CHAPTER 7

Programming Fundamentals, Part II

In This Chapter

How control structures make decisions

How to define functions

Where to initialize variables efficiently

What those darned curly braces are all about

The basics of data arrays

Your tour of programming fundamentals continues in this chapter with subjects that have more intriguing possibilities. For example, I show you how programs make decisions and learn why a program must sometimes repeat statements over and over. Before you're finished here, you will learn how to use one of the most powerful information holders in the JavaScript language, the array.

Decisions and Loops

Every waking hour of every day you make decisions of some kind — most of the time you probably don't even realize it. Don't think so? Well, look at the number of decisions you make at the grocery store, from the moment you enter the store to the moment you clear the checkout aisle.

No sooner do you enter the store than you are faced with a decision: Based on the number and size of items you intend to buy, do you pick up a hand-carried basket or attempt to extricate a shopping cart from the metallic conga line near the front of the store? That key decision may have impact later when you see a special offer on an item that is too heavy to put into the hand basket. Now you head for the food aisles. Before entering an aisle, you compare the range of goods stocked in that aisle against items on your shopping list. If an item you need is likely to be found in this aisle, you turn into the aisle and start looking for the item; otherwise, you skip the aisle and move to the head of the next aisle.

Later you reach the produce section, in search of a juicy tomato. Standing in front of the bin of tomatoes, you begin inspecting them, one by one — picking one up, feeling its firmness, checking the color, looking for blemishes or signs of pests. You discard one, pick up another, and continue this process until one matches the criteria you have set in your mind for an acceptable morsel. Your last stop in the store is the checkout aisle. "Paper or plastic?" the clerk asks. One

more decision to make. What you choose impacts how you get the groceries from the car to the kitchen as well as your recycling habits.

In your trip to the store, you have gone through the same kinds of decisions and repetitions that your JavaScript programs will also encounter. If you understand these frameworks in real life, you can now look into the JavaScript equivalents and the syntax required to make them work.

Control Structures

In the vernacular of programming, the kinds of statements that make decisions and loop around to repeat themselves are called *control structures*. A control structure directs the execution flow through a sequence of script statements based on simple decisions and other factors.

An important part of a control structure is the *condition*. Just as you may have different routes to work depending on certain conditions (for example, nice weather, nighttime, attending a soccer game), so, too, does a program sometimes have to branch to an execution route if a certain condition exists. Each condition is an expression that evaluates to `true` or `false` — one of those Boolean data types mentioned in the previous lesson. The kinds of expressions commonly used for conditions are expressions that include a comparison operator. You do the same in real life: If it is true that the outdoor temperature is less than freezing, then you put on a coat before going outside. In programming, however, the comparisons are strictly comparisons of number or string values.

JavaScript provides several kinds of control structures for different programming situations. Three of the most common control structures you'll use are `if` constructions; `if...else` constructions; `for` loops.

Other common control structures you'll want to know, some of which were introduced only in Navigator 4 and Internet Explorer 4, are covered in great detail in Chapter 31. For this tutorial, however, you need to learn about the three common ones just mentioned.

If constructions

The simplest program decision is to follow a special branch or path of the program if a certain condition is true. Formal syntax for this construction follows. Items in italics get replaced in a real script with expressions and statements that fit the situation.

```
if (condition) {
        statement[s] if true
}
```

Don't worry about the curly braces yet. Instead, get a feel for the basic structure. The keyword, `if`, is a must. In parentheses goes an expression that evaluates to a Boolean value. This is the condition being tested as the program runs past this point. If the condition evaluates to `true`, then one or more statements inside the curly braces execute before continuing on with the next statement after the closing brace; if the condition evaluates to `false`, the statements inside the curly brace are ignored, and processing continues with the next statement after the closing brace.

The following example assumes that a variable, myAge, has had its value set earlier in the script (exactly how is not important for this example). The condition expression compares the value myAge against a numeric value of 18.

```
if (myAge < 18) {
        alert("Sorry, you cannot vote.")
}
```

The data type of the value inside myAge must be a number so that the proper comparison (via the < comparison operator) does the right thing. For all instances of myAge being less than 18, the nested statement inside the curly braces runs, displaying the alert to the user. After the user closes the alert dialog box, the script continues with whatever statement follows the entire if construction.

If . . . else constructions

Not all program decisions are as simple as the one shown for the if construction. Rather than specifying one detour for a given condition, you might want the program to follow either of two branches depending on that condition. It is a fine but important distinction. In the plain if construction, no special processing is performed when the condition evaluates to false. But if processing must follow one of two special paths, you need the if...else construction. The formal syntax definition for an if...else construction is as follows:

```
if (condition) {
        statement[s] if true
} else {
        statement[s] if false
}
```

Everything you know about the condition for an if construction applies here. The only difference is the else keyword, which provides an alternate path for execution to follow if the condition evaluates to false.

As an example, the following if...else construction determines how many days are in February for a given year. To simplify the demo, the condition simply tests whether the year divides equally by 4 (true testing for this value includes special treatment of end-of-century dates, but I'm ignoring that for now). The % operator symbol is called the *modulus* operator (covered in more detail in Chapter 32). The result of an operation with this operator yields the remainder of division of the two values. If the remainder is zero, it means the first value divides evenly by the second.

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

The important point to see from the example is that by the end of the if...else construction, the febDays variable is set to either 28 or 29. No other value is possible. For years evenly divisible by 4, the first nested statement runs;

for all other cases, the second statement runs. Processing then picks up with the next statement after the `if...else` construction.

About Repeat Loops

Repeat loops in real life generally mean repeating a series of steps until some condition is met, allowing you to break out of that loop. Such was the case earlier in this chapter when you looked through a bushel of tomatoes for the one that came closest to your ideal tomato. The same could be said for driving around the block in a crowded neighborhood until a parking space opens up.

A repeat loop lets a script cycle through a sequence of statements until some condition is met. For example, a JavaScript data validation routine might look at every character that has been entered into a form text field to make sure that every character is a number. Or if you have a collection of data stored in a list, the loop can check whether an entered value is in that list. Once that condition is met, the script can then break out of the loop and continue with the next statement after the loop construction.

The most common repeat loop construction used in JavaScript is called the `for` loop. It gets its name from the keyword that begins the construction. A `for` loop is a powerful device, because you can set it up to keep track of the number of times the loop repeats itself. The formal syntax of the `for` loop is as follows:

```
for ([initial expression]; [condition]; [update expression]) {
        statement[s] inside loop
}
```

The square brackets mean that the item is optional, but until you get to know the `for` loop better, I recommend designing your loops to utilize all three items inside the parentheses. The *initial expression* portion usually sets the starting value of a counter. The *condition* — the same kind of condition you saw for `if` constructions — defines the condition that forces the loop to stop going around and around. Finally, the *update expression* is a statement that executes each time all of the statements nested inside the construction have completed running.

A common implementation initializes a counting variable, `i`, increments the value of `i` by one each time through the loop, and repeats the loop until the value of `i` exceeds some maximum value, as in the following:

```
for (var i = startValue; i <= maxValue; i++) {
        statement[s] inside loop
}
```

Placeholders `startValue` and `maxValue` represent any numeric values, including direct numbers or variables holding numbers. In the update expression is an operator you have not seen yet. The ++ operator adds 1 to the value of `i` each time the update expression runs at the end of the loop. If `startValue` is 1, the value of `i` is 1 the first time through the loop, 2 the second time through, and so on. Therefore, if `maxValue` is 10, the loop will repeat itself 10 times. Generally speaking, the statements inside the loop use the value of the counting variable in

their execution. Later in this lesson, I show how the variable can play a key role in the statements inside a loop. At the same time you will see how to break out of a loop prematurely and why you may need to do this in a script.

Functions

In Chapter 5, you saw a preview of the JavaScript function. A function is a definition of a set of deferred actions. Functions are invoked by event handlers or by statements elsewhere in the script. Whenever possible, good functions are designed to be reusable in other documents. They can become building blocks you can use over and over again.

If you have programmed before, you will see parallels between JavaScript functions and other languages' subroutines. But unlike some languages that distinguish between procedures (which carry out actions) and functions (which carry out actions and return values), only one classification of routine exists for JavaScript. A function is capable of returning a value to the statement that invoked it, but this is not a requirement. However, when a function does return a value, the calling statement treats the function call like any expression — plugging in the returned value right where the function call is made. I will show some examples in a moment.

Formal syntax for a function is as follows:

```
function functionName ( [parameter1]...[,parameterN] ) {
        statement[s]
}
```

Names you assign to functions have the same restrictions as names you assign HTML elements and variables. You should devise a name that succinctly describes what the function does. I tend to use multiword names with the interCap (internally capitalized) format that start with a verb, since functions are action items, even if they do nothing more than get or set a value.

Another practice to keep in mind as you start to create functions is to keep the focus of each function as narrow as possible. It is possible to generate functions literally hundreds of lines long. Such functions are usually difficult to maintain and debug. Chances are that the long function can be divided up into smaller, more tightly focused fragments.

Function parameters

In Chapter 5, you saw how an event handler invokes a function by calling the function by name. Any call to a function, including one that comes from another JavaScript statement, works the same way: the function name is followed by a set of parentheses.

Functions can also be defined so they receive parameter values from the calling statement. Listing 7-1 shows a simple document that has a button whose `onClick=` event handler calls a function while passing text data to the function. The text string in the event handler call is in a nested string — a set of single quotes inside the double quotes required for the entire event handler attribute.

Listing 7-1: **Calling a Function from an Event Handler**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function showMsg(msg) {
       alert("The button sent: " + msg)
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
       <INPUT TYPE="button" VALUE="Click Me" onClick="showMsg ('The
button has been clicked!')">
</FORM>
</BODY>
</HTML>
```

Parameters provide a mechanism for "handing off" a value from one statement to another by way of a function call. If no parameters occur in the function definition, both the function definition and call to the function have only empty sets of parentheses, as shown in Chapter 5, Listing 5-8.

When a function receives parameters, it assigns the incoming values to the variable names specified in the function definition's parentheses. Consider the following script segment:

```
function sayHiToFirst(a, b, c) {
       alert("Say hello, " + a)
}
sayHiToFirst("Gracie", "George", "Harry")
sayHiToFirst("Larry", "Moe", "Curly")
```

After the function is defined in the script, the next statement calls that very function, passing three strings as parameters. The function definition automatically assigns the strings to variables `a`, `b`, and `c`. Therefore, before the `alert()` statement inside the function ever runs, `a` evaluates to `"Gracie"`, `b` evaluates to `"George"`, and `c` evaluates to `"Harry"`. In the `alert()` statement, only the `a` value is used, and the alert reads

```
Say hello, Gracie
```

When the user closes the first alert, the next call to the function occurs. This time through, different values are passed to the function and assigned to `a`, `b`, and `c`. The alert dialog reads

```
Say hello, Larry
```

Unlike other variables that you define in your script, function parameters do not use the `var` keyword to initialize them. They are automatically initialized whenever the function is called.

Variable scope

Speaking of variables, it's time to distinguish between variables that are defined outside and those defined inside of functions. Variables defined outside of functions are called *global* variables; those defined inside functions are called *local* variables.

A global variable has a slightly different connotation in JavaScript than it has in most other languages. For a JavaScript script, the "globe" of a global variable is the current document loaded in a browser window or frame. Therefore, when you initialize a variable as a global variable, it means that all statements in the page, including those inside functions, have direct access to that variable value. Statements can retrieve and modify global variables from anywhere in the page. In programming terminology, this kind of variable is said to have global *scope*, because everything on the page can "see" it.

It is important to remember that the instant a page unloads itself, all global variables defined in that page are erased from memory. If you need a value to persist from one page to another, you must use other techniques to store that value (for example, as a global variable in a framesetting document, as described in Chapter 14, or in a cookie, as described in Chapter 16). While the `var` keyword is usually optional for initializing global variables, I strongly recommend you use it for *all* variable initializations to guard against future changes to the JavaScript language.

In contrast to the global variable, a local variable is defined inside a function. You already saw how parameter variables are defined inside functions (without `var` keyword initializations). But you can also define other variables with the `var` keyword (absolutely required for local variables). The scope of a local variable is only within the statements of the function. No other functions or statements outside of functions have access to a local variable.

Local scope allows for the reuse of variable names within a document. For most variables, I strongly discourage this practice because it leads to confusion and bugs that are difficult to track down. At the same time, it is convenient to reuse certain kinds of variables names, such as `for` loop counters. These are safe because they are always reinitialized with a starting value whenever a `for` loop starts. You cannot, however, nest a `for` loop inside another without specifying a different loop counting variable.

To demonstrate the structure and behavior of global and local variables — and show you why you shouldn't reuse most variable names inside a document — Listing 7-2 defines two global and two local variables. I intentionally use bad form by initializing a local variable that has the same name as a global variable.

Listing 7-2: **Variables Scope Demonstration**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
var aBoy = "Charlie Brown"      // global
var hisDog = "Snoopy"           // global
function sampleFunction() {
```

(continued)

Listing 7-2 *Continued*

```
        // using improper design to demonstrate a point
        var hisDog = "Gromit"    // local version of hisDog
        var output = hisDog + " does not belong to " + aBoy + ".<BR>"
        document.write(output)
}
</SCRIPT>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
sampleFunction()  // runs as document loads
document.write(hisDog + " belongs to " + aBoy + ".")
</SCRIPT>
</BODY>
</HTML>
```

When the page loads, the script in the Head portion initializes the two global variables and defines the function in memory. In the Body, another script begins by invoking the function. Inside the function a local variable is initialized with the same name as one of the global variables — `hisDog`. In JavaScript such a local initialization overrides the global variable for all statements inside the function. (But note that if the `var` keyword had been left off of the local initialization, the statement would have reassigned the value of the global version to `"Gromit"`.)

Another local variable, `output`, is merely a repository for accumulating the text that is to be written to the screen. It begins by evaluating the local version of the `hisDog` variable. Then it concatenates some hard-wired text (note the extra spaces at the edges of the string segment). Next comes the evaluated value of the `aBoy` global variable — any global not overridden by a local is available for use inside the function. Since the expression is accumulating HTML to be written to the page, it ends with a period and a `<BR>` tag. The final statement of the function writes the content to the page.

After the function completes its task, the next statement in the Body script writes another string to the page. Because this script statement is executing in global space (that is, not inside any function), it accesses only global variables — including those that were defined in another `<SCRIPT>` tag set in the document. By the time the complete page finishes loading, it contains the following text lines:

```
Gromit does not belong to Charlie Brown.
Snoopy belongs to Charlie Brown.
```

About Curly Braces

Despite the fact that you probably rarely — if ever — use curly braces ({ }) in your writing, there is no mystery to their usage in JavaScript (and many other languages). Curly braces enclose blocks of statements that belong together. While they do assist humans in reading scripts to know what's going on, the browser also relies on curly braces to know which statements belong together. Curly braces must always be used in matched pairs.

You use curly braces most commonly in function definitions and control structures. For example, in the function definition in Listing 7-2, curly braces enclose four statements that make up the function definition (including the comment line). The closing brace lets the browser know that whatever statement that may come next is a statement outside of the function definition.

Physical placement of curly braces is not critical (nor is the indentation style you see in the code I provide). The following function definitions are treated identically by scriptable browsers:

```
function sayHiToFirst(a, b, c) {
        alert("Say hello, " + a)
}

function sayHiToFirst(a, b, c)
{
        alert("Say hello, " + a)
}

function sayHiToFirst(a, b, c) {alert("Say hello, " + a)}
```

Throughout this book, I use the style shown in the first example because I find it makes lengthy and complex scripts easier to read, including scripts that have many levels of nested control structures.

Arrays

The JavaScript array is one of the most useful data constructions you have available to you. The structure of a basic array resembles that of a single-column spreadsheet. Each row of the column holds a distinct piece of data, and each row is numbered. Numbers assigned to rows are in strict numerical sequence, starting with zero as the first row (programmers always start counting with zero). This row number is called an *index*. To access an item in an array, you need to know the name of the array and the index for the row. Because index values start with zero, the count of items of the array (as determined by the array's `length` property) is always one more than the highest index value of the array. More advanced array concepts allow you to essentially create arrays with multiple columns (described in Chapter 29), but for this tutorial, I stay with the single column basic array.

Data elements inside JavaScript arrays can be any data type, including objects. And, unlike a lot of other programming languages, different rows of the same JavaScript array can contain different data types.

Creating an array

An array is stored in a variable, so when you create an array, you assign the new array object to the variable (yes, arrays are JavaScript objects, but they belong to the core JavaScript language, rather than the document object model). A special keyword — `new` — preceding a call to the JavaScript function that generates arrays creates space in memory for the array. An optional parameter to the `Array()` function lets you specify at the time of creation how many elements (rows) of data will eventually occupy the array. JavaScript is very forgiving about this, because you can change the size of an array at any time. Therefore, if you omit a parameter when generating a new array, no penalty is incurred.

To demonstrate the array creation process, I create an array that holds the names of the 50 States and the District of Columbia. The first task is to create that array and assign it to a variable of any name that helps me remember what this collection of data is about:

```
var USStates = new Array(51)
```

At this point, the `USStates` array is sitting in memory like a 51-row table with no data in it. To fill the rows, I must assign data to each row. Addressing each row of an array requires a special way of indicating the index value of the row: square brackets after the name of the array. The first row of the `USStates` array is addressed as

```
USStates[0]
```

And to assign the string name of the first state of the alphabet to that row, I use a simple assignment operator:

```
USStates[0] = "Alabama"
```

To fill in the rest of the rows, I include a statement for each row:

```
USStates[1] = "Alaska"
USStates[2] = "Arizona"
USStates[3] = "Arkansas"
...
USStates[50] = "Wyoming"
```

Therefore, if you want to include a table of information in a document from which a script can look up information without accessing the server, you include the data in the document in the form of an array creation sequence. When the statements run as the document loads, by the time the document has finished loading into the browser, the data collection array is built and ready to go. Despite what appears to be the potential for a lot of statements in a document for such a data collection, the amount of data that must download for typical array collections is small enough not to severely impact page loading, even for dial-up users at 28.8 Kbps.

Accessing array data

The array index is the key to accessing an array element. The name of the array and an index in square brackets evaluates to the content of that array location. For example, after the `USStates` array has been built, a script could display an alert with Alaska's name in it with the following statement:

```
alert("The largest state is " + USStates[1] + ".")
```

Just as you can retrieve data from an indexed array element, so can you change the element by reassigning a new value to any indexed element in the array.

Although I won't dwell on it in this tutorial, you can also use string names as index values instead of numbers. In essence, this allows you to create an array that has named labels for each row of the array — a definite convenience for certain circumstances. But whichever way you use to assign data to an array element the first time is the way you must access that element thereafter in the page's scripts.

Parallel arrays

Now I show you why the numeric index methodology works well in JavaScript. To help with the demonstration, I will generate another array that is parallel with the `USStates` array. This new array is also 51 elements long, and it contains the year in which the state in the corresponding row of `USStates` became a state. That array construction would look like the following:

```
var stateEntered = new Array(51)
stateEntered [0] = 1819
stateEntered [1] = 1959
stateEntered [2] = 1912
stateEntered [3] = 1836
...
stateEntered [50] = 1890
```

In the browsers memory, then, are two tables that you can visualize as looking like the model in Figure 7-1. I could build more arrays that are parallel to these for items such as the postal abbreviation and capital city. The important point is that the zeroth element in each of these tables applies to Alabama, the first state in the `USStates` array.

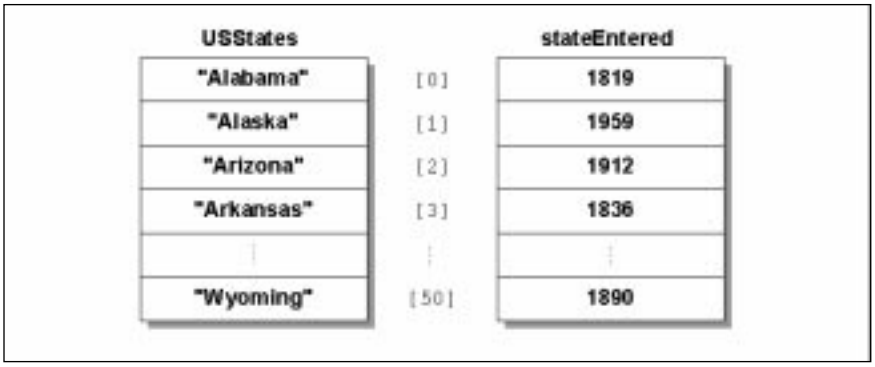

Figure 7-1: Visualization of two related parallel tables

If a Web page included these tables and a way for a user to look up the entry date for a given state, the page would need a way to look through all of the `USStates` entries to find the index value of the one that matches the user's entry. Then that index value could be applied to the `stateEntered` array to find the matching year.

For this demo, the page includes a text entry field in which the user types the name of the state to look up. In a real application, this methodology is fraught with peril, unless the script performs some error checking in case the user makes a mistake. But for now, I assume that the user will always type a valid state name (don't ever make this assumption in your Web site's pages). An event handler from either the text field or a clickable button will call a function that looks up the state name, fetches the corresponding entry year, and displays an alert message with the information. The function is as follows:

```
function getStateDate() {
        var selectedState = document.entryForm.entry.value
        for ( var i = 0; i < USStates.length; i++) {
            if (USStates[i] == selectedState) {
                break
            }
        }
        alert("That state entered the Union in " + stateEntered[i] +
"." )
}
```

In the first statement of the function, I grab the value of the text box and assign the value to a variable, `selectedState`. This is mostly for convenience, because I can use the shorter variable name later in the script. In fact, because the usage of that value is inside a `for` loop, the script is marginally more efficient, as the browser doesn't have to evaluate that long reference to the text field each time through the loop.

The key to this function is in the `for` loop. Here is where I combine the natural behavior of incrementing a loop counter with the index values assigned to the two arrays. Specifications for the loop indicate that the counter variable, `i`, is initialized with a value of zero. The loop is directed to continue as long as the value of `i` is less than the length of the `USStates` array. Remember that the length of an array is always one more than the index value of the last item. Therefore, the last time the loop will run is when `i` is 50, which is both less than the length of 51 and equal to the index value of the last element. Each time after the loop runs, the counter increments by one.

Nested inside the `for` loop is an `if` construction. The condition it is testing is the value of an element of the array against the value typed in by the user. Each time through the loop, the condition tests a different row of the array, starting with row zero. In other words, this `if` construction might be performed dozens of times before a match is found, but each time the value of `i` will be one larger than the previous try.

When a match is found, the statement nested inside the `if` construction runs. The `break` statement is designed to help control structures bail out if the program needs it. For this application, it is imperative that the `for` loop stop running when a match for the state name is found. When the `for` loop breaks, the value of the `i` counter is fixed at the row of the `USStates` array containing the entered state. I need that index value to find the corresponding entry in the other array. Even though the counting variable, `i`, is initialized in the `for` loop, it is still "alive" and in the scope of the function for all statements after the initialization. That's why I can use it to extract the value of the row of the `stateEntered` array in the final statement that displays the results in an alert message.

This application of a `for` loop and array indexes is a common one in JavaScript. Study the code carefully and be sure you understand how it works. This way of cycling through arrays plays a role not only in the kinds of arrays you create in your code, but also with the arrays that browsers generate for the document object model.

Document objects in arrays

If you look at the document object portion of the JavaScript Object Road Map in Appendix A, you will see that the properties of some objects are listed with square brackets after them. These are, indeed, the same kind of square brackets you saw above for array indexes. That's because when a document loads, the browser creates arrays of like objects in the document. For example, if your page includes two <FORM> tag sets, then two forms appear in the document. The browser maintains an array of form objects for that document. References to those forms would be

```
document.forms[0]
document.forms[1]
```

Index values for document objects are assigned according to the loading order of the objects. In the case of form objects, the order is dictated by the order of the <FORM> tags in the document. This indexed array syntax is another way to reference forms in an object reference. You can still use a form's name if you prefer — and I heartily recommend using object names wherever possible, because even if you change the physical order of the objects in your HTML, references that use names will still work without modification. But if your page contains only one form, you can use the reference types interchangeably, as in the following examples of equivalent references to a text field's value property in a form:

```
document.entryForm.entry.value
document.forms[0].entry.value
```

In examples throughout this book, you can see that I often use the array type of reference to simple forms in simple documents. But in my production pages, I almost always use named references.

Exercises

1. With your newly acquired knowledge of functions, event handlers, and control structures, use the script fragments from this chapter to complete the page that has the lookup table for all of the states and the years they were entered into the Union. If you do not have a reference book for the dates, then use different year numbers starting with 1800 for each entry. In the page, create a text entry field for the state and a button that triggers the lookup in the arrays.
2. Examine the following function definition. Can you spot any problems with the definition? If so, how would you fix the problems?

```
function format(ohmage) {
        var result
        if ohmage >= 10e6 {
                ohmage = ohmage / 10e5
                result = ohmage + " Mohms"
        } else {
                if (ohmage >= 10e3)
```

```
                        ohmage = ohmage / 10e2
                        result = ohmage + " Kohms"
                else
                        result = ohmage + " ohms"
}
alert(result)
```

3. Devise your own syntax for the scenario of looking for a ripe tomato at the grocery store, and write a `for` loop using that object and property syntax.
4. Modify Listing 7-2 so it does not reuse the `hisDog` variable inside the function.
5. Given the following table of data about several planets of our solar system, create a Web page that lets users enter a planet name and, at the click of a button, have the distance and diameter appear either in an alert box or (as extra credit) in separate fields of the page.

Planet	*Distance*	*Diameter*
Mercury	36 million miles	3,100 miles
Venus	67 million miles	7,700 miles
Earth	93 million miles	7,920 miles
Mars	141 million miles	4,200 miles

✦ ✦ ✦