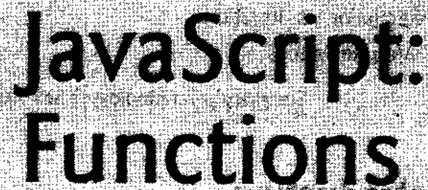


JavaScript: Functions

Form ever follows function.
—Louis Sullivan

E pluribus unum.
(One composed of many.)
—Virgil

O! call back yesterday, bid time return.
—William Shakespeare

Call me Ishmael.
—Herman Melville

When you call me that, smile.
—Owen Wister

OBJECTIVES

In this chapter you will learn:

- To construct programs modularly from small pieces called functions.
- To create new functions.
- How to pass information between functions.
- Simulation techniques that use random number generation.
- How the visibility of identifiers is limited to specific regions of programs.

Outline

9.1 Introduction
9.2 Program Modules in JavaScript
9.3 Programmer-Defined Functions
9.4 Function Definitions
9.5 Random Number Generation
9.6 Example: Game of Chance
9.7 Another Example: Random Image Generator
9.8 Scope Rules
9.9 JavaScript Global Functions
9.10 Recursion
9.11 Recursion vs. Iteration
9.12 Web Resources
Summary | Terminology | Self-Review Exercises | Exercises

9.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters of this book. Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or **modules**. This technique is called **divide and conquer**. This chapter describes many key features of JavaScript that facilitate the design, implementation, operation and maintenance of large scripts.

9.2 Program Modules in JavaScript

Modules in JavaScript are called **functions**. JavaScript programs are written by combining new functions that the programmer writes with "prepackaged" functions and objects available in JavaScript. The prepackaged functions that belong to JavaScript objects (such as `Math.pow` and `Math.round`, introduced previously) are called **methods**. The term method implies that the function belongs to a particular object. We refer to functions that belong to a particular JavaScript object as methods; all others are referred to as functions.

JavaScript provides several objects that have a rich collection of methods for performing common mathematical calculations, string manipulations, date and time manipulations, and manipulations of collections of data called arrays. These objects make your job easier, because they provide many of the capabilities programmers frequently need. Some common predefined objects of JavaScript and their methods are discussed in Chapter 10, JavaScript: Arrays, and Chapter 11, JavaScript: Objects.

Good Programming Practice 9.1

Familiarize yourself with the rich collection of objects and methods provided by JavaScript.

Software Engineering Observation 9.1

Avoid reinventing the wheel. Use existing JavaScript objects, methods and functions instead of writing new ones. This reduces script-development time and helps avoid introducing errors.

Portability Tip 9.1

Using the methods built into JavaScript objects helps make scripts more portable.

You can write functions to define specific tasks that may be used at many points in a script. These functions are referred to as **programmer-defined functions**. The actual statements defining the function are written only once and are hidden from other functions.

A function is **invoked** (i.e., made to perform its designated task) by a **function call**. The function call specifies the function name and provides information (as **arguments**) that the called function needs to perform its task. A common analogy for this structure is the hierarchical form of management. A boss (the **calling function**, or **caller**) asks a worker (the **called function**) to perform a task and **return** (i.e., report back) the results when the task is done. The boss function does not know how the worker function performs its designated tasks. The worker may call other worker functions—the boss will be unaware of this. We'll soon see how this "hiding" of implementation details promotes good software engineering. Figure 9.1 shows the `boss` function communicating with several worker functions in a hierarchical manner. Note that `worker1` acts as a "boss" function to `worker4` and `worker5`, and `worker4` and `worker5` report back to `worker1`.

Functions are invoked by writing the name of the function, followed by a left parenthesis, followed by a comma-separated list of zero or more arguments, followed by a right parenthesis. For example, a programmer desiring to convert a string stored in variable `inputValue` to a floating-point number and add it to variable `total` might write

```
total += parseFloat( inputValue );
```

When this statement executes, JavaScript function **`parseFloat`** converts the string in the `inputValue` variable to a floating-point value and adds that value to `total`. Variable `inputValue` is function `parseFloat`'s argument. Function `parseFloat` takes a string representation of a floating-point number as an argument and returns the corresponding floating-point numeric value. Function arguments may be constants, variables or expressions.

Methods are called in the same way, but require the name of the object to which the method belongs and a dot preceding the method name. For example, we've already seen the syntax `document.writeln("Hi there.");`. This statement calls the `document` object's `writeln` method to output the text.

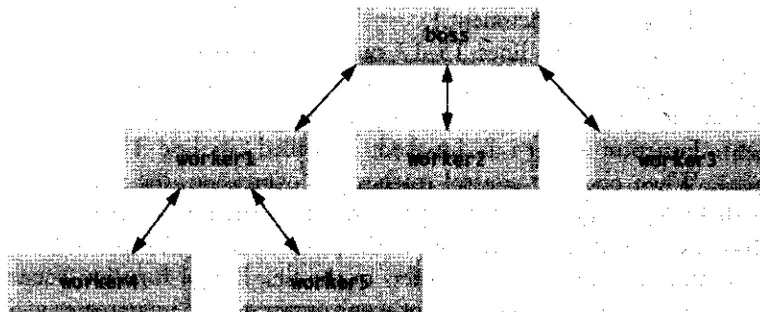


Fig. 9.1 | Hierarchical boss-function/worker-function relationship.

9.3 Programmer-Defined Functions

Functions allow you to modularize a program. All variables declared in function definitions are **local variables**—this means that they can be accessed only in the function in which they are defined. Most functions have a list of **parameters** that provide the means for communicating information between functions via function calls. A function's parameters are also considered to be local variables. When a function is called, the arguments in the function call are assigned to the corresponding parameters in the function definition.

There are several reasons for modularizing a program with functions. The divide-and-conquer approach makes program development more manageable. Another reason is **software reusability** (i.e., using existing functions as building blocks to create new programs). With good function naming and definition, programs can be created from standardized functions rather than built by using customized code. For example, we did not have to define how to convert strings to integers and floating-point numbers—JavaScript already provides function `parseInt` to convert a string to an integer and function `parseFloat` to convert a string to a floating-point number. A third reason is to avoid repeating code in a program. Code that is packaged as a function can be executed from several locations in a program by calling the function.

Software Engineering Observation 9.2

If a function's task cannot be expressed concisely, perhaps the function is performing too many different tasks. It is usually best to break such a function into several smaller functions.

9.4 Function Definitions

Each script we have presented thus far in the text has consisted of a series of statements and control structures in sequence. These scripts have been executed as the browser loads the web page and evaluates the `<head>` section of the page. We now consider how you can write your own customized functions and call them in a script.

Programmer-Defined Function `square`

Consider a script (Fig. 9.2) that uses a function `square` to calculate the squares of the integers from 1 to 10. [*Note:* We continue to show many examples in which the `body` element of the XHTML document is empty and the document is created directly by JavaScript. In later chapters, we show many examples in which JavaScripts interact with the elements in the `body` of a document.]

The `for` statement in lines 15–17 outputs XHTML that displays the results of squaring the integers from 1 to 10. Each iteration of the loop calculates the `square` of the current value of control variable `x` and outputs the result by writing a line in the XHTML document. Function `square` is invoked, or called, in line 17 with the expression `square(x)`. When program control reaches this expression, the program calls function `square` (defined in lines 23–26). The parentheses `()` represent the **function-call operator**, which has high precedence. At this point, the program makes a copy of the value of `x` (the argument) and program control transfers to the first line of function `square`. Function `square` receives the copy of the value of `x` and stores it in the parameter `y`. Then `square` calculates `y * y`. The result is passed back (returned) to the point in line 17 where `square` was invoked. Lines 16–17 concatenate `"The square of "`, the value of `x`, the string `" is "`,

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 9.2: SquareInt.html -->
 6  <!-- Programmer-defined function square. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>A Programmer-Defined square Function</title>
10        <script type = "text/javascript">
11           <!--
12           document.writeln( "<h1>Square the numbers from 1 to 10</h1>" );
13
14           // square the numbers from 1 to 10
15           for ( var x = 1; x <= 10; x++ )
16              document.writeln( "The square of " + x + " is " +
17                 square( x ) + "<br />" );
18
19           // The following square function definition is executed
20           // only when the function is explicitly called.
21
22           // square function definition
23           function square( y )
24           {
25              return y * y;
26           } // end function square
27           // -->
28        </script>
29     </head><body></body>
30  </html>
```

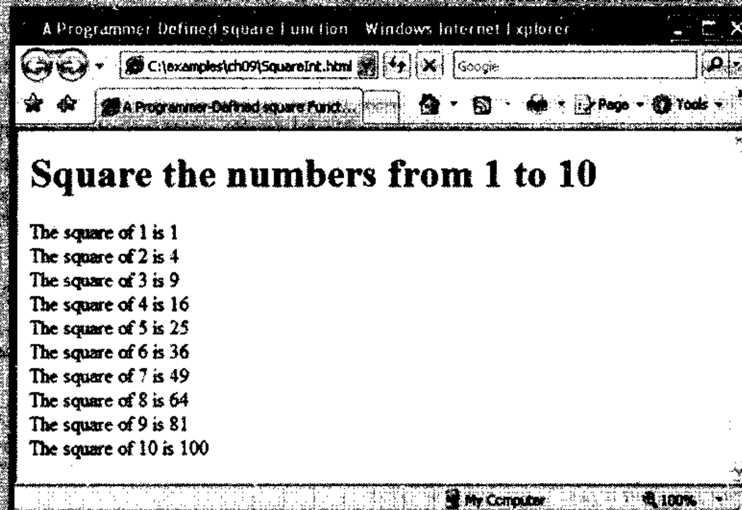


Fig. 9.2 | Programmer-defined function square.

the value returned by function square and a <br /> tag and write that line of text in the XHTML document. This process is repeated 10 times.

The definition of function square (lines 23–26) shows that square expects a single parameter y. Function square uses this name in its body to manipulate the value passed to square from line 17. The **return statement** in square passes the result of the calculation y * y back to the calling function. Note that JavaScript keyword var is not used to declare variables in the parameter list of a function.

Common Programming Error 9.1

Using the JavaScript var keyword to declare a variable in a function parameter list results in a JavaScript runtime error.

In this example, function square follows the rest of the script. When the for statement terminates, program control does *not* flow sequentially into function square. A function must be called explicitly for the code in its body to execute. Thus, when the for statement terminates in this example, the script terminates.

Good Programming Practice 9.2

Place a blank line between function definitions to separate the functions and enhance program readability.

Software Engineering Observation 9.3

Statements that are enclosed in the body of a function definition are not executed by the JavaScript interpreter unless the function is invoked explicitly.

The format of a function definition is

```
function function-name( parameter-list )
{
   declarations and statements
}
```

The *function-name* is any valid identifier. The *parameter-list* is a comma-separated list containing the names of the parameters received by the function when it is called (remember that the arguments in the function call are assigned to the corresponding parameter in the function definition). There should be one argument in the function call for each parameter in the function definition. If a function does not receive any values, the *parameter-list* is empty (i.e., the function name is followed by an empty set of parentheses). The *declarations* and *statements* in braces form the **function body**.

Common Programming Error 9.2

Forgetting to return a value from a function that is supposed to return a value is a logic error.

Common Programming Error 9.3

Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition results in a JavaScript runtime error.

Common Programming Error 9.4

Redefining a function parameter as a local variable in the function is a logic error.

Common Programming Error 9.5

Passing to a function an argument that is not compatible with the corresponding parameter's expected type is a logic error and may result in a JavaScript runtime error.

Good Programming Practice 9.3

Although it is not incorrect to do so, do not use the same name for an argument passed to a function and the corresponding parameter in the function definition. Using different names avoids ambiguity.

Software Engineering Observation 9.4

To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should express that task effectively. Such functions make programs easier to write, debug, maintain and modify.

Error-Prevention Tip 9.1

A small function that performs one task is easier to test and debug than a larger function that performs many tasks.

There are three ways to return control to the point at which a function was invoked. If the function does not return a result, control returns when the program reaches the function-ending right brace or by executing the statement

```
return;
```

If the function does return a result, the statement

```
return expression;
```

returns the value of *expression* to the caller. When a `return` statement is executed, control returns immediately to the point at which the function was invoked.

Programmer-Defined Function `maximum`

The script in our next example (Fig. 9.3) uses a programmer-defined function called `maximum` to determine and return the largest of three floating-point values.

The three floating-point values are input by the user via `prompt` dialogs (lines 12–14). Lines 16–18 use function `parseFloat` to convert the strings entered by the user to floating-point values. The statement in line 20 passes the three floating-point values to function `maximum` (defined in lines 28–31), which determines the largest floating-point value. This value is returned to line 20 by the `return` statement in function `maximum`. The value returned is assigned to variable `maxValue`. Lines 22–25 display the three floating-point values input by the user and the calculated `maxValue`.

Note the implementation of the function `maximum` (lines 28–31). The first line indicates that the function's name is `maximum` and that the function takes three parameters (`x`, `y` and `z`) to accomplish its task. Also, the body of the function contains the statement which returns the largest of the three floating-point values, using two calls to the `Math` object's `max` method. First, method `Math.max` is invoked with the values of variables `y` and `z` to determine the larger of the two values. Next, the value of variable `x` and the result of the first call to `Math.max` are passed to method `Math.max`. Finally, the result of the second call to `Math.max` is returned to the point at which `maximum` was invoked (i.e., line 20). Note

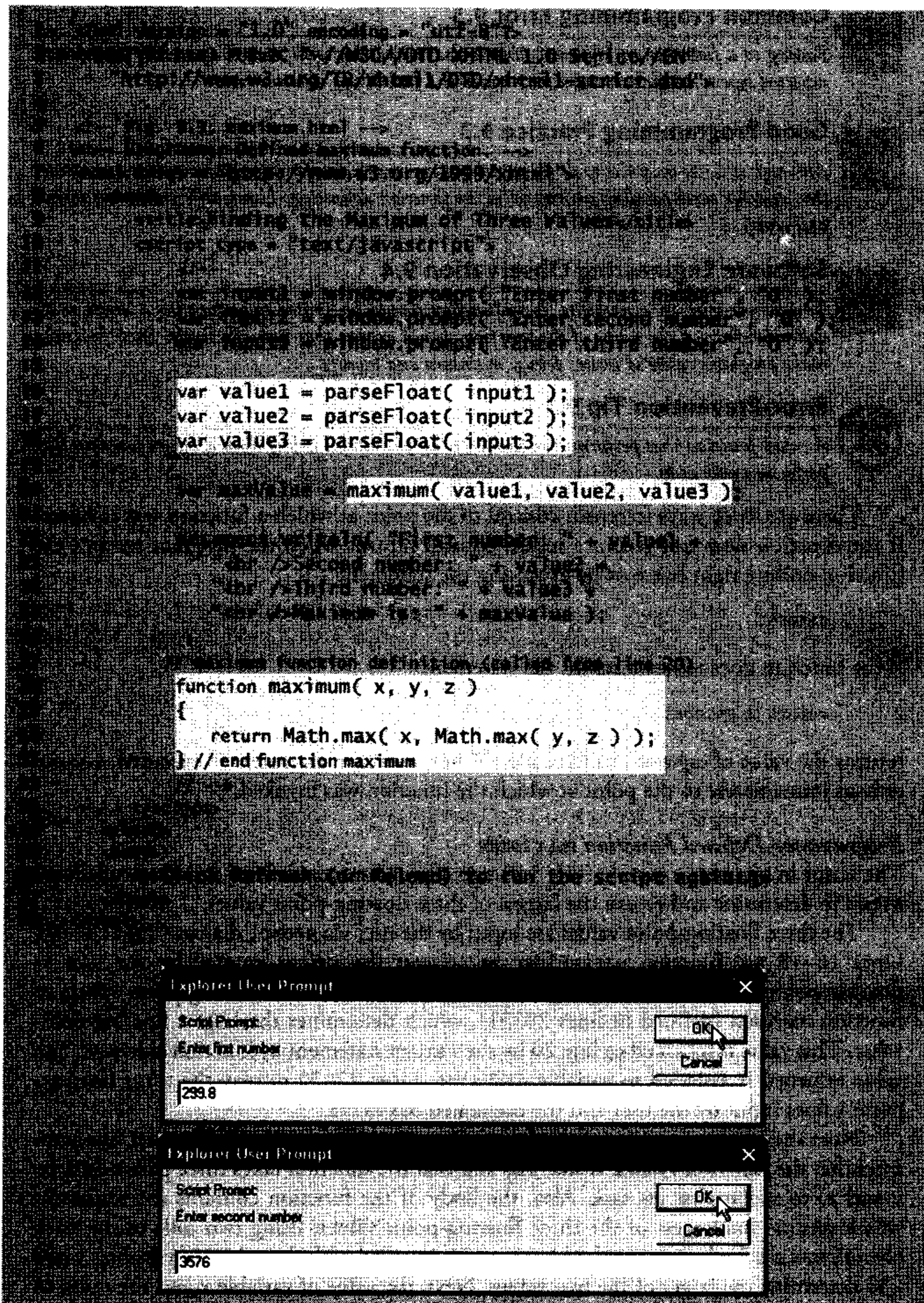


Fig. 9.3 | Programmer-defined `maximum` function. (Part 1 of 2.)

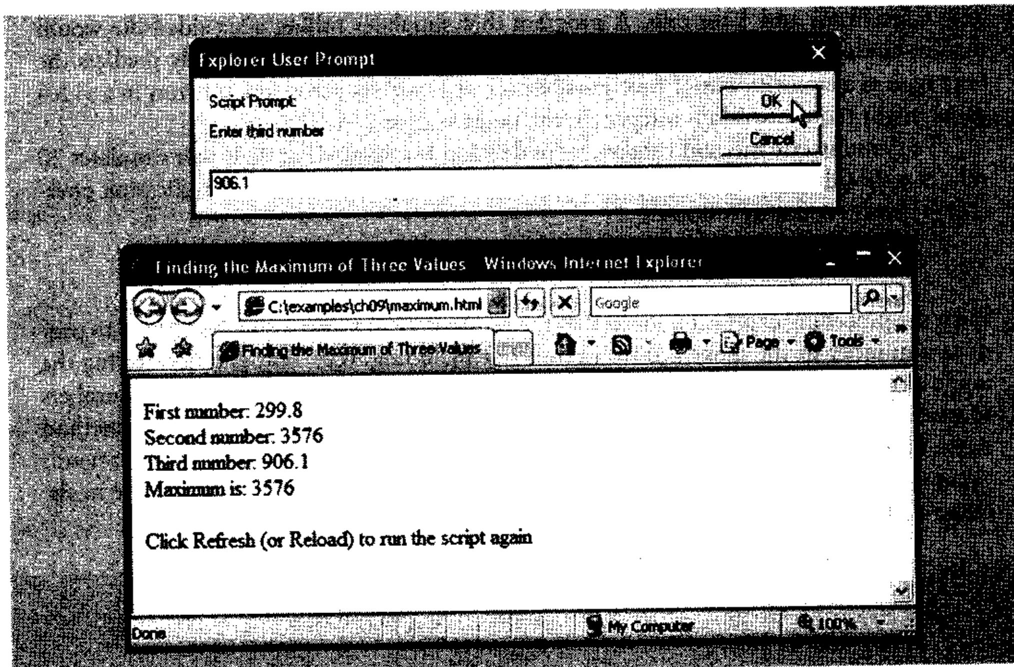


Fig. 9.3 | Programmer-defined `maximum` function. (Part 2 of 2.)

once again that the script terminates before sequentially reaching the definition of function `maximum`. The statement in the body of function `maximum` executes only when the function is invoked from line 20.

9.5 Random Number Generation

We now take a brief and, it is hoped, entertaining diversion into a popular programming application, namely simulation and game playing. In this section and the next, we develop a nicely structured game-playing program that includes multiple functions. The program uses most of the control structures we have studied.

There is something in the air of a gambling casino that invigorates people, from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It is the **element of chance**, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced through the `Math` object's **`random` method**. (Remember, we are calling `random` a method because it belongs to the `Math` object.)

Consider the following statement:

```
var randomValue = Math.random();
```

Method `random` generates a floating-point value from 0.0 up to, but not including, 1.0. If `random` truly produces values at random, then every value from 0.0 up to, but not including, 1.0 has an equal **chance** (or **probability**) of being chosen each time `random` is called.

The range of values produced directly by `random` is often different than what is needed in a specific application. For example, a program that simulates coin tossing might require

only 0 for heads and 1 for tails. A program that simulates rolling a six-sided die would require random integers in the range from 1 to 6. A program that randomly predicts the next type of spaceship, out of four possibilities, that will fly across the horizon in a video game might require random integers in the range 0–3 or 1–4.

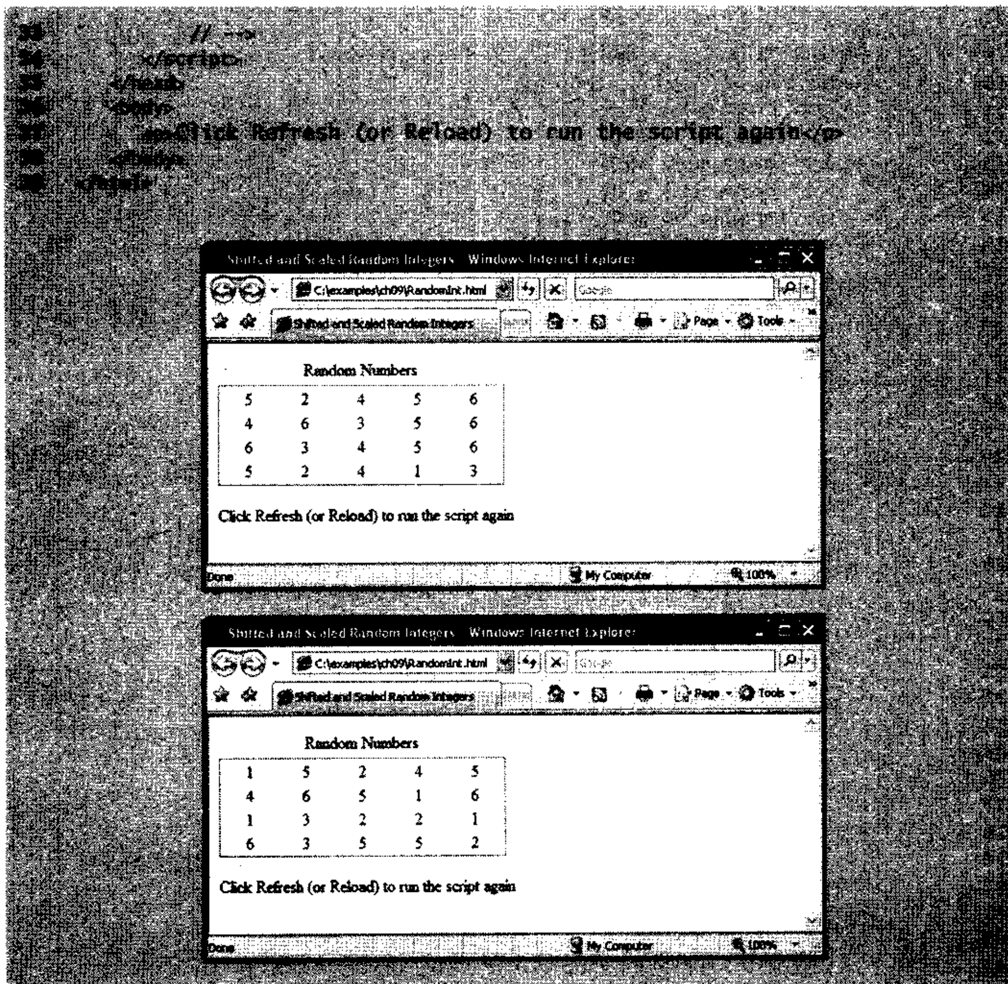
To demonstrate method `random`, let us develop a program (Fig. 9.4) that simulates 20 rolls of a six-sided die and displays the value of each roll. We use the multiplication operator (`*`) with `random` as follows:

```
Math.floor( 1 + Math.random() * 6 )
```

First, the preceding expression multiplies the result of a call to `Math.random()` by 6 to produce a number in the range 0.0 up to, but not including, 6.0. This is called scaling the range of the random numbers. Next, we add 1 to the result to shift the range of numbers to produce a number in the range 1.0 up to, but not including, 7.0. Finally, we use method **`Math.floor`** to *round* the result down to the closest integer not greater than the argument's value—for example, 1.75 is rounded to 1. Figure 9.4 confirms that the results are in the range 1 to 6.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 9.4: RandomInt.html -->
6  <!-- Random integers, shifting and scaling. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Shifted and Scaled Random Integers</title>
10       <style type = "text/css">
11          table { width: 50%;
12                  border: 1px solid gray;
13                  text-align: center }
14       </style>
15       <script type = "text/javascript">
16          <!--
17          var value;
18
19          document.writeln( "<table>" );
20          document.writeln( "<caption>Random Numbers</caption><tr>" );
21
22          for ( var i = 1; i <= 20; i++ )
23          {
24             value = Math.floor( 1 + Math.random() * 6 );
25             document.writeln( "<td>" + value + "</td>" );
26
27             // start a new table row every 5 entries
28             if ( i % 5 == 0 && i != 20 )
29                document.writeln( "</tr><tr>" );
30          } // end for
31
32          document.writeln( "</tr></table>" );
```

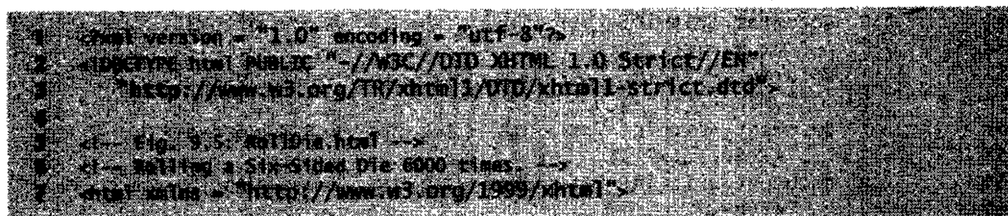
Fig. 9.4 | Random integers, shifting and scaling. (Part 1 of 2.)



```
33    // -->
34    </script>
35    </head>
36    <body>
37       <p>Click Refresh (or Reload) to run the script again</p>
38    </body>
39 </html>
```


Fig. 9.4 | Random integers, shifting and scaling. (Part 2 of 2.)

To show that these numbers occur with approximately equal likelihood, let us simulate 6000 rolls of a die with the program in Fig. 9.5. Each integer from 1 to 6 should appear approximately 1000 times. Use your browser's **Refresh** (or **Reload**) button to execute the script again.



```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml11-strict.dtd">
4
5  <!-- Fig. 9.5: RollDie.html -->
6  <!-- Rolling a Six-Sided Die 6000 times. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
```

Fig. 9.5 | Rolling a six-sided die 6000 times. (Part 1 of 3.)

```
      <title>Roll a Six-Sided Die 6000 Times</title>
      <script type = "text/javascript">

         var frequency1 = 0;
         var frequency2 = 0;
         var frequency3 = 0;
         var frequency4 = 0;
         var frequency5 = 0;
         var frequency6 = 0;
         var face;

         // roll die 6000 times and accumulate results
         for ( var roll = 1; roll <= 6000; roll++ )
         {
            face = Math.floor( 1 + Math.random() * 6 );

            switch ( face )
            {
               case 1:
                  ++frequency1;
                  break;
               case 2:
                  ++frequency2;
                  break;
               case 3:
                  ++frequency3;
                  break;
               case 4:
                  ++frequency4;
                  break;
               case 5:
                  ++frequency5;
                  break;
               case 6:
                  ++frequency6;
                  break;
            } // end switch
         } // end for

         document.writeln( "<table border = \"1\">" );
         document.writeln( "<thead><th>Face</th>" +
            "<th>Frequency</th></thead>" );
         document.writeln( "<tbody><tr><td>1</td><td>" +
            frequency1 + "</td></tr>" );
         document.writeln( "<tr><td>2</td><td>" + frequency2 +
            "</td></tr>" );
         document.writeln( "<tr><td>3</td><td>" + frequency3 +
            "</td></tr>" );
         document.writeln( "<tr><td>4</td><td>" + frequency4 +
            "</td></tr>" );
         document.writeln( "<tr><td>5</td><td>" + frequency5 +
            "</td></tr>" );
```

Fig. 9.5 | Rolling a six-sided die 6000 times. (Part 2 of 3.)

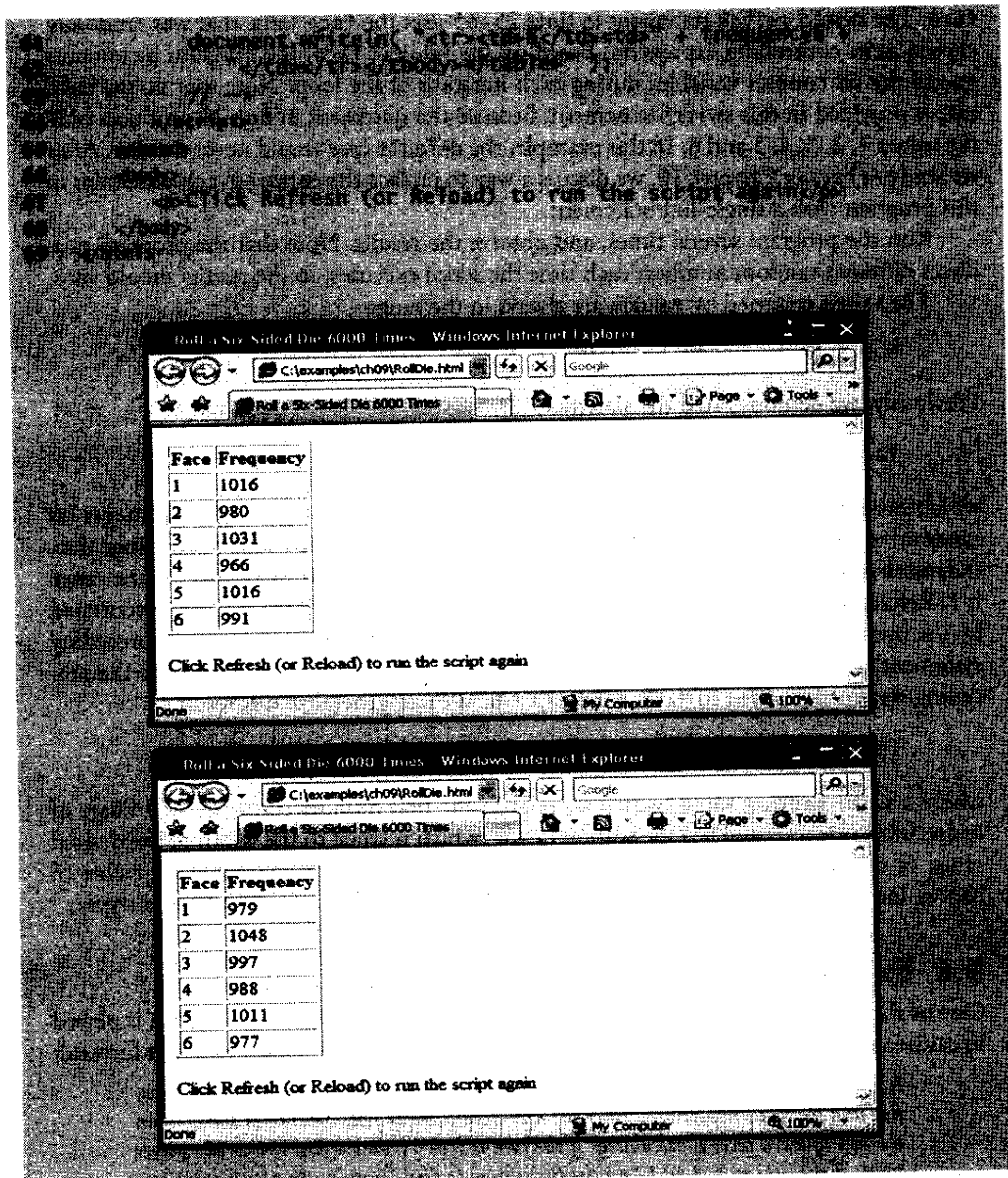


Fig. 9.5 | Rolling a six-sided die 6000 times. (Part 3 of 3.)

As the output of the program shows, we used `Math` method `random` and the scaling and shifting techniques of the previous example to simulate the rolling of a six-sided die. Note that we used nested control structures to determine the number of times each side of the six-sided die occurred. Lines 12–17 declare and initialize counter variables to keep track of the number of times each of the six die values appears. Line 18 declares a variable to store the face value of the die. The `for` statement in lines 21–46 iterates 6000 times. During each iteration of the loop, line 23 produces a value from 1 to 6, which is stored in

face. The nested switch statement in lines 25–45 uses the face value that was randomly chosen as its controlling expression. Based on the value of face, the program increments one of the six counter variables during each iteration of the loop. Note that no default case is provided in this switch statement, because the statement in line 23 produces only the values 1, 2, 3, 4, 5 and 6. In this example, the default case would never execute. After we study Arrays in Chapter 10, we discuss a way to replace the entire switch statement in this program with a single-line statement.

Run the program several times, and observe the results. Note that the program produces different random numbers each time the script executes, so the results should vary.

The values returned by random are always in the range

```
0.0 ≤ Math.random() < 1.0
```

Previously, we demonstrated the statement

```
face = Math.floor( 1 + Math.random() * 6 );
```

which simulates the rolling of a six-sided die. This statement always assigns an integer (at random) to variable face, in the range 1 ≤ face ≤ 6. Note that the width of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale random with the multiplication operator (6 in the preceding statement) and that the starting number of the range is equal to the number (1 in the preceding statement) added to Math.random() * 6. We can generalize this result as

```
face = Math.floor( a + Math.random() * b );
```

where a is the **shifting value** (which is equal to the first number in the desired range of consecutive integers) and b is the **scaling factor** (which is equal to the width of the desired range of consecutive integers). In this chapter's exercises, you'll see that it's possible to choose integers at random from sets of values other than ranges of consecutive integers.

9.6 Example: Game of Chance

One of the most popular games of chance is a dice game known as craps, which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> *A player rolls two dice. Each die has six faces. These faces contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll your point value). You lose by rolling a 7 before making the point.*

The script in Fig. 9.6 simulates the game of craps.

Note that the player must roll two dice on the first and all subsequent rolls. When you execute the script, click the **Roll Dice** button to play the game. A message below the **Roll Dice** button displays the status of the game after each roll.

Until now, all user interactions with scripts have been through either a prompt dialog (in which the user types an input value for the program) or an alert dialog (in which a

message is displayed to the user, and the user can click **OK** to dismiss the dialog). Although these dialogs are valid ways to receive input from a user and to display messages, they are fairly limited in their capabilities. A `prompt` dialog can obtain only one value at a time from the user, and a message dialog can display only one message.

More frequently, multiple inputs are received from the user at once via an XHTML form (such as one in which the user enters name and address information) or to display many pieces of data at once (e.g., the values of the dice, the sum of the dice and the point in this example). To begin our introduction to more elaborate user interfaces, this program uses an XHTML form (discussed in Chapter 4) and a new graphical user interface concept—GUI **event handling**. This is our first example in which the JavaScript executes in response to the user's interaction with a GUI component in an XHTML form. This interaction causes an event. Scripts are often used to respond to events.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.6: Craps.html -->
<!-- Craps game simulation. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Program that Simulates the Game of Craps</title>
      <style type = "text/css">
         table  { text-align: right }
         body   { font-family: arial, sans-serif }
         div.red { color: red }
      </style>
      <script type = "text/javascript">
         <!--
         // variables used to test the state of the game
         var WON = 0;
         var LOST = 1;
         var CONTINUE_ROLLING = 2;

         // other variables used in program
         var firstRoll = true; // true if current roll is first
         var sumOfDice = 0; // sum of the dice
         var myPoint = 0; // point if no win/loss on first roll
         var gameStatus = CONTINUE_ROLLING; // game not over yet

         // process one roll of the dice
         function play()
         {
            // get the point field on the page
            var point = document.getElementById( "pointfield" );

            // get the status div on the page
            var statusDiv = document.getElementById( "status" );
            if ( firstRoll ) // first roll of the dice
            {
```

Fig. 9.6 | Craps game simulation. (Part 1 of 4.)

```
38          sumOfDice = rollDice();
39
40          switch ( sumOfDice )
41          {
42             case 7: case 11: // win on first roll
43                gameStatus = WON;
44                // clear point field
45                point.value = "";
46                break;
47             case 2: case 3: case 12: // lose on first roll
48                gameStatus = LOST;
49                // clear point field
50                point.value = "";
51                break;
52             default:                    // remember point
53                gameStatus = CONTINUE_ROLLING;
54                myPoint = sumOfDice;
55                point.value = myPoint;
56                firstRoll = false;
57          } // end switch
58       } // end if
59       else
60       {
61          sumOfDice = rollDice();
62
63          if ( sumOfDice == myPoint ) // win by making point
64             gameStatus = WON;
65          else
66             if ( sumOfDice == 7 )     // lose by rolling 7
67                gameStatus = LOST;
68       } // end else
69
70       if ( gameStatus == CONTINUE_ROLLING )
71          statusDiv.innerHTML = "Roll again";
72       else
73       {
74          if ( gameStatus == WON )
75             statusDiv.innerHTML = "Player wins. " +
76                "Click Roll Dice to play again.";
77          else
78             statusDiv.innerHTML = "Player loses. " +
79                "Click Roll Dice to play again.";
80
81          firstRoll = true;
82       } // end else
83    } // end function play
84
85    // roll the dice
86    function rollDice()
87    {
88       var die1;
89       var die2;
90       var workSum;
```

Fig. 9.6 | Craps game simulation. (Part 2 of 4.)

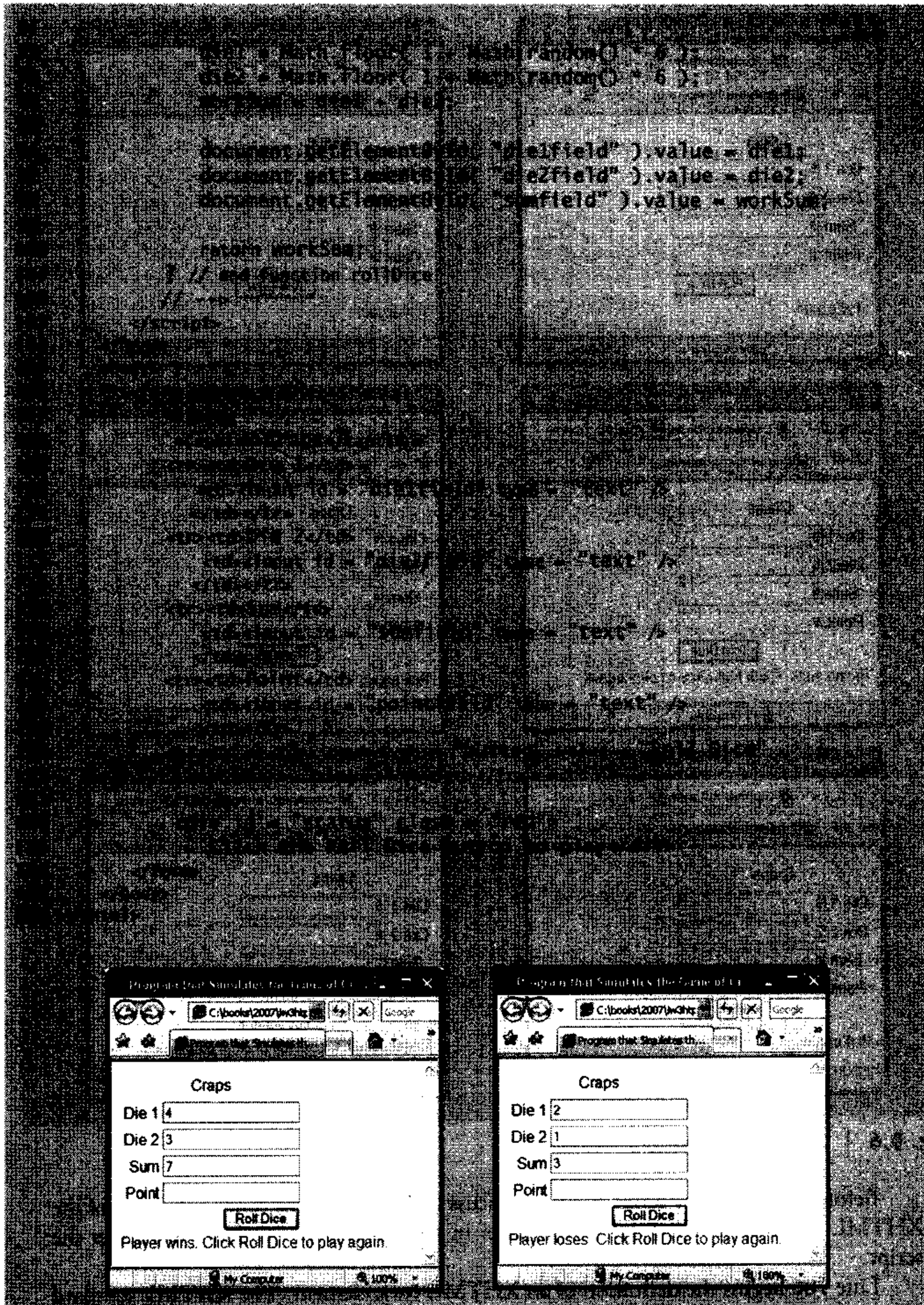


Fig. 9.6 | Craps game simulation. (Part 3 of 4.)

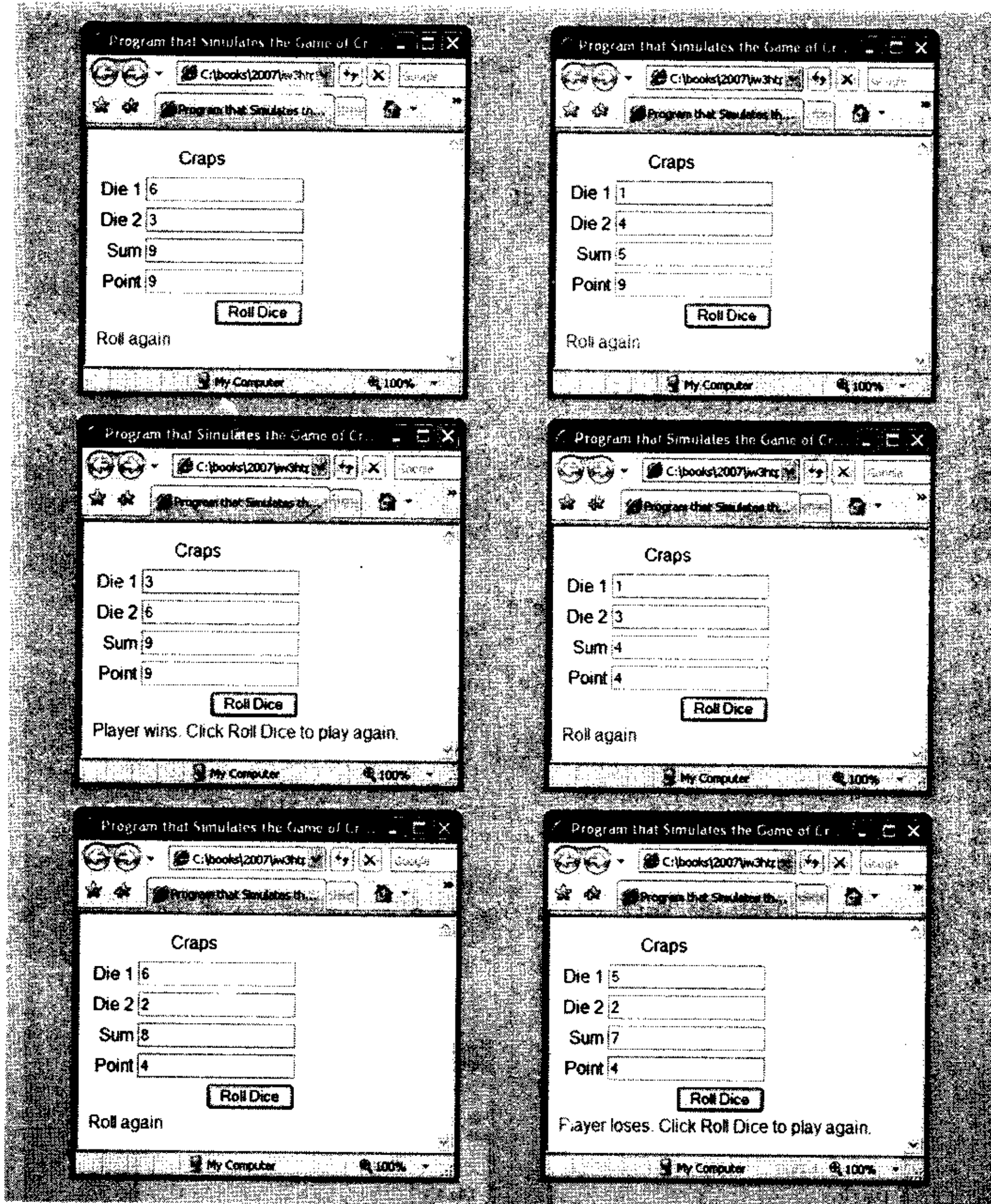


Fig. 9.6 | Craps game simulation. (Part 4 of 4.)

Before we discuss the script code, we discuss the `body` element (lines 105–126) of the XHTML document. The GUI components in this section are used extensively in the script.

Line 106 begins the definition of an XHTML `form` element. The XHTML standard requires that every `form` contain an `action` attribute, but because this form does not post its information to a web server, the empty string ("") is used.

In this example, we have decided to place the form's GUI components in an XHTML `table` element, so line 107 begins the definition of the XHTML table. Lines 109–120 create four table rows. Each row contains a left cell with a text label and an `input` element in the right cell.

Four `input` fields (lines 110, 113, 116 and 119) are created to display the value of the first die, the second die, the sum of the dice and the current point value, if any. Their `id` attributes are set to `die1field`, `die2field`, `sumfield`, and `pointfield`, respectively. The `id` attribute can be used to apply CSS styles and to enable script code to refer to an element in an XHTML document. Because the `id` attribute, if specified, must have a unique value, JavaScript can reliably refer to any single element via its `id` attribute. We see how this is done in a moment.

Lines 121–122 create a fifth row with an empty cell in the left column before the **Roll Dice** button. The button's **`onclick` attribute** indicates the action to take when the user of the XHTML document clicks the **Roll Dice** button. In this example, clicking the button causes a call to function `play`.

This style of programming is known as **event-driven programming**—the user interacts with a GUI component, the script is notified of the event and the script processes the event. The user's interaction with the GUI "drives" the program. The button click is known as the **event**. The function that is called when an event occurs is known as an **event-handling function** or **event handler**. When a GUI event occurs in a form, the browser calls the specified event-handling function. Before any event can be processed, each GUI component must know which event-handling function will be called when a particular event occurs. Most XHTML GUI components have several different event types. The event model is discussed in detail in Chapter 13, JavaScript: Events. By specifying `onclick = "play()"` for the **Roll Dice** button, we instruct the browser to **listen for events** (button-click events in particular). This **registers the event handler** for the GUI component, causing the browser to begin listening for the click event on the component. If no event handler is specified for the **Roll Dice** button, the script will not respond when the user presses the button.

Lines 123–125 end the `table` and `form` elements, respectively. After the table, a `div` element is created with an `id` attribute of `"status"`. This element will be updated by the script to display the result of each roll to the user. A style declaration in line 13 colors the text contained in this `div` red.

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Lines 18–20 create variables that define the three game states—game won, game lost and continue rolling the dice. Unlike many other programming languages, JavaScript does not provide a mechanism to define a **constant** (i.e., a variable whose value cannot be modified). For this reason, we use all capital letters for these variable names, to indicate that we do not intend to modify their values and to make them stand out in the code—a common industry practice for genuine constants.

Good Programming Practice 9.4

Use only uppercase letters (with underscores between words) in the names of variables that should be used as constants. This format makes such variables stand out in a program.

Good Programming Practice 9.5

Use meaningfully named variables rather than literal values (such as 2) to make programs more readable.

Lines 23–26 declare several variables that are used throughout the script. Variable `firstRoll` indicates whether the next roll of the dice is the first roll in the current game. Variable `sumOfDice` maintains the sum of the dice from the last roll. Variable `myPoint` stores the point if the player does not win or lose on the first roll. Variable `gameStatus` keeps track of the current state of the game (`WON`, `LOST` or `CONTINUE_ROLLING`).

We define a function `rollDice` (lines 86–101) to roll the dice and to compute and display their sum. Function `rollDice` is defined once, but is called from two places in the program (lines 38 and 61). Function `rollDice` takes no arguments, so it has an empty parameter list. Function `rollDice` returns the sum of the two dice.

The user clicks the **Roll Dice** button to roll the dice. This action invokes function `play` (lines 29–83) of the script. Lines 32 and 35 create two new variables with objects representing elements in the XHTML document using the `document` object's **`getElementById` method**. The `getElementById` method, given an `id` as an argument, finds the XHTML element with a matching `id` attribute and returns a JavaScript object representing the element. Line 32 stores an object representing the `pointfield` input element (line 119) in the variable `point`. Line 35 gets an object representing the `status div` from line 124. In a moment, we show how you can use these objects to manipulate the XHTML document.

Function `play` checks the variable `firstRoll` (line 36) to determine whether it is `true` or `false`. If `true`, the roll is the first roll of the game. Line 38 calls `rollDice`, which picks two random values from 1 to 6, displays the value of the first die, the value of the second die and the sum of the dice in the first three text fields and returns the sum of the dice. (We discuss function `rollDice` in detail shortly.) After the first roll (if `firstRoll` is false), the nested `switch` statement in lines 40–57 determines whether the game is won or lost, or whether it should continue with another roll. After the first roll, if the game is not over, `sumOfDice` is saved in `myPoint` and displayed in the text field `point` in the XHTML form.

Note how the text field's value is changed in lines 45, 50 and 55. The object stored in the variable `point` allows access to the `pointfield` text field's contents. The expression `point.value` accesses the **`value` property** of the text field referred to by `point`. The `value` property specifies the text to display in the text field. To access this property, we specify the object representing the text field (`point`), followed by a **dot** (.) and the name of the property to access (`value`). This technique for accessing properties of an object (also used to access methods as in `Math.pow`) is called **dot notation.** We discuss using scripts to access elements in an XHTML page in more detail in Chapter 13.

The program proceeds to the nested `if...else` statement in lines 70–82, which uses the `statusDiv` variable to update the `div` that displays the game status. Using the object's **`innerHTML` property**, we set the text inside the `div` to reflect the most recent status. In lines 71, 75–76 and 78–79, we set the `div`'s `innerHTML` to

```
Roll again.
```

if `gameStatus` is equal to `CONTINUE_ROLLING`, to

```
Player wins. Click Roll Dice to play again.
```

if `gameStatus` is equal to `WON` and to

```
Player loses. Click Roll Dice to play again.
```

if `gameStatus` is equal to `LOST`. If the game is won or lost, line 81 sets `firstRoll` to `true` to indicate that the next roll of the dice begins the next game.

The program then waits for the user to click the button **Roll Dice** again. Each time the user clicks **Roll Dice**, the program calls function `play`, which, in turn, calls the `rollDice` function to produce a new value for `sumOfDice`. If `sumOfDice` matches `myPoint`, `gameStatus` is set to `WON`, the `if...else` statement in lines 70–82 executes and the game is complete. If `sum` is equal to `7`, `gameStatus` is set to `LOST`, the `if...else` statement in lines 70–82 executes and the game is complete. Clicking the **Roll Dice** button starts a new game. The program updates the four text fields in the XHTML form with the new values of the dice and the sum on each roll, and updates the text field `point` each time a new game begins.

Function `rollDice` (lines 86–101) defines its own local variables `die1`, `die2` and `workSum` (lines 88–90). Because they are defined inside the `rollDice` function, these variables are accessible only inside that function. Lines 92–93 pick two random values in the range 1 to 6 and assign them to variables `die1` and `die2`, respectively. Lines 96–98 once again use the `document`'s `getElementById` method to find and update the correct `input` elements with the values of `die1`, `die2` and `workSum`. Note that the integer values are converted automatically to strings when they are assigned to each text field's `value` property. Line 100 returns the value of `workSum` for use in function `play`.

Software Engineering Observation 9.5

Variables that are declared inside the body of a function are known only in that function. If the same variable names are used elsewhere in the program, they will be entirely separate variables in memory.

Note the use of the various program-control mechanisms. The craps program uses two functions—`play` and `rollDice`—and the `switch`, `if...else` and nested `if` statements. Note also the use of multiple `case` labels in the `switch` statement to execute the same statements (lines 42 and 47). In the exercises at the end of this chapter, we investigate various interesting characteristics of the game of craps.

Error-Prevention Tip 9.2

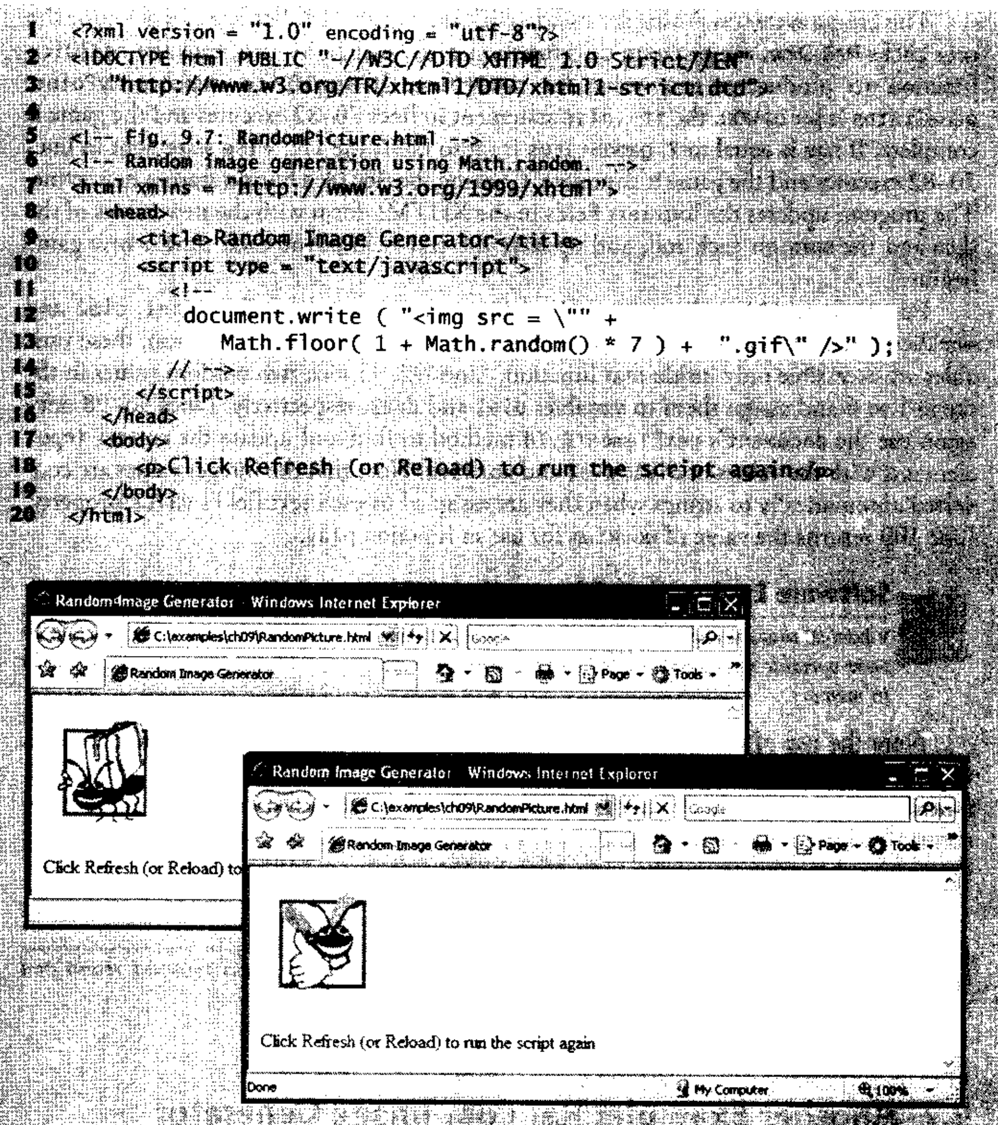
Initializing variables when they are declared in functions helps avoid incorrect results and interpreter messages warning of uninitialized data.

9.7 Another Example: Random Image Generator

Web content that varies randomly adds dynamic, interesting effects to a page. In the next example, we build a **random image generator**, a script that displays a randomly selected image every time the page that contains the script is loaded.

For the script in Fig. 9.7 to function properly, the directory containing the file `RandomPicture.html` must also contain seven images with integer filenames (i.e., `1.gif`, `2.gif`, ..., `7.gif`). The web page containing this script displays one of these seven images, selected at random, each time the page loads.

Lines 12–13 randomly select an image to display on a web page. This `document.write` statement creates an image tag in the web page with the `src` attribute set to a random integer from 1 to 7, concatenated with `".gif"`. Thus, the script dynamically sets the source of the image tag to the name of one of the image files in the current directory.



```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 9.7: RandomPicture.html -->
6  <!-- Random image generation using Math.random. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Random Image Generator</title>
10       <script type = "text/javascript">
11          <!--
12          document.write ( "<img src = \"" +
13             Math.floor( 1 + Math.random() * 7 ) + ".gif\" />" );
14          // -->
15       </script>
16    </head>
17    <body>
18       <p>Click Refresh (or Reload) to run the script again</p>
19    </body>
20 </html>
```


Fig. 9.7 | Random image generation using `Math.random`.

9.8 Scope Rules

Chapters 6–8 used identifiers for variable names. The attributes of variables include name, value and data type (e.g., string, number or boolean). We also use identifiers as names for user-defined functions. Each identifier in a program also has a scope.

The **scope** of an identifier for a variable or function is the portion of the program in which the identifier can be referenced. **Global variables** or **script-level variables** that are declared in the head element are accessible in any part of a script and are said to have **global scope**. Thus every function in the script can potentially use the variables.

Identifiers declared inside a function have **function** (or **local**) **scope** and can be used only in that function. Function scope begins with the opening left brace ({) of the function in which the identifier is declared and ends at the terminating right brace (}) of the function. Local variables of a function and function parameters have function scope. If a local variable in a function has the same name as a global variable, the global variable is "hidden" from the body of the function.

Good Programming Practice 9.6

Avoid local-variable names that hide global-variable names. This can be accomplished by simply avoiding the use of duplicate identifiers in a script.

The script in Fig. 9.8 demonstrates the **scope rules** that resolve conflicts between global variables and local variables of the same name. This example also demonstrates the **onload event** (line 52), which calls an event handler (start) when the <body> of the XHTML document is completely loaded into the browser window.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 9.8: scoping.html -->
6  <!-- Scoping example. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>A Scoping Example</title>
10       <script type = "text/javascript">
11          <!--
12          var x = 1; // global variable
13
14          function start()
15          {
16             var x = 5; // variable local to function start
17
18             document.writeln( "local x in start is " + x );
19
20             functionA(); // functionA has local x
21             functionB(); // functionB uses global variable x
22             functionA(); // functionA reinitializes local x
23             functionB(); // global variable x retains its value
24
25             document.writeln(
26                "<p>local x in start is " + x + "</p>" );
27          } // end function start
28
29          function functionA()
30          {
31             var x = 25; // initialized each time
32                         // functionA is called
33
34             document.writeln( "<p>local x in functionA is " +
35                x + " after entering functionA" );
```

Fig. 9.8 | Scoping example. (Part 1 of 2.)

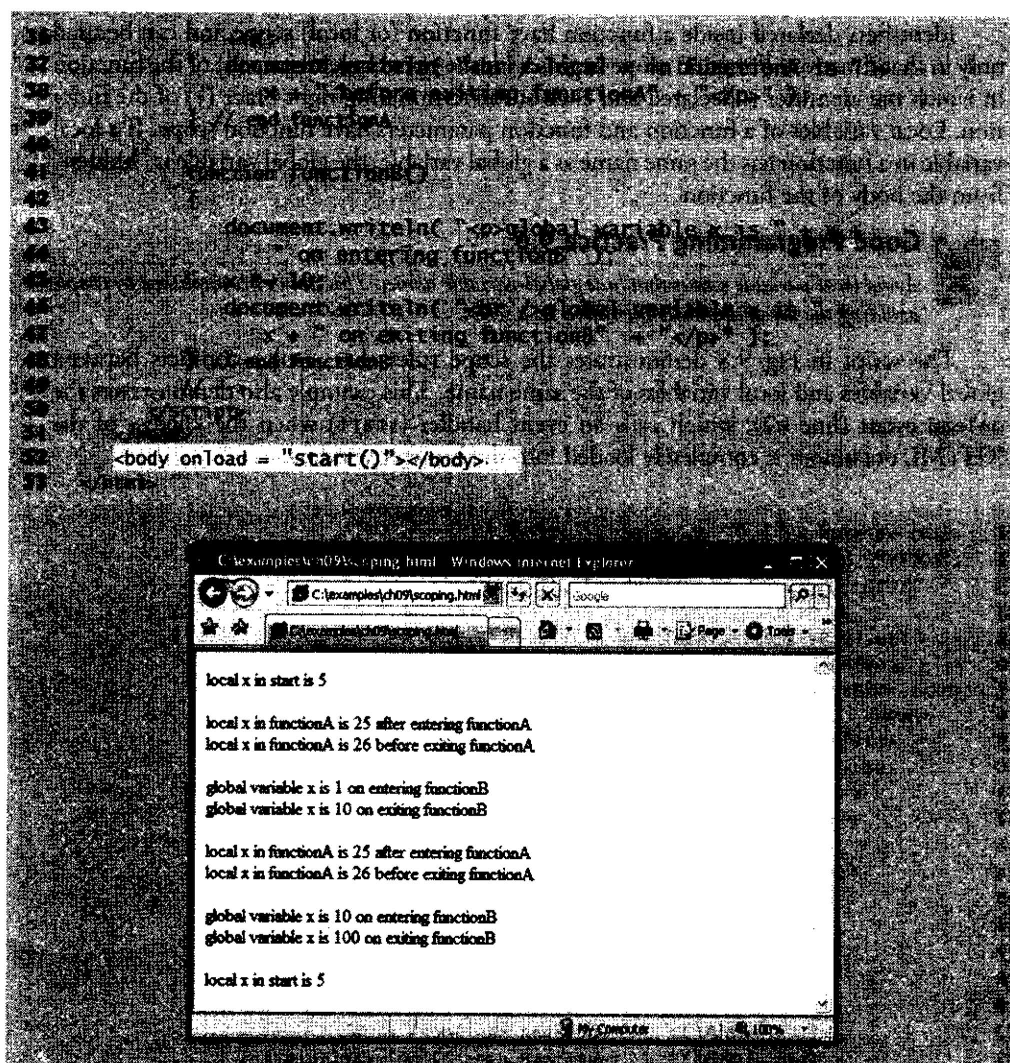


Fig. 9.8 | Scoping example. (Part 2 of 2.)

Global variable `x` (line 12) is declared and initialized to 1. This global variable is hidden in any block (or function) that declares a variable named `x`. Function `start` (line 14–27) declares a local variable `x` (line 16) and initializes it to 5. This variable is output in a line of XHTML text to show that the global variable `x` is hidden in `start`. The script defines two other functions—`functionA` and `functionB`—that each take no arguments and return nothing. Each function is called twice from function `start`.

Function `functionA` defines local variable `x` (line 31) and initializes it to 25. When `functionA` is called, the variable is output in a line of XHTML text to show that the global variable `x` is hidden in `functionA`; then the variable is incremented and output in a line of XHTML text again before the function is exited. Each time this function is called, local variable `x` is re-created and initialized to 25.

Function `functionB` does not declare any variables. Therefore, when it refers to variable `x`, the global variable `x` is used. When `functionB` is called, the global variable is output in a line of XHTML text, multiplied by `10` and output in a line of XHTML text again before the function is exited. The next time function `functionB` is called, the global variable has its modified value, `10`, which again gets multiplied by `10`, and `100` is output. Finally, the program outputs local variable `x` in `start` in a line of XHTML text again, to show that none of the function calls modified the value of `x` in `start`, because the functions all referred to variables in other scopes.

9.9 JavaScript Global Functions

JavaScript provides seven global functions. We have already used two of these functions—`parseInt` and `parseFloat`. The global functions are summarized in Fig. 9.9.

Actually, the global functions in Fig. 9.9 are all part of JavaScript's **`Global` object**. The `Global` object contains all the global variables in the script, all the user-defined functions in the script and all the functions listed in Fig. 9.9. Because global functions and user-defined functions are part of the `Global` object, some JavaScript programmers refer to these functions as methods. We use the term method only when referring to a function that is called for a particular object (e.g., `Math.random()`). As a JavaScript programmer, you do not need to use the `Global` object directly; JavaScript references it for you.

`escape`	Takes a string argument and returns a string in which all spaces, punctuation, accent characters and any other character that is not in the ASCII character set (see Appendix D, ASCII Character Set) are encoded in a hexadecimal format that can be re-presented on all platforms.
`eval`	Takes a string argument representing JavaScript code to execute. The JavaScript interpreter evaluates the code and executes it when the `eval` function is called. This function allows JavaScript code to be stored as strings and executed dynamically. [*Note:* It is considered a serious security risk to use `eval` to process any data entered by a user because a malicious user could exploit this to run dangerous code.]
`isFinite`	Takes a numeric argument and returns `true` if the value of the argument is not `NaN`, `Number.POSITIVE_INFINITY` or `Number.NEGATIVE_INFINITY` (values that are not numbers or numbers outside the range that JavaScript supports)—otherwise, the function returns `false`.
`isNaN`	Takes a numeric argument and returns `true` if the value of the argument is not a number; otherwise, it returns `false`. The function is commonly used with the return value of `parseInt` or `parseFloat` to determine whether the result is a proper numeric value.

Fig. 9.9 | JavaScript global functions. (Part 1 of 2.)

Global function	Description
parseFloat	Takes a string argument and attempts to convert the beginning of the string into a floating-point value. If the conversion is unsuccessful, the function returns NaN; otherwise, it returns the converted value (e.g., parseFloat("abc123.45") returns NaN, and parseFloat("123.45abc") returns the value 123.45).
parseInt	Takes a string argument and attempts to convert the beginning of the string into an integer value. If the conversion is unsuccessful, the function returns NaN; otherwise, it returns the converted value (e.g., parseInt("abc123") returns NaN, and parseInt("123abc") returns the integer value 123). This function takes an optional second argument, from 2 to 36, specifying the **radix** (or **base**) of the number. Base 2 indicates that the first argument string is in **binary** format, base 8 indicates that the first argument string is in **octal** format and base 16 indicates that the first argument string is in **hexadecimal** format.
unescape	Takes a string as its argument and returns a string in which all characters previously encoded with escape are decoded.

Fig. 9.9 | JavaScript global functions. (Part 2 of 2.)

9.10 Recursion

The programs we have discussed thus far are generally structured as functions that call one another in a disciplined, hierarchical manner. A **recursive function** is a function that calls *itself*, either directly, or indirectly through another function. **Recursion** is an important topic discussed at length in computer science courses. In this section, we present a simple example of recursion.

We consider recursion conceptually first; then we examine several programs containing recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or **base case(s)**. If the function is called with a base case, the function returns a result. If the function is called with a more complex problem, it divides the problem into two conceptual pieces—a piece that the function knows how to process (the base case) and a piece that the function does not know how to process. To make recursion feasible, the latter piece must resemble the original problem, but be a simpler or smaller version it. Because this new problem looks like the original problem, the function invokes (calls) a fresh copy of itself to go to work on the smaller problem; this invocation is referred to as a **recursive call**, or the **recursion step**. The recursion step also normally includes the keyword return, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the function is still open (i.e., it has not finished executing). The recursion step can result in many more recursive calls as the function divides each new subproblem into two conceptual pieces. For the recursion eventually to terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original function call eventually returns the final result to the caller. This process sounds exotic when compared with the conventional problem solving we have performed to this point.

As an example of these concepts at work, let us write a recursive program to perform a popular mathematical calculation. The factorial of a nonnegative integer n, written $n!$ (and pronounced "n factorial"), is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

where 1! is equal to 1 and 0! is defined as 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer (`number` in the following example) greater than or equal to zero can be calculated **iteratively** (nonrecursively) using a `for` statement, as follows:

```
var factorial = 1;

for ( var counter = number; counter >= 1; --counter )
   factorial *= counter;
```

A recursive definition of the factorial function is arrived at by observing the following relationship:

$$n! = n \cdot (n-1)!$$

For example, *5!* is clearly equal to 5 * 4!, as is shown by the following equations:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 9.10. Figure 9.10 (a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion. Figure 9.10 (b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 9.11 uses recursion to calculate and print the factorials of the integers 0 to 10. The recursive function `factorial` first tests (line 24) whether a terminating condition is `true`, i.e., whether `number` is less than or equal to 1. If so, `factorial` returns 1, no further recursion is necessary and the function returns. If `number` is greater than 1, line 27 expresses the problem as the product of `number` and the value returned by a recursive call to `factorial` evaluating the factorial of `number - 1`. Note that `factorial( number - 1 )` is a simpler problem than the original calculation, `factorial( number )`.

Function `factorial` (lines 22–28) receives as its argument the value for which to calculate the factorial. As can be seen in the screen capture in Fig. 9.11, factorial values become large quickly.

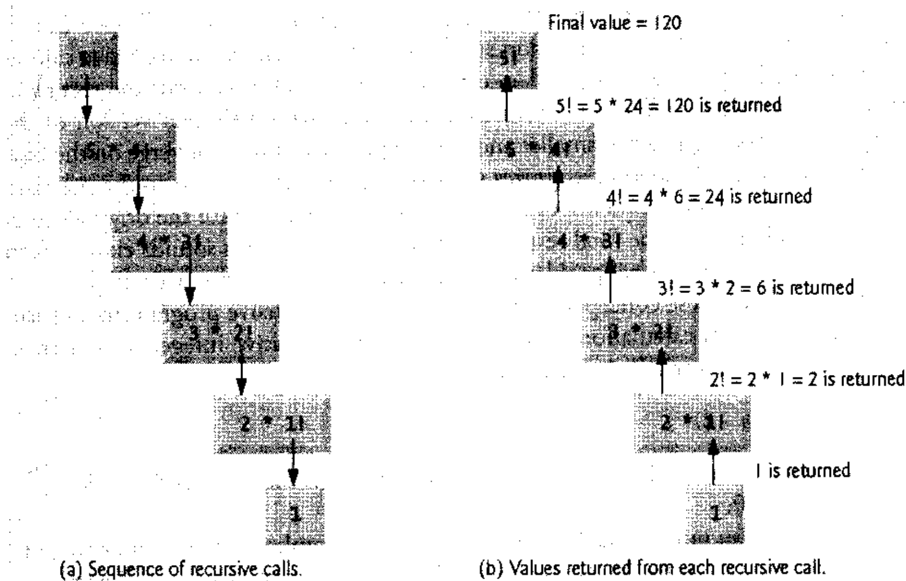


Fig. 9.10 | Recursive evaluation of 5!.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.11: FactorialTest.html -->
<!-- Factorial calculation with a recursive function. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Recursive Factorial Function</title>
      <script type = "text/javascript">
         <!--
         document.writeln( "<h1>Factorials of 1 to 10</h1>" );
         document.writeln( "<table>" );

         for ( var i = 0; i <= 10; i++ )
            document.writeln( "<tr><td>" + i + "!</td><td>" +
               factorial( i ) + "</td></tr>" );

         document.writeln( "</table>" );

         // Recursive definition of function factorial
         function factorial( number )
         {
            if ( number <= 1 )  // base case
               return 1;
            else
               return number * factorial( number - 1 );
         } // end function factorial
```

Fig. 9.11 | Factorial calculation with a recursive function. (Part 1 of 2.)

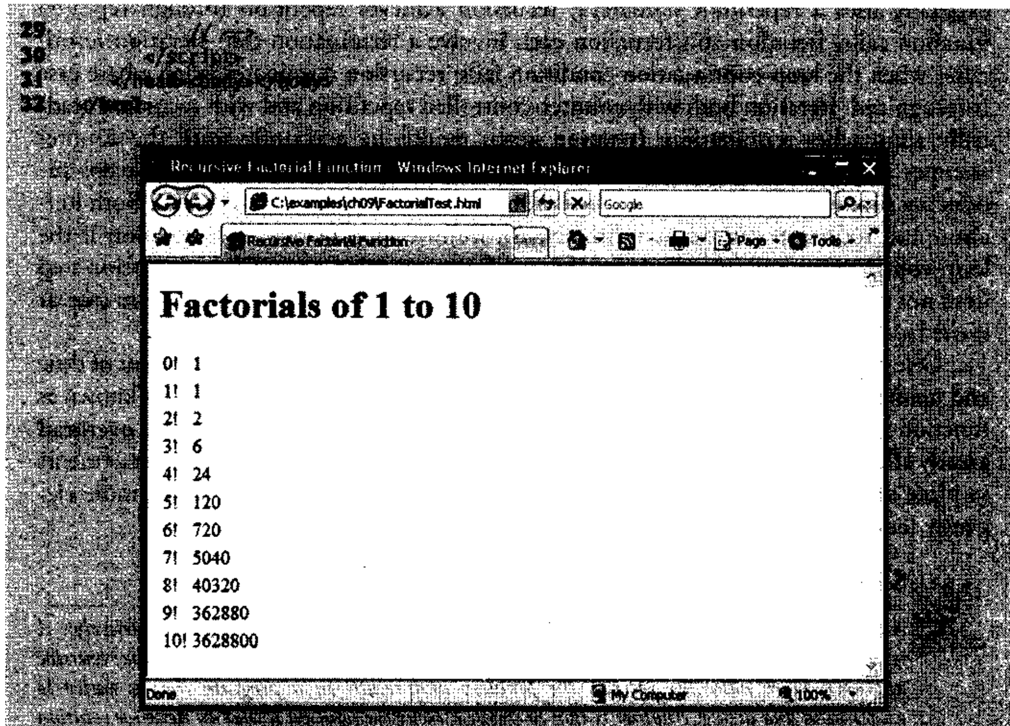


Fig. 9.11 | Factorial calculation with a recursive function. (Part 2 of 2.)

Common Programming Error 9.6

Forgetting to return a value from a recursive function when one is needed results in a logic error.

Common Programming Error 9.7

Omitting the base case and writing the recursion step incorrectly so that it does not converge on the base case are both errors that cause infinite recursion, eventually exhausting memory. This situation is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

Error-Prevention Tip 9.3

Internet Explorer displays an error message when a script seems to be going into infinite recursion. Firefox simply terminates the script after detecting the problem. This allows the user of the web page to recover from a script that contains an infinite loop or infinite recursion.

9.11 Recursion vs. Iteration

In the preceding section, we studied a function that can easily be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control statement: Iteration uses a repetition statement (e.g., `for`, `while` or `do...while`); recursion uses a selection statement (e.g., `if`, `if...else` or `switch`). Both iteration and recursion involve repetition: Iteration

explicitly uses a repetition statement; recursion achieves repetition through repeated function calls. Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration both with counter-controlled repetition and with recursion gradually approaches termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time via a sequence that converges on the base case or if the base case is incorrect.

One negative aspect of recursion is that function calls require a certain amount of time and memory space not directly spent on executing program instructions. This is known as function-call overhead. Because recursion uses repeated function calls, this overhead greatly affects the performance of the operation. In many cases, using repetition statements in place of recursion is more efficient. However, some problems can be solved more elegantly (and more easily) with recursion.

Software Engineering Observation 9.6

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Performance Tip 9.1

Avoid using recursion in performance-oriented situations. Recursive calls take time and consume additional memory.

Common Programming Error 9.8

Accidentally having a nonrecursive function call itself, either directly, or indirectly through another function, can cause infinite recursion.

In addition to the Factorial function examnple (Fig. 9.11), we also provide several recursion exercises—raising an integer to an integer power (Exercise 9.20), visualizing recursion and sum of two integers (Exercise 9.21). Also, Fig. 14.26 uses recursion to traverse an XML document tree.

9.12 Web Resources

`www.deitel.com/javascript/`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on XHTML (`www.deitel.com/xhtml/`) and CSS 2.1 (`www.deitel.com/css21/`).

Summary

Section 9.1 Introduction

- The best way to develop and maintain a large program is to construct it from small, simple pieces, or modules. This technique is called divide and conquer.

Section 9.2 Program Modules in JavaScript

- JavaScript programs are written by combining new functions that the programmer writes with "prepackaged" functions and objects available in JavaScript.
- The term method implies that the function belongs to a particular object. We refer to functions that belong to a particular JavaScript object as methods; all others are referred to as functions.
- JavaScript provides several objects that have a rich collection of methods for performing common mathematical calculations, string manipulations, date and time manipulations, and manipulations of collections of data called arrays. These objects make your job easier, because they provide many of the capabilities programmers frequently need.
- Whenever possible, use existing JavaScript objects, methods and functions instead of writing new ones. This reduces script-development time and helps avoid introducing errors.
- You can define functions that perform specific tasks and use them at many points in a script. These functions are referred to as programmer-defined functions. The actual statements defining the function are written only once and are hidden from other functions.
- Functions are invoked by writing the name of the function, followed by a left parenthesis, followed by a comma-separated list of zero or more arguments, followed by a right parenthesis.
- Methods are called in the same way as functions, but require the name of the object to which the method belongs and a dot preceding the method name.
- Function (and method) arguments may be constants, variables or expressions.

Section 9.3 Programmer-Defined Functions

- All variables declared in function definitions are local variables—this means that they can be accessed only in the function in which they are defined.
- A function's parameters are considered to be local variables. When a function is called, the arguments in the call are assigned to the corresponding parameters in the function definition.
- Code that is packaged as a function can be executed from several locations in a program by calling the function.
- Each function should perform a single, well-defined task, and the name of the function should express that task effectively. This promotes software reusability.

Section 9.4 Function Definitions

- The return statement passes information from inside a function back to the point in the program where it was called.
- A function must be called explicitly for the code in its body to execute.
- The format of a function definition is

```
function function-name( parameter-list )
{
    declarations and statements
}
```

- There are three ways to return control to the point at which a function was invoked. If the function does not return a result, control returns when the program reaches the function-ending right brace or by executing the statement `return;`. If the function does return a result, the statement `return` *expression*; returns the value of *expression* to the caller.

Section 9.5 Random Number Generation

- Method `random` generates a floating-point value from 0.0 up to, but not including, 1.0.
- Random integers in a certain range can be generated by scaling and shifting the values returned by `random`, then using Math.floor to convert them to integers. The scaling factor determines the size of the range (i.e. a scaling factor of 4 means four possible integers). The shift number is added to the result to determine where the range begins (i.e. shifting the numbers by 3 would give numbers between 3 and 7.)

Section 9.6 Example: Game of Chance

- JavaScript can execute actions in response to the user's interaction with a GUI component in an XHTML form. This is referred to as GUI event handling
- An XHTML element's `onclick` attribute indicates the action to take when the user of the XHTML document clicks on the element.
- In event-driven programming, the user interacts with a GUI component, the script is notified of the event and the script processes the event. The user's interaction with the GUI "drives" the program. The function that is called when an event occurs is known as an event-handling function or event handler.
- The `getElementById` method, given an `id` as an argument, finds the XHTML element with a matching `id` attribute and returns a JavaScript object representing the element.
- The `value` property of a JavaScript object representing an XHTML text input element specifies the text to display in the text field.
- Using an XHTML container (e.g. `div`, `span`, `p`) object's `innerHTML` property, we can use a script to set the contents of the element.

Section 9.7 Another Example: Random Image Generator

- We can use random number generation to randomly select from a number of images in order to display a random image each time a page loads.

Section 9.8 Scope Rules

- Each identifier in a program has a scope. The scope of an identifier for a variable or function is the portion of the program in which the identifier can be referenced.
- Global variables or script-level variables (i.e., variables declared in the head element of the XHTML document) are accessible in any part of a script and are said to have global scope. Thus every function in the script can potentially use the variables.
- Identifiers declared inside a function have function (or local) scope and can be used only in that function. Function scope begins with the opening left brace ({) of the function in which the identifier is declared and ends at the terminating right brace (}) of the function. Local variables of a function and function parameters have function scope.
- If a local variable in a function has the same name as a global variable, the global variable is "hidden" from the body of the function.
- The `onload` property of the body element calls an event handler when the body of the XHTML document is completely loaded into the browser window.

Section 9.9 JavaScript Global Functions

- JavaScript provides seven global functions as part of a `Global` object. This object contains all the global variables in the script, all the user-defined functions in the script and all the built-in global functions functions listed in Fig. 9.9.
- You do not need to use the `Global` object directly; JavaScript uses it for you.

Section 9.10 Recursion

- A recursive function calls itself, either directly, or indirectly through another function.
- A recursive function knows how to solve only the simplest case, or base case. If the function is called with a base case, it returns a result. If the function is called with a more complex problem, it knows how to divide the problem into two conceptual pieces—a piece that the function knows how to process (the base case) and a simpler or smaller version of the original problem.
- The function invokes (calls) a fresh copy of itself to go to work on the smaller problem; this invocation is referred to as a recursive call, or the recursion step.
- The recursion step executes while the original call to the function is still open (i.e., it has not finished executing).
- For recursion eventually to terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original function call eventually returns the final result to the caller.

Section 9.11 Recursion vs. Iteration

- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition statement; recursion achieves repetition through repeated function calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached.
- Recursion repeatedly invokes the mechanism and, consequently, the overhead of function calls. This effect can be expensive in terms of processor time and memory space.
- Some problems can be understood or solved more easily with recursion than with iteration.

Terminology

argument in a function call
base case
binary format
block
called function
caller
calling function
computer-assisted instruction (CAI)
constant
converge on the base case
copy of a value
divide and conquer
dot (`.`)
dot notation
element of chance
`escape` function
`eval` function
element of chance
event
event handler
event-handling function
event-driven programming

floor method of the Math object
function
function (local) scope
function argument
function body
function call
function definition
function name
function parameter
function-call operator ()
getElementById method of the document object
Global object
global scope
global variable
hexadecimal
innerHTML property
invoke a function
isFinite function
isNaN function
iterative solution
listen for events
local scope
local variable
max method of the Math object
method
modularize a program
module
object
octal
onclick event
onload event
parameter in a function definition
parseFloat function
parseInt function
programmer-defined function
probability
programmer-defined function
radix
random method of the Math object
random-number generation
recursion
recursive function
recursive step
registering an event handler
respond to an event
return statement
scaling
scaling factor
scope
script-level variable
shifting value
simulation
software engineering
software reusability
unescape function
value property of an XHTML text field

Self-Review Exercises

9.1 Fill in the blanks in each of the following statements:

a) Program modules in JavaScript are called ________.
b) A function is invoked using a(n) ________.
c) A variable known only inside the function in which it is defined is called a(n) ________.
d) The ________ statement in a called function can be used to pass the value of an expression back to the calling function.
e) The keyword ________ indicates the beginning of a function definition.

9.2 For the given program, state the scope (either global scope or function scope) of each of the following elements:

a) The variable x.
b) The variable y.
c) The function cube.
d) The function output.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3     "http://www.w3.org/TR/xhtml11/DTD/xhtml11-transitional.dtd">
4
5  <!-- Exercise 9.2: cube.html -->
6  <html xmlns = "http://www.w3.org/1999/xhtml">
```