

Chapter 8

JavaScript: Control Statements, Part 2

Internet & World Wide Web
How to Program, 5/e

8.2 Essentials of Counter-Controlled Repetition

- ▶ Counter-controlled repetition requires
 - name of a control variable
 - initial value of the control variable
 - the increment (or decrement) by which the control variable is modified each time through the loop
 - the condition that tests for the final value of the control variable to determine whether looping should continue

8.2 Essentials of Counter-Controlled Repetition (Cont.)

- ▶ The double-quote character delimits the beginning and end of a string literal in JavaScript
 - it cannot be used in a string unless it is preceded by a \ to create the escape sequence \"

8.2 Essentials of Counter-Controlled Repetition (Cont.)

- ▶ HTML5 allows either single quotes ('') or double quotes ("") to be placed around the value specified for an attribute
- ▶ JavaScript allows single quotes to be placed in a string literal

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 8.1: WhileCounter.html -->
4  <!-- Counter-controlled repetition. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Counter-Controlled Repetition</title>
9          <script>
10
11         var counter = 1; // initialization
12
13         while ( counter <= 7 ) // repetition condition
14         {
15             document.writeln( "<p style = 'font-size: " +
16                 counter + "ex'>HTML5 font size " + counter + "ex</p>" );
17             ++counter; // increment
18         } //end while
19
20         </script>
21     </head><body>
22 </html>
```

Fig. 8.1 | Counter-controlled repetition. (Part 1 of 2.)



Fig. 8.1 | Counter-controlled repetition. (Part 2 of 2.)

8.3 for Repetition Statement

- ▶ **for statement**
 - Specifies each of the items needed for counter-controlled repetition with a control variable
 - Can use a block to put multiple statements into the body
- ▶ If the loop's condition uses a < or > instead of a <= or >=, or vice-versa, it can result in an off-by-one error
- ▶ **for statement header contains three expressions**
 - Initialization
 - Condition
 - Increment Expression
- ▶ The increment expression in the for statement acts like a stand-alone statement at the end of the body of the for statement
- ▶ Place only expressions involving the control variable in the initialization and increment sections of a for statement

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.2: ForCounter.html -->
4 <!-- Counter-controlled repetition with the for statement. -->
5 <html>
6   <head>
7     <meta charset="utf-8">
8     <title>Counter-Controlled Repetition</title>
9     <script>
10
11       // Initialization, repetition condition and
12       // incrementing are all included in the for
13       // statement header.
14       for ( var counter = 1; counter <= 7; ++counter )
15         document.writeln( "<p style = 'font-size: " +
16                           counter + "ex'>HTML5 font size " + counter + "ex</p>" );
17
18     </script>
19   </head><body></body>
20 </html>
```

Fig. 8.2 | Counter-controlled repetition with the for statement.

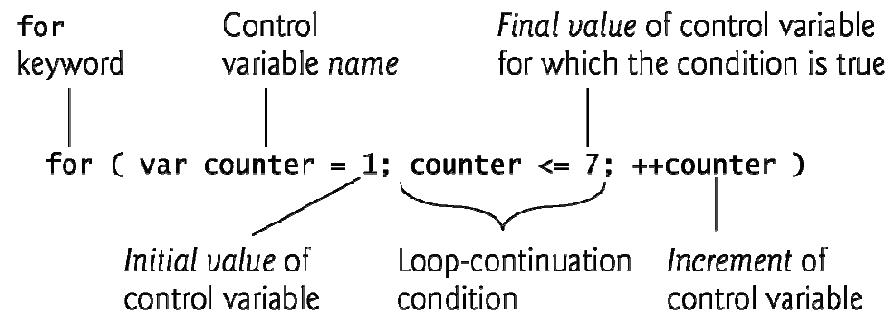


Fig. 8.3 | for statement header components.

8.3 for Repetition Statement (Cont.)

- ▶ The three expressions in the for statement are optional
- ▶ The two semicolons in the for statement are required
- ▶ The initialization, loop-continuation condition and increment portions of a for statement can contain arithmetic expressions

8.3 for Repetition Statement (Cont.)

- ▶ The part of a script in which a variable name can be used is known as the variable's scope
- ▶ The “increment” of a for statement may be negative, in which case it is called a decrement and the loop actually counts downward
- ▶ If the loop-continuation condition initially is false, the body of the for statement is not performed
 - Execution proceeds with the statement following the for statement

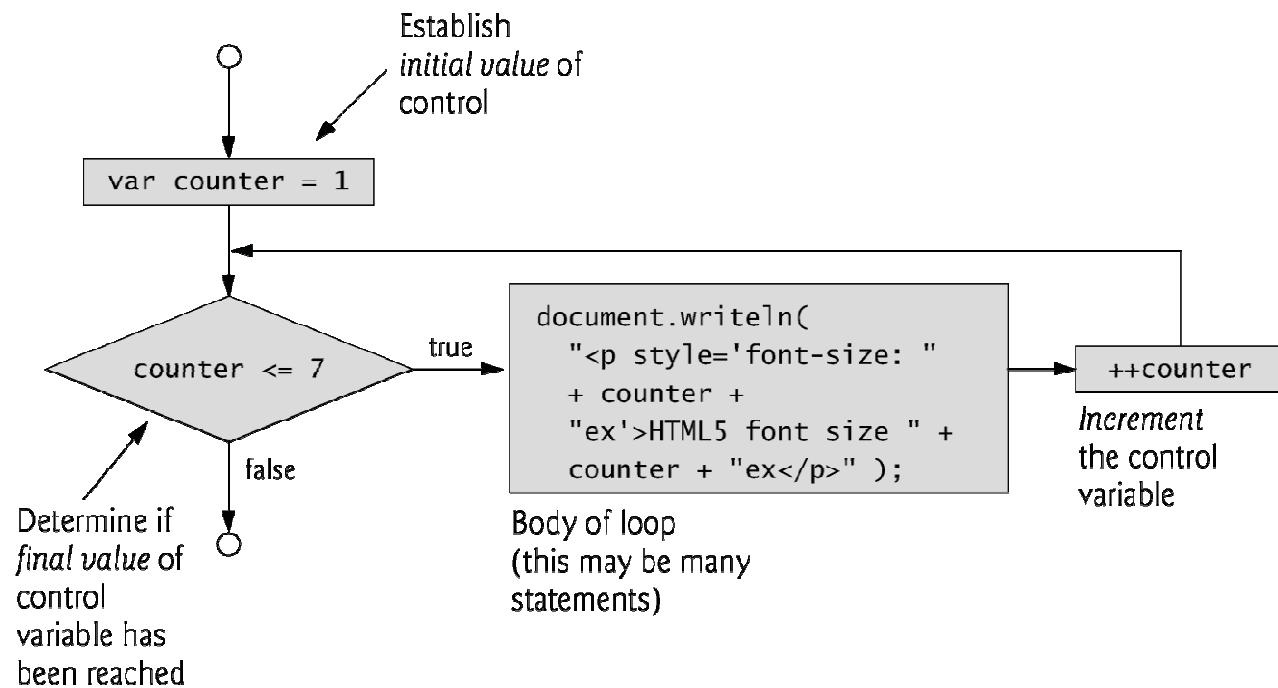


Fig. 8.4 | for repetition statement flowchart.

8.4 Examples Using the for Statement

- ▶ Figure 8.5 uses the for statement to sum the even integers from 2 to 100.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 8.5: Sum.html -->
4  <!-- Summation with the for repetition structure. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Sum the Even Integers from 2 to 100</title>
9          <script>
10
11             var sum = 0;
12
13             for ( var number = 2; number <= 100; number += 2 )
14                 sum += number;
15
16             document.writeln( "The sum of the even integers " +
17                 "from 2 to 100 is " + sum );
18
19         </script>
20     </head><body></body>
21 </html>
```

Fig. 8.5 | Summation with the for repetition structure. (Part 1 of 2.)



Fig. 8.5 | Summation with the for repetition structure. (Part 2 of 2.)

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.6: Interest.html -->
4 <!-- Compound interest calculation with a for loop. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Calculating Compound Interest</title>
9     <style type = "text/css">
10       table { width: 300px;
11             border-collapse: collapse;
12             background-color: lightblue; }
13       table, td, th { border: 1px solid black;
14                     padding: 4px; }
15       th { text-align: left;
16             color: white;
17             background-color: darkblue; }
18       tr.oddrow { background-color: white; }
19     </style>
```

Fig. 8.6 | Compound interest calculation with a for loop. (Part 1 of 4.)

```
20 <script>
21
22     var amount; // current amount of money
23     var principal = 1000.00; // principal amount
24     var rate = 0.05; // interest rate
25
26     document.writeln("<table>"); // begin the table
27     document.writeln(
28         "<caption>Calculating Compound Interest</caption>");
29     document.writeln(
30         "<thead><tr><th>Year</th>" ); // year column heading
31     document.writeln(
32         "<th>Amount on deposit</th>" ); // amount column heading
33     document.writeln( "</tr></thead><tbody>" );
34
```

Fig. 8.6 | Compound interest calculation with a for loop. (Part 2 of 4.)

```
35      // output a table row for each year
36      for ( var year = 1; year <= 10; ++year )
37      {
38          amount = principal * Math.pow( 1.0 + rate, year );
39
40          if ( year % 2 !== 0 )
41              document.writeln( "<tr class='oddrow'><td>" + year +
42                                "</td><td>" + amount.toFixed(2) + "</td></tr>" );
43          else
44              document.writeln( "<tr><td>" + year +
45                                "</td><td>" + amount.toFixed(2) + "</td></tr>" );
46      } //end for
47
48      document.writeln( "</tbody></table>" );
49
50      </script>
51  </head><body></body>
52 </html>
```

Fig. 8.6 | Compound interest calculation with a for loop. (Part 3 of 4.)

A screenshot of a web browser window titled "Calculating Compound Interest". The browser interface includes standard buttons for back, forward, search, and refresh, along with links for "Links" and "Other bookmarks". A "Sync Error" icon is also present. The main content area displays a table titled "Calculating Compound Interest" with two columns: "Year" and "Amount on deposit". The table shows the growth of an initial deposit over 10 years at a 5% annual interest rate, with the amount increasing exponentially.

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

Fig. 8.6 | Compound interest calculation with a for loop. (Part 4 of 4.)

8.4 Examples Using the for Statement (cont.)

- ▶ JavaScript does not include an exponentiation operator
 - Math object's pow method for this purpose.
`Math.pow(x, y)` calculates the value of x raised to the y th power.

8.5 switch Multiple-Selection Statement

- ▶ **switch multiple-selection statement**
 - Tests a variable or expression separately for each of the values it may assume
 - Different actions are taken for each value
- ▶ **CSS property list-style-type**
 - Allows you to set the numbering system for a list
 - Possible values include
 - decimal (numbers—the default)
 - lower-roman (lowercase roman numerals)
 - upper-roman (uppercase roman numerals)
 - lower-alpha (lowercase letters)
 - upper-alpha (uppercase letters)
 - others

8.5 switch Multiple-Selection Statement (Cont.)

- ▶ **switch statement**
 - Consists of a series of case labels and an optional default case
 - When control reaches a switch statement
 - The script evaluates the controlling expression in the parentheses
 - Compares this value with the value in each of the case labels
 - If the comparison evaluates to true, the statements after the case label are executed in order until a break statement is reached
- ▶ The break statement is used as the last statement in each case to exit the switch statement immediately
- ▶ The default case allows you to specify a set of statements to execute if no other case is satisfied
 - Usually the last case in the switch statement

8.5 switch Multiple-Selection Statement (Cont.)

- ▶ Each case can have multiple actions (statements)
- ▶ Braces are not required around multiple actions in a case of a switch
- ▶ The break statement is not required for the last case because program control automatically continues with the next statement after the switch
- ▶ Having several case labels listed together (e.g., case 1: case 2: with no statements between the cases) executes the same set of actions for each case

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.7: SwitchTest.html -->
4 <!-- Using the switch multiple-selection statement. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Switching between HTML5 List Formats</title>
9     <script>
10
11       var choice; // user's choice
12       var startTag; // starting list item tag
13       var endTag; // ending list item tag
14       var validInput = true; // true if input valid else false
15       var listType; // type of list as a string
16
17       choice = window.prompt( "Select a list style:\n" +
18         "1 (numbered), 2 (lettered), 3 (roman numbered)", "1" );
19
```

Fig. 8.7 | Using the switch multiple-selection statement. (Part I of 6.)

```
20     switch ( choice )
21 {
22     case "1":
23         startTag = "<ol>";
24         endTag = "</ol>";
25         listType = "<h1>Numbered List</h1>";
26         break;
27     case "2":
28         startTag = "<ol style = 'list-style-type: upper-alpha'>";
29         endTag = "</ol>";
30         listType = "<h1>Lettered List</h1>";
31         break;
32     case "3":
33         startTag = "<ol style = 'list-style-type: upper-roman'>";
34         endTag = "</ol>";
35         listType = "<h1>Roman Numbered List</h1>";
36         break;
37     default:
38         validInput = false;
39         break;
40     } //end switch
41 }
```

Fig. 8.7 | Using the `switch` multiple-selection statement. (Part 2 of 6.)

```
42     if ( validInput === true )
43     {
44         document.writeln( listType + startTag );
45
46         for ( var i = 1; i <= 3; ++i )
47             document.writeln( "<li>List item " + i + "</li>" );
48
49         document.writeln( endTag );
50     } //end if
51     else
52         document.writeln( "Invalid choice: " + choice );
53
54     </script>
55     </head><body></body>
56 </html>
```

Fig. 8.7 | Using the switch multiple-selection statement. (Part 3 of 6.)

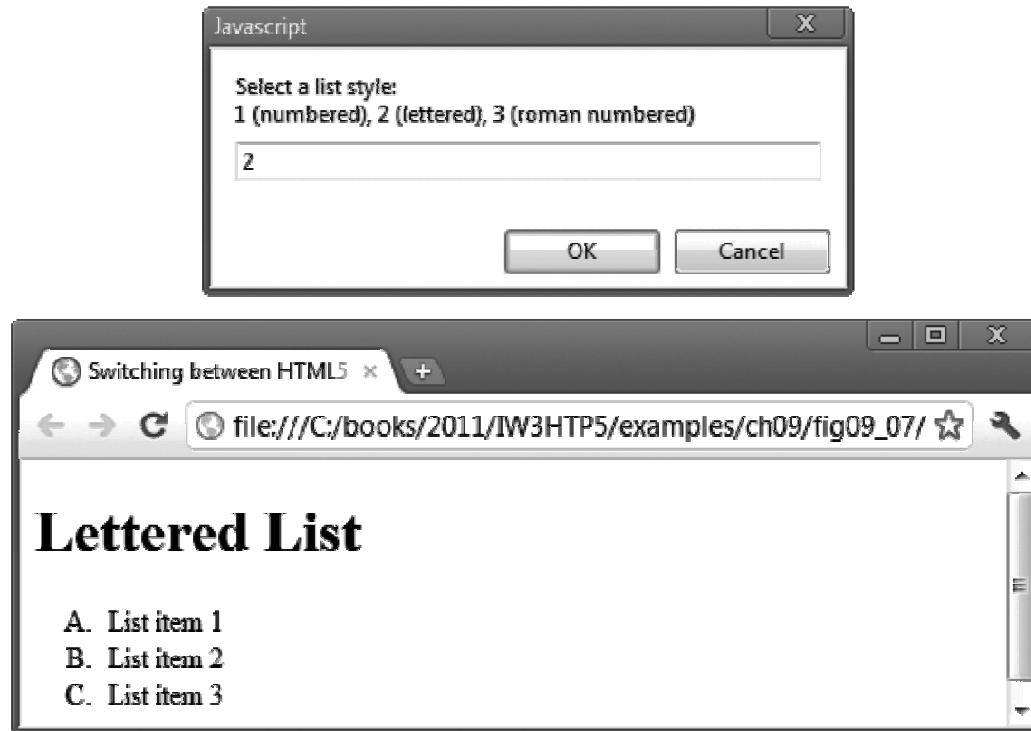


Fig. 8.7 | Using the switch multiple-selection statement. (Part 4 of 6.)

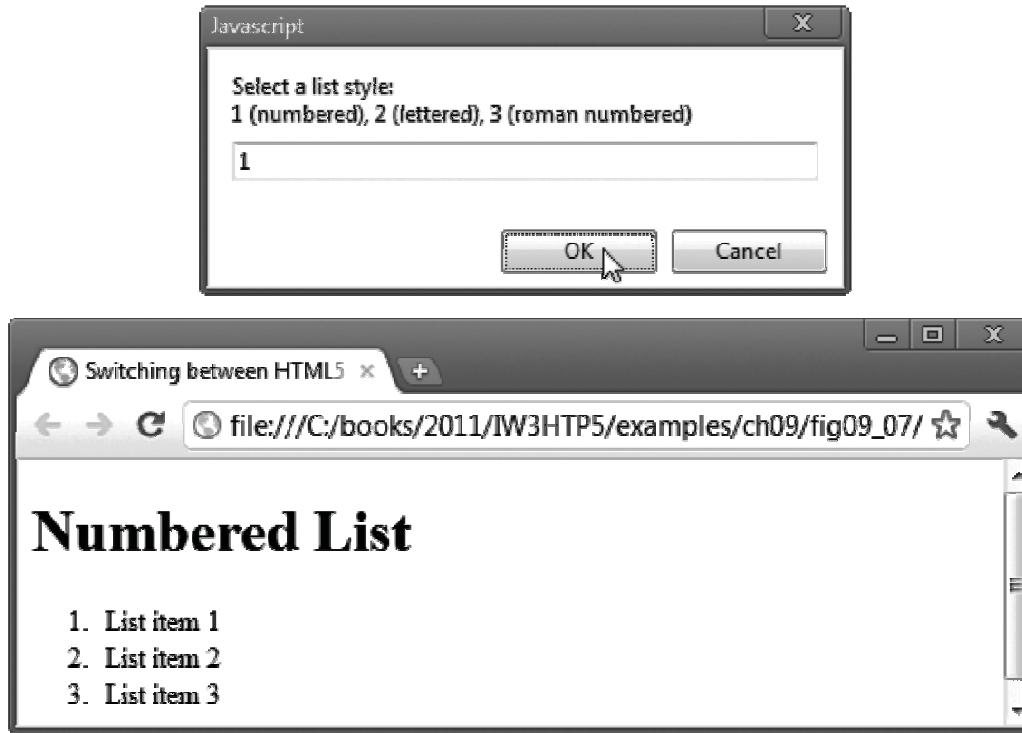


Fig. 8.7 | Using the switch multiple-selection statement. (Part 5 of 6.)

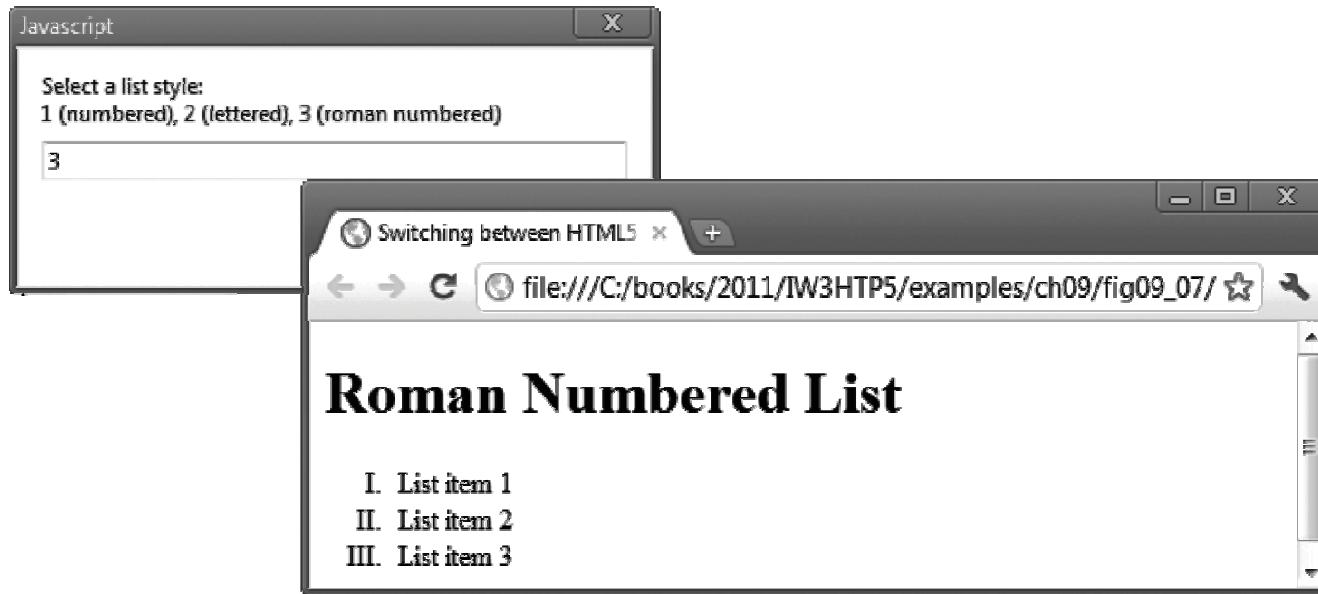


Fig. 8.7 | Using the `switch` multiple-selection statement. (Part 6 of 6.)

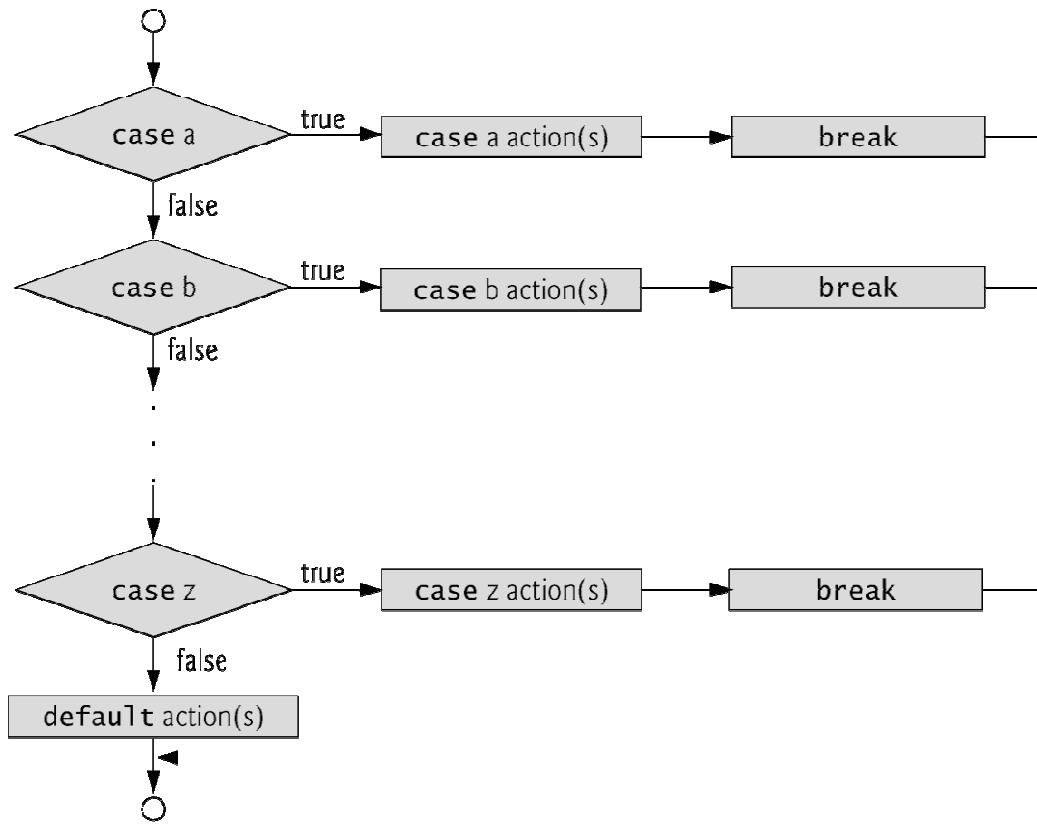


Fig. 8.8 | `switch` multiple-selection statement.

8.6 do...while Repetition Statement

- ▶ do...while statement
 - tests the loop-continuation condition *after* the loop body executes
 - *The loop body always executes at least once*

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 8.9: DoWhileTest.html -->
4  <!-- Using the do...while repetition statement. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Using the do...while Repetition Statement</title>
9          <script>
10
11      var counter = 1;
12
13      do {
14          document.writeln( "<h" + counter + ">This is " +
15              "an h" + counter + " level head" + "</h" +
16              counter + ">" );
17          ++counter;
18      } while ( counter <= 6 );
19
20      </script>
21
22      </head><body></body>
23  </html>
```

Fig. 8.9 | Using the do...while repetition statement. (Part 1 of 2.)

8.7 break and continue Statements

- ▶ break statement in a while, for, do...while or switch statement
 - Causes *immediate exit* from the statement
 - Execution continues with the next statement in sequence
- ▶ break statement common uses
 - Escape early from a loop
 - Skip the remainder of a switch statement

8.7 break and continue Statements (Cont.)

- ▶ **continue statement in a while, for or do...while**
 - skips the remaining statements in the body of the statement and proceeds with the next iteration of the loop
 - In while and do...while statements, the loop-continuation test evaluates immediately after the continue statement executes
 - In for statements, the increment expression executes, then the loop-continuation test evaluates

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.11: BreakTest.html -->
4 <!-- Using the break statement in a for statement. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>
9       Using the break Statement in a for Statement
10    </title>
11    <script>
12
13      for ( var count = 1; count <= 10; ++count )
14    {
15      if ( count == 5 )
16        break; // break loop only if count == 5
17
18      document.writeln( count + " " );
19    } //end for
20
```

Fig. 8.11 | Using the break statement in a for statement. (Part 1 of 2.)

```
21     document.writeln(
22         "<p>Broke out of loop at count = " + count + "</p>" );
23
24     </script>
25 </head><body></body>
26 </html>
```

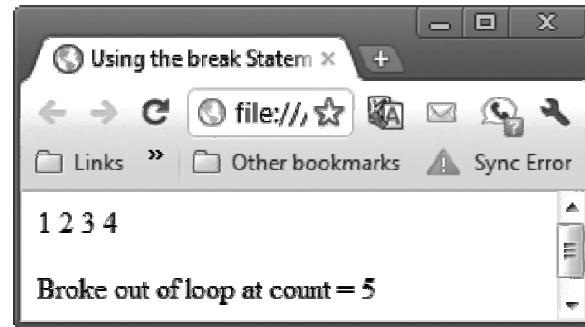


Fig. 8.11 | Using the break statement in a for statement. (Part 2 of 2.)

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.12: ContinueTest.html -->
4 <!-- Using the continue statement in a for statement. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>
9       Using the continue Statement in a for Statement
10    </title>
11
12   <script>
13
14     for ( var count = 1; count <= 10; ++count )
15     {
16       if ( count == 5 )
17         continue; // skip remaining loop code only if count == 5
18
19       document.writeln( count + " " );
20     } //end for
21
```

Fig. 8.12 | Using the `continue` statement in a `for` statement. (Part 1 of 2.)

```
22         document.writeln( "<p>Used continue to skip printing 5</p>" );
23
24     </script>
25
26     </head><body></body>
27 </html>
```

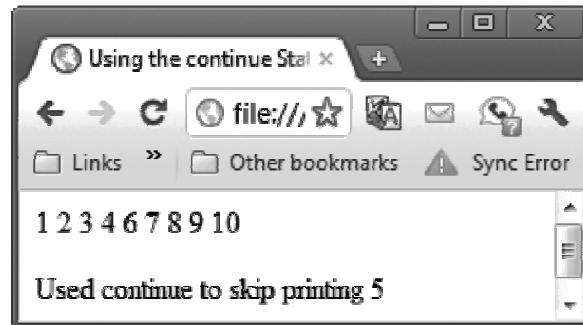


Fig. 8.12 | Using the `continue` statement in a `for` statement. (Part 2 of 2.)

8.8 Logical Operators

- ▶ Logical operators can be used to form complex conditions by combining simple conditions
 - `&&` (logical AND)
 - `||` (logical OR)
 - `!` (logical NOT, also called logical negation)
- ▶ The `&&` operator is used to ensure that two conditions are both true before choosing a certain path of execution
- ▶ JavaScript evaluates to `false` or `true` all expressions that include relational operators, equality operators and/or logical operators

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 8.13 | Truth table for the **&&** (logical AND) operator.

8.8 Logical Operators (Cont.)

- ▶ The `||` (logical OR) operator is used to ensure that either or both of two conditions are true before choosing choose a certain path of execution

expression1	expression2	expression1 expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 8.14 | Truth table for the `||` (logical OR) operator.

8.8 Logical Operators (Cont.)

- ▶ The `&&` operator has a higher precedence than the `||` operator
- ▶ Both operators associate from left to right.
- ▶ An expression containing `&&` or `||` operators is evaluated only until truth or falsity is known
 - This is called short-circuit evaluation

8.8 Logical Operators (Cont.)

- ▶ ! (logical negation) operator
 - reverses the meaning of a condition (i.e., a true value becomes false, and a false value becomes true)
 - Has only a single condition as an operand (i.e., it is a unary operator)
 - Placed before a condition to evaluate to true if the original condition (without the logical negation operator) is false

expression	<code>! expression</code>
false	true
true	false

Fig. 8.15 | Truth table for operator `!` (logical negation).

8.9 Logical Operators (Cont.)

- ▶ Most nonboolean values can be converted to a boolean true or false value
- ▶ Nonzero numeric values are considered to be true
- ▶ The numeric value zero is considered to be false
- ▶ Any string that contains characters is considered to be true
- ▶ The empty string is considered to be false
- ▶ The value null and variables that have been declared but not initialized are considered to be false
- ▶ All objects are considered to be true

Operator		Associativity	Type				
<code>++</code>	<code>--</code>	right to left	unary				
<code>*</code>	<code>/</code>	<code>%</code>	left to right	multiplicative			
<code>+</code>	<code>-</code>		left to right	additive			
<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>	left to right	relational		
<code>==</code>	<code>!=</code>	<code>---</code>	<code>!==</code>	left to right	equality		
<code>&&</code>				left to right	logical AND		
<code> </code>				left to right	logical OR		
<code>:</code>				right to left	conditional		
<code>=</code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	right to left	assignment

Fig. 8.16 | Precedence and associativity of the operators discussed so far.

9

JavaScript: Functions

9.1 Introduction

- ▶ To develop and maintain a large program
 - construct it from small, simple pieces
 - divide and conquer

9.2 Program Modules in JavaScript

- ▶ 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**.

9.2 Program Modules in JavaScript (Cont.)

- ▶ You can define programmer-defined functions that perform specific tasks and use them at many points in a script
 - The actual statements defining the function are written only once and are hidden from other functions
- ▶ Functions are invoked by writing the name of the function, followed by a left parenthesis, followed by a comma-separated list of zero or more arguments, followed by a right parenthesis
- ▶ Methods are called in the same way as functions, but require the name of the object to which the method belongs and a dot preceding the method name
- ▶ Function (and method) arguments may be constants, variables or expressions

9.3.1 Programmer-Defined Function square

- ▶ **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*
 - }

```
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
```

Fig. 9.2 | Programmer-defined function square. (Part 1 of 3.)

```
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`. (Part 2 of 3.)

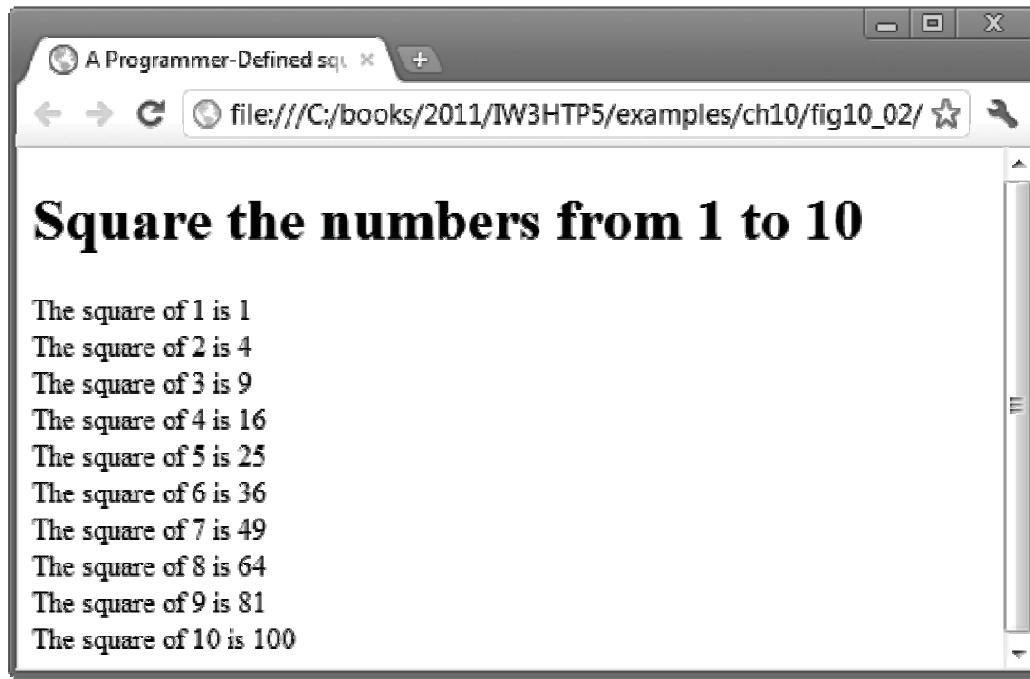


Fig. 9.2 | Programmer-defined function square. (Part 3 of 3.)

9.3.1 Programmer-Defined Function `square` (cont.)

- ▶ Three ways to return control to the point at which a function was invoked
 - Reaching the function-ending right brace
 - Executing the statement `return;`
 - Executing the statement “`return expression;`” to return 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 (cont.)

- ▶ 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.

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 9.3: maximum.html -->
4 <!-- Programmer-Defined maximum function. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Maximum of Three Values</title>
9     <style type = "text/css">
10       p { margin: 0; }
11     </style>
12     <script>
13
14       var input1 = window.prompt( "Enter first number", "0" );
15       var input2 = window.prompt( "Enter second number", "0" );
16       var input3 = window.prompt( "Enter third number", "0" );
17
18       var value1 = parseFloat( input1 );
19       var value2 = parseFloat( input2 );
20       var value3 = parseFloat( input3 );
21
22       var maxValue = maximum( value1, value2, value3 );
23
```

Fig. 9.3 | Programmer-defined maximum function. (Part I of 4.)

```
24     document.writeln( "<p>First number: " + value1 + "</p>" +
25         "<p>Second number: " + value2 + "</p>" +
26         "<p>Third number: " + value3 + "</p>" +
27         "<p>Maximum is: " + maxValue + "</p>" );
28
29     // maximum function definition (called from line 22)
30     function maximum( x, y, z )
31     {
32         return Math.max( x, Math.max( y, z ) );
33     } // end function maximum
34
35     </script>
36     </head><body></body>
37 </html>
```

Fig. 9.3 | Programmer-defined maximum function. (Part 2 of 4.)

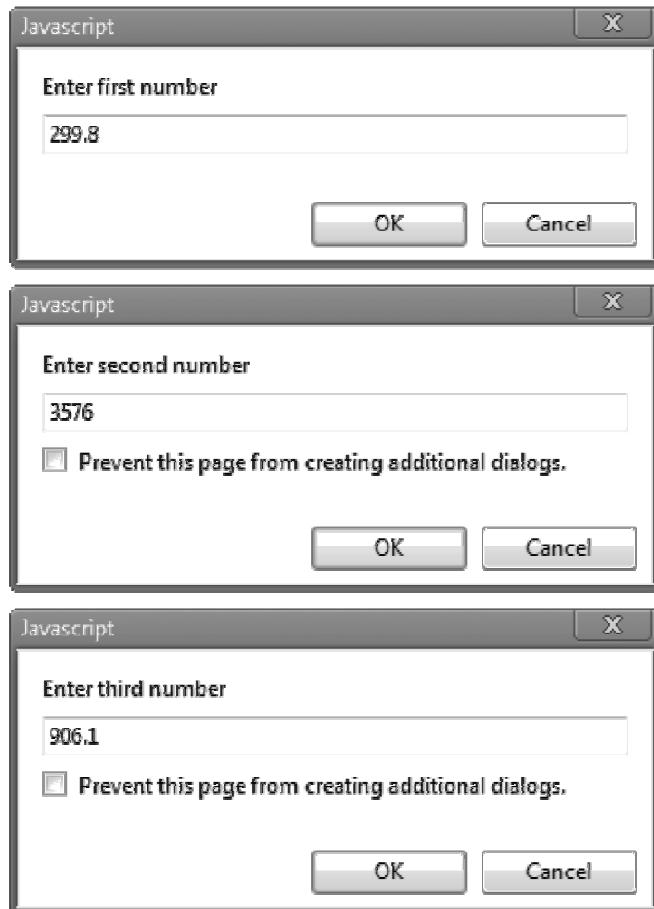


Fig. 9.3 | Programmer-defined maximum function. (Part 3 of 4.)

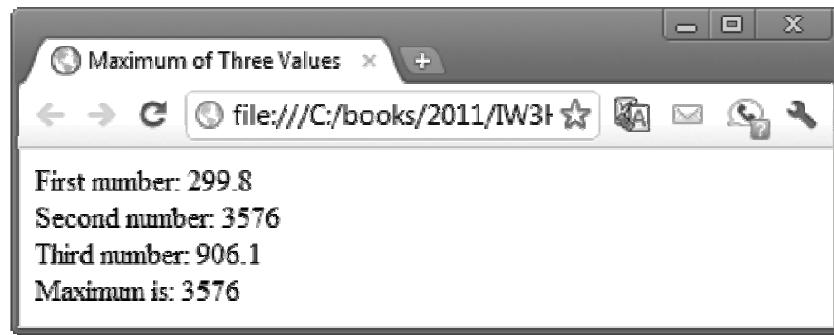


Fig. 9.3 | Programmer-defined maximum function. (Part 4 of 4.)

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.
 - Divide-and-conquer approach makes program development more manageable.
 - Software reusability.
 - Avoid repeating code in a program.

9.5 Random Number Generation

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

```
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 );
22                 document.writeln( "<li>" + value + "</li>" );
23             } // end for
24
```

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

```
25         document.writeln( "</ol>" );
26
27     </script>
28 </head><body></body>
29 </html>
```



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

9.5.2 Displaying Random Images

- ▶ 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—these are included with this chapter’s examples.

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 9.5: RollDice.html -->
4 <!-- Random dice image generation using Math.random. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Random Dice Images</title>
9     <style type = "text/css">
10       li { display: inline; margin-right: 10px; }
11       ul { margin: 0; }
12     </style>
13     <script>
14       // variables used to interact with the img elements
15       var die1Image;
16       var die2Image;
17       var die3Image;
18       var die4Image;
19 
```

Fig. 9.5 | Random dice image generation using Math.random. (Part I of 4.)

```
20     // register button listener and get the img elements
21     function start()
22     {
23         var button = document.getElementById( "rollButton" );
24         button.addEventListener( "click", rollDice, false );
25         die1Image = document.getElementById( "die1" );
26         die2Image = document.getElementById( "die2" );
27         die3Image = document.getElementById( "die3" );
28         die4Image = document.getElementById( "die4" );
29     } // end function rollDice
30
31     // roll the dice
32     function rollDice()
33     {
34         setImage( die1Image );
35         setImage( die2Image );
36         setImage( die3Image );
37         setImage( die4Image );
38     } // end function rollDice
39
```

Fig. 9.5 | Random dice image generation using Math.random. (Part 2 of 4.)

```
40      // set image source for a die
41      function setImage( dieImg )
42      {
43          var dieValue = Math.floor( 1 + Math.random() * 6 );
44          dieImg.setAttribute( "src", "die" + dieValue + ".png" );
45          dieImg.setAttribute( "alt",
46              "die image with " + dieValue + " spot(s)" );
47      } // end function setImage
48
49      window.addEventListener( "load", start, false );
50  </script>
51  </head>
```

Fig. 9.5 | Random dice image generation using `Math.random`. (Part 3 of 4.)

```
52     <body>
53         <form action = "#">
54             <input id = "rollButton" type = "button" value = "Roll Dice">
55         </form>
56         <ol>
57             <li><img id = "die1" src = "blank.png" alt = "die 1 image"></li>
58             <li><img id = "die2" src = "blank.png" alt = "die 2 image"></li>
59             <li><img id = "die3" src = "blank.png" alt = "die 3 image"></li>
60             <li><img id = "die4" src = "blank.png" alt = "die 4 image"></li>
61         </ol>
62     </body>
63 </html>
```

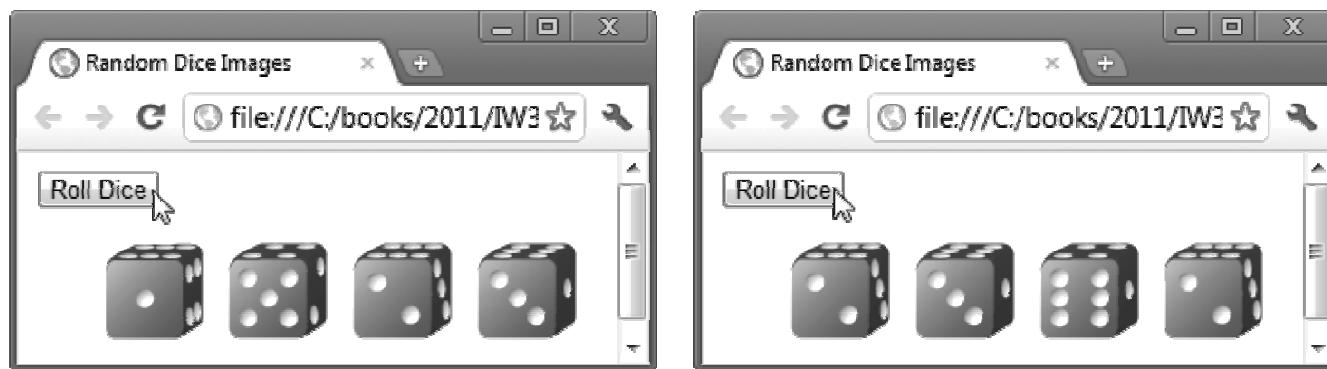


Fig. 9.5 | Random dice image generation using Math.random. (Part 4 of 4.)

9.5.2 Displaying Random Images

User Interactions Via Event Handling

- ▶ Until now, all user interactions with scripts have been through
 - a prompt dialog or
 - an alert dialog.
- ▶ These dialogs are valid ways to receive input from a user and to display messages, but 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.
- ▶ Outputs are typically displayed to the user in the web page.
- ▶ This program uses an HTML5 form and a new graphical user interface concept—**GUI event handling**.
- ▶ 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.

9.5.2 Displaying Random Images

The body Element

- ▶ The elements in the body are used extensively in the script.

The 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.

9.5.2 Displaying Random Images

The button input Element and Event-Driven Programming

- ▶ **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.

9.5.2 Displaying Random Images

The img Elements

- ▶ The four `img` elements 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.
- ▶ Each `img` element displays the image `blank.png` (an empty white image) when the page first renders.

9.5.2 Displaying Random Images

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

- ▶ Many examples will execute a JavaScript function when the document finishes loading.
- ▶ This is accomplished by handling the window object's load event.
- ▶ To specify the function to call when an event occurs, you register an event handler for that event.
- ▶ 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

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).

```
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">
10       img      { margin-right: 10px; }
11       table    { width: 200px;
12                     border-collapse: collapse;
13                     background-color: lightblue; }
14       table, td, th { border: 1px solid black;
15                     padding: 4px;
16                     margin-top: 20px; }
17       th        { text-align: left;
18                     color: white;
19                     background-color: darkblue; }
20     </style>
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 1 of 9.)

```
21 <script>
22     var frequency1 = 0;
23     var frequency2 = 0;
24     var frequency3 = 0;
25     var frequency4 = 0;
26     var frequency5 = 0;
27     var frequency6 = 0;
28     var totalDice = 0;
29
30     // register button event handler
31     function start()
32     {
33         var button = document.getElementById( "rollButton" );
34         button.addEventListener( "click", rollDice, false );
35     } // end function start
36
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 2 of 9.)

```
37      // roll the dice
38  function rollDice()
39  {
40      var face; // face rolled
41
42      // loop to roll die 12 times
43      for ( var i = 1; i <= 12; ++i )
44      {
45          face = Math.floor( 1 + Math.random() * 6 );
46          tallyRolls( face ); // increment a frequency counter
47          setImage( i, face ); // display appropriate die image
48          ++totalDice; // increment total
49      } // end die rolling loop
50
51      updateFrequencyTable();
52  } // end function rollDice
53
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 3 of 9.)

```
54     // increment appropriate frequency counter
55     function tallyRolls( face )
56     {
57         switch ( face )
58         {
59             case 1:
60                 ++frequency1;
61                 break;
62             case 2:
63                 ++frequency2;
64                 break;
65             case 3:
66                 ++frequency3;
67                 break;
68             case 4:
69                 ++frequency4;
70                 break;
71             case 5:
72                 ++frequency5;
73                 break;
74             case 6:
75                 ++frequency6;
76                 break;
77         } // end switch
78     } // end function tallyRolls
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 4 of 9.)

```
79
80    // set image source for a die
81    function setImage( dieNumber, face )
82    {
83        var dieImg = document.getElementById( "die" + dieNumber );
84        dieImg.setAttribute( "src", "die" + face + ".png" );
85        dieImg.setAttribute( "alt", "die with " + face + " spot(s)" );
86    } // end function setImage
87
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 5 of 9.)

```

88     // update frequency table in the page
89     function updateFrequencyTable()
90     {
91         var tableDiv = document.getElementById( "frequencyTableDiv" );
92
93         tableDiv.innerHTML = "<table>" +
94             "<caption>Die Rolling Frequencies</caption>" +
95             "<thead><th>Face</th><th>Frequency</th>" +
96             "<th>Percent</th></thead>" +
97             "<tbody><tr><td>1</td><td>" + frequency1 + "</td><td>" +
98             formatPercent(frequency1 / totalDice) + "</td></tr>" +
99             "<tr><td>2</td><td>" + frequency2 + "</td><td>" +
100            formatPercent(frequency2 / totalDice)+ "</td></tr>" +
101            "<tr><td>3</td><td>" + frequency3 + "</td><td>" +
102            formatPercent(frequency3 / totalDice) + "</td></tr>" +
103            "<tr><td>4</td><td>" + frequency4 + "</td><td>" +
104            formatPercent(frequency4 / totalDice) + "</td></tr>" +
105            "<tr><td>5</td><td>" + frequency5 + "</td><td>" +
106            formatPercent(frequency5 / totalDice) + "</td></tr>" +
107            "<tr><td>6</td><td>" + frequency6 + "</td><td>" +
108            formatPercent(frequency6 / totalDice) + "</td></tr>" +
109            "</tbody></table>";
110        } // end function updateFrequencyTable
111

```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 6 of 9.)

```
112      // format percentage
113      function formatPercent( value )
114      {
115          value *= 100;
116          return value.toFixed(2);
117      } // end function formatPercent
118
119      window.addEventListener( "load", start, false );
120  
```

```
  </script>
```

```
121  
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 7 of 9.)

```
122 <body>
123     <p><img id = "die1" src = "blank.png" alt = "die 1 image">
124         <img id = "die2" src = "blank.png" alt = "die 2 image">
125         <img id = "die3" src = "blank.png" alt = "die 3 image">
126         <img id = "die4" src = "blank.png" alt = "die 4 image">
127         <img id = "die5" src = "blank.png" alt = "die 5 image">
128         <img id = "die6" src = "blank.png" alt = "die 6 image"></p>
129     <p><img id = "die7" src = "blank.png" alt = "die 7 image">
130         <img id = "die8" src = "blank.png" alt = "die 8 image">
131         <img id = "die9" src = "blank.png" alt = "die 9 image">
132         <img id = "die10" src = "blank.png" alt = "die 10 image">
133         <img id = "die11" src = "blank.png" alt = "die 11 image">
134         <img id = "die12" src = "blank.png" alt = "die 12 image"></p>
135     <form action = "#">
136         <input id = "rollButton" type = "button" value = "Roll Dice">
137     </form>
138     <div id = "frequencyTableDiv"></div>
139 </body>
140 </html>
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 8 of 9.)

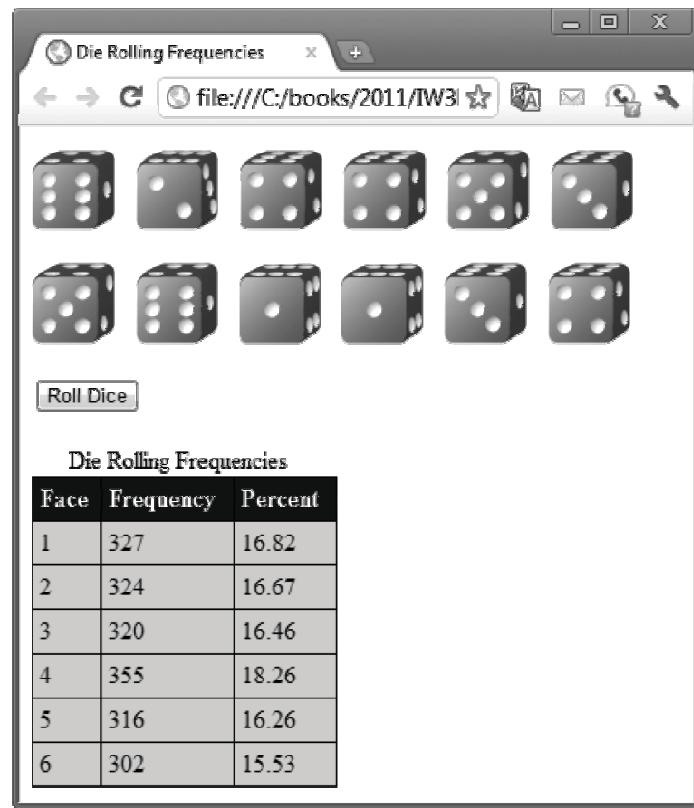


Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 9 of 9.)

9.5.3 Rolling Dice Repeatedly and Displaying Statistics

Generalized Scaling and Shifting of Random Values

- ▶ The values returned by random are always in the range
 $0.0 \leq \text{Math.random()} < 1.0$
- ▶ 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 $1 \leq \text{face} \leq 6$.
- ▶ 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`.

9.5.3 Rolling Dice Repeatedly and Displaying Statistics (cont.)

- ▶ 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

- ▶ The script in Fig. 9.7 simulates the game of craps.

```
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>
```

Fig. 9.7 | Craps game simulation. (Part 1 of 12.)

```
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
24 var playButton; // refers to Play button
25 var rollButton; // refers to Roll button
26 var dicerolling; // refers to audio clip for dice
27
28 // other variables used in program
29 var myPoint; // point if no win/loss on first roll
30 var die1Value; // value of first die in current roll
31 var die2Value; // value of second die in current roll
32
```

Fig. 9.7 | Craps game simulation. (Part 2 of 12.)

```
33     // starts a new game
34     function startGame()
35     {
36         // get the page elements that we'll interact with
37         dicerolling = document.getElementById( "dicerolling" );
38         pointDie1Img = document.getElementById( "pointDie1" );
39         pointDie2Img = document.getElementById( "pointDie2" );
40         rollDie1Img = document.getElementById( "rollDie1" );
41         rollDie2Img = document.getElementById( "rollDie2" );
42         messages = document.getElementById( "messages" );
43         playButton = document.getElementById( "play" );
44         rollButton = document.getElementById( "roll" );
45
46         // prepare the GUI
47         rollButton.disabled = true; // disable rollButton
48         setImage( pointDie1Img ); // reset image for new game
49         setImage( pointDie2Img ); // reset image for new game
50         setImage( rollDie1Img ); // reset image for new game
51         setImage( rollDie2Img ); // reset image for new game
52
53         myPoint = 0; // there is currently no point
54         firstRoll(); // roll the dice to start the game
55     } // end function startGame
56
```

Fig. 9.7 | Craps game simulation. (Part 3 of 12.)

```
57     // perform first roll of the game
58     function firstRoll()
59     {
60         var sumOfDice = rollDice(); // first roll of the dice
61
62         // determine if the user won, lost or must continue rolling
63         switch (sumOfDice)
64         {
65             case 7: case 11: // win on first roll
66                 messages.innerHTML =
67                     "You Win!!! Click Play to play again.";
68                 break;
69             case 2: case 3: case 12: // lose on first roll
70                 messages.innerHTML =
71                     "Sorry. You Lose. Click Play to play again.";
72                 break;
73             default: // remember point
74                 myPoint = sumOfDice;
75                 setImage( pointDie1Img, die1Value );
76                 setImage( pointDie2Img, die2Value );
77                 messages.innerHTML = "Roll Again!";
78                 rollButton.disabled = false; // enable rollButton
79                 playButton.disabled = true; // disable playButton
80                 break;
81         } // end switch
82     } // end function firstRoll
```

Fig. 9.7 | Craps game simulation. (Part 4 of 12.)

```
83
84    // called for subsequent rolls of the dice
85    function rollAgain()
86    {
87        var sumOfDice = rollDice(); // subsequent roll of the dice
88
89        if (sumOfDice == myPoint)
90        {
91            messages.innerHTML =
92                "You Win!!! Click Play to play again.";
93            rollButton.disabled = true; // disable rollButton
94            playButton.disabled = false; // enable playButton
95        } // end if
96        else if (sumOfDice == 7) // craps
97        {
98            messages.innerHTML =
99                "Sorry. You Lose. Click Play to play again.";
100           rollButton.disabled = true; // disable rollButton
101           playButton.disabled = false; // enable playButton
102       } // end else if
103   } // end function rollAgain
104
```

Fig. 9.7 | Craps game simulation. (Part 5 of 12.)

```
105     // roll the dice
106     function rollDice()
107     {
108         dicerolling.play(); // play dice rolling sound
109
110         // clear old die images while rolling sound plays
111         die1Value = NaN;
112         die2Value = NaN;
113         showDice();
114
115         die1Value = Math.floor(1 + Math.random() * 6);
116         die2Value = Math.floor(1 + Math.random() * 6);
117         return die1Value + die2Value;
118     } // end function rollDice
119
```

Fig. 9.7 | Craps game simulation. (Part 6 of 12.)

```
120     // display rolled dice
121     function showDice()
122     {
123         setImage( rollDie1Img, die1Value );
124         setImage( rollDie2Img, die2Value );
125     } // end function showDice
126
127     // set image source for a die
128     function setImage( dieImg, dieValue )
129     {
130         if ( isFinite( dieValue ) )
131             dieImg.src = "die" + dieValue + ".png";
132         else
133             dieImg.src = "blank.png";
134     } // end function setImage
135
```

Fig. 9.7 | Craps game simulation. (Part 7 of 12.)

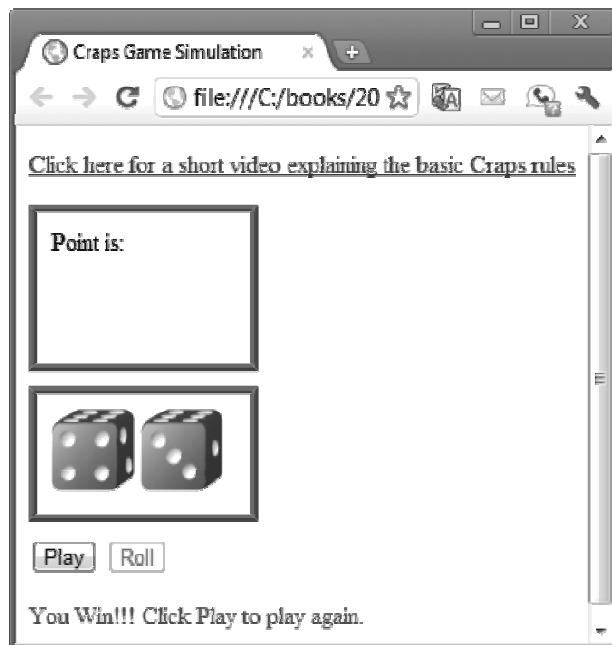
```
136     // register event listeners
137     function start()
138     {
139         var playButton = document.getElementById( "play" );
140         playButton.addEventListener( "click", startGame, false );
141         var rollButton = document.getElementById( "roll" );
142         rollButton.addEventListener( "click", rollAgain, false );
143         var diceSound = document.getElementById( "dicerolling" );
144         diceSound.addEventListener( "ended", showDice, false );
145     } // end function start
146
147     window.addEventListener( "load", start, false );
148 </script>
149 </head>
150 <body>
151     <audio id = "dicerolling" preload = "auto">
152         <source src = "http://test.deitel.com/dicerolling.mp3"
153             type = "audio/mpeg">
154         <source src = "http://test.deitel.com/dicerolling.ogg"
155             type = "audio/ogg">
156         Browser does not support audio tag</audio>
```

Fig. 9.7 | Craps game simulation. (Part 8 of 12.)

```
157 <p><a href = "CrapsRules.html">Click here for a short video  
158   explaining the basic Craps rules</a></p>  
159 <div id = "pointDiv">  
160   <p class = "point">Point is:</p>  
161   <img id = "pointDie1" src = "blank.png"  
162     alt = "Die 1 of Point Value">  
163   <img id = "pointDie2" src = "blank.png"  
164     alt = "Die 2 of Point Value">  
165 </div>  
166 <div class = "rollDiv">  
167   <img id = "rollDie1" src = "blank.png"  
168     alt = "Die 1 of Roll Value">  
169   <img id = "rollDie2" src = "blank.png"  
170     alt = "Die 2 of Roll Value">  
171 </div>  
172 <form action = "#">  
173   <input id = "play" type = "button" value = "Play">  
174   <input id = "roll" type = "button" value = "Roll">  
175 </form>  
176   <p id = "messages" class = "red">Click Play to start the game</p>  
177 </body>  
178 </html>
```

Fig. 9.7 | Craps game simulation. (Part 9 of 12.)

a) Win on the first roll. In this case, the pointDiv does not show any dice and the Roll button



b) Loss on the first roll. In this case, the pointDiv does not show any dice and the Roll button

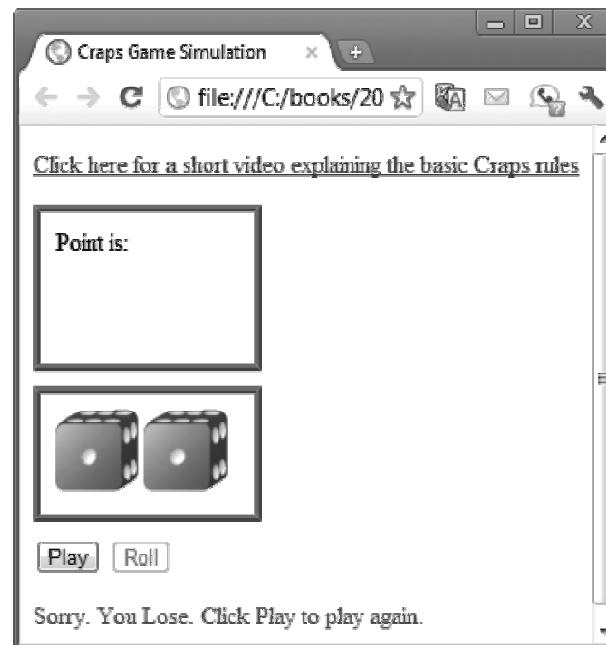
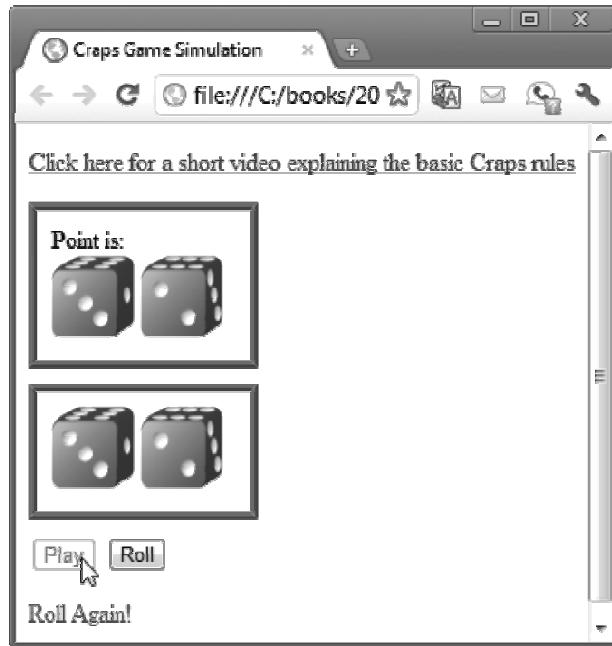


Fig. 9.7 | Craps game simulation. (Part 10 of 12.)

c) First roll is a 5, so the user's point is 5. The **Play** button is disabled and the **Roll** button is enabled.



d) User won on a subsequent roll. The **Play** button is enabled and the **Roll** button is disabled.

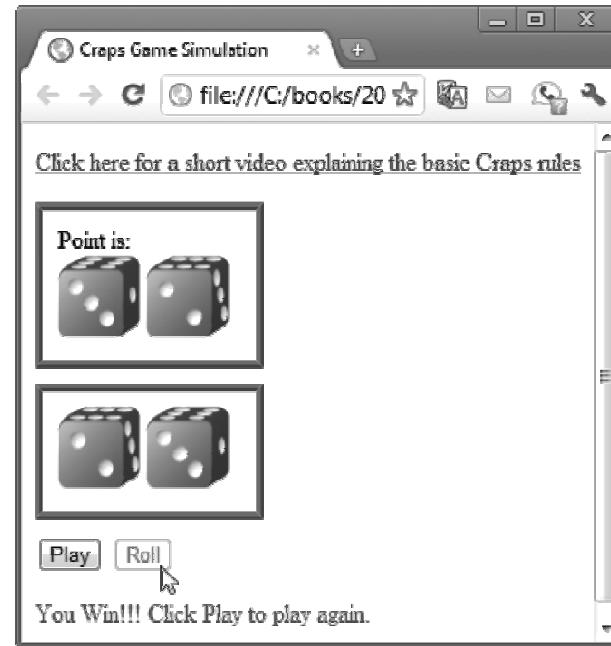
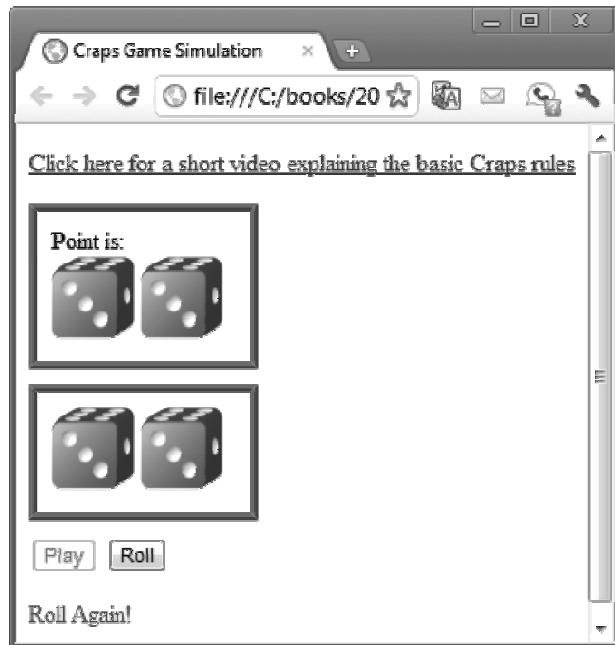


Fig. 9.7 | Craps game simulation. (Part 11 of 12.)

e) First roll is a 6, so the user's point is 6. The **Play** button is disabled and the **Roll** button is enabled.



f) User lost on a subsequent roll. The **Play** button is enabled and the **Roll** button is disabled.

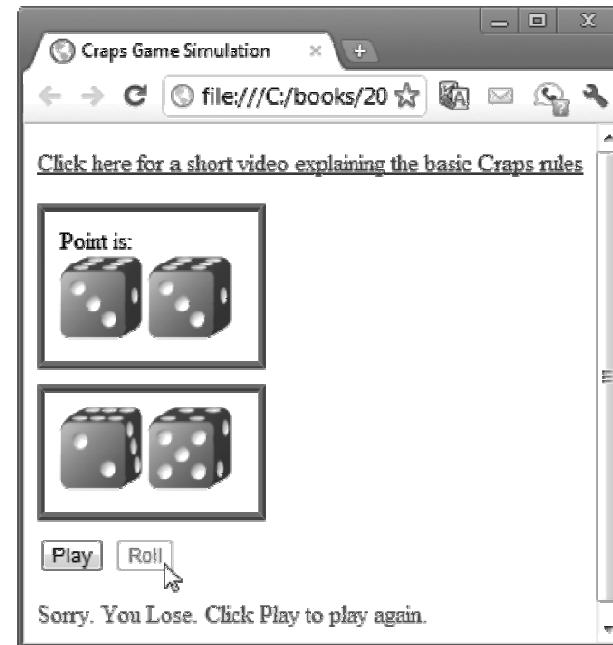


Fig. 9.7 | Craps game simulation. (Part 12 of 12.)

9.6 Example: Game of Chance; Introducing the HTML5 audio and video Elements (Cont.)

The HTML5 audio Element

- ▶ An HTML5 audio element 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.

9.6 Example: Game of Chance; Introducing the HTML5 audio and video Elements (Cont.)

- ▶ Most browsers support MP3, OGG and/or WAV format.
- ▶ 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.

9.6 Example: Game of Chance; Introducing the HTML5 audio and video Elements (Cont.)

CrapsRules.html and the HTML5 video Element

- ▶ When the user clicks the hyperlink in Craps.html, the CrapsRules.html is displayed in the browser.
- ▶ This page consists of a link back to Craps.html (Fig. 9.8) and an HTML5 video element 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
```

Fig. 9.8 | Web page that displays a video of the basic rules for the dice game Craps. (Part 1 of 3.)

```
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. (Part 2 of 3.)

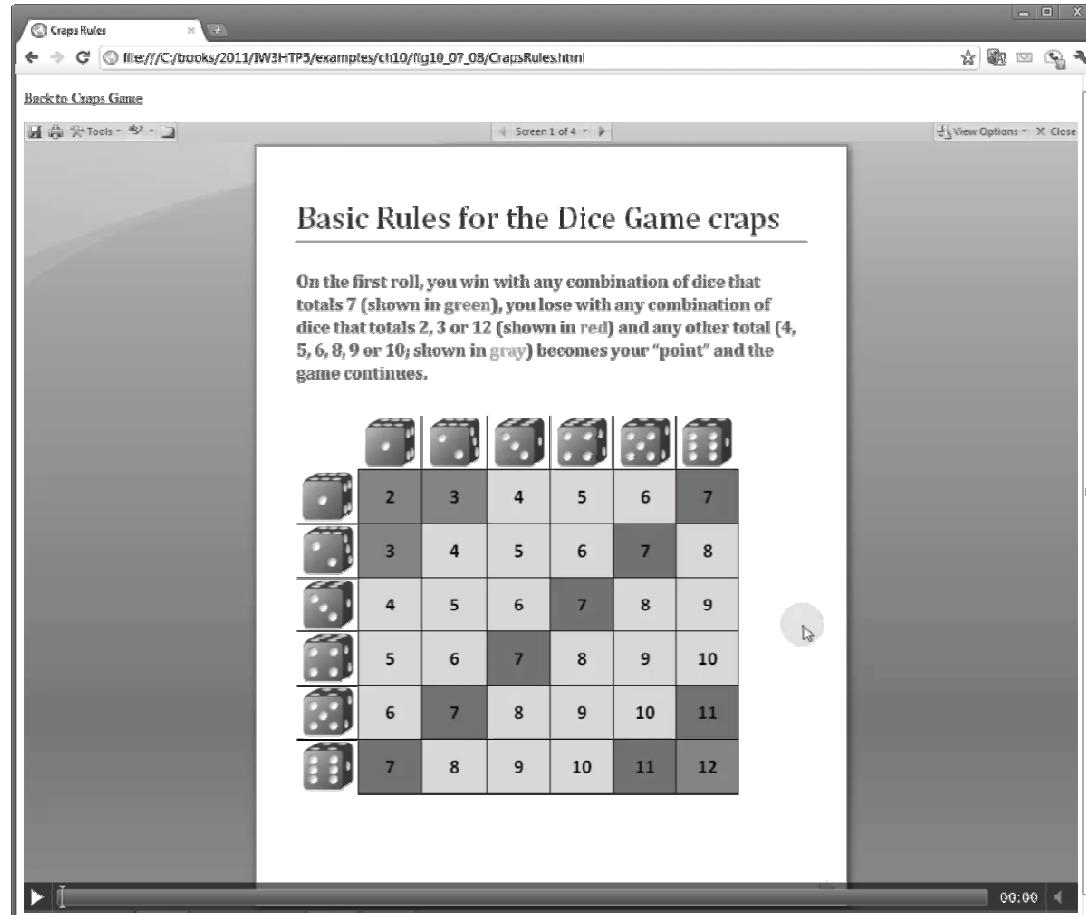


Fig. 9.8 | Web page that displays a video of the basic rules for the dice game Craps. (Part 3 of 3.)

9.7 Scope Rules

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

9.7 Scope Rules (Cont.)

- ▶ Identifiers declared inside a function have function (or local) scope and can be used only in that function
- ▶ Function scope begins with the opening left brace ({) of the function in which the identifier is declared and ends at the terminating right brace (})
- ▶ 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.

```
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>
```

Fig. 9.9 | Scoping example. (Part 1 of 4.)

```
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 +
29                     "</p>";
30         document.getElementById( "results" ).innerHTML = output;
31     } // end function start
32
```

Fig. 9.9 | Scoping example. (Part 2 of 4.)

```
33     function functionA()
34     {
35         var x = 25; // initialized each time functionA is called
36
37         output += "<p class='space'>local x in functionA is " + x +
38             " after entering functionA</p>";
39         ++x;
40         output += "<p>local x in functionA is " + x +
41             " before exiting functionA</p>";
42     } // end functionA
43
44     function functionB()
45     {
46         output += "<p class='space'>global variable x is " + x +
47             " on entering functionB";
48         x *= 10;
49         output += "<p>global variable x is " + x +
50             " on exiting functionB</p>";
51     } // end functionB
52
53     window.addEventListener( "load", start, false );
54 </script>
55 </head>
```

Fig. 9.9 | Scoping example. (Part 3 of 4.)

```
56     <body>
57         <div id = "results"></div>
58     </body>
59 </html>
```

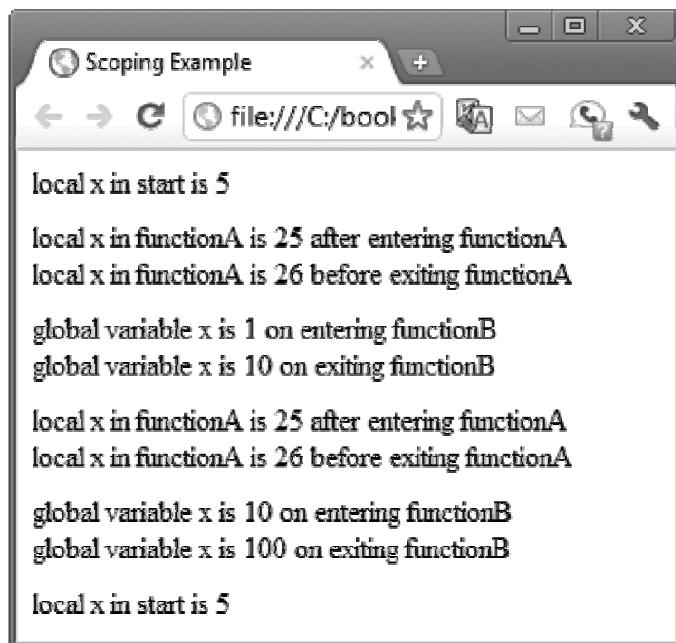


Fig. 9.9 | Scoping example. (Part 4 of 4.)

9.8 JavaScript Global Functions

- ▶ JavaScript provides nine global functions as part of a `Global` object
- ▶ This object contains
 - all the global variables in the script
 - all the user-defined functions in the script
 - all the built-in global functions listed in the following slide
- ▶ You do not need to use the `Global` object directly; JavaScript uses it for you

Global function	Description
<code>isFinite</code>	Takes a numeric argument and returns <code>true</code> if the value of the argument is not <code>Nan</code> , <code>Number.POSITIVE_INFINITY</code> or <code>Number.NEGATIVE_INFINITY</code> (values that are not numbers or numbers outside the range that JavaScript supports)—otherwise, the function returns <code>false</code> .
<code>isNaN</code>	Takes a numeric argument and returns <code>true</code> if the value of the argument is not a number; otherwise, it returns <code>false</code> . The function is commonly used with the return value of <code>parseInt</code> or <code>parseFloat</code> to determine whether the result is a proper numeric value.
<code>parseFloat</code>	Takes a string argument and attempts to convert the <i>beginning</i> of the string into a floating-point value. If the conversion is unsuccessful, the function returns <code>Nan</code> ; otherwise, it returns the converted value (e.g., <code>parseFloat("abc123.45")</code> returns <code>Nan</code> , and <code>parseFloat("123.45abc")</code> returns the value <code>123.45</code>).

Fig. 9.10 | JavaScript global functions. (Part 1 of 2.)

Global function	Description
<code>parseInt</code>	<p>Takes a string argument and attempts to convert the beginning of the string into an integer value. If the conversion is unsuccessful, the function returns <code>Nan</code>; otherwise, it returns the converted value (for example, <code>parseInt("abc123")</code> returns <code>Nan</code>, and <code>parseInt("123abc")</code> returns the integer value <code>123</code>). This function takes an optional second argument, from 2 to 36, specifying the radix (or base) of the number. Base 2 indicates that the first argument string is in binary format, base 8 that it's in octal format and base 16 that it's in hexadecimal format. See Appendix E, for more information on binary, octal and hexadecimal numbers.</p>

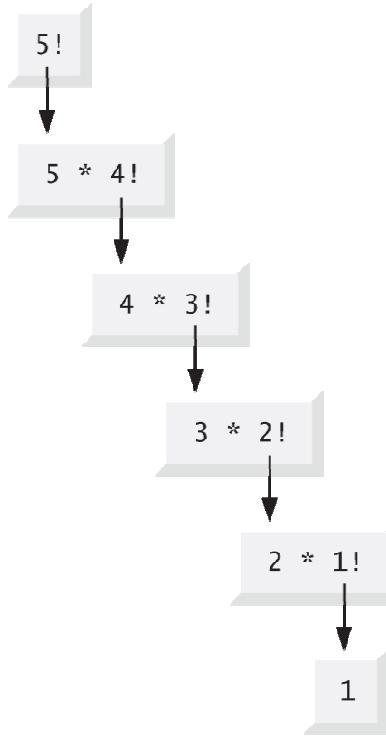
Fig. 9.10 | JavaScript global functions. (Part 2 of 2.)

9.9 Recursion

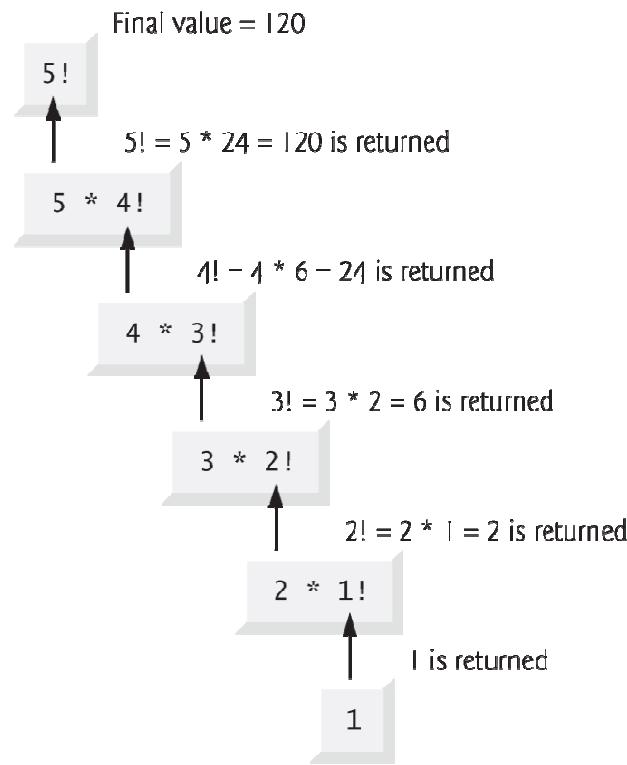
- ▶ A recursive function calls itself, either directly, or indirectly through another function.
- ▶ A recursive function knows how to solve only the simplest case, or base case
 - If the function is called with a base case, it returns a result
 - If the function is called with a more complex problem, it divides the problem into two conceptual pieces—a piece that the function knows how to process (the base case) and a simpler or smaller version of the original problem.
- ▶ The function invokes (calls) a fresh copy of itself to go to work on the smaller problem; this invocation is referred to as a recursive call, or the recursion step.

9.10 Recursion (Cont.)

- ▶ 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



(a) Sequence of recursive calls.



(b) Values returned from each recursive call.

Fig. 9.11 | Recursive evaluation of $5!$.

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 9.12: FactorialTest.html -->
4 <!-- Factorial calculation with a recursive function. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Recursive Factorial Function</title>
9     <style type = "text/css">
10       p { margin: 0px; }
11     </style>
12     <script>
13       var output = ""; // stores the output
14
15       // calculates factorials of 0 - 10
16       function calculateFactorials()
17     {
18         for ( var i = 0; i <= 10; ++i )
19             output += "<p>" + i + "!" + factorial( i ) + "</p>";
20
21         document.getElementById( "results" ).innerHTML = output;
22     } // end function calculateFactorials
23
```

Fig. 9.12 | Factorial calculation with a recursive function. (Part 1 of 3.)

```
24      // Recursive definition of function factorial
25      function factorial( number )
26      {
27          if ( number <= 1 ) // base case
28              return 1;
29          else
30              return number * factorial( number - 1 );
31      } // end function factorial
32
33      window.addEventListener( "load", calculateFactorials, false );
34  
```

</script>

```
35  
```

</head>

```
36  
```

<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. (Part 2 of 3.)

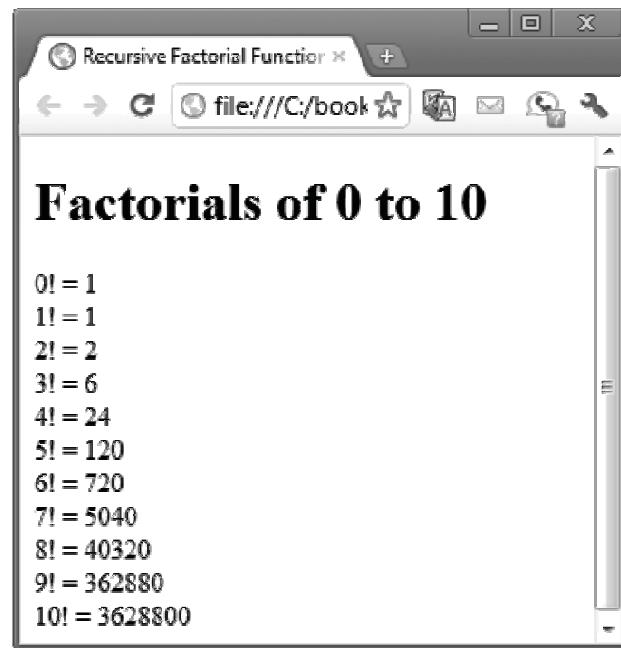


Fig. 9.12 | Factorial calculation with a recursive function. (Part 3 of 3.)

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

9.11 Recursion vs. Iteration

- ▶ Iteration both with counter-controlled repetition and recursion gradually approach termination
 - Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail
 - Recursion keeps producing simpler versions of the original problem until the base case is reached
- ▶ 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.