Chapter 4: Movie Clips

What is a movie clip?

A movie clip is a LiveMotion object that you can manipulate programmatically through Player 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.

Movie clips have, in addition, a set of built-in properties and methods that are defined by the SWF 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 it's lifetime. A movie clip's built-in methods include functionality that you can perform on movie clips such as creating copies of them, 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 methods" on page 72.

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 that event on that movie clip; the handler is then executed whenever the event occurs. For details on movie clip event handling, see "Movie Clip Events and Event Handlers" on page 79.

Unlike other JavaScript objects, movie clip objects cannot be instantiated: that is, you cannot create a new, original movie clip programatically. 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 programatically create copies of existing movie clips.

How do you 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 creating movie clips and by creating movie clip groups.

To create a movie clip,

Select one object and click the Make Movie Clip button at the bottom of the Timeline window, or choose Timeline>Make Movie Clip from LiveMotion's main menu.

To create a movie clip group,

Select one or more objects in the timeline and click the Make Movie Clip Group button at the bottom of the Timeline window, or choose Timeline>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 group with its own independent timeline. Movie clip groups can contain regular objects, 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>Object Group from the main menu. Alternately, you can press Ctrl+ G (Windows) or Command+G (Mac OS).

This selects the objects. Then it creates an object group out of them.

2 Click the Make Movie Clip button in the timeline, or choose Timeline>Make Movie Clip from LiveMotion's main menu.

This turns the group of objects into a movie clip group with its own independent timeline. 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 is not a group and, as such, cannot contain a child object.

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 4.1 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.

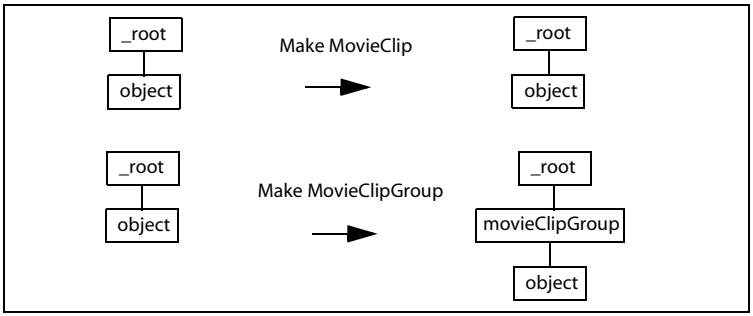

Figure 4.1 Before and after creating a movie clip and creating a movie clip group

Movie clip hierarchy

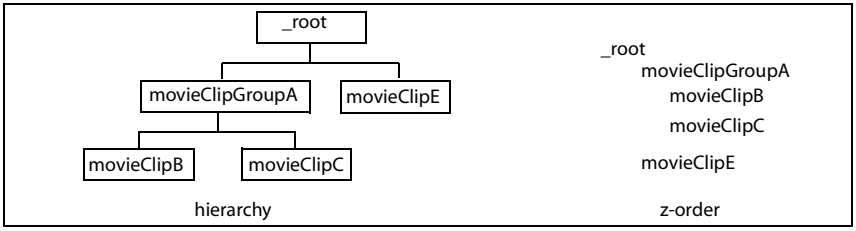

Figure 4.2 Movie clip hierarchy and z-order

All movie clips in a composition are arranged in a hierarchy. At the top of the hierarchy is the main composition timeline. It is referred to as `_root`. In Figure 4.2, `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`.

In LiveMotion, you create a parent-child relationship any time you place (or create) a movie clip or move 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 do you create a movie clip using LiveMotion?" on page 56.

Relationship of movie clip hierarchy to z-order

In the movie clip hierarchy shown in Figure 4.2, 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 the 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 "Programmatic stack order" on page 74. The order just takes on some more detail.

If, for example, you were to open the Timeline window for the composition shown in Figure 4.2, z-order would show the composition timeline at the top and `movieClipGroupA`, above `movieClipE`. But because `movieClipGroupA` is just a movie clip group containing movie clips B and C, the movie clips would appear from front to back in this order in the Composition window: `movieClipB`, `movieClipC`, `movieClipE`.

Note in Figure 4.2 that all movie clips are represented by different names. This is intentional. To be able to refer to child movie clips in scripting, each child must have a unique name. 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 Player scripting, children are accessed as properties of their parent using dot (.) notation. For example, `_root` can access its child `movieClipGroupA` as:

```
_root._movieClipGroupA
```

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 attached. 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 4.2 `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 Player 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 4.3 to illustrate the addressing types.

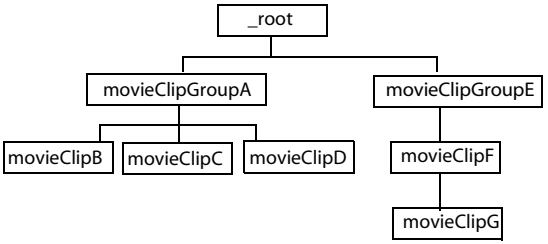

Figure 4.3 Movie clip addressing

What is an absolute reference?

An absolute reference is a reference to a move 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 begins with `_root`, and uses dot (.) notation to access the children of `_root`, and the children's children, and so on until obtaining 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 first 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 begin with `this`, and access the next movie clip in the reference either as a child, or through the `_parent` property until a reference to the desired movie clip is obtained. 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 global function calls and movie clip method function calls.*

Relative reference examples

Here is an example of the relative reference from `movieClipGroupA` 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 rules of thumb:

- 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 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 accessing it is going to be, it is probably better to use an absolute reference. Then, regardless of where the source movie clip is in the hierarchy, the reference to the target will be correct. If you know 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 what the relationship of the movie clips will be, 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 4.4 to movie clip `movieClipD`.

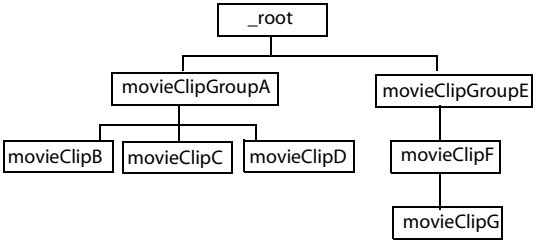

Figure 4.4 Object hierarchy for examples

There is only one absolute reference to `movieClipD`:

```
_root.movieClipGroupA.movieClipD
```

Table 4.1 shows all the relative references to `movieClipD` from each of the other movie clips in Figure 4.4.

Table 4.1 Relative references to movieClipD

Source	Relative reference to movieClipD
`movieClipGroupA`	`this.movieClipD`
`movieClipB`	`this._parent.movieClipD`

Source	Relative reference to movieClipD
movieClipC	this._parent.movieClipD
movieClipD	this
movieClipGroupE	this._parent.movieClipGroupA.movieClipD
movieClipF	this._parent._parent.movieClipGroupA.movieClipD
movieClipG	this._parent._parent._parent.movieClipGroupA.movieClipD

Hands-on example: How to address a movie clip from a location relative to the timeline

This hands-on example illustrates the movie clip addressing concepts described above. The example uses a change state event to move an object. It requires creating two movie clips. One is accessed by the other. The other is a movie clip whose down state causes the movie clip it accesses to move to the right.

The example assumes that you know how to create rollover states. If you need help, see the *Adobe LiveMotion User Guide* for details.

Figure 4.5 shows the main timeline.

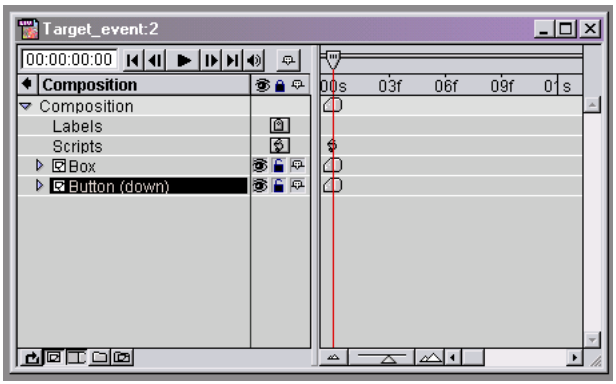

Figure 4.5 Addressing a movie clip

To address the movie clip,

1 Create a rectangular shape in the Composition window.

2 Create a second, elliptical shape with a different color in the Composition window.

3 In the Timeline window, name the objects Box and Button, respectively, and make each a Movie Clip.

4 For Button, use the States and Color palettes to create a normal, over, and down state, respectively—each with a different color.

5 Select Button in the timeline.

6 Select the Button down state in the States palette.

7 Click the scripts button in the States palette.

8 Enter the following script, which uses an absolute reference:

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

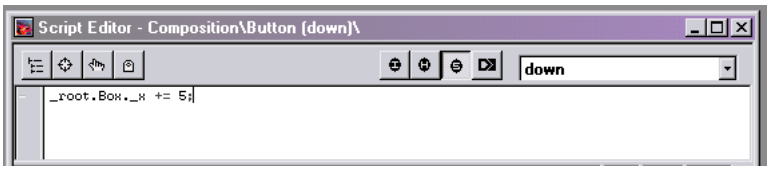

Figure 4.6 Button down state

9 Preview the composition. Each time you click on Ball, the down state calls the down state event handler, which moves Box to the right 5 pixels.

10 Repeat steps 5 through 8, and select the Button down state again. Change the script to:

```
// _root.Box._x += 5; comment out this statement
this._parent.Box._x += 5;
```

This script shows the relative reference from Ball to Box. To access Box from Button, Ball goes "up" one level to the composition timeline (`this._parent`) and then "down" again one level to `Box`.

11 Preview the movie again. The behavior should be exactly the same as when you previewed the composition in step 10. Each click on Ball advances Box to the right 5 pixels.

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 4.2 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.

Table 4.2 Movie clip built-in properties

Property	Description
`_alpha`	Opacity of the movie clip on a scale of 0 (transparent) to 100 (opaque).
`_currentframe`	Position of the playhead in the movie clip's timeline.
`_droptarget`	Absolute reference (in slash notation) of a movie clip over which a movie clip passes during drag operations performed by the user.
`_framesloaded`	Number of the movie clip frames that have been loaded. Also a global movie clip property.
`_height`	Height of the movie clip in pixels.
`_name`	Name of the movie clip.
`_parent`	Movie clip containing this movie clip.
`_rotation`	Rotation angle of the movie clip in degrees.
`_target`	Absolute reference of the movie clip in slash notation.
`_totalframes`	Number of frames in the movie clip.
`_url`	URL of the SWF file that this movie clip is a part of.
`_visible`	Boolean indicating whether the movie clip is visible.
`_width`	Width of the movie clip in pixels.
`_x`	Horizontal location of the movie clip in pixels.
`_xmouse`	Horizontal location of mouse pointer in pixels relative to the anchor point of the movie clip.
`_xscale`	Horizontal percentage scale factor of the movie clip (100% is full size).

Property	Description
_y	Vertical location of the movie clip in pixels.
_ymouse	Vertical location of mouse pointer in pixels.
_yscale	The vertical percentage scale factor of the movie clip (100% is full size).

Movie clip methods

Movie clip methods are functions attached to the movie clip object and are called using `()`. Player 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 "Movie clip methods and global functions that copy movie clips" on page 68.)

Table 4.3 lists the built-in movie clip methods and describes their functions. **???(need to create live link.)** See "Reference" for details on the arguments to each of these methods.

Table 4.3 Movie clip built-in methods

Method	Description
`attachMovie()`	Attach the named movie clip (passed in as an argument) to the movie clip. For details see "Movie clip methods and global functions that copy movie clips" on page 68.
`duplicateMovieClip()`	Duplicate this movie clip. For details see "Movie clip methods and global functions that copy movie clips" on page 68. Also a global movie clip method. See "DuplicateMovieClip() Global Function" on page 160.
`getBounds()`	Return bounds of the movie clip. The returned object contains the values in the properties `xMin`, `XMax`, `yMin` and `yMax`.
`getBytesLoaded()`	Return the number of bytes already loaded if the movie clip is external (loaded with *`MovieClip.`*`loadMovie()`). If the movie clip is internal, the number returned is always the same as that returned by *`MovieClip.`*`getBytesTotal()`.

Method	Description
`getBytesTotal()`	Return the size of the movie clip in bytes. When running under the preview tool in LiveMotion, this number is always 1000.
`getURL()`	Load the URL into the browser. Also a global movie clip method. See "GetURL Global Function" on page 163
`globalToLocal()`	Convert the given global point to the movie clip's coordinate space.
`gotoAndPlay()`	Go to the specified frame or label and plays. Also a global movie clip method. See "GotoAndPlay() Global Function" on page 164
`gotoAndStop()`	Go to the specified frame or label and stops. Also a global movie clip method. See "GotoAndStop() Global Function" on page 165
`hitTest()`	Return a Boolean indicating whether the movie clip intersects with a given clip (passed in as an argument) or given x/y coordinates.
`lmSetCurrentState()`	Change the state of the movie. The LiveMotion state of the movie must already be defined and appear in the state browser.
`loadMovie()`	Load an external SWF file into the movie clip. The contents of the movie clip are replaced with the contents of the SWF file. Also a global movie clip method. See "LoadMovie() Global Function" on page 180.
`loadVariables()`	Load variables into the movie clip fetched from the specified URL. The movie clip's `onData` handler is called when the variables have been loaded. Also a global movie clip method. See "LoadVariables() Global Function" on page 182.
`localToGlobal()`	Convert a point in the movie's coordinate space to global coordinates.
`nextFrame()`	Go to the next frame and stop playing. Also a global movie clip method. See "NextFrame() Global Function" on page 226.
`play()`	Start playing. Also a global movie clip method. See "Play() Global Function" on page 238.
`prevFrame()`	Go to the previous frame and stop playing. Also a global movie clip method. See "PrevFrame() Global Function" on page 239.
`removeMovieClip()`	Delete a duplicated or attached instance. Also a global movie clip method. See "RemoveMovieClip() Global Function" on page 244

Method	Description
`startDrag()`	Start dragging a movie clip. Also a global movie clip method. See "StopDrag() Global Function" on page 258
`stop()`	Stop playing. Also a global movie clip method. See "Stop() Global Function" on page 257
`stopDrag()`	Stop any drag operation in progress. Also a global movie clip method. See "StartDrag() Global Function" on page 256
`swapDepths()`	Swap the movie clips's depth with that of another movie clip. For details on depth, see "Movie clip methods and global functions that copy movie clips" on page 68.
`unloadMovie()`	Unload a movie that was previously loaded with `loadMovie()`. Also a global movie clip method. See "UnloadMovie() Global Function" on page 274
`valueOf()`	Returns the absolute reference to the movie in absolute terms using dot (as opposed to slash) notation.

Movie clip methods and global functions that copy movie clips

Besides creating a movie clip manually in the Composition window, you can create a movie clip programmatically using two movie clip built-in methods. These methods are `duplicateMovieClip()` and `attachMovie()`.

Using duplicateMovieClip() to create movie clip copies

You can call the `duplicateMovieClip()` movie clip method to create a copy of itself that is a sibling of the original. The syntax of the method is:

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

`newName`	String indicating the name of the movie clip copy.
`depth`	Integer that tells where in the programmatic stack to place the movie clip copy.

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

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

`target`	String indicating the path to the movie clip to copy.
`newName`	String indicating the name of the movie clip copy.
`depth`	Integer that tells where in the programmatic stack to place the movie clip copy.

Using attachMovie() to create movie clip copies

The `attachMovie()` movie clip method attaches a stored movie clip stored (**???TBA**) and identified by a string identifier to the specified `depth` in the programmatic stack. It's syntax is:

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

`ID`	**???TBA**
`newName`	String indicating the name of the movie clip copy.
`depth`	Integer that tells where in movieClip's programmatic stack to place the movie clip copy.

Swapping 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`	String indicating the name of the movie clip to swap depths with `movieClip`.
`depth`	Integer that tells where in movieClip's parent programmatic stack to place movieClip.

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 its parent's programmatic stack. If that position is occupied, the movie clip occupying it is moved to `movieClip`'s old position.

Movie clip methods and global functions for loading SWF files

You can load `SWF` files into the Flash Player with `loadMovie()` and `loadMovieNum()`.

loadMovie()

The `loadMovie()` method takes a SWF file at `URL` and loads it into *`movieClip`*.. The syntax of the method is:

```
movieClip.loadMovie(URL);
```

`URL`	String specifying the location of an external SWF file to load.

You can also call `loadMovie()` as a global function. The function replaces a movie clip target or SWF file level with the SWF file. It replaces an occupied SWF file level or fills an empty one. The syntax is:

```
loadMovie(URL, target);
```

`URL`	String specifying the location of an external SWF file to load.
`target`	String indicating the name of the movie clip or a SWF file level into which the SWF file is loaded.

Movie clip methods and global functions to unload movie clips

Using unloadMovie() to unload a movie clip

You can call the `unloadMovie()` movie clip method to unload a movie clip. The syntax of the method is:

```
movieClip.unloadMovie();
```

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

```
unloadMovie(target);
```

target	String indicating the name of the movie clip or SWF file level to remove from the Player.

Global functions to load and unload SWF files

Using the loadMovieNum() global function to load a SWF file

You can call the `loadMovieNum()` global function to load the SWF file at *URL* into an empty SWF file *level* or to replace an occupied SWF file *level*. This function does not load the SWF file into a movie clip. The syntax is:

```
loadMovieNum(URL, level);
```

URL	A string specifying the location of an external SWF file to load.
level	A non-negative integer or expression that evaluates to one that indicates the level into which the SWF file will be loaded.

Using unloadMovieNum() global function to unload a SWF file

You can use the `unloadMovieNum()` global function to unload the SWF file in the specified SWF file level. This function does not unload a movie clip. The syntax is:

```
unloadMovieNum(level);
```

level	A non-negative integer or expression that evaluates to one that indicates the level of the SWF file to be un loaded.

Creating movie clip methods

In addition to using the built-in movie clip methods, you can create movie clip methods of your own. To do so, you can navigate to the movie clip's timeline and define a function. The following method definitions, for example, can be placed in the onLoad handler of the `greeting` movie clip.

```
var clock = 30;
var ctr_clock = 45;

greeting.rotate_clock = function () {
    this._rotation += clock;
}

greeting.rotate_ctr_clock = function() {
    this._rotation -= ctr_clock;
}
```

You call a method that you create in the same way that you would call a method on any object. Provide the name of the movie clip and the method name. When `rotate_ctr_clock()` is called from `greeting`'s own timeline, it appears as:

```
this.rotate_ctr_clock();
```

This function rotates `greeting` counterclockwise.

Creating movie clips programmatically

You create movie clips programmatically using either of two movie clip methods: `attachMovie()` and `duplicateMovieClip()`. The programmatically generated movie clips are placed in a programmatic stack. This section starts by describing static and programmatic stacks. Then it explains which stack is used by each movie clip method. Finally, it shows movie clip order of a composition that includes manually created and programmatically created movie clips.

Static and programmatic stacks

There are two movie clip stacks: a static stack containing manually created movie clips and a programmatic stack containing programmatically created movie clips.

Figure 4.7 illustrates the static and programmatic stacks of manually created movie clip A. Movie clip A's static stack contains its manually created children. Immediately above 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 4.7 depicts four levels 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 programmatic child of movie clip A. In the programmatic stack, the movie clip at depth 3, the highest numeric depth, is the topmost movie clip overlapping all others when when the movie clip executes in the Composition window in Preview mode or in the exported SWF file. The movie clip at depth 3 overlaps the movie clip at depth 2, which overlaps the movie clip at depth 1, and so forth.

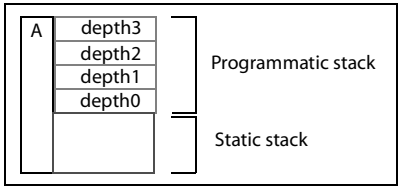

Figure 4.7 Programmatic stack of movie clip A

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

Stack depth

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 yet. 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(--, E, 3);
A.attachMovie(--, B, 0);
A.attachMovie(--, C, 1);
```

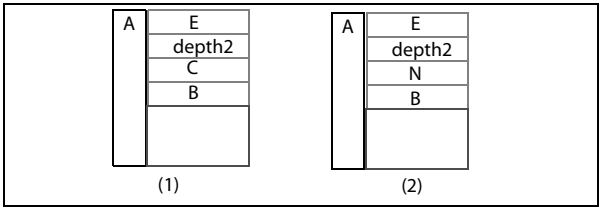

Figure 4.8 Using attachMovie()

Figure 4.8 (1) depicts the placement of the programmatically generated movie clips in movie clip A's stack. 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(--, N, 1);
```

Movie clip `N` will replace movie clip `C`, as shown in Figure 4.8 (2).

The `duplicateMovieClip()` method also creates movie clip copies. However the copies are placed in the caller's parent's programmatic stack. The new movie becomes a sibling of the movie from which it was duplicated. Here is an example of movie clip A creating a duplicate movie clip D:

```
A.duplicateMovieClip(D, 3);
```

In this case, new movie clip `D` is placed in the programmatic stack of movie clip `A's` parent. To understand where this parent stack is, you need to understand programmatic stack order in a composition, as described next.

Programmatic stack order

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 58. For a composition consisting of manually and programmatically generated movie clips, z-order has more detail. You can't view this order in the Composition window, however, until you preview the composition (or you export the SWF file to a browser). The programmatically generated movie clips appear during the course of execution at the time they are generated.

Figure 4.9 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 4.9 (left) shows the manually created movie clips. Figure 4.9 (right) shows the location of the programmatic stack for _root, movie clip `A`, and movie clip `B`.

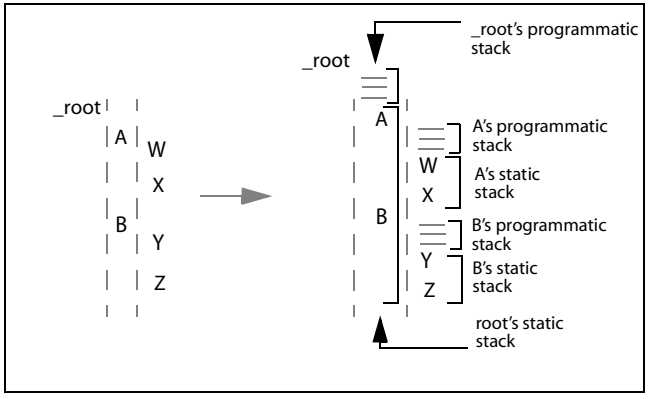

Figure 4.9 Manually and programmatically created movie clips

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(--, 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 4.4 illustrates some more examples of programmatically generated movie clips and indicates the stack in which the movie clips are placed.

Table 4.4 Placement of programmatically generated movie clips

Method call	Stack and depth where movie clip is placed
`A.attachMovie(--, R, 1);`	R is placed in A's programmatic stack at depth 1.
`B.duplicateMovieClip(M, 0);`	M is placed in `_root`'s programmatic stack at depth 0.
`B.attachMovie(--, N, 4);`	N is placed in B's programmatic stack at depth 4.
`Y.duplicateMovieClip(P, 4);`	P is placed in B's programmatic stack at depth 4, replacing movie clip N.
`Z.attachMovie(--, Q, 2);`	Q is placed in Z's programmatic stack at depth 2 (not shown in Figure 4.9);

In Table 4.4, Z's programmatic stack *would be* represented as a fourth view of the composition shown in Figure 4.9. If Z had manually created children, they would appear in Z's manual stack just below its programmatic stack.

Viewing z-order during execution

If you were to run the Composition shown in Figure 4.9 in Preview mode, you would see the most detailed representation of each movie clip in the Timeline window. The z-order, complete with programmatic stacks, is shown in Figure 4.10.

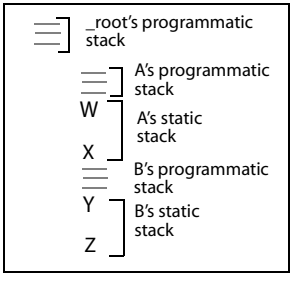

Figure 4.10 Z-order showing programmatic and static stacks

Levels of SWF files

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 level above `_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 4.11 illustrates SWF file stacking order.

`_level` is a global property that you can use to refer to a SWF file when multiple SWF files are loaded into the Player. It is also an argument to the global functions for loading and unloading SWF files described in "Global functions to load and unload SWF files" on page 71. For more information, see the description of this property in "Reference" on page 109.

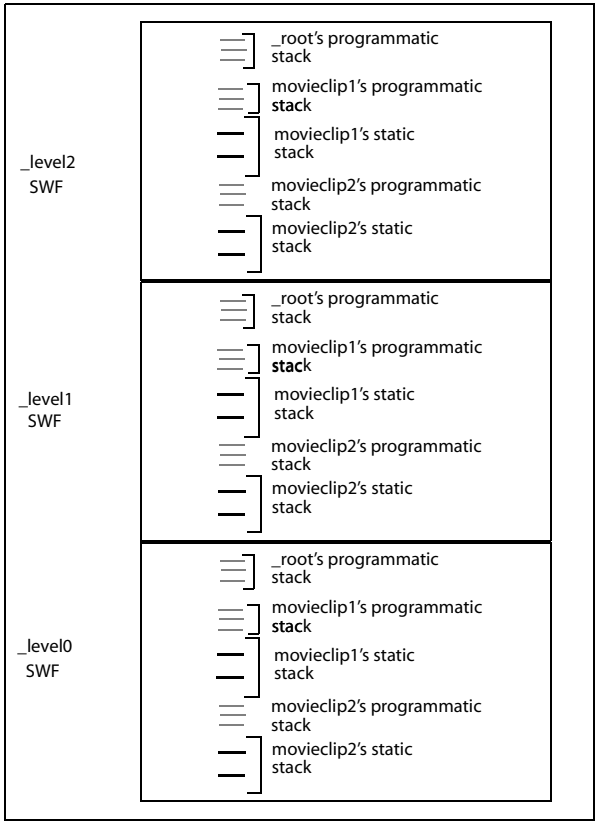

Figure 4.11 Stacking order of SWF files