure 22. If Z had manually created children, they would appear in Z's manual stack just below its programmatic stack.

## Making shareable movie clips (and shareable sounds)

*This section describes how to make movie clips available for use with `attachMovie()`, so they can be shared in compositions that you create or in compositions created by other people. The procedures for setting up the mechanism that makes movie clip sharing possible also applies to sharing sounds.*

LiveMotion supports sharing movie clips (and sounds) that can be reused and reproduced in external SWF files. You can share movie clips with LiveMotion and other applications that export to the SWF file format. This feature enables you to leverage content from the vast number of existing SWF files. To make a movie clip shareable, you have to set them up in your composition as described below.

### Setting up shareable movie clips in your composition

#### To make a movie clip sharable in your composition:

- 1 Create any simple object in the Composition window, and convert it to a movie clip.
- 2 Select the movie clip's name in the Timeline window.
- 3 Choose File > Export Settings... or Window > Export to bring up the Export palette.
- 4 In the Export palette menu, select Macromedia® Flash™ (SWF) from the drop-down menu of file types at the top of the palette.
- 5 Click the Animation tab (with the bouncing ball icon) shown in Figure 23.
- 6 To activate the fields and checkboxes beneath the Frame Rate drop-down menu in this tab, click the Object export settings button at the bottom of the tab. See Figure 23.

**Note:** Do not click the Multiple selections button next to Object export settings. It can cause scripts to execute abnormally.

- 7 Check the Attachable checkbox, and enter a sharing name for the movie clip in the text box just below it, as shown in Figure 23.

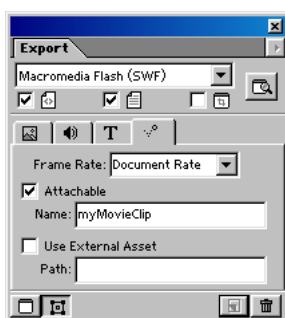


Figure 23 Export palette filled out to make myMovieClip shareable

With the Export palette set up as described, you can make more copies of the movie clip using the `attachMovie()` method. Here is the syntax:

```
movieClip.attachMovie(exportName, newName, depth);
```

If you are attaching a sound instead, this is the syntax of the sound object method:

```
soundObj.attachSound(exportName);
```

To use either of these methods, fill in the shareable name for `exportName`, and provide a unique name for the copy that you are going to make. The `depth` argument to `attachMovieClip()` is described in detail in “Creating movie clips programmatically” on page 55.

If you don’t want the shareable movie clip to appear (or sound to be heard) in your SWF file until it is accessed by scripting code, you can hide it by turning off its eye icon (movie clip) or its speaker icon (sound) in the timeline. The movie clip or sound will be included in the exported SWF file but will not be visible (or audible) until it is accessed dynamically in a script.

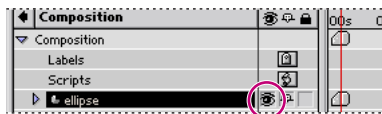


Figure 24 Timeline showing the eye icon toggled on for a movie clip

## Accessing movie clips and sounds in an external SWF file

To access a shareable movie clip (or sound) in an external SWF file, you first create a “placeholder” movie clip in your own composition that you give a sharing name. Then when you export your composition to SWF file format and play it in the Flash Player, your placeholder is replaced by the movie clip in the external file that has the same sharing name.

What is important here is that you must know in advance the sharing name of the movie clip in an external file that you want to use in place of your “placeholder” movie clip. This is a feature from which you can really leverage, because if sophisticated movie clips exist that can be reused, there is little reason to reconstruct them when they can be swapped into a SWF file during playback. Here are the details of the procedure for accessing movie clips in external files.

### To access a movie clip (or sound) in an external SWF:

- 1 Create a simple object such as an ellipse and convert it to a movie clip.
- 2 Give the movie clip a sharing name by repeating steps 1 through 6 in “To make a movie clip sharable in your composition:” on page 60. This procedure uses the sharing name `myMovieClip`.
- 3 In the Export palette Animation tab, check the Use external asset checkbox. See Figure 23.

**4** In the Path: field, enter a reference to the external SWF file containing a movie clip that also has the sharing name (myMovieClip).

**Note:** You can use a relative or an absolute reference to the external SWF file. If, however, you use an absolute reference, it must be a reference within the same domain as the final SWF file. If you plan to move your project to a file server or a Web server, you should use a relative reference.

When you export your SWF file to the Flash Player, your placeholder movie clip named myMovieClip is replaced with movie clip with the same name that is located in the external SWF. If you want to create more copies of myMovieClip from the external SWF file, you can call `attachMovie()` and provide “myMovieClip” as the value of the first argument. If you are working with sounds, call `attachSound()` and provide the sharing name of the sound from an external SWF file as the sole argument to this object method.

## Levels of the Flash Player

In addition to a programmatic stacking order, there is a stacking order that determines the overlapping of SWF files when multiple files are loaded into the Flash Player. The first file loaded is placed in the lowest level of the stack (`_level0`). If additional SWF files are loaded, you can place them at any numeric player level above `_level0`. You can also replace the current SWF file at any level, including `_level0`. The contents of the SWF file at the highest level appears in front of all other SWF files in the player. The contents of the SWF file in the next lower level appears behind the highest, and so forth. A complete SWF file stack can consist of multiple SWF files, each of which can contain multiple movie clips with movie clip duplicates and attached movie clips, each with its own programmatic stack. Figure 25 illustrates SWF file stacking order.

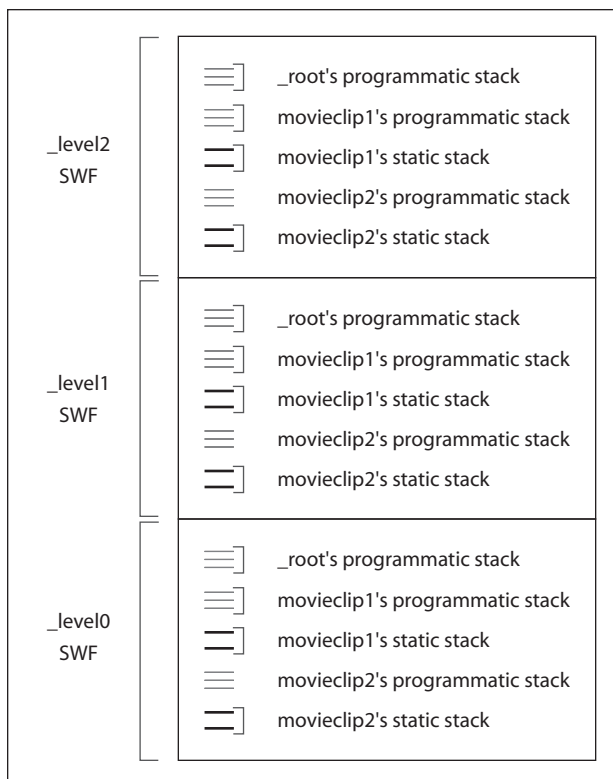


Figure 25 Stacking order of SWF files

`_leveln` (where *n* represents 0 or a non-negative integer value) is a global property that you can use to refer to a SWF file when multiple SWF files are loaded into the Flash Player. It is also an argument to the global functions for loading and unloading SWF files described below. For more information, see the description of this property in “Reference” on page 105.

### Movie clip global functions that use `_leveln` as an argument

You can load SWF files into the Flash Player and unload them from the player using the respective `loadMovie()` and `unloadMovie()` global functions.

#### Using `loadMovie()` to load a SWF file

You can use the `loadMovie()` global function to replace the contents of a movie clip or SWF file level with another SWF file. It replaces an occupied SWF file level or fills an empty one. The syntax is:

```
loadMovie(url, target);
```

|               |                                                                                                                                  |
|---------------|----------------------------------------------------------------------------------------------------------------------------------|
| <i>url</i>    | String specifying the location of an external SWF file to load.                                                                  |
| <i>target</i> | Path or reference to a movie clip indicating the name of the movie clip or Flash Player level into which the SWF file is loaded. |

#### Using `unloadMovie()` to unload a SWF file

You can use the `unloadMovie()` global function, which removes a movie clip or a SWF file. The function takes a `target` parameter. The syntax is:

```
unloadMovie(target);
```

|               |                                                                                                          |
|---------------|----------------------------------------------------------------------------------------------------------|
| <i>target</i> | Path or reference indicating the name of the movie clip or Flash Player level to remove from the player. |
|---------------|----------------------------------------------------------------------------------------------------------|

# Movie Clip Events and Event Handlers

---

## Introduction to events

Events are actions that take place at indeterminate times during the playback of a composition. They are said to occur asynchronously because they occur at any time, not as a result of reaching a particular keyframe on a timeline. Events include such actions as pressing a key, clicking a mouse, and loading a movie clip into the Composition window. For the purposes of this chapter, a state change also is treated as a type of event.

### Event types

LiveMotion supports two basic types of events: movie clip and state change events.

Movie clip events are associated with movie clips. They can be further broken down into system key, mouse, and button events. System-based movie clip events occur as a result of composition playback or loading variables into a movie clip. Key, mouse, and button events occur as a result of a user action such as moving the mouse or pressing a key.

State change events are associated with states. A state change event occurs when a movie clip enters a new state as the result of a call to `lmSetCurrentState()`. The call could be part of a remote rollover, part of some user-defined script, or part of the default button handler scripts added to the movie clip's button handlers when predefined states (over, down or out) are added to the movie clip.

### Event handlers

If you intend to have something occur in your composition as a result of an event that takes place, you must write an event handler. An event handler contains the code that you want to execute in response to the event. When the event occurs, the interpreter in the Flash Player checks if there is a handler written for that event. If there is, the interpreter executes the event handler code.

Each movie clip event handler has a unique name that describes the action to which that handler responds. For example, `onKeyPress`, `onMouseDown`, and `onLoad` are the names of movie clip event handlers that respond to the respective actions: key press, mouse down stroke, and movie clip loading. The event handler names themselves do nothing until you write the code to implement them. A user can click the mouse forever, and nothing special will happen if there is no code written for `onMouseDown` events. The code you write causes the interpreter to execute that code each time a mouse click is detected.

A state change handler responds to the action of changing to the state for which that handler is written. The interpreter executes the handler code whenever the movie clip enters that state.





## System-based events and event handlers

System-based events are actions that are generated by the Flash Player. Table 9 lists the names of the system-based event handlers that LiveMotion supports.

Table 9 System-based event handlers and events

| Event handler | Event causing the handler to be called                                                                                                                      |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| onData        | Either of two unrelated situations: Completion of variables loading into a movie clip or receipt of a portion of an external SWF file by a host movie clip. |
| onLoad        | First appearance of a movie clip in the Composition window.                                                                                                 |
| onEnterFrame  | Each time the playhead enters a frame, before the frame is rendered.                                                                                        |
| onUnload      | Removal of a movie clip from the Composition window.                                                                                                        |

### onData

A data event can occur when all the variables are loaded into the movie clip that were sent by a server-side application as a result of a call to `loadVariables()`. The `onData` handler can notify the composition that the variables are available for use.

A data event can occur in a second situation as well: when a SWF file, or a specific portion of one, has completed loading into a movie clip or a specified SWF file level using the `loadMovie()` function. For information on SWF file levels, see “Levels of the Flash Player” on page 62.

**Note:** The following event handlers are mutually exclusive: only one handler can execute on any given frame.

### onLoad

A load event marks the first appearance of a movie clip in the composition. The `onLoad` event handler executes only once in the lifetime of a movie clip. It occurs on the first frame of a movie clip when the movie clip appears in the composition. If the movie clip executes in a loop that causes its first frame to be replayed, this would not constitute a load event. If, however, a movie clip is unloaded, reloading it again is a new lifetime, and a load event occurs on the movie clip’s first frame.

### onEnterFrame

An enter frame event occurs when the playhead enters a frame. The `onEnterFrame` handler executes on every frame except the first frame, when the `onLoad` event handler of the movie clip executes.

### onUnload

An unload event occurs when a movie clip is removed from the Composition window. The `onUnload` event handler executes on the first frame *after* the movie clip is removed.

## Hands-on example 5\_1: Using system-based event handlers to rotate a movie clip

This hands-on example illustrates how to use the `onLoad` and `onEnterFrame` handlers to define and call a movie clip method that causes a movie clip to rotate itself on every frame.

**To rotate a movie clip:**

- 1 Create a new composition, and save it as `Ex5_1.liv`.
- 2 Create an object in the Composition window, and give it a fill color.
- 3 Select the object, and choose Object > Movie Clip from the main menu to convert it into a movie clip.
- 4 Open the Script Editor by choosing Scripts > Script Editor from the main menu.
- 5 Click the system-based event handler `onLoad` in the drop-down menu of handlers.
- 6 Write an `onLoad` event handler that defines a function to rotate the movie clip when it is called. Here is a script that does this:

```
function rotate(){
 this._rotation += 40;
}
```

- 7 Click the system-based event `onEnterFrame`, in the drop-down menu.
- 8 Write an `onEnterFrame` event handler that calls the `rotate()` function. Here is the call:

```
this.rotate();
```

This function is called to rotate the movie clip on every frame.

- 9 Preview.

**Hands-on example 5\_2: Programmatic bounce**

This example creates a programmatic bouncing ball. Like the previous example, it uses `onLoad` and `onEnterFrame` event handlers. This example uses `onLoad` to initialize conditions, and `onEnterFrame`, to update conditions as the playhead enters each frame. The example also demonstrates the use of the `hitTest()` and `getBounds()` movie clip methods.

**To create a programmatic bounce:**

- 1 Create a new composition, and save it as `Ex5_2.liv`.
- 2 Choose Edit > Composition Settings, and set the frames per second to 20.
- 3 To create the ground, create a rectangle in the Composition window, and position it where you would like the ground to be.
- 4 Convert the rectangle into a movie clip, and name it Ground.
- 5 To create the ball, create an ellipse in the Composition window, and position it at the location from which you would like it to fall.
- 6 Convert the ellipse into a movie clip, and name it Ball.

The movie clips in the Composition should appear something like the ones that are shown in Figure 26.

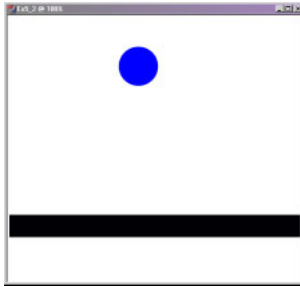


Figure 26 Composition window showing Ground and Ball

**7** Move the anchor point of Ball to the bottom of the ellipse.

The anchor point is the position of the object in scripting. This example sets Ball's position by its bottom.

**8** Double click Ball in the Timeline window, and open the Script Editor.

**9** Click the Handler scripts button (if not already toggled on). Then select the onLoad event handler, and enter this code:

```
this.dx = 0; // initial velocity in x direction pixels/frame
this.dy = 0; // initial velocity in y direction pixels/frame
this.gravity = 2000; //in pixels/frame^2
this.dt = 1/20; //the amount of time that passes between each frame
// with the frame rate is 20 fps.
```

This code initializes the velocity of Ball, the value of `gravity`, and the time between frames. The initial velocity of Ball is 0 in the `x` and `y` directions. The value of `gravity` is 2000 pixels/frame/frame. The time between frames is 1/20 of a second, because the composition is set to 20 frames/second.

**10** Click onEnterFrame in the drop-down menu of event handlers, and enter this code for the handler:

```
// move the ball in the x direction
this._x = this._x + this.dx * this.dt;

// move the ball in the y direction
this._y = this._y + this.dy * this.dt +
.5*this.gravity*this.dt*this.dt;

// if it hits the ground
if(this.hitTest(_root.Ground))
{
 //get the bounds of the ground
```

```
var bounds = _root.Ground.getBounds(_root);
//set the ball at ground level
this._y = bounds.yMin;
//reverse the direction of the y velocity
this.dy = -(this.dy + this.gravity * this.dt);
}
//otherwise
else
{
 //increase the velocity
 this.dy += this.gravity * this.dt;
}
```

This code updates the position and velocity of Ball on every frame. It also checks to see if Ball has hit the ground. If the movie clip intersects the ground, it is moved to be on top of the ground, and its *y* velocity reverse 'bounces' it.

**11** Export and open in your browser.

## Key events and event handlers

Key events are triggered by key actions that are performed by the user while the movie clip is in the Composition window. Unlike button events, key events are not tied to the mouse cursor being over an area of the movie clip for the key handlers to execute. See “Button events and event handlers” on page 71. The only requirement is that the movie clip timeline to which the event handler is added is in the Composition window. Table 10 lists the names of the key event handlers supported by LiveMotion.

Table 10 Key event handlers and events

| Event handler | Event causing the handler to be called                             |
|---------------|--------------------------------------------------------------------|
| onKeyDown     | Pressing a key while the movie clip is in the Composition window.  |
| onKeyUp       | Releasing a key while the movie clip is in the Composition window. |

### onKeyDown

The key down event is generated by pressing a key on the keyboard. The onKeyDown event handler simply indicates that a key has been pressed.

### onKeyUp

The key up event is generated by releasing a key on the keyboard. The onKeyUp event handler simply indicates that a key has been released.

## Using key event handlers

Because the key event handlers just tell you that a key has been pressed or released (but not *which* key), you generally use a key event handler in combination with the `Key` object.

## Key object

There is only one `Key` object. The `Key` object is a built-in object that provides four built-in methods that, when used in combination with a key event handler, can be used to get information about which keyboard keys were pressed, are held down, and are locked down.

### Methods that handle the last key pressed

The `getAscii()` and `getCode()` methods return information about the last key pressed whether or not that key is still pressed. These are useful if you want to know the last key pressed only. To ensure that you have captured the last key pressed, the methods are only useful when called in an `onKeyDown` event handler.

The `getAscii()` method returns the ASCII value of the last key pressed. Values exist for uppercase (shifted state) and lowercase characters.

Each key on the keyboard has a numerical value assigned to it. This value is the keycode. The `getCode()` method returns the keycode of the last key pressed. At the time this method is called, the key may no longer be down.

**Note:** Using the ASCII value alone is less portable than using the keycode, as character codes may differ across different keyboards. If you are writing scripts for international or cross-platform use, the keycode may be more useful.

### Methods that handle keys pressed at the time the method is called

The `isDown()` and `isToggled()` methods handle keys that are pressed when the methods are called regardless of the key last pressed. If, for example, you press 'a' and then press 'b', the event handler `onKeyDown` detects 'b' as the last key pressed. However calling `isDown()` on 'a' still returns true. These methods are useful in many places such as in `onKeyDown`, `onKeyUp`, and `onEnterFrame` handlers.

The `isDown()` method determines if a specific key is currently pressed. `isToggled()` determines whether Caps Lock, Num Lock, or Scroll Lock is toggled on or off.

## Hands-on example 5\_3: Creating an `onKeyDown` event handler

The `onKeyDown` handler in this example uses the `isDown()` method to determine which Arrow key is being pressed and takes the appropriate action, depending on the key.

### To create an `onKeyDown` event handler:

- 1 Create a new composition, and save it as `Ex5_3.liv`.
- 2 Create a simple shape in the Composition window, and give it a fill color.
- 3 Select the object. Choose Object > Movie Clip from the main menu to convert it into a movie clip, and name it Mover.
- 4 Select Mover in the Timeline window, and choose Scripts > Script Editor to open the Script Editor.
- 5 Expand the drop-down menu of events, and click the `onKeyDown` event in the list.
- 6 In the Script window, enter the following code for the `onKeyDown` handler:

```
if (Key.isDown(Key.LEFT))
 _root.Mover._x -= 10;
```

```
if (Key.isDown(Key.RIGHT))
 _root.Mover._x += 10;

if (Key.isDown(Key.UP))
 _root.Mover._y -= 10;

if (Key.isDown(Key.DOWN))
 _root.Mover._y += 10;
```

7 Preview.

Click your mouse cursor on the Composition window to make it the active window. Then, use the arrow keys to move Mover around the window.

# Mouse events and event handlers

Mouse events are triggered by mouse actions that are performed by the user while the movie clip is in the Composition window. Unlike button events, mouse events are not tied to the mouse cursor being over an area of the movie clip for the handlers to execute. See “Button events and event handlers” on page 71. The only requirement is that the movie clip timeline to which the event handler is added is in the Composition window. Table 11 lists the names of the mouse event handlers supported by LiveMotion.

Table 11     Mouse event handlers and events

| Event handler | Event causing the handler to be called                                              |
|---------------|-------------------------------------------------------------------------------------|
| onMouseMove   | Any movement of the mouse cursor while the movie clip is in the Composition window. |
| onMouseDown   | Pressing the mouse button while the movie clip is in the Composition window.        |
| onMouseUp     | Releasing the mouse button while the movie clip is in the Composition window.       |

## onMouseMove

A mouse move event occurs when the mouse position changes. The onMouseMove event handler detects mouse position changes by repeatedly issuing events while the mouse is being moved. You can use the onMouseMove handler to display a mouse trailer. To create a mouse trailer, see “Hands-on example 4\_1: Mouse trailer” on page 51.

## onMouseDown

The onMouseDown event handler is the mouse counterpart to onKeyDown. It detects pressing the mouse button. Mouse down events can be detected only when the mouse cursor is in the Composition window.

## onMouseUp

The onMouseUp event handler is the mouse counterpart onKeyUp. It detects releasing the mouse button. Mouse up events can be detected only when the mouse cursor is in the Composition window.

## Button events and event handlers

Button event handlers execute only when the mouse cursor is on the movie clip in the Composition window. Table 12 lists the names of the button event handlers supported by LiveMotion.

**Note:** `_root` does not support button events, because the composition as a whole cannot not be a button.

Table 12 Button event handlers and events

| Event handler          | Event causing the handler to be called                                                                                                         |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| onButtonPress          | Clicking the mouse button while the cursor is on the movie clip.                                                                               |
| onButtonRelease        | Releasing the mouse button while the cursor is on the movie clip.                                                                              |
| onButtonReleaseOutside | After pressing the mouse button and holding the cursor on the movie clip, moving the mouse cursor off the movie clip and releasing the button. |
| onButtonRollOver       | Moving the mouse cursor on the movie clip.                                                                                                     |
| onButtonRollOut        | Moving the mouse cursor off the movie clip.                                                                                                    |
| onButtonDragOver       | After pressing the mouse button while the mouse cursor is on the movie clip, moving the cursor off and then back on the movie clip.            |
| onButtonDragOut        | After pressing the mouse button while the mouse cursor is on the movie clip, moving the mouse cursor off the movie clip.                       |

### onButtonPress

Button press events occur on the downstroke of a button click. The onButtonPress handler should be used when the user must be decisive. As soon as the button is pressed, the onButtonPress event handler executes.

A mouse down event also is triggered for a button press event if an onMouseDown event handler is defined.

### onButtonRelease

Button release events occur on the upstroke of a button click. Use the onButtonRelease handler when the user should be allowed to change his mind by keeping a button pressed until completely off the button.

A mouse up event also is triggered for a button release event if an onMouseUp event handler is defined.

### **onButtonReleaseOutside**

An event in which the button is released outside is one in which the button must initially be pressed while the mouse cursor is on the movie clip. The event is then generated by holding the mouse button down and moving off the movie clip before releasing the button. The `onButtonReleaseOutside` event handler detects this type of action.

A mouse up event also is triggered for a button release outside event if an `onMouseUp` event handler is defined.

### **onButtonRollOver**

A button rollover event occurs when the mouse cursor is moved onto the movie clip (but not pressed). This action is handled by the `onButtonRollOver` handler.

A mouse move event also is triggered for a button rollover event if an `onMouseMove` event handler is defined.

### **onButtonRollOut**

A button rollout event occurs when the mouse is moved off the movie clip (but not pressed). This action is handled by the `onButtonRollOut` handler.

A mouse move event also is triggered for a button rollout event if an `onMouseMove` event handler is defined.

### **onButtonDragOut**

A button drag out event is similar to a button rollout event except the mouse button is pressed while the mouse is moved off the movie clip. An `onButtonDragOut` handler should be written to handle this action.

A mouse move event also is triggered for a button drag out event if an `onMouseMove` event handler is defined.

### **onButtonDragOver**

A button drag over event starts with the mouse button pressed while on the movie clip. Then the mouse is moved off the movie clip (generating the `onButtonDragOut` event) and moved back on again—all movement taking place while the mouse button is pressed. An `onButtonDragOver` handler should be written to handle this action.

A mouse move event also is triggered for a button drag over event if an `onMouseMove` event handler is defined.

## **Hands-on example 5\_4: Creating a simple button event handler**

It is important to understand that a button is simply a movie clip that has a button event handler defined for it. This example creates a button.

### **To create a button event handler:**

- 1 Create a new composition, and save it as `Ex5_4.liv`.
- 2 Create an object in the Composition window, and give it a fill color.
- 3 Select the object, and choose **Object > Movie Clip** from the main menu to convert it into a movie clip.
- 4 Name the movie clip `Rotate_button`.
- 5 Open the Script Editor by choosing **Scripts > Script Editor** from the main menu.



6 In the Script Editor, click the Handler scripts button if the button is not already toggled on.

7 Expand the drop-down menu of events, and click the `onButtonPress` event.

8 Enter this code for the `onButtonPress` event handler:

```
this._rotation += 30;
```

The code causes `Rotate_button` to rotate itself 30 degrees each time the user presses the button.

9 Preview.

### Hands-on example 5\_5: Creating a toggle button

For LiveMotion 1.0 users, recall that you created a “button” by applying predefined or custom states to an object in the Rollovers palette. This example creates a simple toggle button that has two states: normal and on. By clicking the button, it switches between these states. This example is very useful for creating user interface elements such as radio buttons and check boxes.

#### To create a toggle button:

- 1 Create a new composition, and save it as `Ex5_5.liv`.
- 2 Create an ellipse in the Composition window.
- 3 Give the ellipse the color red.

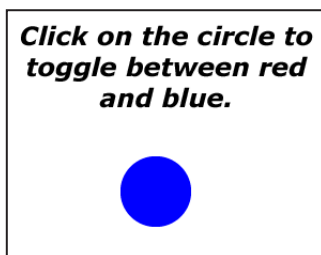


Figure 27 Composition with ellipse

- 4 In the States palette, give the object a custom state, and name the state “on.”

This automatically converts the object to a movie clip.

- 5 Select the “on” state, and give it the color blue.

- 6 In the Timeline window, select your newly created movie clip, press Enter, and name it Toggle.

- 7 With Toggle still selected, open the Script Editor by choosing `Scripts > Script Editor` from the main menu.

- 8 Click the Handler scripts button (if not already toggled on).

- 9 Click the `onLoad` event in the drop-down menu of events, and enter this script:

```
toggleState = false;
```

This `onLoad` event handler code creates the variable `toggleState`, and initializes it to false.

The variable will track the state and value of Toggle.

**10** In the Handler scripts drop-down menu, click the `onButtonPress` event, and enter the following script:

```
if (toggleState == false)
{
 this.lmSetCurrentState("on");
 toggleState = true;
} else {
 this.lmSetCurrentState("normal");
 toggleState = false;
}
```

This `onButtonPress` event handler code creates a simple toggle effect. It switches the current state of Toggle between “normal” and “on,” depending on the value of `toggleState`.

**11** Preview.

**12** Click on Toggle to switch between its normal state and on states.

## State change events and handlers

State change events are triggered when the state of a movie clip changes. All state changes are the result of a call to `lmSetCurrentState()`. However this call could be part of a remote rollover, part of some user defined script, or part of the default button handler scripts associated with the predefined button states (normal, over, down, and out) that give them their default button behavior. For additional information on the default button handlers, see the next section.

## Automatically generated button event handlers

LiveMotion automatically generates code in the movie clip's button handlers to implement the default button behavior for the predefined states. These automatically generated button event handlers are set up to change the state of the movie clip in responses to the appropriate button event. The method used to change the state of the movie clip is `lmSetCurrentState()`. This is the same method that you can use anywhere in your scripts to change state. If, for example, you define the over state for an object, LiveMotion automatically generates this code to set the state to the over state when the mouse cursor is over the movie clip. LiveMotion generates this code to return the movie clip to the normal state when the mouse cursor is no longer over the movie clip.

### Hands on example 5\_6: Experimenting with automatically generated button event handlers

This example creates a predefined state for a button, resulting in LiveMotion automatically generating button event handlers. Then it comments out the automatically generated code to demonstrate that the state change will not occur.

**To automatically generate a button event handler:**

- 1 Create a new composition, and save it as `Ex5_6.liv`.
- 2 Create an ellipse in the Composition window.
- 3 Give the ellipse the color red.
- 4 In the States palette, add the over state to the ellipse.  
This automatically converts the object to a movie clip.
- 5 Select the over state, and give it the color blue.
- 6 In the Timeline window, select your newly created movie clip, and name it Button.
- 7 Open the Script Editor by choosing Scripts > Script Editor from the main menu.
- 8 Click the Handler scripts button (if not already toggled on).
- 9 Click the arrow to the right in the drop-down menu to display all the event handlers. The asterisk (\*) to the left of these button handlers in the list indicates that code (shown here) has automatically been generated.

| button handler   | code generated                                    |
|------------------|---------------------------------------------------|
| onButtonRollover | <code>this._lmSetCurrentState( "over" );</code>   |
| onButtonRollOut  | <code>this._lmSetCurrentState( "normal" );</code> |
| onButtonDragOut  | <code>this._lmSetCurrentState( "normal" );</code> |

- 10 Select the onButtonRollover event. The script associated with this event is:

```
this._lmSetCurrentState("over");
```

- 11 Preview the rollover to verify that it is working.

Button should turn blue when the mouse cursor is over it.

- 12 Exit Preview mode.

- 13 Open the Script Editor, and click the Handler scripts button.

- 14 Select the onButtonRollover event, and comment out the automatically generated code, as shown here.

```
// this._lmSetCurrentState("over");
```

- 15 Preview.

When you pass the mouse cursor over Button, its color does not change from its normal state color to the blue color you gave it for the over state, because you disabled the over state change.

The LiveMotion button behaviors of the predefined states are the default. You don't need to retain these behaviors. You can easily define a new button behavior style. Just comment out LiveMotion's button handler code as you did in this example and write your own. For example, you could create a toggle behavior for the down state such that clicking the button places it in the down state until such time that the button is clicked again to place it in the normal state.

Be aware that if you simply delete the LiveMotion state change script instead of commenting it out, you may not recall why a behavior is not working as it was originally defined.

# Dynamic Data

## Introduction to dynamic data

In LiveMotion, dynamic data refers to the ability to dynamically take data input from a user to set variables and to respond based on the user's specific query. This usually involves communication with a remote Web server or a database. Communications occur over standard Web browser protocols (HTTP or HTTPS) or over TCP/IP sockets. Responses are displayed within a LiveMotion movie clip or within a browser window.

## Forms and text fields

Dynamic data applications are usually based on forms. LiveMotion makes it easy to create powerful forms. A well-designed form ensures that you are soliciting the right information from the user. A form may consist of a single text field into which the user enters information, or it may consist of dozens of text fields strategically laid out on the screen so that it's crystal clear to see how to fill out the form.

Dynamic data user input occurs via the mouse or the keyboard. Mouse input is handled by LiveMotion's `onMouseMove`, `onMouseDown`, and `onMouseUp` event handlers. Keyboard input can also be handled entirely via the event system using the `key` object, but for most dynamic data applications it is handled using LiveMotion text fields in conjunction with on-screen buttons.

### Text field properties

Text fields are used to create forms and to display information received from remote sources. This information can be updated by the user and returned in the same—now updated—text field variables.

LiveMotion allows you to set a variety of text field properties. This occurs through the Properties palette. For example, dynamic text fields can have the Password flag set from the Properties palette pop-up menu (shown in Figure 28), which prevents characters from being displayed when the user types in his password.

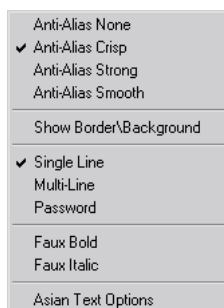


Figure 28 Properties palette pop-up menu



Another important property that is set from the text field Properties palette is the variable name assigned to the text field. The variable name is typed into the Var field (see Figure 29). For example, in the following code, `display` is the name of the text field, and `"My first text field"` is the string value associated with it.

```
_root.display = "My first text field";
```

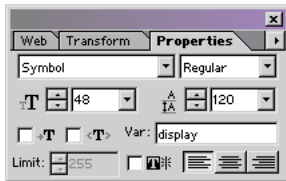


Figure 29 Properties palette

The contents of the `display` variable can be updated by the user, and/or sent to a remote application, and/or modified and returned by a remote application. These tasks are usually accomplished using the `loadVariables()`, `loadVariablesNum()`, `loadMovie()`, `loadMovieNum()`, and `getURL()` global functions and the `loadVariables()`, `loadMovie()`, and `getURL()` MovieClip object methods to send and (in the case of the `loadVariables()` calls) receive variables over the network.

Two other important properties that are set from the text field Properties palette are whether the text field allows users to enter text when it is exported as a SWF file, and whether the text in the text field is interpreted as HTML code. Both of these properties are important to keep in mind when creating text fields for dynamic data applications.

#### To create a text field:

- 1 Select the text field tool.
- 2 Click and drag to create the bounding box of the text field.
- 3 Type into the text field to add default text (initialize the text field with a value).
- 4 In the Timeline window, select the text field (named Dynamic Text by default).
- 5 Choose Object > Edit Name from the main menu, and enter a new name for the text field.
- 6 Choose Window > Properties. In the Properties palette, enter a variable name in the Var field as shown in Figure 29. Then set any other properties of the text field you wish to specify.

Once you assign a variable name to a text field, the text in that text field becomes the value of the variable. The text field is of type string. Even if there are only numbers in the text field, it is still considered a string. If you want to work with the data as numbers, use the `parseInt()` global function. After the text field has been initialized with a string, any value that you enter into the text field—or any modification that you make to the text in the text field—causes the value of the variable to change. In addition, through the scripting language, text field variables can be manipulated like any other variables. Note that when the text field is exported or when it is previewed, any changes to it are automatically saved. Also, there is no real need for a form's "enter" or "submit" button other than to move the user to the next text field or to submit the text field variables to the server.

You will probably want to set the Show Border\Background option in the pop-up menu of the Properties palette (see Figure 28). This places borders around your text fields so that they are easy to see. In addition, the Properties palette allows you to set the text font and size, and to indicate which fonts to embed.

## loadVariables(), loadMovie(), and getURL()

Taking user input is one way of using dynamic data variables. Other ways include using the `loadVariables()`, `loadVariablesNum()`, `loadMovie()`, `loadMovieNum()`, and `getURL()` global functions and the `loadVariables()`, `loadMovie()`, and `getURL()` movie clip methods. These functions and methods allow you to interact with an external data source, usually an application running on a Web server. The `loadVariables()` and `loadVariablesNum()` global functions and the `movieClip.loadVariables()` method allow you to send and receive variable values. The other global functions and movie clip methods only allow you to send variables and their values—the results may then be sent back by the application as a SWF file (`loadMovie()`) or an HTML page (`getURL()`).

**Note:** The `loadVariables()` global function, the `loadVariablesNum()` global function, and the `movieClip.loadVariables()` method are asynchronous in nature—the variables aren't loaded immediately. The timeline continues while data is being retrieved and loaded, at the end of which the `onData` event is raised. The `_root` movie clip, however, has no `onData` event, so an immediate child of `_root` is usually used.

To send variables, you must specify whether the `GET` or `POST` HTTP method is used. For example, the last argument of the `loadVariables()` global function is used to specify the HTTP method:

```
loadVariables("http://www.myServer.com/cgi-bin/stockdata.pl",this,"GET");
```

For all of the `loadVariables()`, `loadMovie()`, and `getURL()` calls, the HTTP method argument is always the last argument and is optional; in each case this argument also indicates that you want to send the variables. If provided, the argument causes LiveMotion to send all of the movie clip's user-defined variables, including the text field variables, according to the method indicated. The Flash Player automatically URL-encodes the outgoing variable strings. The `GET` method has a 1024-character limit and sends the variables tacked onto the URL that is used to contact the remote application (see the `loadVariables()` invocation above). The `POST` method is used for larger amounts of data; this data is sent separately from the URL, and thus data sent via `POST` is not visible to the user of the application, so is more secure. For more information regarding the syntax used to send and receive variables, see "Reference" on page 105.

**Note:** Repeated use of `GET` with the same variables and their values might cause the Web browser to cache the data that's supposed to be returned. To avoid this, use `POST`.

In addition to encoding outgoing variable strings, the player decodes incoming variable strings. To encode and decode, the Flash Player uses the application/x-www-form-urlencoded MIME format. During encoding, this format:

- replaces spaces with a plus (+) sign;
- replaces non-alphanumeric characters by %HH where HH are two hexadecimal digits representing the ASCII code of the character;
- represents line breaks (for multi-line text fields) as CR LF pairs—%0D%0A;

- lists fields in the order that they appear with the variable name separated from the value by an equal sign (=) and from each other by an ampersand (&).

Table 13 summarizes how variables are sent and received using LiveMotion.

Table 13 Calls for Remote Transmission and Reception of Variables

| Global Function or Movie Clip Method            | Use                                                                                                                                                                                                         |
|-------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>loadVariables()</code> global function    | Sends and receives variables. Loads received variable values into a movie clip identified by player level, path, or movie clip reference.                                                                   |
| <code>loadVariablesNum()</code> global function | Same as <code>loadVariables()</code> global function, except that variable values can only be loaded into a movie clip identified by player level.                                                          |
| <code>loadMovie()</code> global function        | Sends variable values. Receives a SWF file, possibly generated based on the values supplied. This file can then be loaded into either a player level or a movie clip, replacing existing contents.          |
| <code>loadMovieNum()</code> global function     | Same as <code>loadMovie()</code> global function except that the SWF file can only be loaded into a player level.                                                                                           |
| <code>getURL()</code> global function           | Sends variable values. Receives results as an HTML file for display in a browser window. Also allows you to execute JavaScript and VBScript code and to execute the <code>fscommand</code> global function. |
| <code>movieClip.loadVariables()</code> method   | Same as <code>loadVariables()</code> global function except that variable values can only be sent from and loaded into <i>movieClip</i> .                                                                   |
| <code>movieClip.loadMovie()</code> method       | Same as <code>loadMovie()</code> global function except that the variables can only be sent from and the SWF file can only be loaded into <i>movieClip</i> .                                                |
| <code>movieClip.getURL()</code> method          | Same as <code>getURL()</code> global function except that variable values can only be sent from <i>movieClip</i> .                                                                                          |

On the server side, the application that receives and sends variables and values can be written in any of a variety of server side scripting languages. The SWF file format is not dependent upon server technology. Some of the more common scripting languages are Perl, Microsoft Active Server Pages (ASP), and PHP. The scripting languages used to create server-side applications that send and receive data have built-in facilities for handling the types of communications described above. The exception is an application that can generate SWF files “on the fly.” Typically, such an application is highly customized.

## How to create a form and send its data to a server

Use the following steps as a guideline for developing a form that takes user input, sends the input to a server, and receives data back. The steps can be modified to create and populate a form that is updated by the user; the contents of the updated text field variables would then be sent to the server.



**To create a dynamic data form in LiveMotion:**

- 1 Start a new composition.
- 2 Create a text field.
- 3 Give the text field the variable name `input` and set the Allow Input option.
- 4 Create a button with three predefined states—normal, over, and down.
- 5 Select the text field and the button and make them into a movie clip group.
- 6 Give the movie clip group the name `formGroup`.

**To create a form to receive data from a server:**

- 1 Create a text field.
- 2 Give the text field the variable name `output`.
- 3 Select the text field, and make it a movie clip group with the name `outputGroup`.

**To send data to a server:**

- 1 Double click on `formGroup` in the Timeline window.
- 2 Select the button.
- 3 In the States palette, select the down state.
- 4 In the Timeline window, double click on the down state for the button to open the Timeline window for the down state. Then click on the Scripts button.
- 5 Enter the following:

```
loadVariables("http://www.myserver.com/processForm.asp" ,
"_root.outputGroup" , "POST");
```

The final step adds the down state button code that will load variables from the `formGroup` movie clip and post them to the ASP page on `www.myserver.com`. This code also causes the loading of the variables from `processForm.asp`. Those variables are then placed into the movie clip `outputGroup`. If those variables already exist in `outputGroup`, they are updated. Otherwise, new variables are created that are actually properties of the `outputGroup` movie clip (to be accessed in the same way as any other movie clip properties or movie clip variables).

The ASP file can specify any number of variable-value pairs. Each pair must be separated with an ampersand and spaces must be URL-encoded so they are replaced with a + sign, as described above where the rules for the application/x-www-form-urlencoded MIME format are outlined.

For example:

```
output=the+form+submitted+correctly&additionalData=valid&eof=1
```

## XML communications

LiveMotion also supports transmission and reception of eXtensible Markup Language (XML) files. Using XML, a LiveMotion application can take input from the user, generate an XML file, and send the file to a server application that parses the XML and stores the data. The application then responds with either an XML file for processing by a movie clip or with an HTML file for display in a Web browser window.

The LiveMotion `XML` class enables you to load, parse, send, build, and manipulate XML document trees. Unlike HTML, which uses a defined set of tags, XML allows you to define your own document tags. For example, the following code shows a simple XML document:

```
<?xml version='1.0'?>
<doc>
<p>Text</p>
<p>More text</p>
<p>See also <xref doc="bestDoc.xml" /></p>
</doc>
```

LiveMotion allows you to either build an XML document from scratch or read in and modify an existing XML document.

Only version r41 and above of the Flash 5.0 Player support XML (r41 was released in December, 2000). Use the `getVersion()` global function to get the version of the Flash Player that you currently have installed. Use of XML with the Flash Player is not dependent on the browser; your browser does not need to support XML to use this capability.

The LiveMotion `XML` class's `send()`, `load()`, and `sendAndLoad()` methods are used to send and retrieve XML documents to/from URLs. Table 14 provides a brief description of each method. The difference between `send()` and `sendAndLoad()` is that the Web server's response to `send()` is an HTML file, whereas the response to `sendAndLoad()` is an XML document. Since they tend to be too large for the `GET` method, the `POST` HTTP method is usually used for sending and receiving XML documents. To support parsing of the data returned from the XML methods, the methods also work in Preview mode. The table below summarizes the XML class's methods used to send and retrieve XML documents. See "Reference" on page 105 for further details.

Table 14 XML Class Methods for Sending and Receiving XML Data

Method	Description
<code>load()</code>	Gets an XML file from a URL.
<code>send()</code>	Sends an XML file to a URL; expects the server to respond with an HTML page for display in a browser window.
<code>sendAndLoad()</code>	Sends an XML file to a URL; expects the server to respond with an XML file for processing and display in a LiveMotion movie clip.

## XML socket communications

LiveMotion also supports XML socket-based communications. Communications using XML sockets are implemented using the `XMLSocket` class.

The `XMLSocket` class implements a client socket that allows the Flash Player to communicate with a server using an “open” connection. A connection using a socket is useful because it remains open—that is, an IP connection doesn’t have to be made between the client and the server each time communications occur between the Flash Player and a server, as is required when the HTTP protocol is used. A “permanent,” two-way, TCP/IP link is set up instead. This enables the Flash Player to listen for incoming messages and process them as they come in. On the server side, this creates a connection where the server can push data directly down to the Flash Player. Real-time communications are enabled.

Only the `XMLSocket` object uses a full-duplex, continuous, TCP/IP connection. The `getURL()`, `loadVariables()`, `loadMovie()`, `XML.send()`, `XML.load()`, and `XML.sendAndLoad()` calls use the HTTP or HTTPS protocol.

The primary characteristics of an XML socket-based application between a Flash Player movie clip and a server are the following:

- There must be a server-side application to wait for the socket connection request and respond to the Flash Player.
- XML messages are sent over a full-duplex TCP/IP connection.
- Each XML message is a complete XML document, terminated by a zero byte (ASCII null character).
- An unlimited number of XML messages can be sent and received over a single connection.

If these are not requirements of your application, use LiveMotion’s other dynamic data functions, objects, and methods, already discussed in this chapter.

The `XMLSocket` implementation in LiveMotion is event-based. These events are coded separately from the built-in event handlers in the LiveMotion scripting environment. The implementation uses four event handlers that use user-defined callback functions to respond to activity on the socket-based connection. The implementation’s three core methods are used to set up a connection and to send XML files. The `XMLSocket` methods are summarized in Table 15. The `XMLSocket` event handlers are summarized in Table 16. See “Reference” on page 105 for further details.

Table 15 XMLSocket Class methods

Method	Description
<code>close()</code> method	Closes an open socket connection.
<code>connect()</code> method	Creates a socket connection to a specified server.
<code>send()</code> method	Sends an XML object to the server.

Table 16 XMLSocket event handlers

Event Handler	Description
<code>onClose()</code> event handler	Callback function that is called when a connection is closed by the server.
<code>onConnect()</code> event handler	Callback function that is called when a connection is created.
<code>onData()</code> event handler	Callback function that is called when data is received but has not yet been parsed as XML.
<code>onXML()</code> event handler	Callback function that is called when data has been received and parsed into an XML object hierarchy.

The application on the server side of an XML connection is more sophisticated than a standard Perl or ASP application. These tend to be applications that work well over UNIX sockets connections on TCP/IP networks and they are often written in Java. They usually host custom-written front ends tuned to handle stringent XML translation and generation.

### Processing incoming data

The following is an example of `XMLSocket` code used to process incoming data.

```
function showData(dataXML)
{
 // act on the XML from the socket
 trace(dataXML.firstChild.nodeValue);
}

// define the socket
dataSocket = new XMLSocket();

// connect to the server at a specified port
dataSocket.connect("http://www.adobeServerOrSomething.com/", 1024);

dataSocket.onXML = showData;
```

# Script Editor

## Introduction to the Script Editor

This section provides details on LiveMotion's Script Editor. It describes the capabilities of each physical component and explains how you can use the functionality to assist in developing your scripts. Every hands-on example in this guide uses the Script Editor at a very high level. You learned how to open the Editor to write scripts to timelines and to movie clip states. This section takes you through all the Script Editor functionality. As you start to create more advanced scripts, you can refer to this section to take advantage of the Script Editor's features.

## Exploring the Script Editor

The Script Editor enables you to write and maintain scripts for your composition while you are in the LiveMotion application. To access the Script Editor you must have a new or an existing composition open in LiveMotion.

### Script Editor window

Figure 30 shows the Script Editor window.

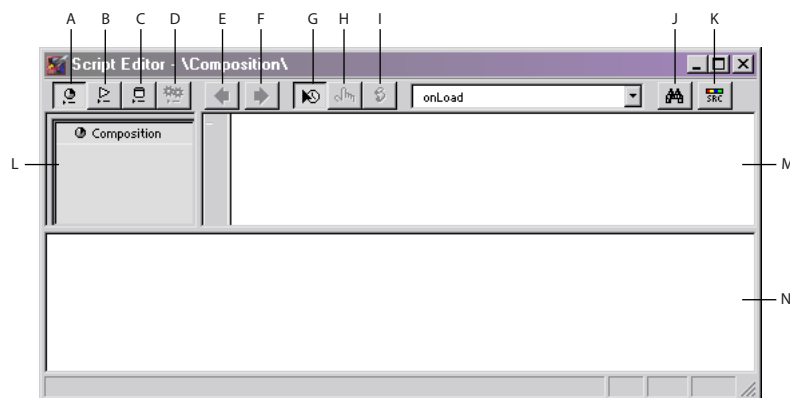


Figure 30 Script Editor main window

- A.** Movie clip navigator **B.** Scripting syntax helper **C.** Composition browser
- D.** Automation syntax helper **E.** Go to previous script **F.** Go to next script
- G.** Handler scripts **H.** State scripts **I.** Keyframe scripts **J.** Find **K.** Syntax highlighting
- L.** Scripting helper window **M.** Script window **N.** Description window

The title bar of the Script Editor window displays a reference to the movie clip whose scripts you are currently editing.

The Script Editor main window is further divided into three main informational views. Clockwise starting with the top left in Figure 30, these are:



- Scripting helper window
- Script window
- Description window

The Scripting helper window displays the tools that can assist you in developing scripting code. These are: Movie clip navigator, Scripting syntax helper, and Composition browser. The Automation syntax helper is not available for creating scripts to be exported to the Flash Player.

The Script window is where you write JavaScript code clip, or view existing scripts, for the current movie clip. To enter code, you can select code from the Scripting syntax helper, or you can simply insert the cursor in this window and start writing code.

The Description window displays descriptions of syntax that you select using the Scripting syntax helper button (described below).

You can adjust the size of the Script Editor's windows. By placing your mouse cursor on the vertical border between the upper windows, you can drag the border left or right to expand or contract window width. By placing your mouse cursor on the horizontal border separating the upper windows from the lower and dragging the mouse up or down, you can expand or contract window height.

## Script Editor buttons

The Script Editor displays a row of buttons just beneath the title bar. Table 17 summarizes the functionality of each of these buttons. Details on these buttons follow the table summary.

Table 17 Script editor buttons and windows

Button or window	Description
Movie clip navigator	Lists all the movie clips in a composition in hierarchical order. Selecting a movie clip in this window allows you to see and edit scripts on that movie clip.
Scripting syntax helper	Lists the LiveMotion 1.0 Behaviors, ActionScript syntax, and JavaScript syntax. Selecting an item in the list displays a brief description of the argument in the Description window. Double-clicking a syntax entry adds the item's syntax to the current script.
Composition browser	Lists all the movie clips, labels, and states in the composition. Selecting an item in the list displays the reference text that will be entered in the Script window. Double-clicking a movie clip, label, or state adds the respective movie clip reference, label name, or state name to the current script.
Automation syntax helper	Lists and describes the global objects and properties in the JavaScript core supported by automation scripting and the pre-defined objects, their methods, and properties in the Automation scripting DOM. For details on automation scripts, see the LiveMotion 2.0 SDK. This button is available when the export format is Live Tab when you are editing an automation script.

Button or window	Description
Go to previous script	Switches the script view to the previously edited script. This button works like the Back button in a Web browser.
Go to next script	Switches the script view to the more recently edited script. This button works like the Forward button in a Web browser.
Handler scripts	Lists all the event handlers in the drop-down menu for which you can write scripts.  This button, as well as the State scripts and Keyframe scripts buttons described below, display a blue triangle when they contain scripts. The contents displayed in the drop-down menu (handler or state names, or keyframe numbers) depend on which of the three buttons is selected. Items in this menu display an asterisk if scripts exist on them.
State scripts	Lists all states in the drop-down menu that are defined for the current movie clip. The list contains the normal state, and it can include the predefined states over, down, and out, plus any custom states defined for the movie clip.
Keyframe scripts	Lists all script keyframes in the drop-down menu for the current movie clip.
Drop-down menu	Displays the keyframes, event handlers, or states for the current movie clip. The contents displayed depend on which of the previous three buttons is selected. Items in this menu display an asterisk if scripts exist on them.
Find	Opens a dialog for finding and replacing text strings in the current script.
Syntax highlighting	Turns syntax highlighting on and off.
Script window	Displays existing scripts and new scripts written to the current movie clip.
Description window	Displays brief descriptions of the syntax listed in the Scripting syntax helper.
Scripting helper window	Displays contents of the Scripting Editor's Movie clip navigator, syntax helper, and browser buttons. The contents displayed depend on which of the buttons is selected.

## Movie clip navigator

The Movie clip navigator indicates which movie clip timeline you are on. When you first open the Script Editor, the Movie clip navigator button is toggled on, and its contents are displayed to the Scripting helper window. Initially, the window displays an expanded list of all the manually created movie clips in hierarchal order.

**Note:** If any movie clip names in your composition contain invalid JavaScript characters such as spaces or punctuation, they are displayed in red in the Movie clip navigator window.

In the Movie clip navigator, the movie clips on the composition timeline are one indent from the left margin. Any movie clips on the timelines of these movie clips are two indents from the left margin, and so on. Figure 31 shows the movie clip hierarchy for the mouse trailer that you created in “Levels of the Flash Player” on page 62. The movie clip icon is displayed to the left of each movie clip name.



Figure 31 Movie clip navigator

### Expanding and collapsing movie clips

By clicking the triangle to the left of a movie clip group name in the Movie clip navigator, you can expand or collapse the movie clip children in that group. For example, if you were to click the triangle next to `MouseTrailer` shown in Figure 31, `Base0` is no longer displayed. Clicking `Composition` collapses everything in the movie clip hierarchy below the composition timeline.

### Navigating the hierarchy

The Movie clip navigator can assist you in locating the correct movie clip to add new scripts to or to locate existing scripts. To access a movie clip's scripts, for example, select the movie clip name in the hierarchy. This takes you to the movie clip's timeline and also updates the contents of the Script Editor's title bar to display the absolute reference to that movie clip. If a movie clip has states defined for it, and a state other than normal is selected when the Script Editor is open, that state appears in parentheses to the right of the movie clip reference. To access the children of movie clip groups, click the triangle next to the group to expand it as necessary, until you locate the child whose scripts you want to access. Once you have accessed the movie clip that you want, you can either select the type of script you want to write, or you can open an existing script you want to access by using the Handler scripts, State scripts, or Keyframe scripts buttons.

### Scripting syntax helper

The Scripting syntax helper assists you with creating the syntax for the LiveMotion 1.0 behaviors, the ActionScript syntax (that is, the extensions to JavaScript that enable you to manipulate movie clips), and the JavaScript core syntax. With the Scripting syntax helper button toggled on, the window displays these syntax groups. By clicking the triangle to the left of a group name, the contents of that group are expanded and displayed to the Scripting helper window. The LM 1.0 Behaviors group lists all the LiveMotion 1.0 behaviors by behavior name. The ActionScript Syntax Helpers group lists the names of all JavaScript extensions for writing movie clip scripts. The JavaScript Syntax Helpers group lists the JavaScript core utilities.



## Syntax helper group entries

The ActionScript and JavaScript groups contain entries with of their own with triangles next to them that you can click to expand to another level of entries. Clicking the triangle next to the Movie Clip Methods entry in the ActionScript Syntax Helpers group, for example, expands the entry to show an alphabetical list of all the movie clip methods. See the Scripting helper window in Figure 32.

Selecting a movie clip method name causes a brief description of that movie clip method to be displayed in the Description window, as shown in Figure 32. The information briefly describes what that method does, what the syntax of the method is, and what each argument to the method is. This is helpful when you want quick access information about how to use the method. For detailed descriptions of all the scripting interfaces that LiveMotion supports, see “Reference” on page 105.

Selecting the method name and pressing Enter (or double clicking the method) generates the syntax for the method in the Script window, as shown in Figure 32.

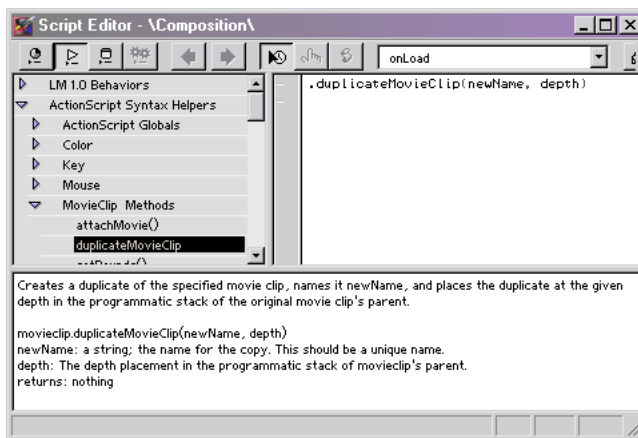


Figure 32 Generating the syntax for the duplicateMovieClip() method

The Scripting syntax helper generates the syntax, but it is up to you to fill in the necessary argument values and anything else that would make the script complete. In the example shown in Figure 32, you would need to provide values for the arguments, *newName* and *depth*. Use the descriptions displayed to help you determine what these arguments represent. If you know the reference to the movie clip making the call, you can fill that in. Otherwise, you can use the Composition browser, described next.

## Composition browser

The Composition browser assists you with generating the correct reference to a movie clip, state, or label. At any time, you can click the Composition browser button to open the browser in the Scripting helper window. The window displays all the movie clips in a composition in hierarchical order. The movie clips on the composition timeline are one indent from the left margin. Any movie clips on the timelines of these movie clips are two indents from the left margin, and so on. Just below the movie clip name, the browser displays the movie clip's states and any label names on its timeline. At the bottom of the Scripting helper window, two radio buttons allow you to choose between generating the absolute or relative reference for a movie clip.

Clicking once on a movie clip name, on a label, or on a state generates the respective movie clip reference (in the style specified by the radio button), label name, or state name in the Description window. This information is generated in this window for your information only. You do not need to delete it. Clicking once on another movie clip name, label, or state removes the current information and generates information for the movie clip, label, or state that you just clicked. This feature enables you to use the Composition browser to examine for possible use the movie clip references, labels, and states at any time as you write scripts.

Double clicking a movie clip name, label, or state generates the respective reference to that movie clip (in the style specified by the radio button), label name, or state name in the Script window at the position of the cursor.

**Note:** If you decide not to use the syntax elements you generated, you must select and delete them from the window.

### Using the Composition browser with the Scripting syntax helper

You also can use the Composition browser in combination with the Scripting syntax helper to fill in placeholders or arguments requiring a movie clip reference, label name, or state name.

When you double click an item from the Scripting syntax helper, the code that gets passed into the editing area (Script window) may not be complete. You may be required to fill in argument values and movie clip references. The procedure below uses the `duplicateMovieClip()` movie clip method as an example.

#### To complete a call to the `duplicateMovieClip()` method:

- 1 Click the Scripting syntax helper to display the ActionScript syntax helpers in the Scripting helper window.
- 2 Expand the Movie Clip Methods list, and double click the movie clip *method* `duplicateMovieClip()`. (Do *not* double click the global method by the same name for this example.)

The code that gets displayed in the Script window appears as:

```
.duplicateMovieClip(newName, depth)
```

If you check the Description window, you will see that the complete syntax for using the `duplicateMovieClip()` method requires that you provide a reference to the movie clip that you want to duplicate. This is indicated by the `movieclip` “placeholder” in the complete syntax, which is shown here:

```
movieclip.duplicateMovieClip(newName, depth)
```

- 3 To correctly form the reference, click the Composition browser button to display its contents in the Scripting helper window.
- 4 Click the radio button at the bottom of the Scripting helper window to select the absolute or relative reference to the movie clip. (This procedure uses the absolute reference.)
- 5 Place the mouse cursor in the Script window to the left of the dot (.) in the syntax.

**6** In the Composition browser, select the movie clip that you want to reference. Then press Return.

The correct reference to the movie clip is inserted before the dot, for example:

```
_root.myMovieClip.duplicateMovieClip(newName, depth)
```

To complete this script, you would provide the appropriate values for the arguments (`newName` and `depth`), and add a semicolon to the end of the statement. You can use the Description window to help you with the meanings of arguments. Here is an example of a completed statement:

```
_root.myMovieClip.duplicateMovieClip("movieClipA", 3);
```

## Go to previous script and Go to next script buttons

These buttons take you to the previous and next scripts. Go to previous script behaves like the Back button in a browser. It traces the history of where you have been. Each time you press the button, it displays the script that was displayed just before the script that currently is being displayed. The Go to next script button does just the opposite: pressing the button displays the script after the current script, and so on. If either of these buttons is active, that means there is another script to go to in that direction. When a button dims, you have reached the last script in the direction you are going. Using these buttons enables you to navigate back and forth through the scripts you have displayed.

## Handler scripts button

The Handler scripts button is used to write event handler scripts to a movie clip and to access existing handlers that have been written. To quickly check if the current movie clip has any event handlers written to it, see if the Handler scripts button has a blue triangle in the top right corner (as shown in Figure 33). If it does, that means event handlers are present.

To quickly see which event handlers have code written for them, see if an asterisk appears in front of the handler's name in the Handler drop-down menu. This indicates that scripts have been written for that handler. Figure 33 shows the Handler scripts button activated. The asterisk indicates that an event handler is written for `onLoad`. Event handler scripts may be written for any number of the handlers listed in the drop-down menu.

### To edit an existing handler, or to write a new handler for current movie clip:

- 1 Click the Handler scripts button to display the current movie clip's handlers in the drop-down menu.
- 2 Expand the drop-down menu, and select the event handler name from the list.
- 3 Write or edit the handler code in the Script window.

You can use the Scripting syntax helper and the Composition browser to help you.

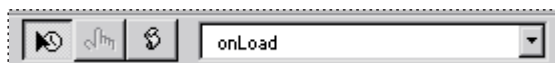


Figure 33 Handler scripts button activated

## State scripts button

The State scripts button is used to write scripts to movie clip states and to access existing state scripts. If the State script button has a blue triangle in the top right corner (as shown in Figure 34), one or more states has scripts written for them.

To quickly see which states have code written for them, see if an asterisk appears in front of the state's name in the state script drop-down menu. This indicates that scripts have been written for that state. The Script Editor window in Figure 34 shows the States scripts button activated and an asterisk indicating that a script is written to the down state.

### To edit an existing state script or to write a new script to a state to the current movie clip:

- 1 Click the State scripts button to display the current movie clip's states as the contents of the drop-down menu.

**Note:** States must be defined for a movie clip before they can be edited in the Script Editor.

- 2 Select the state name in the drop-down menu.

- 3 Write or edit the script in the Script window.

You can use the Scripting syntax helper and the Composition browser to help you.

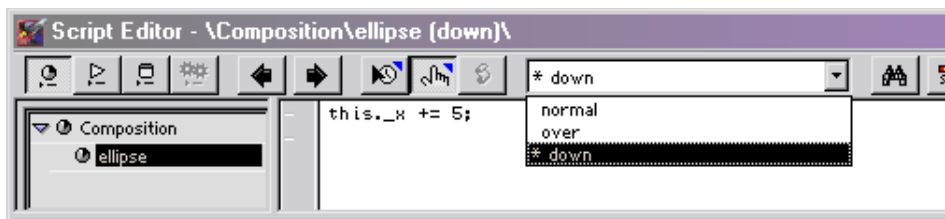


Figure 34 States scripts button activated

## Keyframe scripts button

The Keyframe scripts button is used to write scripts to script keyframes in a movie clip's timeline and to access existing keyframe scripts. A quick way to tell if a movie clip's timeline contains script keyframes is to look for a blue triangle in the top right corner of the keyframe scripts button (as shown in Figure 35). If present, this means script keyframes with custom scripts exist on the timeline.

To quickly see which frames have code written for them, see if an asterisk appears in front of the frame number in the drop-down menu of frame numbers. This indicates that scripts have been written for that frame. The Script Editor window in Figure 35 shows the keyframe scripts button toggled on and an asterisk indicating that a keyframe script is written to frame number 2.

### To edit a keyframe script:

- 1 Click the Keyframes scripts button to display the current movie clip's script keyframes as the contents of the drop-down menu.

**Note:** Script keyframes must be added on the movie clip's timeline before they can be edited in the Script Editor.

- 2 Select the script keyframe from in the drop-down menu.

- 3 Write or edit the script in the Script window.

You can use the Scripting syntax helper and the Composition browser to help you.

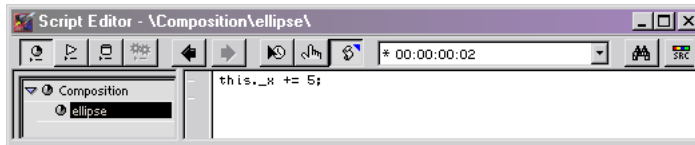


Figure 35 Keyframe scripts button activated

## Find button

The Find button enables you to find and replace text in a script. Clicking the Find button displays a text box in which you can enter the text you are looking for. You have several options for performing your search, including the direction of the search and whether the search should be case sensitive. You can replace the text with text you enter in the Replace with: text box. Click the Close button to end a search.

**Note:** Only the currently displayed script is searched, not all scripts in the composition.

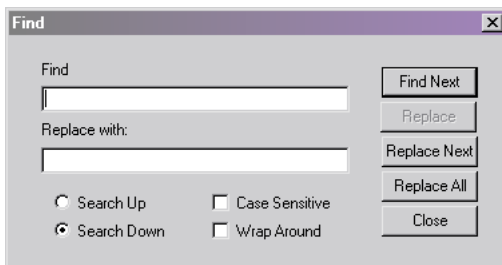


Figure 36 Find text box

## Syntax highlighting button

The Syntax highlighting button is for your coding convenience. If, for example, you want to see all reserved words and values in your code, you can toggle the button to turn on or off the blue font for reserved words and the red font for values.

In addition to these buttons, keyboard shortcuts in the online Help file can assist you in making selections and navigating through code.

# Debugger

## Introduction to the Debugger

LiveMotion has an integrated JavaScript source Debugger that enables you to troubleshoot scripts while you are in the LiveMotion application. This section describes the capabilities of the Debugger's physical components. It explains how you can use the functionality to assist you in troubleshooting your scripts, and it includes short examples illustrating its features. It also describes how the Debugger can be used in combination with the Script Editor and the Script Console window to check output at various points during the execution of the scripts. As you start to create more advanced scripts, you can refer to this section to review ways to take advantage of the Debugger's powerful features.

## Exploring the Debugger

### Bringing up the Debugger

To bring up the Debugger, you must have a composition open. You can choose if and when to activate the Debugger by selecting the appropriate menu item from the Script Editor menu (in LiveMotion's main menu). The Script menu provides three options:

Scripts > Don't Debug	Disables the Debugger.
Scripts > Debug on Errors	Brings up the Debugger when it detects an error during execution of your composition.
Scripts > Debug at Start	Brings up the Debugger when you start Preview mode.

These Debugger modes also are available from a drop-down menu in the Debugger window so that you can change modes during a debugging session.

### Debugger window

#### Main informational views

The main Debugger window is further divided into three main informational views. Clockwise starting with the top left in Figure 37, these are:

- Call stack window
- Variable window
- Source window



You can adjust the size of the windows by dragging your mouse on the window frames. By dragging your mouse on the vertical border between the Call stack and Variable windows, you can move the frame left or right to expand or contract window width. Dragging your mouse up or down on the horizontal border separating the upper windows from the Source window expands or contracts window height.

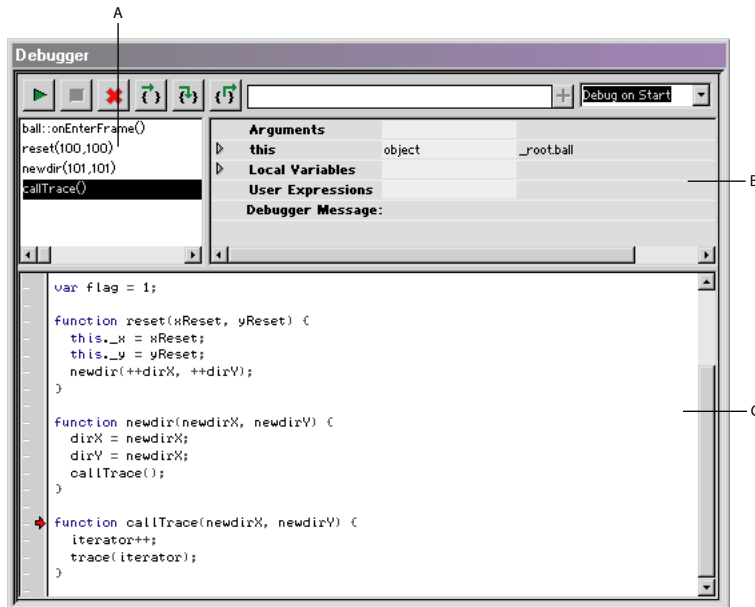


Figure 37 Debugger window  
**A.** Call stack window **B.** Variable window **C.** Source window

The Call Stack window contains a list of functions that are in the process of being executed. The call stack gets deeper as functions call other functions. As functions complete, they are no longer displayed.

The Variable window displays the following types of information:

- Arguments to functions
- Current movie clip object and detailed information about this object's properties
- Local variables
- User expressions
- Debugger messages

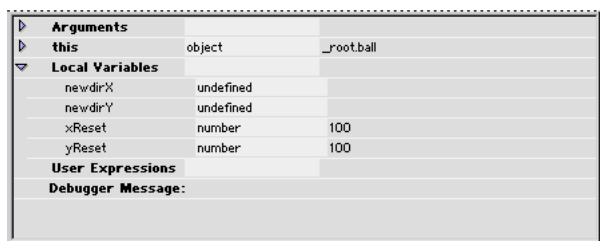


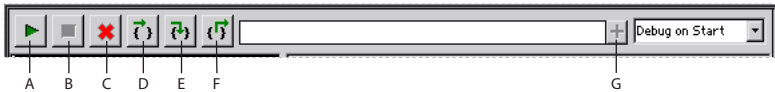
Figure 38 Variable window

By expanding the triangles next to entries in this window, you can view additional window content. Figure 38 shows the types and values of arguments and local variables in the Source code shown in Figure 37.

The Source window displays the JavaScript source when a script is stopped. The position indicator (red arrow in the column on the left side of the window in Figure 37) indicates where execution has most recently stopped. For example, Figure 37 shows the position indicator just before a call to the `trace()` function.

**Debugger buttons**

Just beneath the Debugger window title bar, there is a row of Debugger buttons. These buttons are shown in .



Debugger buttons  
**A.** Run **B.** Stop **C.** Kill **D.** Step **E.** Step into **F.** Step out **G.** Add variable

Table 18 summarizes the functionality of each of the Debugger buttons. Details on these buttons follow the table.

Table 18     Debugger buttons

Button	Description
Run	Plays a script.
Stop	Halts execution.
Kill	Terminates script execution and the Debugger.
Step	Single-steps through instructions.
Step into	Single-steps through instructions, and enters each function call that is encountered.
Step out	Executes the code out of a function call, and stops on the instruction immediately following the call to the function in the calling script.
Add variable (+)	Adds variables and calculations entered in the Variable field to the User Expressions list.

**Run**

The Run button plays a composition until it reaches one of the following:

- The next script to execute
- The next breakpoint
- The next error encountered

You can halt execution by clicking the Stop button or exiting Preview mode.

**Stop**

The Stop button halts execution of the current script. When the button is active, it displays in red.



## Kill

The Kill button terminates the debugging session, closes the Debugger, and returns to your normal editing session. Terminating a debugging session clears all variable values that may have been set during the session. However, it does not clear breakpoints you may have set in the Editor. For details, see “Setting breakpoints” on page 99.

## Step

The Step button single-steps through instructions. Clicking Step at a method call executes the entire method rather than executing one instruction at a time with each click of the button. Say, for example, the Source window shows the position indicator arrow to the left of the `blink()` method, as shown here:

```
-> _root.Ellipse.blink();
```

This location is immediately before the call to `blink()`. Assuming that there are no errors or breakpoints in `blink()`, clicking Step executes the entire `blink()` method, and moves the position indicator arrow to the next script instruction following the method call.

## Step into

The Step into button single-steps through instructions in the code, and enters each function call that is encountered. The `blink()` method definition shown below illustrates how this button works:

```
_root.Ellipse.ctr = 0; // make ctr an Ellipse movie clip property
```

```
// Define the blink method
->_root.Ellipse.blink = function(){
 this.ctr++;
 // _alpha is a built-in movie clip property
 if(this.ctr % 2 == 0)
 this._alpha = 50;
 else
 this._alpha = 100;
}
```

When the position pointer is to the left of the function call, as shown here, clicking Step into takes you to the first statement inside the `blink()` method:

```
-> _root.Ellipse.blink();
```

The first statement in `blink()` is:

```
// Define the blink method
_root.Ellipse.blink = function(){
-> this.ctr++;
.
.
.
```

Each additional click of the Step into button executes the next instruction in `blink()`.

### Step out

The Step out button executes the code out of a function call, and stops on the instruction immediately following the call to the function. Using this button, you can quickly finish executing the current function after determining that a bug is not present. Say, for example, that you are clicking Step into to execute each line of code in `blink()` to monitor the value of `ctr` (as described in “Watching variables” on page 98). If you find that the value is correct, you can click Step out. Doing so executes the remainder of the code in `blink()`, and places the position pointer at the beginning of the next instruction to execute.

### Add variable

The Add variable (+) button accepts the names of variables and expressions that you enter into the Variable field to the immediate left of this button. It displays the current values in the Variable window. If an expression has not yet been defined, the Variable window displays “undefined.”

## Watching variables

While executing code in the Debugger, you can enter the names of variables and expressions whose values you want to monitor in the Variable window.

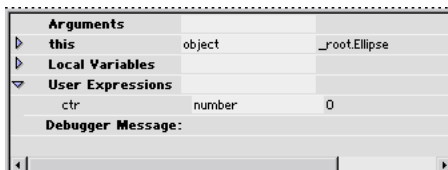


Figure 39 Variable window with ctr variable

### To watch a variable:

- 1 Click your cursor in the expression entry field to the immediate left of the Add variable button
- 2 Enter the name of a variable or an expression whose value you want to monitor.
- 3 Click the Add variable button (or press Enter) to display the variable and its current value in User Expressions in the Variable window.

To save multiple variables in the Variable window, click the Add variable button instead of pressing Enter. Pressing Enter does not save a variable in the window. The variable is replaced by the next one that you enter.

See Figure 39. As long as a variable exists inside the scope of the currently executing function, its value is updated and displayed in the Variable window. If execution takes the Debugger outside of the function, the variable goes out of scope and is no longer displayed.

## Setting breakpoints

A breakpoint is a signal to the interpreter to stop execution at that location, and to enter the Debugger. You can set breakpoints to verify that the values of variables, the current display in your composition, and so forth are what you expect at that point during execution. Breakpoints can be set in two locations: in the Script Editor and in the Debugger.

### To set a breakpoint in the Script Editor:

- 1 Open the Script Editor, and navigate to the script where you want to set a breakpoint.
- 2 Click your cursor in the gray column to the left of the code line at which you want execution to halt.

A breakpoint appears as a red dot in the column. Figure 40 shows a breakpoint to the immediate left of the call to `gotoAndPlay()`.

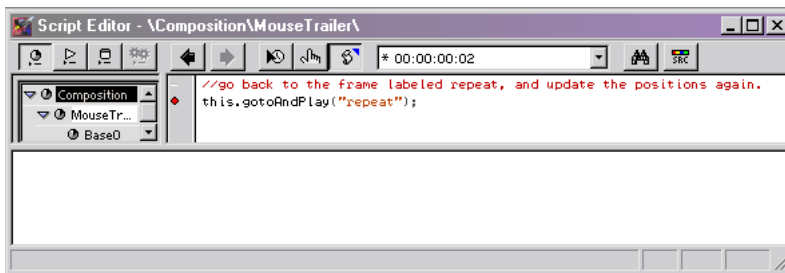


Figure 40 Setting a breakpoint in the Script Editor

### Executing to the breakpoint set in the Script Editor

To execute to the breakpoint just set in the previous section, begin Preview mode. Execution halts at the breakpoint, bringing up the Debugger.

Figure 41 shows the Debugger display after execution has stopped as a result of the breakpoint set in the Script Editor. After executing code to a breakpoint, you can perform whatever checks you need such as noting the values of variables you entered into the Variable window or observing changes in the Composition window.

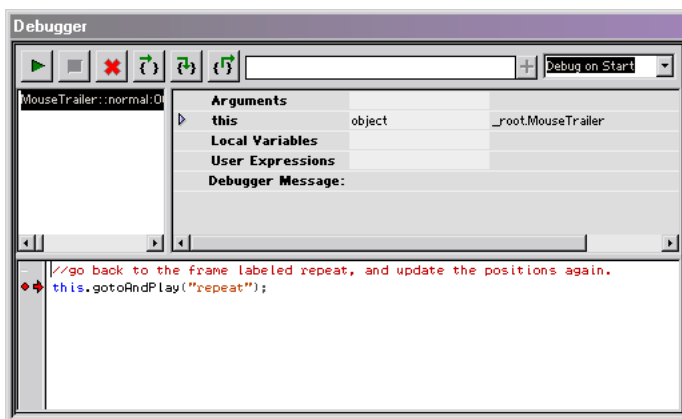


Figure 41 Debugger display after setting a breakpoint in the Script Editor

**To set a breakpoint in the Debugger:**

- 1 Click your cursor in the column to the immediate left of the code line where you want execution to halt.
- 2 Optionally, enter into the Variable window the names of any variables or expressions whose values you would like to examine after executing to the breakpoint.

**Clearing breakpoints**

To clear a breakpoint, click the red dot again. You can clear a break point from either the Script Editor or the Debugger regardless of where it was set. You can also disable breakpoints by Alt clicking them (Windows) or Opt clicking them (Mac OS). This changes them from red to grey.

**Setting a breakpoint in the MouseTrailer onLoad script**

This example leverages on the MouseTrailer hands-on example in “Levels of the Flash Player” on page 62. The code for executing the MouseTrailer is given in that section.

**To set a breakpoint in MouseTrailer:**

- 1 Open your MouseTrailer composition in LiveMotion (Ex4\_1.liv).
- 2 From the Scripts menu, select the Debugger mode, Debug on Start.
- 3 Preview.

When the Debugger first opens, it displays the MouseTrailer’s `onLoad()` handler code in the Source window.

- 4 Click in the gray column to the left of this statement in the onLoad handler:

```
this.trailers[i]._xscale = 100 - i * 10;
```

This sets a breakpoint just before the statement, as shown in Figure 42.

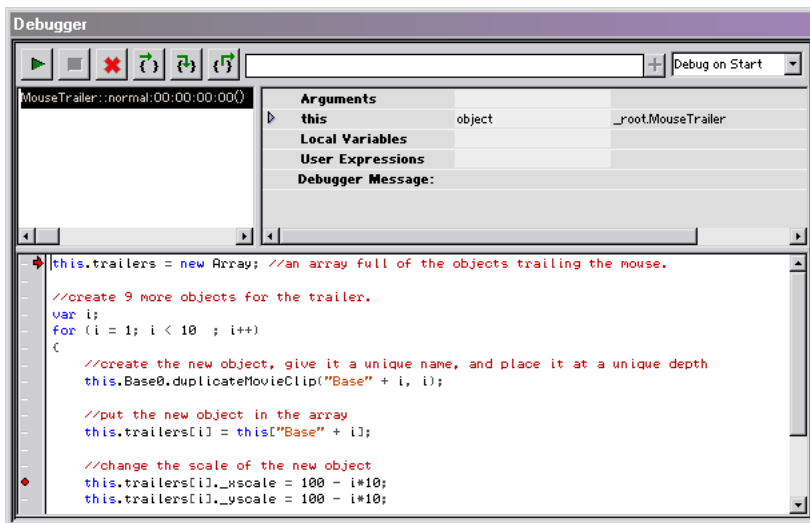


Figure 42 Setting a breakpoint in the Debugger

**To examine variable values in the Mouse Trailer example:**

1 After setting the break point in the previous steps, click the Run button to execute to the breakpoint.

Figure 8.8 shows the result of executing to the breakpoint.

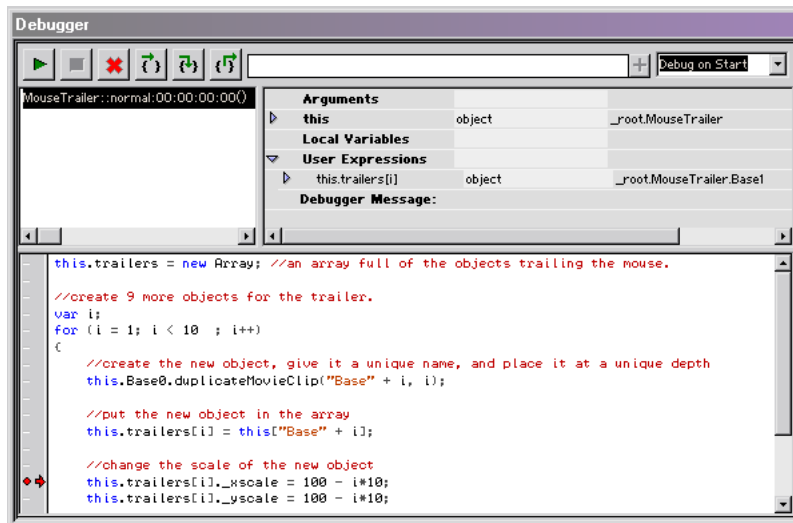


Figure 43 Checking results up to the breakpoint

2 Click the cursor in the expression entry field, and enter this expression:

```
this.trailers[1]
```

3 Click the Add variable button to insert the expression into the User Expressions in the Variable window.

4 By clicking the triangle next to `this.trailers[1]` in the Variable window, you find the values for all of Base1's properties.

Figure 44 shows just some of the information about a movie clip that you can track in the Debugger.

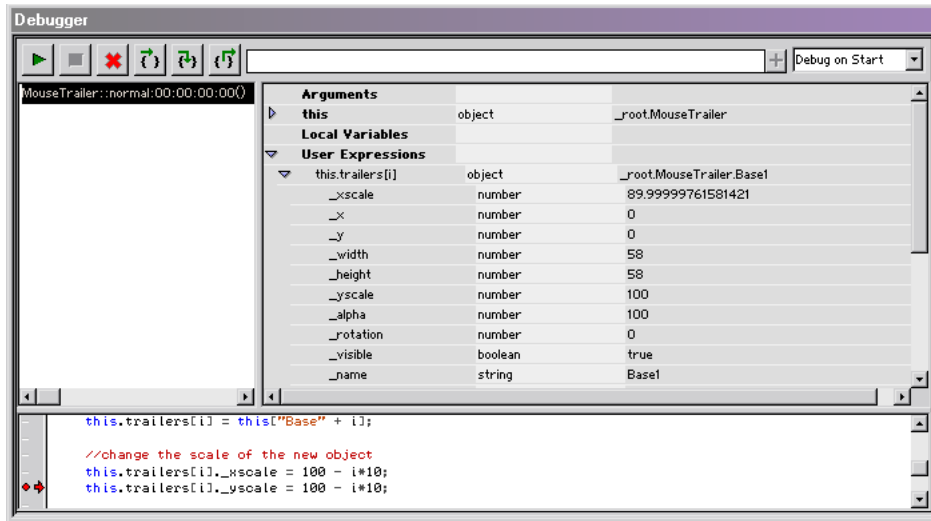


Figure 44 Variable window showing values of Base1's properties

## Using the Console window

The Console window displays script output and the results of `trace()` statements. The types of output displayed include string values, numeric values, and object types. You can keep the window open to monitor results as you preview your composition or execute it in the Debugger.

### Exploring the Console window

To open the Console window, choose **Window > Script Console** from LiveMotion's main menu.

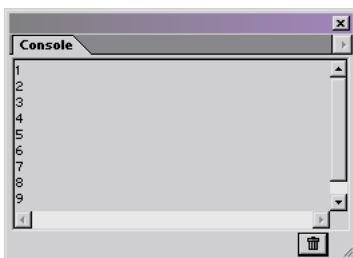


Figure 45 Console window

#### To write to the Console window using a `trace()` statement:

- 1 Open the Script Editor, and navigate to the location where you want to insert a `trace()` statement.
- 2 Insert a `trace()` statement in your script for each variable value that you want to be displayed to the Console window.

For example, to view the values of a counter variable in a `for` loop, you would insert a `trace()` statement as shown here:

```
var i;
for (i = 1; i < 10 ; i++)
{
 trace (i);
.
.
.
```

By playing back your composition in Preview mode or executing through the code in the Debugger, each argument to a `trace()` statement is printed to the Console window followed by a new line character. Each time that you display values to the Console window, the results are appended to the previous output.

**3** To clear the display, click the trash icon at the bottom of the window.

### Using the Console window with the Debugger

You can use the Console window along with the Debugger to watch your variable values. Say for example, you set a breakpoint in your code. Up to that point, you can insert `trace()` statements to monitor the values of certain variables until you identify variables that you would like to see in greater detail in the Debugger. You also can insert `trace()` statements to record multiple values that a variable takes on while a script is executing.

### Comparing Console window output to Debugger output

Although the Console window displays a continuous stream of output for `trace()` statements that are evaluated, it provides less detailed output than you can obtain by watching the evaluation of expressions in the Debugger's Variable window. You can choose which type of output that you want to examine, depending on your needs.

This section looks at the `for` loop code in MouseTrailer's `onLoad` handler. For details on MouseTrailer, see "Levels of the Flash Player" on page 62.

The code below creates the `trailers` array and fills the array elements with duplicated movie clips. Two `trace()` statements have been added to the code shown here. One will display the value of the counter `i`, and the other, the value of the array element

```
this.trailers[i]:
for (i = 1; i < 10 ; i++)
{
 trace (i);
 // create the new object, give it a unique name, and
 // place it at a unique depth

 this.Base0.duplicateMovieClip("Base" + i, i);

 // put the new object in the array

 this.trailers[i] = this["Base" + i];
```

```
trace (this.trailers[i]);

// change the scale of the new object

this.trailers[i]._xscale = 100 - i*10;
this.trailers[i]._yscale = 100 - i*10;

}
```

The Console window shown in Figure 46 displays the `trace()` statement output after five iterations of the `for` loop.



Figure 46 Console output

This is less information than you would get had you entered the counter `i` and `this.trailers[i]` into the Debugger Variable window and stepped through the `for` loop five times. When you expand the triangle next to `this.trailers[i]`, the Debugger displays detailed information about the current movie clip, some of which is shown in Figure 47.

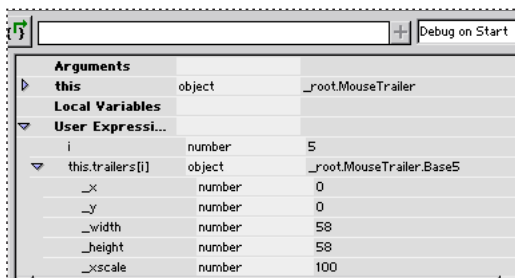


Figure 47 Variable window showing the results of evaluating `i` and `this.trailers[i]`



# Reference

## Introduction

This chapter lists and describes all syntax (keywords, statements, operators, objects, methods, properties, and globals) recognized by the LiveMotion scripting engine.

## Keywords and statement syntax

Table 19 lists and describes all keywords and statements recognized by the LiveMotion scripting engine.

Table 19      Keywords and Statement Syntax

Keyword/State- ment	Description
<code>break</code>	Standard JavaScript construct. Exit the currently executing loop.
<code>continue</code>	Standard JavaScript construct. Cease execution of the current loop iteration.
<code>do - while</code>	Standard JavaScript construct. Similar to the <code>while</code> loop, except loop condition evaluation occurs at the end of the loop.
<code>false</code>	Literal representing boolean false.
<code>for</code>	Standard JavaScript loop construct.
<code>for - in</code>	Standard JavaScript construct. Provides a way to easily loop through the properties of an object.
<code>function</code>	Used to define a function.
<code>if/if - else</code>	Standard JavaScript conditional constructs.
<code>#include</code>	Standard directive used to import files located elsewhere.
<code>null</code>	Assigned to a variable, array element, or object property to indicate that it does not contain a legal value.
<code>return</code>	Standard JavaScript way of returning a value from a function or exiting a function.
<code>switch</code>	Standard JavaScript way of evaluating an expression and attempting to match the expression's value to a <code>case</code> label.
<code>this</code>	Standard JavaScript method of indicating the current object.
<code>true</code>	Literal representing boolean true.
<code>undefined</code>	Indicates that the variable, array element, or object property has not yet been assigned a value.



Keyword/State- ment	Description
<code>var</code>	Standard JavaScript syntax used to declare a local variable.
<code>while</code>	Standard JavaScript construct. Similar to the <code>do - while</code> loop, except loop condition evaluation occurs at the beginning of the loop.
<code>with</code>	Standard JavaScript construct used to specify an object to use in ensuing statements.

## Operators

Table 20 lists and describes all operators recognized by the LiveMotion scripting engine. Table 21 shows the precedence and associativity for all operators.

Table 20 Description of Operators

Operators	Description
<code>new</code>	Allocate object.
<code>delete</code>	Deallocate object.
<code>typeof</code>	Returns data type.
<code>void</code>	Returns undefined value.
<code>.</code>	Structure member.
<code>[ ]</code>	Array element.
<code>()</code>	Function call.
<code>++</code>	Pre- or post-increment.
<code>--</code>	Pre- or post-decrement.
<code>-</code>	Unary negation or subtraction.
<code>~</code>	Bitwise NOT.
<code>!</code>	Logical NOT.
<code>*</code>	Multiply.
<code>/</code>	Divide.
<code>%</code>	Modulo division.
<code>+</code>	Add.
<code>&lt;&lt;</code>	Bitwise left shift.
<code>&gt;&gt;</code>	Bitwise right shift.
<code>&gt;&gt;&gt;</code>	Unsigned bitwise right shift.
<code>&lt;</code>	Less than.
<code>&lt;=</code>	Less than or equal.
<code>&gt;</code>	Greater than.

Operators	Description
>=	Greater than or equal.
==	Equal.
!=	Not equal.
&	Bitwise AND.
^	Bitwise XOR.
	Bitwise OR.
&&	Logical AND.
	Logical OR.
? :	Conditional (ternary).
=	Assignment.
+=	Assignment with add operation.
-=	Assignment with subtract operation.
*=	Assignment with multiply operation.
/=	Assignment with divide operation.
%=	Assignment with modulo operation.
<<=	Assignment with bitwise left shift operation.
>>=	Assignment with bitwise right shift operation.
>>>=	Assignment with bitwise right shift unsigned operation.
&=	Assignment with bitwise AND operation.
^=	Assignment with bitwise XOR operation.
=	Assignment with bitwise OR operation.
,	Multiple evaluation.

Table 21 Operator Precedence

Operators (Listed from highest precedence —top row—to lowest)	Associativity
[ ], ( ), .	left to right
new, delete, -(unary negation), ~, !, typeof, void, ++, --	right to left
*, /, %	left to right
+, -(subtraction)	left to right
<<, >>, >>>	left to right
<, <=, >, >=	left to right
==, !=	left to right
&	left to right

Operators (Listed from highest precedence —top row—to lowest)	Associativity
<code>^</code>	left to right
<code> </code>	left to right
<code>&amp;&amp;</code>	left to right
<code>  </code>	left to right
<code>? :</code>	right to left
<code>=, /=, %=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;=, &amp;=, ^=,  =, +=, -=, *=</code>	right to left
<code>,</code>	left to right

## Reference for Objects, Methods, Properties, and Globals

The remainder of this chapter lists and describes all predefined identifiers recognized by LiveMotion.

### Arguments Object

#### Description

The Arguments object provides two types of information about an executing function:

- the name of the function itself, and
- the arguments that were passed to the function.

The Arguments object is a static object—to use the object, do not create an instance using a constructor. With square bracket notation, the object can be used as an array to access the values of the arguments passed to the function.

#### Properties

<code>callee</code>	See “Arguments.callee Property” on page 108.	Name of the currently executing function.
<code>length</code>	See “Arguments.length Property” on page 109.	Number of parameters passed to the currently executing function. This value can be used to access the individual parameters themselves.

#### Methods

None.

### Arguments.callee Property

`arguments.callee`

#### Description

The `callee` property holds a reference to the currently executing function. This property can only be read.

### Example

```
function selfReferenceTest()
{
 if (arguments.callee == selfReferenceTest)
 trace("true");
 else
 trace("false");
}
```

```
selfReferenceTest();//prints "true"
```

## Arguments.length Property

`arguments.length`

### Description

The `length` property stores an integer specifying the number of parameters passed to the currently executing function. The property can be used to access the names of the individual arguments themselves, using the `arguments` object as an array. The `length` property, however, is not zero-based, so always has a value of one greater than the largest index into the array. This property can only be read.

### Example

```
function baseball(glove, bat)
{
 trace(arguments.length);
 trace(arguments[0]);
 trace(arguments[1]);
}
```

```
baseball("catchers", "wooden");
//prints
//2
//catchers
//wooden
```

# Array Object

## Description

The `Array` object provides the ability to create and manipulate arrays of data. If the `Array` constructor is invoked with a single integer value, the value sets the array length. If two or more values are used, they become the initial values of the array elements, and the array length is determined by the number of values provided. Similarly, a single non-numeric value can be used to initialize the array with a single element with that value.

To call the `Array` object's methods, you must create a new object using the constructor. Alternatively, you may use the square bracket syntax (e.g., `var x = [a,b]` populates the first two elements of the array with the values `a` and `b`). If the `Array` constructor is invoked without passing arguments to `Array`, then an empty array is created with zero elements.

## Constructor

```
new Array()
new Array(length)
new Array(element0, ...elementn)
```

## Parameters

<i>length</i>	A non-negative integer indicating the number of elements in the array.
<i>element0</i> , ... <i>elementn</i>	One or more values that are assigned as array elements.

## Properties

<code>length</code>	See "Array.length Property" on page 112.	Number of elements in the array.
---------------------	------------------------------------------	----------------------------------

## Methods

<code>concat()</code>	See "Array.concat() Method" on page 111.	Concatenate elements to an existing array to create a new array.
<code>join()</code>	See "Array.join() Method" on page 112.	Join all elements of the array into a string.
<code>pop()</code>	See "Array.pop() Method" on page 113.	Pop the last element in the array (return the value and remove from the array).
<code>push()</code>	See "Array.push() Method" on page 113.	Push an array element onto the end of the array (add an element).

<code>reverse()</code>	See “Array.reverse() Method” on page 114.	Reverse the order of the elements in the array in place (last element becomes first; first element becomes last).
<code>shift()</code>	See “Array.shift() Method” on page 114.	Same as <code>pop()</code> except the <i>first</i> element is returned and removed from the array.
<code>slice()</code>	See “Array.slice() Method” on page 115.	Copy a subset of an existing array to create a new array consisting of just those elements.
<code>sort()</code>	See “Array.sort() Method” on page 116.	Sort the elements of the array in place.
<code>splice()</code>	See “Array.splice() Method” on page 117.	Add or delete array elements.
<code>toString()</code>	See “Array.toString() Method” on page 119.	Convert an array to a string of comma-delimited values (can also be achieved using <code>join()</code> without a parameter).
<code>unshift()</code>	See “Array.unshift() Method” on page 119.	Add one or more elements to the beginning of the array and return the new length of the array.

## Array.concat() Method

`arrayObj.concat(value1, ...valuen)`

### Description

The `concat()` method concatenates elements to an existing array to create a new array. The original array is left unmodified. If an array is provided as a parameter to `concat()`, each of its elements are appended as separate array elements to the end of the new array.

### Parameters

`value1, ...valuen` Any number of values to be added to the end of the array. Can also be arrays to be concatenated to the current array.

### Returns

A new array formed by the concatenation of the specified values or arrays to the current array.

### Example

```
var a=[1,2,3];
b = a.concat(4,5);
c = b.concat([5,6]);
d = c.concat([7,8],[9,10]);
```

```
e = 0;
for(i=0; i<d.length;i++)
 e = e + d[i];
trace(e); //prints 60
```

**See also**

“Array.push() Method” on page 113, “Array.pop() Method” on page 113, “Array.shift() Method” on page 114, “Array.unshift() Method” on page 119

## Array.join() Method

```
arrayObj.join()
arrayObj.join(delimiter)
```

**Description**

The `join()` method joins all elements of the array into a string; each element is separated by *delimiter*.

**Parameters**

<i>delimiter</i>	(Optional) A string to separate each element of the array. If omitted, the array elements are separated with a comma and results are the same as those achieved with <code>arrayObj.toString()</code> .
------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Returns**

The string containing the joined elements and delimiters.

**Example**

```
baseball = new Array("bat", "ball");
baseballString = baseball.join();
trace(baseballString); // prints "bat,ball"
newString = baseball.join(" + ");
trace(newString); // prints "bat + ball"
```

**See also**

“Array.toString() Method” on page 119, “String.split() Method” on page 220, “Array.sort() Method” on page 116, “Array.reverse() Method” on page 114

## Array.length Property

```
arrayObj.length
```

**Description**

The `length` property is a positive integer that represents the length of the array. Since array indices start with 0 (zero-based indexing), `length` is one greater than the last index value of the array. `length` is initialized when the array is created. This property can be read or written.



### Example

```
baseball = new Array();
trace(baseball.length);//prints 0
moreBaseball = new Array("bat", "ball");
trace(moreBaseball.length);//prints 2
moreBaseball[2] = "glove";
trace(moreBaseball.length);//prints 3
```

## Array.pop() Method

*arrayObj.pop()*

### Description

The `pop()` method pops the last element of the array, returns the value of the element, removes the element from the array, and decreases `length` by 1.

### Returns

The value of the deleted array element.

### Example

```
var stack = [1,2,3];
trace(stack.pop());//stack is now [1,2] and pop prints 3
```

### See also

“Array.push() Method” on page 113, “Array.shift() Method” on page 114, “Array.unshift() Method” on page 119, “Array.concat() Method” on page 111

## Array.push() Method

*arrayObj.push(value1, ...valuen)*

### Description

The `push()` method appends one or more values onto the end of the array and increases `length` by *n*.

### Parameters

<i>value1, ...valuen</i>	Any number of values to be pushed onto the end of the array.
--------------------------	--------------------------------------------------------------

### Returns

The new `length` of the array.

### Example

```
var stack = [1,2,3];
trace(stack.push(4,5));//stack is now [1,2,3,4,5] and push() prints 5
for(i=0; i<stack.length;i++)
```

```
 trace(stack[i]);
//prints
//1
//2
//3
//4
//5
```

**See also**

“Array.pop() Method” on page 113, “Array.shift() Method” on page 114, “Array.unshift() Method” on page 119, “Array.concat() Method” on page 111

## Array.reverse() Method

```
arrayObj.reverse()
```

**Description**

The `reverse()` method reverses the order of the elements in the array in place (last element becomes first; first element becomes last).

**Example**

```
var baseball = ["bat", "ball", "glove", "base"];
for(i=0; (i != 4); ++i)
 trace(baseball[i]);
//prints
//bat
//ball
//glove
//base
baseball.reverse();
for(i=0; (i != 4); ++i)
 trace(baseball[i]);
//prints
//base
//glove
//ball
//bat
```

**See also**

“Array.sort() Method” on page 116

## Array.shift() Method

```
arrayObj.shift()
```

## Description

The `shift()` method is the same as `pop()` except the *first* element is returned and removed from the array. As a result, the array `length` is reduced by 1.

## Returns

The value of the deleted array element.

## Example

```
fish = ["shark", "guppy", "red fish", "blue fish"];
trace(fish.shift()); //prints "shark"
i=0;
while (fish[i] != "blue fish")
{
 trace(fish[i]);
 ++i;
}
trace(fish[i]);
//prints
//guppy
//red fish
//blue fish
```

## See also

“[Array.push\(\) Method](#)” on page 113, “[Array.pop\(\) Method](#)” on page 113, “[Array.unshift\(\) Method](#)” on page 119, “[Array.concat\(\) Method](#)” on page 111

## Array.slice() Method

```
arrayObj.slice(start)
arrayObj.slice(start, end)
```

## Description

The `slice()` method copies a subset of an existing array to create a new array consisting of just those elements. `start` and `end` are indices into the array (zero-based indexing). The slice begins with `start` and continues up to, but not including, `end`. If `start` or `end` is a negative number, the index is equal to the total number of elements in the array minus the absolute value of the number.

## Parameters

start	The array index at which to begin the slice. Can also be a negative number.
end	( <i>Optional</i> ) The array index at which to end the slice. The slice does not include this element. If this argument is not present, the slice extends all the way to the end of the array. Can also be a negative number.

## Returns

A new array that begins with array element *start* and contains all array elements between *start* up to, but not including, array element *end* of the original array.

## Example

```
function printArray(arrayId)
{
 for(i=0; i<arrayId.length; i++)
 trace(arrayId[i]);
}

var a = [1,2,3,4,5];
b = a.slice(0,3);
printArray(b); //prints 1,2,3
b = a.slice(3);
printArray(b); //prints 4,5
b = a.slice(1,-1);
printArray(b); //prints 2,3,4
b = a.slice(-3,-2);
printArray(b); //prints 3
```

## See also

“Array.splice() Method” on page 117

## Array.sort() Method

```
arrayObj.sort()
arrayObj.sort(userFunction)
```

## Description

The `sort()` method sorts the elements of *arrayObj* in place. If no argument is provided, the elements are sorted in alphabetical order. To sort the array in any other order, you have to supply a function that compares two array elements and returns a value indicating how they should be sorted. For *userFunction(a,b)*, if the return value is:

- less than 0, then *b* is sorted to a lower index than *a*;
- 0, then *a* and *b* are left unchanged with respect to each other, but are sorted with respect to all different elements;
- greater than 0, then *b* is sorted to a higher index than *a*.

## Parameters

userFunction	(Optional) A user-supplied function that dictates sort order. If omitted, the array is sorted lexicographically (in dictionary order) according to the string conversion of each element.
--------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Example

```
fish = new Array("shark", "guppy", "red fish", "blue fish");
fish.sort();
for(i=0; (i != fish.length); ++i)
 trace(fish[i]);
//prints
//blue fish
//guppy
//red fish
//shark
```

```
function numberOrder(a,b) { return a - b; }
a = new Array(33,4,1111,222);
a.sort();
for (i=0;i<a.length;i++) {
 trace(a[i]);
}
a.sort(numberOrder);
for (i=0;i<a.length;i++) {
 trace(a[i]);
}
//prints
//1111
//222
//33
//4
//4
//33
//222
//1111
```

### See also

“Array.join() Method” on page 112, “Array.reverse() Method” on page 114

## Array.splice() Method

```
arrayObj.splice(start)
arrayObj.splice(start, num)
arrayObj.splice(start, num, value1, ...valuen)
```

## Description

The `splice()` method removes *num* elements from an array beginning at *start.splice()* optionally inserts new elements starting at zero-based index *start*. To ensure element contiguity, `splice()` moves elements up to fill in any gaps.

## Parameters

<i>start</i>	The (zero-based) index of the first array element to remove. If <i>start</i> is a negative value, <i>start</i> is relative to the end of the array (the index is the number of elements in the array minus the absolute value of the value).
<i>num</i>	(Optional) The number of array elements to remove, including <i>start</i> . If 0, no elements are removed. If <i>num</i> is omitted, all elements from array index <i>start</i> to the end of the array are removed.
<i>value1,...valuen</i>	(Optional) Any number of values to be added to the array starting at index <i>start</i> .

## Returns

An array consisting of any elements that were spliced from the array.

## Example

```
fishAndNumbers = new Array(1,2, "shark", 3, "guppy");
fishAndNumbers.splice(2,2,6,"red fish");
for(i=0; (i != fishAndNumbers.length); ++i)
 trace(fishAndNumbers[i]);
//prints
//1
//2
//6
//redfish
//guppy

fishAndNumbers = new Array(1,2, "shark", 3, "guppy");
fishAndNumbers.splice(-3,2,6,"red fish");//negative start index
for(i=0; (i != fishAndNumbers.length); ++i)
 trace(fishAndNumbers[i]);
//prints
//1
//2
//6
//red fish
//guppy
```

**See also**

“Array.slice() Method” on page 115

## Array.toString() Method

```
arrayObj.toString()
```

**Description**

The `toString()` method converts an array to a string and returns the string. Yields the same result as the `arrayObj.join()` method when that method is used without a parameter.

**Returns**

A comma-separated list of all the elements of the array.

**Example**

```
fishAndNumbers = new Array(1,2, "shark", 3, "guppy");
trace(fishAndNumbers.toString()); //prints "1,2,shark,3,guppy"
```

**See also**

“Array.join() Method” on page 112, “Array.reverse() Method” on page 114, “Array.sort() Method” on page 116, “Object.toString() Method” on page 200

## Array.unshift() Method

```
arrayObj.unshift(value1, ...valuen)
```

**Description**

The `unshift()` method adds elements to the beginning of the array.

**Parameters**

<i>value1, ...valuen</i>	The values of one or more elements to be added to the beginning of the array, starting at index 0.
--------------------------	----------------------------------------------------------------------------------------------------

**Returns**

The new array length.

**Example**

```
fishAndNumbers = new Array(1,2, "shark", 3, "guppy");
trace(fishAndNumbers.unshift(2,6,"red fish")); //prints return value of 8
for(i=0; (i != fishAndNumbers.length); ++i)
trace(fishAndNumbers[i]);
//prints
//2
//6
//red fish
```

```
//1
//2
//shark
//3
//guppy
```

**See also**

“Array.push() Method” on page 113, “Array.pop() Method” on page 113, “Array.shift() Method” on page 114, “Array.concat() Method” on page 111

## Boolean() Global Function

`Boolean(value)`

**Description**

The `Boolean()` global function converts its parameter to a primitive boolean value and returns the value. Do not confuse this global function with the `Boolean` object.

**Parameters**

*value*                      The value to convert to primitive boolean.

**Returns**

The primitive boolean value of *value* (`true` or `false`).

**Example**

```
var testFalse = 0;
var testTrue = true;
trace(Boolean(0)); //prints "false"
trace(Boolean(1)); //prints "true"
trace(Boolean(true)); //prints "true"
trace(Boolean("true")); //prints "false" - not a valid non-zero number
trace(Boolean(false)); //prints "false"
trace(Boolean(testFalse)); //prints "false"
trace(Boolean(testTrue)); //prints "true"
```

**See also**

“Boolean Object” on page 121, “String() Global Function” on page 214, “Number() Global Function” on page 194



# Boolean Object

## Description

The `Boolean` object provides support for boolean values. The `Boolean()` constructor with the `new` operator converts its parameter to a boolean value and returns a `Boolean` object wrapper containing the value. This allows the object to inherit the methods of the `Object` class (see “Object Class” on page 199).

## Constructor

```
new Boolean()
new Boolean(value)
```

## Parameters

value	( <i>Optional</i> ) The value that is converted to a boolean—can be a number, string, boolean, or object. The values <code>0</code> , <code>NaN</code> , <code>null</code> , the empty string ( <code>""</code> ), and <code>undefined</code> all return <code>false</code> . All other values return <code>true</code> . If this parameter is omitted, the <code>Boolean</code> object is initialized with a value of <code>false</code> .
-------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Properties

None.

## Methods

<code>toString()</code>	See “ <code>Boolean.toString()</code> Convert the value of the <code>Boolean</code> object to a string. Method” on page 121.
<code>valueOf()</code>	See “ <code>Boolean.valueOf()</code> Return the primitive boolean value of the object. Method” on page 121.

## Boolean.toString() Method

```
bool.toString()
```

### Description

The `toString()` method returns the string representation of the value of `bool`. The method returns the string `true` if the primitive value of `bool` is `true`; otherwise it returns the string `false`.

### Example

```
bool = new Boolean(1);
trace(bool.toString()); // displays "true"
```

## Boolean.valueOf() Method

```
bool.valueOf()
```

## Description

The `valueOf()` method returns the primitive value of `bool`. The method returns `true` if the primitive value of `bool` is true; otherwise it returns `false`.

## Example

```
bool = new Boolean("");
trace(bool.valueOf()); //prints "false"
```

# Color Object

## Description

The `Color` object supports access to and control of the color of a movie clip. It allows you to get and set the red, green, and blue (RGB) color values and transformation information. You must create an instance of the `Color` object for a specific target before using any of the `Color` methods.

## Constructor

```
new Color(target)
```

## Parameters

<code>target</code>	A path or a reference to the movie clip for which the <code>Color</code> object is created.
---------------------	---------------------------------------------------------------------------------------------

## Properties

None.

## Methods

<code>getRGB()</code>	See “Color.getRGB() Method” on page 122.	Return the RGB offset values for the object.
<code>getTransform()</code>	See “Color.getTransform() Method” on page 123.	Return the current offset and percentage values as an object of type <code>Object</code> . For more information on the <code>Object</code> class, see “Object Class” on page 199.
<code>setRGB()</code>	See “Color.setRGB() Method” on page 124.	Set the RGB offset values for the object.
<code>setTransform()</code>	See “Color.setTransform Method” on page 124.	Set the offset and/or percentage values using an object of type <code>Object</code> . For more information on the type <code>Object</code> , see “Object Class” on page 199.

## Color.getRGB() Method

```
colorObject.getRGB()
```

## Description

The `getRGB()` method returns the RGB color offset values for *colorObject* as one number. These are the values that were set by a call to `setRGB()`. If the offsets have never been set (via `setRGB()`) then the default values for the RGB offsets are 0, 0, 0.

## Returns

A number indicating the RGB color offsets of *colorObject* in the form

`red<<16|green<<8|blue`.

## Example

```
redBaseball = new Color(_root.baseball);
redBaseball.setRGB(0xFF0000);
trace(redBaseball.getRGB()); //prints 16711680
```

## See also

"Color.setRGB() Method" on page 124.

# Color.getTransform() Method

*colorObject*.getTransform()

## Description

The `getTransform()` method returns an object of type `Object` whose properties are the transformation values of *colorObject*.

The properties are the following:

- `ra` is the red transformation percentage (-100 to 100)
- `rb` is the red offset (-255 to 255)
- `ga` is the green transformation percentage (-100 to 100)
- `gb` is the green offset (-255 to 255)
- `ba` is the blue transformation percentage (-100 to 100)
- `bb` is the blue offset (-255 to 255)
- `aa` is the alpha transformation percentage (-100 to 100)
- `ab` is the alpha offset (-255 to 255)

The final value for each color is computed as: `value = original * (transformation percentage) + offset`.

## Returns

An object of type `Object` whose properties contain the transformation values of the movie clip *colorObject*.

## Example

```
redFish= new Color(_root.fish);
fishChanger = new Object();
fishChanger.ra = 100; //Red percentage
fishChanger.rb = 200; //Red offset
```

```
fishChanger.ga = 0;//Green percentage
fishChanger.gb = 0;//Green offset
fishChanger.ba = 100;//Blue percentage
fishChanger.bb = 50;//Blue offset
fishChanger.aa = 40;//Alpha percentage
fishChanger.ab = -10;//Alpha offset
redFish.setTransform(fishChanger);
fishChanger = redFish.getTransform();
fishChanger.rb = 300;//set the Red offset
fishChanger.ga = 20;//set the Green transformation percentage
redFish.setTransform(fishChanger);//changes the transformation values
```

**See also**

“Color.setTransform Method” on page 124, “Object Class” on page 199

## Color.setRGB() Method

```
colorObject.setRGB(offsetValue)
```

**Description**

The `setRGB()` method sets the RGB color offsets for `colorObject`. It also sets all the transformation percentages to 0, which results in the ignoring of the movie clip's original color and the setting of its color to the values of the offsets. The following are suggestions for creating `offsetValue`:

- `offsetValue = red<<16|green<<8|blue` where *red*, *green*, and *blue* are values from 0 to 255;
- `offsetValue = 0xRRGGBB`, where *RR*, *GG*, and *BB* are hexadecimal values for each color and are in the range from 00 to FF.

**Parameters**

<code>offsetValue</code>	An integer in the range of 0 to 16777215 (0xFFFFFFFF), can be a hexadecimal number (0x) indicating the offsets for each of the color offset values.
--------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------

**Example**

```
redBaseball = new Color("_root.baseball");
redBaseball.setRGB(0xFF0000);
trace(redBaseball.getRGB());//prints 16711680
```

**See also**

“Color.getRGB() Method” on page 122

## Color.setTransform Method

```
colorObject.setTransform(transformObj)
```

## Description

The `setTransform()` method sets the color transform information for `colorObject`. To use `setTransform()`, you first must create an object of type `Object` (for more information on the type `Object`, see “Object Class” on page 199) with a series of properties, and pass the object as the parameter to `setTransform()`. `setTransform()` uses the values as the new offsets and percentages of `colorObject`. The properties are the following:

- `ra` is the red transformation percentage (-100 to 100)
- `rb` is the red offset (-255 to 255)
- `ga` is the green transformation percentage (-100 to 100)
- `gb` is the green offset (-255 to 255)
- `ba` is the blue transformation percentage (-100 to 100)
- `bb` is the blue offset (-255 to 255)
- `aa` is the alpha transformation percentage (-100 to 100)
- `ab` is the alpha offset (-255 to 255)

The final value for each color is computed as:  $\text{value} = \text{original} * (\text{transformation percentage}) + \text{offset}$ .

## Parameters

*transformObj*      An object created using the constructor of the generic `Object` class whose properties specify color transformation percentages and color offsets.

## Example

```
redFish= new Color(_root.fish);
fishChanger = new Object();
fishChanger.ra = 100;//Red percentage
fishChanger.rb = 200;//Red offset
fishChanger.ga = 0;//Green percentage
fishChanger.gb = 0;//Green offset
fishChanger.ba = 100;//Blue percentage
fishChanger.bb = 50;//Blue offset
fishChanger.aa = 40;//Alpha percentage
fishChanger.ab = -10;//Alpha offset
redFish.setTransform(fishChanger);//sets the new transformation
values
```

## See also

“Color.getTransform() Method” on page 123

## Date() Global Function

`Date()`

## Description

The `Date()` global function returns a string containing the current date, the current time in the local time zone, and the offset in hours between Coordinated Universal Time (UTC—formerly called the Greenwich Mean Time, or GMT) and the local time. Do not confuse this global function with the `Date` object.

For example:

Mon Sep 10, 16:30:29 GMT-0700 2001

## Example

```
var now = Date();
trace(now); //prints string
```

# Date Object

## Description

The `Date` object allows you to get and set the local date and time or the Coordinated Universal Time (UTC—formerly called the Greenwich Mean Time, or GMT). To call the `Date` object's methods, you must create a new object using the constructor.

System-supplied dates and times are based on (and are as accurate as) the clock settings of the operating system upon which the Flash Player is running.

## Constructor

```
new Date()
new Date(ms)
new Date(year, month, date, hour, min, sec, ms)
```

## Description

You can create a `Date` object in three ways:

- With no arguments. This creates a new `Date` object holding the current date and time based on the local system clock. For example:

```
var now = new Date();
trace(now.getDate()); //prints the day of the month
```

- With one argument representing milliseconds. This creates a `Date` object holding the number of milliseconds relative to midnight January 1, 1970. For example:

```
var now = new Date(999901885456);
trace(now.getTime()); //prints 999901885456
```

- With three or more arguments. This creates a `Date` object indicating the year (required), month (required), day (required), hour, minute, second, and millisecond. To use an optional argument, all the arguments previous to it in the function call must be present.

```
var now = new Date(99, 11, 31, 9, 52, 54, 999);
trace(now.getFullYear()); //prints 1999
```

```

trace(now.getMonth()); //prints 11
trace(now.getDate()); //prints 31
trace(now.getHours()); //prints 9
trace(now.getMinutes()); //prints 52
trace(now.getSeconds()); //prints 54
trace(now.getMilliseconds()); //prints 999

```

## Parameters

ms	(Optional) An integer value representing the number of milliseconds since 1 January 1970 00:00:00.
year	The year expressed in four digits—for example, 2001. Alternatively, if you need to indicate a year from 1900 to 1999, specify a value from 0 to 99.
month	An integer value from 0 (Jan.) to 11 (Dec.).
date	An integer value from 1 to 31. If this argument is not supplied, its value is set to 0.
hour	(Optional) An integer value from 0 (midnight) to 23 (11 PM). If this argument is not supplied, its value is set to 0.
min	(Optional) An integer value from 0 to 59. If this argument is not supplied, its value is set to 0.
sec	(Optional) An integer value from 0 to 59. If this argument is not supplied, its value is set to 0.
ms	(Optional) An integer value from 0 to 999. If this argument is not supplied, its value is set to 0.

## Properties

None.

## Methods

getDate()	See “Date.getDate() Method” on page 129.	Return the day of the month.
getDay()	See “Date.getDay() Method” on page 130.	Return the day of the week.
getFullYear()	See “Date.getFullYear() Method” on page 130.	Return the year expressed in four-digit format.
getHours()	See “Date.getHours() Method” on page 130.	Return the hour.
getMilliseconds()	See “Date.getMilliseconds() Method” on page 131.	Return the milliseconds.
getMinutes()	See “Date.getMinutes() Method” on page 131.	Return the minutes.

<code>getMonth()</code>	See “Date.getMonth() Method” on page 131.	Return the month.
<code>getSeconds()</code>	See “Date.getSeconds() Method” on page 132.	Return the seconds.
<code>getTime()</code>	See “Date.getTime() Method” on page 132.	Return the number of milliseconds that have passed since January 1, 1970.
<code>getTimezoneOffset()</code>	See “Date.getTimezoneOffset() Method” on page 132.	Return the number of minutes between UTC and local time.
<code>getUTCDate()</code>	See “Date.getUTCDate() Method” on page 133.	Return the day of the month in UTC.
<code>getUTCDay()</code>	See “Date.getUTCDay() Method” on page 133.	Return the day of the week in UTC.
<code>getUTCFullYear()</code>	See “Date.getUTCFullYear() Method” on page 134.	Return the year as four-digits in UTC.
<code>getUTCHours()</code>	See “Date.getUTCHours() Method” on page 134.	Return the hour in UTC.
<code>getUTCMilliseconds()</code>	See “Date.getUTCMilliseconds() Method” on page 134.	Return the milliseconds in UTC.
<code>getUTCMinutes()</code>	See “Date.getUTCMinutes() Method” on page 135.	Return the minutes in UTC.
<code>getUTCMonth()</code>	See “Date.getUTCMonth() Method” on page 135.	Return the month in UTC.
<code>getUTCSeconds()</code>	See “Date.getUTCSeconds() Method” on page 135.	Return the seconds in UTC.
<code>getYear()</code>	See “Date.getYear() Method” on page 136.	Return the year relative to 1900.
<code>setDate()</code>	See “Date.setDate() Method” on page 136.	Set the day of the month.
<code>setFullYear()</code>	See “Date.setFullYear() Method” on page 137.	Set the year in four-digit format.
<code>setHours()</code>	See “Date.setHours() Method” on page 137.	Set the hour of the day.
<code>setMilliseconds()</code>	See “Date.setMilliseconds() Method” on page 138.	Set the milliseconds.
<code>setMinutes()</code>	See “Date.setMinutes() Method” on page 138.	Set the minutes.
<code>setMonth()</code>	See “Date.setMonth() Method” on page 139.	Set the month.
<code>setSeconds()</code>	See “Date.setSeconds() Method” on page 139.	Set the seconds.



<code>setTime()</code>	See “Date.setTime() Method” on page 140.	Set the date in number of milliseconds that have passed since January 1, 1970.
<code>setUTCDate()</code>	See “Date.setUTCDate() Method” on page 140.	Set the day of the month in UTC.
<code>setUTCFullYear()</code>	See “Date.setUTCFullYear() Method” on page 141.	Set the year in four-digit format in UTC.
<code>setUTCHours()</code>	See “Date.setUTCHours() Method” on page 141.	Set the hour in UTC.
<code>setUTCMilliseconds()</code>	See “Date.setUTCMilliseconds() Method” on page 142.	Set the milliseconds in UTC.
<code>setUTCMinutes()</code>	See “Date.setUTCMinutes() Method” on page 142.	Set the minutes in UTC.
<code>setUTCMonth()</code>	See “Date.setUTCMonth() Method” on page 143.	Set the month in UTC.
<code>setUTCSeconds()</code>	See “Date.setUTCSeconds() Method” on page 143.	Set the seconds in UTC.
<code>setYear()</code>	See “Date.setYear() Method” on page 144.	Set the year in four-digit format.
<code>toString()</code>	See “Date.toString() Method” on page 145.	Return the date and time values as a string.
<code>UTC()</code>	See “Date.UTC() Method” on page 145.	Return the number of milliseconds between January 1, 1970 in UTC and the time specified.
<code>valueOf()</code>	See “Date.valueOf() Method” on page 146.	Return the number of milliseconds that have passed since midnight, January 1, 1970 UTC. Equivalent to <code>getTime()</code> .

## Date.getDate() Method

`dateObj.getDate()`

### Description

The `getDate()` method returns the day of the month.

### Returns

An integer value from 1 to 31.

### Example

```
var now = new Date();
trace(now.getDate()); //prints the day of the month
```

**See also**

“Date.getUTCDate() Method” on page 133, “Date.setDate() Method” on page 136

## Date.getDay() Method

```
dateObj.getDay()
```

**Description**

The `getDay()` method returns the day of the week.

**Returns**

An integer value from 0 (Sunday) to 6 (Saturday).

**Example**

```
var now = new Date();
trace(now.getDay()); //prints the day of the week as an integer
```

**See also**

“Date.getUTCDay() Method” on page 133

## Date.getFullYear() Method

```
dateObj.getFullYear()
```

**Description**

The `getFullYear()` method returns the year expressed in four-digit format.

**Returns**

The year expressed in four digits—for example, 2001.

```
var now = new Date();
trace(now.getFullYear()); //prints the year in four digits
```

**See also**

“Date.getYear() Method” on page 136, “Date.getUTCFullYear() Method” on page 134,  
“Date.setFullYear() Method” on page 137

## Date.getHours() Method

```
dateObj.getHours()
```

**Description**

The `getHours()` method returns the hour of the day.

**Returns**

An integer value in the range of 0 (midnight) to 23 (11 PM).

**Example**

```
var now = new Date();
```

```
trace(now.getHours()); //prints the hour
```

**See also**

“Date.getUTCHours() Method” on page 134, “Date.setHours() Method” on page 137

## Date.getMilliseconds() Method

```
dateObj.getMilliseconds()
```

**Description**

The `getMilliseconds()` method returns the milliseconds.

**Returns**

An integer value from 0 to 999.

```
var now = new Date();
trace(now.getMilliseconds()); //prints the milliseconds
```

**See also**

“Date.getUTCMilliseconds() Method” on page 134, “Date.setMilliseconds() Method” on page 138

## Date.getMinutes() Method

```
dateObj.getMinutes()
```

**Description**

The `getMinutes()` method returns the minutes.

**Returns**

An integer value in the range 0 to 59.

**Example**

```
var now = new Date();
trace(now.getMinutes()); //prints the minutes
```

**See also**

“Date.getUTCMinutes() Method” on page 135, “Date.setMinutes() Method” on page 138

## Date.getMonth() Method

```
dateObj.getMonth()
```

**Description**

The `getMonth()` method returns the month.

**Returns**

An integer value from 0 (Jan.) to 11 (Dec.).

**Example**

```
var now = new Date();
trace(now.getMonth()); //prints the month as an integer
```

**See also**

“Date.getUTCMonth() Method” on page 135, “Date.setMonth() Method” on page 139

## Date.getSeconds() Method

```
dateObj.getSeconds()
```

**Description**

The `getSeconds()` method returns the seconds.

**Returns**

An integer value in the range of 0 to 59.

**Example**

```
var now = new Date();
trace(now.getSeconds()); //prints the seconds
```

**See also**

“Date.getUTCSeconds() Method” on page 135, “Date.setSeconds() Method” on page 139

## Date.getTime() Method

```
dateObj.getTime()
```

**Description**

The `getTime()` method returns the number of milliseconds that have passed since January 1, 1970.

**Returns**

An integer value representing milliseconds.

**Example**

```
var now = new Date();
trace(now.getTime()); //prints a very large integer
```

**See also**

“Date.setTime() Method” on page 140, “Date.setMilliseconds() Method” on page 138

## Date.getTimezoneOffset() Method

```
dateObj.getTimezoneOffset()
```

### Description

The `getTimezoneOffset()` method returns the number of minutes between UTC and local time. Accounts for daylight savings time.

### Returns

An integer value representing the number of minutes.

### Example

```
var now = new Date();
trace(now.getTimezoneOffset());
// for California, prints 420 (7 hours) if daylight savings;
// if not daylight savings, prints 480
```

## Date.getUTCDate() Method

`dateObj.getUTCDate()`

### Description

The `getUTCDate()` method returns the day of the month in UTC.

### Returns

An integer value from 1 to 31.

### Example

```
var now = new Date();
trace(now.getUTCDate()); //prints the day of the month
```

### See also

“Date.getDate() Method” on page 129, “Date.setUTCDate() Method” on page 140

## Date.getUTCDay() Method

`dateObj.getUTCDay()`

### Description

The `getUTCDay()` method returns the day of the week in UTC.

### Returns

An integer value from 0 (Sunday) to 6 (Saturday).

### Example

```
var now = new Date();
trace(now.getUTCDay()); //prints the day of the week as an integer
```

### See also

“Date.getDay() Method” on page 130

## Date.getUTCFullYear() Method

```
dateObj.getUTCFullYear()
```

### Description

The `getUTCFullYear()` method returns the year as four-digits in UTC.

### Returns

The year expressed in four digits—for example, 2001.

### Example

```
var now = new Date();
trace(now.getUTCFullYear()); //prints the year in four digits
```

### See also

“Date.getFullYear() Method” on page 130, “Date.setUTCFullYear() Method” on page 141

## Date.getUTCHours() Method

```
dateObj.getUTCHours()
```

### Description

The `getUTCHours()` method returns the hour in UTC.

### Returns

An integer value in the range of 0 (midnight) to 23 (11 PM).

### Example

```
var now = new Date();
trace(now.getUTCHours()); //prints the hour
```

### See also

“Date.getHours() Method” on page 130, “Date.setUTCHours() Method” on page 141

## Date.getUTCMilliseconds() Method

```
dateObj.getUTCMilliseconds()
```

### Description

The `getUTCMilliseconds()` method returns the milliseconds in UTC.

### Returns

An integer value from 0 to 999.

### Example

```
var now = new Date();
trace(now.getUTCMilliseconds()); //prints the milliseconds
```

**See also**

“Date.getMilliseconds() Method” on page 131, “Date.setUTCMilliseconds() Method” on page 142

## Date.getUTCMinutes() Method

```
dateObj.getUTCMinutes()
```

**Description**

The `getUTCMinutes()` method returns the minutes in UTC.

**Return**

An integer value in the range of 0 to 59.

**Example**

```
var now = new Date();
trace(now.getUTCMinutes()); //prints the minutes
```

**See also**

“Date.getMinutes() Method” on page 131, “Date.setUTCMinutes() Method” on page 142

## Date.getUTCMonth() Method

```
dateObj.getUTCMonth()
```

**Description**

The `getUTCMonth()` method returns the month in UTC.

**Returns**

An integer value from 0 (Jan.) to 11 (Dec.).

**Example**

```
var now = new Date();
trace(now.getUTCMonth()); //prints the month as an integer
```

**See also**

“Date.getMonth() Method” on page 131, “Date.setUTCMonth() Method” on page 143

## Date.getUTCSeconds() Method

```
dateObj.getUTCSeconds()
```

**Description**

The `getUTCSeconds()` method returns the seconds in UTC.

**Returns**

An integer value in the range of 0 to 59.

**Example**

```
var now = new Date();
trace(now.getUTCSeconds()); //prints the seconds
```

**See also**

“Date.getSeconds() Method” on page 132, “Date.setUTCSeconds() Method” on page 143

## Date.getYear() Method

```
dateObj.getYear()
```

**Description**

The `getYear()` method returns the year relative to 1900. For example, 101 is returned for the year 2001.

**Returns**

An integer value representing the number of years that have passed since 1900.

**Example**

```
var now = new Date();
trace(now.getYear()); //prints current year minus 1900
```

**See also**

“Date.getFullYear() Method” on page 130, “Date.getUTCFullYear() Method” on page 134, “Date.setYear() Method” on page 144

## Date.setDate() Method

```
dateObj.setDate(date)
```

**Description**

The `setDate()` method sets the day of the month of `dateObj`. This does not affect the system clock or anything else.

**Parameters**

*date*                      An integer value from 1 to 31 indicating the day of the month to set.

**Returns**

The number of milliseconds between the date set and midnight, January 1, 1970.

**Example**

```
var now = new Date();
trace(now.setDate(6)); //prints a very large integer
trace(now.getDate()); //prints 6
```



**See also**

“Date.getDate() Method” on page 129, “Date.setUTCDate() Method” on page 140

## Date.setFullYear() Method

```
dateObj.setFullYear(year, month, date)
```

**Description**

The `setFullYear()` method sets the year of `dateObj`. The method also sets `month` and `date` when these optional parameters are specified. This does not affect the system clock or anything else.

**Parameters**

<code>year</code>	A four-digit integer value indicating the year to set—for example, 2001.
<code>month</code>	<i>(Optional)</i> An integer value from 0 (Jan.) to 11 (Dec.) indicating the month of the year to set.
<code>date</code>	<i>(Optional)</i> An integer value from 1 to 31 indicating the day of the month to set.

**Returns**

The number of milliseconds between the date set and midnight, January 1, 1970.

**Example**

```
var now = new Date();
trace(now.setFullYear(2001)); //prints a very large integer
trace(now.getFullYear()); //prints 2001
trace(now.getMonth()); //prints month
trace(now.getDate()); //prints day of the month
```

**See also**

“Date.setUTCFullYear() Method” on page 141, “Date.setYear() Method” on page 144,  
“Date.getFullYear() Method” on page 130

## Date.setHours() Method

```
dateObj.setHours(hour)
```

**Description**

The `setHours()` method sets the hour of `dateObj`. This does not affect the system clock or anything else.

**Parameters**

<code>hour</code>	An integer value from 0 (midnight) to 23 (11 PM) indicating the hour of the day to set.
-------------------	-----------------------------------------------------------------------------------------

### Returns

The number of milliseconds between the date set and midnight, January 1, 1970.

### Example

```
var now = new Date();
trace(now.setHours(22)); //prints a very large integer
trace(now.getHours()); //prints 22
```

### See also

“Date.getHours() Method” on page 130, “Date.setUTCHours() Method” on page 141

## Date.setMilliseconds() Method

```
dateObj.setMilliseconds(ms)
```

### Description

The `setMilliseconds()` method sets the milliseconds of `dateObj`. This does not affect the system clock or anything else.

### Parameters

<i>ms</i>	An integer value from 0 to 999 indicating the milliseconds to set.
-----------	--------------------------------------------------------------------

### Returns

The number of milliseconds between the date set and midnight, January 1, 1970.

### Example

```
var now = new Date();
trace(now.setMilliseconds(847)); //prints a very large integer
trace(now.getMilliseconds()); //prints 847
```

### See also

“Date.getMilliseconds() Method” on page 131, “Date.setUTCMilliseconds() Method” on page 142

## Date.setMinutes() Method

```
dateObj.setMinutes(min)
```

### Description

The `setMinutes()` method sets the minutes of `dateObj`. This does not affect the system clock or anything else.

### Parameters

<i>min</i>	An integer value from 0 to 59 indicating the number of minutes to set.
------------	------------------------------------------------------------------------

### Returns

The number of milliseconds between the date set and midnight, January 1, 1970.

### Example

```
var now = new Date();
trace(now.setMinutes(59)); //prints a very large integer
trace(now.getMinutes()); //prints 59
```

### See also

“Date.getMinutes() Method” on page 131, “Date.setUTCMinutes() Method” on page 142

## Date.setMonth() Method

```
dateObj.setMonth(month)
```

### Description

The `setMonth()` method sets the month of `dateObj`. This does not affect the system clock or anything else.

### Parameters

<code>month</code>	An integer value from 0 (Jan.) to 11 (Dec.) indicating the month to set.
--------------------	--------------------------------------------------------------------------

### Returns

The number of milliseconds between the date set and midnight, January 1, 1970.

### Example

```
var now = new Date();
trace(now.setMonth(0)); //prints a very large integer
trace(now.getMonth()); //prints 0
```

### See also

“Date.getMonth() Method” on page 131, “Date.setUTCMonth() Method” on page 143

## Date.setSeconds() Method

```
dateObj.setSeconds(sec)
```

### Description

The `setSeconds()` method sets the seconds of `dateObj`. This does not affect the system clock or anything else.

### Parameters

<code>sec</code>	An integer value from 0 to 59 indicating the seconds to set.
------------------	--------------------------------------------------------------

### Returns

The number of milliseconds between the date set and midnight, January 1, 1970.

### Example

```
var now = new Date();
trace(now.setSeconds(59)); //prints a very large integer
trace(now.getSeconds()); //prints 59
```

### See also

“Date.getSeconds() Method” on page 132, “Date.setUTCSeconds() Method” on page 143

## Date.setTime() Method

```
dateObj.setTime(ms)
```

### Description

The `setTime()` method sets the date in number of milliseconds that have passed since January 1, 1970. This does not affect the system clock or anything else.

### Parameters

<code>ms</code>	An integer value indicating the number of milliseconds between the date to be set and midnight, January 1, 1970.
-----------------	------------------------------------------------------------------------------------------------------------------

### Returns

The number of milliseconds set.

### Example

```
var now = new Date();
trace(now.setTime(999930239559)); //prints a very large integer
trace(now.getTime()); //prints 999930239559
```

### See also

“Date.getTime() Method” on page 132

## Date.setUTCDate() Method

```
dateObj.setUTCDate(date)
```

### Description

The `setUTCDate()` method sets the date of the month in UTC of `dateObj`. This does not affect the system clock or anything else.

### Parameters

<code>date</code>	An integer value from 1 to 31 indicating the day to be set.
-------------------	-------------------------------------------------------------

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date();
trace(now.setUTCDate(2)); //prints a very large integer
trace(now.getUTCDate()); //prints 2
```

## See also

“Date.getUTCDate() Method” on page 133, “Date.setDate() Method” on page 136

# Date.setUTCFullYear() Method

```
dateObj.setUTCFullYear(year, month, date)
```

## Description

The `setUTCFullYear()` method sets the year in UTC of `dateObj`, and optionally sets the month and day of the month. This does not affect the system clock or anything else.

## Parameters

<i>year</i>	The year expressed in four digits—for example, 2001.
<i>month</i>	(Optional) An integer from 0 (Jan.) to 11 (Dec.).
<i>date</i>	(Optional) An integer value from 1 to 31.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date();
trace(now.setUTCFullYear(2001,3,1)); //prints a very large integer
trace(now.getUTCFullYear()); //prints 2001
trace(now.getUTCMonth()); //prints 3
trace(now.getUTCDate()); //prints 1
```

## See also

“Date.getUTCFullYear() Method” on page 134, “Date.setFullYear() Method” on page 137

# Date.setUTCHours() Method

```
dateObj.setUTCHours(hour)
```

## Description

The `setUTCHours()` method sets the hour of the day in UTC of `dateObj`. This does not affect the system clock or anything else.

## Parameters

**hour** An integer value from 0 (midnight) to 23 (11 PM) indicating the hour to be set.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date();
trace(now.setUTCHours(22)); //prints a very large integer
trace(now.getUTCHours()); //prints 22
```

## See also

“Date.getUTCHours() Method” on page 134, “Date.setHours() Method” on page 137

# Date.setUTCMilliseconds() Method

`dateObj.setUTCMilliseconds(ms)`

## Description

The `setUTCMilliseconds()` method sets the milliseconds in UTC of `dateObj`. This does not affect the system clock or anything else.

## Parameters

**ms** An integer value in the range of 0 to 999 indicating the number of milliseconds to set.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date();
trace(now.setUTCMilliseconds(220)); //prints a very large integer
trace(now.getUTCMilliseconds()); //prints 220
```

## See also

“Date.getUTCMilliseconds() Method” on page 134, “Date.setMilliseconds() Method” on page 138

# Date.setUTCMinutes() Method

`dateObj.setUTCMinutes(min)`

## Description

The `setUTCMinutes()` method sets the minutes in UTC of `dateObj`. This does not affect the system clock or anything else.

## Parameters

`min`                      An integer value in the range 0 to 59 indicating the number of minutes to be set.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date();
trace(now.setUTCMinutes(45)); //prints a very large integer
trace(now.getUTCMinutes()); //prints 45
```

## See also

"Date.getUTCMinutes() Method" on page 135, "Date.setMinutes() Method" on page 138

# Date.setUTCMonth() Method

`dateObj.setUTCMonth(month)`

## Description

The `setUTCMonth()` method sets the month in UTC of `dateObj`. This does not affect the system clock or anything else.

## Parameters

`month`                      An integer value in the range 0 (Jan.) to 11 (Dec.) indicating the month to set.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date();
trace(now.setUTCMonth(11)); //prints a very large integer
trace(now.getUTCMonth()); //prints 11
```

## See also

"Date.getUTCMonth() Method" on page 135, "Date.setMonth() Method" on page 139

# Date.setUTCSeconds() Method

`dateObj.setUTCSeconds(sec)`

## Description

The `setUTCSeconds()` sets the seconds in UTC of `dateObj`. This does not affect the system clock or anything else.

## Parameters

**sec** An integer value in the range 0 to 59 indicating the number of seconds to set.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date();
trace(now.setUTCSeconds(44)); //prints a very large integer
trace(now.getUTCSeconds()); //prints 44
```

## See also

"Date.getUTCSeconds() Method" on page 135, "Date.setSeconds() Method" on page 139

# Date.setYear() Method

`dateObj.setYear(year, month, date)`

## Description

The `setYear()` method sets the year of `dateObj`, and optionally the month and day of the month. This does not affect the system clock or anything else.

## Parameters

**year** An integer value indicating the year to set. The method interprets a 1- or 2-digit value to mean the 1900s—for example, 13 is interpreted to mean 1913.

**month** *(Optional)* An integer value in the range of 0 (Jan.) to 11 (Dec.) indicating the month to set.

**date** *(Optional)* An integer value in the range of 1 to 31 indicating the day to be set.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970.

## Example

```
var now = new Date();
trace(now.setYear(2001,3,1)); //prints a very large integer
trace(now.getFullYear()); //prints 2001
trace(now.getMonth()); //prints 3
trace(now.getDate()); //prints 1
```



**See also**

“Date.getYear() Method” on page 136, “Date.setFullYear() Method” on page 137,  
“Date.setUTCFullYear() Method” on page 141

## Date.toString() Method

```
dateObj.toString()
```

**Description**

The `toString()` method returns the date and time values as a string.

**Returns**

The following string is an example of the format returned by this method:

```
Mon Aug 13, 10:54:21 GMT-0700 2001
```

**Example**

```
var now = new Date();
trace(now.toString()); //string with the date
```

## Date.UTC() Method

```
Date.UTC(year, month, date, hour, min, sec, ms)
```

**Description**

The `Date.UTC()` method returns the date as the number of milliseconds between the time specified (passed in as the arguments to the method) and midnight, January 1, 1970, in UTC. The first three parameters are required. `Date.UTC()` and `Date()` accept the same arguments; the only difference between the two is that the new `Date` object created using `Date.UTC()` assumes UTC while the new `Date` object created using only `Date()` assumes local time. A new UTC date object is normally created like this:

```
now = new Date(Date.UTC(2001, 9, 30));
```

In addition, `Date.UTC()` is commonly used with the `setTime()` method to set a UTC date.

**Parameters**

<i>year</i>	The year expressed in four digits— for example, 2001. To indicate for a year from 1900 to 1999, you can specify a value from 0 to 99.
<i>month</i>	An integer value from 0 (Jan.) to 11 (Dec.).
<i>date</i>	An integer value from 1 to 31.
<i>hour</i>	(Optional) An integer value in the range of 0 (midnight) to 23 (11 PM).
<i>min</i>	(Optional) An integer value in the range of 0 to 59.
<i>sec</i>	(Optional) An integer value in the range of 0 to 59.
<i>ms</i>	(Optional) An integer value in the range of 0 to 999.

## Returns

The number of milliseconds between the date set and midnight, January 1, 1970, in UTC.

## Example

```
var now = new Date(Date.UTC(96, 11, 29, 11, 58, 59, 345));
trace(now.getTime()); //prints milliseconds
trace(now.getUTCFullYear()); //prints 1996
trace(now.getMonth()); //prints 11
trace(now.getUTCDate()); //prints 29
trace(now.getUTCHours()); //prints 11
trace(now.getUTCMinutes()); //prints 58
trace(now.getUTCSeconds()); //prints 59
trace(now.getUTCMilliseconds()); //prints 345
```

## See also

“Date.setTime() Method” on page 140

# Date.valueOf() Method

`dateObj.valueOf()`

## Description

The `valueOf()` method returns the number of milliseconds that have passed since midnight, January 1, 1970 UTC. Equivalent to `getTime()`.

## Returns

An integer value representing milliseconds.

## Example

```
var now = new Date();
trace(now.valueOf()); //prints the number of milliseconds
```

## See also

“Date.getTime() Method” on page 132

# duplicateMovieClip() Global Function

`duplicateMovieClip(target, newName, depth)`

## Description

The `duplicateMovieClip()` global function creates a duplicate of `target` while `target` is playing. The duplicate movie clip always starts at its frame 1 regardless of `target`'s frame at the time of duplication. The duplicate movie clip inherits shape transformations but not the current values of `target`'s user-defined variables. The duplicate movie clip is placed in `target`'s parent's programmatic stack. A programmatic stack holds child movie clips; when you duplicate a movie clip, the duplicate has the same parent as the original, and thus resides in the parent's programmatic stack.

The `removeMovieClip()` global function is used to delete duplicate movie clips. `movieClip.removeMovieClip()` can also be used by duplicate movie clips to delete themselves. Duplicate movie clips can also be removed by placing another movie clip at the same depth in the programmatic stack.

### Parameters

<code>target</code>	A path or reference to the movie clip that is duplicated.
<code>newName</code>	A string specifying the name of the duplicate movie clip. This must be a unique name.
<code>depth</code>	The depth of the movie clip in <code>target</code> 's parent's programmatic stack.

### Example

```
duplicateMovieClip (_root.baseball, "newBaseball", 1); //creates new
baseball
_root.newBaseball._x += 25; //moves new baseball along x axis
_root.newBaseball._y += 25; //moves new baseball along y axis
```

### See also

"`removeMovieClip()` Global Function" on page 204, "`MovieClip.duplicateMovieClip()` Method" on page 179, "`MovieClip.removeMovieClip()` Method" on page 188

## escape() Global Function

`escape(string)`

### Description

The `escape()` global function creates an encoded string from `string`. In the new string, characters of `string` that require encoding are replaced with the format `%xx`, where `xx` is the hexadecimal value of the character. The encoding is basically URL encoding, except that spaces are replaced with `%20` instead of a `+` sign. Use the `unescape()` global function to translate the string back into its original format.

### Parameters

<code>string</code>	The string to be encoded.
---------------------	---------------------------

### Example

```
//prints Billy%20went%20fishing%21%24%23%21
trace(escape("Billy went fishing!$#!"));
```

### See also

"`unescape()` Global Function" on page 225

## eval() Global Function

`eval(expression)`

### Description

The `eval()` global function returns the value of, or a reference to, `expression`.

**Note:** This implementation of `eval()` is different from the traditional JavaScript implementation.

### Parameters

<code>expression</code>	An expression that evaluates to a variable, property, object, movie clip, or function.
-------------------------	----------------------------------------------------------------------------------------

### Returns

If `expression` is a variable or property, the value of the variable or property is returned. If `expression` is an object, movie clip, or function, a reference to the item is returned.

### Example

```
x=4;
trace(eval(x)); //prints 4
str = "baseball";
hitBaseball = eval("_root."+ str);
hitBaseball._x += 50; //moves movie clip 50 pixels along x axis

trace(eval(this._x)); //returns _x property for "this" reference
```

## \_focusrect Global Property

`_focusrect`

### Description

The `_focusrect` global property is a boolean value that specifies whether a button with the over state defined and that currently has keyboard focus is displayed with a yellow border. Keyboard focus is obtained using the Tab key. As a boolean, it can be assigned only one of two values: `true` or `false`. If assigned `true`, the yellow border appears; if `false`, it does not. The default value is `true`. This property can be read or written.

## fscommand() Global Function

`getURL("fscommand:command", argument)`

### Description

The `fscommand` global function is used only within the context of `getURL()`. See `getURL()` for details. In LiveMotion, `fscommand` communication is only supported for use with the standalone Flash Player.

## Parameters

<code>command</code>	The command to execute.
<code>argument</code>	The argument for the command.

## See also

“`getURL()` Global Function” on page 149

## `getTimer()` Global Function

`getTimer()`

### Description

The `getTimer()` global function gets the number of milliseconds that have elapsed since the SWF started playing.

### Returns

The elapsed time in milliseconds.

## `getURL()` Global Function

`getURL(url)`

`getURL(url, window)`

`getURL(url, window, howToSendVariables)`

### Description

The `getURL()` global function gets a document from a specified URL and loads it into the Web browser in the specified *window*. It is also used to execute a script on a server and receive the results in a Web browser window or frame. Additionally, it can be used to execute JavaScript code (`"javascript:command"`) or VBScript code (`"vbscript:command"`) in a Web browser, and it provides support for the `fscommand` global function. The `file`, `ftp`, `http`, and `print` protocols are supported.

**Note:** This method is not supported in Preview mode.

### Parameters

<code>url</code>	A string specifying the URL to which to hyperlink (HTTP or FTP). This may be a relative or an absolute pathname. It can be the name of a document or it can be a script, and the <code>fscommand</code> global function can be used here.
------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

window	(Optional) The target frame in the browser—e.g., <code>_self</code> (the default), <code>_parent</code> , <code>_top</code> , <code>_blank</code> . If omitted, <code>_self</code> is used. Custom names can also be used.
howToSendVariables	(Optional) Omit this parameter if you don't want to send variables. This parameter is a string literal. Specify <code>GET</code> to send variables via get (i.e., tacked onto the end of the URL) or <code>POST</code> to send them with post (i.e., put into the body of the request). Both methods send them in application/x-www-form-urlencoded MIME format. All user-defined variables are sent.

The `fscommand` options are as follows:

- `getURL("fscommand: allowscale", value)`—Tells the standalone Flash Player whether its contents should scale with the size of the player's window. `value` is the string `true` or `false`, indicating whether or not (respectively) the contents of the Flash Player should scale.
- `getURL("fscommand: exec", applicationName)`—Tells the standalone Flash Player to launch an external application. `applicationName` is a string showing an absolute path to the application.
- `getURL("fscommand: fullscreen", value)`—Tells the standalone Flash Player whether to maximize, filling the entire screen. `value` is the string `true` or `false`, indicating whether or not (respectively) to maximize.
- `getURL("fscommand: quit")`—Tells the standalone Flash Player to quit.
- `getURL("fscommand: showmenu", value)`—Tells the standalone Flash Player whether to suppress the display of the controls in the context menu. `value` is the string `true` or `false`, indicating whether or not (respectively) to suppress.
- `getURL("fscommand: trapallkeys", value)`—Tells the standalone Flash Player whether to send all keystrokes to the SWF file(s) executing in the Flash Player. `value` is the string `true` or `false`, indicating whether or not (respectively) to send.

### Example

```
getURL("ftp://download.intel.com");
getURL("http://www.adobe.com", "_parent");
getURL("file:///C:/coolestFile.html");
getURL("javascript: alert(\"Hi\");");
```

### See also

"loadVariables() Global Function" on page 163, "MovieClip.getURL() Method" on page 182, "MovieClip.loadVariables() Method" on page 186, "fscommand() Global Function" on page 148

## getVersion() Global Function

```
getVersion()
```

## Description

The `getVersion()` global function returns, in string form, the version of the Flash Player that the user currently has installed. The first number refers to the major version number of the Flash Player; the second number gives the minor version; the third number is the build (revision); and the fourth number is the patch.

For example, from LiveMotion's Preview mode:

```
LM 5,0,42,0
```

For example, from an exported SWF file (on a Windows machine):

```
WIN 5,0,30,0
```

## Returns

The version of the Flash Player installed on the user's system.

## gotoAndPlay() Global Function

```
gotoAndPlay(label)
```

### Description

The `gotoAndPlay()` global function sends the current timeline's playhead to the specified *label* and continues playing from *label*.

**Note:** *Frame numbers should not be passed to this global function. The use of labels is recommended.*

### Parameters

label	A string indicating the destination of the playhead.
-------	------------------------------------------------------

### See also

"gotoAndStop() Global Function" on page 151, "MovieClip.gotoAndPlay() Method" on page 183

## gotoAndStop() Global Function

```
gotoAndStop(label)
```

### Description

The `gotoAndStop()` global function sends the current timeline's playhead to the specified *label* and stops playing.

**Note:** *Frame numbers should not be passed to this global function. The use of labels is recommended.*

### Parameters

label	A string indicating the destination of the playhead.
-------	------------------------------------------------------

**See also**

“gotoAndPlay() Global Function” on page 151, “MovieClip.gotoAndStop() Method” on page 183

## Infinity Global Property

`Infinity`

**Description**

The `Infinity` global property is a predefined variable with the value for infinity. It is any value larger than `Number.MAX_VALUE`, which is the largest number that can be represented in JavaScript. This property can only be read.

**See also**

“-Infinity Global Property” on page 152, “Number.POSITIVE\_INFINITY Property” on page 198, “Number.MAX\_VALUE Property” on page 196

## -Infinity Global Property

`-Infinity`

**Description**

The `-Infinity` global property is a predefined variable with the value of -infinity. This property can only be read.

**See also**

“Infinity Global Property” on page 152, “Number.NEGATIVE\_INFINITY Property” on page 197

## isFinite Global Function

`isFinite(expression)`

**Description**

The `isFinite()` global function evaluates an expression and returns `true` if the expression is a finite number. Otherwise, it returns `false`—the value is infinity or negative infinity.

**Parameters**

<i>expression</i>	Any valid JavaScript expression.
-------------------	----------------------------------

**Returns**

`true` if the expression is a finite number, `false` otherwise.

**See also**

“Infinity Global Property” on page 152, “-Infinity Global Property” on page 152



## isNaN() Global Function

`isNaN(expression)`

### Description

The `isNaN()` global function returns `true` if the expression is Not-a-Number (NaN).

### Parameters

*expression* Any valid JavaScript expression.

### Returns

`true` if the expression is not a number (NaN), `false` otherwise.

### See also

“Number.NaN Property” on page 197

## Key Object

### Description

The `Key` object is used to retrieve the state of the keyboard. The `Key` object and its constants and methods are static—you do not create `Key` objects using a constructor.

### Constants

BACKSPACE	See “Key.BACKSPACE Constant” on page 154.	<code>Key.BACKSPACE</code> constant represents the key code for the BACKSPACE key.
CAPSLOCK	See “Key.CAPSLOCK Constant” on page 155.	<code>Key.CAPSLOCK</code> constant represents the key code for the CAPSLOCK key.
CONTROL	See “Key.CONTROL Constant” on page 155.	<code>Key.CONTROL</code> constant represents the key code for the CONTROL key.
DELETEKEY	See “Key.DELETEKEY Constant” on page 155.	<code>Key.DELETEKEY</code> constant represents the key code for the DELETEKEY key.
DOWN	See “Key.DOWN Constant” on page 155.	<code>Key.DOWN</code> constant represents the key code for the DOWN key.
END	See “Key.END Constant” on page 156.	<code>Key.END</code> constant represents the key code for the END key.
ENTER	See “Key.ENTER Constant” on page 156.	<code>Key.ENTER</code> constant represents the key code for the ENTER key.
ESCAPE	See “Key.ESCAPE Constant” on page 156.	<code>Key.ESCAPE</code> constant represents the key code for the ESCAPE key.
HOME	See “Key.HOME Constant” on page 157.	<code>Key.HOME</code> constant represents the key code for the HOME key.

INSERT	See “Key.INSERT Constant” on page 157.	<code>Key.INSERT</code> constant represents the key code for the <code>INSERT</code> key.
LEFT	See “Key.LEFT Constant” on page 159.	<code>Key.LEFT</code> constant represents the key code for the <code>LEFT</code> key.
PGDN	See “Key.PGDN Constant” on page 159.	<code>Key.PGDN</code> constant represents the key code for the <code>PGDN</code> key.
PGUP	See “Key.PGUP Constant” on page 159.	<code>Key.PGUP</code> constant represents the key code for the <code>PGUP</code> key.
RIGHT	See “Key.RIGHT Constant” on page 159.	<code>Key.RIGHT</code> constant represents the key code for the <code>RIGHT</code> key.
SHIFT	See “Key.SHIFT Constant” on page 160.	<code>Key.SHIFT</code> constant represents the key code for the <code>SHIFT</code> key.
SPACE	See “Key.SPACE Constant” on page 160.	<code>Key.SPACE</code> constant represents the key code for the <code>SPACE</code> key.
TAB	See “Key.TAB Constant” on page 160.	<code>Key.TAB</code> constant represents the key code for the <code>TAB</code> key.
UP	See “Key.UP Constant” on page 160.	<code>Key.UP</code> constant represents the key code for the <code>UP</code> key.

## Methods

<code>getAscii()</code>	See “Key.getAscii() Method” on page 156.	Get the ASCII code of the last key pressed.
<code>getCode()</code>	See “Key.getCode() Method” on page 157.	Get the key code of the last key pressed.
<code>isDown()</code>	See “Key.isDown() Method” on page 158.	Check whether the specified key is currently down.
<code>isToggled()</code>	See “Key.isToggled() Method” on page 158.	Check whether the Num lock, Caps lock, or Scroll lock key is toggled on.

## Key.BACKSPACE Constant

`Key.BACKSPACE`

### Description

The `Key.BACKSPACE` constant represents the key code for the `BACKSPACE` key. It is passed to `Key.isDown()` to determine whether the `BACKSPACE` key is pressed. It is returned by `Key.getCode()` if the `BACKSPACE` key was last key pressed.

### See also

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.CAPSLOCK Constant

`Key.CAPSLOCK`

### Description

The `Key.CAPSLOCK` constant represents the key code for the `CAPSLOCK` key. It is passed to `Key.isToggled` to determine whether the `CAPSLOCK` key is on. It is returned by `Key.getCode()` if `CAPSLOCK` key was last key pressed.

### See also

“`Key.isToggled()` Method” on page 158, “`Key.getCode()` Method” on page 157

## Key.CONTROL Constant

`Key.CONTROL`

### Description

The `Key.CONTROL` constant represents the key code for the `CONTROL` key. It is passed to `Key.isDown()` to determine whether the `CONTROL` key is pressed. It is returned by `Key.getCode()` if `CONTROL` key was last key pressed.

### See also

“`Key.getCode()` Method” on page 157, “`Key.isDown()` Method” on page 158

## Key.DELETEKEY Constant

`Key.DELETEKEY`

### Description

The `Key.DELETEKEY` constant represents the key code for the `DELETEKEY` key. It is passed to `Key.isDown()` to determine whether the `DELETEKEY` key is pressed. It is returned by `Key.getCode()` if the `DELETEKEY` key was last key pressed.

### See also

“`Key.getCode()` Method” on page 157, “`Key.isDown()` Method” on page 158

## Key.DOWN Constant

`Key.DOWN`

### Description

The `Key.DOWN` constant represents the key code for the `DOWN` key. It is passed to `Key.isDown()` to determine whether the `DOWN` key is pressed. It is returned by `Key.getCode()` if the `DOWN` key was last key pressed.

### See also

“`Key.getCode()` Method” on page 157, “`Key.isDown()` Method” on page 158

## Key.END Constant

`Key.END`

### Description

The `Key.END` constant represents the key code for the `END` key. It is passed to `Key.isDown()` to determine whether the `END` key is pressed. It is returned by `Key.getCode()` if the `END` key was last key pressed.

### See also

"`Key.getCode()` Method" on page 157, "`Key.isDown()` Method" on page 158

## Key.ENTER Constant

`Key.ENTER`

### Description

The `Key.ENTER` constant represents the key code for the `ENTER` key. It is passed to `Key.isDown()` to determine whether the `ENTER` key is pressed. It is returned by `Key.getCode()` if the `ENTER` key was last key pressed.

### See also

"`Key.getCode()` Method" on page 157, "`Key.isDown()` Method" on page 158

## Key.ESCAPE Constant

`Key.ESCAPE`

### Description

The `Key.ESCAPE` constant represents the key code for the `ESCAPE` key. It is passed to `Key.isDown()` to determine whether the `ESCAPE` key is pressed. It is returned by `Key.getCode()` if the `ESCAPE` key was last key pressed.

### See also

"`Key.getCode()` Method" on page 157, "`Key.isDown()` Method" on page 158

## Key.getAscii() Method

`Key.getAscii()`

### Description

The `Key.getAscii()` method returns the ASCII code of the last key pressed.

### Example

In the `onKeyUp` or `onKeyDown` event:

```
var asciiVal = Key.getAscii();
if (asciiVal == 102)
{
 trace("Lower case 'f' has been pressed");
}
```

```
//your code
}
```

**See also**

“Key.getCode() Method” on page 157

## Key.getCode() Method

`Key.getCode()`

**Description**

The `Key.getCode()` method returns the key code of the last key pressed.

**Example**

In the `onKeyUp` or `onKeyDown` event:

```
if (Key.getCode() == Key.ESCAPE)
{
 trace("Key.ESCAPE was pressed.");
 //your code
}
```

**See also**

“Key.getAscii() Method” on page 156

## Key.HOME Constant

`Key.HOME`

**Description**

The `Key.HOME` constant represents the key code for the `HOME` key. It is passed to `Key.isDown()` to determine whether the `HOME` key is pressed. It is returned by `Key.getCode()` if the `HOME` key was last key pressed.

**See also**

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.INSERT Constant

`Key.INSERT`

**Description**

The `Key.INSERT` constant represents the key code for the `INSERT` key. It is passed to `Key.isDown()` to determine whether `INSERT` key is pressed. It is returned by `Key.getCode()` if the `INSERT` key was last key pressed.

**See also**

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.isDown() Method

`Key.isDown(keycode)`

### Description

The `Key.isDown()` method is used to check whether the specified key is currently down.

### Parameters

*keycode*                      The key code to check for.

### Returns

`true` if the key is pressed; `false` otherwise.

### Example

In the `onKeyUp` or `onKeyDown` event:

```
if (Key.isDown(key.RIGHT))
{
 trace("Right arrow key was pressed.");
 //your code
}
```

### See also

"`Key.isToggled()` Method" on page 158

## Key.isToggled() Method

`Key.isToggled(keycode)`

### Description

The `Key.isToggled()` method is used to see if the Caps lock, Num lock, or Scroll lock key is on.

### Parameters

*keycode*                      If this parameter is `Key.CAPSLock` or the integer 20, then the method checks for whether the Caps lock key is toggled on. If the parameter is the integer 144, then the method checks for whether the Num lock key is toggled on. If the parameter is the integer 145, then the method checks for whether the Scroll lock key is toggled on.

### Returns

`true` if the Num lock, Caps lock, or Scroll lock key is toggled on; `false` otherwise.

### Example

In the `onKeyUp` or `onKeyDown` event:

```
if (Key.isToggled(20))//detect whether Caps lock key is toggled on
```

```
{
 trace("Caps lock key is on.");
 //your code
}
```

**See also**

“Key.isDown() Method” on page 158

## Key.LEFT Constant

Key.LEFT

**Description**

The `Key.LEFT` constant represents the key code for the `LEFT` key. It is passed to `Key.isDown()` to determine whether the `LEFT` key is pressed. It is returned by `Key.getCode()` if the `LEFT` key was last key pressed.

**See also**

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.PGDN Constant

Key.PGDN

**Description**

The `Key.PGDN` constant represents the key code for the `PGDN` key. It is passed to `Key.isDown()` to determine whether the `PGDN` key is pressed. It is returned by `Key.getCode()` if the `PGDN` key was last key pressed.

**See also**

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.PGUP Constant

Key.PGUP

**Description**

The `Key.PGUP` constant represents the key code for the `PGUP` key. It is passed to `Key.isDown()` to determine whether the `PGUP` key is pressed. It is returned by `Key.getCode()` if the `PGUP` key was last key pressed.

**See also**

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.RIGHT Constant

Key.RIGHT

### Description

The `Key.RIGHT` constant represents the key code for the `RIGHT` key. It is passed to `Key.isDown()` to determine whether the `RIGHT` key is pressed. It is returned by `Key.getCode()` if the `RIGHT` key was last key pressed.

### See also

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.SHIFT Constant

`Key.SHIFT`

### Description

The `Key.SHIFT` constant represents the key code for the `SHIFT` key. It is passed to `Key.isDown()` to determine whether the `SHIFT` key is pressed. It is returned by `Key.getCode()` if the `SHIFT` key was last key pressed.

### See also

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.SPACE Constant

`Key.SPACE`

### Description

The `Key.SPACE` constant represents the key code for the `SPACE` key. It is passed to `Key.isDown()` to determine whether the `SPACE` key is pressed. It is returned by `Key.getCode()` if the `SPACE` key was last key pressed.

### See also

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.TAB Constant

`Key.TAB`

### Description

The `Key.TAB` constant represents the key code for the `TAB` key. It is passed to `Key.isDown()` to determine whether the `TAB` key is pressed. It is returned by `Key.getCode()` if the `TAB` key was last key pressed.

### See also

“Key.getCode() Method” on page 157, “Key.isDown() Method” on page 158

## Key.UP Constant

`Key.UP`



## Description

The `Key.UP` constant represents the key code for the UP key. It is passed to `Key.isDown()` to determine whether the UP key is pressed. It is returned by `Key.getCode()` if the UP key was last key pressed.

## See also

"Key.getCode() Method" on page 157, "Key.isDown() Method" on page 158

# \_leveln Global Property

`_leveln`

## Description

The `_leveln` global property is used to explicitly refer to the levels of the Flash Player and it is used to access the contents of those levels. It is used to specify the level into which to load a SWF file using the `loadMovie()` global function or to load variables using the `loadVariables()` global function and it is used to refer to that level after loading. The `_root` level movie clip loads at level 0 by default. This property can only be read.

**Note:** This global property is not supported in Preview mode (except for `_level0`).

## Example

```
loadMovie("http://devtech.corp.adobe.com/livemotion/billys.swf",
"_level1");
_level1.stop();
```

## See also

"loadMovie() Global Function" on page 162, "loadVariables() Global Function" on page 163

# lmFrameOfLabel() Global Function

`lmFrameOfLabel(label)`

## Description

The `lmFrameOfLabel()` global function returns the frame number at which `label` resides.

## Parameters

<i>label</i>	A string identifying the label on the composition ( <code>_root</code> 's) timeline.
--------------	--------------------------------------------------------------------------------------

## Returns

The frame number associated with `label`, or 0 if `label` is not found on the composition timeline.

## Example

```
//returns frame number of "firstThrow" label
lmFrameOfLabel("firstThrow");
```

## loadMovie() Global Function

```
loadMovie(url, target)
```

```
loadMovie(url, target, howToSendVariables)
```

### Description

The `loadMovie()` global function loads additional SWF files into the Flash Player. These SWF files can be loaded into Flash Player levels, or they can be loaded into existing movie clips. A movie clip can replace itself, even if it is at `_level0`.

If a new main movie clip is loaded at level 0, every level is unloaded and the effect is the same as starting a new SWF file in the Flash Player. The movie clip loaded in level 0 sets the frame rate, background color, and frame size for all other loaded movie clips.

*Note: `_root` does not always refer to `_level0`. It refers to the root of the current level where the reference is being made. For instance, if a movie clip in `_level2` references `_root`, it is the same as referencing `_level2`.*

Movie clips loaded with the `loadMovie()` global function can be unloaded using the `unloadMovie()` global function or the `unloadMovieNum()` global function. Likewise, a new movie clip can be loaded into an existing movie clip using the `loadMovie()` or `loadMovieNum()` global function.

When a SWF file is loaded into an existing movie clip, the `onData` event handler is called. Even though the contents of the movie clip are replaced, the movie clip handlers are not. These include `onEnterFrame`, `onLoad`, `onUnload`, `onData`, `onMouseDown`, `onMouseUp`, `onMouseMove`, `onKeyDown`, and `onKeyUp`. Everything else—including button handlers, state scripts, and objects—are replaced. This movie clip “shell” concept is important to keep in mind because it means that, when using `loadMovie()` and `unloadMovie()`, a movie clip instance is never really removed from the composition. Movie clip content is simply moved in and out of the shell.

**Note:** This method is not supported in Preview mode.

### Parameters

<i>url</i>	A string specifying the URL from which to load the SWF file.
<i>target</i>	A path or a reference to another movie clip that the new SWF file will replace, or the player level. The loaded movie clip inherits the position, scaling, and rotation of the movie clip it's replacing.
<i>howToSendVariables</i>	(Optional) Omit this parameter if you don't want to send variables. This parameter is a string literal. Specify <code>GET</code> to send variables via get (i.e., tacked onto the end of the URL) or <code>POST</code> to send them with post (i.e., put into the body of the request). Both methods send them in application/x-www-form-urlencoded MIME format. All user-defined variables are sent.

### Example

```
loadMovie("http://devtech.corp.adobe.com/docs/livemotion/billys.swf", "_level1");
loadMovie("file:///C:/coolestMovie.swf", "_level1");
```

**See also**

“loadMovieNum() Global Function” on page 163, “unloadMovie() Global Function” on page 225, “unloadMovieNum() Global Function” on page 226, “MovieClip.loadMovie() Method” on page 185

## loadMovieNum() Global Function

```
loadMovieNum(url, level)
```

```
loadMovieNum(url, level, howToSendVariables)
```

**Description**

The `loadMovieNum()` global function is the same as `loadMovie()` except that the second parameter must be specified as a number. With `loadMovieNum()` you cannot specify the name of another movie clip to be replaced.

**Note:** This method is not supported in Preview mode.

**Parameters**

<i>url</i>	A string specifying the URL from which to load the SWF file.
<i>level</i>	The player level number into which to load the SWF file. Must be a non-negative integer.
<i>howToSendVariables</i>	(Optional) A string literal. GET or POST.

**See also**

“loadMovie() Global Function” on page 162, “unloadMovie() Global Function” on page 225, “unloadMovieNum() Global Function” on page 226, “MovieClip.loadMovie() Method” on page 185

## loadVariables() Global Function

```
loadVariables(url, target)
```

```
loadVariables(url, target, howToSendVariables)
```

**Description**

The `loadVariables()` global function loads variables fetched from the specified URL into *target*. The movie clip's `onData` event handler is called when the variables have been loaded. The data that's loaded is scoped to the movie clip/level into which it's loaded. All the values loaded are considered the string data type. If a variable to be loaded is not already declared within *target*, then it is added as a new property of *target* and can be accessed using the standard *target.property* syntax or handled in the same way as any other variable.

The data fetched from the URL must be in the application/x-www-form-urlencoded MIME format.

**Note:** Variables cannot be loaded from a local file in Preview mode. However, HTTP requests for external data can be made.

## Parameters

<i>url</i>	A string specifying the URL from which to get the variables. For security reasons, the URL must be in the same domain as that from which the movie clip was downloaded.
<i>target</i>	A path or reference to an existing movie clip or player level in which the loaded variables are defined.
<i>howToSendVariables</i>	(Optional) Omit this parameter if you don't want to send variables. If omitted, variables are retrieved but none are sent. This parameter is a string literal. Specify <code>GET</code> to send variables via get (i.e., tacked onto the end of the URL) or <code>POST</code> to send them with post (i.e., put into the body of the request). Both methods send them in application/x-www-form-urlencoded MIME format. All user-defined variables are sent.

## Example

```
loadVariables("http://www.myServer.com/cgi-bin/stockdata.pl",this,"GET");
```

## See also

“loadVariablesNum() Global Function” on page 164, “getURL() Global Function” on page 149, “MovieClip.getURL() Method” on page 182, “MovieClip.loadVariables() Method” on page 186

# loadVariablesNum() Global Function

```
loadVariablesNum (url, level)
```

```
loadVariablesNum (url, level, howToSendVariables)
```

## Description

The `loadVariablesNum()` global function is the same as `loadVariables()` except the second argument must be a player level number.

## Parameters

<i>url</i>	A string specifying the URL from which to get the variables.
<i>level</i>	The player level number in which the loaded variables are defined. Must be a non-negative integer.
<i>howToSendVariables</i>	(Optional) A string literal. GET or POST.

## See also

“loadVariables() Global Function” on page 163, “getURL() Global Function” on page 149, “loadMovie() Global Function” on page 162, “loadMovieNum() Global Function” on page 163

# Math Object

## Description

The `Math` object has constants and methods to facilitate use of common mathematical functions and values. The `Math` object and its constants and methods are static—you do not create `Math` objects using a constructor. For example, you refer to the constant `PI` as `Math.PI` and you call the sine function as `Math.sin(x)`, where `x` is the method's argument. Constants are defined with the full precision of real numbers.

## Constants

<code>E</code>	See "Math.E Constant" on page 169.	Euler's constant and the base of natural logarithms (approximately 2.718).
<code>LN2</code>	See "Math.LN2 Constant" on page 169.	Natural logarithm of 2 (approximately 0.693).
<code>LN10</code>	See "Math.LN10 Constant" on page 169.	Natural logarithm of 10 (approximately 2.302).
<code>LOG2E</code>	See "Math.LOG2E Constant" on page 170.	NBase 2 logarithm of E (approximately 1.442).
<code>LOG10E</code>	See "Math.LOG10E Constant" on page 170.	Base 10 logarithm of E (approximately 0.434).
<code>PI</code>	See "Math.PI Constant" on page 171.	Ratio of the circumference of a circle to its diameter (approximately 3.14159).
<code>SQRT1_2</code>	See "Math.SQRT1_2 Constant" on page 172.	Square root of 1/2; equivalently, 1 over the square root of 2 (approximately 0.707).
<code>SQRT2</code>	See "Math.SQRT2 Constant" on page 173.	Square root of 2 (approximately 1.414).

## Methods

<code>abs()</code>	See "Math.abs() Method" on page 166.	Return the absolute value of a number.
<code>acos()</code>	See "Math.acos() Method" on page 166.	Return the arccosine (in radians) of a number.
<code>asin()</code>	See "Math.asin() Method" on page 167.	Return the arcsine (in radians) of a number.
<code>atan()</code>	See "Math.atan() Method" on page 167.	Return the arctangent (in radians) of a number.
<code>atan2()</code>	See "Math.atan2() Method" on page 167.	Return the arctangent (in radians) of the quotient of the arguments (y/x).
<code>ceil()</code>	See "Math.ceil() Method" on page 168.	Return the value rounded up.

<code>cos()</code>	See “Math.cos() Method” on page 168.	Return the cosine of an angle provided in radians.
<code>exp()</code>	See “Math.exp() Method” on page 169.	Return <code>Math.E</code> raised to the power of a number.
<code>floor()</code>	See “Math.floor() Method” on page 169.	Return the value rounded down.
<code>log()</code>	See “Math.log() Method” on page 170.	Return the natural logarithm of a number.
<code>max()</code>	See “Math.max() Method” on page 170.	Return the maximum of two numbers.
<code>min()</code>	See “Math.min() Method” on page 171.	Return the minimum of two numbers.
<code>pow()</code>	See “Math.pow() Method” on page 171.	Return $X^Y$ .
<code>random()</code>	See “Math.random() Method” on page 171.	Return a pseudo-random number from 0.0 up to but not including 1.0.
<code>round()</code>	See “Math.round() Method” on page 172.	Return the value of a number rounded to the nearest integer.
<code>sin()</code>	See “Math.sin() Method” on page 172.	Return the sine of an angle provided in radians.
<code>sqrt()</code>	See “Math.sqrt() Method” on page 172.	Return the square root of a number.
<code>tan()</code>	See “Math.tan() Method” on page 173.	Return the tangent of an angle provided in radians.

## Math.abs() Method

`Math.abs(x)`

### Description

The `abs()` method returns the absolute value of a number.

### Parameters

<code>x</code>	A number.
----------------	-----------

## Math.acos() Method

`Math.acos(x)`

### Description

The `acos()` method returns the arccosine (in radians) of a number. `x` must fall in the range of -1.0 and 1.0. If it does not, the method returns `NaN`.

## Parameters

`x` A number between -1.0 and 1.0.

## See also

“Math.asin() Method” on page 167, “Math.atan() Method” on page 167, “Math.atan2() Method” on page 167, “Math.cos() Method” on page 168, “Math.sin() Method” on page 172, “Math.tan() Method” on page 173

## Math.asin() Method

`Math.asin(x)`

### Description

The `asin()` method returns the arcsine (in radians) of a number. `x` must fall in the range of -1.0 and 1.0. If it does not, the method returns `NaN`.

## Parameters

`x` A number between -1.0 and 1.0.

## See also

“Math.acos() Method” on page 166, “Math.atan() Method” on page 167, “Math.atan2() Method” on page 167, “Math.cos() Method” on page 168, “Math.sin() Method” on page 172, “Math.tan() Method” on page 173

## Math.atan() Method

`Math.atan(x)`

### Description

The `atan()` method returns the arctangent (in radians) of a number. `x` must be in the range of `-Infinity` and `Infinity`, inclusive.

## Parameters

`x` A number in the range of `-Infinity` and `Infinity`, inclusive.

## See also

“Math.acos() Method” on page 166, “Math.asin() Method” on page 167, “Math.atan2() Method” on page 167, “Math.cos() Method” on page 168, “Math.sin() Method” on page 172, “Math.tan() Method” on page 173

## Math.atan2() Method

`Math.atan2(y, x)`

## Description

The `atan2()` method returns the arctangent (in radians) of the quotient of its arguments (`y/x`). Note that the arguments to this function pass the y-coordinate first and the x-coordinate second.

## Parameters

<code>x,y</code>	Two numbers representing a point.
------------------	-----------------------------------

## See also

“[Math.acos\(\) Method](#)” on page 166, “[Math.asin\(\) Method](#)” on page 167, “[Math.atan\(\) Method](#)” on page 167, “[Math.cos\(\) Method](#)” on page 168, “[Math.sin\(\) Method](#)” on page 172, “[Math.tan\(\) Method](#)” on page 173

## Math.ceil() Method

`Math.ceil(x)`

## Description

The `ceil()` method returns the value rounded up to the nearest integer.

## Parameters

<code>x</code>	A number.
----------------	-----------

## See also

“[Math.floor\(\) Method](#)” on page 169

## Math.cos() Method

`Math.cos(x)`

## Description

The `cos()` method returns the cosine of an angle provided in radians. The result is a value between -1 and 1.

## Parameters

<code>x</code>	An angle, in radians.
----------------	-----------------------

## See also

“[Math.acos\(\) Method](#)” on page 166, “[Math.asin\(\) Method](#)” on page 167, “[Math.atan\(\) Method](#)” on page 167, “[Math.atan2\(\) Method](#)” on page 167, “[Math.sin\(\) Method](#)” on page 172, “[Math.tan\(\) Method](#)” on page 173



## Math.E Constant

`Math.E`

### Description

The `E` constant represents Euler's constant and the base of natural logarithms (approximately 2.718).

## Math.exp() Method

`Math.exp(x)`

### Description

The `exp()` method returns `Math.E` raised to the power of `x`.

### Parameters

<code>x</code>	A number.
----------------	-----------

### See also

"Math.E Constant" on page 169, "Math.log() Method" on page 170, "Math.pow() Method" on page 171

## Math.floor() Method

`Math.floor(x)`

### Description

The `floor()` method returns the value rounded down to the nearest integer.

### Parameters

<code>x</code>	A number.
----------------	-----------

### See also

"Math.ceil() Method" on page 168

## Math.LN2 Constant

`Math.LN2`

### Description

The `LN2` constant is the natural logarithm of 2 (approximately 0.693).

## Math.LN10 Constant

`Math.LN10`

**Description**

The `LN10` constant is the natural logarithm of 10 (approximately 2.302).

## Math.log() Method

`Math.log(x)`

**Description**

The `log()` method returns the natural logarithm of a number.

**Parameters**

<code>x</code>	A number.
----------------	-----------

**See also**

“`Math.exp()` Method” on page 169, “`Math.pow()` Method” on page 171

## Math.LOG2E Constant

`Math.LOG2E`

**Description**

The `LOG2E` constant is the base 2 logarithm of E (approximately 1.442).

## Math.LOG10E Constant

`Math.LOG10E`

**Description**

The `LOG10E` constant is the base 10 logarithm of E (approximately 0.434).

## Math.max() Method

`Math.max(x,y)`

**Description**

The `max()` method returns the maximum of two numbers.

**Parameters**

<code>x,y</code>	Two numbers.
------------------	--------------

**See also**

“`Math.min()` Method” on page 171

## Math.min() Method

`Math.min(x,y)`

### Description

The `min()` method returns the minimum of two numbers.

### Parameters

<i>x</i> , <i>y</i>	Two numbers.
---------------------	--------------

### See also

“Math.max() Method” on page 170

## Math.PI Constant

`Math.PI`

### Description

The `PI` constant is the ratio of the circumference of a circle to its diameter (approximately 3.14159).

## Math.pow() Method

`Math.pow(base,exponent)`

### Description

The `pow()` method returns  $x^y$ .

### Parameters

<i>base</i>	The base number.
<i>exponent</i>	The exponent to which <i>base</i> is raised.

### See also

“Math.exp() Method” on page 169, “Math.log() Method” on page 170

## Math.random() Method

`Math.random()`

### Description

The `random()` method returns a pseudo-random number from 0.0 up to but not including 1.0. The random number generator is seeded from the current time.

## Math.round() Method

`Math.round(x)`

### Description

The `round()` method returns the value of a number rounded to the nearest integer. If the fractional portion of number is .5 or greater, the argument is rounded to the next higher integer. If the fractional portion of number is less than .5, the argument is rounded to the next lower integer.

### Parameters

x	A number.
---	-----------

## Math.sin() Method

`Math.sin(x)`

### Description

The `sin()` method returns the sine of an angle provided in radians.

### Parameters

x	An angle, in radians.
---	-----------------------

### See also

“Math.acos() Method” on page 166, “Math.asin() Method” on page 167, “Math.atan() Method” on page 167, “Math.atan2() Method” on page 167, “Math.cos() Method” on page 168, “Math.tan() Method” on page 173

## Math.sqrt() Method

`Math.sqrt(x)`

### Description

The `sqrt()` method returns the square root of a number.

### Parameters

x	A number.
---	-----------

## Math.SQRT1\_2 Constant

`Math.SQRT1_2`

### Description

The `SQRT1_2` constant represents the square root of 1/2—equivalently, 1 over the square root of 2, approximately 0.707.

## Math.SQRT2 Constant

`Math.SQRT2`

### Description

The `SQRT2` constant represents the square root of 2 (approximately 1.414).

## Math.tan() Method

`Math.tan(x)`

### Description

The `tan()` method returns the tangent of an angle provided in radians.

### Parameters

<code>x</code>	An angle, in radians.
----------------	-----------------------

### See also

“`Math.acos()` Method” on page 166, “`Math.asin()` Method” on page 167, “`Math.atan()` Method” on page 167, “`Math.atan2()` Method” on page 167, “`Math.cos()` Method” on page 168, “`Math.sin()` Method” on page 172

## Mouse Object

### Description

The `Mouse` object is used to show or hide the cursor. The `Mouse` object and its methods are static—you do not create `Mouse` objects using a constructor.

### Properties

None.

### Methods

<code>hide()</code>	See “ <code>Mouse.hide()</code> Method” on page 173.	Hide the mouse cursor.
<code>show()</code>	See “ <code>Mouse.show()</code> Method” on page 174.	Show the mouse cursor.

## Mouse.hide() Method

`Mouse.hide()`

### Description

The `hide()` method hides the mouse cursor.

**See also**

“Mouse.show() Method” on page 174

## Mouse.show() Method

```
Mouse.show()
```

**Description**

The `show( )` method shows the mouse cursor.

**See also**

“Mouse.hide() Method” on page 173

## MovieClip Object

**Description**

The MovieClip object is the object at the heart of LiveMotion. `_root` itself is an instance of the MovieClip object, and many of the MovieClip methods are also available as global functions.

**Constructor**

None. Movie clips are created manually using the LiveMotion Composition window. In addition, new movie clips can be added with `attachMovie( )` and `duplicateMovieClip( )`.

**Properties**

<code>_alpha</code>	See “MovieClip._alpha Property” on page 177.	Opacity of the movie clip on a scale of 0 (transparent) to 100 (opaque).
<code>_currentframe</code>	See “MovieClip._currentframe Property” on page 178.	Location of the movie clip playhead.
<code>_droptarget</code>	See “MovieClip._droptarget Property” on page 179.	Absolute path of a movie clip over which the movie clip passes during drag operations by the user.
<code>_framesloaded</code>	See “MovieClip._framesloaded Property” on page 180.	Number of movie clip frames that have been loaded.
<code>_height</code>	See “MovieClip._height Property” on page 184.	Height of the movie clip in pixels.
<code>_name</code>	See “MovieClip._name Property” on page 187.	Name of the movie clip.
<code>_parent</code>	See “MovieClip._parent Property” on page 187.	Movie clip containing this movie clip.
<code>_rotation</code>	See “MovieClip._rotation Property” on page 188.	Rotation angle of the movie clip in degrees.

<code>_target</code>	See “MovieClip._target Property” on page 190.	Absolute path of the movie clip.
<code>_totalframes</code>	See “MovieClip._totalframes Property” on page 190.	Number of frames in the movie clip.
<code>_url</code>	See “MovieClip._url Property” on page 191.	URL from which the movie clip was loaded.
<code>_visible</code>	See “MovieClip._visible Property” on page 191.	Boolean indicating whether the movie clip is visible.
<code>_width</code>	See “MovieClip._width Property” on page 192.	Width of the movie clip in pixels.
<code>_x</code>	See “MovieClip._x Property” on page 192.	Horizontal location of the movie clip in pixels.
<code>_xmouse</code>	See “MovieClip._xmouse Property” on page 192.	Horizontal location of the mouse cursor in pixels.
<code>_xscale</code>	See “MovieClip._xscale Property” on page 193.	Horizontal scaling factor of the movie clip.
<code>_y</code>	See “MovieClip._y Property” on page 193.	Vertical location of the movie clip in pixels.
<code>_ymouse</code>	See “MovieClip._ymouse Property” on page 193.	Vertical location of the mouse cursor in pixels.
<code>_yscale</code>	See “MovieClip._yscale Property” on page 194.	Vertical scaling factor of the movie clip.

## Methods

<code>attachMovie()</code>	See “MovieClip.attachMovie() Method” on page 177.	Attach the named movie clip (passed in as an argument) to the movie clip.
<code>duplicateMovieClip()</code>	See “MovieClip.duplicateMovieClip() Method” on page 179.	Duplicate this movie clip. Also a global movie clip function. See “duplicateMovieClip() Global Function” on page 146
<code>getBounds()</code>	See “MovieClip.getBounds() Method” on page 180.	Return bounds of the movie clip. The returned object contains the values in the properties <code>xMin</code> , <code>xMax</code> , <code>yMin</code> and <code>yMax</code> .
<code>getBytesLoaded()</code>	See “MovieClip.getBytesLoaded() Method” on page 181.	Return the number of bytes already loaded if the movie clip is external (loaded with <code>movieClip.loadMovie()</code> ). If the movie clip is internal, the number returned is always the same as that returned by <code>movieClip.getBytesTotal()</code> .

<code>getBytesTotal()</code>	See “MovieClip.getBytesTotal() Method” on page 181.	Return the size of the movie clip in bytes. When running in Preview mode, you will get an arbitrary number.
<code>getURL()</code>	See “MovieClip.getURL() Method” on page 182.	Load the URL into the browser. Also a global movie clip function. See “getURL() Global Function” on page 149.
<code>globalToLocal()</code>	See “MovieClip.globalToLocal() Method” on page 182.	Convert the given global point to local coordinates.
<code>gotoAndPlay()</code>	See “MovieClip.gotoAndPlay() Method” on page 183.	Go to the specified label and play. Also a global movie clip function. See “gotoAndPlay() Global Function” on page 151.
<code>gotoAndStop()</code>	See “MovieClip.gotoAndStop() Method” on page 183.	Go to the specified label and stop. Also a global movie clip function. See “gotoAndStop() Global Function” on page 151.
<code>hitTest()</code>	See “MovieClip.hitTest() Method” on page 184.	Return a boolean indicating whether the movie clip intersects with a given clip (passed in as an argument) or given x/y coordinates.
<code>lmSetCurrentState()</code>	See “MovieClip.lmSetCurrentState() Method” on page 185.	Change the state of the movie clip.
<code>loadMovie()</code>	See “MovieClip.loadMovie() Method” on page 185.	Load an external SWF file into the player. Also a global movie clip function. See “loadMovie() Global Function” on page 162.
<code>loadVariables()</code>	See “MovieClip.loadVariables() Method” on page 186.	Load variables fetched from the specified URL. The movie clip’s onData handler is called when the variables have been loaded. Also a global movie clip function. See “loadVariables() Global Function” on page 163.
<code>localToGlobal()</code>	See “MovieClip.localToGlobal() Method” on page 187.	Convert the given local point to global coordinates.
<code>nextFrame()</code>	See “MovieClip.nextFrame() Method” on page 187.	Go to the next frame and stop playing. Also a global movie clip function. See “nextFrame() Global Function” on page 194.
<code>play()</code>	See “MovieClip.play() Method” on page 187.	Start playing. Also a global movie clip function. See “play() Global Function” on page 203.



<code>prevFrame()</code>	See “MovieClip.prevFrame() Method” on page 188.	Go to the previous frame and stop playing. Also a global movie clip function. See “prevFrame() Global Function” on page 203.
<code>removeMovieClip()</code>	See “MovieClip.removeMovieClip() Method” on page 188.	Delete a duplicate or attached movie clip. Also a global movie clip function. See “removeMovieClip() Global Function” on page 204.
<code>startDrag()</code>	See “MovieClip.startDrag() Method” on page 188.	Start dragging a movie clip. Also a global movie clip function. See “startDrag() Global Function” on page 213.
<code>stop()</code>	See “MovieClip.stop() Method” on page 189.	Stop playing. Also a global movie clip function. See “stop() Global Function” on page 213.
<code>stopDrag()</code>	See “MovieClip.stopDrag() Method” on page 189.	Stop any drag operation in progress. Also a global movie clip function. See “stopDrag() Global Function” on page 214.
<code>swapDepths()</code>	See “MovieClip.swapDepths() Method” on page 190.	Swap the movie clip’s depth with that of another movie clip.
<code>unloadMovie()</code>	See “MovieClip.unloadMovie() Method” on page 191.	Unload a movie clip that was previously loaded with <code>loadmovie()</code> . Also a global movie clip function. See “unloadMovie() Global Function” on page 225.
<code>valueOf()</code>	See “MovieClip.valueOf() Method” on page 191.	Returns the absolute path to the movie clip using dot (as opposed to slash) notation.

## MovieClip.\_alpha Property

*movieClip.\_alpha*

### Description

The `_alpha` property sets the opacity of the movie clip. 0 is transparent; 100 is opaque. This property can be read or written.

## MovieClip.attachMovie() Method

*movieClip.attachMovie(*exportName*, *newName*, *depth*)*

### Description

The `attachMovie()` method creates a new instance of *exportName* and attaches it to the movie clip by placing it at the designated depth in *movieClip*’s programmatic stack. Remove the attached movie clip by using the *movieClip.removeMovieClip()* method or the `removeMovieClip()` global function. The movie clip may also be removed by placing another movie clip at the same depth in the programmatic stack.

`exportName` is the sharing name of the movie clip that is to be attached.

A movie clip can be attached to the `_root` movie clip as well using the syntax

```
_root.attachMovie(exportName, newName, depth).
```

A movie clip instanced using `attachMovie()` becomes a child of the movie clip through which the method was called, and is in that movie clip's programmatic stack. For example:

```
clipA.attachMovie(exportName, "clipB", depth);
```

`clipB` is a child of `clipA` and is in `clipA`'s programmatic stack.

In contrast, a movie clip instanced using `duplicateMovieClip()` becomes a child of the parent of the movie clip through which the method was called, and is in the parent's programmatic stack. For example:

```
clipA.duplicateMovieClip("clipB", depth);
```

`clipB` is a child of `clipA._parent` and is in `clipA._parent`'s programmatic stack.

**Note:** In Preview mode, the movie clip that is attached is the local version only. If the "Use External Asset" feature is used from the Export palette, this will not be the same movie clip that is actually used when the SWF file is executing in the Flash Player.

## Parameters

<code>exportName</code>	The movie clip to be attached. This movie clip already exists in the current SWF file. It was assigned its sharing name ( <code>exportName</code> ) via the Export palette. A remote copy may or may not have been loaded when the SWF file was loaded into the Flash player, depending on whether the "Use External Asset" feature was used from the Export palette.
<code>newName</code>	A string indicating the name for the attached movie clip.
<code>depth</code>	The depth for the movie clip in the programmatic stack.

## See also

"removeMovieClip() Global Function" on page 204, "MovieClip.removeMovieClip() Method" on page 188, "loadMovie() Global Function" on page 162, "unloadMovie() Global Function" on page 225, "MovieClip.loadMovie() Method" on page 185, "MovieClip.unloadMovie() Method" on page 191, "duplicateMovieClip() Global Function" on page 146, "MovieClip.duplicateMovieClip() Method" on page 179, "Sound.attachSound() Method" on page 208

## MovieClip.\_currentframe Property

`movieClip._currentframe`

### Description

The `_currentframe` property specifies the location (frame number) of the playhead of `movieClip`. This property can only be read.

## MovieClip.\_droptarget Property

*movieClip.\_droptarget*

### Description

The `_droptarget` property is a string value that specifies the absolute path (in slash notation) of a movie clip over which *movieClip* passes during drag operations by the user. To convert a `_droptarget` string to a movie clip reference, use `eval()`. This property can only be read.

## MovieClip.cloneMovieClip() Method

*movieClip.cloneMovieClip(newName, depth)*

### Description

The `cloneMovieClip()` method duplicates *movieClip*. Duplicate movie clips always start playing at frame 1. The duplicate movie clip inherits shape transformations but not the current values of *movieClip*'s user-defined variables. The duplicate movie clip is placed in *movieClip*'s parent's programmatic stack. A programmatic stack holds child movie clips; when you duplicate a movie clip the new movie clip will have the same parent as the original, and thus reside in the parent's programmatic stack. The `removeMovieClip()` method is used to delete duplicate movie clips.

*movieClip.removeMovieClip()* can be used by duplicate movie clips to delete themselves, or the `removeMovieClip()` global function can be used to delete duplicate movie clips. Duplicate movie clips can also be removed by placing another movie clip at the same depth in the programmatic stack.

A movie clip instanced using `cloneMovieClip()` becomes a child of the parent of the movie clip through which the method was called, and is in the parent's programmatic stack. For example:

```
clipA.cloneMovieClip("clipB", depth);
```

*clipB* is a child of *clipA.\_parent* and is in *clipA.\_parent*'s programmatic stack.

In contrast, a movie clip instanced using `attachMovie()` becomes a child of the movie clip through which the method was called, and is in that movie clip's programmatic stack. For example:

```
clipA.attachMovie(exportName, "clipB", depth);
```

*clipB* is a child of *clipA* and is in *clipA*'s programmatic stack.

### Parameters

<i>newName</i>	A string indicating the new name for the duplicate movie clip.
<i>depth</i>	An integer indicating the depth at which the duplicate movie clip is placed in <i>movieClip</i> 's parent's programmatic stack.

### Example

```
_root.baseball.duplicateMovieClip ("newBaseball", 1); //creates new
baseball
_root.newBaseball._x += 25; //moves new baseball along x axis
_root.newBaseball._y += 25; //moves new baseball along y axis
```

### See also

"removeMovieClip() Global Function" on page 204, "MovieClip.removeMovieClip() Method" on page 188, "loadMovie() Global Function" on page 162, "unloadMovie() Global Function" on page 225, "MovieClip.loadMovie() Method" on page 185, "MovieClip.unloadMovie() Method" on page 191, "duplicateMovieClip() Global Function" on page 146, "MovieClip.attachMovie() Method" on page 177

## MovieClip.\_framesloaded Property

*movieClip.\_framesloaded*

### Description

The `_framesloaded` property holds the number of frames that have already been downloaded. This property can only be read.

This property is often used in conjunction with the `_totalframes` property to create a preloader for the `_root` movie clip. For example, you could place the following code in a keyframe script on a frame somewhere between the `beginLoop` and `Start` labels. The `_root` movie clip loops between the `beginLoop` label and the frame where the keyframe script is, then jumps to the `Start` label when the entire `_root` movie clip has downloaded.

```
if (_root._framesloaded == _root._totalframes)
{
 _root.gotoAndPlay("Start");
}
else
{
 _root.gotoAndPlay("beginLoop");
}
```

### See also

"MovieClip.\_totalframes Property" on page 190

## MovieClip.getBounds() Method

*movieClip.getBounds()*

*movieClip.getBounds(targetCoordinateSpace)*

### Description

The `getBounds()` method returns the bounds of the movie clip as an object. If specified, the values returned represent the coordinate space of *targetCoordinateSpace*.

## Parameters

*targetCoordinateSpace* (Optional) A path or reference to a movie clip in which *movieClip*'s bounds are measured. Defaults to *movieClip* if not specified.

## Returns

An object with four properties: *obj.xMin*, *obj.xMax*, *obj.yMin*, *obj.yMax*.

## Example

```
var coordinates = _root.baseball.getBounds();
trace(coordinates.xMin); //prints value
trace(coordinates.xMax); //prints value
trace(coordinates.yMin); //prints value
trace(coordinates.yMax); //prints value

var coordinates = _root.baseball.getBounds("_root");
trace(coordinates.xMin); //prints value
trace(coordinates.xMax); //prints value
trace(coordinates.yMin); //prints value
trace(coordinates.yMax); //prints value
```

## See also

"[MovieClip.globalToLocal\(\) Method](#)" on page 182, "[MovieClip.localToGlobal\(\) Method](#)" on page 187

# MovieClip.getBytesLoaded() Method

*movieClip.getBytesLoaded()*

## Description

The `getBytesLoaded()` method returns the number of bytes already loaded if *movieClip* is external. If internal, the number returned is always the same as that returned by `movieClip.getBytesTotal()`.

## Returns

The number of bytes already loaded for *movieClip*.

## See also

"[MovieClip.getBytesTotal\(\) Method](#)" on page 181

# MovieClip.getBytesTotal() Method

*movieClip.getBytesTotal()*

### Description

The `getBytesTotal()` method returns the size of *movieClip* in bytes. When running in Preview mode, the number returned is arbitrary.

### Returns

The size of *movieClip* in bytes.

### See also

“MovieClip.getBytesLoaded() Method” on page 181

## MovieClip.getURL() Method

```
movieClip.getURL(url, window)
```

```
movieClip.getURL(url, window, howToSendVariables)
```

### Description

The `getURL()` method loads a URL into the web browser. It operates the same as the global form, except when variables are sent they are sent from the *movieClip* timeline.

**Note:** This method is not supported in Preview mode.

### Parameters

<i>url</i>	A string specifying the URL to which to hyperlink. This may be a relative or an absolute pathname, or the name of a document or script.
<i>window</i>	(Optional) A string specifying the target frame in the browser—e.g., <code>_self</code> (the default), <code>_parent</code> , <code>_top</code> , <code>_blank</code> . If omitted, <code>_self</code> is used. Custom names can also be used.
<i>howToSendVariables</i>	(Optional) Omit this parameter if you don't want to send variables. This parameter is a string literal. Specify <code>GET</code> to send variables via get (i.e., tacked onto the end of the URL) or <code>POST</code> to send them with post (i.e., put into the body of the request). Both methods send them in application/x-www-form-urlencoded MIME format. All user-defined variables are sent.

### See also

“getURL() Global Function” on page 149

## MovieClip.globalToLocal() Method

```
movieClip.globalToLocal(point)
```

### Description

The `globalToLocal()` method converts the given global point to local (*movieClip*) coordinates.

## Parameters

*point* An object of type `Object` with two properties: `x` and `y`. `x` and `y` are set to the global coordinates before the object *point* is passed to `globalToLocal()`.

## Example

```
wheresTheMouse = new Object();
wheresTheMouse.x = _root._xmouse;
wheresTheMouse.y = _root._ymouse;
this.globalToLocal(wheresTheMouse);
//wheresTheMouse.x and wheresTheMouse.y now contain local coordinates
```

## See also

“`MovieClip.getBounds()` Method” on page 180, “`MovieClip.localToGlobal()` Method” on page 187, “`Object Class`” on page 199

# MovieClip.gotoAndPlay() Method

`movieClip.gotoAndPlay(label)`

## Description

The `gotoAndPlay()` method goes to the specified *label* and continues playing from *label*.

**Note:** Frame numbers should not be passed to this method. The use of labels is recommended.

## Parameters

*label* A string indicating the destination of the playhead.

## See also

“`MovieClip.gotoAndStop()` Method” on page 183, “`gotoAndPlay()` Global Function” on page 151

# MovieClip.gotoAndStop() Method

`movieClip.gotoAndStop(label)`

## Description

The `gotoAndStop()` method goes to the specified *label* and stops playing.

**Note:** Frame numbers should not be passed to this method. The use of labels is recommended.

## Parameters

*label* A string indicating the destination of the playhead.

## See also

“MovieClip.gotoAndPlay() Method” on page 183, “gotoAndStop() Global Function” on page 151

## MovieClip.\_height Property

*movieClip.\_height*

### Description

The `_height` property represents the height of the movie clip in pixels. The `_height` property is based on the content within *movieClip*. If *movieClip* has no content, then `_height` is 0. `_height` is also determined by placement of the objects within *movieClip*: the farthest object toward the top or bottom determines the value of `_height`. If objects within *movieClip* are moved, `_height` can change. This property can be read or written.

**Note:** Only `_root.height` and `_root.width` return dimensions of the `_root` movie clip.

## See also

“MovieClip.\_width Property” on page 192

## MovieClip.hitTest() Method

*movieClip.hitTest(x, y, shapeFlag)*

*movieClip.hitTest(target)*

### Description

The `hitTest()` method returns a boolean indicating whether *movieClip* intersects with a specific point in the composition, or overlaps with another movie clip. When specifying the hit test, you indicate whether the test involves matching a specific `x/y` point in the composition (first form) against just the border of *movieClip* or all of it, or (second form) finding any overlap with another clip.

## Parameters

<i>x</i>	The horizontal component of the hit test. Defined in global coordinate space.
<i>y</i>	The vertical component of the hit test. Defined in global coordinate space.
<i>shapeFlag</i>	A boolean value indicating whether to test just the bounding box ( <code>false</code> ) or all pixels ( <code>true</code> ) of <i>movieClip</i> for overlap with the point.
<i>target</i>	A path or reference to a movie clip against which the hit test is made.



### Returns

`true` if a hit occurred; `false` otherwise.

### Example

```
if (this.hitTest(_root._xmouse, _root._ymouse, true))
 trace("The mouse has passed over the movie clip");
```

### See also

"[MovieClip.getBounds\(\) Method](#)" on page 180

## MovieClip.lmSetCurrentState() Method

*movieClip*.lmSetCurrentState(*label*)

### Description

The `lmSetCurrentState()` method sets the state of *movieClip*.

### Parameters

<i>label</i>	A string representing a <i>movieClip</i> state that was already defined for <i>movieClip</i> . This can be a predefined state like <code>over</code> , or a custom state. Must appear in quotes.
--------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Example

```
if (_root._xmouse < 175 && _root._ymouse > 100)
 _root.Spiral.lmSetCurrentState("Purple");
if (_root._xmouse > 175 && _root._ymouse > 100)
 _root.Spiral.lmSetCurrentState("Green");
```

## MovieClip.loadMovie() Method

*movieClip*.loadMovie(*url*)

*movieClip*.loadMovie(*url*, *howToSendVariables*)

### Description

The `loadMovie()` method brings an external SWF file into the player. It optionally sends variables to *url.movieClip* and any programmatically generated movie clips associated with it are replaced with the new SWF file. Use `unloadMovie()` to remove the movie clip. The `unloadMovie()` global function can also be used to remove the movie clip.

**Note:** This method is not supported in Preview mode.

## Parameters

<i>url</i>	A string representing the URL from which to get the SWF file to load. This can be an absolute or a relative URL.
<i>howToSendVariables</i>	(Optional) Omit this parameter if you don't want to send variables. This parameter is a string literal. Specify <code>GET</code> to send variables via get (i.e., tacked onto the end of the URL) or <code>POST</code> to send them with post (i.e., put into the body of the request). Both methods send them in application/x-www-form-urlencoded MIME format. All user-defined variables are sent.

## Example

```
_root.baseball.loadMovie("http://devtech.corp.adobe.com/docs/livemotion/billys.swf");
```

## See also

"loadMovie() Global Function" on page 162, "unloadMovie() Global Function" on page 225, "MovieClip.unloadMovie() Method" on page 191

# MovieClip.loadVariables() Method

```
movieClip.loadVariables(url, howToSendVariables)
```

## Description

The `loadVariables()` method loads variables fetched from the specified URL. The movie clip's `onData` event handler is called when all of the variables have been loaded.

The data fetched from the URL must be in the application/x-www-form-urlencoded MIME format.

**Note:** Variables cannot be loaded from a local file in Preview mode. However, HTTP requests for external data can be made.

## Parameters

<i>url</i>	The URL from which to get the variables. For security reasons, the URL must be in the same domain as that from which the movie clip was downloaded.
<i>howToSendVariables</i>	(Optional) Omit this parameter if you don't want to send variables. This parameter is a string literal. If omitted, variables are loaded only. Specify <code>GET</code> to send variables via get (i.e., tacked onto the end of the URL) or <code>POST</code> to send them with post (i.e., put into the body of the request). Both methods send them in application/x-www-form-urlencoded MIME format. All user-defined variables are sent.

## See also

"loadVariables() Global Function" on page 163, "loadVariablesNum() Global Function" on page 164, "getURL() Global Function" on page 149, "MovieClip.getURL() Method" on page 182,

## MovieClip.localToGlobal() Method

*movieClip.localToGlobal(point)*

### Description

The `localToGlobal()` method converts the given local (*movieClip*) point to global coordinates.

### Parameters

<i>point</i>	An object of type <code>Object</code> with two properties: <code>x</code> and <code>y</code> . <code>x</code> and <code>y</code> are set to the local coordinates before the object <i>point</i> is passed to <code>localToGlobal()</code> .
--------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### See also

“`MovieClip.getBounds()` Method” on page 180, “`MovieClip.globalToLocal()` Method” on page 182, “`Object Class`” on page 199

## MovieClip.\_name Property

*movieClip.\_name*

### Description

The `_name` property of the movie clip represents the name of the movie clip as a string (as opposed to a reference). This is a relative reference (no pathname is returned). This property can be read or written.

## MovieClip.nextFrame() Method

*movieClip.nextFrame()*

### Description

The `nextFrame()` method moves the playhead to the next frame and stops the playhead.

### See also

“`nextFrame()` Global Function” on page 194, “`MovieClip.prevFrame()` Method” on page 188, “`MovieClip.stop()` Method” on page 189, “`MovieClip.play()` Method” on page 187

## MovieClip.\_parent Property

*movieClip.\_parent*

### Description

The `_parent` property is a reference (not a string) to the parent of *movieClip*. This allows syntax such as: `_parent._parent.stop()`. This property can only be read.

## MovieClip.play() Method

*movieClip.play()*

## Description

The `play()` method starts playing the timeline of *movieClip*.

## See also

“play() Global Function” on page 203, “MovieClip.prevFrame() Method” on page 188, “MovieClip.nextFrame() Method” on page 187, “MovieClip.stop() Method” on page 189

# MovieClip.prevFrame() Method

*movieClip*.prevFrame()

## Description

The `prevFrame()` method moves the playhead to the previous frame and stops it there.

## See also

“prevFrame() Global Function” on page 203, “MovieClip.nextFrame() Method” on page 187, “MovieClip.stop() Method” on page 189, “MovieClip.play() Method” on page 187

# MovieClip.removeMovieClip() Method

*movieClip*.removeMovieClip()

## Description

The `removeMovieClip()` method deletes the movie clip from the player. Unlike the `removeMovieClip()` global function, movie clips that call this method can only delete themselves.

## See also

“removeMovieClip() Global Function” on page 204, “duplicateMovieClip() Global Function” on page 146, “MovieClip.duplicateMovieClip() Method” on page 179, “MovieClip.attachMovie() Method” on page 177

# MovieClip.\_rotation Property

*movieClip*.\_rotation

## Description

The `_rotation` property specifies the rotation of the movie clip in degrees. This property can be read or written.

# MovieClip.startDrag() Method

*movieClip*.startDrag()

*movieClip*.startDrag(*lockCenter*)

*movieClip*.startDrag(*lockCenter*, *left*, *top*, *right*, *bottom*)

## Description

The `startDrag()` method causes *movieClip* to visually follow the mouse cursor. Use `stopDrag()` to halt dragging.

## Parameters

<i>lockCenter</i>	(Optional) A boolean indicating whether the draggable <i>movieClip</i> should be centered under the mouse cursor ( <i>true</i> ) or dragged relative to the mouse cursor's location when clicked ( <i>false</i> ). Default is <i>false</i> .
<i>left</i>	(Optional) The <i>x</i> -coordinate boundary to the left of which <i>movieClip</i> cannot be dragged.
<i>top</i>	(Optional) The <i>y</i> -coordinate boundary above which <i>movieClip</i> cannot be dragged.
<i>right</i>	(Optional) The <i>x</i> -coordinate boundary to the right of which <i>movieClip</i> cannot be dragged.
<i>bottom</i>	(Optional) The <i>y</i> -coordinate boundary below which <i>movieClip</i> cannot be dragged.

## Example

```
//onButtonPress event
this.startDrag();
//onButtonRelease event
this.stopDrag();
```

## See also

"MovieClip.stopDrag() Method" on page 189, "startDrag() Global Function" on page 213

## MovieClip.stop() Method

*movieClip*.stop()

### Description

The *stop()* method stops playing the timeline of *movieClip*.

### See also

"stop() Global Function" on page 213, "MovieClip.play() Method" on page 187

## MovieClip.stopDrag() Method

*movieClip*.stopDrag()

### Description

The *stopDrag()* method ends any drag operation currently in progress.

### Example

```
//onButtonPress event
this.startDrag();
//onButtonRelease event
```

```
this.stopDrag();
```

**See also**

“[MovieClip.startDrag\(\) Method](#)” on page 188, “[stopDrag\(\) Global Function](#)” on page 214

## MovieClip.swapDepths() Method

```
movieClip.swapDepths(target)
```

```
movieClip.swapDepths(depth)
```

**Description**

The `swapDepths()` method changes the position of `movieClip` in `movieClip`'s parent's visual stacking order (z-order). Movie clips at the top of the stack (higher level numbers) cover those lower in the stack. You can swap the depths of attached or duplicate movie clips with manually created clips, but be sure that you test extensively since this has been a problem area with the Flash Player in the past.

**Parameters**

<i>target</i>	A path or reference to a movie clip to be swapped with <i>movieClip</i> . The movie clip and <i>movieClip</i> must have the same parent.
<i>depth</i>	An integer specifying the level in <i>movieClip</i> 's parent's visual stack with which to swap. If another movie clip resides at this level, then full swapping occurs. Otherwise, <i>movieClip</i> is simply moved to that level. May be 0. The higher the number, the more visible is the layer.

**Example**

```
movieClip.swapDepths(_root.ellipse); //swaps depths with the movie clip ellipse
movieClip.swapDepths(3); //swaps depths at level 3
```

## MovieClip.\_target Property

```
movieClip._target
```

**Description**

The `_target` property represents the target path of `movieClip` in absolute terms using slash notation. To get the path in dot notation, use the `targetPath()` global function. This property can only be read.

**See also**

“[targetPath\(\) Global Function](#)” on page 224

## MovieClip.\_totalframes Property

```
movieClip._totalframes
```

### Description

The `_totalframes` property specifies the total number of frames in *movieClip*. It is often used in conjunction with the `_framesloaded` property to determine the percentage of total frames that have already downloaded; when an acceptable number are ready, the movie clip is started. This property can only be read.

### See also

“MovieClip.\_framesloaded Property” on page 180

## MovieClip.unloadMovie() Method

*movieClip*.unloadMovie()

### Description

The `unloadMovie()` method unloads a movie clip that was previously loaded with `loadMovie()`.

### See also

“unloadMovie() Global Function” on page 225, “MovieClip.loadMovie() Method” on page 185

## MovieClip.\_url Property

*movieClip*.\_url

### Description

The `_url` property specifies the URL of the file from which *movieClip* was loaded. This property can only be read.

### See also

“loadMovie() Global Function” on page 162, “MovieClip.loadMovie() Method” on page 185

## MovieClip.valueOf() Method

*movieClip*.valueOf()

### Description

The `valueOf()` method returns a string that is the path to *movieClip* in absolute terms using dot notation.

### See also

“Object.valueOf() Method” on page 201, “targetPath() Global Function” on page 224

## MovieClip.\_visible Property

*movieClip*.\_visible

### Description

The `_visible` property is a boolean indicating whether `movieClip` is visible. `Visibility:true` if visible; `false` if hidden. This property can be read or written.

### See also

“`MovieClip.swapDepths()` Method” on page 190, “`MovieClip._alpha` Property” on page 177

## MovieClip.\_width Property

`movieClip._width`

### Description

The `_width` property represents the width of the movie clip in pixels. The `_width` property is based on the content within `movieClip`. If `movieClip` has no content, then `_width` is 0. `_width` is also determined by placement of the objects within `movieClip`: the farthest object to the left or right determines the value of `_width`. If objects within `movieClip` are moved, `_width` can change. This property can be read or written.

**Note:** Only `_root._width` and `_root._height` return dimensions of the `_root` movie clip.

### See also

“`MovieClip._height` Property” on page 184

## MovieClip.\_x Property

`movieClip._x`

### Description

The `_x` property specifies the horizontal position of `movieClip` in pixels. If `movieClip` is on the `_root` timeline, then the coordinate system is based on 0,0 x/y coordinates in the upper left corner of the composition. If `movieClip` is contained within another movie clip, `movieClip`'s coordinates are relative to the position of the enclosing movie clip's anchor point. This property can be read or written.

### See also

“`MovieClip._y` Property” on page 193

## MovieClip.\_xmouse Property

`movieClip._xmouse`

### Description

The `_xmouse` property specifies the horizontal location of the mouse cursor in pixels in the local coordinate system of `movieClip`. If `movieClip` is `_root`, then the coordinate system is based on 0,0 x/y coordinates in the upper left corner of the composition. Otherwise, the `_xmouse` coordinate is relative to the position of `movieClip`'s anchor point. This property can only be read.

**Note:** The `_xmouse` and `_ymouse` coordinates are relative to the movie clip. Only `_root._xmouse` and `_root._ymouse` return absolute positions.



**See also**

“MovieClip.\_ymouse Property” on page 193

## MovieClip.\_xscale Property

`movieClip._xscale`

**Description**

The `_xscale` property of `movieClip` represents the horizontal scaling percentage of the movie clip relative to its original size. This property can be read or written.

**See also**

“MovieClip.\_yscale Property” on page 194

## MovieClip.\_y Property

`movieClip._y`

**Description**

The `_y` property specifies the vertical position of `movieClip` in pixels. If `movieClip` is on the `_root` timeline, then the coordinate system is based on 0,0 x/y coordinates in the upper left corner of the composition. If `movieClip` is contained within another movie clip, `movieClip`'s coordinates are relative to the position of the enclosing movie clip's anchor point. This property can be read or written.

**Note:** In the Flash Player, the y-axis is inverted—that is, positive values increase in the “downward” direction rather than upward.

**See also**

“MovieClip.\_x Property” on page 192

## MovieClip.\_ymouse Property

`movieClip._ymouse`

**Description**

The `_ymouse` property specifies the vertical location of the mouse cursor in pixels in the local coordinate system of `movieClip`. If `movieClip` is `_root`, then the coordinate system is based on 0,0 x/y coordinates in the upper left corner of the composition. Otherwise, the `_ymouse` coordinate is relative to the position of `movieClip`'s anchor point. This property can only be read.

**Note:** The `_ymouse` and `_xmouse` coordinates are relative to the movie clip. Only `_root._ymouse` and `_root._xmouse` return absolute positions.

**Note:** In the Flash Player, the y-axis is inverted—that is, positive values increase in the “downward” direction rather than upward.

**See also**

“MovieClip.\_xmouse Property” on page 192

## MovieClip.\_yscale Property

*movieClip.\_yscale*

### Description

The `_yscale` property of *movieClip* represents the vertical scaling percentage of the movie clip relative to its original size. This property can be read or written.

### See also

“MovieClip.\_xscale Property” on page 193

## NaN Global Property

`NaN`

### Description

The `NaN` global property is a predefined variable with the value NaN (Not-a-Number), as specified by the IEEE-754 standard. This property can only be read.

### Example

```
trace(NaN); //prints NaN
var redFish = NaN;
trace(redFish); //prints NaN
```

### See also

“isNaN() Global Function” on page 153, “Number.NaN Property” on page 197

## newline Constant

`newline`

### Description

The `newline` constant is used wherever a `\n` could be used in text to force a line break. It is equivalent to the ASCII value of 10.

## nextFrame() Global Function

`nextFrame()`

### Description

The `nextFrame()` global function moves the playhead of the current timeline to the next frame and stops it.

### See also

“MovieClip.nextFrame() Method” on page 187, “prevFrame() Global Function” on page 203

## Number() Global Function

`Number(expression)`

## Description

The `Number()` global function converts *expression* into a number. Do not confuse this global function with the `Number` object.

## Parameters

<i>expression</i>	A string, boolean, or other expression to convert into a number.
-------------------	------------------------------------------------------------------

## Returns

A number representing the expression, or `NaN` if the expression cannot be converted into a number.

## Example

```
trace(Number(2 * 2)); //prints 4
```

## See also

"Number Object" on page 195, "parseFloat() Global Function" on page 202, "parseInt() Global Function" on page 202, "String() Global Function" on page 214, "Boolean() Global Function" on page 120

# Number Object

## Description

The `Number` object helps you work with numeric values. It is an object wrapper for primitive numeric values.

The primary uses for the `Number` object are to access constant properties that represent the largest and smallest representable numbers, positive and negative infinity, and the Not-a-Number (NaN) value.

The properties of `Number` are properties of the object itself, not of individual `Number` objects. You need to create an instance of type `Number` only when you wish to use its methods.

## Constructor

```
new Number(value)
```

## Parameters

<i>value</i>	The numeric value of the object being created.
--------------	------------------------------------------------

## Properties

<code>MAX_VALUE</code>	See "Number.MAX_VALUE Property" on page 196.	Constant representing the largest representable number
<code>MIN_VALUE</code>	See "Number.MIN_VALUE Property" on page 196.	Constant representing the smallest representable number.

NaN	See “Number.NaN Property” on page 197.	Constant representing the special Not-a -Number value.
NEGATIVE_INFINITY	See “Number.NEGATIVE_INFINITY Property” on page 197.	Constant representing negative infinity.
POSITIVE_INFINITY	See “Number.POSITIVE_INFINITY Property” on page 198.	Constant representing positive infinity.

## Methods

toString()	See “Number.toString() Method” on page 198.	Return a string representing the object.
valueOf()	See “Number.valueOf() Method” on page 199.	Return the primitive value of the object.

## Number.MAX\_VALUE Property

Number.MAX\_VALUE

### Description

The `MAX_VALUE` property represents the maximum representable numeric value. It has value of approximately  $1.79e+308$ , though this may vary depending on platform. Values larger than `MAX_VALUE` are represented as infinity (see “Number.POSITIVE\_INFINITY Property” on page 198 and “Number.NEGATIVE\_INFINITY Property” on page 197). This property can only be read.

### Example

```
if (1000 * 100001 <= Number.MAX_VALUE)
 trace("No overflow");//prints "No overflow"
else
 trace("Overflow");
```

### See also

“Number.MIN\_VALUE Property” on page 196, “Number.POSITIVE\_INFINITY Property” on page 198, “Number.NEGATIVE\_INFINITY Property” on page 197, “Infinity Global Property” on page 152, “-Infinity Global Property” on page 152

## Number.MIN\_VALUE Property

Number.MIN\_VALUE

### Description

The `MIN_VALUE` property represents the smallest positive representable numeric value. It is the number closest to 0—not the most negative number that can be represented.

`MIN_VALUE` has a value of approximately 2.22e-308, though this may vary depending on platform. Values smaller than `MIN_VALUE` ("underflow values") are converted to 0.

### Example

```
if (1/100<= Number.MAX_VALUE)
 trace("No underflow");//prints "No underflow"
else
 trace("Underflow");
```

## See also

"Number.MAX\_VALUE Property" on page 196

## Number.NaN Property

Number.NaN

### Description

The NaN property is a special value representing Not-A-Number. This value complies with the IEEE-754 value for Not-A-Number. This property can only be read.

### Example

```
var twoFish = 1;
if (twoFish < 2 || twoFish > 2)
 twoFish = Number.NaN;
trace(twoFish); //prints "NaN"
```

## Number.NEGATIVE\_INFINITY Property

Number.NEGATIVE INFINITY

### Description

The `NEGATIVE_INFINITY` property is a special numeric value representing negative infinity. Mathematically, this value behaves like infinity—for example, anything multiplied by infinity is infinity, and anything divided by infinity is 0. This property can only be read.

### Example

```
var IQ = -Number.MAX_VALUE*10;
if (IQ == Number.NEGATIVE_INFINITY)
 trace("Really low");//prints "Really low"
else
 trace("Not so low");
```

**See also**

“Number.POSITIVE\_INFINITY Property” on page 198, “Infinity Global Property” on page 152, “-Infinity Global Property” on page 152

## Number.POSITIVE\_INFINITY Property

`Number.POSITIVE_INFINITY`

**Description**

The `POSITIVE_INFINITY` property is a special numeric value representing infinity. This value behaves mathematically like infinity—for example, anything multiplied by infinity is infinity, and anything divided by infinity is 0. This property can only be read.

**Example**

```
var IQ = Number.MAX_VALUE*10;
if (IQ == Number.POSITIVE_INFINITY)
 trace("Really high");//prints "Really high"
else
 trace("Not so high");
```

**See also**

“Number.NEGATIVE\_INFINITY Property” on page 197, “Infinity Global Property” on page 152, “-Infinity Global Property” on page 152

## Number.toString() Method

`num.toString()`  
`num.toString(radix)`

**Description**

The `toString()` method returns a string representing *num*.

**Parameters**

<i>radix</i>	(Optional) An integer between 2 and 36 specifying the base to use for representing numeric values. Default is 10.
--------------	-------------------------------------------------------------------------------------------------------------------

**Returns**

A string representing *num*.

**Example**

```
var tenFish = new Number(10);
trace("Billy and Monica caught " + tenFish.toString() + " fish.");
//prints "Billy and Monica caught 10 fish."
```

**See also**

“Object.toString() Method” on page 200

## Number.valueOf() Method

*num*.valueOf()

### Description

The `valueOf()` method returns the value of *num* as a primitive number.

### Returns

The primitive value of *num*.

### See also

“Object.valueOf() Method” on page 201

## Object Class

### Description

The `Object` class provides the primitive JavaScript object type. All JavaScript objects are derived from the `Object` class. That is, all JavaScript objects have the methods and properties defined for the `Object` class available to them. In C++ terminology, `Object` is the base class that is inherited by all JavaScript objects.

In addition to using a constructor to create a new instance of the `Object` class, you can also use the bracket syntax (e.g., `newObject = { value1: 1, value2: 2}`).

### Constructor

`new Object()`

### Parameters

None.

### Properties

<code>constructor</code>	See “Object.constructor Property” on page 199.	Reference to the function used to create an object.
<code>__proto__</code>	See “Object.__proto__ Property” on page 200.	Reference to an object’s prototype object.

### Methods

<code>toString()</code>	See “Object.toString() Method” on page 200.	Returns a string representing the object.
<code>valueOf()</code>	See “Object.valueOf() Method” on page 201.	Returns the primitive value of the object.

## Object.constructor Property

*obj*.constructor

## Description

The `constructor` property is a reference to the prototype function used to create *obj*. The value of this property is a reference to the function itself, not a string containing the function's name. This property can be read or written.

## Example

```
beret = new Object();
trace (beret.constructor == Object); //prints "true"
```

```
beret = {}
trace (beret.constructor == Object); //prints "true"
```

## Object.\_\_proto\_\_ Property

*obj*.\_\_proto\_\_

## Description

The `__proto__` (double underscores) property is a reference to *obj*'s prototype object. This property can be read or written.

The `prototype` object of the `Object` class, on the other hand, is used to pass properties and methods to objects that inherit the `Object` class. Note that the `__proto__` property and `prototype` object are common to all scripting objects. Since all LiveMotion objects are derived from the `Object` class, you can use the `prototype` object to add methods and properties to all LiveMotion objects. These become global methods and properties. When adding a global property this way, you are in essence creating a global variable.

## Example

```
Object.prototype.newProp = "office"; //create a true global variable
oval = new Date();
trace(oval.newProp); //prints "office"
trace(oval.__proto__); //prints "Date"
```

## Object.toString() Method

*obj*.toString()

## Description

The `toString()` method returns a string representing *obj*. Many objects override this method in favor of their own implementation (for example, `Date.toString()`).

If an object has no string value and no user-defined `toString()` method, `toString()` returns `[object type]`, where *type* is the object type or the name of the constructor function that created the object.

## Returns

A string representing *obj*.



### Example

```
function Cat(name,breed,color,sex) {
 this.name=name
 this.breed=breed
 this.color=color
 this.sex=sex
}
theCat = new Cat("Socks","Calico","chocolate","girl");
```

The following code creates `catToString()`, the function that will be used in place of the default `toString()` method. This function generates a string containing each property, of the form “property = value”.

```
function catToString() {
 var ret = "Cat " + this.name + " is [";
 for (var prop in this)
 ret += " " + prop + " is " + this[prop] + ";";
 return ret + "]"
}
```

The following code assigns the user-defined function to the object's `toString()` method:

```
Cat.prototype.toString = catToString;
```

With the preceding code in place, any time `theCat` is used in a string context, (for example, `trace(theCat.toString())`) JavaScript automatically calls the `catToString` function, which returns the following string:

```
Cat Socks is [name is Socks; breed is Calico; color is chocolate;
sex is girl;]
```

### See also

“Array.toString() Method” on page 119, “Date.toString() Method” on page 145,  
“Boolean.toString() Method” on page 121, “Number.toString() Method” on page 198,  
“Object.valueOf() Method” on page 201

## Object.valueOf() Method

```
obj.valueOf()
```

### Description

The `valueOf()` method returns the primitive value of `obj`. If `obj` has no primitive value, `valueOf()` returns the object itself. Note that you rarely need to invoke the `valueOf()` method yourself. JavaScript automatically invokes it when encountering an object where a primitive value is expected.

The following shows the object types for which the `valueOf()` method is most useful. Most other objects have no primitive values.

- **Number** object type—`valueOf()` returns primitive numeric value associated with the object.
- **Boolean** object type—`valueOf()` returns primitive boolean value associated with the object.
- **String** object type—`valueOf()` returns string associated with the object.

You can create a `valueOf()` method to be called in place of the default `valueOf()` method. Your function must take no arguments.

### Returns

The primitive value of *obj*; if *obj* has no primitive value, `valueOf()` returns the object itself.

### See also

“Boolean.valueOf() Method” on page 121, “MovieClip.valueOf() Method” on page 191, “Number.valueOf() Method” on page 199, “Object.toString() Method” on page 200

## parseFloat() Global Function

`parseFloat(string)`

### Description

The `parseFloat()` global function parses *string* to find the first set of characters that can be converted to a floating-point number and returns that number. If the function does not encounter characters that it can convert to a number, it returns `NaN`. The function supports exponential notation.

### Parameters

*string*                      The string from which to extract a floating-point number.

### Returns

A floating-point number, or `NaN` if no number was found.

### Example

```
trace(parseFloat("2.12")); //prints 2.12
trace(parseFloat("a23")); //prints NaN
trace(parseFloat("25e10")); //prints 250000000000
```

### See also

“Number() Global Function” on page 194, “parseInt() Global Function” on page 202

## parseInt() Global Function

`parseInt(string)`

`parseInt(string, base)`

## Description

The `parseInt()` global function parses *string* to find the first set of characters that can be converted to an integer in the specified *base* and returns that integer. If the function does not encounter characters that it can convert to an integer, it returns `NaN`.

## Parameters

<i>string</i>	The string from which to extract an integer.
<i>base</i>	(Optional) The base of the string to parse (from base 2 to base 36). If not supplied, <i>base</i> is determined by the format of <i>string</i> .

## Returns

An integer in base 10, or `NaN` if no number was found.

## Example

```
trace(parseInt("10")); //prints 10
trace(parseInt("10", 2)); //prints 2 (decimal equivalent of binary 10)
trace(parseInt("0xFF")); //prints 255 (decimal equivalent of hex FF)
trace(parseInt("0377")); //prints 255 (decimal equivalent of octal 377)
```

## See also

“Number() Global Function” on page 194, “parseFloat() Global Function” on page 202

# play() Global Function

`play()`

## Description

The `play()` global function moves the playhead of the current timeline forward.

## See also

“gotoAndPlay() Global Function” on page 151, “MovieClip.play() Method” on page 187, “stop() Global Function” on page 213

# prevFrame() Global Function

`prevFrame()`

## Description

The `prevFrame()` global function moves the playhead of the current timeline to the previous frame and stops it there.

## See also

“MovieClip.prevFrame() Method” on page 188, “nextFrame() Global Function” on page 194

## **`_quality` Global Property**

`_quality`

### **Description**

The `_quality` global property sets the level of rendering quality. It takes one of the following strings (must be used with quotes):

- "LOW"—Graphics aren't anti-aliased; bitmaps aren't smoothed.
- "MEDIUM"—Graphics are anti-aliased using a 2x2 grid; bitmaps aren't smoothed.
- "HIGH"—Graphics are anti-aliased using a 4x4 grid; bitmaps are smoothed if the movie clip is static.
- "BEST"—Graphics are anti-aliased using a 4x4 grid; bitmaps are always smoothed.

## **`removeMovieClip()` Global Function**

`removeMovieClip(target)`

### **Description**

The `removeMovieClip()` global function deletes a movie clip. It can be used to delete movie clips created with the `duplicateMovieClip()`, `movieClip.duplicateMovieClip()`, or `movieClip.attachMovie()`.

### **Parameters**

*target*                      A path or a reference to an existing movie clip.

### **See also**

"`duplicateMovieClip()` Global Function" on page 146, "`MovieClip.duplicateMovieClip()` Method" on page 179, "`MovieClip.attachMovie()` Method" on page 177, "`MovieClip.removeMovieClip()` Method" on page 188

## **`_root` Global Property**

`_root`

### **Description**

`_root` is a special case of the `MovieClip` object. `_root` is a reference to the root movie clip in the current player level, and as such it can be used in absolute paths to any object. This property can only be read. It's equivalent to saying `_level14` if the script is also at `_level14`. It is most often used to invoke methods and reference properties that are members of the `_root` movie clip. For example:

```
_root.attachMovie(exportName, newName, depth) //attaches movie clip to
_root
```

```
_root._x = -150 //causes a horizontal offset of the entire SWF file
```

### **See also**

"`_levelN` Global Property" on page 161, "`MovieClip._parent` Property" on page 187

# Selection Object

## Description

The `Selection` object contains information about the text field that currently has focus. A text field gets focus when the user clicks on the text field with the mouse. Since only one text field can have focus at a time, the `Selection` object is static. No constructor is required. In LiveMotion, text fields are created using the text field tool.

Using the `Selection` object you can control a user's interaction with text fields and capture text from the text fields. You can position or get the position of the cursor in a text field.

## Properties

None.

## Methods

<code>getBeginIndex()</code>	See "Selection.getBeginIndex() Method" on page 205.	Return the index of the beginning of the selection span. Return -1 if there is no currently selected field.
<code>getCaretIndex()</code>	See "Selection.getCaretIndex() Method" on page 206.	Return the index of the current caret (vertical text cursor).
<code>getEndIndex()</code>	See "Selection.getEndIndex() Method" on page 206.	Return the index of the end of the current selection. Returns -1 if there is no currently selected field.
<code>getFocus()</code>	See "Selection.getFocus() Method" on page 206.	Return a string that is the absolute path to the text field with the current focus.
<code>setFocus()</code>	See "Selection.setFocus() Method" on page 206.	Set the focus of the editable text field associated with the variable in the argument.
<code>setSelection()</code>	See "Selection.setSelection() Method" on page 207.	Set the beginning and ending indices of the selection span.

## Selection.getBeginIndex() Method

`Selection.getBeginIndex()`

### Description

The `getBeginIndex()` method returns the index of the first character of the selection span. It returns -1 if there is no currently selected field. The index is zero-based, where the first position in the text field is 0. If no text is selected, the position of the cursor is returned.

### Returns

Index of the beginning of the selection span. Returns -1 if there is no currently selected field. If no text is selected, the position of the cursor is returned.

### See also

"Selection.getEndIndex() Method" on page 206

## Selection.getCaretIndex() Method

`Selection.getCaretIndex()`

### Description

The `getCaretIndex()` method returns the index of the current caret (vertical text cursor) in the selection that currently has focus. If there is no current selection, `-1` is returned.

### Returns

Index of the current caret (vertical text cursor) in the selection that currently has focus. If there is no current selection, `-1` is returned.

## Selection.getEndIndex() Method

`Selection.getEndIndex()`

### Description

The `getEndIndex()` method returns the index of the character after the last character of the selection span. It returns `-1` if there is no currently selected field. The index is zero-based, where the first position in the text field is `0`. If no text is selected, the position of the cursor is returned.

### Returns

Index of the character after the last character of the selection span. Returns `-1` if there is no currently selected field. If no text is selected, the position of the cursor is returned.

### See also

“[Selection.getBeginIndex\(\) Method](#)” on page 205

## Selection.getFocus() Method

### Description

The `getFocus()` method returns a string that is the absolute path to the text field with the current focus. If no text field is selected, `null` is returned. The result can be `eval()`'ed—i.e., `eval(Selection.getFocus())` returns a reference to the text field.

### Returns

A string that is the absolute path to the text field with the current focus. If no text field is selected, `null` is returned.

### See also

“[Selection.setFocus\(\) Method](#)” on page 206

## Selection.setFocus() Method

`Selection.setFocus(textFieldPath)`

## Description

The `setFocus()` method sets the focus of the editable text field associated with the variable in the argument.

## Parameters

<i>textFieldPath</i>	A string representing the path to the text field that will gain focus.
----------------------	------------------------------------------------------------------------

## Returns

`true` if the focus was set, `false` otherwise.

## Example

```
trace(Selection.setFocus("_root.display")); //prints "true" if there
is a text box whose var = display
```

## See also

“Selection.setFocus() Method” on page 206

# Selection.setSelection() Method

`Selection.setSelection(start, end)`

## Description

The `setSelection()` method sets the beginning and ending indices of the selection span. The indices are zero-based, where the first position in the text field is 0. The method has no effect if there is no currently selected text field. If `start = end`, the cursor is set at that point in the text.

## Parameters

<i>start</i>	The index of the beginning of the selection.
<i>end</i>	The index of the character after the last character to be included in the new selection.

## See also

“Selection.getBeginIndex() Method” on page 205, “Selection.getEndIndex() Method” on page 206

# Sound Object

## Description

The `Sound` object is used to create an object that plays a sound. The object can be set and controlled to provide the sounds for an individual movie clip, including `_root`, or for the global timeline. All of the movie clip's children are affected by a `Sound` object created for it.

## Constructor

`new Sound()`

`new Sound(target)`

## Parameters

*target* (Optional) A path or reference to a player level or an existing movie clip. If not specified, the `Sound` object created controls all sounds in the global timeline. All of the sound in the movie clip hierarchy from this point down will be controlled by the new `Sound` object.

## Properties

None.

## Methods

<code>attachSound()</code>	See “ <code>Sound.attachSound()</code> Method” on page 208.	Add a new sound to a movie clip.
<code>getPan()</code>	See “ <code>Sound.getPan()</code> Method” on page 209.	Get the current pan value of a sound.
<code>getTransform()</code>	See “ <code>Sound.getTransform()</code> Method” on page 209.	Get the current panning transform value of a sound.
<code>getVolume()</code>	See “ <code>Sound.getVolume()</code> Method” on page 210.	Get the current volume of a sound.
<code>setPan()</code>	See “ <code>Sound.setPan()</code> Method” on page 210.	Set the current pan value of a sound.
<code>setTransform()</code>	See “ <code>Sound.setTransform()</code> Method” on page 210.	Set the current panning transform value of a sound.
<code>setVolume()</code>	See “ <code>Sound.setVolume()</code> Method” on page 211.	Set the current volume of a sound.
<code>start()</code>	See “ <code>Sound.start()</code> Method” on page 212.	Play a sound.
<code>stop()</code>	See “ <code>Sound.stop()</code> Method” on page 212.	Stop playing a sound or all sounds.

## Sound.attachSound() Method

`soundObj.attachSound(exportName)`



## Description

The `attachSound()` method attaches a sound to a `Sound` object. `exportName` is the sharing name of the sound. This is the sound file that was imported into LiveMotion, then assigned a sharing name using the Export palette. Only one sound at a time can be attached to `soundObj`.

**Note:** In Preview mode, the sound that is attached is the local version only. If the “Use External Asset” feature is used from the Export palette, this will not be the same sound that is actually used when the SWF file is executing in the Flash Player.

## Parameters

<code>exportName</code>	The sharing name of the sound to attach. This name was assigned to the sound using the Export palette.
-------------------------	--------------------------------------------------------------------------------------------------------

## See also

“`MovieClip.attachMovie()` Method” on page 177

## Sound.getPan() Method

`soundObj.getPan()`

### Description

The `getPan()` method gets the current pan value of the sound. This value was set by the last call to `setPan()`. The pan value is used to implement the balance function between audio channels.

### Returns

The pan value of the sound (a number in the range of -100 to 100).

### See also

“`Sound.setPan()` Method” on page 210

## Sound.getTransform() Method

`soundObj.getTransform()`

### Description

The `getTransform()` method returns the current panning transform values of a `Sound` object. The panning transform values are similar to the pan value, but they let you specify the relative amounts of right channel sound to be included in the left speaker, and vice versa.

### Returns

An object of type `Object` with the following properties:

- `l1`— the percentage of the left channel to play in the left speaker (an integer value in the range of 0 to 100).

- `lr`—the percentage of the left channel to play in the right speaker (an integer value in the range of 0 to 100).
- `rl`—the percentage of the right channel to play in the left speaker (an integer value in the range of 0 to 100).
- `rr`—the percentage of the right channel to play in the right speaker (an integer value in the range of 0 to 100).

**See also**

“`Sound.setTransform()` Method” on page 210, “Object Class” on page 199

## Sound.getVolume() Method

*soundObj.getVolume()*

**Description**

The `getVolume()` method gets the current volume of a sound. This is the volume set by the last `setVolume()` call. Values are from 0 - 100.

**Returns**

The volume of the sound (an integer value in the range from 0 - 100).

**See also**

“`Sound.setVolume()` Method” on page 211

## Sound.setPan() Method

*soundObj.setPan(pan)*

**Description**

The `setPan()` method sets the current pan value of a `Sound` object. The pan value is used to implement the balance function between audio channels. A value of -100 routes all sound through the left channel only; a value of 100 routes all sound through the right channel. Values in between reflect the range between these two extremes, with a value of 0 indicating equal balance between the two channels. Default value is 0.

**Parameters**

<i>pan</i>	The pan value of the sound (a number in the range of -100 to 100).
------------	--------------------------------------------------------------------

**See also**

“`Sound.getPan()` Method” on page 209

## Sound.setTransform() Method

*soundObj.setTransform(transform)*

## Description

The `setTransform()` method sets the current panning transform values of a `Sound` object. The panning transform values are similar to the `pan` value, but they let you specify the relative amounts of right channel sound to be included in the left speaker, and vice versa. The panning transform values are passed into the `setTransform()` method by instantiating an object of type `Object` and setting the following four properties:

- `ll`— the percentage of the left channel to play in the left speaker (an integer value in the range of 0 to 100);
- `lr`—the percentage of the left channel to play in the right speaker (an integer value in the range of 0 to 100);
- `rl`—the percentage of the right channel to play in the left speaker (an integer value in the range of 0 to 100);
- `rr`—the percentage of the right channel to play in the right speaker (an integer value in the range of 0 to 100).

An `ll` value of, for example, 50% indicates that 50% of the left channel content should be played through the left speaker.

## Parameters

<i>transform</i>	An object with <code>ll</code> , <code>lr</code> , <code>rl</code> , and <code>rr</code> properties.
------------------	------------------------------------------------------------------------------------------------------

## Example

```
waveringVoice = new Object();
voice.ll = 50;
voice.lr = 50;
voice.rl = 50;
voice.rr = 50;
soundObj.setTransform(waveringVoice);
```

## See also

“`Sound.getTransform()` Method” on page 209, “Object Class” on page 199

## Sound.setVolume() Method

*soundObj.setVolume(volume)*

## Description

The `setVolume()` method sets the current volume of a sound.

## Parameters

<i>volume</i>	The volume of the sound (an integer in the range of 0 - 100).
---------------	---------------------------------------------------------------

**See also**

“Sound.getVolume() Method” on page 210

## Sound.start() Method

```
soundObj.start(offset, loops)
```

**Description**

The `start()` method plays the sound attached to *soundObj*.

**Parameters**

<i>offset</i>	The number of seconds to wait before playing the sound. Default value is 0.
<i>loops</i>	The number of times to loop the sound before stopping. Default value is 1.

**See also**

“Sound.stop() Method” on page 212

## Sound.stop() Method

```
soundObj.stop()
soundObj.stop(exportName)
```

**Description**

The `stop()` method stops playing a sound or all sounds. All sounds controlled by *soundObj* are stopped if no argument is provided.

**Parameters**

<i>exportName</i>	(Optional) The sharing name of the sound to stop. This name was assigned to the sound using the Export palette.
-------------------	-----------------------------------------------------------------------------------------------------------------

**See also**

“Sound.start() Method” on page 212

## \_soundbuftime Global Property

```
_soundbuftime
```

**Description**

The `_soundbuftime` global property is an integer indicating the number of seconds of streaming sound to load before playing starts. Default value is 5 seconds. This property can be read or written.

## startDrag() Global Function

```
startDrag(target)
```

```
startDrag(target, lockCenter)
```

```
startDrag(target, lockCenter, left, top, right, bottom)
```

### Description

The `startDrag()` global function causes *target* to visually follow the mouse cursor. Use the `stopDrag()` global function to halt dragging.

### Parameters

<i>target</i>	A path or reference to the existing movie clip to drag.
<i>lockCenter</i>	(Optional) A boolean indicating whether the draggable <i>target</i> should be centered under the mouse cursor ( <code>true</code> ) or dragged relative to the mouse cursor's location when clicked ( <code>false</code> ). Default is <code>false</code> .
<i>left</i>	(Optional) The <i>x</i> -coordinate boundary to the left of which <i>target</i> cannot be dragged.
<i>top</i>	(Optional) The <i>y</i> -coordinate boundary above which <i>target</i> cannot be dragged.
<i>right</i>	(Optional) The <i>x</i> -coordinate boundary to the right of which <i>target</i> cannot be dragged.
<i>bottom</i>	(Optional) The <i>y</i> -coordinate boundary below which <i>target</i> cannot be dragged.

### See also

"`stopDrag()` Global Function" on page 214, "`MovieClip.startDrag()` Method" on page 188

## stop() Global Function

```
stop()
```

### Description

The `stop()` global function stops playing the timeline of the current movie clip.

### See also

"`play()` Global Function" on page 203

## stopAllSounds() Global Function

```
stopAllSounds()
```

### Description

The `stopAllSounds()` global function stops all sounds currently playing in the composition. It doesn't stop the playhead and it doesn't stop new sounds from starting.

**See also**

“Sound.stop() Method” on page 212

## stopDrag() Global Function

`stopDrag()`

**Description**

The `stopDrag()` global function stops the dragging of the currently draggable object.

**See also**

“startDrag() Global Function” on page 213, “MovieClip.stopDrag() Method” on page 189

## String() Global Function

`String(value)`

**Description**

The `String()` global function returns a primitive string representation of `value`. Do not confuse this global function with the `String` object.

**Parameters**

<i>value</i>	A number, string, variable, or boolean to convert to a string.
--------------	----------------------------------------------------------------

**Returns**

- If *value* is a boolean, returns `true` or `false`.
- If *value* is a string, returns the string.
- If *value* is a number, returns a string representation of the number.
- If *value* is a MovieClip object, returns the absolute path.
- If *value* is an object, returns a string representation of the object.
- If *value* is undefined, returns an empty string.

**See also**

“String Object” on page 214, “Object.toString() Method” on page 200, “Boolean() Global Function” on page 120, “Number() Global Function” on page 194

## String Object

**Description**

The `String` object is a wrapper around the string primitive data type. Do not confuse a string literal with the `String` object. For example, the following code creates the string literal `s1` and also the `String` object `s2`:

```
s1 = "foo" // creates a string literal value
s2 = new String("foo") // creates a String object
```

```
trace(s1.valueOf()); //prints "foo"
trace(s2.valueOf()); //prints "foo"
```

You can call any of the methods of the `String` object on a string literal value— JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `length` property with a string literal.

## Constructor

```
new String(value)
```

## Parameters

<i>value</i>	The initial value of the string object, or a number, variable, or boolean to convert to a string. If this parameter is not supplied, the string will be set to "" (the empty string).
--------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Properties

<code>length</code>	See “String.length Property” on page 219.	The length of the string.
---------------------	-------------------------------------------	---------------------------

## Methods

<code>charAt()</code>	See “String.charAt() Method” on page 216.	Return the character at the specified index.
<code>charCodeAt()</code>	See “String.charCodeAt() Method” on page 217.	Return the ASCII value of the character at the specified index.
<code>concat()</code>	See “String.concat() Method” on page 217.	Concatenate the text of two or more strings and return the new string.
<code>fromCharCode()</code>	See “String.fromCharCode() Method” on page 218.	Return a string created from the characters specified in the argument list.
<code>indexOf()</code>	See “String.indexOf() Method” on page 218.	Return the index of the first occurrence of the specified value in the string, or -1 if not found.
<code>lastIndexOf()</code>	See “String.lastIndexOf() Method” on page 219.	Return the index of the last occurrence of the specified value in the string, or -1 if not found.
<code>slice()</code>	See “String.slice() Method” on page 220.	Return a string consisting of the substring specified in the argument list.
<code>split()</code>	See “String.split() Method” on page 220.	Split a string into an array of substrings.
<code>substr()</code>	See “String.substr() Method” on page 221.	Return the specified number of characters in a string beginning at the specified location.

<code>substring()</code>	See “String.substring() Method” on page 222.	Return the characters between the two indices into the string.
<code>toLowerCase()</code>	See “String.toLowerCase() Method” on page 223.	Convert the string to lowercase and return.
<code>toUpperCase()</code>	See “String.toUpperCase() Method” on page 223.	Convert the string to uppercase and return.

## String.charAt() Method

*stringObj*.charAt(*index*)

### Description

The `charAt()` method returns the specified character from the string. Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character is the length of string minus 1 (zero-based indexing). If the index is out of range, JavaScript returns an empty string.

### Parameters

<i>index</i>	An integer between 0 and the length of the string minus 1 (zero-based indexing).
--------------	----------------------------------------------------------------------------------

### Returns

A string consisting of one character or an empty string (if the index is out of range).

### Example

The following example displays characters at sequential locations in the string “Billy”:

```
var anyString="Billy";

trace("The character at index 0 is " + anyString.charAt(0));
trace("The character at index 1 is " + anyString.charAt(1));
trace("The character at index 2 is " + anyString.charAt(2));
trace("The character at index 3 is " + anyString.charAt(3));
trace("The character at index 4 is " + anyString.charAt(4));

//prints
//The character at index 0 is B
//The character at index 1 is i
//The character at index 2 is l
//The character at index 3 is l
//The character at index 4 is y
```

### See also

“String.indexOf() Method” on page 218, “String.lastIndexOf() Method” on page 219



## String.charCodeAt() Method

*stringObj*.charCodeAt(*index*)

### Description

The `charCodeAt()` method returns the ASCII value of the character at the given index.

### Parameters

<i>index</i>	An integer between 0 and the length of the string minus 1 (zero-based indexing). Default value is 0.
--------------	------------------------------------------------------------------------------------------------------

### Returns

The ASCII value of the character.

### Example

```
trace("ICE".charCodeAt(0)); // prints 73 - the ASCII value of "I"
trace("ICE".charCodeAt()); // prints 73 - the ASCII value of "I"
trace("ICE".charCodeAt(1)); // prints 67 - the ASCII value of "C"
trace("ICE".charCodeAt(2)); // prints 69 - the ASCII value of "E"
```

## String.concat() Method

*stringObj*.concat(*value1*, *value2*, ...*valuen*)

### Description

The `concat()` method concatenates the text of one or more strings to *stringObj* and returns the new string. If necessary, it first converts a given *value* to a string. The original string is not affected.

### Parameters

<i>value1</i> , <i>value2</i> , ... <i>valuen</i>	The values to concatenate to <i>stringObj</i> .
---------------------------------------------------	-------------------------------------------------

### Returns

The concatenated string.

### Example

The following example combines two strings into a new string.

```
s1="Billy ";
s2="and ";
s3="Monica are fishing.";
trace(s1.concat(s2,s3)); // prints "Billy and Monica are fishing."
```

## String.fromCharCode() Method

`String.fromCharCode(value1, value2, ...valuen)`

### Description

The `fromCharCode()` method returns a string created by using the specified sequence of ASCII values. Because `fromCharCode()` is a static method of `String`, you always use it as `String.fromCharCode()`, rather than as a method of a `String` object you create.

### Parameters

*value1, value2, ...valuen*                      A sequence of ASCII values.

### Returns

A string consisting of the characters provided as ASCII values.

### Example

```
trace(String.fromCharCode(66,105,108,108,121)); //Returns "Billy"
```

## String.indexOf() Method

`stringObj.indexOf(searchValue, fromIndex)`

### Description

The `indexOf()` method returns the index within the string of the first occurrence of the specified value, starting the search at *fromIndex* if provided. The method returns `-1` if the value is not found.

Characters in a string are indexed from left to right. The index of the first character is `0`, and the index of the last character is length of the string minus `1` (zero-based indexing).

### Parameters

*searchValue*                      The string value for which to search.

*fromIndex*                      *(Optional)* The location within the current string from which to start the search. Can be any integer between `0` and the length of the string minus `1` (zero-based indexing). If this argument is not supplied, the default value is `0`.

### Returns

The position (zero-based) within the string where the first occurrence of *searchValue* was found, or `-1` if it was not found.

### Example

```
trace("Favorite beret".indexOf("Favorite")); // prints 0
trace("Favorite beret".indexOf("Hat")); // prints -1
trace("Favorite beret".indexOf("beret",0)); // prints 9
trace("Favorite beret".indexOf("beret",9)); // prints 9
```

**See also**

“String.charAt() Method” on page 216, “String.lastIndexOf() Method” on page 219

## String.lastIndexOf() Method

*stringObj.lastIndexOf(searchValue, fromIndex)*

**Description**

The `lastIndexOf()` method returns the index within the string of the last occurrence of the specified value, or `-1` if not found. The string is searched backward, starting at *fromIndex*.

Characters in a string are indexed from left to right. The index of the first character is `0`, and the index of the last character is the length of the string minus `1`.

**Parameters**

<i>searchValue</i>	A string representing the value to search for.
<i>fromIndex</i>	(Optional) The location within the current string from which to start the search. Can be any integer between <code>0</code> and the length of the string minus <code>1</code> (zero-based indexing). If this argument is not supplied, the default value is <code>0</code> .

**Returns**

The position (zero-based) within the string where the last occurrence of *searchValue* was found, or `-1` if it was not found.

**Example**

```
trace("Billy".lastIndexOf("l")); // prints 3
trace("Billy".lastIndexOf("l",2)); // prints 2
trace("Billy".lastIndexOf("x")); // prints -1
```

**See also**

“String.charAt() Method” on page 216, “String.indexOf() Method” on page 218

## String.length Property

*stringObj.length*

**Description**

The `length` property is the length of the string. An empty ("" ) string has a length of `0`. This property can only be read.

**Example**

```
var x="Billy";
trace("Length is " + x.length); //prints "Length is 5"
```

## String.slice() Method

*stringObj.slice(startSlice, endSlice)*

### Description

The `slice()` method extracts a section of the string and returns the new string. `slice()` extracts up to but not including `endSlice`. Indexing is zero-based. For example, `slice(1,4)` extracts the second character through the fourth character (characters indexed 1, 2, and 3). The original string is unchanged.

As a negative index, `startSlice` or `endSlice` indicates an offset from the end of the string, where the last character is -1, the second is -2, etc. For example, `slice(2,-1)` extracts the third character through the second to last character in the string.

### Parameters

<i>startSlice</i>	The zero-based index at which to begin extraction.
<i>endSlice</i>	(Optional) The zero-based index at which to end extraction. If omitted, slice extracts to the end of the string.

### Returns

A substring of characters from *stringObj*, starting at *startSlice* and ending with *endSlice* minus 1.

### Example

```
str1="Billy and Monica are ice skating.";
str2=str1.slice(10,-5);
trace(str2); //Prints "Monica are ice ska"
```

### See also

"String.substring() Method" on page 222, "String.substr() Method" on page 221

## String.split() Method

*stringObj.split(delimiter)*

### Description

The `split()` method splits the string into a group of substrings, places those strings into an array, and returns the array. The substrings are created by breaking the original string at points that match *delimiter*. When found, *delimiter* is removed from the string and the resulting substring is added to the array.

### Parameters

<i>delimiter</i>	(Optional) The character to use for delimiting. The delimiter is treated as a string. If omitted, the array returned contains one element consisting of the entire string.
------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Returns

An array whose elements are the substrings.

## Example

```
myString = "Hello Billy. Let's go fishing.";
splits = myString.split(" ");
for(i=0; (splits[i] != "fishing."); ++i)
 trace(splits[i]);
trace(splits[i]);
//Displays
//Hello
//Billy.
//Let's
//go
//fishing.
```

## See also

"String.charAt() Method" on page 216, "String.lastIndexOf() Method" on page 219,  
"String.indexOf() Method" on page 218, "Array.join() Method" on page 112

# String.substr() Method

*stringObj.substr(start, length)*

## Description

The `substr()` method returns the characters in the string beginning at *start* and continuing through the specified number of characters. *start* is a character index. The index of the first character is 0, and the index of the last character is the length of the string minus 1 (zero-based indexing). `substr()` begins extracting characters at *start* and collects *length* number of characters. The original string is unchanged.

If *start* is negative, `substr()` uses it as a character index from the end of the string (*stringObj.length* plus *start*). If *length* is omitted, *start* extracts characters to the end of the string.

## Parameters

<i>start</i>	The location at which to begin extracting characters.
<i>length</i>	(Optional) The number of characters to extract.

## Returns

A string containing the extracted characters.

## Example

```
str = "phonecall"
trace("(1,2): " + str.substr(1,2));
```

```
trace("(-2,2): " + str.substr(-2,2));
trace("(1): " + str.substr(1));
trace("(20, 2): " + str.substr(20,2));
//prints
//(1,2): ho
//(-2,2): ll
//(1): honecall
//(20, 2):
```

### See also

"String.substring() Method" on page 222, "String.slice() Method" on page 220

## String.substring() Method

*stringObj*.substring(*indexA*, *indexB*)

### Description

The `substring()` method returns a substring of the string by extracting characters from *indexA* up to but not including *indexB*. The original string is unchanged. Specifically:

- If *indexA* is less than 0, *indexA* is treated as if it were 0.
- If *indexB* is less than 0, *indexB* is treated as if it were 0.
- If *indexB* is greater than or equal to *stringObj*.length, characters are extracted to the end of the string.
- If *indexA* equals *indexB*, `substring()` returns an empty string.
- If *indexB* is omitted, characters are extracted to the end of the string.
- If *indexB* is less than *indexA*, the two indices are automatically re-ordered.

### Parameters

<i>indexA</i>	An integer between 0 and the length of the string minus 1 (zero-based indexing).
<i>indexB</i>	(Optional) An integer between 0 and the length of the string minus 1 (zero-based indexing).

### Returns

A substring of characters from *stringObj*.

### Example

```
var str="trolling";
// Prints "tro"
trace(str.substring(0,3));
trace(str.substring(3,0)); //automatic re-ordering
// Prints "lin"
trace(str.substring(4,7));
```

```
trace(str.substring(7,4));
// Prints "trollin"
trace(str.substring(0,7));
// Prints "trolling"
trace(str.substring(0,8));
trace(str.substring(0,10));
```

**See also**

“String.substr() Method” on page 221, “String.slice() Method” on page 220

## String.toLowerCase() Method

*stringObj*.toLowerCase()

**Description**

The `toLowerCase()` method returns *stringObj* converted to lower case without affecting the value of the string itself.

**Returns**

A lower case string.

**Example**

The following example prints the lower case string “white house”:

```
var upperCase="WHITE HOUSE";
trace(upperCase.toLowerCase())//prints "white house"
```

**See also**

“String.toUpperCase() Method” on page 223

## String.toUpperCase() Method

*stringObj*.toUpperCase()

**Description**

The `toUpperCase()` method returns *stringObj* converted to upper case without affecting the value of the string itself.

**Returns**

An upper case string.

**Example**

The following example prints the string “WHITE HOUSE”:

```
var lowerCase="white house";
trace(lowerCase.toUpperCase())//prints "WHITE HOUSE"
```

**See also**

“String.toLowerCase() Method” on page 223

## targetPath() Global Function

```
targetPath(movieClip)
```

### Description

The `targetPath()` global function returns the absolute path to *movieClip* as a string in dot notation. To get the path in slash notation, use the `_target` property of `MovieClip`.

### Parameters

<i>movieClip</i>	A reference to the movie clip for which the path is requested.
------------------	----------------------------------------------------------------

### Returns

A string representing the absolute path to *movieClip*.

### Example

```
targetPath(oval);
```

### See also

"`MovieClip._target` Property" on page 190

## Text Field Properties

```
variableName.scroll
```

```
variableName.maxscroll
```

### Description

The `scroll` and `maxscroll` text field properties give you control over the display of text in a text field. *variableName* is the name of the variable (var=) associated with the text field.

The `scroll` text field property allows you to control the display of information in a text field by moving the text in the text field to a specific position. It is set to the line number of the line that you want to be the topmost visible line in the text field. It is used in conjunction with the `maxscroll` property. This property can be read or written.

The `maxscroll` text field property specifies the maximum value allowed for the `scroll` text field property. It serves as a value that you can use to ensure that the `scroll` property is not assigned a value larger than the number of the last line in the text field. This property can only be read.

## trace() Global Function

```
trace(expression)
```

### Description

The `trace()` global function evaluates *expression* and outputs the results as a string to the Script Console window followed by a newline character. Used for debugging.



`trace()` is only useful from within LiveMotion's Preview mode. You can display similar results to a text field of the executing SWF file using the following code, where *display* is the variable (var=) name of your text field:

```
_root.display = expression;
```

### Parameters

<i>expression</i>	The expression to evaluate. It needs to result in a string, or something that can be converted to a string.
-------------------	-------------------------------------------------------------------------------------------------------------

### Example

`trace()` is used extensively for output in the examples of this reference chapter.

```
trace(this); //prints MovieClip (primitive type)
trace(2 * 2); //prints 4
trace("Monica and Billy were here."); //prints "Monica and Billy were here."
```

## unescape() Global Function

`unescape(stringExpression)`

### Description

The `unescape()` global function translates the encoded string *stringExpression* into a regular string. In *stringExpression*, characters that required encoding were replaced with the format `%xx`, where *xx* is the hexadecimal value of the character. This type of encoding is basically URL encoding except that spaces are replaced with `%20` instead of a `+` sign. Use the `escape()` global function to encode strings.

### Parameters

<i>stringExpression</i>	A string encoded with the <code>escape()</code> global function.
-------------------------	------------------------------------------------------------------

### Returns

A regular string version of *stringExpression*.

### Example

```
//prints "Billy went fishing!#?!"
trace(unescape("Billy%20went%20fishing%21%24%23%21"));
```

### See also

"`escape()` Global Function" on page 147

## unloadMovie() Global Function

`unloadMovie(target)`

## Description

The `unloadMovie()` global function unloads the SWF file from *target* that was previously loaded using the `loadMovie()` global function, the `loadMovieNum()` global function, or the `movieClip.loadMovie()` method.

When a SWF file is unloaded from an existing movie clip, the contents of the movie clip are unloaded, but the movie clip handlers are not. These include `onEnterFrame`, `onLoad`, `onUnload`, `onData`, `onMouseDown`, `onMouseUp`, `onMouseMove`, `onKeyDown`, and `onKeyUp`. Everything else—including button handlers, state scripts, and objects—are removed from the movie clip “shell.” This movie clip shell concept is important to keep in mind because it means that, when using `loadMovie()` and `unloadMovie()`, a movie clip instance is never really removed from the composition. Movie clip content is simply moved in and out of the shell with `loadMovie()` and `unloadMovie()`.

## Parameters

<i>target</i>	A path or reference to a level of the player or an existing movie clip.
---------------	-------------------------------------------------------------------------

## See also

“loadMovie() Global Function” on page 162, “loadMovieNum() Global Function” on page 163, “unloadMovieNum() Global Function” on page 226, “MovieClip.loadMovie() Method” on page 185, “MovieClip.unloadMovie() Method” on page 191

# unloadMovieNum() Global Function

`unloadMovieNum(number)`

## Description

Same as `unloadMovie()` except that a number is used to specify the player level. Therefore, it can only be used to unload SWF files previously loaded using the `loadMovie()` global function or the `loadMovieNum()` global function.

## Parameters

<i>number</i>	A non-negative integer specifying the level of the player containing the SWF file to unload.
---------------	----------------------------------------------------------------------------------------------

## See also

“loadMovie() Global Function” on page 162, “loadMovieNum() Global Function” on page 163, “unloadMovie() Global Function” on page 225, “MovieClip.loadMovie() Method” on page 185, “MovieClip.unloadMovie() Method” on page 191

# updateAfterEvent() Global Function

`updateAfterEvent()`

## Description

The `updateAfterEvent()` global function is used to update the display when one of the following events occurs: `onMouseMove`, `onMouseDown`, `onMouseUp`, `onKeyDown`, `onKeyUp`. Place this function in the appropriate event handler to cause refresh to occur.

## XML Object

### Description

The `XML` object enables you to load, parse, send, build, and manipulate eXtensible Markup Language (XML) document trees. Unlike HTML, which uses a defined set of tags, XML allows you to define your own document tags. LiveMotion allows you to either build an XML document from scratch or read in and modify an existing XML document.

The following shows three levels of child nodes (the document itself is the parent):

```
<fish>//level 1 child node
 <type>Bass</type>//"type" tag is level 2 child node; "Bass" is
level 3
</fish>
```

For example, the following creates an XML document:

```
xmlDocument = new XML("<fish><type>Bass</type></fish>");
```

The text can then be accessed as follows:

```
//prints "Bass"
trace(xmlDocument.firstChild.firstChild.firstChild.nodeValue);
```

### Constructor

```
new XML()
new XML(source)
```

### Parameters

<i>source</i>	(Optional) Source XML document. If not provided, the XML object will contain a new, empty XML document.
---------------	---------------------------------------------------------------------------------------------------------

### Properties

<code>attributes</code>	See "XML.attributes Property" on page 229.	Object whose properties store the attributes defined by the node.
<code>childNodes</code>	See "XML.childNodes Property" on page 230.	Array of child nodes of node.
<code>contentType</code>	See "XML.contentType Property" on page 231.	MIME content type.

<code>docTypeDecl</code>	See “XML.docTypeDecl Property” on page 232.	DOCTYPE declaration of the XML document.
<code>firstChild</code>	See “XML.firstChild Property” on page 233.	First child of the node, <code>null</code> if there are no children.
<code>ignoreWhite</code>	See “XML.ignoreWhite Property” on page 234.	Whether to ignore whitespace during XML parsing.
<code>lastChild</code>	See “XML.lastChild Property” on page 234.	Last child of the node, <code>null</code> if there are no children.
<code>loaded</code>	See “XML.loaded Property” on page 235.	<code>true</code> if the <code>load()</code> or <code>sendAndLoad()</code> operation has completed.
<code>nextSibling</code>	See “XML.nextSibling Property” on page 236.	Next sibling of the node, <code>null</code> if this node is the last node.
<code>nodeName</code>	See “XML.nodeName Property” on page 236.	Tag name of the node. <code>null</code> if this node is a text node.
<code>nodeType</code>	See “XML.nodeType Property” on page 236.	Type of the node. Either <code>1</code> if the node is an element node, or <code>3</code> if the node is a text node.
<code>nodeValue</code>	See “XML.nodeValue Property” on page 236.	Text contained in the node. <code>null</code> if the node is not a text node.
<code>parentNode</code>	See “XML.parentNode Property” on page 238.	Parent node of the node. <code>null</code> if the node is at the top of the hierarchy.
<code>previousSibling</code>	See “XML.previousSibling Property” on page 239.	Previous sibling of the node, <code>null</code> if the node is the first node.
<code>status</code>	See “XML.status Property” on page 240.	Whether there was an error parsing the XML document. <code>0</code> indicates no errors.
<code>xmlDecl</code>	See “XML.xmlDecl Property” on page 242.	DOCTYPE declaration of the XML document.

## Methods

<code>appendChild()</code>	See “XML.appendChild() Method” on page 229.	Append a child to the node.
<code>cloneNode()</code>	See “XML.cloneNode() Method” on page 230.	Clone the node.
<code>createElement()</code>	See “XML.createElement() Method” on page 231.	Create an XML element node.
<code>createTextNode()</code>	See “XML.createTextNode() Method” on page 232.	Create an XML text node.
<code>hasChildNodes()</code>	See “XML.hasChildNodes() Method” on page 233.	Return an indication whether the node has children.
<code>insertBefore()</code>	See “XML.insertBefore() Method” on page 234.	Insert a child node before another child node.

<code>load()</code>	See “XML.load() Method” on page 235.	Load and parse an XML document from the given URL.
<code>parseXML()</code>	See “XML.parseXML() Method” on page 238.	Parse the given text as an XML document.
<code>removeNode()</code>	See “XML.removeNode() Method” on page 239.	Delete the node and all of its children from the containing document.
<code>send()</code>	See “XML.send() Method” on page 240.	Convert the XML document into a string and send it to the given URL.
<code>sendAndLoad()</code>	See “XML.sendAndLoad() Method” on page 240.	Convert the XML document into a string and send it to the given URL. The receiving application is to reply with an XML document.
<code>toString()</code>	See “XML.toString() Method” on page 241.	Convert the XML object into a string.

## Event Handlers

<code>onData</code>	See “XML.onData() Event Handler” on page 237.	Indicates that the XML document parsing can begin.
<code>onLoad</code>	See “XML.onLoad() Event Handler” on page 237.	Indicates that the load of an XML document completed successfully.

## XML.appendChild() Method

`node.appendChild(childNode)`

### Description

The `appendChild()` method appends an existing XML node to `node` as its last child.

### Parameters

*childNode*      An existing XML node to append to `node` as a child.

### See also

“XML.createElement() Method” on page 231, “XML.createTextNode() Method” on page 232, “XML.cloneNode() Method” on page 230, “XML.insertBefore() Method” on page 234

## XML.attributes Property

`node.attributes`

### Description

The `attributes` property stores the names and values of attributes defined by `node`. This property can be read or written.

For example, in the following line of code, `name` is an attribute and `value` is the value of that attribute:

```
<testtag name=\"value\">Bass</testtag>
```

### See also

“XML.nodeType Property” on page 236

## XML.childNodes Property

```
node.childNodes[n]
```

### Description

The `childNodes` property holds an array of child nodes of `node`. Each element `n` in the array is a reference to a child node. Use the methods `appendChild()`, `insertBefore()`, and `removeNode()` to manipulate child nodes. This property can only be read.

### Example

```
xmlDocument = new XML("<fish><type>Bass</type><color>grey</color></fish>");

trace(xmlDocument.childNodes[0].childNodes[0].nodeValue);//prints
"type"

trace(xmlDocument.childNodes[0].childNodes[1].nodeValue);//prints
"color"
```

### See also

“XML.firstChild Property” on page 233, “XML.hasChildNodes() Method” on page 233, “XML.lastChild Property” on page 234, “XML.nextSibling Property” on page 236, “XML.previousSibling Property” on page 239, “XML.appendChild() Method” on page 229, “XML.insertBefore() Method” on page 234, “XML.removeNode() Method” on page 239

## XML.cloneNode() Method

```
node.cloneNode(deep)
```

### Description

The `cloneNode()` method clones `node` and, optionally, all of its children.

### Parameters

<i>deep</i>	A boolean indicating whether a deep clone (all of the node’s children as well as <i>node</i> ) should be performed. If <code>true</code> , a deep clone is performed. If <code>false</code> , only <i>node</i> is cloned.
-------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Returns

The cloned node and, if *deep* is `true`, all of its children.

### Example

```
xmlDocument = new XML("<fish><type>Bass</type></fish>");
```

```
newDocument = new XML();
node = xmlDocument.firstChild.cloneNode(true);
newDocument.appendChild(node);
trace(newDocument.firstChild.nodeValue); //prints "fish"
```

**See also**

"XML.appendChild() Method" on page 229, "XML.createElement() Method" on page 231, "XML.createTextNode() Method" on page 232, "XML.insertBefore() Method" on page 234

## XML.contentType Property

```
root.contentType
```

**Description**

The `contentType` property holds the MIME content type. The MIME type is sent to the server when either the `send()` or `sendAndLoad()` methods are used. Only available on the root node of the document. This property can be read or written. The default is `application/x-www-form-urlencoded`.

**See also**

"XML.send() Method" on page 240, "XML.sendAndLoad() Method" on page 240

## XML.createElement() Method

```
root.createElement(tagName)
```

**Description**

The `createElement()` method creates a new element, or tag, node (not a text node). Only available on the root node of the document. The new node has no parent and no children. Note that the new node that is returned is not appended to `root`. To do that, you must use `appendChild()`.

As an example of a tag node, examine the line:

```
<type>Bass</type>
```

`type` is a tag node, whereas `Bass` is the associated text node.

**Parameters**

<i>tagName</i>	The tag name of the node to create.
----------------	-------------------------------------

**Returns**

The new tag node.

**Example**

```
xmlDocument = new XML();
node = xmlDocument.createElement("fish");
xmlDocument.appendChild(node);
```

```
trace(xmlDocument.firstChild.nodeValue);//prints "fish"
```

**See also**

“XML.appendChild() Method” on page 229, “XML.cloneNode() Method” on page 230, “XML.createTextNode() Method” on page 232, “XML.insertBefore() Method” on page 234,

## XML.createTextNode() Method

```
root.createTextNode(text)
```

**Description**

The `createTextNode()` method creates a text node (as opposed to an element, or tag, node). Only available on the root node of the document. The new node has no parent and no children. Note that the new node that is returned is not appended to `root`. To do that, you must use `appendChild()`.

As an example of a text node, examine the line:

```
<type>Bass</type>
```

`type` is a tag node, whereas `Bass` is the associated text node.

**Parameters**

<i>text</i>	The text of the node to create.
-------------	---------------------------------

**Returns**

The new text node.

**Example**

```
xmlDocument = new XML();
node = xmlDocument.createElement("fish");
xmlDocument.appendChild(node);
textString = xmlDocument.createTextNode("Bass");
xmlDocument.firstChild.appendChild(textString);
trace(xmlDocument.firstChild.nodeValue);//prints "fish"
trace(xmlDocument.firstChild.firstChild.nodeValue);//prints "Bass"
```

**See also**

“XML.appendChild() Method” on page 229, “XML.cloneNode() Method” on page 230, “XML.createElement() Method” on page 231, “XML.insertBefore() Method” on page 234

## XML.docTypeDecl Property

```
root.docTypeDecl
```



## Description

The `docTypeDecl` property specifies the DOCTYPE declaration of the XML document. If there is no DOCTYPE, then this property is `undefined`. Only available on the root node of the document. This property can be read or written.

## Example

```
xmlDocument = new XML("<fish><type>Bass</type><color>grey</color></fish>");
xmlDocument.docTypeDecl = "<!DOCTYPE salutation SYSTEM \"hello.dtd\">";
trace(xmlDocument.docTypeDecl);
//prints "<!DOCTYPE salutation SYSTEM \"hello.dtd\">"
```

## See also

"XML.xmlDecl Property" on page 242

# XML.firstChild Property

`node.firstChild`

## Description

The `firstChild` property specifies the first child of `node`, or `null` if there are no children. This property can only be read.

## Example

```
xmlDocument = new XML("<fish><type>Bass</type></fish>");
trace(xmlDocument.firstChild.nodeValue); //prints "fish"
```

## See also

"XML.childNodes Property" on page 230, "XML.lastChild Property" on page 234, "XML.nextSibling Property" on page 236, "XML.previousSibling Property" on page 239

# XML.hasChildNodes() Method

`node.hasChildNodes()`

## Description

The `hasChildNodes()` method returns an indication of whether `node` has children.

## Returns

`true` if `node` has children; `false` otherwise.

## Example

```
xmlDocument = new XML("<fish><type>Bass</type></fish>");
if (xmlDocument.hasChildNodes())
 trace("yes"); //prints "yes"
else
 trace("no");
```

**See also**

“XML.childNodes Property” on page 230

## XML.ignoreWhite Property

`root.ignoreWhite`

**Description**

The `ignoreWhite` property stores a boolean that indicates whether to ignore whitespace during XML parsing. Only available on the root node of the document. The default is `false`. This property can only be read.

**Note:** Previous to release 41 of the Netscape Flash Player plug-in and release 42 of the Internet Explorer Flash Player plug-in, the Flash 5 Player treated whitespace (carriage returns, tabs, spaces) as nodes. The `ignoreWhite` property is supported in later releases. If your XML code needs to run on earlier versions of the Flash 5 Player, you will need to include code that strips out whitespace from incoming XML documents.

```
temp = new Boolean(true);
trace(temp.valueOf()); //prints "true"
xmlDocument = new XML("<fish><type>Bass</type></fish>");
temp = xmlDocument.ignoreWhite;
trace(temp.valueOf()); //prints "false"
```

## XML.insertBefore() Method

`node.insertBefore(newChild, insertBeforeChild)`

**Description**

The `insertBefore()` method inserts a new child node before an existing child node in the hierarchy.

**Parameters**

<i>newChild</i>	An existing XML node to add as a child to <i>node</i> before <i>insertBeforeChild</i> in the hierarchy.
<i>insertBeforeChild</i>	The child to insert <i>newChild</i> before in <i>node</i> 's child list.

**See also**

“XML.appendChild() Method” on page 229

## XML.lastChild Property

`node.lastChild`

**Description**

The `lastChild` property holds the last child of *node*, or `null` if there are no children. It is equivalent to `childNodes[childNodes.length-1]`. This property can only be read.

### Example

```
xmlDocument = new XML("<color>white</color><color2>grey</color2>");
trace(xmlDocument.lastChild.nodeValue); //prints "color2"
```

### See also

"XML.childNodes Property" on page 230, "XML.firstChild Property" on page 233,  
"XML.nextSibling Property" on page 236, "XML.previousSibling Property" on page 239

## XML.load() Method

```
root.load(url)
```

### Description

The `load()` method loads and parses an XML document from `url` into `root`. Only available on the root node of the document. The load doesn't happen immediately. Use the `root.onLoad()` event handler for code to execute when the document has finished downloading. The loaded document replaces the contents of `root` with the downloaded XML data. When `load()` is first executed, the `loaded` property is set to `false`; then, when the download is complete, the `loaded` property is set to `true` and the root node's `onLoad()` event handler is called. The XML data is not parsed until the entire document is loaded. The parsing may be done using the default parser, or the `root.onData()` event handler may be used to write your own parser.

### Parameters

<code>url</code>	A string specifying the URL of the document to load and parse. Its XML hierarchy is placed into <code>root</code> . For security reasons, the URL must be in the same domain as that from which the movie clip was downloaded.
------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### See also

"XML.loaded Property" on page 235, "XML.onLoad() Event Handler" on page 237,  
"XML.sendAndLoad() Method" on page 240, "XML.status Property" on page 240,  
"XML.onData() Event Handler" on page 237, "XML.parseXML() Method" on page 238

## XML.loaded Property

```
root.loaded
```

### Description

The `loaded` property holds `true` if the `load()` or `sendAndLoad()` operation has completed. Otherwise it holds `false`. Only available on the root node of the document. This property can only be read.

### See also

"XML.load() Method" on page 235, "XML.onLoad() Event Handler" on page 237,  
"XML.sendAndLoad() Method" on page 240, "XML.status Property" on page 240

## XML.nextSibling Property

*node*.nextSibling

### Description

The `nextSibling` property holds a reference to the next node in the same level of the XML object hierarchy, or `null` if *node* is the last node. This property can only be read.

### Example

```
xmlDocument = new XML("<color>white</color><color2>grey</color2>");
tempNode = xmlDocument.childNodes[0];
trace(tempNode.firstChild.nodeValue); //prints "white"
tempNode = tempNode.nextSibling;
trace(tempNode.firstChild.nodeValue); //prints "grey"
```

### See also

"XML.childNodes Property" on page 230, "XML.firstChild Property" on page 233, "XML.lastChild Property" on page 234, "XML.nodeName Property" on page 236, "XML.nodeValue Property" on page 236, "XML.previousSibling Property" on page 239

## XML.nodeName Property

*node*.nodeName

### Description

The `nodeName` property holds the tag name of *node*, or `null` if *node* is a text node. If the tag is `<mynode>` then the `nodeName` is `mynode`. This property can be read or written.

### See also

"XML.nodeType Property" on page 236, "XML.nodeValue Property" on page 236

## XML.nodeType Property

*node*.nodeType

### Description

The `nodeType` property holds the type of *node*. The possible values are `1` if this node is an element node, or `3` if this node is a text node. This property can only be read.

### See also

"XML.nodeName Property" on page 236, "XML.nodeValue Property" on page 236

## XML.nodeValue Property

*node*.nodeValue

## Description

The `nodeValue` property holds the text contained in `node`, or `null` if `node` is an element node. This property can be read or written, though writing to it only makes sense if the node is a text node.

## See also

“XML.nodeName Property” on page 236, “XML.nodeType Property” on page 236

# XML.onData() Event Handler

`root.onData(source)`

## Description

The `onData()` user-defined event handler executes automatically whenever raw XML source has finished loading into the XML document due to a previous `root.load()` or `root.sendAndLoad()` call, but before the XML has been parsed. This allows you to write a custom function that handles the raw XML, or you can simply let the default XML parser execute on the raw XML. This event handler should only be defined if you want to do the XML parsing yourself. It is only available on the root node of the document.

If the raw source that is received is undefined, the `onData()` event handler calls the `root.onLoad()` event handler with the `success` parameter set to `false`. Otherwise, the `onData()` event handler parses the raw XML, sets the `root.loaded` property to `true`, and calls the `root.onLoad()` event handler with the `success` parameter set to `true`.

## Parameters

<code>source</code>	A string with the raw XML source.
---------------------	-----------------------------------

## Example

This example shows how to intercept the raw XML using the `onData()` event handler. It uses a function literal.

```
xmlDocument = new XML();
xmlDocument.onData = function(source)
{
 trace("Print the raw XML: \n" + source);
};
```

## See also

“XML.onLoad() Event Handler” on page 237, “XML.load() Method” on page 235, “XML.sendAndLoad() Method” on page 240, “XML.loaded Property” on page 235

# XML.onLoad() Event Handler

`root.onLoad(success)`

## Description

The `onLoad()` user-defined event handler is automatically executed whenever an external XML file is loaded into `root` via the `root.load()` or `root.sendAndLoad()` method. By default, the `onLoad()` event handler is an empty function: you must provide your own callback handler, as shown in the example. The `onLoad()` event handler is only available on the root node of the document and it offers an alternative to monitoring the state to the `root.loaded` property before proceeding with processing the downloaded XML.

## Parameters

<i>success</i>	A boolean indicating success ( <code>true</code> ) or failure ( <code>false</code> ) of the <code>root.load()</code> or <code>root.sendAndLoad()</code> method.
----------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------

## Example

```
xmlDocument = new XML();
xmlDocument.onLoad = xmlProcessor;
xmlDocument.load("myFile.xml");
function xmlProcessor(success)
{
 //function body
}
```

## See also

“XML.onData() Event Handler” on page 237, “XML.load() Method” on page 235,  
“XML.sendAndLoad() Method” on page 240, “XML.loaded Property” on page 235

## XML.parentNode Property

`node.parentNode`

## Description

The `parentNode` property holds the parent node of `node`, or `null` if `node` is at the top of the hierarchy. This property can only be read.

## See also

“XML.childNodes Property” on page 230, “XML.firstChild Property” on page 233,  
“XML.lastChild Property” on page 234, “XML.previousSibling Property” on page 239

## XML.parseXML() Method

`root.parseXML(source)`

## Description

The `parseXML()` method parses *source* as an XML document. It replaces any existing XML in *root* with the resulting XML tree from *source*. Only available on the root node of the document. This method is similar to the `load()` method, but the source is passed in as a string so can be used, for example, to pass in user input rather than just the contents of a URL or file.

## Parameters

<i>source</i>	The string to parse.
---------------	----------------------

## See also

“XML.load() Method” on page 235, “XML.status Property” on page 240

## XML.previousSibling Property

*node*.previousSibling

## Description

The `previousSibling` property holds a reference to the previous node in the same level of the XML object hierarchy, or `null` if *node* is the first node. This property can only be read.

## Example

```
xmlDocument = new XML("<color>white</color><color2>grey</color2>");
tempNode = xmlDocument.childNodes[1];
trace(tempNode.firstChild.nodeValue); //prints "grey"
tempNode = tempNode.previousSibling;
trace(tempNode.firstChild.nodeValue); //prints "white"
```

## See also

“XML.childNodes Property” on page 230, “XML.firstChild Property” on page 233, “XML.lastChild Property” on page 234, “XML.nextSibling Property” on page 236, “XML.nodeName Property” on page 236, “XML.nodeValue Property” on page 236, “XML.parentNode Property” on page 238

## XML.removeNode() Method

*node*.removeNode()

## Description

The `removeNode()` method deletes *node* and all of its children from the containing document.

## See also

“XML.appendChild() Method” on page 229

## XML.send() Method

```
root.send(url, window)
```

### Description

The `send()` method converts `root` into a string of XML source and sends it as an HTTP request to `url`. The response data is usually an HTML file for display in a browser window; this contrasts with the `sendAndLoad()` method, which receives XML for display directly from the movie clip. Only available on the root node of the document.

### Parameters

<code>url</code>	The URL to which to send the XML text.
<code>window</code>	A string indicating the window in which to display data returned by the server. This may be a custom name or one of the standard JavaScript windows ( <code>_blank</code> , <code>_parent</code> , <code>_self</code> , or <code>_top</code> ).

### See also

“XML.sendAndLoad() Method” on page 240, “XML.load() Method” on page 235

## XML.sendAndLoad() Method

```
root.sendAndLoad(url, responseXML)
```

### Description

The `sendAndLoad()` method converts `root` into a string and sends it as an HTTP request to `url`. The receiving application is supposed to reply with an XML document, which is parsed as XML source and loaded into `responseXML`; this contrasts with the `send()` method, which receives an HTML file for display in a browser window. Only available on the root node of the document.

### Parameters

<code>url</code>	The URL to which to send the XML text. For security reasons, the URL must be in the same domain as that from which the movie clip was downloaded.
<code>responseXML</code>	The XML object into which to load the response.

### See also

“XML.load() Method” on page 235, “XML.loaded Property” on page 235, “XML.send() Method” on page 240, “XML.status Property” on page 240, “XML.onData() Event Handler” on page 237, “XML.onLoad() Event Handler” on page 237

## XML.status Property

```
root.status
```



## Description

The `status` property holds an integer that indicates whether there was an error parsing the XML document. Only available on the root node of the document. This property can only be read. The possible error codes are:

- 0 — No error; parsing 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 properly matched with an end tag.
- -10 — An end tag was not properly matched with a start tag.

Parsing occurs in several instances: when an `XML` object is first instantiated using the `XML` constructor, when an `XML` object is loaded using the `load()` or `sendAndLoad()` method, or XML is passed for parsing to the `parseXML()` method. Before checking the value of this property, check the `loaded` property to ensure that the `load()` or `sendAndLoad()` method has completed successfully.

## See also

“XML.load() Method” on page 235, “XML.loaded Property” on page 235, “XML.onLoad() Event Handler” on page 237, “XML.parseXML() Method” on page 238, “XML.sendAndLoad() Method” on page 240,

## XML.toString() Method

`node.toString()`

### Description

The `toString()` method converts `node` into a string and returns it.

### Returns

A string that is the XML source code equivalent of `node`.

### Example

```
xmlDocument = new XML("<color>white</color><color2>grey</color2>");
trace(xmlDocument.toString());
//displays "<color>white</color><color2>grey</color2>"
```

## See also

“Object.toString() Method” on page 200, “XML.nodeValue Property” on page 236

## XML.xmlDecl Property

`root.xmlDecl`

### Description

The `xmlDecl` property is a string that holds the XML declaration tag of the XML document. It is only available on the root node of the document and is used to identify the version of XML being used in the document. This property can be read or written.

### Example

```
xmlDocument = new XML("<?xml version=\"1.0\"?><type>Bass</type>");
trace(xmlDocument.xmlDecl);
//prints "<?xml version="1.0"?>"
```

### See also

“XML.docTypeDecl Property” on page 232

## XMLnode Object

### Description

The `XMLnode` object is the base class defining core properties and methods of nodes in an XML object hierarchy. Few programmers will need to access this object, but it is possible to use it to extend the default functionality of XML objects.

## XMLSocket Object

### Description

The `XMLSocket` object is used to implement a client socket that allows the Flash Player to communicate with a server via an “open” connection. A socket connection is useful because it remains “open”—that is, a TCP/IP connection doesn’t have to be made between the client and the server each time communications occur between the two, as is required when the HTTP protocol is used. This enables the Flash Player to listen for incoming messages and quickly process them; it also allows it to respond quickly.

### Constructor

```
new XMLSocket()
```

### Parameters

None.

### Properties

None.

## Methods

<code>close()</code>	See “XMLSocket.close() Method” on page 243.	Close an open socket connection.
<code>connect()</code>	See “XMLSocket.connect() Method” on page 243.	Create a connection to a specified server.
<code>send()</code>	See “XMLSocket.send() Method” on page 247.	Send an XML object to the server.

## Event Handlers

<code>onClose()</code>	See “XMLSocket.onClose() Event Handler” on page 244.	Callback function that is called when a connection has closed.
<code>onConnect()</code>	See “XMLSocket.onConnect() Event Handler” on page 245.	Callback function that is called when a connection is created.
<code>onData()</code>	See “XMLSocket.onData() Event Handler” on page 246.	Callback function that is called when data is received but has not yet been parsed as XML.
<code>onXML()</code>	See “XMLSocket.onXML() Event Handler” on page 246.	Callback function that is called when data has been received and parsed into an XML object hierarchy.

## XMLSocket.close() Method

`socket.close()`

### Description

The `close()` method closes an open socket connection.

### See also

“XMLSocket.connect() Method” on page 243, “XMLSocket.onClose() Event Handler” on page 244

## XMLSocket.connect() Method

`socket.connect(host, port)`

### Description

The `connect()` method creates a connection to a specified server. If this method returns `true`, then the `onConnect()` event handler is invoked to complete the connection.

## Parameters

<i>host</i>	A full DNS name or an IP address. <code>null</code> if you want to specify the current server (where the currently executing SWF file was downloaded from). For security reasons, if the Netscape SWF plug-in or an ActiveX control is being used, the host must have the same domain name as the host from which the SWF file was downloaded.
<i>port</i>	The TCP port to which you wish to establish a connection. Must be a number equal to or greater than 1024.

## Returns

`true` if a connection is successfully created; `false` otherwise.

## Example

```
function socketConnect(success)
{
 if (success)
 {
 trace("Full connection achieved");
 //other code
 }
}

newSocket = new XMLSocket();
newSocket.onConnect = socketConnect;
if (newSocket.connect("http://www.adobe.com", 2000))
{
 trace("Initial connection achieved");
 //other code
}
```

## See also

"XMLSocket.close() Method" on page 243, "XMLSocket.onConnect() Event Handler" on page 245

## XMLSocket.onClose() Event Handler

```
socket.onClose = functionName
socket.functionName()
```

## Description

The `onClose()` user-defined callback function is called when a connection is closed by the server. The default implementation of this method performs no action. To override the default implementation, you must write your own handler, as shown in the example.

## Parameters

*functionName*                      The name of the function to call when the indicated connection has closed.

## Example

```
newSocket = new XMLSocket();
newSocket.onClose = socketClosed;
function socketClosed()
{
 trace("The connection was closed by the server");
}
```

## See also

“XMLSocket.close() Method” on page 243

# XMLSocket.onConnect() Event Handler

```
socket.onConnect = functionName
socket.functionName(success)
```

## Description

The `onConnect()` user-defined callback function is called when a connection is created. The default implementation of this method performs no action. To override the default implementation, you must write your own handler, as shown in the example.

## Parameters

*success*                              A boolean indicating success (`true`) or failure (`false`).

*functionName*                      The name of the function to call when the connection is created.

## Returns

`true` if a connection is successfully created; `false` otherwise.

## Example

```
function socketConnect(success)
{
 if (success)
 {
 trace("Full connection achieved");
 //other code
 }
}
```

```
newSocket = new XMLSocket();
newSocket.onConnect = socketConnect;
if (newSocket.connect("http://www.adobe.com", 2000))
{
 trace("Initial connection achieved");
 //other code
}
```

**See also**

“XMLSocket.connect() Method” on page 243

## XMLSocket.onData() Event Handler

*socket.onData(source)*

**Description**

The `onData()` user-defined callback function is called when data is received but has not yet been parsed. The `onData()` event handler executes automatically whenever a zero byte (ASCII null character) is transmitted to the player over *socket*. This allows you to write a function that handles the raw XML instead of the default parser that would otherwise be used before the XML is passed onto the `socket.onXML()` event handler. If you have not supplied `onData()` with a custom callback function, the XML is passed onto the default XML parser, and then `socket.onXML()` is called with the result.

**Parameters**

<i>source</i>	A string with the raw XML source.
---------------	-----------------------------------

**Example**

The following shows how to implement the `onData()` event handler using a function literal.

```
newSocket = new XMLSocket();
newSocket.onData = function(source)
{
 trace("Print the raw XML: \n" + source);
};
```

**See also**

“XMLSocket.onXML() Event Handler” on page 246; “XML.onData() Event Handler” on page 237

## XMLSocket.onXML() Event Handler

*socket.onXML = functionName*  
*socket.functionName(object)*

## Description

The `onXML()` user-defined callback function is called when data has been received and parsed into an XML object hierarchy. It has been parsed either by the default parser or by a custom `onData()` event handler. The default implementation of this method performs no action. To override the default implementation, you must write your own handler.

## Parameters

<i>object</i>	An XML object containing a parsed XML document that was received from the server.
<i>functionName</i>	The name of the function to call when data has been received and parsed into an XML object hierarchy.

## See also

“XMLSocket.send() Method” on page 247, “XMLSocket.onData() Event Handler” on page 246

# XMLSocket.send() Method

*socket.send(object)*

## Description

The `send()` method converts *object* to a string and sends it to the server over the *socket* connection, followed by a zero byte (ASCII null character). This operation is asynchronous: the `send()` is initiated, but the operating system and networking software may not complete the transmission until some amount of time has passed.

## Parameters

<i>object</i>	The XML object to send.
---------------	-------------------------

## See also

“XMLSocket.onXML() Event Handler” on page 246, “XMLSocket.send() Method” on page 247

# Glossary

---

## Glossary Terms

**Absolute reference** Reference that uses `_root` as the starting point of the address to a movie clip. The address is a string of movie clip names delimited by dot (.) notation representing each level in the object hierarchy from `_root` down to and including the name of movie clip being referenced. The absolute reference is the same regardless of where in the object hierarchy the source movie clip that is making the reference is located. An example of an absolute reference is: `_root.movieClipA.movieClipB._x`

**Anchor point** Point that represents the 0,0 (x,y) origin point for all coordinates in a movie clip. For a movie clip group with multiple objects, the anchor point is set to the center of the group.

**Animation** Changes applied to an object over time.

**Button** Movie clip that has a button event handler or has had states added to it by the user.

**Composition** Refers to a `.liv` file that is created in LiveMotion.

**Composition timeline** Main timeline of a composition; also referred to as `_root`'s timeline.

**Composition window** Window in the LiveMotion user interface that displays objects as they are created and edited. The objects are displayed as they appear at the current time, which is determined by the current-time marker in the Timeline window. The Composition window also displays the results of previewing a composition.

**DOM** Document Object Model. All the objects, their methods, and properties that are supported by LiveMotion as extensions to the JavaScript core.

**Event** User interaction, such as pressing a key or dragging the mouse, or system interaction, such as loading a movie clip.

**Filename.liv** Document created in the LiveMotion application using LiveMotion's interface tools, palettes, and (optional) scripting code; also referred to as a *composition*.

**Interactivity** Result of a user event such as pressing a button or moving the mouse over an object in a composition or a system event such as loading a movie clip. The event triggers an event handler that performs a response when the event occurs.

**Keyframe script** Script added to a frame in a timeline.

**Label** String identifier that references a frame in a timeline.

**Movie clip** Copy of the MovieClip object that has its own timeline and unique name and can be manipulated by writing scripts.

**Movie clip group** Parent movie clip containing one or more nested objects.

**Parent** Timeline upon which a movie clip or movie clip group is created.





**Path** Reference enclosed in quotation marks. An example of a path is:

```
"_root.movieClipA.movieClipB._x"
```

**Relative reference** Movie clip names delimited by dot (.) notation that “navigate” through the object hierarchy and include the name of each movie clip from the source movie clip to the movie clip it is referencing. The contents of a relative reference are determined by the hierarchical relationship of the source movie clip to the movie clip it is addressing. Although using the keyword `this` is optional in the relative reference, this scripting guide begins all relative references with `this`. An example relative reference is:

```
this._parent.movieClipA.movieClipB._x
```

**Script keyframe** Keyframe to which a script is added.

**Siblings** Movie clips on the timeline of the same parent.

**Source** Movie clip that is controlling another movie clip by calling MovieClip methods or manipulating MovieClip properties.

**SWF** File format into which a LiveMotion composition is converted on export on export to Macromedia Flash format. SWF files can be viewed with the Flash Player or a Web browser with the Flash plug-in.

# Legal Notices

## Copyright

© 2002 Adobe Systems Incorporated. All rights reserved.

Adobe® LiveMotion™ 2.0 Scripting Guide for Windows® and Macintosh®

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide. Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner. Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, Acrobat Reader, Adobe Dimensions, Adobe Premiere, Adobe Type Manager, After Effects, AlterCast, Classroom in a Book, FrameMaker, GoLive, Illustrator, InDesign, LiveMotion, Minion, Myriad, PageMaker, Photoshop, PostScript, PostScript 3, Reader, Streamline, Type Reunion are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Microsoft, OpenType, Windows, and Windows NT are registered trademarks of Microsoft Corporation in the U.S. and/or other countries. Apple, Macintosh, Power Macintosh, QuickTime, the QuickTime logo, and TrueType are trademarks of Apple Computer, Inc. registered in the U.S. and other countries. QuickTime and the QuickTime logo are trademarks used under license. IBM and OS/2 are registered trademarks of International Business Machines Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. Pentium is a registered trademark of Intel Corporation. Macromedia and Flash are trademarks or registered trademarks of Macromedia, Inc. in the United States and/or other countries. Sun is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

MPEG Iyer-3 audio compression technology licensed by Fraunhofer IIS and THOMSON multimedia. Contains an implementation of the LZW algorithm licensed under U.S. Patent 4,558,302. Digital Imagery® copyright 2001 PhotoDisc, Inc.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.



# Index

## A

absolute reference [46](#)  
 ActionScript Syntax Helpers [88](#)  
 adding states [7](#)  
 attachSound() object method [61](#)  
 Automation syntax helper [15](#), [86](#)

## B

behaviors  
   mapping to scripts [30](#)  
 Behaviors button [27](#)  
 bounds checking [23](#)

## C

children [56](#)  
 clearing breakpoints [100](#)  
 composition [7](#)  
 Composition browser [15](#), [86](#), [89](#)  
 Console window  
   comparing output to Debugger [103](#)  
   using with Debugger [103](#)  
   writing to [102](#)  
 current-time marker [20](#)

## D

Debugger  
   activating [94](#)  
   Add variable [98](#)  
   buttons [96](#)  
   Call stack window [95](#)  
   expression entry field [98](#), [101](#)  
   halting execution [96](#)  
   Kill [97](#)  
   modes [94](#)  
   modes for bringing up [94](#)

Run [96](#)  
   setting breakpoints [100](#)  
   single-stepping [97](#)  
   Source window [96](#)  
   Step [97](#)  
   Step into [97](#)  
   Step out [98](#)  
   Stop [96](#)  
   terminating sessions [97](#)  
   using with Console window [102](#)  
   Variable window [95](#), [102](#)  
   watching variables [98](#), [101](#)  
   windows [95](#)  
 depth [58](#), [61](#)  
 Description window [16](#), [86](#)  
 dot (.) notation [45](#)

## E

event handlers  
   automatically generated [74](#)  
   button [71](#)  
   defined [21](#), [64](#)  
   key [68](#)  
   mouse [70](#)  
   system-based [65](#)  
 event types [64](#)  
 examples, list of hands-on [5](#)  
 Export [60](#)  
 exporting [7](#), [19](#)

## F

Find [15](#), [87](#), [93](#)  
 forms  
   creating [80](#)  
   sending and receiving variables [79](#), [80](#)  
   using text fields [77](#)



**G**

global function  
  fscommand() [148](#)  
global functions  
  Boolean() [120](#)  
  Date() [125](#)  
  duplicateMovieClip() [55](#), [146](#)  
  escape() [147](#)  
  eval() [148](#)  
  getTimer() [149](#)  
  getURL() [79](#), [149](#)  
  gotoAndPlay() [151](#)  
  gotoAndStop() [151](#)  
  isFinite() [152](#)  
  isNaN() [153](#)  
  lmFrameOfLabel() [161](#)  
  loadMovie() [63](#), [79](#), [162](#)  
  loadMovieNum() [163](#)  
  loadVariables() [79](#), [163](#)  
  loadVariablesNum() [164](#)  
  nextFrame() [194](#)  
  Number() [194](#)  
  parseFloat() [202](#)  
  parseInt() [202](#)  
  play() [203](#)  
  prevFrame() [203](#)  
  removeMovieClip() [204](#)  
  startDrag() [213](#)  
  stop() [213](#)  
  stopAllSounds() [213](#)  
  stopDrag() [214](#)  
  String() [214](#)  
  targetPath() [224](#)  
  that use \_leveln [63](#)  
  trace() [224](#)  
  unescape() [225](#)  
  unloadMovie() [63](#), [225](#)  
  unloadMovieNum() [226](#)  
  updateAfterEvent [226](#)  
global properties  
  \_focusrect [148](#)  
  \_leveln [63](#), [161](#)

  \_quality [204](#)  
  \_root [204](#)  
  \_soundbuftime [212](#)  
  -Infinity [152](#)  
  Infinity [152](#)  
  NaN [194](#)  
  newline [194](#)  
Go to Label (and play) [34](#)  
Go to Label (and stop) [34](#)  
Go to next script [15](#), [87](#), [91](#)  
Go to previous script [15](#), [87](#), [91](#)  
Go To Relative Time [34](#)

**H**

Handler scripts [15](#), [87](#), [91](#)  
hands-on examples  
  automatically generated button handlers [74](#)  
  changing movie clip states [32](#)  
  creating a bounds check [23](#)  
  creating a button event handler [72](#)  
  creating a preloader [37](#)  
  creating a simple event handler [22](#)  
  creating a state script [24](#)  
  creating a toggle button [73](#)  
  creating an onKeyDown event handler [69](#)  
  initializing a movie clip property [22](#), [23](#)  
  mouse trailer [51](#)  
  programmatic bounce [66](#)  
  using script keyframes [17](#)  
  using system-based event handlers [65](#)  
  writing a keyframe script to a movie clip timeline [19](#)  
hands-on examples, list of [5](#)  
hierarchy, movie clips [44](#)

**I**

independent timelines [27](#)  
initializing properties [22](#)

**J**

JavaScript  
  ECMA-standard [4](#), [8](#)

LiveMotion implementation of [9](#)  
JavaScript references [6](#)  
JavaScript Syntax Helpers [88](#)

## K

Keyframe scripts [15](#), [87](#), [92](#)  
keyframe scripts  
    on a movie clip timeline [19](#)

## L

labels [8](#), [36](#), [52](#)  
    creating [16](#)  
    defined [16](#)  
    guidelines for creating label names [16](#)  
    jump to [19](#)  
    label names [18](#)  
    names [19](#)  
    specifying as argument values [19](#)  
    string values [19](#)  
    using [16](#)  
\_levelN [63](#)  
.liv files [7](#), [27](#)

## M

Make Movie Clip Group command [43](#)  
modes, Debugger [94](#)  
Movie Clip command [43](#)  
movie clip events [64](#)  
Movie clip navigator [15](#), [86](#), [87](#)  
movie clips  
    \_root [44](#)  
    accessing shareable [61](#)  
    and movie clip groups [43](#)  
    attachMovie() method [55](#), [61](#)  
    built-in methods [42](#), [49](#)  
    built-in properties [42](#), [48](#)  
    creating manually [43](#)  
    creating methods [54](#)  
    creating programmatically [55](#)  
    creating properties [54](#)  
    defined [7](#), [42](#)  
    duplicateMovieClip() method [55](#)

events and handlers [21](#)  
hierarchy [20](#), [44](#), [51](#)  
hierarchy and the programmatic stack [58](#)  
placement of programmatically created [59](#)  
properties [48](#), [51](#)  
sharing [60](#)  
swapDepths() method [58](#)

## N

names, label [16](#)  
new operator [42](#)

## O

objects  
    Arguments [108](#)  
    Array [110](#)  
    Boolean [121](#)  
    Color [122](#)  
    Date [126](#)  
    Key [153](#)  
    Math [165](#)  
    Mouse [173](#)  
    MovieClip [174](#)  
    Number [195](#)  
    Object [199](#)  
    Selection [205](#)  
    Sound [207](#)  
    String [214](#)  
    XML [227](#)  
    XMLnode [242](#)  
    XMLSocketObject [242](#)  
objects, scriptable [17](#)

## P

parent-child relationship [44](#)  
placing scripts [14](#)  
Play [34](#)  
Preview mode [19](#)  
programmatic stacks [56](#)  
properties  
    creating movie clip [54](#)  
    initializing [22](#)

setting [8](#)

## R

relative reference [46](#)

\_root movie clip [44](#), [48](#), [49](#), [79](#)

## S

Script Editor

  buttons [86](#)

  setting breakpoints [99](#)

  window [85](#)

script keyframes [14](#), [30](#)

  accessing scripts [27](#)

  and timelines [26](#), [27](#)

  creating [17](#)

  defined [17](#)

  on the composition timeline [17](#)

Script window [16](#), [86](#)

Scripting helper window [16](#), [86](#)

Scripting syntax helper [15](#), [30](#), [86](#), [88](#)

scripts

  accessing [29](#)

  adding to states [24](#)

  adding to timelines [29](#)

  Change State [31](#)

  creating Flash Player commands [39](#)

  deleting [30](#)

  Go to Label (and play) [35](#)

  Go to Label (and stop) [35](#)

  Go to RelativeTime [34](#)

  Go to URL [40](#)

  locations of [8](#), [14](#)

  on event handlers [22](#)

  on states [24](#)

  opening [30](#)

  placing [14](#)

  Play [34](#)

  Run JavaScript [40](#)

  state [24](#)

  stop [34](#)

  Stop All Sounds [40](#)

  Wait For Download [36](#)

Scripts button [27](#)

Scripts Editor

  opening [20](#)

setting breakpoints

  in Script Editor [99](#)

  in the Debugger [100](#)

setting properties [8](#)

siblings [57](#)

single-stepping [97](#)

sound objects, accessing shareable [61](#)

state change events [64](#)

State scripts [15](#), [87](#), [92](#)

state scripts [24](#)

states

  and timelines [26](#)

  predefined [75](#)

  writing scripts to [27](#)

States palette [27](#), [30](#)

states, adding [7](#)

static stacks [56](#)

Stop [34](#)

SWF files [39](#), [56](#), [62](#), [63](#)

  loading [39](#)

  stacking order of [63](#)

  unloading [39](#)

Syntax highlighting [16](#), [87](#), [93](#)

## T

text fields

  creating and using [77](#)

  scroll and maxscroll properties [224](#)

this [20](#), [45](#), [46](#), [53](#)

time-independent [7](#)

toggle buttons, creating [73](#)

## X

XML

  using for communications [82](#)

XML sockets [82](#)

  processing incoming data [84](#)

## Z

z-order [44](#), [58](#)