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*) may contain independent submovies, christened *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 that contains another clip is referred to as that 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 independently animate the character's blinking eyes. 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 (like 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 may be thought of as the raw material used to produce programmatically generated content in Flash. For example, a movie clip may 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. At the end of this chapter we'll use movie clips as the hands on a clock and the answers in a multiple-choice quiz.

The "Objectness" of Movie Clips

As of Flash 5, movie clips can be manipulated like the objects we learned about in Chapter 12, *Objects and Classes*. We may retrieve and set the properties of a clip, and we may invoke built-in or custom methods on a clip. Unlike other objects, an operation performed on a clip may have a visible or audible result in the Player.

Movie clips are not truly a type of object; there is no *MovieClip* class or constructor, nor can we use an object literal to instantiate a movie clip in our code. 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 true objects is how they are allocated (created) and deallocated (disposed of, or freed). For details, see Chapter 15, *Advanced Topics*. Despite this technicality, however, we nearly always treat movie clips exactly like objects.

So how does the "objectness" of movie clips affect our use of them in ActionScript? Most notably, it dictates the way we control clips and examine their properties. Movie clips can be controlled directly through built-in methods. For example:

```
eyes.play();
```

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

```
ball._xscale = 90;
var radius = ball._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:

```
myClip.myVariable = 14;
x = myClip.myVariable;
```

Submovie 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 in an array element or a variable and 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 moveClip (clip, x, y) {
  clip._x = x;
  clip._y = y;
}
moveClip(ball, 14, 399);
```

Throughout the rest of this chapter, we'll learn the specifics of referencing, controlling, and manipulating movie clips as data objects.

Types of Movie Clips

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

- Main movies
- Regular movie clips
- Smart Clips

In addition to these three official varieties, we may define four further subcategories, based on our use of regular movie clips:

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

While these latter unofficial categories 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 contains the basic timeline and Stage present in every document. The main movie is the foundation for all the content in the document, including all other movie clips. We sometimes call the main movie the *main timeline*, the *main movie timeline*, the *main Stage*, or simply the *root*.

Main movies may be manipulated in much the same way as regular movie clips, however:

- A main movie cannot be removed from a *.suf* file (although a *.swf* file, itself, may be removed from the Flash Player).
- The following movie clip methods do not work when invoked on a main movie: *duplicateMovieClip()*, *removeMovieClip()*, *swapDepths()*.

- Event handlers cannot be attached to a main movie.
- Main movies can be referenced through the built-in, global `_root` and `_leveln` properties.

Note that while each *.swf* file contains only one main movie, more than one *.swf* may reside in the Flash Player at once—we may 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.

Regular Movie Clips

Regular movie clips are the most common and fundamental content containers; they hold visual elements and sounds and 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 being analogous to an HTML document object and regular movie clips as being analogous to that document's layer objects.

Smart Clips

Introduced in Flash 5, a *Smart Clip* is a regular movie clip that includes a graphical user interface used to customize the clip's properties in the authoring tool. Smart Clips are typically developed by advanced programmers to 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 Smart Clips in detail in Chapter 16, *ActionScript Authoring Environment*.

Process Clips

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

Process clips are ActionScript's unofficial alternative to the *setTimeout()* and *setInterval()* methods of the JavaScript window object.

Script Clips

Like a process clip, a *script clip* is an empty movie clip used not for content but for tracking some variable or executing some script. For example, we may use a script clip to hold event handlers that detect keypresses or mouse events.

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()* clip 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()*, the need for seed clips has diminished. However, we still use seed clips and *duplicateMovieClip()* when we wish to retain a clip's event handlers and transformations 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 like data 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 a movie clip constructor function, like this:

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

Instead, we 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.

Movie Clip Symbols and Instances

Just as all object instances are based on one class or another, 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. We must always have a movie clip symbol before we may generate a specific

clip object. Using a symbol, we can both manually and programmatically create clips to be rendered in a movie.

A movie clip that is rendered on the Stage is called an *instance*. Instances are the individual clip objects that can be manipulated with ActionScript; a *symbol* is the mold from which all 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 Symbol Properties dialog box appears.
2. In the Name field, type an identifier for the symbol.
3. Select the Movie Clip radio button.
4. Click OK.

Normally, the next step is to fill in 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 fashion an actual movie clip instance. It is, however, 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. Select the Movie Clip radio button.
5. Click OK.

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

Creating Instances

There are three ways to create a new instance based on a movie clip symbol. Two of these are programmatic; the other is strictly manual and is undertaken in the Flash authoring tool.

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 thus created should be named manually via the Instance panel. (We'll learn more about instance names later.) Refer to "Using Symbols and Instances" 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 independent copy as a completely separate clip. Both manually created and programmatically created clip instances may be duplicated. In other words, it's legal to duplicate a duplicate.

In practice, 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. The ***newName*** parameter is a string that specifies the identifier for the new instance, and ***depth*** is an integer that designates where, in the stack of programmatically generated clips, we want to place the new instance.

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

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

  where ***myClip*** is the name of the clip we wish to duplicate, and ***newName*** and ***depth*** both operate as before.

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:

```
ball.duplicateMovieClip("ball2", 0);
ball2._x += 100;
ball2._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 initial transformation of their seed clips. 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 an instance is duplicated, the instance's initial rotation is 45 degrees:

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

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

```
newClip._rotation += 10;
trace(newClip._rotation);  // Displays: 55
trace(seed._rotation);     // Displays: 45
```

By duplicating many instances in a row and adjusting the transformation of each duplicate slightly, we can achieve interesting compound transformations (the technique is shown under the *load* event in Example 10-2).

Using *duplicateMovieClip()* to duplicate clips via ActionScript 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.
- Assign the layer depth of a duplicated clip relative to other duplicated clips. (This was more of a concern in Flash 4, which did not allow the layer stack of a movie to be altered.)
- Copy a clip's event handlers.

These abilities give us advanced programmatic control over the content in a movie. A salient example is that of a spaceship game in which a missile movie clip might be duplicated when the ship's fire button is pressed. That missile clip might be moved programmatically, then placed behind an obstacle in the movie, and finally, be removed after colliding with an enemy craft. Manual clips do not offer that kind of flexibility. With a manually created clip, we must preordain the birth and death of the clip using the timeline and, in Flash 4, we couldn't change the clip's layer.

Creating instances with attachMovie()

Like *duplicateMovieClip()*, the *attachMovie()* method lets us create a movie clip instance; however, unlike *duplicateMovieClip()* it does not require a previously created instance—it creates a new instance directly from a symbol in a movie's Library. 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 Options menu, select Linkage. The Symbol Linkage Properties dialog box appears.
3. Select the Export This Symbol radio button.
4. In the Identifier field, type a unique name for the clip symbol. The name may be any string—often simply the same name as the symbol itself—but should be different from all other exported clip symbols.
5. Click OK.

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

```
myClip.attachMovie(symbolIdentifier, newName, depth);
```

where *myClip* is the name of the clip to which we want to attach the new instance. If *myClip* 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; *newName* is a string that specifies the identifier for the new instance we're creating; and *depth* is an integer that designates where in the host clip's layered stack to place the new instance.

When we attach an instance to another clip, that instance is positioned in the center of the clip, among the clip's layered stack (we'll discuss clip stacks soon). When we attach an instance to the main movie of a document, that instance is positioned in the upper-left corner of the Stage, at coordinates (0, 0).

Instance Names

When we create instances, we assign them identifiers, or *instance names*, that allow us to refer to them later. Notice how this differs from regular objects. When we create a normal data 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
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 normal data objects, clip instances are accessible in ActionScript via their instance names as soon as they are created. For example:

```
ball._y = 200;
```

Each clip's instance name is stored in its built-in property, `_name`, which can be both retrieved and set:

```
ball._name = "circle";  // Change ball's name to circle
```

When we change an instance's `_name` property, all future references to the instance must use the new name. For example, after the previous code executes, the `ball` reference ceases to exist, and we'd subsequently use `circle` to refer to the instance.

The manner in which an instance initially gets its instance name depends on how it was created. Programmatically generated instances are named by the function that creates them. Manually created instances are normally assigned explicit instance names in the authoring tool through the Instance panel, as follows:

1. Select the instance on stage.
2. Select Modify → Instance.
3. Enter the instance name into the Name field.

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).

Because instance names are identifiers, we must compose them according to the rules for creating a legal identifier, as described in Chapter 14, *Lexical Structure*. Most notably, instance names should not begin with a number, nor include hyphens or spaces.

Importing External Movies

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). By managing content in separate files, we gain precise control over the download process. 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 have finished 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.

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()* may be used both as 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");
```

Because a movie clip reference is converted to a path when used as a string, *location* may 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 unexpected behavior—it loads the external *.swf* into the *current* timeline. See Part III, *Language Reference* for more details, or see "Method versus global function overlap issues," later in this chapter.

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

```
myClip.loadMovie(URL);
```

When used as a method, *loadMovie()* assumes we're loading the external *.swf* into *myClip*, so the *location* parameter required by the standalone *loadMovie()* function is not needed. We, therefore, supply only the path to the *.swf* to load via the *URL* parameter. Naturally, *URL* can be a local 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 *myClip* 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");
```

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.

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 may 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 may no longer attach instances from *movie1.swf*'s Library to `clipA`.

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()*.

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*—a simple bit of 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 in Part III for sample code. See also Example 10-4, which shows how to build a preloader using the *data* clip event.

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 *.suf* files overlap in the Player, one clip (the "higher" of the two) always covers up the other clip (the "lower" of the two). Simple enough in principle, but the main stack, which contains *all* the instances and *.swf* files, is actually divided into many smaller substacks. We'll look at these substacks individually first, then see how they combine to form the main stack. (The stack

in this discussion has no direct relation to the LIFO and FIFO stacks discussed in Chapter 11, *Arrays*.)

The Internal Layer Stack

Instances created *manually* in the Flash authoring tool reside in a stack called 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.

Furthermore, because multiple clips may 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.

As of Flash 5, 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 Programmatically Generated Clip Stack

Programmatically generated instances are stacked separately from the *manually* created instances held in the internal layer stack. Each instance has its own *programmatically generated clip stack* that hold clips created via *duplicateMovieClip()* and *attachMovie()*. The stacking order for these clips varies depending on how they were created.

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

A new instance generated via *attachMovie()* is always stacked above (i.e., in the foreground relative to) the clip to which it was attached. For example, suppose we have two clips—`X` and `Y`—in the internal layer structure 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 would appear from top to bottom in this order: `A`, `X`, `B`, `Y`, as shown in Figure 13-1.

Once a clip is generated, it too provides a separate space above its content for more programmatically generated clips. That is, we may attach clips to attached clips.

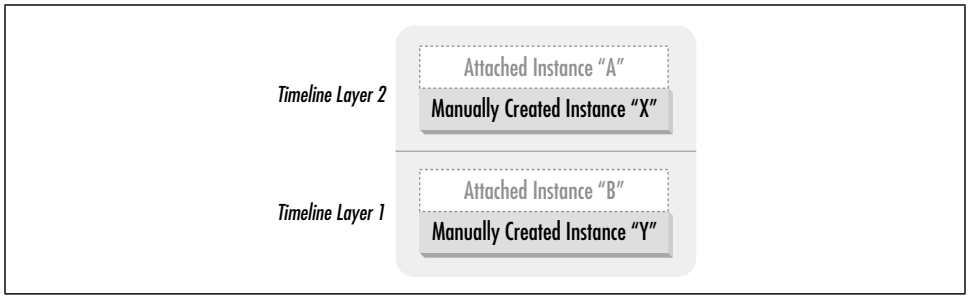

Figure 13-1. A sample instance stack

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

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

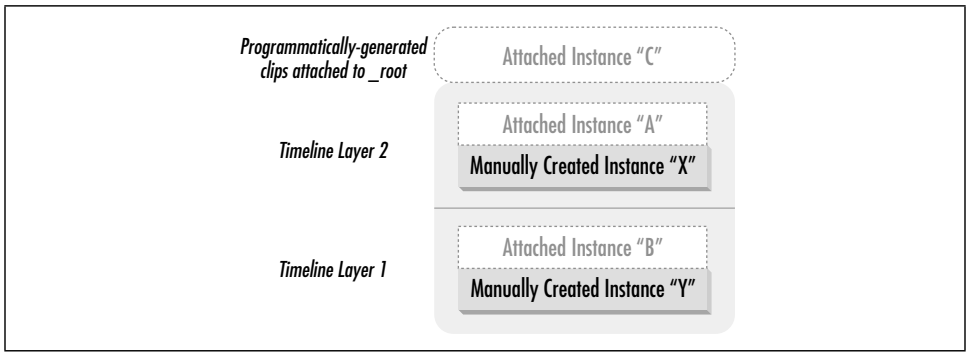

Figure 13-2. An instance stack showing a clip attached to _root

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

Each instance duplicated via *duplicateMovieClip()* is assigned to a programmatic stack in accordance with how that instance's seed clip was created:

- If the instance's seed clip was created manually (or was duplicated using *duplicateMovieClip()* from a clip that was created manually), then the new instance is placed in the stack above `_root`.
- If, on the other hand, the instance's seed clip was created with *attachMovie()*, then the new instance is placed in its seed clip's stack.

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. Similarly, if we create clip E by duplicating clip D (which is derived from clip X, which was created manually), 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 stack above X, with clip A. Figure 13-3 is worth a thousand words.

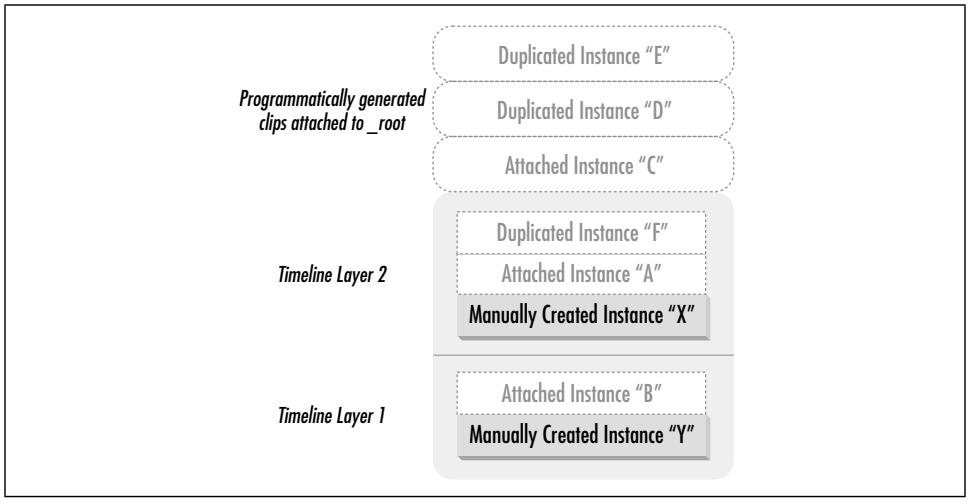

Figure 13-3. An instance stack showing various duplicated clips

Assigning depths to instances in the programmatically generated clip 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()* or *duplicateMovieClip()* function, 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 may be any integer and is measured from the bottom up, so –1 is lower than 0; 1 is higher than (i.e., in front of) depth 0; depth 2 is higher still, and so on. When two programmatically generated clips occupy the same position on screen, the one with the *greater* `depth` value is rendered in front of the other.

Layers are single-occupant dwellings. Only one clip may occupy a layer in the stack at a time—placing a clip into an occupied layer 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. There's no performance hit or

increase in memory consumption that results from having gaps in your depth assignments.

The .swf Document "_level" Stack

In addition to the internal layer stack and the programmatically generated clip 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 may 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 clip 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` would visually obscure the movie in `_level0`, but not remove it from the Player.

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

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.
- When manually created instances are initially loaded, code in their timeline and *load* 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

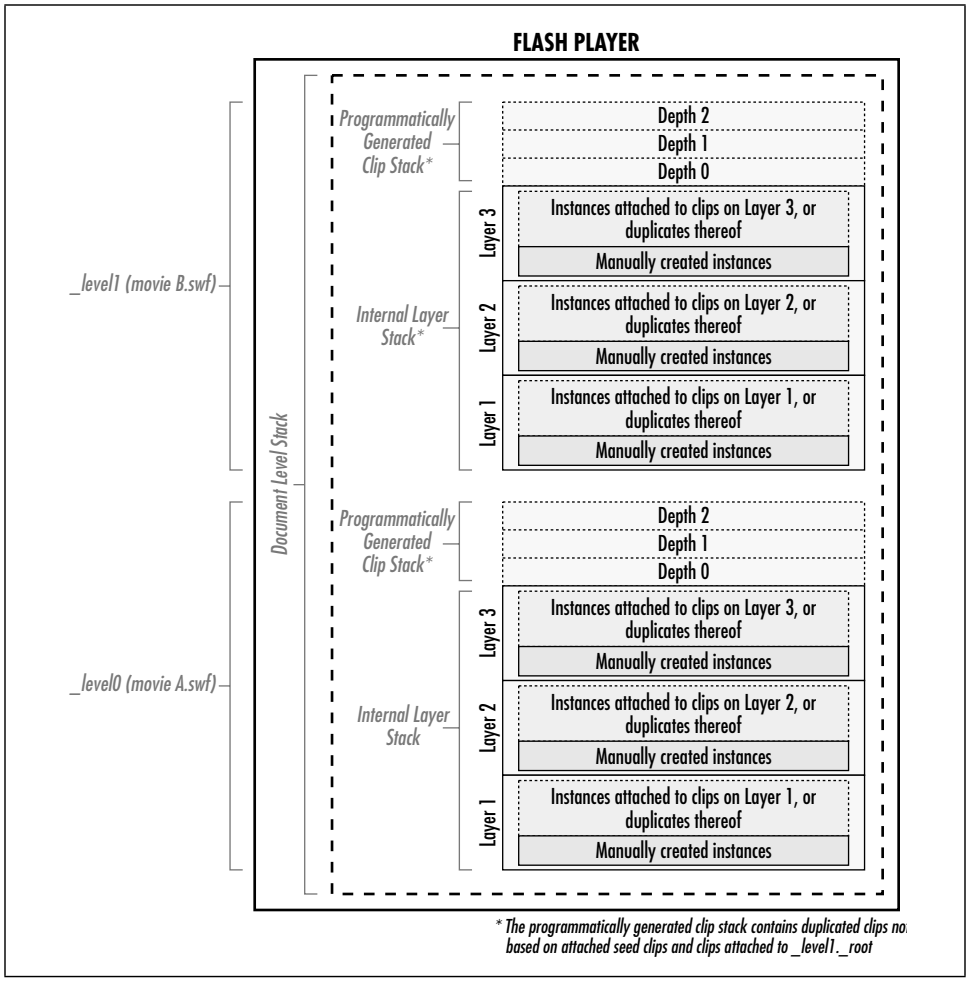

Figure 13-4. The complete Flash Player movie clip stack

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 the earlier sections, we learned 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 a 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 learned that movie clips are referred to by their *instance names*. For example:

```
trace(myVariable);  // Refer to a variable
trace(myClip);      // 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` placed on the main timeline of a document, we may refer to `clouds` from code attached to the main timeline as follows:

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

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 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 may 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`, we could place the following code on a frame in the `cloud` timeline:

```
_alpha = 60;
```

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

```
play();
```

This technique may be used on any timeline, including timelines of main movies. For example, the following two statements would be 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);
```

As we learned in Chapter 10, *Events and Event Handlers*, code in an instance's event handler may, like timeline code, also refer to properties and methods directly. For example, we could attach the following event handler to `cloud`. This handler sets a property of, and then invokes a method on, `cloud` without referring to the `cloud` instance explicitly:

```
onClipEvent (load) {
  _alpha = 60;
  stop();
}
```

However, not all methods may 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 "Method versus global function overlap issues."

Self-references with the this keyword

When we want to *explicitly* refer to the current instance from a frame in its timeline or from one of its event handlers, we may use the `this` keyword. For example, the

following statements would be synonymous when attached to a frame in the timeline of our `cloud` instance:

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

There are two reasons to use `this` to refer to a clip even when we can just refer to the clip directly. 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", 0);  // Invoke a method on an instance

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

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 moveTo (theClip, x, y) {
  theClip._x = x;
  theClip._y = y;
}

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

If you do a lot of object-oriented programming, be cautious when using the `this` keyword to refer to instances and movies. Remember that inside a custom method or an object constructor, `this` has a very different meaning and is not a reference to the current timeline. See Chapter 12 for details.

Referring to Nested Instances

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

Earlier, we saw briefly how we could navigate up or down from any point in the hierarchy of clip instances, much like you might 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;
```

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* it 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:

```
myClip._parent
```

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          // A reference to clipB (the current clip)
this._parent  // Another 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._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 try building the `clipA`, `clipB`, `clipC` hierarchy in Flash and using 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). See Figure 13-5.

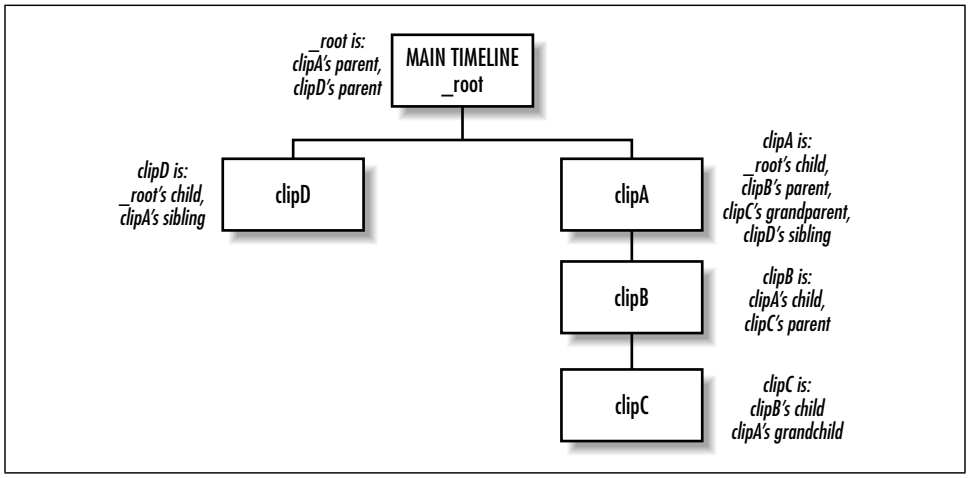

Figure 13-5. A sample clip hierarchy

Therefore, 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 the main timeline also contains `clipD`, which would make `clipD` a "sibling" of `clipA` because both would have the main timeline as their `_parent`. In that 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 learn 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 is the same no matter which clip it is referenced from. These are all equivalent:

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

Therefore, you can and should 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 that code is executed from within `circle`, it will cause the main movie to move 10 pixels to the right. When it is executed from within `square`, it will cause `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, it should 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` would be valid when used in a frame of the main timeline.

The `_root` property may happily 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.

Referencing other documents in the Player using _leveln

If we have multiple *.swf* files loaded in the document stack of the Flash Player, we may 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 may 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. By 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 ActionScript editor provides an Insert Target Path tool (shown in Figure 13-6) which lets us generate a clip reference visually, relieving the burden of creating it manually.

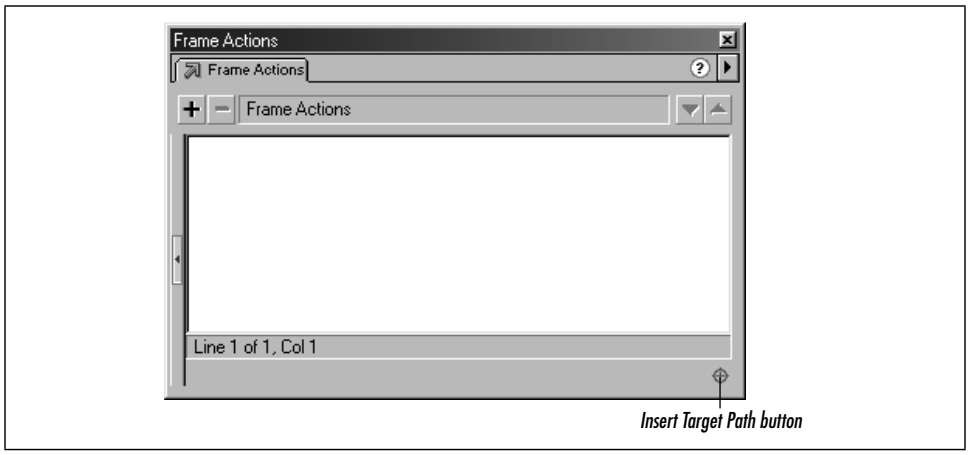

Figure 13-6. The Insert Target Path button

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.
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). See Table 2-1.

The Insert Target Path tool cannot generate 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 manually type the appropriate 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, *Operators*, and Chapter 12, *Objects and Classes*, the properties of an object may be retrieved via the dot operator or through the array-element access operator, `[]`. For example, the following two statements are equivalent:

```
myObject.myProperty = 10;
myObject["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 `myProperty`:

```
myObject["myProp" + "erty"];
```

We can apply the same technique to create our instance and movie references dynamically. We already learned 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 may 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:

```
// Stop 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

I started this chapter by saying that movie clips are effectively data objects in ActionScript. 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 those various clips in data containers, we can control them dynamically using the containers instead of explicit references to the clips. Example 13-1, which shows code that would be placed on a frame in the main timeline, uses data containers to store and control instances.

Example 13-1. 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 myClips = [clipA, clipA.clipB, clipA.clipB.clipC];
myClips[0].play();    // Play clipA
myClips[1]._x = 200;  // Place clipB 200 pixels from the Stage's left edge

// 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.

Using for-in to access movie clips

In Chapter 8, we learned 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 13-2 shows how to use a *for-in* loop to enumerate all the clips that reside on a given timeline.

Example 13-2. Finding Movie Clips on a Timeline

```
for (var property in myClip) {
  // Check if the current property of myClip is a movie clip
  if (typeof myClip[property] == "movieclip") {
    trace("Found instance: " + myClip[property]._name);

    // Now do something to the clip
    myClip[property]._x = 300;
    myClip[property].play();
  }
}
```

The *for-in* loop gives us enormously 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 we know the clip's name or not and whether the clip was created manually or programmatically.

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

Example 13-3. Recursively Finding All Movie Clips on a Timeline

```
function findClips (myClip, 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 myClip) {
    // Check if the current property of myClip is a movie clip
    if (typeof myClip[property] == "movieclip") {
      trace(indent + myClip[property]._name);
      // Check if this clip is parent to any other clips
      findClips(myClip[property], indentSpaces + 4);
    }
  }
}
findClips (_root, 0); // Find all clip instances descended from main timeline
```

For more information on function recursion, see "Recursive Functions" in Chapter 9, *Functions*.

The _name property

As we learned earlier in "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 13-2, to determine the name of the current clip or the name of some other clip in an instance hierarchy:

```
_name;              // The current instance's name
_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 (_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 specifying 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 those 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 Flash 5–syntax equivalent of `_target`. It takes the form:

```
targetPath(movieClip)
```

where *`movieClip`* is the identifier of the clip whose absolute reference we wish to retrieve. Here are some examples, using our now 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 "Selection.setSelection( ) Method" in Part III for a demonstration of *targetPath()* in action.

Whither Tell Target?

In Flash 4, *Tell Target* was our main tool for referring to movie clips. *Tell Target*, bless its soul, was an unwieldy tool and is rendered obsolete by the much more elegant object model introduced in Flash 5. The *Tell Target* function has been deprecated (i.e., retired from recommended use). Although we may still use the *tellTarget()* function to code in a Flash 4 manner, *tellTarget()* will likely disappear in the future.

Consider the following code, which uses *Tell Target* to play an instance named `closingSequence`:

```
Begin Tell Target ("closingSequence")
  Play
End Tell Target
```

As of Flash 5, we simply invoke the much more convenient and readable *play()* method on the `closingSequence` instance:

```
closingSequence.play();
```

Tell Target could also perform multiple operations on an instance within a code block, like so:

```
Begin Tell Target ("ball")
  (Set Property: ("ball", x Scale) = "5")
  Play
End Tell Target
```

As of Flash 5, the *with()* statement, described in Chapter 6, *Statements*, is the preferred way to achieve similar results:

```
with (ball) {
  _xscale = 5;
  play();
}
```

See Appendix C, *Backward Compatibility*, for more details on deprecated Flash 4 ActionScript and the preferred equivalents in Flash 5.

Removing Clip Instances and Main Movies

We've learned 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 created an instance or a movie determines the technique we use to remove that instance or movie later. We can explicitly remove movies and instances using *unloadMovie()* and *removeMovieClip()*. Additionally,

we may evict a clip implicitly by loading, attaching, or duplicating 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 both as a global function and as a method:

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

In global function form, `clipOrLevel` is a string indicating the path to the clip or level to unload. And due to automatic value conversion, `clipOrLevel` may also be a movie clip reference (movie clips are converted to paths when used as strings). In method form, `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`, `_level2`), *unloadMovie()* completely removes the target level and the movie that the level contains. Subsequent references to the 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 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 permanently removed via *removeMovieClip()* (or until the span of frames on which the instance resides ends). Furthermore, any clip event handlers on the shell remain active.

This partial deletion of instances presents an interesting possibility; it lets us maintain a generic container clip whose contents can be repeatedly changed via *loadMovie()* and *unloadMovie()*. For example, we may quite legitimately invoke the following function series on 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 and functions 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 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 attached and duplicated instances from the Player, we can use *removeMovieClip()*. Note that *removeMovieClip()* works on duplicated or attached instances only. It cannot delete a manually created instance or a main movie. Like *unloadMovie()*, *removeMovieClip()* may be used in both method and global function form (though the syntax is different, the effect is the same):

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

In global function form, *clip* is a string indicating the path to the clip to remove. Due to automatic value conversion, *clip* may also be a movie clip reference (movie clips are converted to paths when used as strings). In method form, *clip* must be a reference to a movie clip object.

Unlike *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 *clip*`.removeMovieClip()`, future references to *clip* yield `undefined`.

Removing Manually Created Instances Manually

Clip instances created manually in the Flash authoring tool also have a limited life span—they are removed when the playhead enters a keyframe that does not include them. Manually created movie clips, hence, 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 may 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 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.) To hide a clip while it's present on the timeline, simply position the clip outside the visible area of the Stage, and 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 should also suffice to hide it from the user's view without removing it from memory.

Built-in Movie Clip Properties

Unlike generic objects of the *Object* class, which have few built-in properties, each movie clip comes equipped with a slew of built-in properties. These properties describe, and can be used to modify, the clip's physical features. They are fundamental tools in the ActionScript programmer's toolkit.

All built-in movie clip property names begin with an underscore, which sets them apart from user-defined or custom properties. Built-in properties take the format:

```
_property
```

Built-in property names should be written in lowercase. However, because identifiers are case insensitive in ActionScript, it is possible—though not good form—to capitalize property names.

We're not going to go into heavy descriptions of the built-in properties right now; that information is listed in Part III. However, to get us thinking about properties and what they offer, Table 13-1 provides a list of the built-in movie clip properties and basic descriptions of their functions.

Table 13-1. The Built-in Movie Clip Properties

Property Name	Property Description
`_alpha`	Transparency level
`_currentframe`	Position of the playhead

Table 13-1. The Built-in Movie Clip Properties (continued)

Property Name	Property Description
`_droptarget`	Path to the clip or movie on which a dragged clip was dropped
`_framesloaded`	Number of frames downloaded
`_height`	Physical height, in pixels (of instance, not original symbol)
`_name`	Clip's identifier, returned as a string
`_parent`	Object reference to the timeline containing this clip
`_rotation`	Angle of rotation (in degrees)
`_target`	Full path to the clip, in slash notation
`_totalframes`	Number of frames in the timeline
`_url`	Network location of *.swf*
`_visible`	Boolean indicating whether movie clip is displayed
`_width`	Physical width, in pixels (of instance, not original symbol)
`_x`	Horizontal position, in pixels, from the left of the Stage
`_xmouse`	Horizontal location of the mouse pointer in the clip's coordinate space
`_xscale`	Horizontal size, as a percentage of the original symbol (or main timeline for movies)
`_y`	Vertical position, in pixels, from the top of the Stage
`_ymouse`	Vertical location of the mouse pointer in the clip's coordinate space
`_yscale`	Vertical size, as a percentage of the original symbol (or main timeline for movies)

There's no direct color property attached to instances or main movies. Instead of controlling color through a property, we must use the *Color* class to create an object that is used to control the color of a clip. The methods of a *Color* object let us set or examine the RGB values and transformations of a particular clip. To learn the specific details, see the "Color Class" in Part III.

Movie Clip Methods

In Chapter 12, we learned about a special type of property called a *method*, which is a function attached to an object. Methods are most commonly used to manipulate, interact with, or control the objects to which they are attached. To control movie clips in various programmatic ways, we may use one of the built-in movie clip methods. We may also define our own movie clip methods in an individual instance or in the Library symbol of a movie clip.

Creating Movie Clip Methods

To add a new method to a movie clip, we define a function on the clip's timeline (or in one of the clip's event handlers) or we assign a function to a property of the clip. For example:

```
// Create a method by defining a function on the timeline of a clip
function halfSpin() {
  _rotation += 180;
}
// Create a method by assigning a function literal to a property of a clip
myClip.coords = function() { return [_x, _y]; };
// This method applies a custom transformation to a clip
myClip.myTransform = function () {
  _rotation += 10;
  _xscale -= 25;
  _yscale -= 25;
  _alpha -= 25;
}
```

Invoking Movie Clip Methods

Invoking a method on a movie clip works exactly like invoking a method on any object. We supply the name of the clip and the name of the method, as follows:

```
myClip.methodName();
```

If the method requires arguments, we pass them along during invocation:

```
_root.square(5);   // Provide 5 as an argument to the square() method
```

As we learned earlier, when we're working on the timeline of a clip or in a clip's event handler, we may invoke most methods on the current clip directly, without specifying an instance identifier:

```
square(10);        // Invoke the custom square() method of the current clip
play();            // Invoke the built-in play() method of the current clip
```

But some built-in methods require an instance identifier; see "Method versus global function overlap issues."

Built-in Movie Clip Methods

Recall that the generic *Object* class equips all its member objects with the built-in methods *toString()* and *valueOf()*. Recall similarly that other classes define built-in methods that can be used by their member objects: *Date* objects have a *getHours()* method, *Color* objects have *setRGB()*, *Array* objects have *push()* and *pop()*, and so on. Movie clips are no different. They come equipped with a series of built-in methods that we use to control movie clips' appearance and behavior, to check

their characteristics, and even to create new movie clips. The movie clip methods are one of the central features of ActionScript. Table 13-2 gives an overview of the movie clip methods that are covered in depth in Part III.

Table 13-2. The Built-in Movie Clip Methods

Method Name	Method Description
attachMovie()	Creates a new instance
duplicateMovieClip()	Creates a copy of an instance
getBounds()	Describes the visual region occupied by the clip
getBytesLoaded()	Returns the number of downloaded bytes of an instance or a movie
getBytesTotal()	Returns the physical byte size of an instance or a movie
getURL()	Loads an external document (usually an *.html* file) into the browser
globalToLocal()	Converts main Stage coordinates to clip coordinates
gotoAndPlay()	Moves the playhead to a new frame and plays the movie
gotoAndStop()	Moves the playhead to a new frame and halts it there
hitTest()	Indicates whether a point is within a clip
loadMovie()	Brings an external *.suf* file into the Player
loadVariables()	Brings external variables into a clip or movie
localToGlobal()	Converts clip coordinates to main Stage coordinates
nextFrame()	Moves the playhead ahead one frame
play()	Plays the clip
prevFrame()	Moves the playhead back one frame
removeMovieClip()	Deletes a duplicated or attached instance
startDrag()	Causes the instance or movie to physically follow the mouse pointer around the Stage
stop()	Halts the playback of the instance or movie
stopDrag()	Ends any drag operation currently in progress
swapDepths()	Alters the layering of an instance in an instance stack
unloadMovie()	Removes an instance or main movie from a document level or host clip
valueOf()	A string representing the path to the instance in absolute terms, using dot notation

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. You'll see a long list of Actions

including *gotoAndPlay()*, *gotoAndStop()*, *nextFrame()*, and *unloadMovie()*. Those 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:

```
myClip.gotoAndPlay(5);
```

we're accessing the *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, the subtle difference is of no consequence. But for some overlapping method/function pairs, the difference is potentially quite vexing.

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

```
myClip.unloadMovie();
```

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

```
unloadMovie(target);
```

The global function 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 may 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 may use a reference simply because references to clip objects are converted to movie clip paths when used in a string context. Simple enough, but if the *target* parameter resolves to an empty string or an `undefined` value, the *function operates on the current timeline*! For example:

```
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 may check for the existence of our target before executing a function on it:

```
if (_level1) {
  unloadMovie(_level1);
}
```

We may 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 may use the equivalent numeric function for our operation:

```
unloadMovieNum(1);
```

Or, we may choose to avoid the issue altogether by always using methods:

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

For reference, here are the troublemakers (the Flash 5 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 that 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.

Applied Movie Clip Examples

We've now learned the fundamentals of movie clip programming. Let's put our knowledge to use by creating two very different applications, both of which exemplify the typical role of movie clips as basic content containers.

Building a Clock with Clips

In this chapter we learned 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 with 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. Start a new Flash movie.
2. Create a movie clip symbol named `clockFace` that contains a 100-pixel-wide black circle shape.
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 → Panels → Info.
5. 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 Options → Linkage. The Symbol Linkage Properties dialog box appears.
2. Select Export This Symbol.
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 positions them with each passing second:

1. Add the script shown in Example 13-4 to frame 1 of *Layer 1* of the main timeline.
2. Rename *Layer 1* to *scripts*.

Skim Example 13-4 in its entirety first, then we'll dissect it.

Example 13-4. An Analog Clock

```
// Create clock face and hands
attachMovie("clockFace", "clockFace", 0);
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);

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

// Position, size, and color the clock hands
secondHand._x = clockFace._x;
secondHand._y = clockFace._y;
secondHand._height = clockFace._height / 2.2;
secondHandColor = new Color(secondHand);
secondHandColor.setRGB(0xFFFFFF);
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
minuteHand._height = clockFace._height / 2.5;
hourHand._x = clockFace._x;
hourHand._y = clockFace._y;
hourHand._height = clockFace._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;
  var hourPercent = now.getMinutes()/60;
  var minutePercent = now.getSeconds()/60;
  hourHand._rotation = 360 * dayPercent + hourPercent * (360 / 24);
  minuteHand._rotation = 360 * hourPercent;
  secondHand._rotation = 360 * minutePercent;
}
```

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

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

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

Next we attach three instances of the hand symbol, assigning them the names secondHand, minuteHand, hourHand. Each hand resides on its own layer in the programmatically generated clip stack above the main timeline. The secondHand (depth 3) sits on top of the minuteHand (depth 2), which sits on top of the hourHand (depth 1):

```
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);
```

At this point our code would place the clock in the top-left corner of the Stage. Next, we move the clockFace clip to the center of the Stage and make it larger using the _height and _width properties:

```
clockFace._x = 275;          // Set the horizontal location
clockFace._y = 200;          // Set the vertical location
clockFace._height = 150;     // Set the height
clockFace._width = 150;      // Set the width
```

Then we move the secondHand clip onto the clock and make it almost as long as the radius of the clockFace clip:

```
// Place the secondHand on top of the clockFace
secondHand._X = clockFace._x;
secondHand._y = clockFace._y;
// Set the secondHand's size
secondHand._height = clockFace._height / 2.2;
```

Remember that the line in the hand symbol is red, so all our hand instances thus far are red. To make our secondHand 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 Part III for more information on manipulating color):

```
// Create a new Color object to control secondHand
secondHandColor = new Color(secondHand);
// Assign secondHand the color white
secondHandColor.setRGB(0xFFFFFF);
```

Next we set the position and size of the minuteHand and hourHand, just as we did for the secondHand:

```
// Place the minuteHand on top of the clockFace
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
// Make the minuteHand shorter than the secondHand
minuteHand._height = clockFace._height / 2.5;
```

```
// Place the hourHand on top of the clockFace
hourHand._x = clockFace._x;
hourHand._y = clockFace._y;
// Make the hourHand the shortest of all
hourHand._height = clockFace._height / 3.5;
```

Now we have to set the rotation of our hands on the clock according to the current time. However, we don't just want to set the rotation once. We want to set it repetitively so that our clock animates as time passes. Therefore, we put our rotation code in a function called *updateClock()*, which we'll call repeatedly:

```
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._rotation = 360 * dayPercent + hourPercent * (360 / 24);
  minuteHand._rotation = 360 * hourPercent;
  secondHand._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. For the `hourHand`, we 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 each passing frame. Here's how:

1. Add two keyframes to the *scripts* layer.
2. On frame 2, add the following code: `updateClock();`
3. On frame 3, add the following code: `gotoAndPlay(2);`

Test the movie and 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 13-4. Think of ways to expand on the clock application: Can you convert the main timeline loop (between frames 2 and 3) to a clip event loop? Can

you make the clock more portable by turning it into a Smart Clip? How about dynamically adding minute and hour markings on the `clockFace`?

The Last Quiz

Here's one final version of the multiple-choice quiz we started way back in Chapter 1, *A Gentle Introduction for Non-Programmers*. This updated version of the quiz dynamically generates all of the quiz's questions and answers using movie clips, so our quiz is infinitely scalable and highly configurable. In fact, we're not far off from making the entire quiz a Smart Clip that could be customized by non-programmers.

The code for the quiz is shown in Example 13-5 and available from the online Code Depot. Because the quiz is now completely dynamically generated, 99% of the code fits entirely on one frame; we no longer need to fill a timeline with questions. (All we're missing is a preloader to ensure smooth playback over a network.) Note that we've used `#include` to import a block of code from an external text file. For more information on `#include`, see Part III, and see "Externalizing ActionScript Code" in Chapter 16. As an exercise, try adding new questions to the quiz by creating new objects and placing them in the questions array.

Though the code for the final quiz is relatively short, it's packed full of important techniques. With the exception of `#include`, we've seen all of them in isolation before, but this extended real-world example shows how they can all fit together. Study the comments carefully—when you understand this version of the quiz in its entirety you'll be well-equipped to create advanced applications with ActionScript.

A longer explanation of the code in this quiz is available at:

http://www.moock.org/webdesign/lectures/ff2001sfWorkshop

Example 13-5. The Multiple-Choice Quiz, One Last Time

```
// CODE ON FRAME 1 OF THE MAIN TIMELINE
//  Stop the movie
stop();

//  Init main timeline variables
var displayTotal;               // Text field for user's final score
var totalCorrect = 0;           // Number of questions answered correctly
var userAnswers = new Array();  // Array containing the user's guesses
var currentQuestion = 0;        // Number of the question the user is on

// Import the source file containing our array of question objects
#include "questionsArray.as"    // See explanation later in this example

// Begin the quiz
makeQuestion(currentQuestion);
```

Example 13-5. The Multiple-Choice Quiz, One Last Time (continued)

```
// The Question() constructor
function Question (correctAnswer, questionText, answers) {
  this.correctAnswer = correctAnswer;
  this.questionText = questionText;
  this.answers = answers;
}

// Function to render each question to the screen
function makeQuestion (currentQuestion) {
  // Clear the Stage of the last question
  questionClip.removeMovieClip();

  // Create and place the main question clip
  attachMovie("questionTemplate", "questionClip", 0);
  questionClip._x = 277;
  questionClip._y = 205;
  questionClip.qNum = "question\n  " + (currentQuestion + 1);
  questionClip.qText = questionsArray[currentQuestion].questionText;

  // Create the individual answer clips in the question clip
  for (var i = 0; i < questionsArray[currentQuestion].answers.length; i++) {
    // Attach our linked answerTemplate clip from the Library;
    // It contains a generalized button and a text field for the question
    questionClip.attachMovie("answerTemplate", "answer" + i, i);
    // Place this answer clip in line below the question
    questionClip["answer" + i]._y += 70 + (i * 15);
    questionClip["answer" + i]._x -= 100;
    // Set the text field in the answer clip to the appropriate element of this
    // question's answer array
    questionClip["answer" + i].answerText =
        questionsArray[currentQuestion].answers[i];
  }
}

// Function to register the user's answers
function answer (choice) {
  userAnswers.push(choice);
  if (currentQuestion + 1 == questionsArray.length) {
    questionClip.removeMovieClip();
    gotoAndStop ("quizEnd");
  } else {
    makeQuestion(++currentQuestion);
  }
}

// Function to tally the user's score
function gradeUser() {
  // Count how many questions the user answered correctly
  for (var i = 0; i < questionsArray.length; i++) {
    if (userAnswers[i] == questionsArray[i].correctAnswer) {
      totalCorrect++;
    }
  }
```

Example 13-5. The Multiple-Choice Quiz, One Last Time (continued)

```
  // Show the user's score in an onscreen text field
  displayTotal = totalCorrect + "/" + questionsArray.length;
}

// CODE ON THE DYNAMICALLY GENERATED ANSWER BUTTONS
// Answer clips are generated dynamically and named in the series
// "answer0", "answer1",..."answern". Each answer clip contains a
// button that, when clicked, checks the name of the answer clip it's
// in to determine the user's choice.
on (release) {
  // Trim the prefix "answer" off this clip's name
  choice = _name.slice(6, _name.length);
  _root.answer(choice);
}

// CODE ON THE quizEnd FRAME
gradeUser();
```

The contents of the *questionsArray.as* file are as shown here:

```
// CODE IN THE questionsarray.as FILE
// ------------------------------------------------
// Contains an array of question objects that
// populate the questions and answers of a multiple-
// choice quiz. Compose new question objects according
// to the following example.

/************** EXAMPLE QUESTION OBJECT ***************
  // Invoke the Question constructor with three arguments:
  //   a zero-relative number giving the correct answer,
  //   a string giving the question text, and
  //   an array containing the multiple-choice answers
  new Question
  (
    1,
    "question goes here?",
    ["answer 1", "answer 2", "answer 3"]
  )
*****************************************************/
// Remember to place a comma after each object in the array except the last
questionsArray = [new Question (2,
  "Which version of Flash first introduced movie clips?",
  ["version 1", "version 2", "version 3",
  "version 4", "version 5", "version 6"]),

  new Question (2,
    "When was ActionScript formally declared a scripting language?",
    ["version 3", "version 4", "version 5"]),

  new Question (1,
    "Are regular expressions supported by Flash 5 ActionScript?",
    ["yes", "no"]),
```

```
new Question (0,
  "Which sound format offers the best compression?",
  ["mp3","aiff", "wav"]),

new Question (1,
  "True or False: The post-increment operator (++) returns the
  value of its operand + 1.",
  ["true", "false"]),

new Question (3,
  "Actionscript is based on...",
  ["Java", "JavaScript", "C++", "ECMA-262", "Perl"])];
```

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: the next chapter teaches "lexical structure" (the finicky details of ActionScript syntax). In the chapter following that, we'll consider a variety of advanced topics. Finally, it's on to Part II, *Applied ActionScript* and Part III, *Language Reference*.