

CHAPTER 10

Events and Event Handling

You've learned a lot about composing instructions for the ActionScript interpreter to execute. By now you should be pretty comfortable telling the interpreter *what* you want it to do, but how do you 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

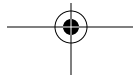
As a movie plays, the timeline's playhead travels from frame to frame. Each time the playhead enters a new frame, the interpreter executes any code attached to that frame. After the code on a frame has been executed, the screen display is updated and sounds are played. Then, the playhead proceeds to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

Event-Based Asynchronous Code Execution

Some code does not execute in a predetermined sequence. Instead, it executes when the ActionScript interpreter notices that an *event* has occurred. Many events involve



a user action, such as clicking the mouse, resizing the movie, or pressing a key. Other events happen without user intervention, such as a sound completing or an XML document loading. 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 interactivity.

This chapter explores asynchronous (event-based) programming in Flash.

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 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 can also be categorized more practically according to the object to which they pertain. All events happen relative to some object in the Flash environment. We'll study objects in depth in Chapter 12, but for now, simply assume that content in Flash (buttons, movie clips, text fields, etc) is normally represented as ActionScript objects and that we define events in relation to those objects.

The core ActionScript objects and classes that support events are:

- *Button*
- *Key*
- *LoadVars*
- *LocalConnection*
- *Mouse*
- *MovieClip*
- *Selection*

- *SharedObject*
- *Sound*
- *Stage*
- *TextField*
- *XML*
- *XMLSocket*

For details on the specific events supported by these objects, see each object's entry in the *Language Reference*.

Event Handling

Not every event triggers the execution of code or affects a movie. For example, a user can 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 create either an *event handler* or an *event listener* that describes the action to take when the specified event occurs.

Due to its unusual evolution, ActionScript has four different ways of handling events:

- Event handler properties
- Event listeners
- Button event handlers with the syntax:

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

- Movie clip event handlers with the syntax:

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

Let's take a look at these event-handling mechanisms individually. Much of the following discussion expects you to understand objects, which are covered in Chapter 12.

Event Handler Properties

Event handlers are so named because they *catch*, or *handle*, the events in a movie. An event handler is simply a function that is invoked automatically when a particular event occurs. We've learned that all events occur in relation to some object. Hence, in order to create an event handler, we must specify three things:

- The object to which the event applies
- The name of the event handler
- The function to be executed when the event occurs (known as the event handler's *callback function*)

For example, suppose we want to animate a movie clip named `ball` by changing its position every time a frame passes in the Flash Player. If we define an `onEnterFrame()` handler for the `ball` clip, it will be executed once for each tick of the movie's frame rate. So, given a movie clip, `ball`, we can create an `onEnterFrame()` event handler for `ball` as follows:

1. Create a new empty Flash movie.
2. Create a new movie clip symbol, named `ballSymbol`, that contains a circle shape.
3. Place an instance of `ballSymbol` on *Layer 1*.
4. Name the `ballSymbol` instance `ball`.
5. On frame 1 of *Layer 1* of the main movie timeline, attach the following code:

```
ball.onEnterFrame = moveDown;
function moveDown () {
    // Move the ball down 5 pixels with each frame.
    this._y += 5;
}
```

In this code, `ball` is the object to which the event applies, `onEnterFrame` is the event we wish to handle, and `moveDown` is the function that should run when `onEnterFrame()` occurs.

Notice that we assign the `moveDown` function reference, not the return value of a `moveDown()` function call, to `onEnterFrame`. Therefore, the function invocation operator `()` must not be included.

```
// The following is incorrect! The () operator must not be used.
ball.onEnterFrame = moveDown();
// The following is correct.
ball.onEnterFrame = moveDown;
```

To be more succinct when assigning an event handler, we can specify the callback function with a function literal, as follows:

```
ball.onEnterFrame = function () {
    this._y += 5;
};
```

The `ball` example shows how to specify a handler for an clip from outside the clip (i.e., from the main timeline). To define an event handler for the current movie clip from a frame on its own timeline, use the following syntax instead:

```
this.theEventHandler = function () {
    statements
};
```

In general, then, the syntax of an event handler is either:

```
someObject.someEventHandler = function () {
    statements
};
```

in which case the event handler is defined as a function literal, or:

```
someObject.someEventHandler = someFunction;
```

in which case *someFunction* must also be defined elsewhere, as in:

```
function someFunction () {
    statements
}
```

The latter approach is preferred in situations where the event handler is added and removed repeatedly, or where multiple objects use the same function as an event handler.

Event handlers can also receive parameters generated by the event. For example, in the following code, a text field, `userName_txt`, defines an event handler, `onSetFocus()`, that receives a reference to the previously focused text field via the parameter `oldFocus`. The handler uses `oldFocus` to color the previously focused text field black.

```
userName_txt.onSetFocus = function (oldFocus) {
    // Color the previously focused field black.
    oldFocus.backgroundColor = 0x000000;
    // Color the newly focused field orange.
    this.backgroundColor = 0xF5BF70;
}
```

Object-oriented programmers will recognize event handler syntax as identical to the syntax for defining an object *method*. This is not a matter of coincidence. Event handlers are, in fact, methods that happen to be invoked automatically by the interpreter. And since methods in ActionScript are object properties that store functions, the event handler format shown in our `onSetFocus()` example is known as an *event handler property*.

Within the body of an event handler callback function, the `this` keyword refers to the object for which the event was triggered. In our `ball`'s `onEnterFrame()` handler, we used `this` to refer to the `ball` movie clip object, which we moved down by 5 pixels:

```
this._y += 5;
```

Similarly, in our `userName_txt.onSetFocus()` handler `this` refers to `userName_txt`:

```
this.backgroundColor = 0xF5BF70;
```

To remove an event handler from an object, use the *delete* operator as follows:

```
delete theObject.theEventHandler;
```

For example, executing the following code from the main timeline causes the `ball` movie clip to stop responding to the `onEnterFrame()` event:

```
delete ball.onEnterFrame;
```

If the code were attached to the `ball` movie clip itself, you could instead write:

```
delete this.onEnterFrame;
```

For a list of event handlers offered by a specific class or object, see the appropriate entry in the *Language Reference*. In particular, be sure to study the *Button* and *MovieClip* classes, which define the majority of user-interaction events. For example, to cause a `submitForm()` function to execute when a Submit button is pressed, we use:

```
submit_btn.onRelease = submitForm;  
function submitForm () {  
    // Validate data and send it to the server...  
}
```

where `submit_btn` is the button object, the `onRelease` event is invoked when the button is pressed and then released, and `submitForm()` is a custom function that sends the form data to the server. For a complete form-submission example, see Chapter 17.

Event handlers are convenient, but they also suffer from a shortcoming: only one callback function can be *registered* (assigned) to an event handler at a time. That is, only one action can be performed for each event when using event handler properties. We'll learn how to overcome this limitation in the next section using *listener events*.

Listener Events

Conceptually, *listener events* are similar to event handler properties. Like event handler properties, listener events are handled with a specially named object method (i.e., a property containing a callback function) that is invoked when a given event occurs. However, listener events add two important event handling features:

- An object can respond to events that occur for other objects, not just itself.
- More than one object can respond to an event.

For example, suppose we want to center a text field, `title_txt`, in a movie by setting its position each time the movie changes size. Consulting the *Language Reference*, we determine that when a movie changes size the `Stage.onResize()` listener event is triggered. Our text field is not an instance of `Stage`, yet we want it to respond to `Stage.onResize()`. This is no problem. Because `Stage.onResize()` is a listener event, it will let `title_txt` (and any other object) sign up to be notified when `onResize()` occurs.

Here's how it works. First, we create our `onResize()` method directly on `title_txt`, just as we created our earlier event handler properties:

```
title_txt.onResize = function () {  
    // Center the text field.  
    // Stage.width and Stage.height always store the current movie dimensions.  
    // In the callback function body we refer to title_txt as this.
```

```

        this._x = Stage.width/2 - this._width/2;
        this._y = Stage.height/2 - this._height/2;
    }

```

Next, we must tell the *Stage* object that we want it to execute *title_txt*'s *onResize()* method when the *onResize()* event occurs. This step is known as "adding a listener." The so-called *listener* is the object (in our case, *title_txt*) that wishes to receive event notifications from the *event source* or, synonymously, the *event broadcaster* (in our case, *Stage*). All listener event sources provide a special method, *addListener()*, that is used to register new listeners. For example, here's how we add *title_txt* as a *Stage* listener:

```

Stage.addListener(title_txt);

```

Henceforth, when the *Stage.onResize()* event occurs, *title_txt.onResize()* is invoked. In fact, as a *Stage* listener, *title_txt* will receive event notices for all of *Stage*'s listener events (though in *Stage's case*, it defines only one listener event). Some event sources define both event handler properties and listener events. In such a case, a listening object will receive notifications for the listener events only, not the event handler properties. Each object's supported listener events and event handler properties are listed in the *Language Reference*.

Generally, an event is implemented as a handler property when it is of interest only to the object that generates it. Listener events are more common when many different kinds of objects have interest in a single event (for example, a movie clip, a button, and a text field may all want to know when a movie is resized, so *Stage.onResize()* is a listener event).

Here is the general syntax for creating an *event listener* (an object that defines a method that handles a listener event):

```

listenerObject.eventName = function () {
    statements
}
eventSource.addListener(listenerObject);

```

The *listenerObject* can be any object. The *eventName* is the name of the predefined event to listen for (e.g., *onResize* in *title_txt.onResize()*). The *eventSource* is the object that notifies *listenerObject* when *eventName* occurs. Therefore, unless *eventName* is one of the events broadcast by *eventSource*, the event listener method will never be executed. For details on specific event sources see the *Key*, *Mouse*, *Selection*, *TextField*, and *Stage* objects in the *Language Reference*.

Getting back to our example, suppose we want to center a second text field, *subtitle_txt*, underneath *title_txt*. We follow the same process we did with *title_txt*. First, we create our *onResize()* method directly on *subtitle_txt*:

```

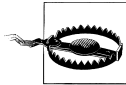
subtitle_txt.onResize = function () {
    this._x = Stage.width/2 - this._width/2;
    this._y = (Stage.height/2) + (title_txt._height/2) + 10;
}

```

Then, we register `subtitle_txt` as a *Stage* listener:

```
Stage.addListener(subtitle_txt);
```

Note that the `onResize()` methods must be written appropriately for each object that we wish to reposition. In this case, we reposition `title_txt` based on the height and width of the Stage and the text object itself. We offset `subtitle_txt` vertically, relative to the center of the Stage and `title_txt`'s height.



In order to function properly, the `Stage.onResize()` event requires `Stage.scaleMode` and `Stage.align` to be set, as shown in Example 10-1.

Adding a listener to an event source is sometimes referred to as *subscribing* the listener. We now have two text fields—each with their own independent `onResize()` method—that respond to a single event, `Stage.onResize()`. We can also register movie clips, buttons, or any other objects as *Stage* listeners, allowing for a entire movie's layout to adjust dynamically when the movie is resized. When multiple listeners are registered to a single event source, their methods are executed in the order the listeners were added.

To remove (or *unsubscribe*) any listener from an event source, use the `removeListener()` method. For example, the following code causes `subtitle_txt` to stop receiving `onResize` events, so it won't reposition itself when the movie is resized:

```
Stage.removeListener(subtitle_txt);
```

The general form to remove a listener is:

```
eventSource.removeListener(listenerObject);
```

Example 10-1 shows our text field centering code in its entirety. To test it, attach the code to the first frame of the main timeline in a new movie; then export the movie and try resizing the movie's window. For complete information on building resizable interfaces, see the *Stage* object in the *Language Reference*.

Example 10-1. Keeping two text fields centered in a movie

```
// Set the scaling behavior of the movie
Stage.scaleMode = "noScale";
Stage.align = "LT";

// Create the first text field
this.createTextField("title_txt", 1, 0, 0, 180, 20);
title_txt.text = "Welcome to my website.";
title_txt.border = true;

// Assign its listener callback function
title_txt.onResize = function () {
    this._x = Stage.width/2 - this._width/2;
    this._y = Stage.height/2 - this._height/2;
}
```

Example 10-1. Keeping two text fields centered in a movie (continued)

```
// Create the second text field
this.createTextField("subtitle_txt", 2, 0, 0, 180, 20);
subtitle_txt.text = "Thanks for visiting.";
subtitle_txt.border = true;

// Assign its listener callback function
subtitle_txt.onResize = function () {
    this._x = Stage.width/2 - this._width/2;
    this._y = (Stage.height/2) + (title_txt._height/2) + 10;
}

// Register the text fields as Stage listeners
Stage.addListener(subtitle_txt);
Stage.addListener(title_txt);
```

Flash 5's on() and onClipEvent() Handlers

Earlier in this chapter, we discussed how to assign callback functions to the event handler properties *MovieClip.onEnterFrame()* and *TextField.onSetFocus()*. In Flash 5, these kinds of event handler properties were available on *XML* and *XMLSocket* only. Movie clips and buttons used idiosyncratic event handler syntax attached directly to physical clips or buttons in the Flash authoring tool. Button event handlers were defined using *on(eventName)*, and movie clip event handlers were defined using *onClipEvent(eventName)*, where *eventName* specified the event to be handled. As of Flash MX, use of these older forms is generally discouraged. Event handler properties and event listeners are preferred because they promote the separation of code from the visual elements of a movie. However, the older syntax is still supported, and there are a few special cases where it is actually required (for examples, see *MovieClip.onLoad()* and *Button keyPress* in the *Language Reference*).

Flash 5–style button *on()* event handlers take the form:

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

The *eventName* is the name of the corresponding Flash MX event handler property, without the word “on” and with the first letter lowercase. For example, the *onRelease()* property in Flash MX was implemented as follows in Flash 5:

```
on (release) {
    statements
}
```

When exporting to Flash 5 format, you must use the Flash 5–style event syntax. 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();  
}
```

To upgrade such code to Flash MX’s preferred syntax, using the same callback handler to respond to both events, you’d use:

```
theButton.onRollOver = playRandomSound;  
theButton.onRollOut = playRandomSound;
```

or, even more tersely:

```
theButton.onRollOver = theButton.onRollOut = playRandomSound;
```

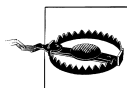
Flash 5–style movie clip *onClipEvent()* handlers take the form:

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

Again, the *eventName* is the name of the corresponding Flash MX event handler property, with the word “on” removed. For example, the Flash MX *MovieClip.onEnterFrame()* event handler was implemented as follows in Flash 5:

```
onClipEvent (enterFrame) {  
    statements  
}
```

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



Events introduced after Flash 5 are not available in *on()* or *onClipEvent()* form. For example, there is no such thing as *onClipEvent(setFocus)*, because the *onSetFocus()* event was introduced in Flash MX and first supported by Flash Player 6.

As of Flash MX, movie clip instances can take both *onClipEvent()* and button-style *on()* event handlers. For details, see “Using Movie Clips as Buttons” in Chapter 13.

Attaching Event Handlers to Buttons and Movie Clips

To attach a Flash 5–style *on()* or *onClipEvent()* event handler to a button or a movie clip, we must physically place the code of the handler onto the desired button or clip. We can do so in the Flash authoring tool only, by selecting the object on stage and entering the appropriate code in the Actions panel, as shown in Figure 10-1.



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

Let's try making two simple event handler functions, one for a button and another for a movie clip. To create a button event handler, follow these instructions:

1. Create a new, blank 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.

When the movie plays and we click and release the mouse over the button, the *release* event is detected by the interpreter, which executes the *on(release)* event handler. Each time we press and release the mouse, 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.

1. Create a new, blank Flash movie.
2. On the main movie Stage, draw a rectangle.
3. Select Insert → Convert to Symbol.
4. In the Convert to Symbol 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) {  
    this._visible = false;  
}  
onClipEvent (keyUp) {  
    this._visible = true;  
}
```

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.

Unlike Flash 6’s event handler properties and event listeners, Flash 5’s *on()* and *onClipEvent()* handlers cannot be attached and removed via ActionScript while the movie is playing. Therefore, in Flash 5 the following event handler property assignment is illegal:

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

We’ll see how to work around this shortcoming later in this chapter under “Dynamic Movie Clip Event Handlers.”

Event Handler Lifespan

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 *TextField* object is deleted, any event handlers associated with those objects die with them. An object must be present on stage or in code for its handlers to remain active.

Event Handler Scope

As with any function, the statements in an event handler or listener execute within a predefined scope. Scope dictates where the interpreter looks to resolve unqualified variable references in an event handler’s body. We’ll consider event handler scope in relation to the four event implementations supported by Flash.

Scope of Event Handler Properties and Event Listener Methods

Event listener methods and event handler properties are scoped exactly as functions are scoped. An event callback function has a scope chain that is defined when the function is defined. Furthermore, event handler properties and event listener methods define a local scope. All the rules of function scope described in Chapter 9 apply to event handler properties and event listener methods.

Scope of `onClipEvent()` Handlers

Unlike event handler properties and event listener methods, `on()` and `onClipEvent()` handlers *do not define a local scope*! When we attach an `onClipEvent()` handler to a clip, the scope of the handler is the clip, not the event handler. This means that all variables are retrieved from the clip's timeline. For example, if we attach an `onClipEvent(enterFrame)` 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 the `onClipEvent()` 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 an `onClipEvent()` 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` on the main timeline that has a variable called `xVelocity` in it. To access `xVelocity` from inside `ball`'s `onClipEvent()` handler, we simply refer to it directly, like this:

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

Supplying the path to the variable as `_root.ball.xVelocity` is optional, 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 can use `_x`, and instead of using `ball.gotoAndStop(5)`, we can 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 an `onClipEvent()` handler, like this:

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

However, unqualified variable references are ambiguous and behave differently in an event handler property or event listener method than they do in an `onClipEvent()` handler. Therefore, it's prudent to always use the `this` keyword when referring to the movie clip that defines the handler:

```
// Preferred reference to the clip with the onClipEvent() handler:  
this._x  
// Legal, but discouraged reference to the clip with the onClipEvent() handler:  
_x  
  
// Preferred:  
this.gotoAndStop(12);  
// Discouraged:  
gotoAndStop(12);
```

Additionally, use of `this` is 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`, etc. to start playing, depending on the current frame of the `ball` clip:

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

The `this` keyword should also be 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 `onClipEvent()` handler:

```
duplicateMovieClip()  
loadMovie()  
loadVariables()  
removeMovieClip()  
startDrag()  
unloadMovie()
```

For example:

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

You'll learn all about the dual nature of these functions in Chapter 13 under "Method Versus Global Function Overlap Issues".

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 the clip's name. In fact, using `this`, we can even pass the current clip as a reference to a function without ever knowing the current clip's name. Here's some code to demonstrate:

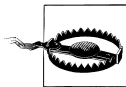
```
// CODE ON MAIN TIMELINE  
// Here is a generic function that moves any clip  
function move (clip, x, y) {
```

```

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 Flash Player 5.0.30.0, a bug prevents *gotoAndStop()* and *gotoAndPlay()* from working inside a clip handler when used with string literal labels. Such commands are simply ignored. For example, this does not work:

```

onClipEvent (load) {
    gotoAndStop("intro"); // Won't work in Flash 5.0.30.0
}

```

To work around the bug, use a self-reflexive clip reference (i.e., *this*), as in:

```

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

```



Remember that statements in *onClipEvent()* handlers are scoped to the *clip's* timeline, not to the handler function's local scope or 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 must refer to the *moveBall()* function explicitly on the main timeline using *_root* like this:

```

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

```

Scope of on() Handlers

The scope of an *on()* handler depends on the type of the object to which it is attached. When attached to a button, an *on()* handler is scoped to the timeline upon

which the button resides. When attached to a movie clip, an *on()* handler is scoped to the clip that bears the handler, as with an *onClipEvent()* handler. Prior to Flash MX, movie clips did not support *on()* handlers (only Flash 5 buttons did).

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
}
```

In contrast, if we place an *on()* handler on a movie clip object named *ball* and declare the variable *speed* in the handler, *speed* is scoped to *ball*:

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

Inside an *on()* handler for a button object, the *this* keyword refers to the timeline on which the button resides:

```
// CODE FOR BUTTON HANDLER
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;
}
```

But inside a movie clip's *on()* handler, the *this* keyword refers to the clip that bears the handler:

```
// CODE FOR MOVIE CLIP HANDLER
on (release) {
    // Make the clip with the handler 50% transparent
    this._alpha = 50;
    // Move the clip with the handler 10 pixels to the right
    this._x += 10;
}
```

See the *MovieClip* and *Button* classes in the *Language Reference* for details on movie clip and button events.

Scope Summary

The various scopes in the previous discussion can be confusing. Use the general rules outlined in Table 10-1 to determine the scope of a specific handler in Flash MX. Note that of the following event handling mechanisms, Flash 5 supported only “MovieClip with *onClipEvent()* handler” and “Button with *on()* handler.”

Table 10-1. Event scope summary

Event handling mechanism	Example	Scope
Event handler property	<code>someTextField.onSetFocus = function () { statements }</code>	The clip in which function declaration occurred.*
Event listener method	<code>GUIManager.onResize = function () { statements } Stage.addListener(GUIManager);</code>	The clip in which function declaration occurred.*
MovieClip with event handler property	<code>theClip_mc.onPress = function () { statements }</code>	The clip in which function declaration occurred.*
MovieClip with <i>on()</i> handler	<code>on (press) { statements }</code>	The clip that bears the handler in the authoring tool.
MovieClip with <i>onClipEvent()</i> handler	<code>onClipEvent (enterFrame) { statements }</code>	The clip that bears the handler in the authoring tool.
Button with event handler property	<code>submit_btn.onRelease = function () { statements }</code>	The clip in which function declaration occurred.*
Button with <i>on()</i> handler	<code>on (release) { statements }</code>	The clip on whose timeline the button resides.

* For handlers and listeners assigned within a function, the scope is that function (in accordance with the rules of nested function scope, discussed in Chapter 9).

Finally, here’s a scenario to test your understanding of event handler scope. Before reading the answer, see if you can determine it yourself.

Suppose a movie clip instance named `intro_mc` is placed on the main timeline of a movie, and the following code is attached to the main timeline:

```
intro_mc.onPress = function () {
    trace(this); // Here, this refers to intro_mc.
    this.play() // This line makes intro_mc play.
    play();     // Why won't this line make intro_mc play?
}
```

Question: In `intro_mc`’s `onPress()` callback function, why does `this` refer to `intro_mc`, whereas unqualified statements do not? In other words, why doesn’t line 3 of the handler body make `intro_mc` play?

Answer: this is a reference to the object through which `onPress()` was invoked: `intro_mc`. But statements inside the `onPress()` function are executed in the scope of the timeline on which the function was defined—the main timeline. Hence, the standalone `play()` command plays the main timeline, not `intro_mc`.



Moral of the story: obviously, the location of the function declaration can change, so it's a little dangerous to use unqualified references inside handlers at all. Qualify all nonlocal identifiers within handlers with `this` or some other explicit object reference.

Values of the `this` Keyword

In our study of ActionScript's event handling tools, we've encountered the `this` keyword several times. Within an event handler, `this` normally refers to the object that received notification of the event. However, Flash's evolution through multiple event systems has complicated the issue. Table 10-2 provides a guide to the values of the `this` keyword in ActionScript's various event handling mechanisms.

Table 10-2. Values of the `this` keyword

Event handling mechanism	Value of <code>this</code>
Event handler property	The object that defines the handler property.
Event listener method	The object registered as a listener.
MovieClip with event handler property	The movie clip that defines the handler property.
MovieClip with <code>on()</code> handler	The clip that bears the handler in the authoring tool.
MovieClip with <code>onClipEvent()</code> handler	The clip that bears the handler in the authoring tool.
Button with event handler property	The button that defines the handler property.
Button with <code>on()</code> handler	The clip on whose timeline the button resides.

Flash 5–style `onClipEvent()` Order of Execution

Some movies have code dispersed across multiple timelines and `on()` and `onClipEvent()` 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.

User-input 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 `onMouseDown()`, `onMouseUp()`, `onMouseMove()`, `onKeyDown()`, and `onKeyUp()` events.

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

Table 10-3. Flash 5–style `onClipEvent()` order of execution

Event handler	Execution timing
<code>load</code>	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. See special notes listed under <code>MovieClip.onLoad()</code> in the <i>Language Reference</i> .
<code>unload</code>	Executes in the first frame in which the clip is not present on stage, before parent-timeline code executes.
<code>enterFrame</code>	Executes in the second frame 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.
<code>data</code>	Executes in any frame in which data is received by the clip. If triggered, it executes before clip-internal code executes and before <code>enterFrame</code> 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` `onClipEvent()` 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.

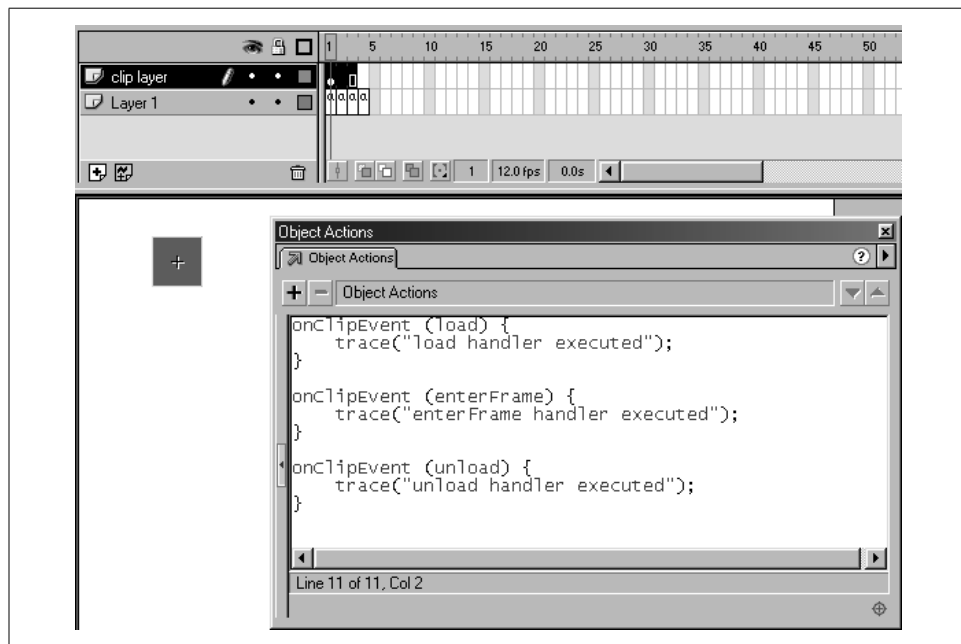


Figure 10-2. A code execution order test movie

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

```
=====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 movie clip event handlers:

- Code in a *load* handler is executed before internal clip code, so a *load* handler can be used to initialize variables that are used immediately on frame 1 of its associated clip, but only if the handler is created with the *onClipEvent()* syntax. See *MovieClip.onLoad()* in the *Language Reference* for further details.
- 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. For a solution to this problem, see *#initclip* in the *Language Reference*.
- 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 or leaves the Stage.
- On each frame, code in a clip's *enterFrame* handler is executed before code on the clip's parent timeline. Therefore, using an *enterFrame* handler, we can 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 Movie Clip Event Handlers

Here's a quick point that has important ramifications: *onClipEvent()* and *on()* handlers are duplicated when a movie clip is duplicated via the *duplicateMovieClip()* function. However, event handler properties, such as *onEnterFrame()*, are not!

Suppose, for example, we have a movie clip on stage named `square`, which has an `onClipEvent(load)` handler defined:

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

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

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

Answer: Because the `onClipEvent(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, refer to `MovieClip.onLoad()` in the *Language Reference*.

In contrast, suppose we have a movie clip named `circle` and we assign it an `onEnterFrame()` event handler property as follows:

```
circle.onEnterFrame = function () {
    trace(this._name);
}
```

What happens when we duplicate `circle` to create `circle2`?

```
circle.duplicateMovieClip("circle2", 1);
```

Answer: Only `circle`'s name appears in the Output window. The `onEnterFrame()` event handler property is not copied to `circle2`, so `circle2`'s name does not appear in the Output window. To force every duplicate of the `circle` clip to define an `onEnterFrame()` event handler property, we can set the property on a frame of `circle`'s timeline:

```
this.onEnterFrame = function () {
    trace(this._name);
}
```

Even more eloquently, we can define `circle` as a `MovieClip` subclass, and define the `onEnterFrame()` handler on the `circle` constructor's prototype. There's much more to come on that topic in Chapter 14.

Refreshing the Screen with `updateAfterEvent()`

The `MovieClip` event handler properties `onMouseDown()`, `onMouseUp()`, `onMouseMove()`, `onKeyDown()`, and `onKeyUp()` are executed immediately upon the occurrence of their corresponding 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 improper coding. By default, the visual effects of *onMouseDown()*, *onMouseUp()*, *onMouseMove()*, *onKeyDown()*, and *onKeyUp()* are not physically rendered by the Flash Player until the next available frame is rendered. To see this in action, create a single-frame movie with a frame rate of 1 frame per second. Place a movie clip named *circle_mc* on the Stage and attach the following code to frame 1 of the main timeline:

```
circle_mc.onMouseDown = function () {
    this._x += 10;
}
```

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 still moves only once per second. So, if you click 6 times between frames, the clip will move 60 pixels to the right when the next frame is rendered. If you click 3 times, the clip will move 30 pixels. The cumulative effect of each execution of the *onMouseDown()* event is "remembered" between frames, but the results are displayed only when each frame is rendered. This can have dramatic effects on certain forms of interactivity.

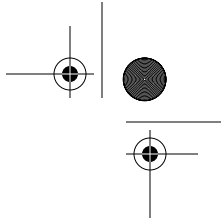
Fortunately, we can force Flash to immediately render any visual change that takes place during a user-input event handler, without waiting for the next frame to come around. We simply use the *updateAfterEvent()* function from inside our event handler, like this:

```
circle_mc.onMouseDown = function () {
    this._x += 10;
    updateAfterEvent();
}
```

The *updateAfterEvent()* function is available for use only with the *onMouseDown()*, *onMouseUp()*, *onMouseMove()*, *onKeyDown()*, and *onKeyUp()* *MovieClip* events and the *setInterval()* function. It is often essential for smooth and responsive visual behavior associated with user input. Later, in Example 10-3, 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. Button events update the Stage automatically between frames.



The *onMouseDown()*, *onMouseMove()*, and *onMouseUp()* listeners were added to the *Mouse* object in Flash Player 6. Likewise, the *onKeyDown()* and *onKeyUp()* listeners were added to the *Key* object. Generally, these *Mouse* and *Key* listeners are preferred over the analogous *MovieClip* events. However, in Flash Player 6, the *updateAfterEvent()* function can be used only in *MovieClip* events. Hence, when a screen refresh is required between frames, the *MovieClip* event handlers must be used. Future versions of the Player will likely include support for *updateAfterEvent()* from *Mouse* and *Key* listeners.



Code Reusability

When using event handlers and listeners, don't forget the code-centralization principles discussed in Chapter 9. Avoid 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, you should try defining it in a single function and pointing each event handler to that function. Try generalizing your code, pulling it off the object and placing it in a code repository somewhere in your movie; global code used throughout a movie should be attached to the `_global` object.

In almost all cases, it's a poor idea to hide statements inside an `on()` or `onClipEvent()` handler. Remember that encapsulating your code in a function makes your code reusable and easy to find. This is particularly true of buttons—I rarely place any code on a button in Flash MX, and I rarely use more than a single function-invocation statement in a Flash 5 `on()` handler. 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.

Dynamic Movie Clip Event Handlers

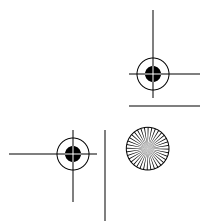
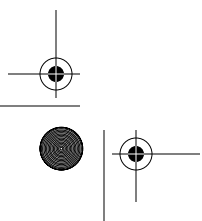
Earlier in this chapter, we saw that Flash 5–style `on()` and `onClipEvent()` handlers cannot be changed or removed during movie playback. Furthermore, `onClipEvent()` event handlers cannot be attached to the main movie timeline (the `_root`) of any movie.

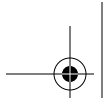
In order to work around these limitations, we can—in the case of `onClipEvent(enterFrame)` and the movie clip mouse and key events—use empty movie clips to simulate dynamic event-handler removal and alteration. Empty movie clips even let us simulate `onClipEvent()` handlers for the main timeline. Follow these steps to see how it works (note that the following technique applies to Flash 5 only, because event handler properties in Flash MX can be removed directly via `delete`):

1. Create an empty movie clip named `process`.
2. Place another empty clip, named `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) {  
    this._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()*.

To see this technique in action, download “Event Loop, Controllable” from the online Code Depot.

Event Handlers Applied

We’ll conclude our exploration of ActionScript events and event handlers with a couple of real-world examples. These are simple applications, but they give us a sense of how flexible event-based programming can be. For more event code samples, see the *Language Reference* coverage of the events associated with the objects listed at the beginning of this chapter.

Example 10-2 makes a clip named `box_mc` shrink and grow.

Example 10-2. Oscillating the size of a movie clip

```
// Set oscillation parameters
box_mc.sizeIncrement = 10;
box_mc.maxHeight = 200;
box_mc.minHeight = 20;

// Size the box with each passing frame
box_mc.onEnterFrame = function () {
    if (this._height >= this.maxHeight || this._height <= this.minHeight) {
        this.sizeIncrement = -this.sizeIncrement;
    }
    this._height += this.sizeIncrement;
    this._width += this.sizeIncrement;
}
```

Example 10-3 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 *onMouseDown()* and *onMouseUp()* handlers resize the custom pointer slightly to indicate mouseclicks. The code is placed inside the clip that acts as the custom pointer.

Example 10-3. A custom mouse pointer

```
Mouse.hide();

this.onMouseMove = function () {
    // When the mouse moves, position the current clip at
    // the mouse pointer's coordinates.
    this._x = _root._xmouse;
    this._y = _root._ymouse;
    updateAfterEvent(); // Refresh the screen for smooth movement.
}
```

Example 10-3. A custom mouse pointer (continued)

```

this.onMouseDown = function () {
    // When the mouse clicks down, shrink the clip.
    this._width *= .5;
    this._height *= .5;
    updateAfterEvent();
}

this.onMouseUp = function () {
    // When the mouse clicks down, return the clip to its previous size.
    this._width *= 2;
    this._height *= 2;
    updateAfterEvent();
}

```

Example 10-4 creates a movie clip with button behaviors from ActionScript. It creates the clip on-the-fly with *MovieClip.createEmptyMovieClip()*, then draws a square in the clip using the *MovieClip* drawing methods, and finally assigns the clip button handlers to create on, off, down, and up states. For more information on the drawing API, see “Drawing in a Movie Clip at Runtime” in Chapter 13.

Example 10-4. A movie clip button from scratch

```

// Create the clip
this.createEmptyMovieClip("optionButton_mc", 1);

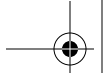
// Draw a 16-pixel dark red square from (-8,-8) to (8,8)
optionButton_mc.lineStyle(undefined);
optionButton_mc.moveTo(-8,-8);
optionButton_mc.beginFill(0x990000);
optionButton_mc.lineTo(8, -8);
optionButton_mc.lineTo(8, 8);
optionButton_mc.lineTo(-8, 8);
optionButton_mc.lineTo(-8, -8);
optionButton_mc.endFill();

// Turn the clip bright red when the mouse is over it
optionButton_mc.onRollOver = function () {
    var c = new Color(this);
    c.setRGB(0xFF0000);
}

// Turn the clip bright red when the mouse is off of it
optionButton_mc.onRollOut = function () {
    var c = new Color(this);
    c.setRGB(0x990000);
}

// Shrink the clip by 20% when it is pressed
optionButton_mc.onPress = function () {
    this._xscale = this._yscale = 80;
}

```



Example 10-4. A movie clip button from scratch (continued)

```
// Restore the clip to its original size when it is released  
optionButton_mc.onRelease = function () {  
    this._xscale = this._yscale = 100;  
}
```

Onward!

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

