

CHAPTER 13

Movie Clips

Every Flash document contains a Stage—on which we place shapes, text, and other visual elements—and a main timeline, through which we define changes to the Stage’s contents over time. The Stage (i.e., the *main movie*) can contain independent submovies, called *movie clips* (or *clips* for short). Each movie clip has its own independent timeline and *canvas* (the Stage is the canvas of the main movie) and can even contain other movie clips. A clip contained within another clip is called a *nested clip*. A clip that contains another clip is referred to as the nested clip’s *host clip* or *parent clip*.

A single Flash document can contain a hierarchy of interrelated movie clips. For example, the main movie may contain a mountainous landscape. A separate movie clip containing an animated character can be moved across the landscape to give the illusion that the character is walking. Another movie clip inside the character clip can be used to animate the character’s blinking eyes independently. When the independent elements in the cartoon character are played back together, they appear as a single piece of content. Furthermore, each component can react intelligently to the others; we can tell the eyes to blink when the character stops moving or tell the legs to walk when the character starts moving.

ActionScript offers detailed control over movie clips; we can play a clip, stop it, move its playhead within its timeline, programmatically set its properties (such as its size, rotation, transparency level, and position on the Stage) and manipulate it as a true programming object. As a formal component of the ActionScript language, movie clips can be thought of as the raw material used to produce programmatically generated content in Flash. For example, a movie clip can serve as a ball or a paddle in a pong game, as an order form in a catalog web site, or simply as a container for background sounds in an animation.

The “Objectness” of Movie Clips

As of Flash 5, movie clips can be manipulated like the objects we learned about in Chapter 12. We can retrieve and set the properties of a clip, and we can invoke built-in



or custom methods on a clip. An operation performed on a clip often has a visible or audible result in the Flash Player.

Movie clips are not truly a type of object, but they are object-like; although we can neither create movie clips via a class constructor nor use an object literal to instantiate a movie clip, we can instantiate movie clips using *attachMovie()*, *duplicateMovieClip()* and *createEmptyMovieClip()* (introduced in Flash Player 6). So what, then, are movie clips, if not objects? They are members of their very own object-like datatype, called *movieclip* (we can prove it by executing *typeof* on a movie clip, which returns the string “movieclip”). The main difference between movie clips and other objects is how they are allocated (created) and deallocated (disposed of, or freed). For details, see:

<http://www.moock.org/asdg/technotes/movieclipDatatype>

Despite this technicality, however, we nearly always treat movie clips exactly as we treat objects.

So how does the “objectness” of movie clips affect our use of them in ActionScript? Most notably, it allows us to call methods on clips and examine their properties, just as we can for other objects. Movie clips can be controlled directly through built-in methods. For example:

```
eyes_mc.play();
```

We can retrieve and set a movie clip’s properties using the dot operator, just as we access the properties of any object:

```
ball_mc._xscale = 90;  
var radius = ball_mc._width / 2;
```

A variable in a movie clip is simply a property of that clip, and we can use the dot operator to set and retrieve variable values:

```
theClip_mc.someVariable = 14;  
x = theClip_mc.someVariable;
```

Nested movie clips can be treated as object properties of their parent movie clips. We therefore use the dot operator to access nested clips:

```
clipA.clipB.clipC.play();
```

and we use the reserved *_parent* property to refer to the clip containing the current clip:

```
_parent.clipC.play();
```

Treating clips as objects affords us all the luxuries of convenient syntax and flexible playback control. But our use of clips as objects also lets us manage clips as data; we can store a movie clip reference in an array element or a variable, and we can even pass a clip reference to a function as an argument. Here, for example, is a function that moves a clip to a particular location on the screen:

```
function moveClipTo (clip, x, y) {  
    clip._x = x;
```

```
        clip._y = y;
    }
    moveClipTo(ball_mc, 14, 399);
```

In this chapter, we'll cover the specifics of referencing, controlling, and manipulating movie clips as data objects.

The MovieClip Class

All individual movie clips are instances of the *MovieClip* class, which defines the properties, methods, and event handlers supported by movie clips. The *MovieClip* class can be manipulated like other built-in classes; we can overwrite its methods or add new methods to it using the techniques we studied in Chapter 12. For example, the following code adds a new method, *getArea()*, to all movie clips:

```
MovieClip.prototype.getArea = function () {
    return this._width * this._height;
};
```

As we'll see in Chapter 14, as of Flash MX, the *MovieClip* class can even be used as the superclass for new classes (i.e., new subclasses can be derived from the *MovieClip* class). For full coverage of every property, method, and event handler supported by the *MovieClip* class, see the *Language Reference*.

Types of Movie Clips

Not all movie clips are created equal. In fact, there are three distinct types of clip available in Flash:

- Main movies
- Regular movie clips
- Components (formerly known as *Smart Clips* in Flash 5)

In addition to these three official varieties, we can distinguish four subcategories of regular movie clips:

- Process clips
- Script clips
- Linked clips
- Seed clips

While these unofficial subcategories are not formal terms used in ActionScript, they provide a useful way to think about programming with movie clips. Let's take a closer look at each movie clip type.

Main Movies

The *main movie* of a Flash document is the main timeline and Stage present in every *.swf* document. The main movie is the foundation for all the content in the document, including all other movie clips. We sometimes refer to the main movie as the *main timeline*, the *main movie timeline*, the *main Stage*, or simply the *root*.

Note that while each *.swf* file contains only one main movie, more than one *.swf* file can reside in the Flash Player at once—we can load multiple *.swf* documents (and therefore multiple main movies) onto a stack of *levels* via the *loadMovie()* and *unloadMovie()* functions, which we'll study later.

Main movies can be manipulated in much the same way as regular movie clips, with the following exceptions:


- A main movie cannot be removed from a *.swf* file (although a *.swf* file, itself, can be removed from the Flash Player).
- The following movie clip methods do not work when invoked on a main movie: *duplicateMovieClip()*, *removeMovieClip()*, and *swapDepths()*.
- The following properties are not supported by main movies: *enabled*, *focusEnabled*, *hitArea*, *_name*, *_parent*, *tabChildren*, *tabEnabled*, *tabIndex*, *trackAsMenu*, and *useHandCursor*.
- In Flash 5, event handlers cannot be attached to the main movie. As of Flash Player 6, the following event handler properties can be assigned to the main movie: *onData()*, *onEnterFrame()*, *onKeyDown()*, *onKeyUp()*, *onMouseDown()*, *onMouseMove()*, and *onMouseUp()*.
- Main movies cannot receive keyboard input focus.
- Main movies can be referenced through the built-in global *_root* and *_levelN* properties.

Regular Movie Clips

Regular movie clips are the most common and fundamental content containers; they hold visual elements and sounds, and they can even react to user input and movie playback through event handlers. For JavaScript programmers who are used to working with DHTML, it may be helpful to think of the main movie as analogous to an HTML document object and regular movie clips as analogous to that document's layer objects.

Components

An improvement over Flash 5 Smart Clips, a Flash MX *component* is a movie clip that includes a graphical user interface used to customize the clip's properties in the




authoring tool. Components typically are developed by advanced programmers to distribute as self-contained program modules, such as a pull-down menu or slider bar. Components also provide an easy way for less-experienced Flash authors to customize a movie clip's behavior without knowing how the code of the clip works. We'll cover components in detail in Chapter 14 and Chapter 16. Flash MX comes with a collection of ready-made interface components known as the Flash UI Components. To access the Flash UI Components, choose Window → Components.

Process Clips


A *process clip* is a movie clip used not for content but simply to execute a block of code repeatedly. Process clips can be built with an *onEnterFrame()* event handler or with a timeline loop, as we saw in Chapter 8 under “Timeline and Clip Event Loops.”

As of Flash MX, the functionality of process clips can be partially replaced by the *setInterval()* function, which executes a function or method periodically. Timed code should be implemented with *setInterval()*, whereas code synched specifically with the frame rate should be implemented with a process clip. See the *Language Reference* for a discussion of *setInterval()*.

Script Clips



Like a process clip, a *script clip* is an empty movie clip used not for content but for tracking some variable, defining some class, or simply executing some arbitrary script. For example, in Flash 5, we can use a script clip to hold event handlers that detect keypresses or mouse events. In Flash MX, we can use a script clip to create a so-called “code only” component.



Linked Clips

A *linked clip* is a movie clip that either exports from or imports into the Library of a movie. Export and import settings are available through every movie clip's Linkage option, found in the Library. We most often use linked clips when dynamically generating an instance of a clip directly from a Library symbol using the *attachMovie()* method, as we'll see later.

Seed Clips

Before the *attachMovie()* method was introduced in Flash 5, we used the *duplicateMovieClip()* function to create new movie clips based on some existing clip, called a *seed clip*. A *seed clip* is a movie clip that resides on stage solely for the purpose of being copied via *duplicateMovieClip()*. With the introduction of *attachMovie()*, which instantiates clips from the Library, the need for on-stage seed clips has diminished. However, we still use seed clips and *duplicateMovieClip()* when we wish to retain a clip's transformations and *onClipEvent()* handlers in the process of copying it.

In a movie that makes heavy use of *duplicateMovieClip()* to dynamically generate content, it's common to see a row of seed clips on the outskirts of the movie canvas. The seed clips are used only to derive duplicate clips and are, therefore, kept off stage.

Creating Movie Clips

We usually treat movie clips just as we treat objects—we set their properties with the dot operator; we invoke their methods with the function-call operator (parentheses); and we store them in variables, array elements, and object properties. We do not, however, create movie clips in the same way we create objects. We cannot literally describe a movie clip in our code as we might describe an object with an object literal. And we cannot generate a movie clip with the *new* operator:

```
myClip = new MovieClip(); // Nice try buddy, but it won't work
```

Although Flash Player 6 supports the *new MovieClip()* command, that command establishes a *MovieClip* subclass; it *cannot be used to create a new movie clip instance in a movie*. Instead, we normally create movie clips directly in the authoring tool, by hand. Once a clip is created, we can use commands such as *duplicateMovieClip()* and *attachMovie()* to make new, independent duplicates of it. As of Flash Player 6, we can also create a completely new movie clip at runtime with the *createEmptyMovieClip()* method.

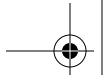
Movie Clip Symbols and Instances

Just as all object instances are based on a class, all movie clip instances are based on a template movie clip, called a *symbol* (sometimes called a *definition*). A movie clip's symbol acts as a model for the clip's content and structure. With the exception of clips created via *createEmptyMovieClip()*, we generate a specific clip instance from a movie clip symbol stored in the Library. Using a symbol, we can both manually and programmatically create clips to be rendered in a movie.

A specific copy of a movie clip symbol is called an *instance*. Instances are the individual clip objects that can be manipulated with ActionScript; a *symbol* is the mold from which instances of a specific movie clip are derived. Movie clip symbols are created in the Flash authoring tool. To make a new, blank symbol, we follow these steps:

1. Select Insert → New Symbol. The Create New Symbol dialog box appears.
2. In the Name field, type an identifier for the symbol.
3. For Behavior, select the Movie Clip radio button.
4. Click OK.

Normally, the next step is to fill the symbol's canvas and timeline with the content of our movie clip. Once a symbol has been created, it resides in the Library, waiting for us to use it to instantiate a movie clip instance. However, it is also possible to



convert a group of shapes and objects that already exist on stage into a movie clip symbol. To do so, we follow these steps:

1. Select the desired shapes and objects.
2. Select Insert → Convert to Symbol.
3. In the Name field, type an identifier for the symbol.
4. For Behavior, select the Movie Clip radio button.
5. Click OK.

The shapes and objects we select to create the new movie clip symbol are replaced by an unnamed instance of that new clip. The corresponding movie clip symbol appears in the Library, ready to be used to create further instances.

Creating Instances

There are four ways to create a new movie clip instance. Three of these are programmatic; the other is strictly manual and is undertaken in the Flash authoring tool. All but one method, *createEmptyMovieClip()*, require an existing movie clip symbol from which to derive the new instance.

Manually creating instances

We can create movie clip instances manually using the Library in the Flash authoring environment. By physically dragging a movie clip symbol out of the Library and onto the Stage, we generate a new instance. An instance created in this way should be named manually via the Instance panel (Flash 5) or Property inspector (Flash MX). You'll learn more about instance names later in this chapter. Refer to "Using Symbols, Instances, and Library Assets" in the Macromedia Flash Help if you've never worked with movie clips in Flash.

Creating instances with *duplicateMovieClip()*

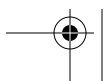
Any instance that already resides on the Stage of a Flash movie can be duplicated with ActionScript. We can then treat that copy as a completely independent clip. Both manually created and programmatically created clip instances can be duplicated. In other words, it's legal to duplicate a duplicate.

There are two ways to duplicate an instance using *duplicateMovieClip()*:

- We can invoke *duplicateMovieClip()* as a *global function*, using the following syntax:

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

where *target* is a string indicating the name of the instance we want to duplicate, *newName* is a string that specifies the identifier for the new instance, and *depth* is an integer that designates where in the *content stack* (we'll discuss the content stack soon) we want to place the new instance.



- We can also invoke `duplicateMovieClip()` as a method of an existing instance:

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

where *theClip* is the name of the clip we wish to duplicate, and *newName* and *depth* both operate as in the previous example.



When created using the `duplicateMovieClip()` function, the newly created clip is attached as a child of either *target*'s or *theClip*'s parent. That is, the new clip becomes a sibling of either *target* or *myClip* in the movie clip hierarchy.

When created via `duplicateMovieClip()`, an instance is initially positioned directly on top of its seed clip. Our first post-duplication task, therefore, is usually moving the duplicated clip to a new position. For example:

```
ball1_mc.duplicateMovieClip("ball2_mc", 0);
ball2_mc._x += 100;
ball2_mc._y += 50;
```

Duplicated instances whose seed clips have been transformed (e.g., colored, rotated, or resized), via ActionScript or manually in the Flash authoring tool, inherit the transformation of their seed clips at duplication time. Subsequent transformations to the seed clip do not affect duplicated instances. Likewise, each instance can be transformed separately. For example, if a seed clip is rotated 45 degrees and then duplicated, the duplicate instance's initial rotation is 45 degrees:

```
seedClip_mc._rotation = 45;
seedClip_mc.duplicateMovieClip("newClip_mc", 0);
trace(newClip_mc._rotation); // Displays: 45
```

If we then rotate the duplicate instance by 10 degrees, its rotation is 55 degrees, but the seed clip's rotation is still 45 degrees:

```
newClip_mc._rotation += 10;
trace(newClip_mc._rotation); // Displays: 55
trace(seedClip_mc._rotation); // Displays: 45
```

By duplicating many instances in a row and adjusting the transformation of each duplicate slightly, we can achieve interesting compound effects, such as stars in the sky, trails effects, and geometric animations (for an example, see "Load Event Starfield" in the online Code Depot).

Using `duplicateMovieClip()` offers other advantages over placing clips manually in a movie, such as the ability to:

- Control exactly when a clip appears on the Stage, relative to a program's execution
- Control exactly when a clip is removed from the Stage, relative to a program's execution
- Copy a clip's event handlers

These abilities give us advanced programmatic control over the content in a movie. With a manually created clip, we must preordain the birth and death of the clip using the timeline.

Creating instances with `attachMovie()`

Like `duplicateMovieClip()`, the `attachMovie()` method lets us create a movie clip instance at runtime; however, `attachMovie()` creates a new instance from a Library symbol instead of from another movie clip instance. To “attach” a movie clip means to make one movie clip the child of either the main timeline or another movie clip in the movie clip hierarchy. In order to use `attachMovie()` to create an instance of a symbol, we must first *export* that symbol from the Library. Here’s how:

1. In the Library, select the desired symbol.
2. In the Library’s pop-up Options menu, select Linkage. The Linkage Properties dialog box appears.
3. Select the Export For ActionScript checkbox.
4. In the Identifier field, type a unique name for the clip symbol. The name can be any string—often simply the same name as the symbol itself—but should be different from all other exported clip symbols.
5. In Flash MX, we can also set the frame at which the link clip will be exported with the movie. For details, see `MovieClip.attachMovie()` in the *Language Reference*.
6. Click OK.

Once a clip symbol has been exported, we can attach new instances of that symbol to an existing clip by invoking `attachMovie()` with the following syntax:

```
theClip.attachMovie(symbolIdentifier, newName, depth, [initObject]);
```

where *theClip* is the name of the movie clip to which we want to attach the new instance. If *theClip* is omitted, `attachMovie()` attaches the new instance to the current clip (the clip on which the `attachMovie()` statement resides). The *symbolIdentifier* parameter is a string containing the name of the symbol we’re using to generate our instance, as specified in the Identifier field of the Linkage options in the Library (Step 4). The *symbolIdentifier* is not necessarily the same name as the symbol itself. The *newName* parameter is a string that specifies the identifier for the new instance we’re creating. If *newName* is not a string that converts to a legal identifier, you’re in for some potentially surprising results, as shown in the examples later in this section. Finally, *depth* is an integer that designates where in the host clip’s content stack to place the new instance. Flash MX adds support for a fourth, optional parameter called *initObject*, which lets you copy properties from an existing object to the new clip. See `MovieClip.attachMovie()` in the *Language Reference* for full details.

For example, to attach a new movie clip, based on the “square” symbol, to the current timeline, use:

```
this.attachMovie("square", "square1_mc", 1);
square1_mc._x = 50; // Position the new clip at a horizontal position of 50
square1_mc._y = 200; // ...and a vertical position of 200
```



The *attachMovie()* method attaches the new clip as a child of *theClip* (or of the current clip if no clip is specified). Contrast this with *duplicateMovieClip()*, which attaches the clip as a sibling of the duplicated clip.

When we attach an instance to another clip, the instance is positioned in the center of the clip, among the clip’s content stack. When we attach an instance to the main movie of a document, the instance is positioned in the upper-left corner of the Stage, at coordinates (0, 0).

In Flash Player 6, the *attachMovie()* function returns a reference to the newly created movie clip, which is useful for debugging. If the function returns *undefined*, then the movie clip creation failed (except in Flash 5, where *attachMovie()* always returns *undefined*). In this case, you most likely specified the wrong name for *symbolIdentifier* (remember to specify the quotes around the string, enter the name correctly, and use the export name specified in the Linkage options, not the symbol’s Library name). The *symbolIdentifier* string is not case-sensitive, so if the symbol is exported as “Symbol 1”, you can (but should not, as a matter of good form) specify “symbol 1” as *symbolIdentifier*.

The return value of the *attachMovie()* function is particularly useful if the operation succeeds and yet you are still having trouble referring to your newly created movie clip by the expected identifier. Technically, *newName* does not need to be a legal identifier, but you’ll have trouble referring to the new clip if it isn’t, unless you store the return value in a variable. That is, you can use the return value to access the clip after it is attached, if the *newName* parameter was specified incorrectly and could not be converted to a legal identifier. The *newName* parameter is actually converted first to a string, and later converted again to an identifier. Let’s see how this works.

In our first example, *newName* is specified as “newClip 1”, which has a space in it, so it can’t be converted to a legal identifier. Luckily, you can use the return value stored in *newClip_mc* to refer to the newly created clip:

```
newClip_mc = _root.attachMovie("Symbol 1", "newClip 1", 0);
trace(newClip_mc); // Displays: _level0.newClip 1
// (which is not a valid identifier)
```

In the next example, *newName* is specified as *newClip1*, which takes the form of a legal identifier but is missing the necessary quotes to make it a string. The *attachMovie()*

command first tries to convert it to a string, but because `newClip1` looks to the interpreter like an undefined variable, the conversion yields the empty string. Luckily, you can again use the return value stored in `newClip_mc` to refer to the newly created clip:

```
newClip_mc = _root.attachMovie("Symbol 1", newClip1, 0);
trace(newClip_mc); // Displays _level0.
                // (the clip's name is an empty string!)
```

What if `newName` is specified as a legal identifier, such as `ball_mc`, that already refers to an existing movie clip? Again, the `attachMovie()` command first tries to convert it to a string, which (referring to Table 3-2 for string conversion of movie clips) yields the interim string “_level0.ball_mc” for `newName`. This series of conversions results in the newly created movie clip having the incorrect name “_level0._level0.ball_mc”. Luckily, yet again you can use the return value stored in `newClip_mc` to refer to the newly created clip properly:

```
//Assume ball_mc is an existing (valid) movie clip identifier
newClip_mc = _root.attachMovie("Symbol 1", ball_mc, 0);
trace(newClip_mc); // Displays "_level0._level0.ball_mc"
```

As you can see, life is much easier when we make sure to provide a string that converts to a legal identifier for `newName`.

Creating instances with `createEmptyMovieClip()`

As of Flash Player 6, completely blank new movie clip instances can be created in an existing clip with the `createEmptyMovieClip()` method, which has the following syntax:

```
theClip.createEmptyMovieClip(newName, depth);
```

where `theClip` is the name of an existing movie clip to which we want to attach a new, empty movie clip. The new clip instance is given a name of `newName` and placed in `theClip`'s content stack at the specified `depth`.

Movie clips created with `createEmptyMovieClip()` are not derived from a Library symbol; they are simply blank movie clip instances with one frame. Empty movie clips can be used as drawing canvases, script clips, or containers for nested clips (e.g., a form clip that contains text fields and a Submit button).

Movie Clip Instance Names

When we create instances, we assign them identifiers, or *instance names*, that allow us to refer to them later. Assigning identifiers to movie clips differs from assigning them to regular objects. When we create a typical object (not a movie clip), we must assign that object to a variable or other data container in order for the object to persist and in order for us to refer to it by name in the future. For example:

```
new Object(); // Object dies immediately after it's created, and
              // we can't refer to it because we didn't store it.
```

```
var thing = new Object(); // Object reference is stored in thing,
                          // and can later be referred to as thing.
```

Movie clip instances need not be stored in variables in order for us to refer to them. Unlike typical objects, movie clip instances are accessible in ActionScript via their instance names as soon as they are created, either programmatically or in the authoring tool. The manner in which an instance gets its initial name depends on how it was created. Programmatically generated instances are named at runtime by the function that creates them. Manually created instances are normally assigned explicit instance names in the authoring tool through the Property inspector, as follows:

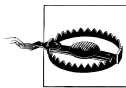
1. Select the instance on stage.
2. In the Property inspector, for <Instance Name>, enter the instance name.

(In Flash 5, the instance name is set via the Instance panel.) Once the instance is named in the authoring tool, it can be accessed via ActionScript using the same name. (It is good practice to add “_mc” as a suffix to the identifier name of any movie clips you create during authoring or at runtime.) For example, if there exists an instance named `ball_mc` on stage, we can access its properties like this:

```
ball_mc.y = 200;
```

If a manually created clip is not given an instance name, it is assigned one automatically by the Flash Player at runtime. Automatic instance names fall in the sequence `instance1`, `instance2`, `instance3`, ...`instancen`, but these names don't meaningfully describe our clip's content (and we must guess at the automatic name that was generated). For example, the first unnamed clip instance can be accessed as:

```
instance1.y = 200;
```



Because instance names assigned during either authoring or runtime are used as identifiers in ActionScript, we should always compose them according to the rules for creating a legal identifier, as described in Chapter 15. Most notably, instance names should not begin with a number or include hyphens or spaces. By convention, movie clip instance names should include the suffix “_mc”, particularly if they are going to be referenced via ActionScript at runtime.

Each clip's instance name is stored in its built-in `_name` property, which can be both retrieved and set. For clips defined manually during authoring, the default `_name` is a string version of the original clip identifier:

```
trace(ball_mc._name); // Displays "ball_mc"
```

For programmatically defined clips, the initial value for `_name` is specified by the `newName` parameter passed to `duplicateMovieClip()` or `attachMovie()`.

The `_name` property is useful for debugging or displaying the name of a clip (without the fully qualified path). Note how the value returned by `_name` differs from other

representations of a movie clip identifier (see also the legacy `_target` property discussed later in this chapter):

```

trace(instance1._name);      // Displays instance1
trace(instance1);           // Displays _level0.instance1
trace(String(instance1));   // Displays _level0.instance1
trace(targetPath(instance1)); // Displays _level0.instance1
trace(instance1.toString()); // Displays [object Object]

```

Example 13-1 uses the `_name` property to find a particular movie clip. See “The `_name` property” later in this chapter for an example showing how to use the `_name` property to prevent an infinite loop.

Example 13-1. Finding movie clips on a timeline

```

// Finds all movie clips inside gameboard_mc with the word "enemy" in their name.
for (var prop in gameboard_mc) {
    if (typeof gameboard_mc[prop] == "movieclip") {
        if (gameboard_mc[prop]._name.indexOf("enemy") != -1) {
            // Found an enemy movie clip...make it attack the player.
            gameboard_mc[prop].attackPlayer();
        }
    }
}

```

If we change an instance’s `_name` property, all future references to the instance must use the new name. For example, if we change the value of `ball_mc._name`, the `ball_mc` reference ceases to exist, and we must subsequently use the new name to refer to the instance:

```

ball_mc._name = "circle_mc"; // Change ball_mc's name to circle_mc
trace(typeof ball_mc);       // Displays "undefined" because ball_mc
                              // no longer exists.
circle_mc._x = 59;           // After the name change, you must
                              // use the clip's new name.

```

Therefore, you shouldn’t change a movie clip’s `_name` property at runtime, as it can make your code fail, or at least make it very difficult to follow.

Importing External Movies and Images

We’ve discussed creating movie clip instances within a single document, but the Flash Player can also display multiple `.swf` documents simultaneously. We can use `loadMovie()`—as either a global function or a movie clip method—to import an external `.swf` file into the Player and place it either in a clip instance or on a numbered level above the base movie (i.e., in the foreground relative to the base movie).



In Flash Player 6, `loadMovie()` can also load JPEG image files into a movie clip or document level. For details, see `loadMovie()` and `MovieClip.loadMovie()` in the *Language Reference*.

Dividing content into separate files gives us precise control over the downloading process and makes partial application updates easier. Suppose, for example, we have a movie containing a main navigation menu and five subsections. Before the user can navigate to section five, sections one through four must finish downloading. But if we place each section in a separate *.swf* file, the sections can be loaded in an arbitrary order, giving the user direct access to each section. To update a section, we can simply replace the appropriate *.swf* file with a new one.

When an external *.swf* is loaded into a level, its main movie timeline becomes the root timeline of that level, and it replaces any prior movie loaded in that level. Similarly, when an external movie is loaded into a clip, the main timeline of the loaded movie replaces that clip's timeline, unloading the existing graphics, sounds, and scripts in that clip.

Like *duplicateMovieClip()*, *loadMovie()* can be used as both a standalone function and an instance method. The standalone syntax of *loadMovie()* is as follows:

```
loadMovie(url, location)
```

where *url* specifies the address of the external *.swf* file to load. The *location* parameter is a string indicating the path to an existing clip or a document level that should host the new *.swf* file (i.e., where the loaded movie should be placed). For example:

```
loadMovie("circle.swf", "_level1");  
loadMovie("photos.swf", "viewClip_mc");
```

Because a movie clip reference is converted to a path when used as a string, *location* can also be supplied as a movie clip reference, such as *_level1* instead of *"_level1"*. Take care when using references, however. If the reference supplied does not point to a valid clip, the *loadMovie()* function has an unexpected behavior—it loads the external *.swf* into the *current* timeline. See “Method Versus Global Function Overlap Issues” later in this chapter for more information on this topic.

The *MovieClip* method version of *loadMovie()* has the following syntax:

```
theClip.loadMovie(url);
```

When used as a clip method, *loadMovie()* assumes we're loading the external *.swf* into *theClip*, so the *location* parameter required by the standalone *loadMovie()* function is not needed. Therefore, we supply only the path to the *.swf* to load via the *url* parameter. Naturally, *url* can be either an absolute or a relative filename, such as:

```
viewClip.loadMovie("photos.swf");
```

When placed into a clip instance, a loaded movie adopts the properties of that clip (e.g., the clip's scale, rotation, color transformation, etc.).

Note that *theClip* must exist in order for *loadMovie()* to be used in its method form. For example, the following attempt to load *circle.swf* will fail if *_level1* is empty:

```
_level1.loadMovie("circle.swf");
```

Using loadMovie() with attachMovie()

Loading an external *.swf* file into a clip instance with *loadMovie()* has a surprising result—it prevents us from attaching instances to that clip via *attachMovie()*. Once a clip has an external *.swf* file loaded into it, that clip can no longer bear attached movies from the Library from which it originated. For example, if *movie1.swf* contains an instance named *clipA*, and we load *movie2.swf* into *clipA*, we can no longer attach instances to *clipA* from *movie1.swf*'s Library.

Why? The *attachMovie()* method works only within a *single* document. That is, we can't attach instances from one document's Library to another document. When we load a *.swf* file into a clip, we are populating that clip with a new document and, hence, a new (different) Library. Subsequent attempts to attach instances from our original document to the clip fail, because the clip's Library no longer matches its original document's Library. However, if we unload the document in the clip via *unloadMovie()*, we regain the ability to attach movies to the clip from its own document Library.

Similarly, loading a *.swf* file into a clip with *loadMovie()* prevents us from copying that clip via *duplicateMovieClip()*.

Load movie execution order

The *loadMovie()* function is not immediately executed when it appears in a statement block. In fact, it is not executed until all other statements in the block have finished executing.



We cannot access an externally loaded movie's properties or methods in the same statement block as the *loadMovie()* invocation that loads it into the Player.

Because *loadMovie()* loads an external file (usually over a network), its execution is *asynchronous*. That is, *loadMovie()* may finish at any time, depending on the speed of the file transfer. Therefore, before we access a loaded movie, we should always check that the movie has finished transferring to the Player. We do so with what's commonly called a *preloader*—code that checks how much of a file has loaded before allowing some action to take place. Preloaders can be built with the *_totalframes* and *_framesloaded* movie clip properties and the *getBytesLoaded()* and *getBytesTotal()* movie clip methods. See the appropriate entries under the *MovieClip* class in the *Language Reference* for sample code.

Movie and Instance Stacking Order

All movie clip instances and externally loaded movies displayed in the Player reside in a visual stacking order akin to a deck of cards. When instances or externally loaded *.swf* files overlap in the Player, one clip (the "higher" of the two) obscures the

other clip (the “lower” of the two). This appears simple enough in principle, but the main *content stack*, which contains all the instances and *.swf* files, is actually divided into many smaller substacks. We’ll first look at these substacks individually first, and then we’ll see how they combine to form the main stack. (The content stack in this discussion has no direct relation to the LIFO and FIFO stacks discussed in Chapter 11.)

Movie Clip Depths

Each movie clip instance, including the main timeline of a movie, places its various contents (movie clips, text fields, and buttons) on one of two stacks: the internal layer stack (for author-time assets) or the programmatically generated content stack (for runtime assets). The items in these stacks (known collectively as the *clip’s content stack*) are given an integer *depth* position that governs how they overlap on screen. Depth positions range from -16384 to 1048575 . Depths from 0 to 1048575 are reserved for dynamically generated content; depths from -16383 to -1 are reserved for author-time content; and depth -16384 is reserved for dynamic content that appears beneath all author-time content in each clip. To retrieve the depth position of an item, we use the *getDepth()* method. To change the depth position of two movie clips, we use the *swapDepths()* method.

The Internal Layer Stack

Instances created manually in the Flash authoring tool reside in the *internal layer stack*. This stack’s order is governed by the actual layers in a movie’s timeline; when two manually created instances on separate timeline layers overlap, the instance on the uppermost layer obscures the instance on the lowermost layer. (Here, “uppermost” means that a layer appears at the top of the timeline panel in the Flash authoring tool).

Furthermore, because multiple clips can reside on a single timeline layer, each layer in the internal layer stack actually maintains its own ministack. Overlapping clips that reside on the same layer of a timeline are stacked in the authoring tool via the *Modify* → *Arrange* commands.

We can swap the position of two instances in the internal layer stack using the *swapDepths()* method, provided they reside on the same timeline (that is, the value of the two clips’ *_parent* property must be the same). Prior to Flash 5, there was no way to alter the internal layer stack via *ActionScript*.

The depth position of author-time assets can be unpredictable and is considered reserved for use by the authoring tool. Therefore, when swapping the depths of an author-time movie clip and a dynamically created clip, always use the *target* parameter of *swapDepths()*—not an integer depth level—as described in the *Language Reference* entry for *MovieClip.swapDepths()*.

The Programmatically Generated Content Stack

Programmatically generated instances are normally stacked separately from the *manually* created instances held in the internal layer stack. Each movie clip has its own *programmatically generated content stack* that holds:

- Movie clip instances created via `duplicateMovieClip()`, `attachMovie()`, and `createEmptyMovieClip()`
- Text field instances created via `createTextField()`

The stacking order for movie clips in the programmatically generated content stack varies, depending on how they were created.

How clips generated via `attachMovie()` and `createEmptyMovieClip()` are added to the stack

A new instance generated via `attachMovie()` or `createEmptyMovieClip()` is always stacked above (i.e., in the foreground relative to) the clip to which it was attached. For example, suppose that we have two clips—X and Y—in the internal layer stack of a movie and that X resides on a layer above Y. Now further suppose we attach a new clip, A, to X and a new clip, B, to Y:

```
x.attachMovie("A", "A", 0);
y.attachMovie("B", "B", 0);
```

In our scenario, the clips appear from top to bottom in this order: A, X, B, Y, as shown in Figure 13-1.

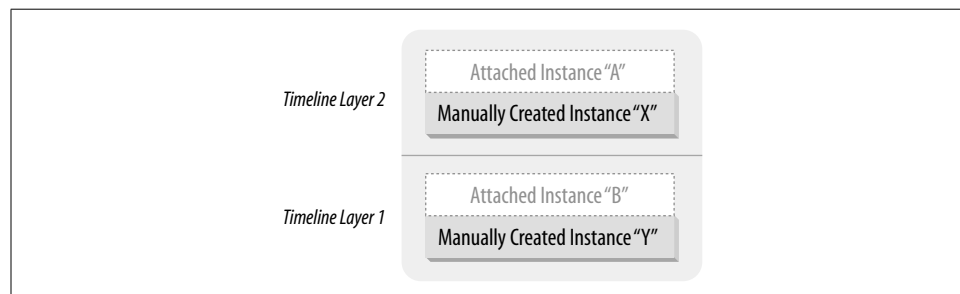


Figure 13-1. A sample instance stack

Once a clip is generated, it too provides a separate space above its internal layer stack for more programmatically generated clips. That is, we can attach clips to attached clips.

Clips attached to the `_root` movie of a Flash document are placed in the `_root` movie's programmatically generated content stack, which appears in front of *all* clips in the `_root` movie, even those that contain programmatically generated content.

Let's extend our example. If we attach clip C to the `_root` of the movie that contains clips X, Y, A, and B, then clip C appears in front of all the other clips. Figure 13-2 shows the extended structure.

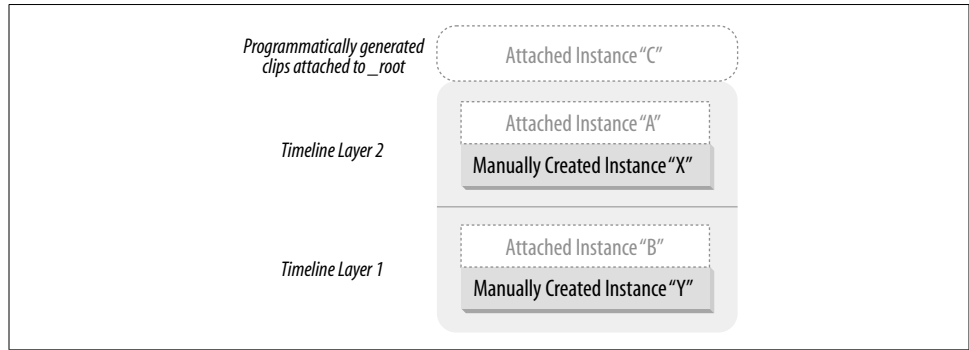


Figure 13-2. An instance stack showing a clip attached to `_root`

How clips generated via `duplicateMovieClip()` are added to the stack

Each instance duplicated via `duplicateMovieClip()` is assigned to the programmatic stack of its seed clip's parent (the movie clip upon whose timeline the seed clip resides). Let's return to our example to see how this works.

If we create clip D by duplicating clip X (which was created manually), then clip D is placed in the stack above `_root`, with clip C. A seed clip and its duplicate always share the same parent (in this example, `_root`). Similarly, if we create clip E by duplicating clip D, then E is also placed in the stack above `_root`, with C and D. But if we create clip F by duplicating clip A—which was created with `attachMovie()`—then F is placed in the programmatic stack above X, with clip A. Again, F and its seed clip, A, share the same parent: X. Figure 13-3 is worth a thousand words.

Assigning depths to instances in the programmatically generated content stack

You may be wondering what determines the stacking order of clips C, D, and E, or of clips A and F, in Figure 13-3. The stacking order of a programmatically generated clip is determined by the `depth` argument passed to the `attachMovie()`, `createEmptyMovieClip()`, or `duplicateMovieClip()` methods, and can be changed at any time using the `swapDepths()` function. Each programmatically generated clip's `depth` (sometimes called its `z-index`) determines its position within a particular stack of programmatically generated clips.

The `depth` of a clip can be any integer and is measured from the bottom up—so, depth 5 is lower than depth 6, depth 7 is higher than (i.e., in front of) depth 6, depth 8 is higher still, and so on. When two programmatically generated clips occupy the

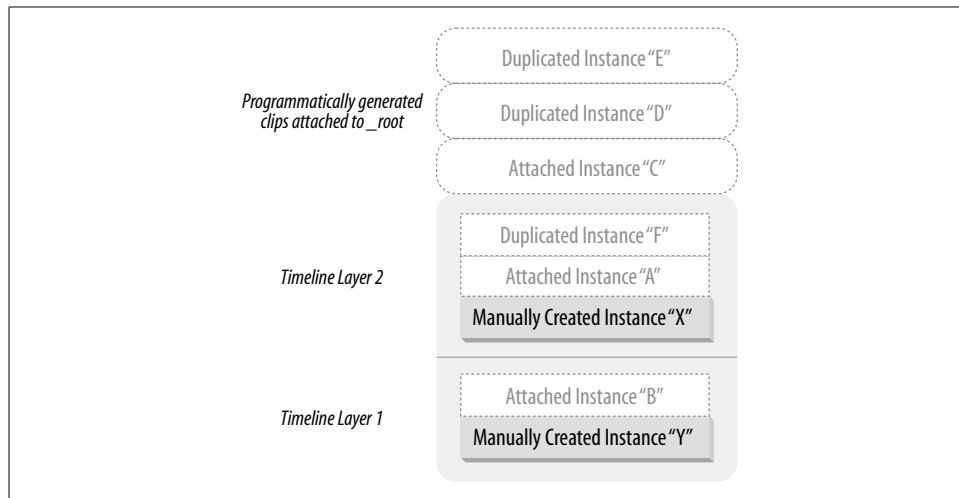


Figure 13-3. An instance stack showing various duplicated clips

same position on screen, the one with the greater *depth* value is rendered in front of the other.

Although multiple clips can occupy a single author-time timeline layer, all depth positions are single-occupant dwellings. Only one clip can occupy a depth position at a time—placing a clip into an occupied position displaces (and deletes) the layer’s previous occupant.

It’s okay for there to be gaps in the depths of clips; you can have a clip at depth 0, another at depth 500, and a third one at depth 1000. No performance hit or increase in memory consumption results from having gaps in your depth assignments.

The .swf Document “_level” Stack

In addition to the internal layer stack and the programmatically generated content stack, there’s a third (and final) kind of stack, the *document stack* (or *level stack*), which governs the overlapping not of instances, but of entire .swf files loaded into the Player via *loadMovie()*.

The first .swf file loaded into the Flash Player is placed in the lowest level of the document stack (represented by the global property *_level0*). If we load any additional .swf files into the Player after that first document, we can optionally place them in front of the original document by assigning them to a level above *_level0* in the document stack. All of the content in the higher-level documents in the level stack appears in front of lower-level documents, regardless of the movie clip stacking order within each document.

Just as the programmatically generated content stack allows only one clip per layer, the document stack allows only one document per level. If we load a .swf file into an occupied level, the level’s previous occupant is replaced by the newly loaded

document. For example, you can supplant the original document by loading a new .swf file into `_level0`. Loading a new .swf file into `_level1` visually obscures the movie in `_level0`, but it does not remove it from the Player.

Figure 13-4 summarizes the relationships of the various stacks maintained by the Flash Player.

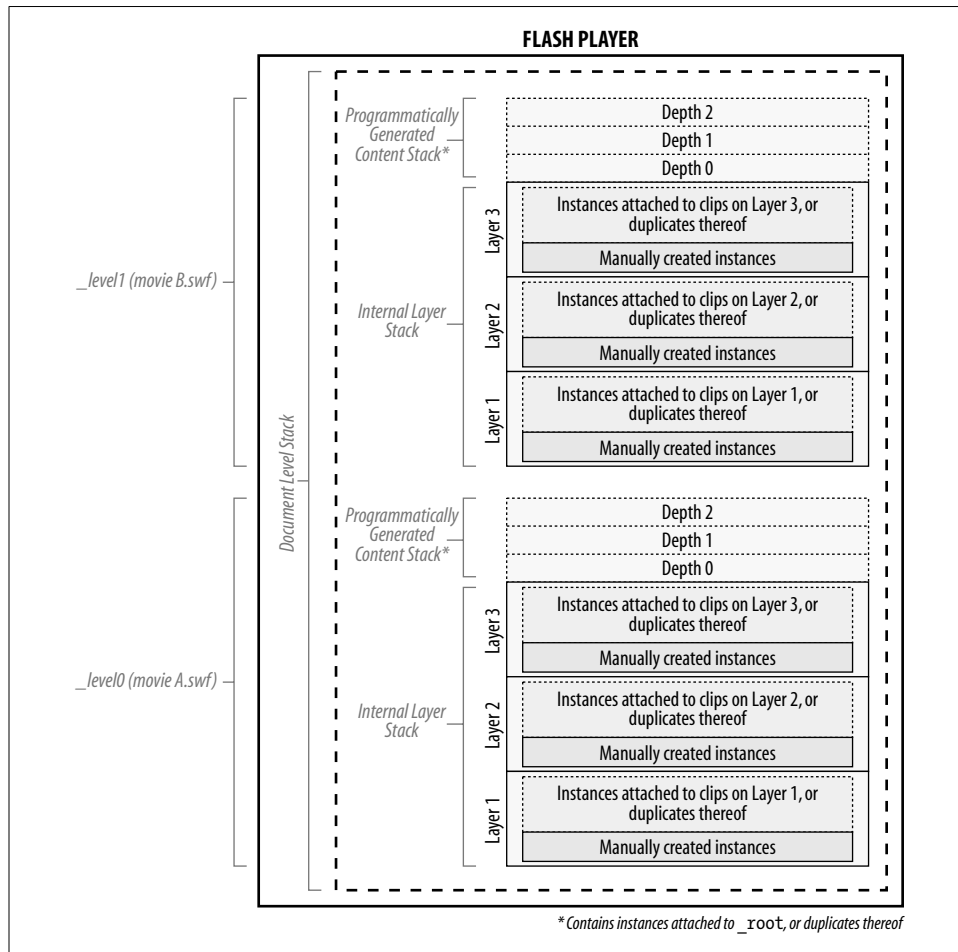


Figure 13-4. The complete Flash Player content stack

Stacks and Order of Execution

The layering of movie clips and timeline layers affects code execution order. The rules are as follows:

- Code on frames in different timeline layers always executes from top to bottom (relative to the timeline panel).

- When manually created instances are initially loaded, code in their timeline and *onLoad()* event handlers executes according to the Load Order set in the Publish Settings of a Flash document—either Bottom Up, which is the default, or Top Down.

For example, suppose we have a timeline with two layers, *top* and *bottom*, where *top* is above *bottom* in the layer stack. We place clip *X* on layer *top* and clip *Y* on layer *bottom*. If the Load Order of the document is set to Bottom Up, then the code in clip *Y* will execute before the code in clip *X*. If, on the other hand, the Load Order of the document is set to Top Down, then the code in clip *X* will execute before the code in clip *Y*. This execution order applies *only* to the frame on which *X* and *Y* appear for the first time.

- Once loaded, all instances of a movie are added to an execution order, which is the reverse of the load order; the last instance added to the movie is always the first to have its code executed.

Use caution when relying on these rules. Layers are mutable, so you should avoid producing code that relies on their relative position. Strive to create code that executes safely without relying on the execution order of the clips in the stack. We can avoid some of the issues presented by the execution stack by keeping all our code on a *scripts* layer at the top of each code-bearing timeline.

Referring to Instances and Main Movies

In earlier sections of this chapter, we saw how to create and layer movie clip instances and external *.swf* files in the Flash Player. We must be able to refer to that content in order to effectively control it with ActionScript.

We refer to instances and main movies under four general circumstances, when we want to:

- Get or set a property of a clip or a movie
- Create or invoke a method of a clip or a movie
- Apply some function to a clip or a movie
- Manipulate a clip or movie as data—for example, by storing it in a variable or passing it as an argument to a function

While the circumstances under which we refer to clip instances and movies are fairly simple, the tools we have for making references are many and varied. We'll spend the rest of this section exploring ActionScript's instance- and movie-referencing tools.

Using Instance Names

Earlier, we saw that movie clips are referred to by their *instance names*. For example:

```
trace(someVariable);    // Refer to a variable
trace(someClip_mc);    // Refer to a movie clip
```

In order to refer to an instance directly (as shown in the preceding *trace()* example), the instance must reside on the timeline to which our code is attached. For example, if we have an instance named `clouds_mc` placed on the main timeline of a document, we can refer to `clouds_mc` from code attached to the main timeline, as follows:

```
// Set a property of the instance
clouds_mc._alpha = 60;
// Invoke a method on the instance
clouds_mc.play();
// Place the instance in an array of other related instances
var background = [clouds_mc, sky_mc, mountains_mc];
```

If the instance we want to reference does not reside on the same timeline as our code, we must use a more elaborate syntax, as described later in this section under “Referring to Nested Instances.”

Referring to the Current Instance or Movie

We don’t always have to use an instance’s name when referring to a clip. Code attached to a frame in an instance’s timeline can refer to that instance’s properties and methods directly, without any instance name.

For example, to set the `_alpha` property of a clip named `cloud_mc`, we can place the following code on a frame in the `cloud_mc` timeline:

```
_alpha = 60;
```

Similarly, to invoke the `play()` method on `cloud_mc` from a frame in the `cloud_mc` timeline, we can simply use:

```
play();
```

This technique can be used on any timeline, including timelines of main movies. For example, the following two statements are synonymous if attached to a frame on the main timeline of a Flash document. The first refers to the main movie implicitly, whereas the second refers to the main movie explicitly via the global `_root` property:

```
gotoAndStop(20);
_root.gotoAndStop(20);
```

However, not all methods can be used with an implicit reference to a movie clip. Any movie clip method that has the same name as a corresponding global function, such as `duplicateMovieClip()` or `unloadMovie()`, must be invoked with an explicit instance reference. Hence, when in doubt, use an explicit reference. We’ll have more to say about method and global function conflicts later in this chapter under “Method Versus Global Function Overlap Issues.”

Note that it’s always safest to use explicit references to variables or movie clips rather than using implicit references. Implicit references are ambiguous, often causing unexpected results and confusing other developers reading your code.

Self-references with the `this` keyword

When we want to refer explicitly to the current instance from a frame in its timeline or from one of its event handlers, we can use the `this` keyword. For example, the following statements are synonymous when attached to a frame in the timeline of our `cloud_mc` instance:

```
alpha = 60; // Implicit reference to the current timeline
this.alpha = 60; // Explicit reference to the current timeline
```

There are three reasons to use `this` to refer to a clip even when we could legitimately refer to the clip's properties and methods directly.

- First, explicit references are easier for other developers to read, because they make the intention of a statement unambiguous.
- Second, when used without an explicit instance reference, certain movie clip methods are mistaken for global functions by the interpreter. If we omit the `this` reference, the interpreter thinks we're trying to invoke the analogous global function and complains that we're missing the *target* movie clip parameter. To work around the problem, we use `this`, as follows:

```
this.duplicateMovieClip("newClouds_mc", 0); // Invoke a method on an instance

// If we omit the this reference, we get an error
duplicateMovieClip("newClouds_mc", 0); // Oops!
```

- Third, using `this`, we can conveniently pass a reference to the current timeline to functions that operate on movie clips:

```
// Here's a function that manipulates clips
function moveClipTo (theClip, x, y) {
    theClip._x = x;
    theClip._y = y;
}

// Now let's invoke it on the current timeline
moveClipTo(this, 150, 125);
```



New and experienced object-oriented programmers alike, take note: the meaning of `this` inside a method is a reference not to the current timeline but to the object through which the method was invoked. If an object's method needs to refer to a specific movie clip, a reference to that clip should be passed to the method as a parameter. See Chapter 12 for details on using the `this` keyword inside methods.

Referring to Nested Instances

As we discussed in the introduction to this chapter, movie clip instances are often nested inside of one another. That is, a clip's canvas can contain an instance of another clip, which can itself contain instances of other clips. For example, a game's spaceship clip can contain an instance of a `blinkingLights` clip or a `burningFuel` clip. Or a character's face clip can include separate eyes, nose, and mouth clips.

Earlier, we saw briefly how we can navigate up or down from any point in the hierarchy of clip instances, much like you navigate up and down a series of subdirectories on your hard drive. Let's examine this in more detail and see some more examples.

Let's first consider how to refer to a clip instance that is nested inside of the current instance.



When a clip is placed on the timeline of another clip, it becomes a property of that clip, and we can access it as we would access any object property (with the dot operator).

For example, suppose we place clipB on the canvas of clipA. To access clipB from a frame in clipA's timeline, we use a direct reference to clipB:

```
clipB._x = 30;
```

We could also use an explicit reference, as in:

```
this.clipB._x = 30;
```

Now suppose clipB contains another instance, clipC. To refer to clipC from a frame in clipA's timeline, we access clipC as a property of clipB, like this:

```
clipB.clipC.play();
clipB.clipC._x = 20;
```

Beautiful, ain't it? And the system is infinitely extensible. Because every clip instance placed on another clip's timeline becomes a property of its host clip, we can traverse the hierarchy by separating the instances with the dot operator, like so:

```
clipA.clipB.clipC.clipD.gotoAndStop(5);
```

Now that we've seen how to navigate down the instance hierarchy, let's see how we navigate up the hierarchy to refer to the instance or movie that contains the current instance. As we saw earlier, every instance has a built-in `_parent` property that refers to the clip or main movie containing it. We use the `_parent` property like so:

```
theClip._parent
```

where `theClip` is a reference to a movie clip instance. Recalling our recent example with `clipA` on the main timeline, `clipB` inside `clipA`, and `clipC` inside `clipB`, let's see how to use `_parent` and dot notation to refer to the various clips in the hierarchy. Assume that the following code is placed on a frame of the timeline of `clipB`:

```
_parent // A reference to clipA
this // An explicit relative reference to clipB (the current clip)
this._parent // An explicit relative reference to clipA
// Sweet Sheila, I love this stuff! Let's try some more...
_parent._parent // A reference to clipA's parent (clipB's grandparent),
// which is the main timeline in this case
```

Note that although it is legal to do so, it is unnecessarily roundabout to traverse down the hierarchy using a reference to the `clipC` property of `clipB` only to traverse

back *up* the hierarchy using `_parent`. These roundabout references are unnecessary but do show the flexibility of dot notation:

```
clipC._parent           // A roundabout reference to clipB
                        // (the current timeline)
clipC._parent._parent  // A roundabout reference to the main timeline
```



Notice how we use the dot operator to descend the clip hierarchy and use the `_parent` property to ascend it.

If this is new to you, you should probably build the `clipA`, `clipB`, `clipC` hierarchy in Flash and test the code in our example. Proper instance referencing is one of the fundamental skills of a good ActionScript programmer.

Note that the hierarchy of clips is like a family tree. Unlike a typical family tree of a sexually reproducing species, in which each offspring has two parents, our clip family tree expands asexually. That is, each household is headed by a single parent who can adopt any number of children. Any clip (i.e., any *node* in the tree) can have one and only one parent (the clip that contains it) but can have multiple *children* (the clips that it contains). Of course, each clip’s parent can in turn have a single parent, which means that each clip can have only one grandparent (not the four grandparents humans typically have). Figure 13-5 shows a sample clip hierarchy.

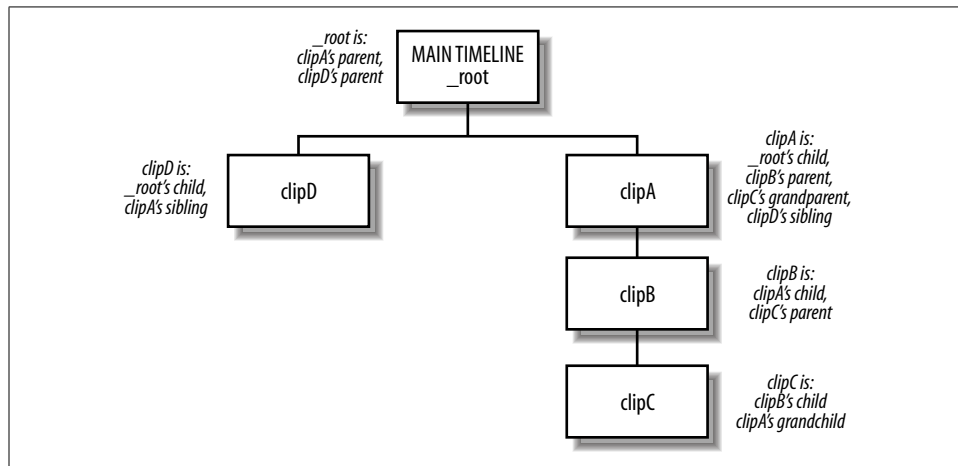


Figure 13-5. A sample clip hierarchy

No matter how far you go down the family tree, if you go back up the same number of steps you will always end up in the same place you started. It is therefore pointless to go down the hierarchy only to come back up. However, it is *not* pointless to go up the hierarchy and then follow a *different* path back down. For example, suppose that our example main timeline also contains `clipD`, which makes `clipD` a “sibling” of

clipA because both have the main timeline as their `_parent`. In this case, you can refer to `clipD` from a script attached to `clipB` as follows:

```
_parent._parent.clipD // This refers to clipD, a child of the main
                       // timeline (clipA's _parent) and therefore
                       // a sibling of clipA
```

Note that the main timeline does not have a `_parent` property (main movies are the top of any clip hierarchy and cannot be contained by another timeline); references to `_root._parent` yield undefined.

Referring to Main Movies with `_root` and `_levelN`

Now that we've seen how to navigate up and down the clip hierarchy *relative* to the current clip, let's explore other ways to navigate along *absolute* pathways and even among other documents stored in other levels of the Player's document stack. In earlier chapters, we saw how these techniques applied to variables and functions; here we'll see how they can be used to control movie clips.

Referencing the current level's main movie using `_root`

When an instance is deeply nested in a clip hierarchy, we can repeatedly use the `_parent` property to ascend the hierarchy until we reach the main movie timeline. But in order to ease the labor of referring to the main timeline from deeply nested clips, we can also use the built-in global property `_root`, which is a shortcut reference to the main movie timeline. For example, here we play the main movie:

```
_root.play();
```

The `_root` property is said to be an *absolute reference* to a known point in the clip hierarchy because unlike the `_parent` and `this` properties, which are relative to the current clip, the `_root` property refers to the main timeline of the current level, no matter which clip within the hierarchy references it (see the exception in the next warning). These are all equivalent (except from scripts attached to the main timeline, where `_parent` is not valid):

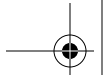
```
_parent._root
this._root
_root
```

Therefore, you can use `_root` when you don't know where a given clip is nested within the hierarchy. For example, consider the following hierarchy in which `circle` is a child of the main movie timeline and `square` is a child of `circle`:

```
main timeline
  circle
    square
```

Now consider this script attached to a frame in both `circle` and `square`:

```
_parent._x += 10 // Move this clip's parent clip 10 pixels to the right
```



When this code is executed from within `circle`, it causes the main movie to move 10 pixels to the right. When it is executed from within `square`, it causes `circle` (not the main movie) to move 10 pixels to the right. In order for the script to move the main movie 10 pixels regardless of where the script is executed from, the script can be rewritten as:

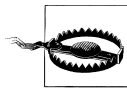
```
_root._x += 10 // Move the main movie 10 pixels to the right
```

Furthermore, the `_parent` property is not valid from within the main timeline; the version of the script using `_root` is valid even when used in a frame of the main timeline.

The `_root` property can be combined with ordinary instance references to descend a nested-clip hierarchy:

```
_root.clipA.clipB.play();
```

References that start with `_root` refer to the same, known, starting point from anywhere in a document. There's no guessing required.



When a `.swf` file is loaded into a movie clip instance, `_root` refers no longer to that `.swf` file's main timeline but to the main timeline of the movie into which the `.swf` was loaded!

If you know your movie will be loaded into a movie clip, you should not use `_root` to refer to the main timeline. Instead, define a global reference to the main timeline by placing the following code on your movie's main timeline:

```
_global.myAppMain = this;
```

where `myApp` is the name of your application and `Main` is used, by convention, to denote the main timeline. Then use `myAppMain` in place of `_root`.

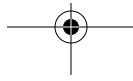
Referencing other documents in the Player using `_leveln`

If we have loaded multiple `.swf` files into the document stack of the Flash Player using `loadMovie()`, we can refer to the main movie timelines of the various documents using the built-in series of global properties `_level0` through `_leveln`, where `n` represents the level of the document we want to reference.

Therefore, `_level0` represents the document in the lowest level of the document stack (documents in higher levels will be rendered in the foreground). Unless a movie has been loaded into `_level0` via `loadMovie()`, `_level0` is occupied by the movie that was initially loaded when the Player started.

Here is an example that plays the main movie timeline of the document in level 3 of the Player's document stack:

```
_level3.play();
```



Like the `_root` property, the `_leveln` property can be combined with ordinary instance references via the dot operator:

```
_level1.clipA.stop();
```

As with references to `_root`, references to `_leveln` properties are called *absolute references* because they lead to the same destination from any point in a document.

Note that `_leveln` and `_root` are not synonymous. The `_root` property is always the *current* document's main timeline, regardless of the level on which the current document resides, whereas the `_leveln` property is a reference to the main timeline of a specific document level. For example, suppose we place the code `_root.play()` in `myMovie.swf`. When we load `myMovie.swf` onto level 5, our code plays `_level5`'s main movie timeline. In contrast, if we place the code `_level2.play()` in `myMovie.swf` and load `myMovie.swf` into level 5, our code plays `_level2`'s main movie timeline, not `_level5`'s. Of course, from within level 2, `_root` and `_level2` are equivalent.

Authoring Instance References with Insert Target Path

When the instance structure of a movie gets very complicated, composing references to movie clips and main movies can be laborious. We may not always recall the exact hierarchy of a series of clips and, hence, may end up frequently selecting and editing clips in the authoring tool just to determine their nested structure. The Actions panel's Insert Target Path tool generates clip references visually, relieving the burden of creating them manually. The Insert Target Path button is shown in Figure 13-6.

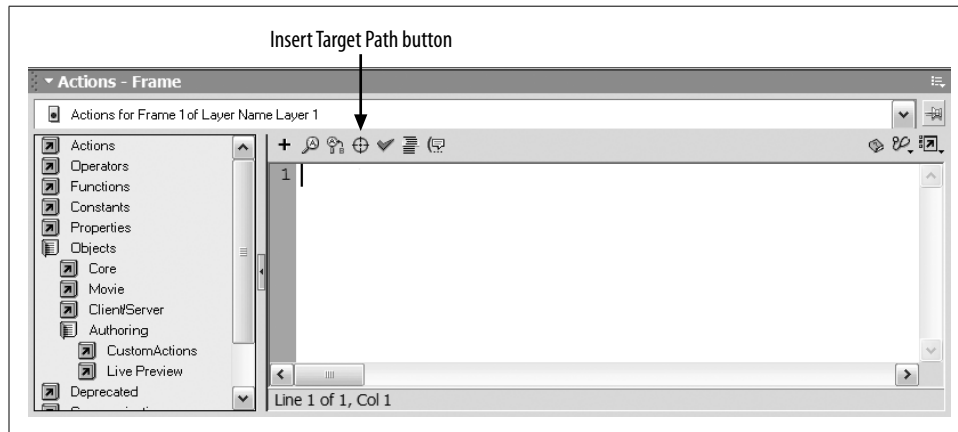


Figure 13-6. The Insert Target Path button

To use Insert Target Path, follow these steps:

1. Position the cursor in your code where you want a clip reference to be inserted.
2. Click the Insert Target Path button, shown in Figure 13-6.
3. In the Insert Target Path dialog box, select the clip to which you want to refer.

4. Choose whether to insert an *absolute reference*, which begins with `_root`, or a *relative reference*, which expresses the reference to the target clip in relation to the clip that contains your code (`this`).
5. If you are exporting to Flash 4 format, choose the Slashes Notation button for Flash 4 compatibility. (The Dot Notation button, selected by default, composes references that won't work in Flash 4).

The Insert Target Path tool cannot generate relative references that ascend a hierarchy of clips. That is, the tool cannot be used to refer to a clip that contains the current clip (unless you want to begin the path from `_root` and proceed downward). To create references that ascend the clip hierarchy, we must either use absolute references starting with `_root` (which therefore become descending references) or manually enter the appropriate relative references in our code using the `_parent` property.

Dynamic References to Clip Objects

Normally, we know the name of the specific instance or movie we are manipulating, but there are times when we'd like to control a clip whose name we don't know. We may, for example, want to scale down a whole group of clips using a loop or create a button that refers to a different clip each time it is clicked. To handle these situations, we must create our clip references dynamically at runtime.

Using the array-element access operator

As we saw in Chapter 5 and Chapter 12, the properties of an object can be retrieved via the dot operator or through the array-element access operator, `[]`. For example, the following two statements are equivalent:

```
someObject.myProperty = 10;  
someObject["myProperty"] = 10;
```

The array-element access operator has one important feature that the dot operator does not; it lets us (indeed requires us to) refer to a property using a *string expression* rather than an *identifier*. For example, here's a string concatenation expression that acts as a valid reference to the property `propertyX`:

```
someObject["prop" + "ertyX"];
```

We can apply the same technique to create our instance and movie references dynamically. We already saw that clip instances are stored as properties of their parent clips. Earlier, we used the dot operator to refer to those instance properties. For example, from the main timeline we can refer to `clipB`—which is nested inside of another instance, `clipA`—as follows:

```
clipA.clipB; // Refer to clipB inside clipA  
clipA.clipB.stop(); // Invoke a method on clipB
```

Because instances are properties, we can also legitimately refer to them with the `[]` operator, as in:

```
clipA["clipB"];           // Refer to clipB inside clipA
clipA["clipB"].stop();    // Invoke a method on clipB
```

Notice that when we use the [] operator to refer to clipB, we provide the name of clipB as a string, not an identifier. That string reference can be any valid string-yielding expression. For example, here's a reference to clipB that involves a string concatenation:

```
var clipCount = "B";
clipA["clip" + clipCount]; // Refer to clipB inside clipA
clipA["clip" + clipCount].stop(); // Invoke a method on clipB
```

We can create clip references dynamically to refer to a series of sequentially named clips:

```
// Here's a loop that stops clip1, clip2, clip3, and clip4
for (var i = 1; i <= 4; i++) {
    _root["clip" + i].stop();
}
```

Now that's powerful!

Storing references to clips in data containers

We began this chapter by saying that though movie clips are technically their own datatype, they are treated as objects in ActionScript. Hence, we can store a reference to a movie clip instance in a variable, an array element, or an object property.

Recall our earlier example of a nested instance hierarchy (clipC nested inside clipB nested inside clipA) placed on the main timeline of a document. If we store these various clips in data containers, we can control them dynamically using the containers instead of explicit references to the clips. Example 13-2, which shows code that is placed on a frame in the main timeline, uses data containers to store and control instances.

Example 13-2. Storing clip references in variables and arrays

```
var x = clipA.clipB; // Store a reference to clipB in the variable x
x.play();           // Play clipB
// Now let's store our clips in the elements of an array
var theClips = [clipA, clipA.clipB, clipA.clipB.clipC];
theClips[0].play(); // Play clipA
theClips[1]._x = 200; // Place clipB 200 pixels from clipA's registration point
// Stop all the clips in our array using a loop
for (var i = 0; i < myClips.length; i++) {
    myClips[i].stop();
}
```

By storing clip references in data containers, we can manipulate the clips (such as playing, rotating, or stopping them) without knowing or affecting the document's clip hierarchy. Storing clip references in variables also make our code more legible. You can use a shorter, simpler variable name instead of a lengthy absolute or relative path through the movie clip hierarchy.

Using *for-in* to access movie clips

In Chapter 8, we saw how to enumerate an object's properties using a *for-in* loop. Recall that a *for-in* loop's iterator variable automatically cycles through all the properties of the object, so that the loop is executed once for each property:

```
for (var prop in someObject) {
    trace("the value of someObject." + prop + " is " + someObject[prop]);
}
```

Example 13-3 shows how to use a *for-in* loop to enumerate all the clips that reside on a given timeline.

Example 13-3. Finding movie clips on a timeline

```
for (var property in someClip) {
    // Check if the current property of someClip is a movie clip
    if (typeof someClip[property] == "movieclip") {
        trace("Found instance: " + someClip[property]._name);
        // Now do something to the clip
        someClip[property]._x = 300;
        someClip[property].play();
    }
}
```

The *for-in* loop gives us convenient access to the clips contained by a specific clip instance or main movie. Using *for-in*, we can control any clip on any timeline, whether or not we know the clip's name and whether the clip was created manually or programmatically.

Example 13-4 shows a recursive version of Example 13-3. It finds all the clip instances on a timeline, plus the clip instances on all nested timelines.

Example 13-4. Finding all movie clips on a timeline recursively

```
function findClips (theClip, indentSpaces) {
    // Use spaces to indent the child clips on each successive tier
    var indent = " ";
    for (var i = 0; i < indentSpaces; i++) {
        indent += " ";
    }
    for (var property in theClip) {
        // Check if the current property of theClip is a movie clip
        if (typeof theClip[property] == "movieclip") {
            trace(indent + theClip[property]._name);
            // Check if this clip is parent to any other clips
            findClips(theClip[property], indentSpaces + 4);
        }
    }
}
findClips(_root, 0); // Find all clip instances descended from main timeline
```

For more information on recursion, see “Recursive Functions” in Chapter 9.

The `_name` property

As we saw earlier in this chapter under “Movie Clip Instance Names,” every instance’s name is stored as a string in the built-in property `_name`. We can use that property, as we saw in Example 13-1, to determine the name of the current clip or the name of some other clip in an instance hierarchy:

```

this._name;           // The current instance's name
this._parent._name   // The name of the clip that contains the current clip
    
```

The `_name` property comes in handy when we want to perform conditional operations on clips according to their identities. For example, here we duplicate the `seedClip` clip when it loads:

```

onClipEvent (load) {
    if (this._name == "seedClip") {
        this.duplicateMovieClip("clipCopy", 0);
    }
}
    
```

By checking explicitly for the `seedClip` name, we prevent infinite recursion—without our conditional statement, the `load` handler of each duplicated clip would cause the clip to duplicate itself.

The `_target` property

Every movie clip instance has a built-in `_target` property, which is a string that specifies the clip’s absolute path using the deprecated Flash 4 “slash” notation. For example, if `clipB` is placed inside `clipA`, and `clipA` is placed on the main timeline, the `_target` property of these clips is as follows:

```

_root._target           // Contains: "/"
_root.clipA._target     // Contains: "/clipA"
_root.clipA.clipB._target // Contains: "/clipA/clipB"
    
```

The `targetPath()` function

The `targetPath()` function returns a string that contains the clip’s absolute reference path, expressed using dot notation. The `targetPath()` function is the modern, object-oriented equivalent of `_target`. It takes the form:

```
targetPath(theClip)
```

where *theClip* is the identifier of the clip whose absolute reference we wish to retrieve. Here are some examples, using our familiar example hierarchy:

```

targetPath(_root);           // Contains: "_level0"
targetPath(_root.clipA);     // Contains: "_level0.clipA"
targetPath(_root.clipA.clipB); // Contains: "_level0.clipA.clipB"
    
```

The `targetPath()` function gives us the complete path to a clip, whereas the `_name` property gives us only the name of the clip. (This is analogous to having a complete file path versus just the filename.) So, we can use `targetPath()` to compose code that

controls clips based not only on their name but also on their location. For example, we might create a generic navigational button that, by examining its *targetPath()*, sets its own color to match the section of content within which it resides. See the example under the *Selection* object in the *Language Reference* for a demonstration of *targetPath()* in action.

Removing Clip Instances and Main Movies

We've seen how to create and refer to movie clips; now let's see how to turn them into so many recycled electrons (in other words, blow 'em away).

The manner in which we create an instance or a movie determines the technique we use to remove that instance or movie later. We can remove movies and instances explicitly using *unloadMovie()* and *removeMovieClip()*. Additionally, we can evict a clip implicitly by using *loadMovie()*, *attachMovie()*, or *duplicateMovieClip()* to place a new clip in its stead. Let's look at these techniques individually.

Using *unloadMovie()* with Instances and Levels

The built-in *unloadMovie()* function can remove any clip instance or main movie—both those created manually and those created via *loadMovie()*, *duplicateMovieClip()*, and *attachMovie()*. It can be invoked either as a global function or as a instance-level method:

```
unloadMovie(clipOrLevel); // Global function
clipOrLevel.unloadMovie(); // Method
```

In the global function form of *unloadMovie()*, *clipOrLevel* is a string indicating the path to the clip or level to unload. Because movie clips are converted to paths when used as strings, *clipOrLevel* can also be a movie clip reference. In the method form of *unloadMovie()*, *clipOrLevel* must be a reference to a movie clip object. The exact behavior of *unloadMovie()* varies according to whether it is used on a level or an instance.

Using *unloadMovie()* with levels

When applied to a level in the document stack (e.g., *_level0*, *_level1*, or *_level2*), *unloadMovie()* completely removes the target level and the movie that the level contains. Subsequent references to the removed level yield undefined. Removing document levels is the most common use of the *unloadMovie()* function:

```
unloadMovie("_level1");
_level1.unloadMovie();
```

Using *unloadMovie()* with instances

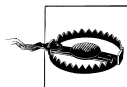
When applied to an instance (whether the instance is manually or programmatically created), *unloadMovie()* removes the *contents* of the clip, *but it does not remove the*

clip itself! The timeline and canvas of the clip are removed, but an empty shell remains on stage. That shell can be referenced until the instance is removed permanently via *removeMovieClip()* or until the span of frames on which the instance resides ends. Furthermore, any *onClipEvent()* handlers on the shell remain active.

This “partial” deletion of the instance presents an interesting possibility; it lets us maintain a generic container clip whose contents can be changed repeatedly via *loadMovie()* and *unloadMovie()*. For example, we could use a single clip to load sections of a web site or images in a photo album. The following series of statements demonstrates the technique with an instance called *clipA* (though in a real application, these statements would include the appropriate preloader code):

```
clipA.loadMovie("section1.swf"); // Load a document into clipA
clipA.unloadMovie();             // Unload the document, leaving clipA intact
clipA.loadMovie("section2.swf"); // Load another document into clipA
```

One note of caution with this approach: when used on an instance, *unloadMovie()* removes all custom properties of the clip contained by the instance. Physical properties—such as *_x* and *_alpha*—persist, but custom variables, functions, and event handler callbacks are lost.



If you use the global function form of *unloadMovie()* with a nonexistent clip or level instance as its argument, the clip from which you invoked the *unloadMovie()* function will, itself, unload.

For example, if *_level1* is undefined, and we issue the following code from the main timeline of *_level0*, then *_level0* will unload:

```
unloadMovie(_level1);
```

Yes, there’s some logic to this behavior, but we’ll cover that later in this chapter under “Method Versus Global Function Overlap Issues.” You can avoid the problem by using a string when specifying the *clipOrLevel* argument of *unloadMovie()* or by checking explicitly that *clipOrLevel* exists before unloading it. Here’s an example of each approach:

```
unloadMovie("_level1"); // clipOrLevel specified as a string
if (_level1) {          // Explicit check to make sure level exists
    unloadMovie(_level1);
}
```

Using *removeMovieClip()* to Delete Instances

To delete instances created via *duplicateMovieClip()*, *attachMovie()* or *createEmptyMovieClip()*, we can use *removeMovieClip()*. We delete an instance when it is no longer needed in our application, such as when an enemy spaceship is destroyed or an alert window is closed. Note that *removeMovieClip()* works on duplicated or attached instances only. It cannot delete a manually created instance or

a main movie. Like *unloadMovie()*, *removeMovieClip()* can be used in both method and global function form (though the syntax is different, the effect is the same):

```
removeMovieClip(theClip) // Global function  
theClip.removeMovieClip() // Method
```

In the global function form of *removeMovieClip()*, *theClip* is a string indicating the path to the clip to remove. Because movie clips are converted to paths when used as strings, *theClip* can also be a movie clip reference. In the method form of *removeMovieClip()*, *theClip* must be a reference to a movie clip object.

Unlike using *unloadMovie()*, deleting an instance via *removeMovieClip()* completely obliterates the entire clip object, leaving no shell or trace of the clip and its properties. When we execute *theClip.removeMovieClip()*, future references to *theClip* yield undefined.

Removing Manually Created Instances Manually

Clip instances created manually in the Flash authoring tool have a limited life span—they are removed when the playhead enters a keyframe that does not include them. Hence, manually created movie clips live in fear of the almighty blank keyframe.

Remember that when a movie clip disappears from the timeline, it ceases to exist as a data object. All variables, functions, methods, and properties that have been defined inside it are lost. Therefore, if we want a clip's information or functions to persist, we should be careful about removing the clip manually, and we should ensure that the span of frames on which the clip resides extends to the point where we need that clip's information. (In fact, to avoid this worry entirely, we should attach most permanent code to a frame in the main movie timeline or to the *_global* object.) To hide a clip while it's present on the timeline, simply set the clip's *_visible* property to false. Setting a clip's *_x* property to a very large positive number or very small negative number will also hide it from the user's view, but this approach is discouraged because the clip is still rendered off screen, consuming resources.

Method Versus Global Function Overlap Issues

As we've mentioned several times during this chapter, some movie clip methods have the same name as equivalent global functions. You can see this for yourself in the Flash authoring tool. Open the Actions panel, make sure you're in Expert Mode, and then take a look in the Actions folder under Movie Control and Movie Clip Control. You'll see a list of Actions, including *gotoAndPlay()*, *gotoAndStop()*, *nextFrame()*, and *unloadMovie()*. These Actions are also available as movie clip methods. The duplication is not purely a matter of categorization; the Actions are global functions, fully distinct from the corresponding movie clip methods.

So, when we execute:

```
theClip.gotoAndPlay(5);
```

we're accessing the *movie clip method* named *gotoAndPlay()*. But when we execute:

```
gotoAndPlay(5);
```

we're accessing the *global function* called *gotoAndPlay()*. These two commands have the same name, but they are not the same thing. The *gotoAndPlay()* global function operates on the current instance or movie. The *gotoAndPlay()* method operates on the clip object through which it is invoked. Most of the time, this subtle difference is of no consequence. But for some overlapping method/function pairs, this difference is potentially quite vexing.

Some global functions require a *target* parameter that specifies the clip on which the function should operate. This *target* parameter is not required by the comparable clip methods because the methods automatically operate on the clips through which they are invoked. For example, in its method form, *unloadMovie()* works like this:

```
theClip.unloadMovie();
```

As a method, *unloadMovie()* is invoked without parameters, and it automatically affects *theClip*. But in its global function form, *unloadMovie()* works like this:

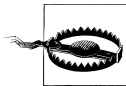
```
unloadMovie(target);
```

The global function version of *unloadMovie()* requires *target* as a parameter that specifies which movie to unload. Why should this be a problem? Well, the first reason is that we may mistakenly expect to be able to unload the current document by using the global version of *unloadMovie()* without any parameters, as we'd use *gotoAndPlay()* without parameters:

```
unloadMovie();
```

This format does *not* unload the current clip. It causes a "Wrong number of parameters" error. The second reason that *target* parameters in global functions can cause problems is a little more complex and can be quite a pain to track down if you're not expecting it. To supply a *target* clip to a global function that requires a *target* parameter, we can use either a string, which expresses the path to the clip we wish to affect, or a clip reference. For example:

```
unloadMovie(_level1); // Target clip is a reference
unloadMovie("_level1"); // Target clip is a string
```



We can use a reference simply because references to clip objects are converted to movie clip paths when used in a string context. This is simple enough, but if the *target* parameter resolves to an empty string or an undefined value, the *function operates on the current timeline!*

These examples demonstrate how an incorrect target clip reference can unintentionally unload the current timeline:

```

unloadMovie(x); // If x doesn't exist, x yields undefined, so
                // the function operates on the current timeline
unloadMovie(""); // The target is the empty string, so the function operates
                // on the current timeline

```

This can cause some quite unexpected results. Consider what happens if we refer to a level that doesn't exist:

```
unloadMovie(_level1);
```

If `_level1` is empty, the interpreter resolves the reference as though it were an undeclared variable. This yields `undefined`, so the function operates on the current timeline, not `_level1`! So, how do we accommodate this behavior? There are a few options. We can check for the existence of our target before executing a function on it:

```

if (_level1) {
    unloadMovie(_level1);
}

```

Or, we can choose to always use a string to indicate the path to our target. If the path specified in our string does not resolve to a real clip, the function fails silently:

```
unloadMovie("_level1");
```

In some cases, we can use the equivalent numeric function for our operation:

```
unloadMovieNum(1);
```

Finally, we can choose to avoid the issue altogether by always using clip methods:

```
_level1.unloadMovie();
```

For reference, here are the troublemakers (the `ActionScript` global functions that take *target* parameters):

```

duplicateMovieClip()
loadMovie()
loadVariables()
print()
printAsBitmap()
removeMovieClip()
startDrag()
unloadMovie()

```

If you're experiencing unexplained problems in a movie, you may want to check this list to see if you're misusing a global function. When passing a clip reference as a *target* parameter, be sure to double-check your syntax.

Drawing in a Movie Clip at Runtime

Flash MX introduces the ability to draw strokes, curves, shapes, and fills in a movie clip using the *Drawing API*, a collection of methods for drawing at runtime. The

Drawing API methods are documented in the *Language Reference*, under the following entries:

```
MovieClip.beginFill()  
MovieClip.beginGradientFill()  
MovieClip.clear()  
MovieClip.curveTo()  
MovieClip.endFill()  
MovieClip.lineStyle()  
MovieClip.lineTo()  
MovieClip.moveTo()
```

The *Drawing API* uses the concept of a *drawing pen* (or simply *pen*) to refer to the current drawing position, similar to the drawing pen used in old line plotters. Initially, the drawing pen resides at the registration point of a movie clip. Using the drawing methods, we can:

- Move the pen without drawing any lines or fills, via *MovieClip.moveTo()*
- Draw a straight line from the pen's current position to a specific point, via *MovieClip.lineTo()*
- Draw a curved line from the pen's current position to a specific point, via *MovieClip.curveTo()*
- Draw a shape, via *MovieClip.beginFill()* and *MovieClip.endFill()* or *MovieClip.beginGradientFill()* and *MovieClip.endFill()*.

Before, during, or after drawing, we can specify the characteristics of the drawing stroke used in any drawing operation, via *MovieClip.lineStyle()*. To remove a drawing, we use *MovieClip.clear()*.

Notice that the Drawing API does not include methods for drawing shapes such as a triangle, rectangle, or circle. We must draw these using the primitive drawing methods, as demonstrated under *MovieClip.beginFill()* in the *Language Reference*.

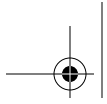
Detailed coverage of the drawing methods is left to the *Language Reference*. For now, let's take a look at a couple of examples showing the general use of the Drawing API.

Example 13-5 creates a 100-pixel-long, 2-pixel-thick, straight line extending to the right from the current clip's registration point.

Example 13-5. Drawing a straight line

```
// Set stroke to 2-point  
this.lineStyle(2);  
// Draw the line  
this.lineTo(100,0);
```

Example 13-6 creates a 200-pixel-wide, red square centered on the current clip's registration point.



Example 13-6. Drawing a square

```
// Set stroke to 3-point
this.lineStyle(3);
// Move the pen to 100 pixels left and above the registration point
this.moveTo(-100,-100);
// Start our red square shape
this.beginFill(0xFF0000);
// Draw the lines of our square
this.lineTo(100, -100);
this.lineTo(100, 100);
this.lineTo(-100, 100);
this.lineTo(-100, -100);
// Close our red square shape
this.endFill();
```

For a variety of interesting applications of the Drawing API (including drawing arcs, polygons, dashed lines, stars, and wedges), see:

<http://www.formeequalsfunction.com/downloads/drawmethods.html>

Using Movie Clips as Buttons

As of Flash Player 6, movie clips (but not main movies) have all the features previously reserved for button symbols. A movie clip can dynamically define and redefine button event handlers, a *hit region*, and *Up*, *Over*, and *Down* states; and, unlike button symbols, movie clips can be instantiated dynamically at runtime. If you are implementing simple interactivity at authoring time, you can continue to use button symbols happily. But if you are generating complex, dynamic button behavior at runtime, you'll want to use movie clips.

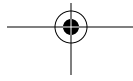
The first step in implementing button-like behaviors for a movie clip is to define one or more button events for the clip. Normally, this means assigning a callback function to a predefined button event property (shown next), but the *on(event)* button syntax can also be applied to a movie clip directly in the authoring tool. The following code creates a movie clip, adds a text field to it, and then defines the *onRelease()* button event handler:

```
// Create clip
this.createEmptyMovieClip("submit_mc", 1);

// Add text field
this.submit_mc.createTextField("submit_txt", 1, 0, 0, 50, 20);
this.submit_mc.submit_txt.text = "Submit";

// Define button handler
this.submit_mc.onRelease = function () {
    trace("You pressed the submit button.");
}
```

As is, this movie clip operates perfectly well as a button. But suppose we want to make the word "Submit" easier for users to click. To define a larger *hit area* (the



region that activates the button) for our movie clip, we'll create a separate *hit area movie clip* as follows:

1. Create a movie clip the size of the desired hit area.
2. Conceal the hit area movie clip by setting its `_visible` property to `false`.
3. Assign the hit area movie clip to the `hitArea` property of the clip with the button events.

The movie clip that acts as the hit area can reside on any timeline, but it is normally placed inside the clip with the button events so that the hit area moves and scales with its parent. For example, the following code uses the Drawing API to create a hit area movie clip larger than the word "Submit":

```
// Create hit_mc inside submit_mc
this.submit_mc.createEmptyMovieClip("hit_mc", 0);

// Draw a rectangle in hit_mc
this.submit_mc.hit_mc.moveTo(-30,-15);
this.submit_mc.hit_mc.beginFill(0xFF0000);
this.submit_mc.hit_mc.lineTo(80, -15);
this.submit_mc.hit_mc.lineTo(80, 35);
this.submit_mc.hit_mc.lineTo(-30, 35);
this.submit_mc.hit_mc.lineTo(-30, -15);
this.submit_mc.hit_mc.endFill();

// Hide the hit area movie clip
this.submit_mc.hit_mc._visible = false;

// Set hit_mc as submit_mc's hit area
this.submit_mc.hitArea = this.submit_mc.hit_mc;
```

So far, our example does not change visually when the mouse is pressed or moved over the Submit button. Using ActionScript, visual changes to a button can be implemented dynamically within each button event handler. For example, we can bold the word "Submit" from our `submit_mc`'s `onRollOver()` event as follows:

```
// Bold text on roll over
this.submit_mc.onRollOver = function () {
    this.submit_txt.setTextFormat(new TextFormat(null, null, null, true));
}

// Normal text on roll out
this.submit_mc.onRollOut = function () {
    this.submit_txt.setTextFormat(new TextFormat(null, null, null, false));
}
```

Alternatively, if we are creating our movie clip in the authoring tool, we can define button-state visual changes by creating keyframes with the special labels `_up`, `_over`, and `_down`, corresponding to the button states Up (mouse is not over the button), Over (mouse is over the button), and Down (button is being pressed). (Use the Property inspector to assign a label to a keyframe.) When the mouse interacts with the movie clip, Flash automatically moves the clip's playhead to the appropriately

labeled frame. For example, when the mouse moves over the clip, Flash performs the equivalent of this:

```
theClip.gotoAndStop("_over");
```

At each of the labeled button-state frames, we can issue a *play()* command to create animated button effects. However, we must be sure to also issue a *stop()* command on frame 1 of the movie clip, which prevents the movie from playing through all its button states. Normally, this first frame is labeled *_up* (the default inactive state for the button).

Movie clips with button events can also use the button properties enabled (used to toggle button behavior on and off), *useHandCursor* (used to prevent or enable changes to the mouse pointer when it is over the button), and *trackAsMenu* (used to modify the button's *onRelease()* handler requirements, enabling menu-style behavior). For example, we can suppress the display of the hand mouse pointer for our *submit_mc* clip as follows:

```
submit_mc.useHandCursor = false;
```

Typical web browser Submit buttons do not show a hand pointer on rollover. For deeper discussion of the button properties available to movie clips, see the *MovieClip* class in the *Language Reference*.

Nested button behavior is not supported for movie clips. That is, movie clips inside a movie clip with button behavior cannot themselves define button behaviors. Furthermore, if a mask and a masked movie clip both define button handlers, only the mask movie clip's button handlers are activated.

The various types of event handlers for buttons and movie clips have important scope differences, which are discussed in Chapter 10 under "Event Handler Scope" and summarized in Table 10-1.

Input Focus and Movie Clips

As of Flash Player 6, movie clips instances (but not main movies) can receive keyboard input focus, allowing them to be controlled by keystrokes, much like a button object is controlled. Input focus is most commonly used by components that take input in complex graphical user interfaces. For example, a list box can allow the user to scroll through its items via the up and down arrow keys. The Enter key activates the selected item.



Generally speaking, if a keyboard-driven action is not tied directly to a specific movie clip, you should use the *Key* object's listener events—not movie clips with input focus—to trap the keystrokes. See *Key* in the *Language Reference*.

Despite what you might assume, movie clips do not receive keyboard input focus when they are clicked. Instead, they receive focus either due to the Tab key being pressed or, programmatically, by using the *setFocus()* method. To allow a movie clip to receive input focus programmatically, we must explicitly set its *focusEnabled* property to true. In contrast, movie clips with button handlers can always receive input focus, even when *focusEnabled* is false. Once *focusEnabled* is true for a clip, we can focus the clip programmatically with *Selection.setFocus()*. The user can also focus the clip with the Tab key, provided that the movie clip's *tabEnabled* property is true. The movie clip's position in the tab order is dictated by its *tabIndex* property.

When a movie clip has input focus, its *onKeyUp()* and *onKeyDown()* event handlers become active in their callback-function form. From these handlers, we can implement keyboard-specific behaviors, such as expanding a hierarchical menu when the right arrow key is pressed or jumping to the nearest item in a list when a letter is pressed.

We can detect when a movie clip gains and loses input focus via the *onSetFocus()* and *onKillFocus()* events, defined both by the *MovieClip* class and the *Selection* object.

By default, keyboard focus is indicated by a yellow rectangle. The highlight is useful for debugging the tab order, but it may not be visually appealing. To remove the yellow rectangle, set the *MovieClip._focusrect* property to false. If you do this, you should give the end user some indicator as to which screen element has keyboard focus at runtime. To do so, use the clip's *onSetFocus()* and *onKillfocus()* handlers, as shown in the next example, to toggle some indicator of focus.



In Flash Player 6, moving the mouse while a movie clip or button has focus removes focus from that clip or button.

The following simplified code shows how an item in a shopping basket might respond to being focused and deleted via the X key. In a real application, the behavior would most likely be implemented at the class level rather than on a specific movie clip instance. The *doHighlight()* and *doRemoveHighlight()* methods are left as an exercise for the reader. You might use those handlers to jump the playhead to a label that shows or hides some focus indicator, such as the dotted line or thick border typically seen around the button with focus in Windows or Macintosh dialog boxes.

```
// Allow user to focus item_mc via Tab key
item_mc.tabEnabled = true;

// Allow Selection.setFocus() to focus item_mc
item_mc.focusEnabled = true;
```

```

// Highlight item_mc when it is focused
item_mc.onSetFocus = function () {
    this.doHighlight();
}

// Remove highlight from item_mc when it loses focus
item_mc.onKillFocus = function () {
    this.doRemoveHighlight();
}

// Remove item_mc when "x" is pressed and item_mc has focus
item_mc.onKeyDown = function () {
    if (Key.getCode() == 88) { // The keycode for "x" is 88
        this.removeMovieClip();
    }
}

```

For much more information on movie clip input focus, consult the *Language Reference* entries for the properties and methods discussed in this section.

Building a Clock with Clips

Now that you've learned the fundamentals of movie clip programming, let's put this knowledge to use by creating a sample analog clock application, which exemplifies the typical role of movie clips as basic content containers. See also the various versions of the multiple-choice quiz posted at the online Code Depot.

In this chapter we saw how to create movie clips with *attachMovie()* and how to set movie clip properties with the dot operator. With these relatively simple tools and a little help from the *Date* and *Color* classes, we have everything we need to make a clock with functional hour, minute, and second hands.

First, we'll make the face and hands of the clock using the following steps (notice that we don't place the parts of our clock on the main Stage—our clock will be generated entirely through *ActionScript*):

1. Create a new, empty Flash movie.
2. Create a movie clip symbol, named *clockFace*, that contains a 100-pixel-wide black circle shape, centered on the clip's registration point.
3. Create a movie clip symbol, named *hand*, that contains a 50-pixel-long, vertical red line.
4. Select the line in *hand*; then choose Window → Info (Flash MX) or Window → Panels → Info (Flash 5).
5. Because we want the hands to rotate around the center of the clock, we must position the line so that one end is at the registration point (the center) of the hand clip. Therefore, position the bottom of the line at the center of the clip by setting the line's x-coordinate to 0 and its y-coordinate to -50.

Now we have to export our *clockFace* and *hand* symbols, so that instances of them can be attached dynamically to our movie:

1. In the Library, select the *clockFace* clip; then select Linkage from the pop-up Options menu. The Linkage Properties dialog box appears.
2. Select the Export For ActionScript checkbox.
3. In the Identifier box, type **clockFace**, and then click OK.
4. Repeat Steps 1 through 3 to export the *hand* clip, giving it the identifier **hand**.

The face and hands of our clock are complete and ready to be attached to our movie. Now let's write the script that places the clock assets on stage and refreshes them with each passing second:

1. Add the script shown in Example 13-7 to frame 1 of *Layer 1* of the main timeline.
2. Rename *Layer 1* to *scripts*.

Skim Example 13-7 in its entirety first; then we'll dissect it.

Example 13-7. An analog clock

```
// Create clock face and hands
attachMovie("clockFace", "clockFace_mc", 0);
attachMovie("hand", "secondHand_mc", 3);
attachMovie("hand", "minuteHand_mc", 2);
attachMovie("hand", "hourHand_mc", 1);

// Position and size the clock face
clockFace_mc._x = 275;
clockFace_mc._y = 200;
clockFace_mc._height = 150;
clockFace_mc._width = 150;

// Position, size, and color the clock hands
secondHand_mc._x = clockFace_mc._x;
secondHand_mc._y = clockFace_mc._y;
secondHand_mc._height = clockFace_mc._height / 2.2;
secondHandColor = new Color(secondHand_mc);
secondHandColor.setRGB(0xFFFFFF);
minuteHand_mc._x = clockFace_mc._x;
minuteHand_mc._y = clockFace_mc._y;
minuteHand_mc._height = clockFace_mc._height / 2.5;
hourHand_mc._x = clockFace_mc._x;
hourHand_mc._y = clockFace_mc._y;
hourHand_mc._height = clockFace_mc._height / 3.5;

// Update the rotation of hands with each passing frame
function updateClock () {
    var now = new Date();
    var dayPercent = (now.getHours() > 12 ?
        now.getHours() - 12 : now.getHours()) / 12;
```

Example 13-7. An analog clock (continued)

```

var hourPercent = now.getMinutes()/60;
var minutePercent = now.getSeconds()/60;
hourHand_mc._rotation = 360 * dayPercent + hourPercent * (360 / 12);
minuteHand_mc._rotation = 360 * hourPercent;
secondHand_mc._rotation = 360 * minutePercent;
}

// Update the clock every 100 milliseconds
setInterval(updateClock, 100);

```

That’s a lot of code, so let’s review it.

We first attach the `clockFace` clip and assign it a depth of 0 (we want it to appear behind our clock’s hands):

```
attachMovie("clockFace", "clockFace_mc", 0);
```

Next, we attach three instances of the *hand* symbol, assigning them the names `secondHand_mc`, `minuteHand_mc`, and `hourHand_mc`. Each hand resides on its own layer in the programmatically generated content stack above the main timeline. The `secondHand_mc` clip (depth 3) sits on top of the `minuteHand_mc` clip (depth 2), which sits on top of the `hourHand_mc` clip (depth 1):

```
attachMovie("hand", "secondHand_mc", 3);
attachMovie("hand", "minuteHand_mc", 2);
attachMovie("hand", "hourHand_mc", 1);
```

We want our clock centered—not in the top-left corner of the Stage—so, we center the `clockFace_mc` clip on stage and make it larger using the `_height` and `_width` properties. In this example, we assume the movie size is the default (550 × 400 pixels), but we could have used `Stage.width` and `Stage.height` to dynamically retrieve the dimensions of the movie at runtime. As an exercise, use `Stage.onResize()` to keep the clock centered even when the movie is resized.

```
clockFace_mc._x = 275;
clockFace_mc._y = 200;
clockFace_mc._height = 150;
clockFace_mc._width = 150;
```

Next, we move the `secondHand_mc` clip onto the clock and make it almost as long as the radius of the `clockFace_mc` clip:

```
secondHand_mc._x = clockFace_mc._x;
secondHand_mc._y = clockFace_mc._y;
secondHand_mc._height = clockFace_mc._height / 2.2;
```

Remember that the line in the *hand* symbol is red, so all our *hand* instances thus far are also red. To make our `secondHand_mc` clip stand out, we color it white using the `Color` class. Note the use of the hexadecimal color value `0xFFFFFF` (see the *Color* Class in the *Language Reference* for more information on manipulating color):

```
// Create a new Color object to control secondHand_mc
secondHandColor = new Color(secondHand_mc);
```

```
// Assign secondHand_mc the color white
secondHandColor.setRGB(0xFFFFFFFF);
```

Next, we set the position and size of the `minuteHand_mc` and `hourHand_mc` clips, just as we did for the `secondHand_mc` clip:

```
// Place minuteHand_mc on top of clockFace_mc
minuteHand_mc._x = clockFace_mc._x;
minuteHand_mc._y = clockFace_mc._y;
// Make minuteHand_mc shorter than secondHand_mc
minuteHand_mc._height = clockFace_mc._height / 2.5;
// Place hourHand_mc on top of clockFace_mc
hourHand_mc._x = clockFace_mc._x;
hourHand_mc._y = clockFace_mc._y;
// Make hourHand_mc the shortest of all
hourHand_mc._height = clockFace_mc._height / 3.5;
```

Now we have to set the rotation of the hands on the clock to reflect the current time. However, we don't just want to set the rotation once. We want to set it repeatedly, so that our clock hands animate over time. Therefore, we put our rotation code in a function called `updateClock()`, which we'll call periodically:

```
function updateClock () {
    // Store the current time in now
    var now = new Date();

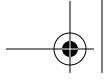
    // getHours() works on a 24-hour clock. If the current hour is greater
    // than 12, we subtract 12 to convert to a regular 12-hour clock.
    var dayPercent = (now.getHours() > 12 ?
        now.getHours() - 12 : now.getHours()) / 12;

    // Determine how many minutes of the current hour have passed, as a percentage
    var hourPercent = now.getMinutes()/60;

    // Determine how many seconds of the current minute have passed, as a percentage
    var minutePercent = now.getSeconds()/60;

    // Rotate the hands by the appropriate amount around the clock
    hourHand_mc._rotation = 360 * dayPercent + hourPercent * (360 / 12);
    minuteHand_mc._rotation = 360 * hourPercent;
    secondHand_mc._rotation = 360 * minutePercent;
}
```

The first task of `updateClock()` is to retrieve and store the current time. This is done by creating an instance of the `Date` class and placing it in the local variable `now`. Next we determine, as a percentage, how far around the clock each hand should be placed—much like determining where to slice a pie. The current hour always represents some portion of 12, while the current minute and second always represent some portion of 60. We assign the `_rotation` of each hand based on those percentages. We calculate the `hourHand_mc` clip's position to reflect not only the percent of the day but also the percent of the current hour.



Our clock is essentially finished. All that's left to do is call the `updateClock()` function with the following line of code, which refreshes the clock display every 100 milliseconds:

```
setInterval(updateClock, 100);
```

Test the movie to see if your clock works. If it doesn't, compare it to the sample clock `.fla` file provided at the online Code Depot, or check your code against Example 13-7. Updating the clock 10 times per second should create a nice smooth sweep of the second hand, but it may also be overkill (or a drain on performance of a larger piece).

Reader Exercise: Modify Example 13-7 to update the clock only once per second, and add a "tick-tock" sound synchronized with the second hand's movement (which should give the illusion of a nice firm snap).

Think of other ways to expand on the clock application: Can you draw the clock programmatically with the Drawing API? Can you make the clock more portable by turning it into a component? How about dynamically adding minute and hour markings on the `clockFace_mc` clip? Can you modify it to display different or multiple time zones? Can you create a stopwatch, complete with a reset button?

Onward!

We've come so far that there's not much more to move on to! Once you understand objects and movie clips thoroughly, you can tackle most ActionScript projects on your own. But there's still some interesting ground ahead. Chapter 14 shows how to create *MovieClip* subclasses and components, such as the UI Components shipped with Flash MX.

