9. JavaScript: Functions

E pluribus unum. (One composed of many.)

—Virgil

Call me Ishmael.

—Herman Melville

When you call me that, smile.

—Owen Wister

O! call back yesterday, bid time return.

—William Shakespeare

Objectives

In this chapter you will:

- Construct programs modularly from small pieces called functions.
- Define new functions.
- Pass information between functions.
- Use simulation techniques based on random number generation.
- Use the new HTML5 `audio` and `video` elements
- Use additional global methods.
- See how the visibility of identifiers is limited to specific regions of programs.

Outline

9.1. Introduction

Most computer programs that solve real-world problems are much larger than those 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.

You'll start using JavaScript to interact programatically with elements in a web page so you can obtain values from elements (such as those in HTML5 `form` s) and place content into web-page elements. We'll also take a brief excursion into simulation techniques with random number generation and develop a version of the casino dice game called craps that uses most of the programming techniques you've used to this point in the book. In the game, we'll also introduce HTML5's new `audio` and `video` elements that enable you to embed audio and video in your web pages. We'll also programmatically interact with the `audio` element to play the audio in response to a user interaction with the game.

9.2. Program Modules in JavaScript

Scripts that you write in JavaScript typically contain of one or more pieces called **functions**. You'll combine new functions that you write with prepackaged functions and objects available in JavaScript. The prepackaged functions that belong to JavaScript objects (such as `Math.pow` , introduced previously) are called **methods**.

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 (discussed in Chapters 10–11) make your job easier, because they provide many of the capabilities you'll frequently need.

You can write functions to define tasks that may be used at many points in a script. These 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** (that is, 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` also acts as a "boss" function to `worker4` and `worker5`, and `worker4` and `worker5` report back to `worker1`.

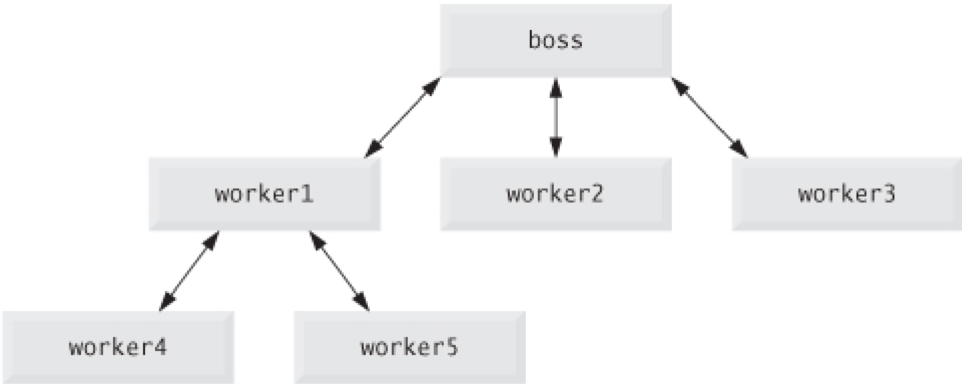


Fig. 9.1. Hierarchical boss-function/worker-function relationship.

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, the 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.

9.3. Function Definitions

We now consider how you can write your own customized functions and call them in a script.

9.3.1. 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 HTML5 document is empty and the document is created directly by JavaScript. In this chapter and later ones, we also show examples in which scripts interact with the elements in the `body` of a document.]

Invoking Function `square`

The `for` statement in lines 17–19 outputs HTML5 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 HTML5 document. Function `square` is invoked, or called, in line 19 with the expression `square(x)`. When program control reaches this expression, the program calls function `square` (defined in lines 23–26). The parentheses `()` in line 19 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 the function `square`'s definition (line 23). 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 returned (passed back) to the point in line 19 where `square` was invoked. Lines 18–19 concatenate the string `"<p>The square of "`, the value of `x`, the string `" is "`, the value returned by function `square` and the string `"`

`</p>"` , and write that line of text into the HTML5 document to create a new paragraph in the page. This process is repeated 10 times.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.2: SquareInt.html -->
4  <!-- Programmer-defined function square. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>A Programmer-Defined square Function</title>
9      <style type = "text/css">
10        p { margin: 0; }
11     </style>
12     <script>
13
14        document.writeln( "<h1>Square the numbers from 1 to 10</h1>" );
15
16        // square the numbers from 1 to 10
17        for ( var x = 1; x <= 10; ++x )
18          document.writeln( "<p>The square of " + x + " is " +
19            square( x ) + "</p>" );
20
21        // The following square function definition's body is executed
22        // only when the function is called explicitly as in line 19
23        function square( y )
24        {
25          return y * y;
26        } // end function square
27
28     </script>
29   </head><body></body> <!-- empty body element -->
30 </html>
```

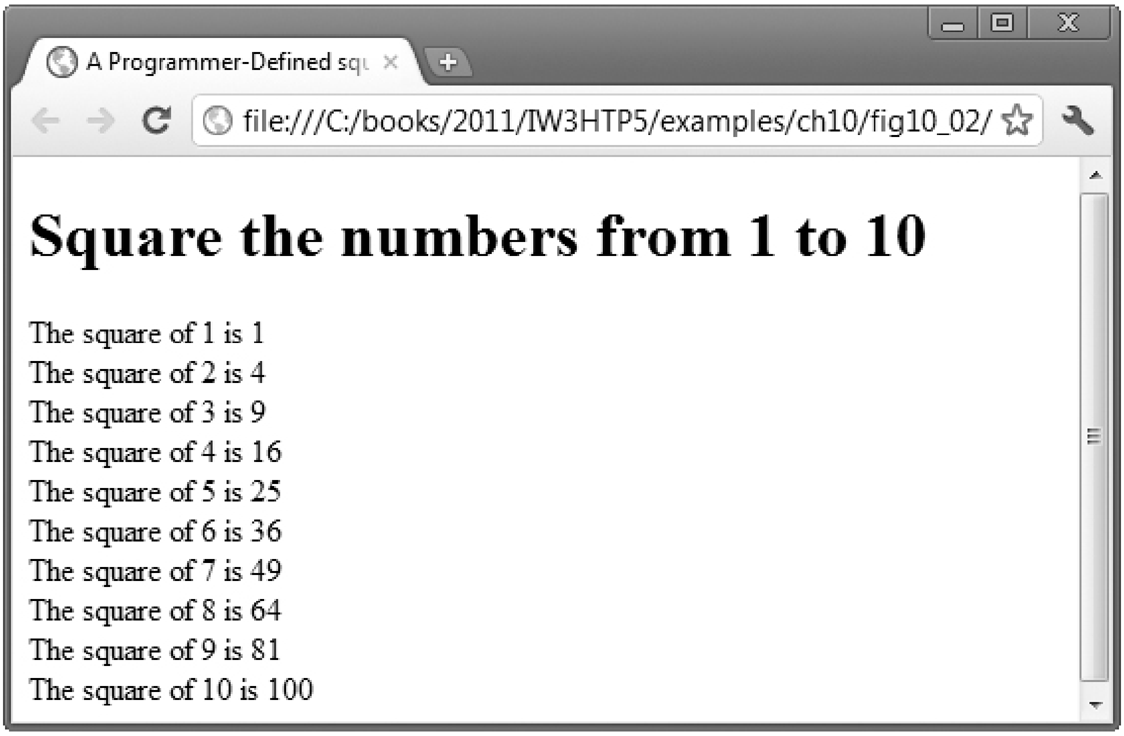


Fig. 9.2. Programmer-defined function `square` .

`square` Function Definition

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 the function call in line 19. The **return statement** in `square` passes the result of the calculation `y * y` back to the calling function. JavaScript keyword `var` is *not* used to declare function parameters (line 25).

Flow of Control in a Script That Contains Functions

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 in this example terminates, the script terminates.

General Format of a Function Definition

The general 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's called (remember that the arguments in the function call are assigned to the corresponding parameters 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* between the braces form the **function body**.

COMMON PROGRAMMING ERROR 9.1

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

Returning Program Control from a Function Definition

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 executes 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 executes, control returns immediately to the point at which the function was invoked.

9.3.2. 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.]

```
<!DOCTYPE html>

<!-- Fig. 9.3: maximum.html -->
<!-- Programmer-Defined maximum function. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Maximum of Three Values</title>
    <style type = "text/css">
       p { margin: 0; }
     </style>
     <script>

       var input1 = window.prompt( "Enter first number", "0" );
       var input2 = window.prompt( "Enter second number", "0" );
       var input3 = window.prompt( "Enter third number", "0" );

       var value1 = parseFloat( input1 );
       var value2 = parseFloat( input2 );
       var value3 = parseFloat( input3 );

       var maxValue = maximum( value1, value2, value3 );

       document.writeln( "<p>First number: " + value1 + "</p>" +
         "<p>Second number: " + value2 + "</p>" +
         "<p>Third number: " + value3 + "</p>" +
         "<p>Maximum is: " + maxValue + "</p>" );

       // maximum function definition (called from line 22)
```

```
         function maximum( x, y, z )
         {
            return Math.max( x, Math.max( y, z ) );
         } // end function maximum

      </script>
   </head><body></body>
</html>
```

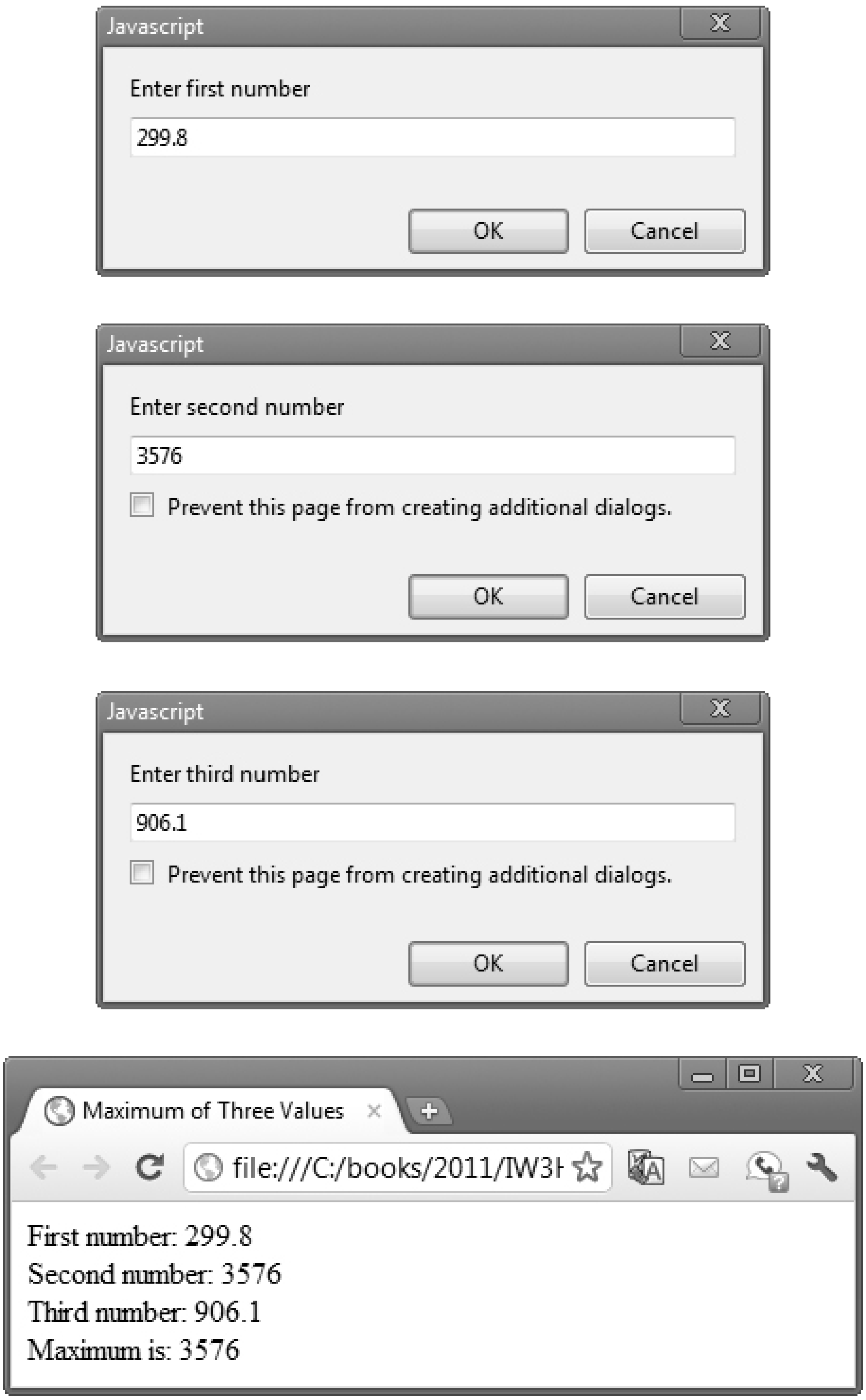


Fig. 9.3. Programmer-defined **`maximum`** function.

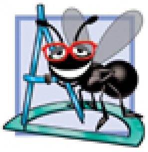
The three floating-point values are input by the user via `prompt` dialogs (lines 14–16). Lines 18–20 use function `parseFloat` to convert the strings entered by the user to floating-point values. The statement in line 22 passes the three floating-point values to function `maximum` (defined in lines 30–33). The function then determines the largest value and returns that value to line 22 by using the `return` statement (line 32). The returned value is assigned to variable `maxValue`. Lines 24–27 display the three floating-point values entered by the user and the calculated `maxValue`.

The first line of the function definition indicates that the function is named `maximum` and takes parameters `x`, `y` and `z`. 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 (line 22).

9.4. Notes on Programmer-Defined Functions

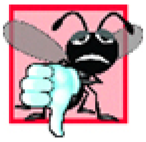
All variables declared with the keyword `var` in function definitions are **local variables**—this means that they can be accessed *only* in the function in which they're defined. A function's parameters are also considered to be local variables.

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, significant portions of 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.



SOFTWARE ENGINEERING OBSERVATION 9.1

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

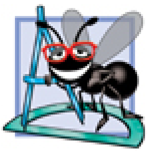


COMMON PROGRAMMING ERROR 9.2

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

GOOD PROGRAMMING PRACTICE 9.1

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 9.2

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

9.5. Random Number Generation

We now take a brief and hopefully entertaining excursion into a popular programming application, namely simulation and game playing. In this section and the next, we develop a carefully structured game-playing program that includes multiple functions. The program uses most of the control statements we've studied.

There's 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's 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**.

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 in that range has an equal **chance** (or **probability**) of being chosen each time `random` is called.

9.5.1. Scaling and Shifting Random Numbers

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 1–6. 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's develop a program that simulates 30 rolls of a six-sided die and displays the value of each roll (Fig. 9.4). We use the multiplication operator (`*`) with `random` as follows (line 21):

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

The preceding expression multiplies the result of a call to `Math.random()` by 6 to produce a value from 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 determine the closest integer *not greater than* the argument's value—for example, `Math.floor(1.75)` is `1` and `Math.floor(6.75)` is `6`. Figure 9.4 confirms that the results are in the range 1 to 6. To add space between the values being displayed, we output each value as an `li` element in an ordered list. The CSS style in line 11 places a margin of 10 pixels to the right of each `li` and indicates that they should display inline rather than vertically on the page.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.4: RandomInt.html -->
4  <!-- Random integers, shifting and scaling. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Shifted and Scaled Random Integers</title>
9      <style type = "text/css">
10        p, ol { margin: 0; }
11        li    { display: inline; margin-right: 10px; }
12     </style>
13     <script>
14
15        var value;
16
17        document.writeln( "<p>Random Numbers</p><ol>" );
18
19        for ( var i = 1; i <= 30; ++i )
20        {
21           value = Math.floor( 1 + Math.random() * 6 );
```

```
            document.writeln( "<li>" + value + "</li>" );
         } // end for

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

      </script>
   </head><body></body>
</html>
```

Fig. 9.4. Random integers, shifting and scaling.

9.5.2. Displaying Random Images

Web content that varies randomly can add dynamic, interesting effects to a page. In the next example, we build a **random image generator**—a script that displays four randomly selected die images every time the user clicks a **Roll Dice** button on the page. For the script in Fig. 9.5 to function properly, the directory containing the file `RollDice.html` must also contain the six die images with the filenames `die1.png`, `die2.png`, `die3.png`, `die4.png`, `die5.png` and `die6.png`—these are included with this chapter's examples.

```
<!DOCTYPE html>

<!-- Fig. 9.5: RollDice.html -->
<!-- Random dice image generation using Math.random. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Random Dice Images</title>
```

```
      <style type = "text/css">
         li { display: inline; margin-right: 10px; }
         ul { margin: 0; }
      </style>
      <script>
         // variables used to interact with the i mg elements
         var die1Image;
         var die2Image;
         var die3Image;
         var die4Image;

         // register button listener and get the img elements
         function start()
         {
            var button = document.getElementById( "rollButton" );
            button.addEventListener( "click", rollDice, false );
            die1Image = document.getElementById( "die1" );
            die2Image = document.getElementById( "die2" );
            die3Image = document.getElementById( "die3" );
            die4Image = document.getElementById( "die4" );
         } // end function rollDice

         // roll the dice
         function rollDice()
         {
            setImage( die1Image );
            setImage( die2Image );
            setImage( die3Image );
            setImage( die4Image );
         } // end function rollDice

         // set image source for a die
         function setImage( dieImg )
         {
            var dieValue = Math.floor( 1 + Math.random() * 6 );
            dieImg.setAttribute( "src", "die" + dieValue + ".png" );
```

```
          dieImg.setAttribute( "alt",
             "die image with " + dieValue + " spot(s)" );
        } // end function setImage

        window.addEventListener( "load", start, false );
      </script>
   </head>
   <body>
      <form action = "#">
         <input id = "rollButton" type = "button" value = "Roll Dice">
      </form>
      <ol>
         <li><img id = "die1" src = "blank.png" alt = "die 1 image"></li>
         <li><img id = "die2" src = "blank.png" alt = "die 2 image"></li>
         <li><img id = "die3" src = "blank.png" alt = "die 3 image"></li>
         <li><img id = "die4" src = "blank.png" alt = "die 4 image"></li>
      </ol>
   </body>
</html>
```

Fig. 9.5. Random dice image generation using Math.random.

User Interactions Via Event Handling

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're 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.

Inputs are typically received from the user via an HTML5 `form` (such as one in which the user enters name and address information). Outputs are typically displayed to the user in the web page (e.g., the die images in this example). To begin our introduction to more elaborate user interfaces, this program uses an HTML5 `form` (discussed in Chapters 2–3) 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 an element in a `form`. This interaction causes an *event*. Scripts are often used to respond to user initiated events.

The `body` Element

Before we discuss the script code, consider the `body` element (lines 52–62) of this document. The elements in the `body` are used extensively in the script.

The `form` Element

Line 53 begins the definition of an HTML5 `form` element. The HTML5 standard requires that every `form` contain an `action` attribute, but because this form does not post its information to a web server, the string `"#"` is used simply to allow this document to validate. The `#` symbol by itself represents the current page.

The `button input` Element and Event-Driven Programming

Line 54 defines a **`button`** `input` element with the `id "rollButton"` and containing the value **Roll Dice** which is displayed on the `button`. As you'll see, this example's script will handle the `button`'s **`click` event**, which occurs when the user clicks the `button`. In this example, clicking the button will call function `rollDice`, which we'll discuss shortly.

This style of programming is known as **event-driven programming**—the user *interacts* with an element in the web page, 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's called when an event occurs is known as an **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 element must know which event-handling function will be called when a particular event occurs. Most HTML5 elements have several different event types. The event model is discussed in detail in Chapter 13.

The `img` Elements

The four `img` elements (lines 57–60) will display the four randomly selected dice. Their `id` attributes (`die1`, `die2`, `die3` and `die4`, respectively) can be used to apply CSS styles and to enable script code to refer to these element in the HTML5 document. Because the `id` attribute, if specified, must have a unique value among all `id` attributes in the page, JavaScript can reliably refer to any single element via its `id` attribute. In a moment we'll see how this is done. Each `img` element displays the image `blank.png` (an empty white image) when the page first renders.

Specifying a Function to Call When the Browser Finishes Loading a Document

From this point forward, many of our examples will execute a JavaScript function when the document finishes loading in the web browser window. This is accomplished by handling the `window` object's **`load` event**. To specify the function to call when an event occurs, you **registering an event handler** for that event. We register the `window`'s `load` event handler at line 49. Method **addEventListener** is available for every DOM node. The method takes three arguments:

- the first is the name of the event for which we're registering a handler
- the second is the function that will be called to handle the event
- the last argument is typically `false`—the `true` value is beyond this book's scope

Line 49 indicates that function `start` (lines 21–29) should execute as soon as the page finishes loading.

Function `start`

When the `window`'s `load` event occurs, function start registers the **Roll Dice** button's `click` event handler (lines 23–24), which instructs the browser to **listen for events** (click events in particular). If no event handler is specified for the **Roll Dice** button, the script will not respond when the user presses the button. Line 23 uses the `document` object's **getElementById method**, which, given an HTML5 element's `id` as an argument, finds the element with the matching `id` attribute and returns a JavaScript object representing the element. This object allows the script to programmatically interact with the corresponding element in the web page. For example, line 24 uses the object representing the `button` to call function `addEventListener`—in this case, to indicate that function `rollDice` should be called when the button's click event occurs. Lines 25–28 get the objects representing the four `img` elements in lines 57–60 and assign them to the script variables in declared in lines 15–18.

Function `rollDice`

The user clicks the **Roll Dice** button to roll the dice. This event invokes function `rollDice` (lines 32–38) in the script. Function `rollDice` takes no arguments, so it has an empty parameter list. Lines 34–37 call function `setImage` (lines 41–47) to randomly select and set the image for a specified `img` element.

Function `setImage`

Function `setImage` (lines 41–47) receives one parameter (`dieImg`) that represents the specific `img` element in which to display a randomly selected image. Line 43 picks a random integer from 1 to 6. Line 44 demonstrates how to access an `img` element's `src` attribute programmatically in JavaScript. Each JavaScript object that represents an element of the HTML5 document has a **`setAttribute`** method that allows you to change the values of most of the HTML5 element's attributes. In this case, we change the `src` attribute of the `img` element referred to by `dieImg`. The `src` attribute specifies the location of the image to display. We set the `src` to a concatenated string containing the word `"die"`, a randomly generated integer from 1 to 6 and the file extension `".png"` to complete the image file name. Thus, the script dynamically sets the `img` element's `src` attribute to the name of one of the image files in the current directory.

Continuing to Roll the Dice

The program then waits for the user to click the **Roll Dice** button again. Each time the user does so, the program calls `rollDice`, which repeatedly calls `setImage` to display new die images.

9.5.3. Rolling Dice Repeatedly and Displaying Statistics

To show that the random values representing the dice occur with approximately equal likelihood, let's allow the user to roll 12 dice at a time and keep statistics showing the number of times each face occurs and the percentage of the time each face is rolled (Fig. 9.6). This example is similar to the one in Fig. 9.5, so we'll focus only on the new features.

Script Variables

Lines 22–28 declare and initialize counter variables to keep track of the number of times each of the six die values appears and the total number of dice rolled. Because these variables are declared outside the script's functions, they're accessible to all the functions in the script.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.6: RollDice.html -->
4  <!-- Rolling 12 dice and displaying frequencies. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Die Rolling Frequencies</title>
9      <style type = "text/css">
```

```
      img         { margin-right: 10px; }
      table       { width: 200px;
                    border-collapse: collapse;
                    background-color: lightblue; }
      table, td, th { border: 1px solid black;
                    padding: 4px;
                    margin-top: 20px; }
      th          { text-align: left;
                    color: white;
                    background-color: darkblue; }
    </style>
    <script>
      var frequency1 = 0;
      var frequency2 = 0;
      var frequency3 = 0;
      var frequency4 = 0;
      var frequency5 = 0;
      var frequency6 = 0;
      var totalDice = 0;

      // register button event handler
      function start()
      {
        var button = document.getElementById( "rollButton" );
        button.addEventListener( "click", rollDice, false );
      } // end function start

      // roll the dice
      function rollDice()
      {
        var face;  // face rolled

        // loop to roll die 12 times
        for ( var i = 1; i <= 12; ++i )
        {
          face = Math.floor( 1 + Math.random() * 6 );
```

```
      tallyRolls( face ); // increment a frequency counter
      setImage( i, face ); // display appropriate die image
      ++totalDice; // increment total
    } // end die rolling loop

    updateFrequencyTable();
  } // end function rollDice

  // increment appropriate frequency counter
  function tallyRolls( face )
  {
    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 function tallyRolls

  // set image source for a die
  function setImage( dieNumber, face )
```

```
{
  var dieImg = document.getElementById( "die" + dieNumber );
  dieImg.setAttribute( "src", "die" + face +".png" );
  dieImg.setAttribute( "alt", "die with " + face +" spot(s)" );
} // end function setImage

// update frequency table in the page
function updateFrequencyTable()
{
  var tableDiv = document.getElementById( "frequencyTableDiv" );

  tableDiv.innerHTML = "<table>" +
    "<caption>Die Rolling Frequencies</caption>" +
    "<thead><th>Face</th><th>Frequency</th>" +
    "<th>Percent</th></thead>" +
    "<tbody><tr><td>1</td><td>" + frequency1 + "</td><td>" +
    formatPercent(frequency1 / totalDice) + "</td></tr>" +
    "<tr><td>2</td><td>" + frequency2 + "</td><td>" +
     formatPercent(frequency2 / totalDice)+ "</td></tr>" +
     "<tr><td>3</td><td>" + frequency3 + "</td><td>" +
     formatPercent(frequency3 / totalDice) + "</td></tr>" +
     "<tr><td>4</td><td>" + frequency4 + "</td><td>" +
     formatPercent(frequency4 / totalDice) + "</td></tr>" +
     "<tr><td>5</td><td>" + frequency5 + "</td><td>" +
     formatPercent(frequency5 / totalDice) + "</td></tr>" +
     "<tr><td>6</td><td>" + frequency6 + "</td><td>" +
     formatPercent(frequency6 / totalDice) + "</td></tr>" +
     "</tbody></table>";
} // end function updateFrequencyTable

// format percentage
function formatPercent( value )
{
  value *= 100;
  return value.toFixed(2);
} // end function formatPercent
```

```
      window.addEventListener( "load", start, false );
    </script>
  </head>
  <body>
    <p><img id = "die1" src = "blank.png" alt = "die 1 image">
      <img id = "die2" src = "blank.png" alt = "die 2 image">
      <img id = "die3" src = "blank.png" alt = "die 3 image">
      <img id = "die4" src = "blank.png" alt = "die 4 image">
      <img id = "die5" src = "blank.png" alt = "die 5 image">
      <img id = "die6" src = "blank.png" alt = "die 6 image"></p>
    <p><img id = "die7" src = "blank.png" alt = "die 7 image">
      <img id = "die8" src = "blank.png" alt = "die 8 image">
      <img id = "die9" src = "blank.png" alt = "die 9 image">
      <img id = "die10" src = "blank.png" alt = "die 10 image">
      <img id = "die11" src = "blank.png" alt = "die 11 image">
      <img id = "die12" src = "blank.png" alt = "die 12 image"></p>
    <form action = "#">
      <input id = "rollButton" type = "button" value = "Roll Dice">
    </form>
    <div id = "frequencyTableDiv"></div>
  </body>
</html>
```

Fig. 9.6. Rolling 12 dice and displaying frequencies.

Function `rollDice`

As in Fig. 9.5, when the user presses the **Roll Dice** button, function `rollDice` (lines 38–52) is called. This function calls functions `tally-Rolls` and `setImage` for each of the twelve `img` elements in the document (lines 123–134), then calls function `updateFrequencyTable` to display the number of times each die face appeared and the percentage of total dice rolled.

Function `tallyRolls`

Function `tallyRolls` (lines 55–78) contains a `switch` statement that uses the randomly chosen `face` value 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. No `default` case is provided in this `switch` statement, because the statement in line 45 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 an elegant way to replace the entire `switch` statement in this program with a *single* line of code.

Function `setImage`

Function `setImage` (lines 81–86) sets the image source and `alt` text for the specified `img` element.

Function `updateFrequencyTable`

Function `updateFrequencyTable` (lines 89–110) creates a table and places it in the `div` element at line 131 in the document's `body`. Line 91 gets the object representing that `div` and assigns it to the local variable `table-Div`. Lines 93–109 build a string representing the table and assign it to the `tableDiv` object's **`innerHTML` property**, which places HTML5 code into the element that `tableDiv` represents and allows the browser to render that HTML5 in the element. Each time we assign HTML markup to an element's `innerHTML` property, the `tableDiv`'s content is completely replaced with the content of the string.

Function `formatPercent`

Function `updateFrequencyTable` calls function `formatPercent` (lines 113–117) to format values as percentages with two digits to the right of the decimal point. The function simply multiplies the value it receives by 100, then returns the value after calling its `toFixed` method with the argument 2, so that the number has two digits of precision to the right of the decimal point.

Generalized Scaling and Shifting of Random Values

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).

9.6. Example: Game of Chance; Introducing the HTML5 `audio` and `video` Elements

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. 9.7 simulates the game of craps. Note that the player must roll *two* dice on the first and all subsequent rolls. When you load this document, you can click the link at the top of the page to browse a separate document (Fig. 9.8) containing a video that explains the basic rules of the game. To start a game, click the **Play** button. A message below the button displays the game's status after each roll. If you don't win or lose on the first roll, click the **Roll** button to roll again. [*Note:* This example uses some features that, at the time of this writing, worked only in Chrome, Safari and Internet Explorer 9.]

The `body` Element

Before we discuss the script code, we discuss the `body` element (lines 150–177) of this document. The elements in the `body` are used extensively in the script.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.7: Craps.html -->
4  <!-- Craps game simulation. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Craps Game Simulation</title>
9      <style type = "text/css">
10        p.red  { color: red }
11        img    { width: 54px; height: 54px; }
12        div    { border: 5px ridge royalblue;
13                 padding: 10px; width: 120px;
14                 margin-bottom: 10px; }
15        .point { margin: 0px; }
16      </style>
17      <script>
18        // variables used to refer to page elements
19        var pointDie1Img; // refers to first die point img
20        var pointDie2Img; // refers to second die point img
21        var rollDie1Img; // refers to first die roll img
22        var rollDie2Img; // refers to second die roll img
23        var messages; // refers to "messages" paragraph
```

```
var playButton; // refers to Play button
var rollButton; // refers to Roll button
var dicerolling; // refers to audio clip for dice

// other variables used in program
var myPoint; // point if no win/loss on first roll
var die1Value; // value of first die in current roll
var die2Value; // value of second die in current roll

// starts a new game
function startGame()
{
  // get the page elements that we'll interact with
  dicerolling = document.getElementById( "dicerolling" );
  pointDie1Img = document.getElementById( "pointDie1" );
  pointDie2Img = document.getElementById( "pointDie2" );
  rollDie1Img = document.getElementById( "rollDie1" );
  rollDie2Img = document.getElementById( "rollDie2" );
  messages = document.getElementById( "messages" );
  playButton = document.getElementById( "play" );
  rollButton = document.getElementById( "roll" );

  // prepare the GUI
  rollButton.disabled = true; // disable rollButton
  setImage( pointDie1Img ); // reset image for new game
  setImage( pointDie2Img ); // reset image for new game
  setImage( rollDie1Img ); // reset image for new game
  setImage( rollDie2Img ); // reset image for new game

  myPoint = 0; // there is currently no point
  firstRoll(); // roll the dice to start the game
} // end function startGame

// perform first roll of the game
function firstRoll()
{
```

```
    var sumOfDice = rollDice(); // first roll of the dice

    // determine if the user won, lost or must continue rolling
    switch (sumOfDice)
    {
      case 7: case 11: // win on first roll
        messages.innerHTML =
          "You Win!!! Click Play to play again.";
        break;
      case 2: case 3: case 12: // lose on first roll
        messages.innerHTML =
          "Sorry. You Lose. Click Play to play again.";
        break;
      default: // remember point
        myPoint = sumOfDice;
        setImage( pointDie1Img, die1Value );
        setImage( pointDie2Img, die2Value );
        messages.innerHTML = "Roll Again!";
        rollButton.disabled = false; // enable rollButton
        playButton.disabled = true; // disable playButton
        break;
    } // end switch
  } // end function firstRoll

  // called for subsequent rolls of the dice
  function rollAgain()
  {
    var sumOfDice = rollDice(); // subsequent roll of the dice

    if (sumOfDice == myPoint)
    {
      messages.innerHTML =
        "You Win!!! Click Play to play again.";
      rollButton.disabled = true; // disable rollButton
      playButton.disabled = false; // enable playButton
    } // end if
```

```
      else if (sumOfDice == 7) // craps
      {
         messages.innerHTML =
            "Sorry. You Lose. Click Play to play again.";
         rollButton.disabled = true; // disable rollButton
         playButton.disabled = false; // enable playButton
      } // end else if
   } // end function rollAgain

   // roll the dice
   function rollDice()
   {
      dicerolling.play(); // play dice rolling sound

   // clear old die images while rolling sound plays
      die1Value = NaN;
      die2Value = NaN;
      showDice();

      die1Value = Math.floor(1 + Math.random() * 6);
      die2Value = Math.floor(1 + Math.random() * 6);
      return die1Value + die2Value;
   } // end function rollDice

   // display rolled dice
   function showDice()
   {
      setImage( rollDie1Img, die1Value );
      setImage( rollDie2Img, die2Value );
   } // end function showDice

   // set image source for a die
   function setImage( dieImg, dieValue )
   {
      if ( isFinite( dieValue) )
         dieImg.src = "die" + dieValue + ".png";
```

```
          else
             dieImg.src = "blank.png";
       } // end function setImage

       // register event liseners
       function start()
       {
          var playButton = document.getElementById( "play" );
          playButton.addEventListener( "click", startGame, false );
          var rollButton = document.getElementById( "roll" );
          rollButton.addEventListener( "click", rollAgain, false );
          var diceSound = document.getElementById( "dicerolling" );
          diceSound.addEventListener( "ended", showDice, false );
       } // end function start

       window.addEventListener( "load", start, false );
    </script>
 </head>
 <body>
    <audio id = "dicerolling" preload = "auto">
       <source src = "http://test.deitel.com/dicerolling.mp3"
          type = "audio/mpeg">
       <source src = "http://test.deitel.com/dicerolling.ogg"
          type = "audio/ogg">
       Browser does not support audio tag</audio>
    <p><a href = "CrapsRules.html">Click here for a short video
       explaining the basic Craps rules</a></p>
    <div id = "pointDiv">
       <p class = "point">Point is:</p>
       <img id = "pointDie1" src = "blank.png"
          alt = "Die 1 of Point Value">
       <img id = "pointDie2" src = "blank.png"
          alt = "Die 2 of Point Value">
    </div>
    <div class = "rollDiv">
       <img id = "rollDie1" src = "blank.png"
```

```
          alt = "Die 1 of Roll Value">
        <img id = "rollDie2" src = "blank.png"
          alt = "Die 2 of Roll Value">
      </div>
      <form action = "#">
        <input id = "play" type = "button" value = "Play">
        <input id = "roll" type = "button" value = "Roll">
      </form>
      <p id = "messages" class = "red">Click Play to start the game</p>
   </body>
</html>
```

Fig. 9.7. Craps game simulation.

The HTML5 `audio` Element

Line 151–156 define an HTML5 **`audio` element**, which is used to embed audio into a web page. We specify an `id` for the element, so that we can *programmatically* control when the audio clip plays, based on the user's interactions with the game. Setting the **`preload` attribute** to `"auto"` indicates to the browser that it should consider downloading the audio clip so that it's ready to be played when the game needs it. Under certain conditions the browser can ignore this attribute—for example, if the user is on a low-bandwidth Internet connection.

Not all browsers support the same audio file formats, but most support MP3, OGG and/or WAV format. All of the browsers we tested in this book support MP3, OGG or both. For this reason, nested in the `audio` element are *two* **`source` elements** specifying the locations of the audio clip in MP3 and OGG formats, respectively. Each `source` element specifies a `src` and a `type` attribute. The `src` attribute specifies the location of the audio clip. The `type` attribute specifies the clip's MIME type—`audio/mpeg` for the MP3 clip and `audio/ogg` for the OGG clip (WAV would be `audio/x-wav`; MIME types for these and other formats can be found online). When a web browser that supports the `audio` element encounters the `source` elements, it will chose the first audio source that represents one of the browser's supported formats. If the browser does not support the `audio` element, the text in line 156 will be displayed.

We used the online audio-file converter at

media.io

to convert our audio clip to other formats. Many other online and downloadable file converters are available on the web.

The Link to the `CrapsRules.html` Page

Lines 157–158 display a link to a separate web page in which we use an HTML5 `video` element to display a short video that explains the basic rules for the game of Craps. We discuss this web page at the end of this section.

`pointDiv` and `rollDiv`

The `div` elements at lines 159–171 contain the `img` elements in which we display die images representing the user's point and the current roll of the dice, respectively. Each `img` element has an `id` attribute so that we can interact with it programmatically. Because the `id` attribute, if specified, must have a unique value, JavaScript can reliably refer to any single element via its `id` attribute.

The `form` Element

Lines 172–175 define an HTML5 `form` element containing two `button` `input` elements. Each `button`'s `click` event handler indicates the action to take when the user clicks the corresponding button. In this example, clicking the **Play** button causes a call to function `startGame` and clicking the **Roll** button causes a call to function `rollAgain`. Initially, the **Roll** button is disabled, which prevents the user from initiating an event with this button.

The `p` Element

Line 176 defines a `p` element in which the game displays status messages to the user.

The Script Variables

Lines 19–31 create variables that are used throughout the script. Recall that because these are declared outside the script's functions, they're accessible to all the functions in the script. The variables in lines 19–26 are used to interact with various page elements in the script. Variable `myPoint` (line 29) stores the point if the player does not win or lose on the first roll. Variables `die1Value` and `die2Value` keep track of the die values for the current roll.

Function `startGame`

The user clicks the **Play** button to start the game and perform the first roll of the dice. This event invokes function `startGame` (lines 34–55), which takes no arguments. Line 37–44 use the `document` object's `getElementById` method to get the page elements that the script interacts with programmatically.

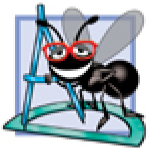
The **Roll** button should be enabled *only* if the user does not win or lose on the first roll. For this reason, line 47 disables the **Roll** button by setting its **disabled property** to `true`. Each `input` element has a `disabled` property.

Lines 48–51 call function `setImage` (defined in lines 128–134) to display the image `blank.png` for the `img` elements in the `pointDiv` and `rollDiv`. We'll replace `blank.png` with die images throughout the game as necessary.

Finally, line 53 sets `myPoint` to `0`, because there can be a point value only *after* the first roll of the dice, and line 54 calls method `firstRoll` (defined in lines 58–82) to perform the first roll of the dice.

Function `firstRoll`

Function `firstRoll` (lines 58–82) calls function `rollDice` (defined in lines 106–118) to roll the dice and get their sum, which is stored in the local variable `sumOfDice`. Because this variable is defined *inside* the `firstRoll` function, it's accessible only inside that function. Next, the `switch` statement (lines 63–81) determines whether the game is won or lost, or whether it should continue with another roll. If the user won or lost, lines 66–67 or 70–71 display an appropriate message in the `messages` paragraph (`p`) element with the object's `innerHTML` property. After the first roll, if the game is not over, the value of local variable `sumOfDice` is saved in `myPoint` (line 74), the images for the rolled die values are displayed (lines 75–76) in the `pointDiv` and the message `"Roll Again!"` is displayed in the displayed in the `messages` paragraph (`p`) element. Also, lines 78–79 enable the **Roll** button and disable the **Play** button, respectively. Function `firstRoll` takes no arguments, so it has an empty parameter list.



SOFTWARE ENGINEERING OBSERVATION 9.3

Variables 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'll be entirely separate variables in memory.

ERROR-PREVENTION TIP 9.1

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

Function `rollAgain`

The user clicks the **Roll** button to continue rolling if the game was not won or lost on the first roll. Clicking this button calls the `rollAgain` function (lines 85–103), which takes no arguments. Line 87 calls function `rollDice` and stores the sum locally in `sumOfDice`, then lines 89–102 determine whether the user won or lost on the current roll, display an appropriate message in the `messages` paragraph (`p`) element, disable the **Roll** and enable the **Play** button. In either case, the user can now click **Play** to play another game. If the user did not win or lose, the program waits for the user to click the **Roll** button again. Each time the user clicks **Roll**, function `rollAgain` executes and, in turn, calls the `rollDice` function to produce a new value for `sumOfDice`.

Function `rollDice`

We define a function `rollDice` (lines 106–118), which takes no arguments, to roll the dice and compute their sum. Function `rollDice` is defined once but is called from lines 60 and 87 in the program. The function returns the sum of the two dice (line 117). Line 108 *plays* the audio clip declared at lines 151–165 by calling its **`play` method**, which plays the clip once. As you'll soon see, we use the `audio` element's **ended event**, which occurs when the clip finishes playing, to indicate when to display the new die images. Lines 111–112 set variables `die1Value` and `die2Value` to `NaN` so that the call to `showDice` (line 113) can display the `blank.png` image while the dice sound is playing. Lines 115–116 pick two random values in the range 1 to 6 and assign them to the script variables `die1Value` and `die2Value`, respectively.

Function `showDice`

Function `showDice` (lines 121–125) is called when the dice rolling sound finishes playing. At this point, lines 123–124 display the die images representing the die values that were rolled in function `rollDice`.

Function `setImage`

Function `setImage` (lines 128–134) takes two arguments—the `img` element that will display an image and the value of a die to specify which die image to display. You might have noticed that we called this function with *one* argument in lines 48–51 and with *two* arguments in lines 75–76 and 123–124. If you call `setImage` with only one argument, the second parameter's value will be *undefined*. In this case, we display the image `blank.png` (line 133). Line 130 uses global JavaScript function **`isFinite`** to determine whether the parameter `dieValue` contains a number—if it does, we'll display the die image that corresponds to that number (line 131). Function `isFinite` returns `true` only if its argument is a valid number in the range supported by JavaScript. You can learn more about JavaScript's valid numeric range in Section 8.5 of the JavaScript standard:

www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Function `start`

Function `start` (lines 137–145) is called when the window's `load` event occurs to register `click` event handlers for this examples two `button` s (lines 139–142) and for the `ended` event of the `audio` element (lines 143–144).

Program-Control Mechanisms

Note the use of the various program-control mechanisms. The craps program uses five functions— `startGame` , `firstRoll` , `rollAgain` , `rollDice` and `setImage` —and the `switch` and nested `if` ... `else` statements. Also, note the use of multiple `case` labels in the `switch` statement to execute the same statements (lines 65 and 69). In the exercises at the end of this chapter, we investigate additional characteristics of the game of craps.

`CrapsRules.html` and the HMTL5 `video` Element

When the user clicks the hyperlink in `Craps.html` (Fig. 9.7, lines 157–158), the `CrapsRules.html` is displayed in the browser. This page consists of a link back to `Craps.html` (Fig. 9.8, line 11) and an HTML5 **`video` ele-**

ment (lines 12–25) that displays a video explaining the basic rules for the game of Craps.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.8: CrapsRules.html -->
4  <!-- Web page with a video of the basic rules for the dice game Craps. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Craps Rules</title>
9    </head>
10   <body>
11     <p><a href = "Craps.html">Back to Craps Game</a></p>
12     <video controls>
13       <source src = "CrapsRules.mp4" type = "video/mp4">
14       <source src = "CrapsRules.webm" type = "video/webm">
15       A player rolls two dice. Each die has six faces that contain
16       one, two, three, four, five and six spots, respectively. The
17       sum of the spots on the two upward faces is calculated. If the
18       sum is 7 or 11 on the first throw, the player wins. If the sum
19       is 2, 3 or 12 on the first throw (called "craps"), the player
20       loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or
21       10 on the first throw, that sum becomes the player's "point."
22       To win, you must continue rolling the dice until you "make your
23       point" (i.e., roll your point value). You lose by rolling a 7
24       before making the point.
25     </video>
26   </body>
27 </html>
```

Fig. 9.8. Web page that displays a video of the basic rules for the dice game Craps.

The `video` element's **`controls` attribute** indicates that we'd like the video player in the browser to display controls that allow the user to control video playback (e.g., play and pause). As with audio, not all browsers support the same video file formats, but most support MP4, OGG and/or WebM formats. For this reason, nested in the `video` element are two `source` elements specifying the locations of this example's video clip in MP4 and WebM formats. The `src` attribute of each specifies the location of the video. The `type` attribute specifies the video's MIME type—`video/mp4` for the MP4 video and `video/webm` for the WebM video (MIME types for these and other formats can be found online). When a web browser that supports the `video` element encounters the `source` elements, it will choose the first video source that represents one of the browser's supported formats. If the browser does not support the `video` element, the text in lines 15–24 will be displayed.

We used the downloadable video converter at

www.mirovideoconverter.com

to convert our video from MP4 to WebM format. For more information on the HTML5 `audio` and `video` elements, visit:

dev.opera.com/articles/view/everything-you-need-to-know-about-html5-video-and-audio/

9.7. 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 page's script(s) 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 function's terminating right brace (`}`). 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 9.2

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. 9.9 demonstrates the **scope rules** that resolve conflicts between global variables and local variables of the same name. Once again, we use the `window`'s `load` event (line 53), which calls the function `start` when the HTML5 document is completely loaded into the browser window. In this example, we build an `output` string (declared at line 14) that is displayed at the end of function `start`'s execution.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.9: scoping.html -->
4  <!-- Scoping example. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Scoping Example</title>
9      <style type = "text/css">
10         p      { margin: 0px; }
11         p.space { margin-top: 10px; }
12      </style>
13      <script>
14         var output; // stores the string to display
15         var x = 1; // global variable
16
17         function start()
18         {
19            var x = 5; // variable local to function start
20
21            output = "<p>local x in start is " + x + "</p>";
22
23            functionA(); // functionA has local x
24            functionB(); // functionB uses global variable x
25            functionA(); // functionA reinitializes local x
26            functionB(); // global variable x retains its value
27
28            output += "<p class='space'>local x in start is " + x +
```

```
            "</p>";
          document.getElementById( "results" ).innerHTML = output;
        } // end function start

        function functionA()
        {
          var x = 25; // initialized each time functionA is called

          output += "<p class='space'>local x in functionA is " + x +
            " after entering functionA</p>";
          ++x;
          output += "<p>local x in functionA is " + x +
            " before exiting functionA</p>";
        } // end functionA

        function functionB()
        {
          output += "<p class='space'>global variable x is " + x +
            " on entering functionB";
          x *= 10;
          output += "<p>global variable x is " + x +
            " on exiting functionB</p>";
        } // end functionB

        window.addEventListener( "load", start, false );
      </script>
    </head>
    <body>
      <div id = "results"></div>
    </body>
</html>
```

Fig. 9.9. Scoping example.

Global variable `x` (line 15) is declared and initialized to 1. This global variable is *hidden* in any block (or function) that declares a variable named `x`. Function `start` (lines 17–31) declares a local variable `x` (line 19) and initializes it to `5`. Line 21 creates a paragraph element containing `x`'s value as a string and assigns the string to the global variable `output` (which is displayed later). In the sample output, this shows that the global variable `x` is *hidden* in `start`.

The script defines two other functions— `functionA` and `functionB` — each taking no arguments and returning nothing. Each function is called twice from function `start` (lines 23–26). Function `functionA` defines local variable `x` (line 35) and initializes it to `25`. When `functionA` is called, the variable's value is placed in a paragraph element and appended to variable `output` to show that the global variable `x` is *hidden* in `functionA`; then the variable is incremented and appended to `output` again before the function exits. 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's value is placed in a paragraph element and appended to variable `output`, then it's multiplied by `10` and appended to variable `output` again before the function exits. 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, lines 28–29 append the value of local variable `x` in `start` to variable `output`, 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. Line 30 uses the `document` object's `getElementById` method to get the `results` `div` element (line 57), then assigns variable `output`'s value to the element's `innerHTML` property, which renders the HTML in variable `output` on the page.

9.8. JavaScript Global Functions

JavaScript provides nine standard global functions. We've already used `parseInt`, `parseFloat` and `isFinite`. Some of the global functions are summarized in Fig. 9.10.

Fig. 9.10. JavaScript global functions.

The global functions in Fig. 9.10 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. 9.10. Because global functions and user-defined functions are part of the `Global` object, some JavaScript programmers refer to these functions as methods. You do not need to use the `Global` object directly—JavaScript references it for you. For information on JavaScript's other global functions, see Section 15.1.2 of the ECMAScript Specification:

www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

9.9. Recursion

The programs we've 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 computer science topic. 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 of 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've performed to this point.

As an example of these concepts at work, let's 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** (non-recursively) 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. 9.11. Figure 9.11(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion. Figure 9.11(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Fig. 9.11. Recursive evaluation of 5!.

Figure 9.12 uses recursion to calculate and print the factorials of the integers 0 to 10. The recursive function `factorial` first tests (line 27) 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 30 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)`.

```
<!DOCTYPE html>

<!-- Fig. 9.12: FactorialTest.html -->
<!-- Factorial calculation with a recursive function. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Recursive Factorial Function</title>
    <style type = "text/css">
      p      { margin: 0px; }
    </style>
    <script>
      var output = ""; // stores the output

      // calculates factorials of 0 - 10
      function calculateFactorials()
      {
        for ( var i = 0; i <= 10; ++i )
          output += "<p>" + i + "! = " + factorial( i ) +"</p>";

        document.getElementById( "results" ).innerHTML = output;
      } // end function calculateFactorials

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

      window.addEventListener( "load", calculateFactorials, false );
    </script>
  </head>
  <body>
```

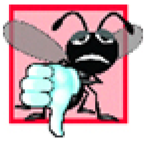
```
37       <h1>Factorials of 0 to 10</h1>
38       <div id = "results"></div>
39     </body>
40  </html>
```

Fig. 9.12. Factorial calculation with a recursive function.

Function `factorial` (lines 25–31) receives as its argument the value for which to calculate the factorial. As can be seen in the screen capture in Fig. 9.12, factorial values become large quickly.

COMMON PROGRAMMING ERROR 9.3

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 (non-recursive) solution.



ERROR-PREVENTION TIP 9.2

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.

9.10. 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`).

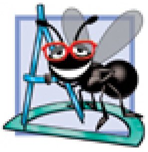
Both iteration and recursion involve repetition: Iteration explicitly uses a repetition statement; recursion achieves repetition through repeated function calls.

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 9.4

Any problem that can be solved recursively can also be solved iteratively (non-recursively). 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's 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-critical situations. Recursive calls take time and consume additional memory.

In addition to the factorial function example (Fig. 9.12), we also provide recursion exercises—raising an integer to an integer power (Exercise 9.29) and "What does the following function do?" (Exercise 9.30). Also, Fig. 15.25 uses recursion to traverse an XML document tree.

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 (p. 279). This technique is called divide and conquer (p. 279).

Section 9.2 Program Modules in JavaScript

- JavaScript programs are written by combining new functions (p. 279) that the programmer writes with “prepackaged” functions and objects available in JavaScript.

- The term method (p. 279) 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.

- 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 (p. 279). The actual statements defining the function are written only once and are hidden from other functions.

- Functions are invoked (p. 280) 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 (p. 280) but require the name of the object to which the method belongs and a dot preceding the method name.

- Function arguments (p. 280) may be constants, variables or expressions.

Section 9.3 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
  }

- Each function should perform a single, well-defined task, and the name of the function should express that task effectively. This promotes software reusability (p. 285).

- 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 when the statement `return;` is executed. If the function does return a result, the statement `return` *expression*; returns the value of *expression* to the caller.

Section 9.4 Notes on Programmer-Defined Functions

- All variables declared with the keyword var in function definitions are local variables (p. 285)—this means that they can be accessed only in the function in which they're defined.

- A function's parameters (p. 285) 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's packaged as a function can be executed from several locations in a program by calling the function.

Section 9.5 Random Number Generation

- Method `random` generates a floating-point value from 0.0 up to, but not including, 1.0.

- JavaScript can execute actions in response to the user's interaction with an element in an HTML5 form. This is referred to as GUI event handling (p. 290).

- An HTML5 element's `click` event handler (p. 289) indicates the action to take when the user of the HTML5 document clicks on the element.

- In event-driven programming (p. 290), the user interacts with an element, the script is notified of the event (p. 290) and the script processes the event. The user's interaction with the GUI "drives" the program. The function that's called when an event occurs is known as an event-handling function or event handler.

- The `getElementById` method (p. 290), given an `id` as an argument, finds the HTML5 element with a matching `id` attribute and returns a JavaScript object representing the element.

- The scaling factor (p. 296) determines the size of the range. The shifting value (p. 296) is added to the result to determine where the range begins.

Section 9.6 Example: Game of Chance; Introducing the HTML5 `audio` and `video` Elements

- An HTML5 `audio` element (p. 301) embeds audio into a web page. Setting the `preload` attribute (p. 301) to `"auto"` indicates to the browser that it should consider downloading the audio clip so that it's ready to be played.

- Not all browsers support the same audio file formats, but most support MP3, OGG and/or WAV format. For this reason, you can use `source` elements (p. 301) nested in the `audio` element to specify the locations of an audio clip in different formats. Each `source` element specifies a `src` and a `type` attribute. The `src` attribute specifies the location of the audio clip. The `type` attribute specifies the clip's MIME type.

- When a web browser that supports the `audio` element encounters the `source` elements, it chooses the first audio source that represents one of the browser's supported formats.

- When interacting with an `audio` element from JavaScript, you can use the `play` method (p. 303) to play the clip once.

- Global JavaScript function `isFinite` (p. 304) returns true only if its argument is a valid number in the range supported by JavaScript.

- The HTML5 `video` element (p. 304) embeds a video in a web page.

- The `video` element's `controls` attribute (p. 305) indicates that the video player in the browser should display controls that allow the user to control video playback.

- As with audio, not all browsers support the same video file formats, but most support MP4, OGG and/or WebM formats. For this reason, you can use `source` elements nested in the `video` element to specify the locations of a video clip's multiple formats.

Section 9.7 Scope Rules

- Each identifier in a program has a scope (p. 306). 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 HTML5 document, p. 306) are accessible in any part of a script and are said to have global scope (p. 306). Thus every function in the script can potentially use the variables.

- Identifiers declared inside a function have function (or local) scope (p. 306) 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.

Section 9.8 JavaScript Global Functions

- JavaScript provides several global functions as part of a `Global` object (p. 309). 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 listed in Fig. 9.10.

- You do not need to use the `Global` object directly; JavaScript uses it for you.

Section 9.9 Recursion

- A recursive function (p. 309) 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, p. 310) 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 (p. 310).

- 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 9.10 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 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.

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's 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 program in Fig. 9.13, 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`.

```
<!DOCTYPE html>

<!-- Exercise 9.2: cube.html -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Scoping</title>
    <script>
      var x;

      function output()
      {
        for ( x = 1; x <= 10; x++ )
          document.writeln( "<p>" + cube( x ) + "</p>" );
      } // end function output

      function cube( y )
      {
        return y * y * y;
      } // end function cube

      window.addEventListener( "load", output, false );
   </script>
   </head><body></body>
</html>
```

Fig. 9.13. Scope exercise.

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

a. Programmer-defined functions, global variables and JavaScript's global functions are all part of the________object.

b. Function ________ determines whether its argument is or is not a number.

c. Function ________ takes a string argument and returns a string in which all spaces, punctuation, accent characters and any other character that's not in the ASCII character set are encoded in a hexadecimal format.

d. Function ________ takes a string argument representing JavaScript code to execute.

e. Function ________ takes a string as its argument and returns a string in which all characters that were previously encoded with `escape` are decoded.

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

a. An identifier's ________ is the portion of the program in which it can be used.

b. The three ways to return control from a called function to a caller are, ________ and ________.

c. The ________ function is used to produce random numbers.

d. Variables declared in a block or in a function's parameter list are of ________ scope.

9.5 Locate the error in each of the following program segments and explain how to correct it:

a.

```
method g()
{
  document.writeln( "Inside method g" );
}
```

b.

```
// This function should return the sum of its arguments
function sum( x, y )
{
  var result;
  result = x + y;
}
```

c.

```
function f( a );
{
  document.writeln( a );
}
```

9.6 Write a complete JavaScript program to prompt the user for the radius of a sphere, then call function `sphereVolume` to calculate and display the volume of the sphere. Use the statement

```
volume = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 );
```

to calculate the volume. The user should enter the radius in an HTML5 `input` element of type `"number"` in a `form`. Give the `input` element the `id` value `"inputField"`. You can use this `id` with the `document` object's `getElementById` method to get the element for use in the script. To access the string in the `inputField`, use its `value` property as in `inputField.value`, then convert the string to a number using `parseFloat`. Use an `input` element of type `"button"` in the `form` to allow the user to initiate the calculation. [*Note:* In HTML5, `input` elements of type `"number"` have a property named `valueAsNumber` that enables a script to get the floating-point number in the `input` element without having to convert it from a string to a number using `parseFloat`. At the time of this writing, `valueAsNumber` was not supported in all browsers.]

Answers to Self-Review Exercises

9.1

a. functions.

b. function call.

c. local variable.

d. `return`.

e. `function`.

9.2

a. global scope.

b. function scope.

c. global scope.

d. global scope.

9.3

a. `Global`.

b. `isNaN`.

c. `escape`.

d. `eval`.

e. `unescape`.

9.4

a. scope.

b. `return;` or `return` *expression* `;` or encountering the closing right brace of a function.

c. `Math.random`.

d. local.

9.5

a. Error: `method` is not the keyword used to begin a function definition. Correction: Change `method` to `function`.

b. Error: The function is supposed to return a value, but does not.

Correction: Either delete variable `result` and place the statement

return x + y;

in the function or add the following statement at the end of the function body:

return result;

c. Error: The semicolon after the right parenthesis that encloses the parameter list. Correction: Delete the semicolon after the right parenthesis of the parameter list.

9.6 The solution below calculates the volume of a sphere using the radius entered by the user.

```
<!DOCTYPE html>

<!-- Exercise 9.6: volume.html -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Calculating Sphere Volume</title>
    <script>
      function start()
      {
        var button = document.getElementById( "calculateButton" );
        button.addEventListener("click", displayVolume, false );
```

```
         } // end function start

         function displayVolume()
         {
            var inputField = document.getElementById( "radiusField" );
            var radius = parseFloat( inputField.value );
            var result = document.getElementById( "result" );
            result.innerHTML = "Sphere volume is: " + sphereVolume( radius );
         } // end function displayVolume

         function sphereVolume( radius )
         {
            return ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 );
         } // end function sphereVolume

         window.addEventListener( "load", start, false );
      </script>
   </head>
   <body>
      <form action = "#">
         <p><label>Radius:
            <input id = "radiusField" type = "number"></label>
            <input id = "calculateButton" type = "button" value = "Calculate"></p>
      </form>
      <p id = "result"></p>
   </body>
</html>
```

Exercises

9.7 Write a script that uses a `form` to get the radius of a circle from the user, then calls the function `circleArea` to calculate the area of the circle and display the result in a paragraph on the page. To get the number from the `form`, use the techniques shown in Self-Review Exercise 9.6.

9.8 A parking garage charges a $2.00 minimum fee to park for up to three hours. The garage charges an additional $0.50 per hour for each hour *or part thereof* in excess of three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time. Write a script that calculates and displays the parking charges for each customer who parked a car in this garage yesterday. You should use a `form` to input from the user the hours parked for each customer. The program should display the charge for the current customer and should calculate and display the running total of yesterday's receipts. The program should use the function `calculateCharges` to determine the charge for each customer. To get the number from the `form`, use the techniques shown in Self-Review Exercise 9.6.

9.9 Write function `distance` that calculates the distance between two points ($x1$, $y1$) and ($x2$, $y2$). All numbers and return values should be floating-point values. Incorporate this function into a script that enables the user to enter the coordinates of the points through an HTML5 form. To get the numbers from the `form`, use the techniques shown in Self-Review Exercise 9.6.

9.10 Answer each of the following questions:

a. What does it mean to choose numbers "at random"?

b. Why is the `Math.random` function useful for simulating games of chance?

c. Why is it often necessary to scale and/or shift the values produced by `Math.random`?

d. Why is computerized simulation of real-world situations a useful technique?

9.11 Write statements that assign random integers to the variable n in the following ranges:

a. $1 \leq n \leq 2$

b. $1 \leq n100$

c. $0 \leq n \leq 9$

d. $1000 \leq n \leq 1112$

e. **–1** $\leq n \leq 1$

f. **–3** $\leq n \leq 11$

9.12 For each of the following sets of integers, write a single statement that will print a number at random from the set:

a. 2, 4, 6, 8, 10.

b. 3, 5, 7, 9, 11.

c. 6, 10, 14, 18, 22.

9.13 Write a function `integerPower( base, exponent )` that returns the value of *base exponent*

For example, `integerPower( 3, 4 )=3 * 3 * 3 * 3`. Assume that `exponent` and `base` are integers. Function `integerPower` should use a `for` or `while` statement to control the calculation. Incorporate this function into a script that reads integer values from an HTML5 form for `base` and `exponent` and performs the calculation with the `integerPower` function. The HTML5 form should consist of two text fields and a button to initiate the calculation. The user should interact with the program by typing numbers in both text fields, then clicking the button.

9.14 Write a function `multiple` that determines, for a pair of integers, whether the second integer is a multiple of the first. The function should take two integer arguments and return `true` if the second is a multiple of the first, and `false` otherwise. Incorporate this function into a script that inputs a series of pairs of integers (one pair at a time). The HTML5 form should consist of two text fields and a button to initiate the calculation. The user should interact with the program by typing numbers in both text fields, then clicking the button.

9.15 Write a script that inputs integers (one at a time) and passes them one at a time to function `isEven`, which uses the modulus operator to determine whether an integer is even. The function should take an integer argument and return `true` if the integer is even and `false` otherwise. Use sentinel-controlled looping and a `prompt` dialog.

9.16 Write program segments that accomplish each of the following tasks:

a. Calculate the integer part of the quotient when integer `a` is divided by integer `b`.

b. Calculate the integer remainder when integer `a` is divided by integer `b`.

c. Use the program pieces developed in parts (a) and (b) to write a function `displayDigits` that receives an integer between `1` and `99999` and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer `4562` should be printed as

4 5 6 2

d. Incorporate the function developed in part (c) into a script that inputs an integer from a `prompt` dialog and invokes `displayDigits` by passing to the function the integer entered.

9.17 Implement the following functions:

a. Function `celsius` returns the Celsius equivalent of a Fahrenheit temperature, using the calculation

C = **5.0** / **9.0** * (F - 32);

b. Function `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature, using the calculation

F = **9.0** / **5.0** * C + 32 ;

c. Use these functions to write a script that enables the user to enter either a Fahrenheit or a Celsius temperature and displays the Celsius or Fahrenheit equivalent.

Your HTML5 document should contain two buttons—one to initiate the conversion from Fahrenheit to Celsius and one to initiate the conversion from Celsius to Fahrenheit.

9.18 Write a function `minimum3` that returns the smallest of three floating-point numbers. Use the `Math.min` function to implement `minimum3`. Incorporate the function into a script that reads three values from the user and determines the smallest value.

9.19 An integer is said to be **prime** if it's greater than 1 and divisible only by 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

a. Write a function that determines whether a number is prime.

b. Use this function in a script that determines and prints all the prime numbers between 1 and 10,000. How many of these 10,000 numbers do you really have to test before being sure that you have found all the primes? Display the results in a `<textarea>`.

c. Initially, you might think that *n*/2 is the upper limit for which you must test to see whether a number is prime, but you need go only as high as the square root of *n*. Why? Rewrite the program using the `Math.sqrt` method to calculate the square root, and run it both ways. Estimate the performance improvement.

9.20 Write a function `qualityPoints` that inputs a student's average and returns 4 if the student's average is 90–100, 3 if the average is 80–89, 2 if the average is 70–79, 1 if the average is 60–69 and 0 if the average is lower than 60. Incorporate the function into a script that reads a value from the user.

9.21 Write a script that simulates coin tossing. Let the program toss the coin each time the user clicks the Toss button. Count the number of times each side of the coin appears. Display the results. The program should call a separate function `flip` that takes no arguments and returns `false` for tails and `true` for heads. [*Note:* If the program realistically simulates the coin tossing, each side of the coin should appear approximately half the time.]

9.22 Computers are playing an increasing role in education. Write a program that will help an elementary-school student learn multiplication. Use `Math.random` to produce two positive one-digit integers. It should then display a question such as

How much is 6 times 7?

The student then types the answer into a text field. Your program checks the student's answer. If it's correct, display the string `"Very good!"` and generate a new question. If the answer is wrong, display the string `"No. Please try again."` and let the student try the same question again repeatedly until he or she finally gets it right. A separate function should be used to generate each new question. This function should be called once when the script begins execution and each time the user answers the question correctly.

9.23 The use of computers in education is referred to as **computer-assisted instruction** (CAI). One problem that develops in CAI environments is student fatigue. This problem can be eliminated by varying the computer's dialogue to hold the student's attention. Modify the program in Exercise 9.22 to print one of a variety of comments for each correct answer and each incorrect answer. The set of responses for correct answers is as follows:

Very good!
Excellent!

Nice work!
Keep up the good work!

The set of responses for incorrect answers is as follows:

No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.

Use random number generation to choose a number from 1 to 4 that will be used to select an appropriate response to each answer. Use a `switch` statement to issue the responses.

9.24 More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program in Exercise 9.23 to count the number of correct and incorrect responses typed by the student. After the student answers 10 questions, your program should calculate the percentage of correct responses. If the percentage is lower than 75 percent, display `Please ask your instructor for extra help`, and reset the quiz so another student can try it.

9.25 Write a script that plays a "guess the number" game as follows: Your program chooses the number to be guessed by selecting a random integer in the range 1 to 1000. The script displays the prompt `Guess a number between 1 and 1000` next to a text field. The player types a first guess into the text field and clicks a button to submit the guess to the script. If the player's guess is incorrect, your program should display `Too high. Try again.` or `Too low. Try again.` to help the player "zero in" on the correct answer and should clear the text field so the user can enter the next guess. When the user enters the correct answer, display `Congratulations. You guessed the number!` and clear the text field so the user can play again. [*Note:* The guessing technique employed in this problem is similar to a **binary search**, which we discuss in Chapter 10, JavaScript: Arrays.]

9.26 Modify the program of Exercise 9.25 to count the number of guesses the player makes. If the number is 10 or fewer, display `Either you know the secret or you got lucky!` If the player guesses the number in 10 tries, display `Ahah! You know the secret!` If the player makes more than 10 guesses, display `You should be able to do better!` Why should it take no more than 10 guesses? Well, with each good guess, the player should be able to eliminate half of the numbers. Now show why any number 1 to 1000 can be guessed in 10 or fewer tries.

9.27 (Project) Exercises 9.22 through 9.24 developed a computer-assisted instruction program to teach an elementary-school student multiplication. This exercise suggests enhancements to that program.

a. Modify the program to allow the user to enter a grade-level capability. A grade level of 1 means to use only single-digit numbers in the problems, a grade level of 2 means to use numbers as large as two digits, and so on.

b. Modify the program to allow the user to pick the type of arithmetic problems he or she wishes to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only and 5 means to intermix randomly problems of all these types.

9.28 Modify the craps program in Fig. 9.7 to allow wagering. Initialize variable `bankBalance` to 1000 dollars. Prompt the player to enter a `wager`. Check whether the `wager` is less than or equal to `bankBalance` and, if not, have the user reenter `wager` until a valid `wager` is entered. After a valid `wager` is entered, run one game of craps. If the player wins, increase `bankBalance` by `wager`, and print the new `bankBalance`. If the player loses, decrease `bankBalance` by `wager`, print the new `bankBalance`, check whether `bankBalance` has become zero and, if so, print the message `Sorry. You busted!` As the game progresses, print various messages to create some chatter, such as `Oh, you're going for broke, huh?` or `Aw c'mon, take a chance!` or `You're up big. Now's the time to cash in your chips!`. Implement the chatter as a separate function that randomly chooses the string to display.

9.29 Write a recursive function `power(` *base*, *exponent* `)` that, when invoked, returns $base^{exponent}$

for example, `power( 3, 4 ) = 3 * 3 * 3 * 3`. Assume that *exponent* is an integer greater than or equal to 1. The recursion step would use the relationship

$$base^{exponent} = base \cdot base^{exponent - 1}$$

and the terminating condition occurs when *exponent* is equal to `1`, because

$$base^{1} = base$$

Incorporate this function into a script that enables the user to enter the *base* and *exponent*.

9.30 What does the following function do?

```
// Parameter b must be a positive
// integer to prevent infinite recursion
function mystery( a, b )
{
  if ( b == 1 )
    return a;
  else
    return a + mystery( a, b - 1 );
}
```

Support Sign Out