10

Events and Event Handlers

We've learned a lot about composing instructions for the ActionScript interpreter to execute. By now we're pretty comfortable telling the interpreter *what* we want it to do, but how do we tell it *when* to perform those actions? ActionScript code doesn't just execute of its own accord—something always provokes its execution.

That "something" is either the synchronous playback of a movie or the occurrence of a predefined asynchronous *event*.

Synchronous Code Execution

As a movie plays, the timeline's playhead travels from frame to frame. Each time the playhead enters a new frame, the interpreter executes any code attached to that frame. After the code on a frame has been executed, the screen display is updated and sounds are played. Then, the playhead proceeds to the next frame.

For example, when we place code directly on frame 1 of a movie, that code executes before the content in frame 1 is displayed. If we place another block of code on a keyframe at frame 5 of the same movie, frames 1 through 4 will be displayed, then the code on frame 5 will be executed, then frame 5 will be displayed. The code executed on frames 1 and 5 is said to be executed *synchronously* because it happens in a linear, predictable fashion.

All code attached to the frames of a movie is executed synchronously. Even if some frames are played out of order due to a *gotoAndPlay()* or *gotoAndStop()* command, the code on each frame is executed in a predictable sequence, synchronized with the movement of the playhead.

,ch10.16364 Page 195 Monday, April 16, 2001 1:52 PM

Event-Based Asynchronous Code Execution

Some code does not execute in a predictable sequence. Instead, it executes when the ActionScript interpreter notices that one of a predetermined set of *events* has occurred. Many *events* involve some action by the user, such as clicking the mouse or pressing a key. Just as the playhead entering a new frame executes synchronous code attached to the frame, events can cause *event-based* code to execute. Event-based code (code that executes in response to an event) is said to be executed *asynchronously* because the triggering of events can occur at arbitrary times.

Synchronous programming requires us to dictate, in advance, the timing of our code's execution. Asynchronous programming, on the other hand, gives us the ability to react dynamically to events as they occur. Asynchronous code execution is critical to ActionScript and to interactivity itself.

This chapter explores asynchronous (event-based) programming in Flash and catalogs the different events supported by ActionScript.

Types of Events

Conceptually, events can be grouped into two categories:

user events
: Actions taken by the user (e.g., a mouseclick or a keystroke)

system events
: Things that happen as part of the internal playback of a movie (e.g., a movie clip appearing on stage or a series of variables loading from an external file)

ActionScript does not distinguish syntactically between user events and system events. An event triggered internally by a movie is no less palpable than a user's mouseclick. While we might not normally think of, say, a movie clip's removal from the Stage as a noteworthy "event," being able to react to system events gives us great control over a movie.

ActionScript events may also be categorized more practically according to the object to which they pertain. All events happen relative to some object in the Flash environment. That is, the interpreter doesn't just say "The user clicked"; it says, "The user clicked *this button*" or "The user clicked while *this movie clip* was on stage." And the interpreter doesn't say, "Data was received"; it says, "*This movie clip* received some data." We define the code that responds to events on the objects to which the events relate.

The ActionScript objects that can receive events are:

,ch10.16364 Page 196 Monday, April 16, 2001 1:52 PM

- Movie Clips
- Buttons
- Objects of the *XML* and *XMLSocket* classes

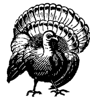
As we'll see throughout this chapter, ActionScript actually has two different event implementations: one for events that relate to movie clips and buttons, and one for all other kinds of objects.

Event Handlers

Not every event triggers the execution of code. Events regularly occur without affecting a movie. A user may, for example, generate dozens of events by clicking repeatedly on a button, but those clicks may be ignored. Why? Because, on their own, events can't cause code to execute—we must write code to react to events explicitly. To instruct the interpreter to execute some code in response to an event, we add a so-called *event handler* that describes the action to take when the specified event occurs. Event handlers are so named because they *catch*, or *handle*, the events in a movie.

An event handler is akin to a specially named function that is automatically invoked when a particular event occurs. Creating an event handler is, hence, very much like creating a function, with a few twists:

- Event handlers have predetermined names such as *keyDown*. You can't name an event handler whatever you like; you have to use the predefined names shown later in Table 10-1 and Table 10-2.
- Event handlers are not declared with the *function* statement.
- Event handlers must be attached to buttons, movie clips, or objects, not frames.



Most events were first introduced in Flash 5. If exporting to Flash 4 format, use only the button event handlers (only button events were supported in Flash 4), and test your work carefully in the Flash 4 Player.

Event Handler Syntax

The names of events (and their corresponding event handlers) are predetermined by ActionScript. Button event handlers are defined using *on* (`eventName`), and movie clip event handlers are defined using *onClipEvent* (`eventName`), where `eventName` is the name of the event to be handled.

,ch10.16364 Page 197 Monday, April 16, 2001 1:52 PM

Hence, all button event handlers (except *keyPress*, which also requires a *key* parameter) take the form:

```
on (eventName) {
  statements
}
```

A single button handler can respond to multiple events, separated by commas. For example:

```
on (rollover, rollOut) {
  // Invoke a custom function in response to both the rollOver and rollOut events
  playRandomSound( );
}
```

All movie clip event handlers take the form:

```
onClipEvent (eventName) {
  statements
}
```

Unlike button handlers, clip handlers can respond only to a single event.

Creating Event Handlers

To create an event handler, we define the handler and attach it to the appropriate object. We'll begin with the most common handlers—those attached to buttons and movie clips.

Attaching Event Handlers to Buttons and Movie Clips

To attach an event handler to a button or a movie clip, we must physically place the code of the handler function onto the desired button or clip. We may do so only in the Flash authoring tool, by selecting the object on stage and entering the appropriate code in the Actions panel, shown in Figure 10-1.

Let's try making a simple event handler function for both a button and a movie clip. To create a button event handler, follow these instructions:

1. Start a new Flash movie.
2. Create a button and drag an instance of it onto the main Stage.
3. With the button selected, type the following code in the Actions panel:

```
on (release) {
  trace("You clicked the button");
}
```

4. Select Control → Test Movie.
5. Click the button. The message, "You clicked the button," appears in the Output window.

,ch10.16364 Page 198 Monday, April 16, 2001 1:52 PM

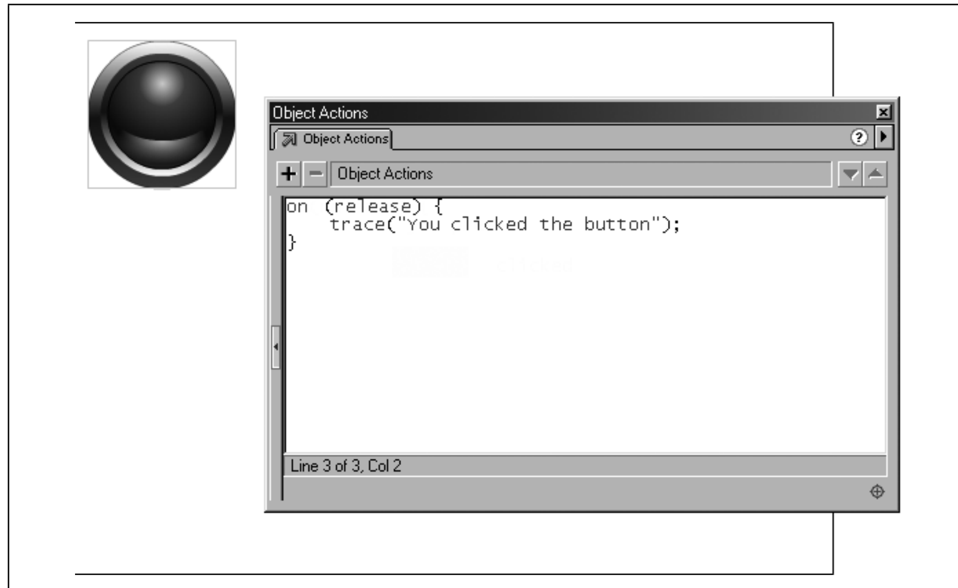


Figure 10-1. Attaching an event handler to a button

When the movie plays and we press and release the button, the *release* event is detected by the interpreter and it executes the *on (release)* event handler. Each time that we press and release the button, the message, "You clicked the button," appears in the Output window.

Now let's try making a slightly more interesting event handler on a movie clip. Once again, follow the instructions:

1. Start a new Flash movie.
2. On the main movie Stage, draw a rectangle.
3. Select Insert → Convert to Symbol.
4. In the Symbol Properties dialog box, name the new symbol `rectangle` and select Movie Clip as the Behavior.
5. Click OK to finish creating the `rectangle` movie clip.
6. Select the `rectangle` clip on stage, and then type the following in the Actions panel:

```
onClipEvent (keyDown) {
  _visible = 0;
}

onClipEvent (keyUp) {
  _visible = 1;
}
```

,ch10.16364 Page 199 Monday, April 16, 2001 1:52 PM

7. Select Control → Test Movie.
8. Click the movie to make sure it has keyboard focus, then press and hold any key. Each time you depress a key, the `rectangle` movie clip disappears. Each time you release the depressed key, `rectangle` reappears.

Notice that we don't manually issue any handler-invocation statements—the interpreter automatically invokes our event handler when the corresponding event occurs.

Flash doesn't support attaching and removing handlers via ActionScript while the movie is playing. Event handlers must be assigned to buttons and movie clips using the Flash authoring tool. The following imaginary syntax, therefore, is not legal:

```
myClip.onKeyDown = function () { _visible = 0; };
```

We'll see how to work around this shortcoming later under "Dynamic Movie Clip Event Handlers."

Attaching Event Handlers to Other Objects

In addition to movie clips and buttons, two built-in object classes—*XML* and *XMLSocket*—support event handlers. For these objects, event handlers are not added to some physical entity in the authoring tool. Rather, they are attached as methods to object instances.

For the *XML* and *XMLsocket* objects, ActionScript uses predefined properties to hold the name of the event handlers. For example, the `onLoad` property holds the name of the handler to be executed when external XML data has loaded.

To set the `onLoad` property for an *XML* object, we use the following code:

```
myDoc = new XML();
myDoc.onLoad = function () { trace("all done loading!"); };
```

Alternatively, we can define the handler function first, and then assign it to the `onLoad` property of our object:

```
function doneMsg () {
  trace("all done loading!");
}
myDoc.onLoad = doneMsg;
```

This syntax closely resembles that of JavaScript, where functions may be assigned to event handler properties, as shown in Example 10-1.

Example 10-1. Assigning a JavaScript Event Handler

```
// Assign a function literal to the onload handler in JavaScript
window.onload = function () { alert("done loading"); };

// Or, alternatively, create and then assign a function to the onload property
function doneMsg () {
  alert("done loading");
```

Example 10-1. Assigning a JavaScript Event Handler (continued)

```
}
window.onload = doneMsg;
```

In the future, more ActionScript objects may support assigning event handlers using object properties, so it's a good idea to get used to this style now. If you're not using the *XML* or *XMLSocket* objects, you can still practice making handlers in this way with HTML documents and JavaScript. The beauty of this approach is its flexibility; any event handler function may be easily reassigned or even removed during movie playback.

We'll learn more about attaching functions to objects in Chapter 12, *Objects and Classes*. Information about the events supported by the *XML* and *XMLSocket* objects may be found in Part III, *Language Reference*.

The lifespan of event handlers is tied to the life of the objects with which they are associated. When a clip or button is removed from the Stage or when an *XML* or *XMLSocket* object dies, any event handlers associated with those objects die with them. An object must be present on stage or exist on the timeline for its handlers to remain active.

Event Handler Scope

As with any function, the statements in an event handler execute within a predefined scope. Scope dictates where the interpreter looks to resolve the variables, subfunctions, objects, or properties referenced in an event handler's body. We'll consider event handler scope in relation to movie clip events, button events, and other object events.

Movie Clip Event Handler Scope

Unlike regular functions, movie clip event handlers *do not define a local scope*! When we attach a handler to a clip, the scope of the handler is the clip, not just the event handler itself. This means that all variables are retrieved from the clip's timeline. For example, if we attach an *enterFrame* event handler to a clip named `navigation` and write *trace(x);* inside the handler, the interpreter looks for the value of `x` on `navigation`'s timeline:

```
onClipEvent (enterFrame) {
  trace(x);  // Displays the value of navigation.x
}
```

The interpreter does not consult a local scope first because there is no local scope to consult. If we write *var y = 10;* in our handler, `y` is defined on `navigation`'s timeline, even though the *var* keyword ordinarily declares a local variable when used in a function.

The easiest way to remember the scope rules of a clip event handler is to treat the handler's statements as though they were attached to a frame of the handler's clip. For example, suppose we have a clip named `ball` that has a variable called `xVelocity` in it. To access `xVelocity` from inside a `ball` event handler, we simply refer to it directly, like this:

```
onClipEvent (mouseDown) {
  xVelocity += 10;
}
```

We don't have to supply the path to the variable as `_root.ball.xVelocity` because the interpreter already assumes we mean the variable `xVelocity` in `ball`. The same is true of properties and methods; instead of using `ball._x`, we simply use `_x`, and instead of using *ball.gotoAndStop(5)*, we simply use *gotoAndStop(5)*. For example:

```
onClipEvent (enterFrame) {
  _x += xVelocity;                 // Move the ball
  gotoAndPlay(_currentframe - 1);  // Do a little loop
}
```

We can even define a function on `ball` using a function declaration statement in a handler, like this:

```
onClipEvent (load) {
  function hideMe() {
    _visibility = 0;
  }
}
```

It's sometimes easy to forget that statements in clip event handlers are scoped to the *clip's timeline*, not the handler function's local scope and not the clip's *parent* timeline (the timeline upon which the clip resides).

For example, suppose we place our `ball` clip on the main timeline of a movie, and the main timeline (not `ball`'s timeline) has a *moveBall()* function defined on it. We may absent-mindedly call *moveBall()* from an event handler on `ball` like this:

```
onClipEvent (enterFrame) {
  moveBall();  // Does nothing! There's no moveBall() function in ball.
               // The moveBall() function is defined on _root
}
```

We have to explicitly refer to the *moveBall()* function on the main timeline using `_root` like this:

,ch10.16364 Page 202 Monday, April 16, 2001 1:52 PM


```
onClipEvent (enterFrame) {
  _root.moveBall();  // Now it works!
}
```

Occasionally, we may need to refer to the current clip object explicitly from within an event handler. We can do so using the `this` keyword, which refers to the current movie clip when used in an event handler. Hence, the following references are synonymous within a clip event handler:

```
this._x  // Same as next line
_x

this.gotoAndStop(12);  // Same as next line
gotoAndStop(12);
```

Use of `this` is most frequently required when we're dynamically generating the name of one of the current clip's properties (either a variable name or a nested clip). Here we tell one of the nested clips in the series `ball.stripe1`, `ball.stripe2`, . . . to start playing, depending on the current frame of the `ball` clip:

```
onClipEvent (enterFrame) {
  this["stripe" + _currentframe].play();
}
```

The keyword `this` is also frequently used with movie clip methods that demand an explicit reference to a movie clip object upon invocation. Any movie clip method with the same name as an ActionScript global function must be used with an explicit clip reference. The `this` keyword is therefore necessary when invoking the following functions as methods inside an event handler:

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

For example:

```
this.duplicateMovieClip("ball2", 1);
this.loadVariables("vars.txt");
this.startDrag(true);
this.unloadMovie();
```

We'll learn all about the dual nature of these functions in "Method versus global function overlap issues," in Chapter 13, *Movie Clips*.

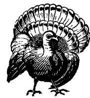
Note that the `this` keyword allows us to refer to the current clip even when that clip has no assigned instance name in the authoring tool or when we don't know

,ch10.16364 Page 203 Monday, April 16, 2001 1:52 PM

the clip's name. In fact, using `this`, we may even pass the current clip as a reference to a function without ever knowing the current clip's name. Here's some quite legal (and quite elegant) code to demonstrate:

```
// CODE ON MAIN TIMELINE
// Here is a generic function that moves any clip
function move (clip, x, y) {
  clip._x += x;
  clip._y += y;
}

// CODE ON CLIP
// Call the main timeline function and tell it to move the
// current clip by passing a reference with the this keyword
onClipEvent (enterFrame) {
  _root.move(this, 10, 15);
}
```



In build 30 of the Flash 5 Player, a bug prevented *gotoAndStop()* and *gotoAndPlay()* from working inside a clip handler when used with string literal labels. Such commands were simply ignored. For example, this would not work:

```
onClipEvent(load) {
  gotoAndStop("intro");  // Won't work in Flash 5 r30
}
```

To work around the bug, use a self-reflexive clip reference, as in:

```
onClipEvent(load) {
  this.gotoAndStop("intro");
}
```

Button Event Handler Scope

Button handlers are scoped to the timeline upon which the button resides. For example, if we place a button on the main timeline and declare the variable `speed` in a handler on that button, `speed` will be scoped to the main timeline (`_root`):

```
// CODE FOR BUTTON HANDLER
on (release) {
  var speed = 10;  // Defines speed on _root
}
```

By contrast, if we place a movie clip, `ball`, on the main timeline and declare the variable `speed` in a handler of `ball`, `speed` is scoped to `ball`:

```
// CODE FOR ball HANDLER
on (load) {
  var speed = 10;  // Defines speed on ball, not _root
}
```

Inside a button handler, the `this` keyword refers to the timeline on which the button resides:

```
on (release) {
  // Make the clip on which this button resides 50% transparent
  this._alpha = 50;
  // Move the clip on which this button resides 10 pixels to the right
  this._x += 10;
}
```

Other Object Event Handler Scope

Unlike movie clip and button handlers, event handlers attached to instances of built-in classes such as *XML* and *XMLSocket* are scoped exactly like functions. An *XML* or *XMLSocket* object's event handler has a scope chain that is defined when the handler function is defined. Furthermore, *XML* and *XMLSocket* event handlers define a local scope. All the rules of function scope described in "Function Scope" in Chapter 9, *Functions*, apply directly to event handler functions attached to objects that are neither buttons nor movie clips.

Button Events

Table 10-1 briefly introduces the various events available for buttons. Using button events, we can easily create code for navigation, forms, games, and other interface elements. Let's explore each button event and learn how a button can be programmed to react to mouse and keyboard events.

Each of the button events in Table 10-1 is handled by a matching button event handler of the form *on* (`eventName`). For example, the *press* event is handled using an event handler beginning with *on (press)*. The exception is the *keyPress* event handler which takes the form *on* (*keyPress* `key`) where `key` is the key to detect. Button events are sent only to the button with which the mouse is interacting. If multiple buttons overlap, the topmost button receives all events; no other buttons can respond, even if the topmost button has no handlers defined. In the following descriptions, the *hit* area refers to the physical region of the button that must be under the mouse pointer in order for the button to be activated. (A button's *hit* area is defined graphically when you create the button in the Flash authoring tool.)

Table 10-1. Button Events

Button Event Name	Button Event Occurs When . . .
`press`	Primary mouse button is depressed while pointer is in the button's *hit* area. Other mouse buttons are not detectable.
`release`	Primary mouse button is depressed and then released while pointer is in the button's *hit* area.

Table 10-1. Button Events (continued)

Button Event Name	Button Event Occurs When . . .
`releaseOutside`	Primary mouse button is depressed while pointer is in the button's *hit* area and then released while pointer is outside of the *hit* area.
`rollOver`	Mouse pointer moves into the button's *hit* area without the mouse button depressed.
`rollOut`	Mouse pointer moves out of the button's *hit* area without the mouse button depressed.
`dragOut`	Primary mouse button is depressed while pointer is in the button's *hit* area, and then, while mouse button is still depressed, pointer is moved out of the *hit* area.
`dragOver`	Primary mouse button is depressed while pointer is in the button's *hit* area, and then, while mouse button is still depressed, pointer is moved out of, then back into, the button's *hit* area.
`keyPress`	Specified *`key`* is depressed. In most cases, the *keyDown* clip event is preferred over the *keyPress* button event.

press

A mouseclick is technically a two-step process: the mouse button is depressed (*press*) and then released (*release*). A *press* event occurs when the mouse pointer is in the *hit* area of a button and the primary mouse button is depressed. Secondary mouse buttons are not detectable. Button *press* events are appropriate for radio buttons or weapons firing in a game, but use *release* events to allow the user to change his mind before releasing the mouse.

release

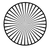
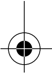
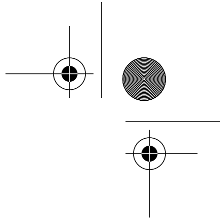
The *release* button event occurs when the following sequence is detected:

1. The mouse pointer is in the *hit* area of a button.
2. The primary mouse button is pressed while the mouse pointer is still in the *hit* area of the button (at which point a *press* event occurs).
3. The primary mouse button is released while the mouse pointer is still in the *hit* area of the original button (at which point the *release* event occurs).

By using the *release* event instead of the *press* event, you give users a chance to move the pointer off of a button even after it has been clicked, thus allowing them to retract their action.

releaseOutside

The *releaseOutside* event typically indicates that the user changed his mind by clicking on a button but moving the pointer off the button before releasing the mouse button. The event is generated when the following sequence is detected:

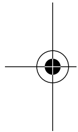
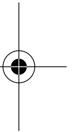
,ch10.16364 Page 206 Monday, April 16, 2001 1:52 PM

1. The mouse pointer is in the *hit* area of a button.
2. The primary mouse button is pressed and held (the *press* event occurs).
3. The mouse pointer moves out of the button's *hit* area (the *dragOut* event occurs).
4. The primary mouse button is released while not in the *hit* area of the original button.

You will rarely bother detecting *releaseOutside* events, as they usually indicate that the user intended not to perform any action.

rollOver

The *rollOver* event occurs when the mouse pointer moves into the *hit* area of a button with no mouse buttons depressed. The *rollOver* event is rarely used in ActionScript because visual button changes are created directly in the authoring tool, not with scripting. You should use the provided *up*, *over*, and *down* frames in the authoring tool to create highlight states for buttons.

The *rollOver* event in Flash 5 provides a handy means of retrieving a text field selection. For more details, see "Selection Object" in Part III.

rollOut

The *rollOut* event is *rollOver*'s counterpart; it occurs when the mouse pointer is moved out of the *hit* area of a button with no mouse buttons depressed. As with *rollOver*, *rollOut* is rarely used because button highlight states are created directly in the authoring tool, so manual image swapping is not required in ActionScript.

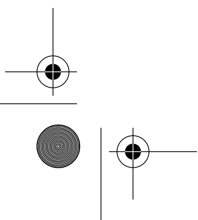
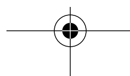
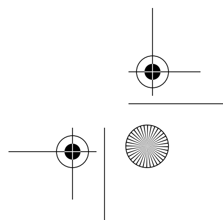
dragOut

The *dragOut* event is similar to *rollOut*, except that it is generated if the mouse button is down when the pointer leaves a button's *hit* area. The *dragOut* event is followed by either the *releaseOutside* event (if the user releases the mouse button) or the *dragOver* event (if the user moves the pointer back into the button's *hit* area without having released the mouse button).

dragOver

The *dragOver* event is a seldom-seen woodland creature. It is conjured up when the following sequence is performed:

1. The mouse pointer moves into the *hit* area of a button (*rollOver* event occurs).
2. The primary mouse button is pressed and held (*press* event occurs).

,ch10.16364 Page 207 Monday, April 16, 2001 1:52 PM

3. The mouse pointer moves out of the button's *hit* area (*dragOut* event occurs).
4. The mouse pointer moves back into the button's *hit* area (*dragOver* event occurs).

Thus, the *dragOver* event indicates that the user has moved the mouse pointer out of and back into the *hit* area, all the while holding the mouse button down. Note that *dragover*, instead of the *rollOver* event, is generated if the mouse button is still down when the pointer reenters the button's *hit* area.

keyPress

The *keyPress* event is unrelated to mouse events and is instead triggered by the pressing of a specified key. We cover it here because it uses the *on* (`eventName`) syntax of other ActionScript button event handlers. This event handler requires us to specify the key that triggers the event:

```
on (keyPress key) {
  statements
}
```

where `key` is a string representing the key associated with the event. The string may be either the character on the key (such as "s" or "S"), or a keyword representing the key in the format `"<Keyword>"`. Only one key may be specified with each handler. To capture multiple keys using *keyPress,* we must create multiple *keyPress* event handlers. For example:

```
// Detects the "a" key
on (keyPress "a") {
  trace("The 'a' key was pressed");
}

// Detects the Enter key
on (keyPress "<Enter>") {
  trace("The Enter key was pressed");
}

// Detects the Down Arrow key
on (keyPress "<Down>") {
  trace("The Down Arrow key was pressed");
}
```

The legal values of `Keyword` are as follows (note that the function keys F1 . . . F12 are not supported by *keyPress*, but are detectable using the *Key* object):

```
<Backspace>
<Delete>
<Down>
<End>
<Enter>
<Home>
```

,ch10.16364 Page 208 Monday, April 16, 2001 1:52 PM


```
<Insert>
<Left>
<PgDn>
<PgUp>
<Right>
<Space>
<Tab>
<Up>
```

In Flash 4, *keyPress* was the only means we had of interacting with the keyboard. In Flash 5 and later, the *Key* object, in combination with the movie clip events *keyDown* and *keyUp* (discussed later), offer much greater control over keyboard interaction. The *keyPress* event detects the pressing of a single key at a time, whereas the *Key* object can detect the simultaneous pressing of multiple keys.

Movie Clip Events Overview

Movie clip events are generated by a wide variety of occurrences in the Flash Player, from mouseclicks to the downloading of data. Clip events can be broken into two categories: user-input events and movie-playback events. User-input events are related to the mouse and keyboard, while movie-playback events are related to the rendering of frames in the Flash Player, the birth and death of movie clips, and the loading of data.

Note that user-input clip events partially overlap the functionality of the button events described earlier. For example, a clip's *mouseDown* event handler can detect a mouse press just as a button's *press* event handler can. Movie clip events, however, are not tied to any kind of *hit* area like button events are and do not affect the look of the mouse pointer.

Let's spend some quality time with the ActionScript movie clip events, summarized in Table 10-2. We'll look at the movie-playback events first (*enterFrame*, *load*, *unload*, and *data*) and then see how the user-input events work (*mouseDown*, *mouseUp*, *mouseMove*, *keyDown*, *keyUp*). Each of the clip events is handled by a matching clip event handler of the form *onClipEvent* (`eventName`). For example, the *enterFrame* event is handled using an event handler beginning with *onClipEvent (enterFrame)*. With the exception of *load*, *unload*, and *data*, movie clip events are sent to *all* movie clips on stage even if, say, the user clicks the mouse while on top of a different movie clip (or no movie clip).

Table 10-2. Movie Clip Events

Clip Event Name	Clip Event Occurs When . . .
enterFrame	Playhead enters a frame (before frame is rendered in the Flash Player)
load	The clip first appears on the Stage
unload	The clip is removed from the Stage

,ch10.16364 Page 209 Monday, April 16, 2001 1:52 PM

Table 10-2. Movie Clip Events (continued)

Clip Event Name	Clip Event Occurs When . . .
data	Variables finish loading into a clip or a portion of a loaded movie loads into a clip
mouseDown	Primary mouse button is depressed while the clip is on stage (secondary mouse buttons are not detectable)
mouseUp	Primary mouse button is released while the clip is on stage
mouseMove	Mouse pointer moves (even a teensy bit) while the clip is on Stage, even if the mouse is not over the clip
keyDown	A key is pressed down while the clip is on Stage
keyUp	A depressed key is released while the clip is on Stage

Movie-Playback Movie Clip Events

The following events are generated without user intervention as Flash loads and plays movies.

enterFrame

If you've ever resorted to empty, looping movie clips to trigger scripts, *enterFrame* offers a welcome respite. The *enterFrame* event occurs once for every frame that passes in a movie. For example, if we place the following code on a movie clip, that clip will grow incrementally by 10 pixels per frame:

```
onClipEvent (enterFrame) {
  _height += 10;
  _width += 10;
}
```

(Notice that, as we learned earlier, the `_height` and `_width` properties are resolved within the scope of the clip to which the *enterFrame* event handler is attached, so no clip instance name is required before `_height` and `_width`.)

The *enterFrame* event is generated before each frame is rendered even if the playhead of the clip with the *enterFrame* handler is stopped. The *enterFrame* event, hence, is always being triggered.

When displayed in the Flash Player, all Flash movies are constantly running, even when nothing is moving on screen or when a movie's playhead is stopped on a frame. An individual movie clip's *enterFrame* handler will, hence, be executed repeatedly for as long as that clip is on stage, regardless of whether the clip is playing or stopped. If a clip's playhead is moved by a *gotoAndStop()* function call, the clip's *enterFrame* event handler is still triggered with each passing frame. And

,ch10.16364 Page 210 Monday, April 16, 2001 1:52 PM

if every playhead of an entire movie has been halted with a *stop()* function, all *enterFrame* event handlers on all clips will still execute.

The *enterFrame* event is normally used to update the state of a movie clip repeatedly over time. But an *enterFrame* event handler need not apply directly to the clip that bears it—*enterFrame* can be used with a single-frame, empty clip to execute code repeatedly. This technique, called a *clip event loop* (or more loosely, a *process*) is demonstrated in "Timeline and Clip Event Loops" in Chapter 8, *Loop Statements*.

Note that the code in an *enterFrame* event handler is executed *before* any code that appears on the timeline of the clip containing the handler.

With a little ambition, we can use *enterFrame* to gain extremely powerful control over a clip. Example 10-7, shown later, extends our earlier clip-enlarging code to make a movie clip oscillate in size.

load

The *load* event occurs when a movie clip is born—that is, when a movie clip appears on stage for the first time. A movie clip "appears on stage" in one of the following ways:

- The playhead moves onto a keyframe that contains a new instantiation of the clip, placed in the authoring tool.
- The clip is duplicated from another clip via the *duplicateMovieClip()* function.
- The clip is programmatically added to the Stage via the *attachMovie()* function.
- An external *.swf* file is loaded into the clip with the *loadMovie()* function.
- The contents of a clip are unloaded with the *unloadMovie()* function. (A *load* event is triggered because an empty placeholder clip is loaded into the clip when its contents are expelled.)

The body of a *load* event handler is executed *after* any code on the timeline where the movie clip first appears.

A *load* event handler is often used to initialize variables in a clip or to perform some setup task (like sizing or positioning a dynamically generated clip). A *load* handler can also provide a nice way to prevent a movie clip from automatically playing:

```
onClipEvent (load) {
  stop();
}
```

The *load* event handler might also be used to trigger some function that relies on the existence of a particular clip in order to execute properly.

,ch10.16364 Page 211 Monday, April 16, 2001 1:52 PM

The *load* event is particularly interesting when combined with the *duplicateMovieClip()* function, which creates new movie clips. In Example 10-2 we generate an entire field of `star` clips using a single *load* event handler in a cascading chain. The load handler is copied to each duplicated `star`, causing it, in turn, to duplicate itself. The process stops when the 100th clip is duplicated. The *.fla* file for Example 10-2 is available from the online Code Depot.

Example 10-2. Generating a Star Field with a load Event

```
onClipEvent (load) {
  // Place the current clip at a random position
  _x = Math.floor(Math.random() * 550);
  _y = Math.floor(Math.random() * 400);

  // Reset clip scale so we don't inherit previous clip's scale
  _xscale = 100;
  _yscale = 100;

  // Randomly size current clip between 50 and 150 percent
  randScale = Math.floor(Math.random() * 100) - 50;
  _xscale += randScale;
  _yscale += randScale;

  // If we're not at the 100th star, make another one
  if (_name != "star100") {
    nextStarNumber = number(_name.substring(4, _name.length)) + 1;
    this.duplicateMovieClip("star" + nextStarNumber, nextStarNumber);
  }
}
```

unload

The *unload* event is the opposite of the *load* event: it occurs when a movie clip expires—that is, immediately *after* the last frame in which the clip is present on stage (but *before* the first frame in which the clip is absent).

The following incidents provoke a movie clip's *unload* event:

- The playhead reaches the end of the span of frames upon which the clip resides.
- The clip is removed via the *removeMovieClip()* function (which kills clips generated by the *attachMovie()* and *duplicateMovieClip()* functions).
- A previously loaded external *.swf* file is removed from the clip via the *unloadMovie()* function.
- The clip has an external *.swf* loaded into it.

This last *unload* event trigger may seem a little odd but is actually a natural result of the way movies are loaded into Flash. Anytime a *.swf* is loaded into a movie clip, the previous contents of that clip are displaced, causing an *unload* event.

,ch10.16364 Page 212 Monday, April 16, 2001 1:52 PM

Here's an example that illustrates the behavior of the *load* and *unload* events in connection with *loadMovie()*:

1. In the Flash authoring tool, we place an empty movie clip on stage at frame 1 of a movie's main timeline. We name our clip `emptyClip`.
2. At frame 5 of the main timeline, we load the movie *test.swf* into `emptyClip` using the following code: `emptyClip.loadMovie("test.swf");`
3. We play the movie using Control → Play movie.

The results are:

1. Frame 1: The `emptyClip` clip appears, causing a *load* event.
2. Frame 5: The *loadMovie()* function is executed in two stages:
   a. The placeholder content of `emptyClip` is removed to make room for the incoming *test.swf*, causing an *unload* event.
   b. The movie *test.swf* loads, causing a *load* event.

The *unload* event is typically used to initiate housecleaning code—code that cleans up the Stage or resets the program environment in some way. An *unload* handler also provides a means for performing some action (such as playing another movie) after a movie clip ends.

data

The *data* event occurs when external data is loaded into a movie clip. The *data* event can be triggered by two quite different circumstances, according to the kind of data being loaded. We'll consider those circumstances separately.

Using a data event handler with loadVariables()

When we request a series of variables from a server using *loadVariables()*, we must wait for them to load completely before using their information. (See Part III.)

When a movie clip receives the end of a batch of loaded variables, the *data* event is triggered, telling us it's safe to execute code that relies on the variables.

For example, suppose we have a guest book movie in which visitors enter comments and we store those comments on a server. When a user attempts to view a comment, we request it from the server using *loadVariables()*. But before we can display the comment, we must pause at a loading screen until we know that the requested data is available. A *data* event handler tells us when our data has loaded, at which point we can safely display the comment to the user.

Example 10-3 is a simplified excerpt of some code from a guest book showing a data event handler used with *loadVariables()*. In the example, a button loads two

,ch10.16364 Page 213 Monday, April 16, 2001 1:52 PM

URL-encoded variables from a text file into a movie clip. The movie clip bears a *data* event handler that executes when the variables have loaded. From inside that handler, we display the values of the variables. We know the variables are safe to display because the code in the handler isn't executed until triggered by the *data* event (i.e., after the data is received).

Example 10-3. Waiting for a data Event

```
// CONTENT OF OUR guestbook.txt FILE
name=judith&message=hello

// BUTTON INSIDE OUR CLIP
on (release) {
  this.loadVariables("guestbook.txt");
}

// HANDLER ON OUR CLIP
onClipEvent (data) {
  trace(name);
  trace(message);
}
```

We'll use the *data* event again when we build a Flash form in Chapter 17, *Flash Forms*.

Using a data event handler with loadMovie()

The second use of the *data* event relates to the loading of external *.swf* files into movie clips with the *loadMovie()* function. When a *.swf* file is loaded into a host clip, by default the file begins playing immediately, even if only partially loaded. This is not always desirable—sometimes we want to guarantee that all or a certain percentage of a *.swf* has loaded before playback begins. We can make that guarantee with a *data* event handler and some preloading code.

The *data* event occurs each time a host movie clip receives a portion of an external *.swf* file. The definition of what constitutes a "portion" is more complex than you might expect. In order for a *data* event to be triggered, at least one complete new frame of the external *.swf* file must have loaded since either: (a) the last *data* event fired or (b) the *.swf* file started loading. (More than one frame of the *.swf* file may actually have loaded in that amount of time, but one frame is the minimum number required to prompt a *data* event.)

The execution of *data* event handlers is tied to the rendering of frames in the Player. With every frame rendered, the interpreter checks to see if part of an external *.swf* file has been loaded into a clip that has a *data* event handler. If part of an external *.swf* file has been loaded into such a clip, and the loaded portion contains at least one new frame, then the *data* event handler is executed. This process happens once—and only once—per frame rendered (even if the playhead is stopped).

,ch10.16364 Page 214 Monday, April 16, 2001 1:52 PM

Note that because the *data* event happens on a per-frame basis, movies with higher frame rates tend to have smoother-looking preloaders because they receive more frequent updates on the status of loading *.swf* files.

The exact number of *data* events triggered during a *loadMovie()* operation depends on the distribution of content in the *.swf* file being loaded and the speed of the connection. A single-frame *.swf* file, no matter how large, will trigger only one *data* event. On the other hand, a *.swf* file with 100 frames may trigger up to 100 separate *data* events, depending on the movie's frame rate, the byte size of each frame and the speed of the network connection. If the frames are large and the connection is slow, more *data* events will be triggered (up to a maximum of one per frame). If the frames are small and the connection is fast, fewer *data* events will be triggered (the entire 100 frames may be transferred between the rendering of two frames in the Player, prompting only one *data* event).

So how do we use a *data* event handler to build a preloader? Well, whenever a *data* event occurs due to a *loadMovie()* function call, we know that an external *.swf* file download is in progress. Therefore, from inside a *data* event handler, we can check whether enough of the file has downloaded before allowing it to play. We do so using the *getBytesLoaded()* and *getBytesTotal()* functions as shown in Example 10-4. (The `_framesloaded` and `_totalframes` movie clip properties may also be used.)

Example 10-4 also provides feedback while the movie is loading. Note that the *.swf* file being loaded should have a *stop()* function call on its first frame to prevent it from automatically playing before it is completely downloaded. A variation of Example 10-4 is available from the online Code Depot.

Example 10-4. A data Event Preloader

```
onClipEvent (data) {
    trace("data received");          // The show's about to start!

    // Turn on data-transfer light
    _root.transferIndicator.gotoAndStop("on");

    // If we're done loading, turn off transfer light, and let the movie play
    if (getBytesTotal() > 0 && getBytesLoaded() == getBytesTotal()) {
      _root.transferIndicator.gotoAndStop("off");
      play();
    }

    // Display some loading details in text field variables on the _root
    _root.bytesLoaded = getBytesLoaded();
    _root.bytesTotal = getBytesTotal();
    _root.clipURL = _url.substring(_url.lastIndexOf("/") + 1, _url.length);
}
```

The User-Input Movie Clip Events

The remainder of the movie clip events relate to user interaction. When any of the user-input clip events occurs, *all* clips on stage (no matter how deeply nested in other clips) receive the event. Hence, multiple clips may react to a single mouse-click, mouse movement, or keystroke.

To execute code based on the proximity of the mouse to a particular clip, an event handler should check the location of the mouse pointer relative to the clip. The built-in *hitTest()* function provides an easy way to check whether a mouse-click occurred within a certain region, as shown later in Example 10-9.

mouseDown

Like the *press* button event, the *mouseDown* clip event detects the downstroke of a mouseclick. The *mouseDown* event occurs each time the primary mouse button is depressed while the mouse pointer is over *any part* of the Stage.

Unlike the button *press* event, *mouseDown* is not tied to the *hit* area of a button. In combination with the *mouseUp* and *mouseMove* events and the *Mouse.hide()* method, the *mouseDown* event can be used to implement a custom mouse pointer, as we'll see later in Example 10-8.

mouseUp

The *mouseUp* event is the counterpart to *mouseDown*. It occurs each time the primary mouse button is released while the mouse pointer is over any part of the Stage. As with *mouseDown*, a clip with a *mouseUp* handler must be present on stage at the time the mouse button is released in order for the event to have any consequence. The *mouseUp, mouseDown,* and *mouseMove* events can be used to create rich levels of mouse interactivity without affecting the appearance of the mouse pointer (as a button does).

mouseMove

The *mouseMove* event lets us detect changes in the mouse pointer's position. Whenever the mouse is in motion, *mouseMove* events are issued repeatedly, as fast as the processor can generate new events. A clip with a *mouseMove* handler must be present on stage at the time the mouse is moving in order for the *mouseMove* event to have any effect.

The *mouseMove* event is useful for code that wakes up idle applications, displays mouse trails, and creates custom pointers, as we'll see later in Example 10-8.

keyDown

The *keyDown* and *keyUp* events are the keyboard analogs of *mouseDown* and *mouseUp*. Together, they provide fundamental tools for coding keyboard-based interactivity. The *keyDown* event occurs whenever a key on the keyboard is depressed. When a key is held down, *keyDown* may occur repeatedly, depending on the operating system and keyboard setup. Unlike the *keyPress* button event, *keyDownclip* events occur when any key—not just a specific key—is pressed.

To *trap* (i.e., detect or *catch*) a *keyDown* event, we must ensure that a movie clip with a *keyDown* event handler is present on stage at the time that a key is pressed. The following code does the trick:

```
onClipEvent (keyDown) {
  trace("Some key was pressed");
}
```

You'll notice that our *keyDown* handler does not tell us which key was pressed. If we're waiting for the user to press any key to continue, we might not care which key it was. But usually, we want to tie some action to a specific key. For example, we might want different keys to turn a spaceship in different directions.

To find out which keys triggered the *keyDown* event, we consult the built-in *Key* object, which describes the keyboard's state. The type of information we require depends on the interactivity we're trying to produce. Games, for example, require instant, continuous feedback from potentially simultaneous keypresses. Navigational interfaces, in contrast, may require only the detection of a single keypress (e.g., the spacebar in a slide show presentation).

The *Key* object can tell us which key was last pressed and whether a particular key is currently being pressed. To determine the state of the keyboard, we use one of the four *Key* object methods:

```
Key.getCode()           // Base-10 keycode value of last key pressed
Key.getAscii()          // Base-10 ASCII value of last key pressed
Key.isDown(keycode)     // Returns true if specified key is currently pressed
Key.isToggled(keycode)  // Determines whether Caps Lock or Num Lock is toggled on
```

Example 10-5 shows a *keyDown* handler that tells us the ASCII value of the last key pressed.

Example 10-5. Checking the Last Key Pressed

```
onClipEvent (keyDown) {
  // Retrieve the ASCII value of the last key pressed and convert it to a character
  lastKeyPressed = String.fromCharCode(Key.getAscii());
  trace("You pressed the '" + lastKeyPressed + "' key.");
}
```

,ch10.16364 Page 217 Monday, April 16, 2001 1:52 PM

Example 10-6 shows a sample *keyDown* handler that checks whether the up arrow was the last key pressed.

Example 10-6. Detecting an Up Arrow Keypress

```
onClipEvent (keyDown) {
  // Check to see if the up arrow was the last key pressed.
  // The up arrow is represented by the Key.UP property.
  if (Key.getCode() == Key.UP) {
    trace("The up arrow was the last key depressed");
  }
}
```

There are several ways to query the state of the keyboard, and you must choose the one that best suits your application. For example, the *Key.getAscii()* method returns the ASCII value of the character associated with the last-pressed key, which may differ across keyboards in different languages (though, in English, the placement of the letters and numbers on a keyboard is standardized). On the other hand, the *Key.getCode()* method returns a value tied to a physical key on the keyboard, not a specific letter. *Key.getCode()* may be more useful for an international or cross-platform audience if you want to, say, use four adjacent keys for navigation regardless of the characters they represent. There's more information on this topic under "Key Object" in Part III.

You can download sample *keyDown* and *keyUp .fla* files from the online Code Depot.

Event handlers that react to keystrokes are executed only if the Flash Player has mouse focus. Users must click the Stage of a movie before the movie's keystroke handlers will become active. Consider forcing users to click a button before entering any keyboard-controlled section of a movie.

Handling special keys

To disable the Flash standalone Player menu commands (Open, Close, Fullscreen, etc.), add the following line of code to the beginning of your movie:

```
fscommand("trapallkeys", "true");
```

That command also prevents the Escape key from exiting fullscreen mode in a Projector. To capture Escape in a Projector, use:

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.ESCAPE) {
    // Respond to Escape keypress
  }
}
```

,ch10.16364 Page 218 Monday, April 16, 2001 1:52 PM

Note that the Escape key cannot be trapped in all browsers. Furthermore, there is no way to disable the Alt key or the Windows Alt-Tab or Ctrl-Alt-Delete key sequences.

To capture Tab keypresses, create a button with the following handler:

```
on (keyPress "<Tab>") {
  // Respond to Tab key
}
```

In the standalone Player, the Tab key may also be captured with a clip event handler such as:

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.TAB) {
    // Respond to Tab keypress
  }
}
```

In some browsers, the Tab key can be detected only with a button *keyPress* event, and it may even be necessary to combine a *keyPress* button event with a *keyUp* clip event. The following code first traps the Tab key with *keyPress*, and then reacts to it in a *keyUp* handler. Note that we don't use *keyDown* because *Key.getCode()* for the Tab key is set only on the key upstroke in Internet Explorer:

```
// CODE ON BUTTON ON MAIN TIMELINE
on (keyPress "<Tab>") {
  // Set a dummy variable here
  foo = 0;
}

// CODE ON MOVIE CLIP ON MAIN TIMELINE
onClipEvent (keyUp) {
  if (Key.getCode() == Key.TAB) {
    // Now place the cursor in myTextField on _level0
    Selection.setFocus("_level0.myTextField");
  }
}
```

We typically trap the Tab key in order to move the insertion point to a particular text field in a form. See the example under "Selection.setFocus( ) Method" in Part III for details.

To capture a shortcut-key-style combination such as Ctrl-F, use an *enterFrame* handler and the *Key.isDown()* method:

```
onClipEvent (enterFrame) {
  if (Key.isDown(Key.CONTROL) && Key.isDown(70)) {
    // Respond to Ctrl-F
  }
}
```

,ch10.16364 Page 219 Monday, April 16, 2001 1:52 PM

To capture the Enter (or Return) key, use either a button handler, such as:

```
on (keyPress "<Enter>") {
  // Respond to Enter key press (e.g., submit a form)
}
```

or a *keyDown* handler, such as:

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.ENTER) {
    // Respond to Enter key press (e.g., submit a form)
  }
}
```

See “Key Object” and “Key.getCode( ) Method” in Part III for more information on capturing other special keys such as the function keys (F1, F2, etc.) or keys on the numeric keypad.

keyUp

The *keyUp* event is triggered when a depressed key is released. The *keyUp* event is an essential component of game programming because it lets us turn off something that was turned on by an earlier *keyDown* event—the classic example being a spaceship’s thrust. As a further example, in the Flash authoring tool, holding down the spacebar temporarily switches to the Hand tool, and releasing the spacebar restores the previous tool. This approach can be used to show and hide things in your application, such as temporary menus.

As with *keyDown*, in order to obtain useful information from a *keyUp* event, we normally use it with the *Key* object:

```
onClipEvent (keyUp) {
  if (!Key.isDown(Key.LEFT)) {
    trace("The left arrow is not depressed");
  }
}
```

Because the *Key.isDown()* method lets us check the status of any key anytime, we may use an *enterFrame* event loop to check whether a certain key is depressed. However, *polling* the keyboard (i.e., checking the status of a key repeatedly) is less efficient than waiting until we *know* that a key has been pressed as indicated by a *keyDown* event triggering our event handler.

The approach we end up taking ultimately depends on the type of system we’re building. In a system that’s constantly in motion, such as a game, polling may be appropriate because we’re cycling through a main game loop with every frame anyway. So, we can just check the *Key* object while we’re doing the rest of our loop. For example:

```
// CODE ON EMPTY CLIP
// This keeps the game process running
```

,ch10.16364 Page 220 Monday, April 16, 2001 1:52 PM


```
onClipEvent (enterFrame) {
  _root.mainLoop();
}

// CORE GAME CODE ON MAIN TIMELINE
// This is executed once per frame
function mainLoop () {
  if (Key.isDown(Key.LEFT)) {
    trace("The left arrow is depressed");
    // Rotate the spaceship to the left
  }

  // Check the other keys, then carry on with our game cycle
}
```

In static-interface environments, there's no need to use an *enterFrame* loop to check for keypresses unless you are trying to detect specific keyboard combinations (i.e., multiple keys being pressed simultaneously). You should ordinarily use *keyDown* and *keyUp* event handlers, which are triggered precisely once for each keypress and key release. When using *keyUp* and *keyDown* event handlers, you need not concern yourself with whether the key is still being pressed at any given instant. This allows you to detect keypresses accurately even if the user releases the key between frames, and it also prevents you from checking the same key twice if it was pressed only once. In any case, you will ordinarily use the *Key.getCode()* and *Key.getASCII()* methods to check for the last key pressed within a *keyDown* or *keyUp* event handler.

Order of Execution

Some movies have code dispersed across multiple timelines and multiple clip event handlers. It's not uncommon, therefore, for a single frame to require the execution of many separate blocks of code—some in event handlers, some on frames in clip timelines, and some on the main timelines of documents in the Player. In these situations, the order in which the various bits of code execute can become quite complex and can greatly affect a program's behavior. We can prevent surprises and guarantee that our code behaves as desired by becoming familiar with the order in which event handlers execute relative to the various timelines in a movie.

Asynchronous event handlers execute independently of the code on a movie's timelines. Button event handlers, for example, are executed immediately when the event that they handle occurs, as are handlers for the *mouseDown*, *mouseUp*, *mouseMove*, *keyDown*, and *keyUp* events.

Handlers for the movie-playback events, however, execute in order, according to the progression of the movie, as shown in Table 10-3.

Table 10-3. Movie Clip Event Handler Order of Execution

Event Handler	Execution Timing
load	Executes in the first frame in which the clip is present on stage after parent-timeline code executes, but before clip-internal code executes, and before the frame is rendered.
unload	Executes in the first frame in which the clip is not present on stage, before parent-timeline code executes.
enterFrame	Executes in the second and all subsequent frames in which the clip is present on stage. It is executed before parent-timeline code executes and before clip-internal code executes.
data	Executes in any frame in which data is received by the clip. If triggered, it executes before clip-internal code executes and before *enterFrame* code executes.

It's easier to see the effect of the rules in Table 10-3 with a practical example. Suppose we have a single-layer movie with four keyframes in the main timeline. We attach some code to each keyframe. Then, we create a second layer where we place a movie clip at frame 1, spanning to frame 3, but not present on frame 4. We add *load*, *enterFrame*, and *unload* handlers to our clip. Finally, inside the clip, we create three keyframes, each of which also contains a block of code. Figure 10-2 shows what the movie looks like.

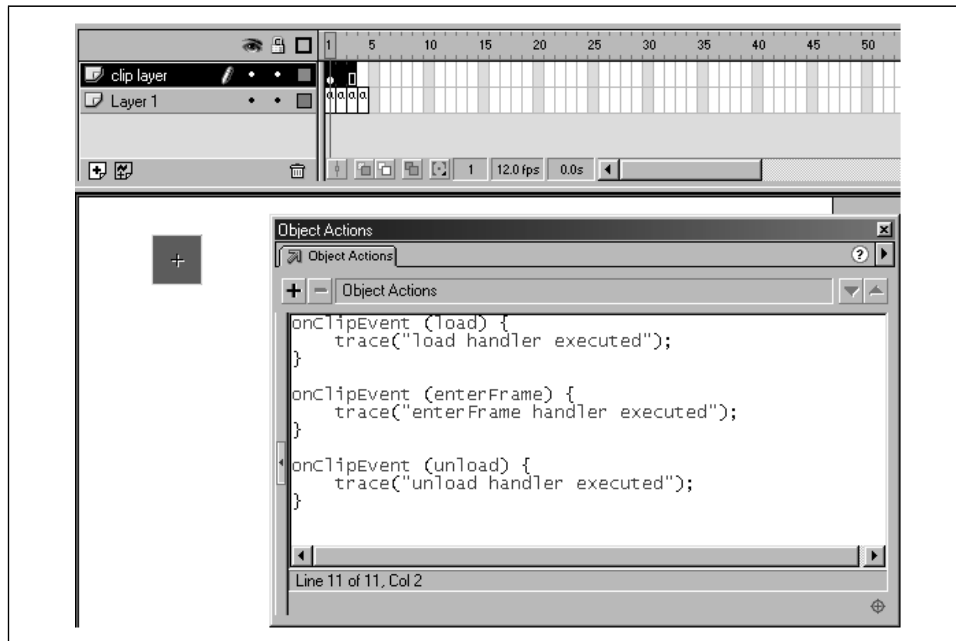


Figure 10-2. A code execution order test movie

When we play our movie, the execution order is as follows:

,ch10.16364 Page 222 Monday, April 16, 2001 1:52 PM


```
========FRAME 1=======
1) Main timeline code executed
2) load handler executed
3) Clip-internal code, frame 1, executed

========FRAME 2=======
1) enterFrame handler executed
2) Clip-internal code, frame 2, executed
3) Main timeline code executed

========FRAME 3=======
1) enterFrame handler executed
2) Clip-internal code, frame 3, executed
3) Main timeline code executed

========FRAME 4=======
1) unload handler executed
2) Main timeline code executed
```

The execution order of the code in our sample movie demonstrates some important rules of thumb to remember when coding with event handlers:

- Code in a *load* handler is executed before internal clip code, so a *load* handler may be used to initialize variables that are used immediately on frame 1 of its associated clip.
- Before a movie clip is instantiated on a frame, the code of that frame is executed. Therefore, user-defined variables and functions in a movie clip are not available to any code on its parent timeline until the frame *after* the clip first appears on stage, even if those variables and functions are declared in the clip's *load* handler.
- The *enterFrame* event never occurs on the same frame as the *load* or the *unload* event. The *load* and *unload* events supplant *enterFrame* for the frames where a clip appears on the Stage and leaves the Stage.
- On each frame, code in a clip's *enterFrame* handler is executed before code on the clip's parent timeline. Using an *enterFrame* handler, we may, therefore, change the properties of a clip's parent timeline and then immediately use the new values in that timeline's code, all on the same frame.

Copying Clip Event Handlers

A quick point that has major ramifications: movie clip event handlers are duplicated when a movie clip is duplicated via the *duplicateMovieClip()* function. Suppose, for example, we have a movie clip on stage called `square`, which has a *load* event handler defined:

```
onClipEvent (load) {
  trace("movie loaded");
}
```

,ch10.16364 Page 223 Monday, April 16, 2001 1:52 PM

What happens when we duplicate `square` to create `square2`?

```
square.duplicateMovieClip("square2", 0);
```

Because the *load* handler is copied to `square2` when we duplicate `square`, the birth of `square2` causes its *load* handler to execute, which displays "movie loaded" in the Output window. By using this automatic retention of handlers, we can create slick recursive functions with very powerful results. For a demonstration that only scratches the surface of what's possible, refer to Example 10-2.

Refreshing the Screen with updateAfterEvent

As we learned earlier in "Order of Execution," the *mouseDown*, *mouseUp*, *mouseMove*, *keyDown*, and *keyUp* event handlers are executed immediately upon the occurrence of those events. Immediately means *immediately*—even if the event in question occurs between the rendering of frames.

This immediacy can give a movie great responsiveness, but that responsiveness can easily be lost. By default, the visual effects of a *mouseDown*, *mouseUp*, *mouseMove*, *keyDown*, or *keyUp* event handler are not physically rendered by the Flash Player until the next available frame is rendered. To really see this in action, create a single-frame movie with a frame rate of 1 frame per second, and place a movie clip with the following code on stage:

```
onClipEvent (mouseDown) {
  _x += 2;
}
```

Then, test the movie and click the mouse as fast as you can. You'll see that all your clicks are registered, but the movie clip moves only once per second. So, if you click 6 times between frames, the clip will move 12 pixels to the right when the next frame is rendered. If you click 3 times, the clip will move 6 pixels. Each execution of the *mouseDown* handler is registered between frames, but the results are displayed only when each frame is rendered. This can have dramatic effects on certain forms of interactivity.

Fortunately, we can force Flash to immediately render any visual change that takes place during a user-input event handler without waiting for the next frame to come around. We simply use the *updateAfterEvent()* function from inside our event handler, like this:

```
onClipEvent (mouseDown) {
  _x += 2;
  updateAfterEvent();
}
```

,ch10.16364 Page 224 Monday, April 16, 2001 1:52 PM

The *updateAfterEvent()* function is available for use only with the *mouseDown*, *mouseUp*, *mouseMove*, *keyDown*, and *keyUp* events. It is often essential for smooth and responsive visual behavior associated with user input. Later, in Example 10-8, we'll use *updateAfterEvent()* to ensure the smooth rendering of a custom pointer. Note, however, that button events do not require an explicit *updateAfterEvent()* function call. Buttons naturally update between frames.

Code Reusability

When using button events and movie clip events, don't forget the code-centralization principles we learned in Chapter 9. Always try to prevent unnecessary duplication and intermingling of code across movie elements. If you find yourself entering the same code in more than one event handler's body, it may not be wise to attach that code directly to the object. Try generalizing your code, pulling it off the object and placing it in a code repository somewhere in your movie; often the best place is the main timeline.

In many cases, it's a poor idea to hide statements inside a button or clip handler. Remember that encapsulating your code in a function and calling that function from your handler makes your code reusable and easy to find. This is particularly true of buttons—I rarely place anything more than a function-invocation statement directly on a button. For movie clips, you'll need to employ keener judgment, as placing code directly on clips can often be a healthy part of a clean, self-contained code architecture. Experiment with different approaches until you find the right balance for your needs and skill level. Regardless, it always pays to be mindful of redundancy and reusability issues.

For an example of the difference between attaching code to buttons versus calling functions from buttons, see "Centralizing Code" in Chapter 9.

Dynamic Movie Clip Event Handlers

Early in this chapter, we learned about two kinds of events in Flash—those that are attached to movie clips and buttons and those that are attached to other data objects such as *XML* and *XMLSocket*. To create event handlers for data objects, we assign the handler function name as a property of the object. Recall the syntax to add a function dynamically:

```
myXMLDoc.onLoad = function () { trace("all done loading!"); };
```

Dynamic function assignment lets us change the behavior of the handler during movie playback. All we have to do is reassign the handler property:

```
myXMLDoc.onLoad = function () { gotoAndPlay("displayData"); };
```

,ch10.16364 Page 225 Monday, April 16, 2001 1:52 PM

Or we can even disable the handler altogether:

```
myXMLDoc.onLoad = function () { return; };
```

Unfortunately, handlers of movie clip and button events are not nearly so flexible; they cannot be changed or removed during movie playback. Furthermore, movie clip event handlers cannot be attached to the main movie timeline of any movie! It's impossible to directly create an event handler for a movie's `_root` clip.

In order to work around these limitations, we can—in the case of the *enterFrame* and the user-input events—use empty movie clips to simulate dynamic event-handler removal and alteration. Empty movie clips even let us simulate `_root`-level events. We've already seen the technique in Chapter 8, where we learned how to create an event loop as follows:

1. Create an empty movie clip named `process`.
2. Place another empty clip called `eventClip` inside `process`.
3. On `eventClip`, attach the desired event handler. The code in the `eventClip`'s handler should target the `process` clip's host timeline, like this:

   ```
   onClipEvent (mouseMove) {
     _parent._parent.doSomeFunction();
   }
   ```

4. To export `process` for use with the *attachMovie()* function, select it in the Library and choose Options → Linkage. Set Linkage to Export This Symbol, and assign an appropriate identifier (e.g., "mouseMoveProcess").
5. Finally, to engage the event handler, attach the `process` clip to the appropriate timeline using *attachMovie()*.
6. To disengage the handler, remove the `process` clip using *removeMovieClip()*.

For step-by-step instructions on how to use this technique with the *enterFrame* event, see "Flash 5 Clip Event Loops" in Chapter 8.

Event Handlers Applied

We'll conclude our exploration of ActionScript events and event handlers with a few real-world examples. These are simple applications, but they give us a sense of how flexible event-based programming can be. The last two examples are available for download from the online Code Depot.

Example 10-7 makes a clip shrink and grow.

Example 10-7. Oscillating the Size of a Movie Clip

```
onClipEvent (load) {
  var shrinking = false;
```

,ch10.16364 Page 226 Monday, April 16, 2001 1:52 PM

Example 10-7. Oscillating the Size of a Movie Clip (continued)

```
  var maxHeight = 300;
  var minHeight = 30;
}

onClipEvent (enterFrame) {
  if (_height < maxHeight && shrinking == false) {
    _height += 10;
    _width  += 10;
  } else {
    shrinking = true;
  }

  if (shrinking == true) {
    if (_height > minHeight) {
      _height -= 10;
      _width  -= 10;
    } else {
      shrinking = false;
      _height += 10;     // Increment here so we don't
      _width  += 10;     // miss a cycle
    }
  }
}
```

Example 10-8 simulates a custom mouse pointer by hiding the normal system pointer and making a clip follow the mouse location around the screen. In the example, the *mouseDown* and *mouseUp* handlers resize the custom pointer slightly to indicate mouseclicks.

Example 10-8. A Custom Mouse Pointer

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

onClipEvent (mouseMove) {
  _x = _root._xmouse;
  _y = _root._ymouse;
  updateAfterEvent();
}

onClipEvent (mouseDown) {
  _width  *= .5;
  _height *= .5;
  updateAfterEvent();
}

onClipEvent (mouseUp) {
  _width  *= 2;
  _height *= 2;
  updateAfterEvent();
}
```

,ch10.16364 Page 227 Monday, April 16, 2001 1:52 PM

Finally, simply to prove the power of the ActionScript movie clip event handlers, Example 10-9 turns a movie clip into a customized button using *mouseMove* to check for rollovers, *mouseDown* and *mouseUp* to check for button clicks, and the *hitTest()* function to make hit detection a snap. This example assumes that the clip with the handlers has three keyframes labeled `up`, `down`, and `over` (corresponding with the usual button states).

Example 10-9. A Movie Clip Button

```
onClipEvent (load) {
  stop();
}

onClipEvent (mouseMove) {
  if (hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown) {
    this.gotoAndStop("over");
  } else if (!hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown) {
    this.gotoAndStop("up");
  }
  updateAfterEvent();
}

onClipEvent (mouseDown) {
  if (hitTest(_root._xmouse, _root._ymouse, true)) {
    buttonDown = true;
    this.gotoAndStop("down");
  }
  updateAfterEvent();
}

onClipEvent (mouseUp) {
  buttonDown = false;
  if (!hitTest(_root._xmouse, _root._ymouse, true)) {
    this.gotoAndStop("up");
  } else {
    this.gotoAndStop("over");
  }
  updateAfterEvent();
}
```

Onward!

With statements, operators, functions, and now events and event handlers under our belt, we've learned how all of the internal tools of ActionScript work. To round out our understanding of the language, in the next three chapters we'll explore three extremely important datatypes: arrays, objects, and movie clips.