,ch08.15995 Page 148 Monday, April 16, 2001 1:51 PM

8

Loop Statements

In the previous chapter, we learned that a *conditional* causes a statement block to be executed once if the value of its test expression is `true`. A *loop*, on the other hand, causes a statement block to be executed repeatedly, for as long as its test expression remains `true`.

Loops come in a variety of tasty flavors: *while*, *do-while*, *for*, and *for-in*. The first three types have very similar effects, but with varying syntax. The last type of loop, *for-in*, is a specialized kind of loop used with objects. We'll start our exploration of loops with the *while* statement, the easiest kind of loop to understand.

The while Loop

Structurally, a *while* statement is constructed much like an *if* statement: a main statement encloses a block of substatements that are executed only when a given condition is `true`:

```
while (condition) {
  substatements
}
```

If the condition is `true`, ***`substatements`*** are executed. But unlike the *if* statement, when the last substatement is finished, execution begins anew at the beginning of the *while* statement (that is the interpreter "loops" back to the beginning of the *while* statement). The second pass through the *while* statement works just like the first: the condition is evaluated, and if it is still `true`, ***`substatements`*** are executed again. This process continues until ***`condition`*** becomes `false`, at which point execution continues with any statements that follow the *while* statement in the script.

Here's an example of a very simple loop:

```
var i = 3;
while (i < 5) {
  trace("x is less than 5");
}
```

The example reliably represents the correct syntax of a *while* loop but is most likely in error. To see why, let's follow along with the interpreter as it executes the example.

We start with the statement before the *while* statement, *var i = 3*, which sets the variable `i` to 3. Because the variable `i` is used in the test expression of the loop, this step is often called the loop *initialization*. Next, we begin executing the *while* statement by resolving the test expression: *i < 5*. Because `i` is 3, and 3 is less than 5, the value of the test expression is `true` so we execute the *trace()* statement in the loop.

With that done, it's time to restart the loop. Once again, we check the value of the test expression. The value of the variable `i` has not changed, so the test expression is still `true` and we execute the *trace()* statement again. At this point, we're done executing the loop body, so it's time to restart the loop. Guess what? The variable `i` still has not changed, so the test expression is still `true` and we must execute the *trace()* statement again, and again, and again, forever. Because the test expression always returns `true`, there's no way to exit the loop—we're trapped forever in an *infinite loop*, unable to execute any other statements that may come after the *while* statement. In ActionScript, an infinite loop causes an error, as we'll see later.

Our loop is infinite because it lacks an *update statement* that changes the value of the variable used in the test expression. An update statement typically causes the test expression to eventually yield `false`, which terminates the loop. Let's fix our infinite loop by adding an update statement:

```
var i = 3;
while (i < 5) {
  trace("x is less than 5");
  i++;
}
```

The update statement, *i++*, comes at the end of the loop body. When the interpreter goes through our loop, it executes the *trace()* statement as before, but it also executes the statement *i++*, which adds one to the variable `i`. With each iteration of the loop, the value of `i` increases. After the second iteration, `i`'s value is 5, so the test expression, *i < 5*, becomes `false`. The loop, therefore, safely ends.

Our loop's update statement performs a fundamental loop activity: it counts. The variable `i` (called a *counter*) runs through a predictable numeric sequence—perfect for methodical tasks such as duplicating movie clips or accessing the elements of an array. Here we duplicate the `square` movie clip five times without using a loop:

```
// Name each new clip sequentially and place it on its own level
duplicateMovieClip("square", "square1", 1);
duplicateMovieClip("square", "square2", 2);
duplicateMovieClip("square", "square3", 3);
duplicateMovieClip("square", "square4", 4);
duplicateMovieClip("square", "square5", 5);
```

And here we do it with a loop:

```
var i = 1;
while (i <= 5) {
  duplicateMovieClip("square", "square" + i, i);
  i++;
}
```

Imagine the difference if we were duplicating `square` 100 times!

Loops are marvelously useful for manipulating data, particularly data stored in arrays. Example 8-1 shows a loop that displays all the elements of an array to the Output window. Note that the first element is number 0, not number 1.

Example 8-1. Displaying an Array with a while Loop

```
var people = ["John", "Joyce", "Margaret", "Michael"];  // Create an array
var i = 0;
while (i < people.length) {
  trace("people element " + i + " is " + people[i]);
  i++;
}
```

The result in the Output window is:

```
people element 0 is John
people element 1 is Joyce
people element 2 is Margaret
people element 3 is Michael
```

Notice that the variable `i` is used both in the test expression and as the array index number, as is typical. Here we use `i` again as an argument for the *charAt()* function:

```
var city = "Toronto";
trace("The letters in the variable 'city' are ");
var i = 0;
while (i < city.length) {
  trace(city.charAt(i));
  i++;
}
```

The Output window shows:

```
The letters in the variable 'city' are:
T
o
r
o
n
t
o
```

Finally, instead of dissecting data, we use a loop to construct a sentence from a series of words stored in an array:

```
var words = ["Toronto", "is", "not", "the", "capital", "of", "Canada"];
var sentence;
var i = 0;
while (i < words.length) {
  sentence += words[i];       // Add the current word to the sentence.

  // If it's not the last word...
  if (i < words.length - 1) {
    sentence += " ";          // ...tack on a space.
  } else {
    sentence += ".";          // ...otherwise, end with a period.
  }
  i++;
}
trace(sentence);              // Displays: "Toronto is not the capital of Canada."
```

Nearly all loops involve some kind of counter (also sometimes called an *iterator* or *index variable*). Counters let us cycle sequentially through data. This is particularly convenient when we determine the counter's maximum limit using the `length` property of the array or string we want to manipulate, as we did in the preceding example.

It's also possible to create a loop whose end point doesn't depend on a counter. As long as the test expression of the loop eventually becomes `false`, the loop will end. Here, for example, we examine the level stack of the Flash Player to determine the first vacant level:

```
var i = 0;
while (typeof eval("_level" + i) == "movieclip") {
  i++;
}
trace("The first vacant level is " + i);

// Now load a movie into the vacant level, knowing it's free
loadMovie("myMovie.swf", i);
```

Loop Terminology

In the previous section we encountered several new terms. Let's look at these more formally, so that you'll understand them well when working with loops:

Initialization
: The statement or expression that defines one or more variables used in the test expression of a loop.

Test expression
: The condition that must be met in order for the substatements in the loop body to be executed. Often called a *condition* or *test*, or sometimes, *control*.

Update
: The statements that modify the variables used in the test expression before a subsequent test. A typical update statement increments or decrements the loop's counter.

Iteration
: One complete execution of the test expression and statements in the loop body. Sometimes referred to as one *loop* or one *pass*.

Nesting or nested loop
: A loop that contains another loop so that you can iterate through some sort of two-dimensional data. For example, you might loop through each row in a column for all the columns in a table. The outer or top-level loop would progress through the columns, and the inner loop would progress through the rows in each column.

Iterator or index variable
: A variable whose value increases or decreases with each iteration of a loop, usually used to count or sequence through some data. Loop iterators are often called *counters*. Iterators are conventionally named `i`, `j`, and `k` or sometimes `x`, `y`, and `z`. In a series of nested loops, `i` is usually the iterator of the top-level loop, `j` is the iterator of the first nested loop, `k` is the iterator of the second nested loop, and so on. You can use any variable name you like for clarity. For example, you can use `charNum` as the variable name to remind yourself that it indicates the current character in a string.

Loop body
: The block of statements that are executed when a loop's condition is met. The body may not be executed at all, or it may be executed thousands of times.

Loop header or loop control
: The portion of a loop that contains the loop statement keyword (*while*, *for*, *do-while*, or *for-in*) and the loop controls. The loop control varies with the type of

loop. In a *for* loop, the control comprises the initialization, the test, and the update; in a *while* loop, the control comprises simply the test expression.

Infinite loop
A loop that repeats forever because its test expression never yields the value `false`. Infinite loops cause an error in ActionScript as discussed later under "Maximum Number of Iterations."

The do-while Loop

As we saw earlier, a *while* statement allows the interpreter to execute a block of code repeatedly while a specified condition remains `true`. Due to a *while* loop's structure, its body will be skipped entirely if the loop's condition is not met the first time it is tested. A *do-while* statement lets us guarantee that a loop body will be executed at least once with minimal fuss. The body of a *do-while* loop *always* executes the first time through the loop. The *do-while* statement's syntax is somewhat like an inverted *while* statement:

```
do {
  substatements
} while (condition);
```

The keyword *do* begins the loop, followed by the *`substatements`* of the body. On the interpreter's first pass through the *do-while* statement, *`substatements`* are executed before *`condition`* is ever checked. At the end of the *`substatements`* block, if *`condition`* is `true`, the loop is begun anew and *`substatements`* are executed again. The loop executes repeatedly until *`condition`* is `false`, at which point the *do-while* statement ends. Note that a semicolon is required following the parentheses that contain the *`condition`*.

Obviously, *do-while* is handy when we want to perform a task at least once and perhaps subsequent times. In Example 8-2 we duplicate a series of twinkling-star movie clips from a clip called `starParent` and place them randomly on the Stage. Our galaxy will always contain at least one star, even if `numStars` is set to 0.

Example 8-2. Using a do-while Loop

```
var numStars = 5;
var i = 1;
do {
  // Duplicate the starParent clip
  duplicateMovieClip(starParent, "star" + i, i);

  // Place the duplicated clip randomly on Stage
  _root["star" + i]._x = Math.floor(Math.random() * 551);
  _root["star" + i]._y = Math.floor(Math.random() * 401);
} while (i++ < numStars);
```

Did you notice that we sneakily updated the variable `i` in the test expression? Remember from Chapter 5, *Operators*, that the post-increment operator both returns the value of its operand and also adds one to that operand. The increment operator is very convenient (and common) when working with loops.

The for Loop

A *for* loop is essentially synonymous with a *while* loop but is written with more compact syntax. Most notably, the loop header can contain both initialization and update statements in addition to the test expression.

Here's the syntax of the *for* loop:

```
for (initialization; condition; update) {
  substatements
}
```

The *for* loop places the key components of a loop tidily in the loop header, separated by semicolons. Before the first iteration of a *for* loop, the ***`initialization`*** statement is performed (once and only once). It is typically used to set the initial value of an iterator variable. As with other loops, if ***`condition`*** is `true`, ***`substatements`*** are executed. Otherwise, the loop ends. At the *end* of each loop iteration, the ***`update`*** statement is executed, before ***`condition`*** is tested again to see if the loop should continue. Here's a typical *for* loop that simply counts from 1 to 10:

```
for (var i = 1; i <= 10; i++) {
  trace("Now serving number " + i);
}
```

It's easier to understand how a *for* loop works when you see its equivalent constructed using the *while* loop syntax:

```
var i = 1;
while (i <= 10) {
  trace("Now serving number " + i);
  i++;
}
```

Once you're used to the *for* syntax, you'll find it saves space and allows for easy scanning of the loop's body and controls.

Multiple Iterators in for Loops

If we want to control more than one factor in a loop, we may optionally use more than one iterator variable. A *while* loop with multiple iterators may look like this:

```
var i = 1;
var j = 10;
while (i <= 10) {
```

,ch08.15995 Page 155 Monday, April 16, 2001 1:51 PM

```
  trace("Going up " + i);
  trace("Going down " + j);
  i++;
  j--;
}
```

The same effect can be achieved in a *for* statement using the comma operator:

```
for (var i = 1, j = 10; i <= 10; i++, j--) {
  trace("Going up " + i);
  trace("Going down " + j);
}
```

The for-in Loop

A *for-in* statement is a specialized loop used to list the properties of an object. New programmers may want to skip this section for now and return to it after reading Chapter 12, *Objects and Classes*.

Rather than repeating a series of statements until a given test expression yields the value `false`, a *for-in* loop iterates once for each property in the specified object. Therefore, *for-in* statements do not need an explicit update statement because the number of loop iterations is determined by the number of properties in the object being inspected. The syntax of a *for-in* loop looks like this:

```
for (var thisProp in object) {
  substatements;  // Statements typically use thisProp in some way
}
```

The *`substatements`* are executed once for each property of *`object`*; *`object`* is the name of any valid object; *`thisProp`* is any variable name or identifier name. During each loop iteration, the *`thisProp`* variable temporarily holds a string that is the name of the object property currently being enumerated. That string value can be used during each iteration to access and manipulate the current property. The simplest example of a *for-in* loop is a script that lists the properties of an object. Here we create an object and then itemize its properties with a *for-in* loop:

```
var ball = new Object();
ball.radius = 12;
ball.color = "red";
ball.style = "beach";

for (var prop in ball) {
  trace("ball has the property " + prop);
}
```

Because `prop` stores the names of the properties of `ball` as strings, we can use `prop` with the `[]` operator to retrieve the values of those properties, like this:

```
for (var prop in ball) {
  trace("ball." + prop + " is " + ball[prop]);
}
```

Retrieving property values with a *for-in* loop also provides a super way to detect the movie clips present on a timeline. For a demonstration of the *for-in* loop used as a movie clip detector, see Example 3-1.

Note that the properties of the object being inspected in a *for-in* loop are not enumerated in any predictable order. Also, *for-in* statements do not always list every property of an object. When the object is user-defined, all properties are enumerated, including any inherited properties. But some properties of built-in objects are skipped by the *for-in* statement. Methods of built-in objects, for example, are not enumerated by a *for-in* loop. If you want to use a *for-in* statement to manipulate the properties of a built-in object, first build a test loop to determine the object's accessible properties.

Input text fields without a default value are not enumerated by a *for-in* loop. Hence, form-validation code that detects empty text fields will not work properly unless those text fields are explicitly declared as normal variables in the timeline upon which they reside. See "Empty Text Fields and the for-in Statement" in Chapter 18.

The *for-in* statement can also be used to extract elements in an array, in which case it takes the form:

```
for (var thisElem in array) {
  substatements; // Statements typically use thisElem in some way
}
```

This example lists the elements of an array:

```
var myArr = [123, 234, 345, 456];
for (var elem in myArr) {
  trace(myArr[elem]);
}
```

Stopping a Loop Prematurely

In a simple loop, the test expression is the sole factor that determines when the loop stops. When the test expression of a simple loop yields `false`, the loop terminates. However, as loops become more complex, we may need to arbitrarily terminate a running loop regardless of the value of the test expression. To do so, we use the *break* and *continue* statements.

The break Statement

The *break* statement ends execution of the current loop. It has the modest syntax:

```
break
```

The only requirement is that *break* must appear within the body of a loop.

The *break* statement provides a way to halt a process that is no longer worth completing. For example, we might use a *for-in* loop to build a form-checking routine that cycles through the input-text variables on a timeline. If a blank input field is found, we alert the user that she hasn't filled in the form properly. We can abort the process by executing a *break* statement. Example 8-3 shows the code. Note that the example assumes the existence of a movie clip called `form` that contains a series of declared input variables named `input01`, `input02`, and so on.

Example 8-3. A Simple Form-Field Validator

```
for (var prop in form) {
  // If this property is one of our "input" text fields
  if (prop.indexOf("input") != -1) {
    // If the form entry is blank, abort the operation
    if (form[prop] == "") {
      displayMessage = "Please complete the entire form.";
      break;
    }
    // Any substatements following the break command are not reached
    // when the break is executed
  }
}
// Execution resumes here after the loop terminates whether
// due to the break command or the test condition becoming false
```

You can use the *break* statement to interrupt a loop that would otherwise be infinite. This allows you to perform, say, the statements in the first half of the code block without necessarily executing the statements following an *if (condition) break;* statement. The generic approach is shown in Example 8-4.

Example 8-4. Breaking out of an Infinite Loop

```
while (true) {
  // Initial statements go here
  if (condition) break;
  // Subsequent statements go here
}
```

The continue Statement

The *continue* statement is similar to the *break* statement in that it causes the current iteration of a loop to be aborted, but unlike *break*, it resumes the loop's execution with the next natural cycle. The syntax of the *continue* statement is simply:

```
continue
```

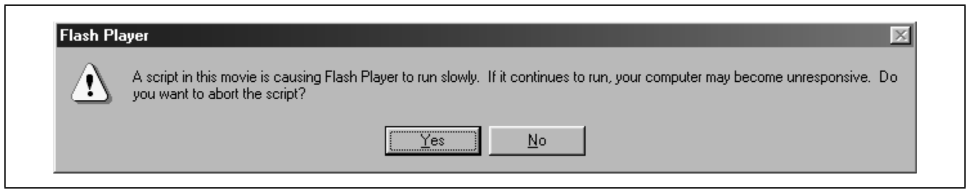
In all types of loops, the *continue* statement interrupts the current iteration of the loop body, but the resumption of the loop varies slightly depending on the type of loop statement. In a *while* loop and a *do-while* loop, the test expression is checked before the loop resumes. But in a *for* loop, the loop update is performed before the test expression is checked. And in a *for-in* loop, the next iteration begins with the next property of the object being inspected (if one exists).

Using the *continue* statement, we can make the execution of the body of a loop optional under specified circumstances. For example, here we move all the movie clip instances that aren't transparent to the left edge of the Stage, and we skip the loop body for transparent instances:

```
for (var prop in _root) {
  if (typeof _root[prop] == "movieclip") {
    if (_root[prop]._alpha < 100) {
      continue;
    }
    _root[prop]._x = 0;
  }
}
```

Maximum Number of Iterations

As noted earlier, loops are not allowed to execute forever in ActionScript. In the Flash 5 Player loops are limited to 15 seconds. The number of iterations that can be achieved in that time depends on what's inside the loop and the computer's speed. To be safe, you shouldn't create loops requiring more than even a few seconds to execute (which is eons in processing terms!). Most loops should take only milliseconds to finish. If a loop takes longer to complete (for example, because it's processing hundreds of strings while initializing a word-scramble game), it's worth rewriting the code using a timeline loop, as described in the next section. Timeline loops allow us to update the progress of a script's execution on screen and avoid the potential display of the error message shown in Figure 8-1.


Flash Player

A script in this movie is causing Flash Player to run slowly. If it continues to run, your computer may become unresponsive. Do you want to abort the script?

Figure 8-1. Bad loop! Down boy!

When a loop has run for more than 15 seconds in the Flash 5 Player, an alert box warns the user that a script in the movie is delaying the movie's playback. The user is offered the choice to either wait for the script to finish or to quit the script.

The Flash 4 player is even stricter—it allows only 200,000 iterations—after which all scripts are disabled without any warning.

Take special heed: the 15-second warning that users see does not mention that canceling a runaway script will actually cause all scripts in the movie to stop functioning! If a user selects "Yes" to stop a loop from continuing, all scripts in the movie are disabled.

Timeline and Clip Event Loops

All the loops we've looked at so far cause the interpreter to repeatedly execute blocks of code. Most of your loops will be of this "ActionScript-statement" type. But it's also sometimes desirable to create a *timeline loop* by looping Flash's playhead in the timeline. To do so, attach a series of statements to any frame; on the next frame, attach a *gotoAndPlay()* function whose destination is the previous frame. When the movie plays, the playhead will cycle between the two frames, causing the code on the first frame to be executed repeatedly.

We can make a simple timeline loop by following these steps:

1. Start a new Flash movie.
2. On frame 1, attach the following statement:

   ```
   trace("Hi there! Welcome to frame 1");
   ```
3. On frame 2, attach the following statements:

   ```
   trace("This is frame 2");
   gotoAndPlay(1);
   ```
4. Select Control → Test Movie.

When we test our movie, we see an endless stream of the following text:

```
Hi there! Welcome to frame 1
This is frame 2
Hi there! Welcome to frame 1
This is frame 2
```

Timeline loops can do two things ordinary loops cannot:

- They can execute a block of code an infinite number of times without causing an error.
- They can execute a block of code that requires a Stage update between loop iterations.

This second feature of timeline loops requires a little more explanation. When any frame's script is executed, the movie Stage is not updated visually until the end of the script. This means that traditional loop statements cannot be used to perform repetitive visual or audio tasks because the task results aren't rendered between each loop iteration. Repositioning a movie clip, for example, requires a Stage update, so we can't programmatically animate a movie clip with a normal loop statement.

You might assume that the following code would visually slide the `ball` movie clip horizontally across the Stage:

```
for (var i = 0; i < 50; i++) {
  ball._x += 10;
}
```

Conceptually, the loop statement has the right approach—it repetitively updates the position of `ball` by small amounts, which should give the illusion of movement. However, in practice, the `ball` doesn't move each time the `_x` position of `ball` is changed because the Stage isn't updated. Instead, we'd see the `ball` suddenly jump 500 pixels to the right—10 pixels for each of the 50 loop iterations—after the script completes.

To allow the Stage to update after each execution of the `ball._x += 10;` statement, we can use a timeline loop like this:

```
// CODE ON FRAME 1
ball._x += 10;

// CODE ON FRAME 2
gotoAndPlay(1);
```

Because Flash updates the Stage between any two frames, the `ball` will appear to animate. But the timeline loop completely monopolizes the timeline it's on. While it's running, we can't play any normal content on that timeline. A better approach is to put our timeline loop into an empty, two-frame movie clip. We'll get the benefit of a Stage update between loop iterations without freezing a timeline we may need for other animation.

Creating an Empty-Clip Timeline Loop

The following steps show how to create an empty-clip timeline loop:

1. Start a new Flash movie.
2. Create a movie clip symbol named *ball* that contains a circle shape.
3. On the main Stage, rename layer *Layer 1* to *ball.*
4. On the *ball* layer, place an instance of the *ball* symbol.
5. Name the instance of the *ball* clip, `ball`.
6. Select Insert → New Symbol to create a blank movie clip symbol.
7. Name the clip symbol `process`.

8. On frame 1 of the `process` clip, attach the following code:

   ```
    _root.ball._x += 10;
   ```

9. On frame 2 of the process clip, add the following code:

   ```
   gotoAndPlay(1);
   ```

10. Return to the main movie timeline and create a layer called *scripts*.
11. On the *scripts* layer, place an instance of the `process` symbol.
12. Name the instance `processMoveBall`.
13. Select Control → Test Movie.

The `processMoveBall` instance will now move `ball` without interfering with the playback of the main timeline upon which `ball` resides.

Note that step 12 isn't mandatory, but it gives us more control over our loop. By giving our timeline-loop instance a name, we can stop and start our loop by starting and stopping the playback of the instance, like this:

```
processMoveBall.play();
processMoveBall.stop();
```

Note that in this example `processMoveBall` and `ball` must both exist on the main timeline for as long as the loop is supposed to work. If we wanted to make the code more portable, we could use a relative reference to our `ball` clip in `process`:

```
_parent.ball._x += 10;
```

And if we wanted to control our ball from any timeline, we'd use an absolute reference to `ball`:

```
_root.ball._x += 10;
```

Timeline loops can't loop on a single frame. That is, if we place a `gotoAndPlay(5)` function on frame 5 of a movie, the function will be ignored. The Player realizes that the playhead is already on frame 5 and simply does nothing.

You'll find the sample timeline loop and empty-clip loop *.fla* files in the online Code Depot.

Flash 5 Clip Event Loops

Timeline loops are effective but not necessarily elegant. In Flash 5, we can use an event handler on a movie clip to achieve the same results as a timeline loop but with more flexibility (just try to follow along with this example, or see Chapter 10, *Events and Event Handlers*, for details on movie clip event handlers).

When placed on a movie clip, an *enterFrame* event handler causes a block of code to execute every time a frame passes in a movie. We can use an *enterFrame* event handler on a single-frame empty clip to repetitively execute a block of code while allowing for a Stage update between each repetition (just as a timeline loop does). Follow these steps to try it out:

1. Follow steps 1 through 7 from the previous section.
2. On the main Stage, create a new layer called *scripts*.
3. On the *scripts* layer, place an instance of the `process` clip.
4. Select the `process` instance and attach the following code:

   ```
   onClipEvent(enterFrame) {
     _root.ball._x += 10;
   }
   ```

5. Select Control → Test Movie.

The `ball` instance should animate across the Stage.

Clip event loops free us from nesting our code inside a movie clip and don't require a two-frame loop, as timeline loops do. All the action of a clip event loop happens in a single event handler. However, the clip event example we just saw has a potential drawback: there's no way to programmatically start or stop the loop once it's started. The only way to stop the loop is to physically remove the `process` instance from the timeline with a blank keyframe.

To create an event loop that can be arbitrarily started and stopped, we have to create an empty clip that contains *another* empty clip that bears an event loop. We can then dynamically attach and remove the whole package whenever we want to start or stop our loop. A little convoluted, yes, but the results are quite flexible. Once again, follow the steps to try it out:

1. Follow steps 1 through 5 under "Creating an Empty-Clip Timeline Loop."
2. Select Insert → New Symbol twice to create two blank movie clip symbols.
3. Name one clip symbol `process` and the other `eventLoop`.
4. In the Library, select the `process` clip, then select Options → Linkage. The Symbol Linkage Properties dialog box appears.
5. Select Export This Symbol.
6. In the Identifier field, type **`processMoveBall`** and then click OK.
7. On frame 1 of the `process` clip, drag an instance of `eventLoop` onto the Stage.

8. Select the `eventLoop` instance, and attach the following code:

```
onClipEvent(enterFrame) {
  _parent._parent.ball._x += 10;
}
```

9. Return to the main movie timeline and attach the following code to frame 1:

```
attachMovie("processMoveBall", "processMoveBall", 5000);
```

10. Whenever you want to stop the event loop, issue the following statement:

```
_root.processMoveBall.removeMovieClip();
```

11. Select Control → Test Movie.

Once again, the `ball` instance should animate across the Stage, but this time we can start and stop it whenever we like by using the *attachMovie()* and *removeMovieClip()* functions shown in steps 9 and 10.

There are examples of regular and controllable clip event loops available from the online Code Depot.

Keeping event loops portable

Both of the clip event loops we just saw included a line of code that updates the position of the `ball` instance on the Stage. For example:

```
onClipEvent(enterFrame) {
  _parent._parent.ball._x += 10;  // Updates ball's position
}
```

Although this approach works, it's sloppy. By attaching meaningful code to our clip event, we've decentralized our code base, dispersing logic and behavior throughout our movie. In order to keep our code accessible during authoring and better structured for reuse, from within event loops, we should *only* call functions. So, instead of actually moving the `ball` clip in our example, we should call a function that moves the `ball` clip, like this:

```
onClipEvent(enterFrame) {
  _parent._parent.moveBall();
}
```

The user-defined function *moveBall()* would be defined on the same timeline we attach the `processMoveBall` clip to, like this:

```
function moveBall() {
  ball._x += 10;
}
```

We'll talk more about functions and code portability in Chapter 9, *Functions.*

If our application is simple, we may wish to forego our empty event-loop clip altogether. In some cases, we can quite legitimately attach an event loop directly to the clip being manipulated. In our `ball` example, we could avoid the need for separate empty clips by attaching the following code directly to the `ball` instance:

```
onClipEvent(enterFrame) {
  _x += 10;
}
```

This approach is ultraconvenient, but it doesn't scale very easily, and like our first example, it suffers from the inability to start and stop the loop.

Frame Rate's Effect on Timeline and Clip Event Loops

Because timeline and clip event loops iterate once per frame, their execution frequency is tied to the frame rate of a movie. If we're moving an object around the screen with a timeline or an event loop, an increase in frame rate can mean an increase in the speed of our animation.

When we programmed the movement of the `ball` clip in our earlier examples, we implicitly specified the velocity of the ball in relation to the frame rate. Our code says, "With each frame that passes, move `ball` ten pixels to the right":

```
_ball += 10;
```

The speed of `ball` is, hence, dependent on the frame rate. If our movie plays at 12 frames per second, then our `ball` clip moves 120 pixels per second. If our movie plays at 30 frames per second, our `ball` clip moves *300* pixels per second!

When timing scripted animations, it's tempting to calculate the distance to move an item in relation to the movie's frame rate. So, if a movie plays 20 frames per second, and we want an item to move 100 pixels per second, we're tempted to set the velocity of the object to 5 pixels per frame (5 pixels * 20 frames per second = 100 pixels per second). There are two serious flaws in this approach:

- By relying on the frame rate to determine the speed of an item, we make it painful to change the frame rate. If we change the frame rate, we have to recalculate our speed and edit our code accordingly.
- The Flash Player does not necessarily play movies back at the frame rate set in the Flash authoring tool; it often plays them slower. If the computer running the movie cannot render frames fast enough to keep up with the designated frame rate, the movie slows down. This slowdown can even vary depending on the system load; if other programs are running or if Flash is performing some processor-intensive task, the frame rate may drop for only a short period and then resume its normal pace.

  You can test this out yourself using the time-tracker tool available at:

  http://www.moock.org/webdesign/flash/actionscript/fps-speedometer

In some cases, an animation that plays back at slightly different speeds could be deemed acceptable. But when visual accuracy matters or when we're concerned with the responsiveness of an action game, it's much more appropriate to calculate the distance to move an object relative to elapsed time instead of the frame rate. Example 8-5 shows a quick-and-dirty sample of time-based animation (i.e., the `ball` speed is independent of the frame rate). The new movie would have three frames and two layers, one layer with the `ball` instance and the other with our scripts.

Example 8-5. Calculating Move Distances Based on Time, Not Frame Rate

```
// CODE ON FRAME 1
var distancePerSecond = 50;  // Pixels to move per second
var now = getTimer();        // The current time
var then = 0;                // The time when last frame was rendered
var elapsed;                 // Milliseconds between frame renders
var numSeconds;              // elapsed expressed in seconds
var moveAmount;              // Distance to move each frame

// CODE ON FRAME 2
then = now;
now = getTimer();
elapsed = now - then;
numSeconds = elapsed / 1000;
moveAmount = distancePerSecond * numSeconds;
ball._x += moveAmount;

// CODE ON FRAME 3
gotoAndPlay(2);
```

Note that our time-based movement might appear jerky if the frame rate suddenly changes. We could smooth things out by using an elapsed-time measurement that averages the time between a series of frames instead of just two.

Onward!

Well, we've come pretty far. The end of this chapter marks a milestone—it ends our examination of the ActionScript statements. That means we have variables, data, datatypes, expressions, operators, and statements under our belt. These components of the language are the foundation of all scripts. If you've read and understood everything up to this point, or at least most of it, you can officially claim that you're able to "speak" ActionScript.

In the remainder of Part I, *ActionScript Fundamentals*, we'll work on making our conversations more eloquent and our commands more powerful. We'll consider the advanced topics of how to make code portable, how to create events that initiate the execution of our code, how to manage complex data, and how to manipulate movie clips programmatically. These techniques will help us build more advanced applied examples.