31 CHAPTER

Control Structures

In This Chapter

How to branch script execution down multiple paths

Looping through ordered collections of data

How to use labels to assist in script navigation

You get up in the morning, go about your day's business, and then turn out the lights at night. That's not much different from what a program does from the time it starts to the time it ends. But along the way, both you and a program take lots of tiny steps, not all of which advance the "processing" in a straight line. At times, you have to control what's going on by making a decision or repeating tasks until the whole job is finished. Control structures are the facilities that make these tasks possible in JavaScript.

JavaScript control structures follow along the same lines of many programming languages, particularly with additions made in Navigator 4 and Internet Explorer 4 (for JavaScript 1.2). Basic decision-making and looping constructions satisfy the needs of just about all programming tasks.

If and If. . .Else Decisions

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

JavaScript programs frequently have to make decisions based on the current values of variables or object properties. Such decisions can have only two possible outcomes at a time. The factor that determines the path the program takes at these decision points is the truth of some statement. For example, when you enter a room of your home at night, the statement under test is something like "It is too dark to see without a light." If that statement is true, you switch on the light; if that statement is false, you carry on with your primary task.

Simple decisions

JavaScript syntax for this kind of simple decision always begins with the keyword `if`, followed by the condition to test, and then the statements that execute if the condition yields a true result. JavaScript uses no "then" keyword (as some other languages do); the keyword is implied by the way the various components of this construction are surrounded by parentheses and braces. The formal syntax is

```
if (condition) {
       statementsIfTrue
}
```

This means that if the condition is true, program execution takes a detour to execute statements inside the braces. No matter what happens, the program continues executing statements beyond the closing brace (}). If household navigation was part of the scripting language, the code would look something like this:

```
if (tooDark == true) {
       feel for light switch
       turn on light switch
}
```

If you're not used to C/C++, the double equals sign may have caught your eye. You learn more about this type of operator in the next chapter, but for now, know that this operator compares the equality of items on either side of it. In other words, the `condition` statement of an `if` construction must always yield a Boolean (true or false) value. Some object properties, you may recall, are Booleans, so you can stick a reference to that property into the `condition` statement by itself. Otherwise, the `condition` statement consists of two values separated by a comparison operator, such as == (equals) or != (does not equal).

Let's look at some real JavaScript. The following function receives a form object containing a text object called `entry`:

```
function notTooHigh(form) {
       if (parseInt(form.entry.value) > 100) {
          alert("Sorry, the value you entered is too high. Try
again.")
          return false
       }
       return true
}
```

The `condition` (in parentheses) tests the contents of the field against a hard-wired value of 100. If the entered value is larger than that, the function alerts you and returns a false value to the calling statement elsewhere in the script. But if the value is less than 100, all intervening code is skipped and the function returns true.

About *(condition)* expressions

A lot of condition testing for control structures compares a value against some very specific condition, such as a string being empty or a value being null. You can use a couple of shortcuts to take care of many circumstances. Table 31-1 details the values that evaluate to a true or false (or equivalent) to satisfy a control structure's *condition* expression.

Table 31-1
Condition value equivalents

True	***False***
Nonempty string	Empty string
Nonzero number	0
Nonnull value	Null
Object exists	Object doesn't exist
Property is defined	Undefined property

Instead of having to spell out an equivalency expression for a condition involving these kinds of values, you can simply supply the value to be tested. For example, if a variable named `myVal` might reach an `if` construction as null, an empty string, or a string value for further processing, you can use the following shortcut:

```
if (myVal) {
       do processing on myVal
}
```

All null or empty string conditions evaluate to false, so only the cases of `myVal` being a processable value get inside the `if` construction.

Complex decisions

The simple type of `if` construction described earlier is fine when the decision is to take a small detour before returning to the main path. But not all decisions — in programming or in life — are like that. To present two alternate paths in a JavaScript decision, you can add a component to the construction. The syntax is

```
if (condition) {
       statementsIfTrue
} else {
       statementsIfFalse
}
```

By appending the `else` keyword, you give the `if` construction a path to follow in case the `condition` evaluates to false. The *statementsIfTrue* and *statementsIfFalse* do not have to be balanced in any way: One statement could be one line of code, the other one hundred lines. But when either one of those branches completes, execution continues after the last closing brace. To demonstrate how this construction can come in handy, the following example is a script fragment that assigns the number of days in February based on whether the year is a leap year (using modulo arithmetic, explained in the next chapter, to determine if the year is evenly divisible by four):

```
var howMany = 0
var theYear = 1993
if (theYear % 4 == 0) {
       howMany = 29
} else {
       howMany = 28
}
```

Here is a case where execution has to follow only one of two possible paths to assign the number of days to the `howMany` variable. Had I not used the `else` portion, as in

```
var howMany = 0
var theYear = 1993
if (theYear % 4 == 0) {
       howMany = 29
}
howMany = 28
```

then the variable would always be set to 28, occasionally after momentarily being set to 29. The `else` construction is essential in this case.

Nesting if. . .else statements

Designing a complex decision process requires painstaking attention to the logic of the decisions your script must process and the statements that must execute for any given set of conditions. The need for many complex constructions disappears with the advent of Navigator 4's `switch` construction (described later in this chapter), but there may still be times when you must fashion complex decision behavior out of a series of nested `if...else` constructions. Without a JavaScript-aware text editor to help keep everything properly indented and properly terminated (with closing braces), you have to monitor the authoring process very carefully. Moreover, the error messages that JavaScript provides when a mistake occurs (see Chapter 45) may not point directly to the problem line, but only to the region of difficulty.

Note

Another important point to remember about nesting `if...else` statements in JavaScript before Version 1.2 is that the language does not provide a mechanism to break out of a nested part of the construction. For that reason, you have to construct complex assemblies with extreme care to make sure only the desired statement executes for each set of conditions. Extensive testing, of course, is also required (see Chapter 45).

To demonstrate a deeply nested set of `if...else` constructions, Listing 31-1 presents a simple user interface to a complex problem. A single text object asks the user to enter one of three letters, A, B, or C. The script behind that field processes a different message for each of the following conditions:

- ✦ The user enters no value.
- ✦ The user enters A.
- ✦ The user enters B.
- ✦ The user enters C.
- ✦ The user enters something entirely different.

What's with the formatting?

Indentation of the `if` construction and the further indentation of the statements executed on a true condition are not required by JavaScript. What you see here, however, is a convention that most JavaScript scripters follow. As you write the code in your text editor, you can use the Tab key to make each indentation level. The browser ignores these tab characters when loading the HTML documents containing your scripts. Until HTML editors are available that automatically format JavaScript listings for you, you have to manually make the listings readable and pretty.

Listing 31-1: **Deeply Nested if...else Constructions**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function testLetter(form){
       inpVal = form.entry.value  // assign to shorter variable name
       if (inpVal != "") {  // if entry is not empty then dive in...
          if (inpVal == "A") {  // Is it an "A"?
             alert("Thanks for the A.")
          } else if (inpVal == "B") {  // No.  Is it a "B"?
             alert("Thanks for the B.")
          } else if (inpVal == "C") {  // No.  Is it a "C"?
             alert("Thanks for the C.")
          } else {          // Nope.  None of the above
             alert("Sorry, wrong letter or case.")
          }
       } else {    // value was empty, so skipped all other stuff
above
          alert("You did not enter anything.")
       }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
Please enter A, B, or C:
<INPUT TYPE="text" NAME="entry" onChange="testLetter(this.form)">
</FORM>
</BODY>
</HTML>
```

Each condition executes only the statements that apply to that particular condition, even if it takes several queries to find out what the entry is. You do not need to break out of the nested construction because when a true response is found, the relevant statement executes, and no other statements occur in the execution path to run.

Even if you understand how to construct a hair-raising nested construction such as the one in Listing 31-1, the trickiest part is making sure that each left brace has a corresponding right brace. My technique for ensuring this pairing is to enter the right brace immediately after I type the left brace. I typically type the left brace, press Enter twice (once to open a free line for the next statement, once for the line that is to receive the right brace), tab, if necessary, to the same indentation as the line containing the left brace, and then type the right brace. Later, if I have to insert something indented, I just push down the right braces that I entered earlier. If I keep up this methodology throughout the process, the right braces appear at the desired indentation when I'm finished, even if they end up being dozens of lines below their original spot.

Conditional Expressions

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

While I'm showing you decision-making constructions in JavaScript, now is a good time to introduce a special type of expression that you can use in place of an `if. . . else` control structure for a common type of decision—the instance where you want to assign one of two values to a variable, depending on the outcome of some condition. The formal definition for the *conditional expression* is as follows:

```
variable = (condition) ? val1 : val2
```

This means that if the Boolean result of the `condition` statement is true, JavaScript assigns *val1* to the variable; otherwise, it assigns *val2* to the variable. Like other instances of `condition` expressions, this one must also be written inside parentheses. The question mark is key here, as is the colon separating the two possible values.

A conditional expression, though not particularly intuitive or easy to read inside code, is very compact. Compare an `if. . .else` version of an assignment decision that follows

```
var collectorStatus
if (CDCount > 500) {
       collectorStatus = "fanatic"
} else {
       collectorStatus = "normal"
}
```

with the conditional expression version:

```
var collectorStatus = (CDCount > 500) ? "fanatic" : "normal"
```

The latter saves a lot of code lines (although the internal processing is the same as an `if...else` construction). Of course, if your decision path contains more statements than just one setting the value of a variable, the `if...else` or `switch` construction is preferable. This shortcut, however, is a handy one to remember when you need to perform very binary actions, such as setting a true-or-false flag in a script.

Repeat (`for`) Loops

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✓	✓	✓	✓	✓	✓

As you have seen in numerous examples throughout previous chapters, the capability to cycle through every entry in an array or through every item of a form element is vital to many JavaScript scripts. Perhaps the most typical operation is inspecting a property of many similar items in search of a specific value, such as to determine which radio button in a group is selected. One JavaScript structure that allows for these repetitious excursions is the `for` loop, so named after the keyword that begins the structure. Two other structures, called the `while` loop and `do-while` loop, are covered in following sections.

The JavaScript `for` loop lets a script repeat a series of statements any number of times and includes an optional loop counter that can be used in the execution of the statements. The following is the formal syntax definition:

```
for ( [initial expression]; [condition]; [update expression]) {
        statements
}
```

The three statements inside the parentheses (parameters to the `for` statement) play a key role in the way a `for` loop executes.

An *initial expression* in a `for` loop is executed one time, the first time the `for` loop begins to run. The most common application of the initial expression is to assign a name and starting value to a loop counter variable. Thus, it's not uncommon to see a `var` statement that both declares a variable name and assigns an initial value (generally 0 or 1) to it. An example is

```
var i = 0
```

You can use any variable name, but conventional usage calls for the letter `i`, which is short for *index*. If you prefer the word `counter` or something else that reminds you of what the variable represents, that's fine, too. In any case, the important point to remember about this statement is that it executes once at the outset of the `for` loop.

The second statement is a *condition*, precisely like the `condition` statement you saw in `if` constructions earlier in this chapter. When a loop-counting variable is established in the initial expression, the condition statement usually defines how high the loop counter should go before the looping stops. Therefore, the most common statement here is one that compares the loop counter variable against some fixed value — is the loop counter less than the maximum allowed value? If the condition is false at the start, the body of the loop is not executed. But if the loop does execute, then every time execution comes back around to the top of the loop, JavaScript reevaluates the condition to determine the current result of the expression. If the loop counter increases with each loop, eventually the counter value goes beyond the value in the condition statement, causing the condition statement to yield a Boolean value of false. The instant that happens, execution drops out of the `for` loop entirely.

The final statement, the *update expression*, is executed at the end of each loop execution — after all statements nested inside the `for` construction have run. Again, the loop counter variable can be a factor here. If you want the counter value to increase by one the next time through the loop (called *incrementing* the value), you can use the JavaScript operator that makes that happen: the ++ operator appended to the variable name. That task is the reason for the appearance of all those `i++` symbols in the `for` loops you've seen already in this book. You're not limited to incrementing by one. You can increment by any multiplier you want or even drive a loop counter backwards by decrementing the value (`i--`).

Now let's take this knowledge and beef up the formal syntax definition with one that takes into account a typical loop-counting variable, `i`, and the common ways to use it:

```
//incrementing loop counter
for (var i = minValue; i <= maxValue; i++) {
       statements
}



//decrementing loop counter
for (var i = maxValue; i >= minValue; i--) {
       statements
}
```

In the top format, the variable, `i`, is initialized at the outset to a value equal to that of *minValue*. Variable `i` is immediately compared against *maxValue*. If `i` is less than or equal to *maxValue*, processing continues into the body of the loop. At the end of the loop, the update expression executes. In the top example, the value of `i` is incremented by 1. Therefore, if `i` is initialized as 0, then the first time through the loop, the `i` variable maintains that 0 value during the first execution of statements in the loop. The next time around, the variable has the value of 1.

As you may have noticed in the formal syntax definition, each of the parameters to the `for` statement is optional. For example, the statements that execute inside the loop may control the value of the loop counter based on data that gets manipulated in the process. Therefore, the update statement would probably interfere with the intended running of the loop. But I suggest that you use all three parameters until such time as you feel absolutely comfortable with their roles in the `for` loop. If you omit the condition statement, for instance, and you don't program a way for the loop to exit on its own, your script may end up in an infinite loop — which does your users no good.

Putting the loop counter to work

Despite its diminutive appearance, the `i` loop counter (or whatever name you want to give it) can be a powerful tool for working with data inside a repeat loop. For example, let’s examine a version of the classic JavaScript function that creates a new Navigator 2–compatible array while initializing entries to a value of 0:

```
// initialize array with n entries
function MakeArray(n) {
        this.length = n
        for (var i = 1; i <= n; i++) {
            this[i] = 0
        }
        return this
}
```

The loop counter, `i`, is initialized to a value of 1, because you want to create an array of empty entries (with value 0) starting with the one whose index value is 1 (the zeroth entry is assigned to the `length` property) in the previous line. In the condition statement, the loop continues to execute as long as the value of the counter is less than or equal to the number of entries being created (`n`). After each loop, the counter increments by 1. In the nested statement that executes within the loop, you use the value of the `i` variable to substitute for the index value of the assignment statement:

```
this[i] = 0
```

The first time the loop executes, the value expression evaluates to

```
this[1] = 0
```

The next time, the expression evaluates to

```
this[2] = 0
```

and so on, until all entries are created and stuffed with 0.

Recall the HTML page in Listing 29-4, where JavaScript extracted the names of planets from a previously constructed array (called `solarSys`). Here is the section of that listing that uses a `for` loop to extract the names and plug them into HTML specifications for a selection pop-up menu:

```
var page = "" // start assembling next part of page and form
page += "Select a planet to view its planetary data: "
page += "<SELECT NAME='planets'> "
// build popup list from array planet names
for (var i = 1; i <= solarSys.length; i++) {
        page += "<OPTION"        // OPTION tags
        if (i == 1) {             // pre-select first item in list
            page += " SELECTED"
        }
        page += ">" + solarSys[i].name
}
page += "</SELECT><P>"  // close selection item tag
document.write(page)    // lay out this part of the page
```

Notice one important point about the condition statement of the `for` loop: JavaScript extracts the `length` property from the array to be used as the loop counter boundary. From a code maintenance and stylistic point of view, this method is preferable to hard-wiring a value there. If someone discovers a new planet, you would make the addition to the array "database," whereas everything else in the code would adjust automatically to those changes, including creating a longer pop-up menu in this case. More to the point, though, is that you use the loop counter as an index value into the array to extract the `name` property for each entry in the array. You also use the counter to determine which is the first option, so you can take a short detour (via the `if` construction) to add the `SELECTED` tag to the first option's definition.

The utility of the loop counter in `for` loops often influences the way you design data structures, such as two-dimensional arrays (Chapter 29) for use as databases. Always keep the loop-counter mechanism in the back of your mind when you begin writing JavaScript script that relies on collections of data you embed in your documents (see Chapter 49 on the CD-ROM for examples).

Breaking out of a loop

Some loop constructions perform their job when a certain condition is met, at which point they have no further need to continue looping through the rest of the values in the loop counter's range. A common scenario for this is the cycling of a loop through an entire array in search of a single entry that matches some criterion. That criterion test is set up as an `if` construction inside the loop. If that criterion is met, you break out of the loop and let the script continue with the more meaningful processing of succeeding statements in the main flow. To accomplish that exit from the loop, use the `break` statement. The following schematic shows how the `break` statement may appear in a `for` loop:

```
for (var i = 0; i < array.length; i++) {
        if (array[i].property == magicValue) {
           statements that act on entry array[i]
           break
        }
}
```

The `break` statement tells JavaScript to bail out of the nearest `for` loop (in case you have nested `for` loops). Script execution then picks up immediately after the closing brace of the `for` statement. The variable value of `i` remains whatever it was at the time of the break, so you can use that variable later in the same script to access, say, that same array entry.

I use a construction like this back in Chapter 23's discussion of radio buttons. In Listing 23-8, I show a set of radio buttons whose `VALUE` attributes contain the full names of four members of the Three Stooges. A function uses a `for` loop to find out which button was selected and then uses that item's index value — after the `for` loop broke out of the loop — to alert the user. Listing 31-2 (not on the CD-ROM) shows the relevant function.

Listing 31-2: Breaking Out of a for Loop

```
function fullName(form) {
        for (var i = 0; i < form.stooges.length; i++) {
            if (form.stooges[i].checked) {
                break
            }
        }
        alert("You chose " + form.stooges[i].value + ".")
}
```

In this case, breaking out of the `for` loop was for more than mere efficiency: I used the value of the loop counter (frozen at the break point) to summon a different property outside of the `for` loop. In Navigator 4, the `break` statement assumes additional powers in cooperation with the new label feature of control structures. This subject is covered later in this chapter.

Directing loop traffic with continue

One other possibility in a `for` loop is that you may want to skip execution of the nested statements for just one condition. In other words, as the loop goes merrily on its way round and round, executing statements for each value of the loop counter, one value of that loop counter may exist for which you don't want those statements to execute. To accomplish this task, the nested statements need to include an `if` construction to test for the presence of the value to skip. When that value is reached, the `continue` command tells JavaScript to immediately skip the rest of the body, execute the update statement, and loop back around to the top of the loop (also skipping the condition statement part of the `for` loop's parameters).

To illustrate this construction, you create an artificial example that skips over execution when the counter variable is the superstitious person's unlucky 13:

```
for (var i = 1; i <= 20; i++) {
        if (i == 13) {
            continue
        }
        statements
}
```

In this example, the `statements` part of the loop executes for all values of `i` except 13. The `continue` statement forces execution to jump to the `i++` part of the loop structure, incrementing the value of `i` for the next time through the loop. In the case of nested `for` loops, a `continue` statement affects the `for` loop in whose immediate scope the `if` construction falls. The `continue` statement is enhanced in Navigator 4 in cooperation with the new label feature of control structures. This subject is covered later in this chapter.

The while Loop

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

The `for` loop is not the only kind of repeat loop you can construct in JavaScript. Another statement, called a `while` statement, sets up a loop in a slightly different format. Rather than providing a mechanism for modifying a loop counter, a `while` repeat loop assumes that your script statements will reach a condition that forcibly exits the repeat loop.

The basic syntax for a `while` loop is

```
while (condition) {
        statements
}
```

The `condition` statement is the same kind you saw in `if` constructions and in the middle parameter of the `for` loop. You introduce this kind of loop if some condition exists in your code (evaluates to true) before reaching this loop. The loop then performs some action that affects that condition repeatedly until that condition becomes false. At that point, the loop exits, and script execution continues with statements after the closing brace. If the statements inside the `while` loop do not affect the values being tested in `condition`, your script never exits, and it becomes stuck in an infinite loop.

Many loops can be rendered with either the `for` or `while` loops. In fact, Listing 31-3 (not on the CD-ROM) shows a `while` loop version of the `for` loop from Listing 31-2.

Listing 31-3: **A while Loop Version of Listing 31-2**

```
function fullName(form) {
        var i = 0
        while (!form.stooges[i].checked) {
           i++
        }
        alert("You chose " + form.stooges[i].value + ".")
}
```

One point you may notice is that if the condition of a `while` loop depends on the value of a loop counter, the scripter is responsible for initializing the counter prior to the `while` loop construction and managing its value within the `while` loop.

Should you need their powers, the `break` and `continue` control statements work inside `while` loops as they do in `for` loops. But because the two loop styles treat their loop counters and conditions differently, be extra careful (do lots of testing) when applying `break` and `continue` statements to both kinds of loops.

No hard-and-fast rules exist for which type of loop construction to use in a script. I generally use `while` loops only when the data or object I want to loop through is already a part of my script before the loop. In other words, by virtue of previous statements in the script, the values for any condition or loop counting (if needed) are already initialized. But if I need to cycle through a new object's properties or new array to extract some piece of data for use later in the script, I favor the `for` loop.

The do-while Loop

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

JavaScript 1.2 brings you one more looping construction, called the `do-while` loop. The formal syntax for this construction is as follows:

```
do {
       statements
} while  (condition)
```

An important difference distinguishes the `do-while` loop from the `while` loop. In the `do-while` loop, the statements in the construction always execute at least one time before the condition can be tested; in a `while` loop, the statements may never execute if the condition tested at the outset evaluates to false.

Use a `do-while` loop when you know for certain that the looped statements are free to run at least one time. If the condition may not be met the first time, use the other `while` loop. For many instances, the two constructions are interchangeable, although the `while` loop is compatible with all scriptable browsers.

Looping through Properties

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

JavaScript includes a variation of the `for` loop, called a `for...in` loop, which has special powers of extracting the names and values of any object property currently in the browser's memory. The syntax looks like this:

```
for (var in object) {
       statements
}
```

The *object* parameter is not the string name of an object, but the object itself. JavaScript delivers an object if you provide the name of the object as an unquoted string, such as `window` or `document`. Using the *var* variable, you can create a script that extracts and displays the range of properties for any given object.

Listing 31-4 shows a page containing a utility function you can insert into your HTML documents during the authoring and debugging stages of designing a JavaScript-enhanced page. In the example, the current window object is examined and its properties presented in the page.

Listing 31-4: **Property Inspector Function**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function showProps(obj,objName) {
        var result = ""
        for (var i in obj) {
           result += objName + "." + i + " = " + obj[i] + "<BR>"
        }
        return result
}
</SCRIPT>
</HEAD>
<BODY>
<B>Here are the properties of the current window:</B><P>
<SCRIPT LANGUAGE="JavaScript">
document.write(showProps(window, "window"))
</SCRIPT>
</BODY>
</HTML>
```

For debugging purposes, you can revise the function slightly to display the results in an alert dialog box. Replace the "`<BR>`" HTML tag with the "`\n`" carriage return character for a nicely formatted display in the alert dialog box. You can call this function from anywhere in your script, passing both the object reference and a string to it to help you identify the object when the results appear in an alert dialog box.

The with Statement

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

A `with` statement enables you to preface any number of statements by advising JavaScript on precisely which object your scripts will be talking about, so you don't have to use full, formal addresses to access properties or invoke methods of the same object. The formal syntax definition of the `with` statement is as follows:

```
with (object) {
       statements
}
```

The *object* reference is any valid object currently in the browser's memory. You saw an example of this in Chapter 27's discussion of the Math object. By embracing several Math-encrusted statements inside a `with` construction, your scripts can call the properties and methods without having to make the object part of every reference to those properties and methods.

An advantage of the `with` structure is that it can make heavily object-dependent statements easier to read and understand. Consider this long version of a function that requires multiple calls to the same object (but different properties):

```
function seeColor(form) {
       newColor =
(form.colorsList.options[form.colorsList.selectedIndex].text)
       return newColor
}
```

Using the `with` structure, you can shorten the long statement:

```
function seeColor(form) {
       with (form.colorsList) {
          newColor = (options[selectedIndex].text)
       }
       return newColor
}
```

When JavaScript encounters an otherwise unknown identifier inside a `with` statement, it tries to build a reference out of the object specified as its parameter and that unknown identifier. You cannot, however, nest `with` statements that build in each other (in the preceding example, you cannot have a `with (colorsList)` nested inside a `with (form)` statement and expect JavaScript to create a reference to options out of the two object names).

Labeled Statements

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

Crafting multiple nested loops can sometimes be difficult when the final condition your script is looking for is met deep inside the nests. The problem is that the `break` or `continue` statement by itself has scope only to the nearest loop level. Therefore, even if you break out of the inner loop, the outer loop(s) continue to execute. If all you want to do is exit the function when the condition is met, a simple `return` statement performs the same job as some other languages' `exit` command. But if you also need some further processing within that function after the condition is met, you're out of luck unless you fashion some rather complex condition testing in all the loop levels.

To the rescue comes a new facility in JavaScript 1.2 that lets you assign labels to blocks of JavaScript statements. Your `break` and `continue` statements can then alter their scope to apply to a labeled block other than the one containing the statement.

A label is any identifier (that is, name starting with a letter and containing no spaces or odd punctuation other than an underscore) followed by a colon preceding a logical block of executing statements, such as an `if...then` or loop construction. The formal syntax looks like the following:

```
label:
       statements
```

For a `break` or `continue` statement to apply itself to a labeled group, the label is added as a kind of parameter to each statement, as in

```
break label
continue label
```

To demonstrate how valuable this can be in the right situation, Listing 31-5 contains two versions of the same nested loop construction. The goal of each version is to loop through two different index variables until both values equal the target values set outside the loop. When those targets are met, the entire nested loop construction should break off and continue processing afterward. To help you visualize the processing that goes on during the execution of the loops, the scripts output intermediate and final results to the Java Console window. Show that window before clicking the button to trigger either script.

In the version without labels, when the targets are met, only the simple `break` statement is issued. This breaks the inner loop at that point, but the outer loop picks up on the next iteration. By the time the entire construction has ended, a lot of wasted processing has gone on. Moreover, the values of the counting variables max themselves out, because the loops execute in their entirety several times after the targets are met.

But in the labeled version, the inner loop breaks out of the outer loop when the targets are met. Far fewer lines of code are executed, and the loop counting variables are equal to the targets, as desired. Experiment with Listing 31-5 by changing the `break` statements to `continue` statements. Then closely analyze the two results in the Java Console window to see how the two versions behave.

Listing 31-5: **Labeled Statements**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
var targetA = 2
var targetB = 2
var range = 5
function run1() {
       java.lang.System.out.println("Running WITHOUT labeled break")
       for (var i = 0; i <= range; i++) {
          java.lang.System.out.println("Outer loop #" + i)
          for (var j = 0; j <= range; j++) {
             java.lang.System.out.println("Inner loop #" + j)
             if (i == targetA && j == targetB) {
                java.lang.System.out.println("BREAKING OUT OF INNER
LOOP")
                break
             }
          }
       }
       java.lang.System.out.println("After looping, i = " + i + ", j =
" + j + "\n")
}
function run2() {
       java.lang.System.out.println("Running WITH labeled break")
       outerLoop:
       for (var i = 0; i <= range; i++) {
          java.lang.System.out.println("Outer loop #" + i)
          innerLoop:
          for (var j = 0; j <= range; j++) {
             java.lang.System.out.println("Inner loop #" + j)
             if (i == targetA && j == targetB) {
                java.lang.System.out.println("BREAKING OUT OF OUTER
LOOP")
                break outerLoop
             }
          }
       }
       java.lang.System.out.println("After looping, i = " + i + ", j =
" + j + "\n")
}
</SCRIPT>
</HEAD>
<BODY>
```

(continued)

Listing 31-5 *(continued)*

```
<B>Breaking Out of Nested Labeled Loops</B>
<HR>
Look in the Java Console window for traces of these button scripts:<P>
<FORM>
<INPUT TYPE="button" VALUE="Execute WITHOUT Label" onClick="run1()"><P>
<INPUT TYPE="button" VALUE="Execute WITH Label" onClick="run2()"><P>
</FORM>
</BODY>
</HTML>
```

The switch Statement

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

In some circumstances, a binary — true or false — decision path is not enough to handle the processing in your script. An object property or variable value may contain any one of several values, and a separate execution path is required for each one. In the past, the way to accommodate this is with a series of `if...else` constructions. The more conditions you must test, the less efficient the processing is, because each condition must be tested. Moreover, the sequence of clauses and braces can get very confusing.

In JavaScript 1.2, a control structure in use by many languages comes to JavaScript. The implementation is similar to that of Java and C, using the `switch` and `case` keywords. The basic premise is that you can create any number of execution paths based on the value of some expression. At the beginning of the structure, you identify what that expression is, and then for each execution path, assign a label matching a particular value.

The formal syntax for the switch statement is

```
switch (expression) {
        case label1:
           statements
           [break]
        case label2:
           statements
           [break]
        ...
        [default:
           statements]
}
```

The *expression* parameter of the switch statement can evaluate to any string or number value. Labels are not surrounded by quotes, even if the labels represent string values of the expression. Notice that the break statements are optional. A break statement forces the switch expression to bypass all other checks of succeeding labels against the expression value. Another option is the default statement, which provides a catch-all execution path when the expression value does not match any of the case statement labels. If you'd rather not have any execution take place with a nonmatching expression value, omit the default part of the construction.

To demonstrate the syntax of a working switch statement, Listing 31-6 provides the skeleton of a larger application of this control structure. The page contains two separate arrays of different product categories. Each product has its name and price stored in its respective array. A select list displays the product names. When a user chooses a product, the script looks up the product name in the appropriate array and displays the price.

The trick behind this application is the values assigned to each product in the select list. While the displayed text is the product name, the VALUE attribute of each <OPTION> tag is the array category for the product. That value is the expression used to decide which branch to follow. Notice, too, that I assigned a label to the entire switch construction. The purpose of that is to let the deeply nested repeat loops for each case completely bail out of the switch construction (via a labeled break statement) whenever a match is made. You could extend this example to any number of product category arrays with additional case statements to match.

Listing 31-6: **The switch Construction in Action**

```
<HTML>
<HEAD>
<TITLE>Switch Statement and Labeled Break</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
// build two product arrays, simulating two database tables
function product(name, price) {
        this.name = name
        this.price = price
}
var ICs = new Array()
ICs[0] = new product("Septium 300MHz","$149")
ICs[1] = new product("Septium Pro 310MHz","$249")
ICs[2] = new product("Octium BFD 350MHz","$329")
var snacks = new Array
snacks[0] = new product("Rays Potato Chips","$1.79")
snacks[1] = new product("Cheezey-ettes","$1.59")
snacks[2] = new product("Tortilla Flats","$2.29")

// lookup in the 'table' associated with the product
function getPrice(selector) {
        var chipName = selector.options[selector.selectedIndex].text
        var outField = document.forms[0].cost
```

(continued)

Listing 31-6 *(continued)*

```
        master:
           switch(selector.options[selector.selectedIndex].value) {
              case "ICs":
                 for (var i = 0; i < ICs.length; i++) {
                        if (ICs[i].name == chipName) {
                               outField.value = ICs[i].price
                               break master
                        }
                 }
                 break
              case "snacks":
                 for (var i = 0; i < snacks.length; i++) {
                        if (snacks[i].name == chipName) {
                               outField.value = snacks[i].price
                               break master
                        }
                 }
                 break
              default:
                 outField.value = "Not Found"
           }
}
</SCRIPT>
</HEAD>
<BODY>
<B>Branching with the switch Statement</B>
<HR>
Select a chip for lookup in the chip price tables:<P>
<FORM>
Chip:<SELECT NAME="chips" onChange="getPrice(this)">
        <OPTION>
        <OPTION VALUE="ICs">Septium 300MHz
        <OPTION VALUE="ICs">Septium Pro 310MHz
        <OPTION VALUE="ICs">Octium BFD 350MHz
        <OPTION VALUE="snacks">Rays Potato Chips
        <OPTION VALUE="snacks">Cheezey-ettes
        <OPTION VALUE="snacks">Tortilla Flats
        <OPTION>Poker Chipset
</SELECT>&nbsp;
Price:<INPUT TYPE="text" NAME="cost" SIZE=10>
</FORM>
</BODY>
</HTML>
```

If you need this kind of functionality in your script but your audience is not all running level 4 or later browsers, see Listing 31-1 for ways to simulate the `switch` statement with `if...else` constructions.

✦ ✦ ✦