CHAPTER 6

Programming Fundamentals, Part I

In This Chapter

What variables are and how to use them

How to evaluate expressions

How to convert data from one type to another

How to use basic operators

The tutorial breaks away from HTML and documents for a while, as you begin to learn programming fundamentals that apply to practically every scripting and programming language you will encounter. You'll start here learning about variables, expressions, data types, and operators — things that might sound scary if you haven't programmed before. Don't worry. With a little practice, you will become quite comfortable with these terms and concepts.

Working with Information

With rare exception, every JavaScript statement you write does something with a hunk of information — *data*. The chunk of data may be text information displayed on the screen by a JavaScript statement or the on/off setting of a radio button in a form. Each single piece of information in programming is also called a *value*. Outside of programming, the term *value* usually connotes a number of some kind; in the programming world, however, the term is not as restrictive. A string of letters is a value. A number is a value. The setting of a checkbox (the fact of whether it is checked or not) is a value.

In JavaScript, a value can be one of several types. Table 6-1 lists JavaScript's formal data types, with examples of the values you will see displayed from time to time.

A language that contains these few data types simplifies programming tasks, especially those involving what other languages consider to be incompatible types of numbers (integers versus real or floating-point values). In some definitions of syntax and parts of objects later in this book, I make specific reference to the type of value accepted in placeholders. When a string is required, any text inside a set of quotes suffices.

Table 6-1
JavaScript Value (Data) Types

Type	*Example*	*Description*
String	`"Howdy"`	A series of characters inside quote marks
Number	`4.5`	Any number not inside quote marks
Boolean	`true`	A logical true or false
Null	`null`	Completely devoid of any value
Object		All properties and methods belonging to the object or array
Function		A function definition

You will encounter situations, however, in which the value type may get in the way of a smooth script step. For example, if a user enters a number into a form's input field, JavaScript stores that number as a string value type. If the script is to perform some arithmetic on that number, the string must be converted to a number before the value can be applied to any math operations. You will see examples of this later in this lesson.

Variables

Cooking up a dish according to a recipe in the kitchen has one advantage over cooking up some data in a program. In the kitchen, you follow recipe steps and work with real things: carrots, milk, or a salmon fillet. A computer, on the other hand, follows a list of instructions to work with data. Even if the data represents something that looks real, such as the text entered into a form's input field, once the value gets into the program, you can no longer reach out and touch it.

In truth, the data that a program works with is merely a collection of bits (on and off states) in your computer's memory. More specifically, data in a JavaScript-enhanced Web page occupies parts of the computer's memory set aside for exclusive use by the browser software. In the olden days, programmers had to know the numeric address in memory (RAM) where a value was stored to retrieve a copy of it for, say, some addition. Although the innards of a program have that level of complexity, programming languages such as JavaScript shield you from it.

The most convenient way to work with data in a script is to first assign the data to what is called a *variable*. It's usually easier to think of a variable as a basket that holds information. How long the variable holds the information depends on a number of factors, but the instant a Web page clears the window (or frame), any variables it knows about are immediately discarded.

Creating a variable

You have a couple of ways to create a variable in JavaScript, but one will cover you properly in all cases. Use the `var` keyword, followed by the name you want to give that variable. Therefore, to *declare* a new variable called `myAge`, the JavaScript statement is

```
var myAge
```

That statement lets the browser know that you can use that variable later to hold information or to modify any of the data in that variable.

To assign a value to a variable, use one of the *assignment operators.* The most common one by far is the equals sign. If I want to assign a value to the `myAge` variable at the same time I declare it (a combined process known as *initializing* the variable), I use that operator in the same statement as the `var` keyword:

```
var myAge = 45
```

On the other hand, if I declare a variable in one statement and later want to assign a value to it, the sequence of statements is

```
var myAge
myAge = 45
```

Use the `var` keyword *only for declaration or initialization* — once for the life of any variable name in a document.

A JavaScript variable can hold any value type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. In fact, the value type of a variable could change during the execution of a program (this flexibility drives experienced programmers crazy because they're accustomed to assigning both a data type and a value to a variable).

Variable names

Choose the names you assign to variables with care. You'll often find scripts that use vague variable names, such as single letters. Other than a few specific times where using letters is a common practice (for example, using `i` as a counting variable in repeat loops in Chapter 7), I recommend using names that truly describe a variable's contents. This practice can help you follow the state of your data through a long series of statements or jumps, especially for complex scripts.

A number of restrictions help instill good practice in assigning names. First, you cannot use any reserved keyword as a variable name. That includes all keywords currently used by the language and all others held in reserve for future versions of JavaScript. The designers of JavaScript, however, cannot foresee every keyword that the language may need in the future. By using the kind of single words that currently appear in the list of reserved keywords (see Appendix B), you always run a risk of a future conflict.

To complicate matters, a variable name cannot contain space characters. Therefore, one-word variable names are fine. Should your description really benefit from more than one word, you can use one of two conventions to join multiple words as one. One convention is to place an underscore character between the words; the other is to start the combination word with a lowercase letter and capitalize the first letter of each subsequent word within the name — I refer to this as the interCap format. Both of the following examples are valid variable names:

```
my_age
myAge
```

My preference is for the second version. I find it easier to type as I write JavaScript code and easier to read later. In fact, because of the potential conflict with future keywords, using multiword combinations for variable names is a good idea. Multiword combinations are less likely to be part of the reserved word list.

Expressions and Evaluation

Another concept closely related to the value and variable is *expression evaluation* — perhaps the most important concept of learning how to program a computer.

We use expressions in our everyday language. Remember the theme song of *The Beverly Hillbillies*?

Then one day he was shootin' at some food
And up through the ground came a-bubblin' crude
Oil that is. Black gold. Texas tea.

At the end of the song, you find four quite different references ("crude," "oil," "black gold," and "Texas tea"). They all mean oil. They're all *expressions* for oil. Say any one of them, and other people know what you mean. In our minds, we *evaluate* those expressions to one thing: Oil.

In programming, a variable always evaluates to its contents, or value. For example, after assigning a value to a variable, such as

```
var myAge = 45
```

any time the variable is used in a statement, its value, 45, is automatically extracted from that variable and applied to whatever operation that statement is calling. Therefore, if you're 15 years my junior, I can assign a value to a variable representing your age based on the evaluated value of `myAge`:

```
var yourAge = myAge - 15
```

The variable, `yourAge`, evaluates to 30 the next time the script uses it. If the `myAge` value changes later in the script, the change has no link to the `yourAge` variable because `myAge` evaluated to 45 when it was used to assign a value to `yourAge`.

Expressions in script1.htm

You probably didn't recognize it at the time, but you saw how expression evaluation can come in handy in your first script of Chapter 3. Recall the second `document.write()` statement:

```
document.write(" of " + navigator.appName + ".")
```

The `document.write()` method (remember, JavaScript uses the term *method* to mean *command*) requires a parameter in parentheses: the text string to be displayed on the Web page. The parameter here consists of one expression that joins three distinct strings:

```
" of "
navigator.appName
"."
```

The plus symbol is one of JavaScript's ways of joining strings. Before JavaScript can display this line, it must perform some quick evaluations. The first evaluation is the value of the `navigator.appName` property. This property evaluates to a string of the name of your browser. With that expression safely evaluated to a string, JavaScript can finish the job of joining the three strings in the final evaluation. That evaluated string expression is what ultimately appears on the Web page.

Expressions and variables

As one more demonstration of the flexibility that expression evaluation offers, in this section I show you a slightly different route to the `document.write()` statement. Rather than join those strings as the direct parameter to the `document.write()` method, I could have gathered the strings together earlier in a variable and then applied the variable to the `document.write()` method. Here's how that method might have looked, as I simultaneously declare a new variable and assign it a value:

```
var textToWrite = " of " + navigator.appName + "."
document.write(textToWrite)
```

This method works because the variable, `textToWrite`, evaluates to the combined string. The `document.write()` method accepts that string value and does its display job. As you read a script or try to work through a bug, pay special attention to how each expression (variable, statement, object property) evaluates. I guarantee that as you're learning JavaScript (or any language), you will end up scratching your head from time to time because you haven't stopped to examine how expressions evaluate when a particular kind of value is required in a script.

Testing evaluation in Navigator

You can begin experimenting with the way JavaScript evaluates expressions with the help of a hidden feature of Navigator. (Note: if you are using the Windows 95/NT version of Navigator 4, make sure you have Version 4.03 or later to use this feature.) The feature is not available in Internet Explorer. Choose Open Location/Open Page from the File menu and enter the following:

```
javascript:
```

Navigator displays a special two-frame window. The bottom frame contains a field where you can type one-line expressions. Press Enter to view the results in the upper frame.

You can assign values to variables, test comparison operators, and even do math here. Following the variable examples earlier in this chapter, type each of the following statements into the type-in field and observe how each expression evaluates. Be sure to observe case-sensitivity in your entries.

```
var myAge = 45
myAge
var yourAge = myAge - 15
myAge - yourAge
myAge > yourAge
```

Figure 6-1 shows the results. To close this display, use the Navigator to open any HTML file or URL.

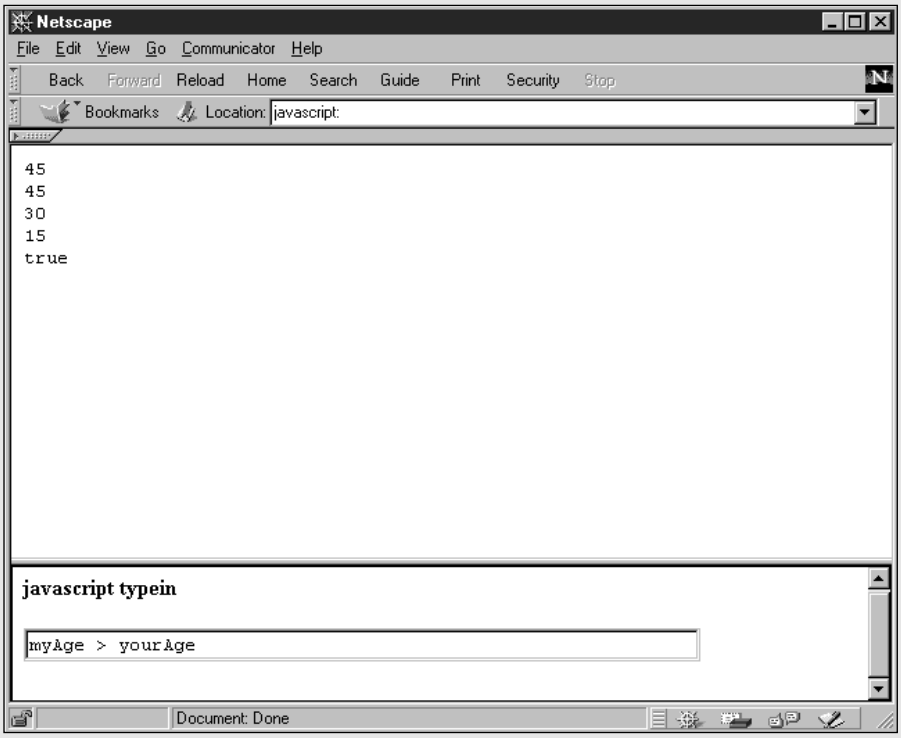

Figure 6-1: Evaluating expressions

Data Type Conversions

I mentioned earlier that the type of data in an expression can trip up some script operations if the expected components of the operation are not of the right type. JavaScript tries its best to perform internal conversions to head off such problems, but JavaScript cannot read your mind. If your intentions are different from the way JavaScript treats the values, you won't get the results you expected.

A case in point is adding numbers that may be in the form of text strings. In a simple arithmetic statement that adds two numbers together, the result is as you'd expect:

```
3 + 3         // result = 6
```

But if one of those numbers is a string, JavaScript leans toward converting the other value to a string, thus turning the plus sign's action from arithmetic addition to joining strings. Therefore, in the statement

```
3 + "3"     // result = "33"
```

the "string-ness" of the second value prevails over the entire operation. The first value is automatically converted to a string, and the result joins the two strings together. Try this yourself in the JavaScript type-in expression evaluator.

If I take this progression one step further, look what happens when another number is added to the statement:

```
3 + 3 + "3"    // result = "63"
```

This might seem totally illogical, but there is logic behind this result. The expression is evaluated from left to right. The first plus operation works on two numbers, yielding a value of 6. But as the 6 is about to be added to the "3," JavaScript lets the "string-ness" of the "3" rule. The 6 is converted to a string, and two string values are joined to yield 63.

Most of your concern about data types will be focused on performing math operations like the ones here. However, some object methods also require one or more parameters that must be particular data types. While JavaScript provides numerous ways to convert data from one type to another, it is appropriate at this stage of the tutorial to introduce you to the two most common data conversions: string to number and number to string.

Converting strings to numbers

As you saw in the last section, if a numeric value is stored as a string — as it is when entered into a form text field — your scripts will have difficulty applying that value to a math operation. The JavaScript language provides two built-in functions to convert string representations of numbers to true numbers: `parseInt()` and `parseFloat()`.

The difference between an integer and a floating-point number in JavaScript is that integers are always whole numbers, with no decimal point or numbers to the right of a decimal. A floating-point number, on the other hand, can have fractional values to the right of the decimal. By and large, JavaScript math operations don't differentiate between integers and floating-point numbers: A number is a number. The only time you need to be cognizant of the difference is when a method parameter requires an integer, because it can't handle fractional values. For example, parameters to the `moveTo()` method of a window require integer values of the coordinates where you want to position the window. That's because you can't move an object a fraction of a pixel on the screen.

To use either of these conversion functions, insert the string value you wish to convert as a parameter to the function. For example, look at the results of two different string values when passed through the `parseInt()` function:

```
parseInt("42")    // result = 42
parseInt("42.33") // result = 42
```

Even though the second expression passes the string version of a floating-point number to the function, the value returned by the function is an integer. No rounding of the value occurs here (although other math functions can help with that if necessary). The decimal and everything to its right are simply stripped off.

The `parseFloat()` function returns an integer if it can; otherwise, it returns a floating-point number:

```
parseFloat ("42")        // result = 42
parseFloat ("42.33")     // result = 42.33
```

Because these two conversion functions evaluate to their results, you simply insert the entire function wherever you need a string value converted to a string. Therefore, modifying an earlier example in which one of three values was a string, the complete expression can evaluate to the desired result:

```
3 + 3 + parseInt("3")    // result = 9
```

Converting numbers to string

You'll have less need for converting a number to its string equivalent than the other way around. As you saw in the previous section, JavaScript gravitates toward strings when faced with an expression containing mixed data types. Even so, it is good practice to perform data type conversions explicitly in your code to prevent any potential ambiguity. The simplest way to convert a number to a string is to take advantage of JavaScript's string tendencies in addition operations. By adding an empty string to a number, you convert the number to its string equivalent:

```
("" + 2500)              // result = "2500"
("" + 2500).length       // result = 4
```

In the second example, you can see the power of expression evaluation at work. The parentheses force the conversion of the number to a string. A string is a JavaScript object that has properties associated with it. One of those properties is the `length` property, which evaluates to the number of characters in the string. Therefore, the length of the string "2500" is 4. Note that the length value is a number, not a string.

Operators

You will use lots of *operators* in expressions. Earlier, you used the equal sign (=) as an assignment operator to assign a value to a variable. In the previous examples with strings, you used the plus symbol (+) to join (*concatenate*) two strings. An operator generally performs some kind of calculation (operation) or comparison with two values to reach a third value. In this lesson I briefly describe two categories of operators — arithmetic and comparison. Many more operators are covered in Chapter 32, but once you understand the basics here, the others will be easier to grasp when you're ready for them.

Arithmetic operators

The string concatenation operator doesn't know about words and spaces, so the programmer must make sure that any two strings to be joined have the proper word spacing as part of the strings, even if that means adding a space:

```
firstName = "John"
lastName = "Doe"
fullName = firstName + " " + lastName
```

JavaScript uses the same plus operator for arithmetic addition. When both values on either side of the plus sign are numbers, JavaScript knows to treat the expression as an arithmetic addition rather than a string concatenation. The standard math operators for addition, subtraction, multiplication, and division (+, -, *, /) are built into JavaScript.

Comparison operators

Another category of operator helps you compare values in scripts — whether two values are the same, for example. These kinds of comparisons return a value of the Boolean type — `true` or `false`. Table 6-2 lists the comparison operators. The operator that tests whether two items are equal consists of a pair of equals signs to distinguish it from the single equals sign assignment operator.

Table 6-2
JavaScript Comparison Operators

Symbol	*Description*
==	Equals
!=	Does not equal
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to

Where comparison operators come into greatest play is in the construction of scripts that make decisions as they run. A cook does this in the kitchen all the time: If the sauce is too watery, add a bit of flour. You see comparison operators in action in the next chapter.

Exercises

1. Which of the following are valid variable declarations or initializations? Explain why each one is or is not valid. If an item is not valid, how would you fix it so that it is?

   a. `my_name = "Cindy"`

   b. `var how many = 25`

   c. `var zipCode = document.form1.zip.value`

   d. `var 1address = document.nameForm.address1.value`

2. For each of the statements in the following sequence, write down how the `someVal` expression evaluates after the statement executes in JavaScript.

```
var someVal = 2
someVal = someVal + 2
someVal = someVal * 10
someVal = someVal + "20"
someVal = "Robert"
```

3. Name the two JavaScript functions that convert strings to numbers. How do you give the function a string value to convert to a number?

4. Type and load the HTML page and script shown in Listing 6-1. Enter a three-digit number into the top two fields and click the Add button. Examine the code and explain what is wrong with the script. How would you fix the script so the proper sum is displayed in the output field?

Listing 6-1: **What's Wrong with This Page?**

```
<HTML>
<HEAD>
<TITLE>Sum Maker</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
function addIt() {
       var value1 = document.adder.inputA.value
       var value2 = document.adder.inputB.value
       document.adder.output.value = value1 + value2
}
// -->
</SCRIPT>
</HEAD>

<BODY>
<FORM NAME="adder">
<INPUT TYPE="text" NAME="inputA" VALUE="0" SIZE=4><BR>
<INPUT TYPE="text" NAME="inputB" VALUE="0" SIZE=4>
<INPUT TYPE="button" VALUE="Add" onClick="addIt()">
<P>_____</P>
<INPUT TYPE="text" NAME="output" SIZE=6> <BR>
</FORM>
</BODY>
</HTML>
```

5. What does the term *concatenate* mean in the context of JavaScript programming?

✦ ✦ ✦