10

JavaScript: Arrays

10.1 Introduction

- Arrays
  - Data structures consisting of related data items
- JavaScript arrays
  - “dynamic” entities that can change size after they are created

10.2 Arrays

- An array is a group of memory locations
  - All have the same name and normally are of the same type (although this attribute is not required in JavaScript)
- Each individual location is called an element
- We may refer to any one of these elements by giving the array's name followed by the position number of the element in square brackets ([])

10.2 Arrays (Cont.)

- The first element in every array is the zeroth element.
- The *i*th element of array c is referred to as `c[i-1]`.
- Array names follow the same conventions as other identifiers
- A subscripted array name
  - can be used on the left side of an assignment to place a new value into an array element
  - can be used on the right side of an assignment operation to use its value
- Every array in JavaScript knows its own length, which it stores in its `length` attribute and can be found with the expression *`arrayname`*`.length`

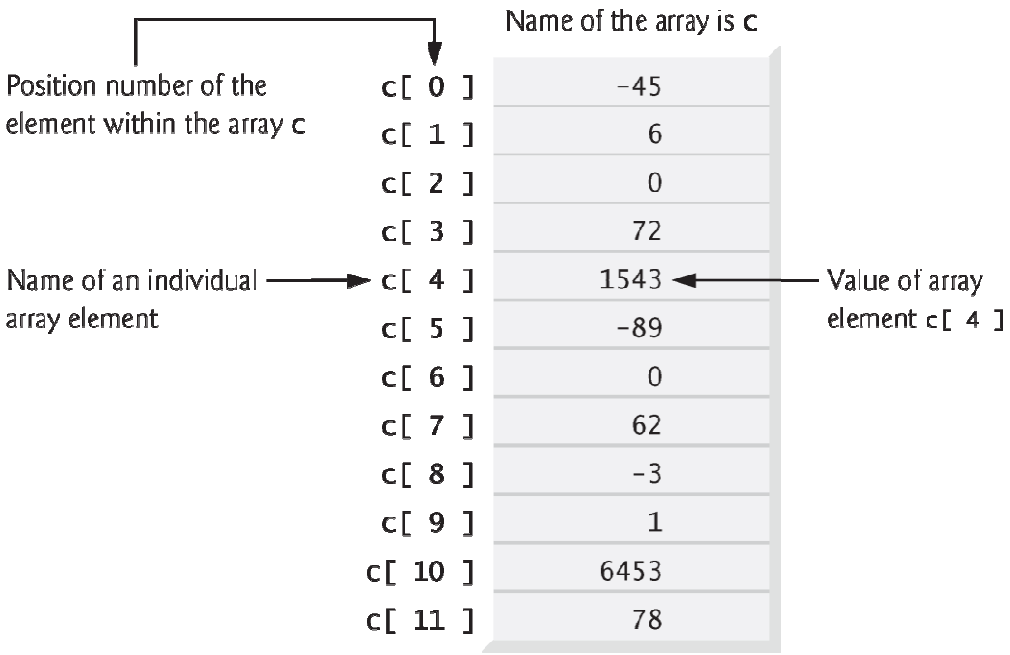


Fig. 10.1 | Array with 12 elements.

Operators	Associativity	Type
`()  []  .`	left to right	highest
`++  --  !`	right to left	unary
`*  /  %`	left to right	multiplicative
`+  -`	left to right	additive
`<  <=  >  >=`	left to right	relational
`==  !=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=  +=  -=  *=  /=  %=`	right to left	assignment

Fig. 10.2 | Precedence and associativity of the operators discussed so far.

10.3 Declaring and Allocating Arrays

- JavaScript arrays are **Array objects**.
- You use the **new operator** to create an array and to specify the number of elements in an array.
- The new operator creates an object as the script executes by obtaining enough memory to store an object of the type specified to the right of new .

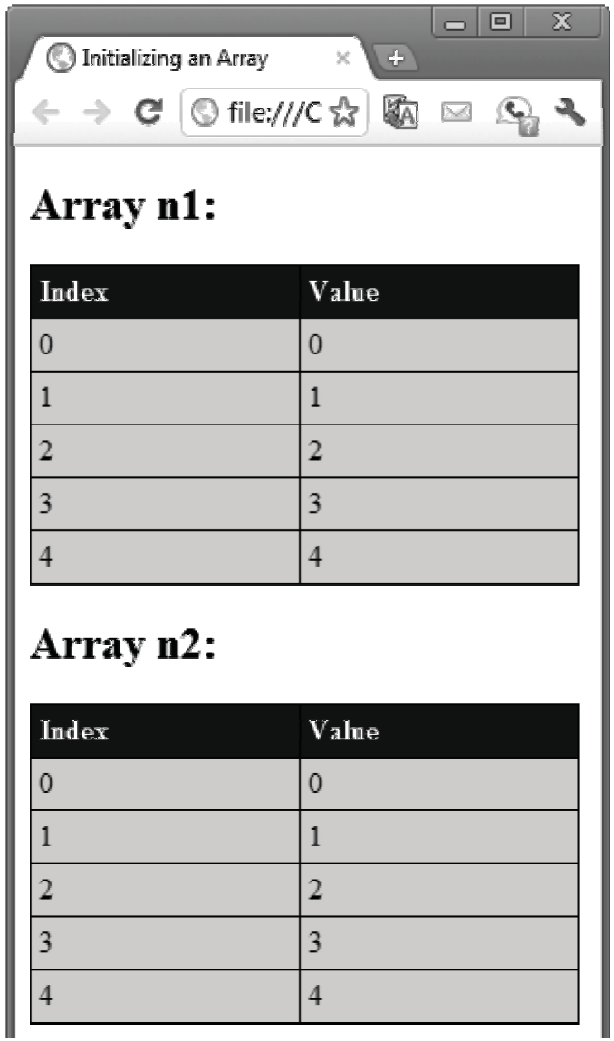
10.4 Examples Using Arrays

- Zero-based counting is usually used to iterate through arrays
- JavaScript reallocates an `Array` when a value is assigned to an element that is outside the bounds of the original `Array`

```
<!DOCTYPE html>

<!-- Fig. 10.3: InitArray.html -->
<!-- Web page for showing the results of initializing arrays. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Initializing an Array</title>
      <link rel = "stylesheet" type = "text/css" href = "tablestyle.css">
      <script src = "InitArray.js"></script>
   </head>
   <body>
      <div id = "output1"></div>
      <div id = "output2"></div>
   </body>
</html>
```

Fig. 10.3 | Web page for showing the results of initializing arrays.
(Part 1 of 2.)


Initializing an Array
file:///C
Array n1:
Index
Value
0
0
1
1
2
2
3
3
4
4
Array n2:
Index
Value
0
0
1
1
2
2
3
3
4
4

```
1  // Fig. 10.4: InitArray.js
2  // Create two arrays, initialize their elements and display them
3  function start()
4  {
5     var n1 = new Array( 5 ); // allocate five-element array
6     var n2 = new Array(); // allocate empty array
7
8     // assign values to each element of array n1
9     var length = n1.length; // get array's length once before the loop
10
11    for ( var i = 0; i < length; ++i )
12    {
13       n1[ i ] = i;
14    } // end for
15
16    // create and initialize five elements in array n2
17    for ( i = 0; i < 5; ++i )
18    {
19       n2[ i ] = i;
20    } // end for
21
22    outputArray( "Array n1:", n1, document.getElementById( "output1" ) );
23    outputArray( "Array n2:", n2, document.getElementById( "output2" ) );
24 } // end function start
```

Fig. 10.4 | Create two arrays, initialize their elements and display them. (Part 1 of 2.)

```
25
26  // output the heading followed by a two-column table
27  // containing indices and elements of "theArray"
28  function outputArray( heading, theArray, output )
29  {
30     var content = "<h2>" + heading + "</h2><table>" +
31        "<thead><th>Index</th><th>Value</th></thead><tbody>";
32
33     // output the index and value of each array element
34     var length = theArray.length; // get array's length once before loop
35
36     for ( var i = 0; i < length; ++i )
37     {
38        content += "<tr><td>" + i + "</td><td>" + theArray[ i ] +
39           "</td></tr>";
40     } // end for
41
42     content += "</tbody></table>";
43     output.innerHTML = content; // place the table in the output element
44  } // end function outputArray
45
46  window.addEventListener( "load", start, false );
```

Fig. 10.4 | Create two arrays, initialize their elements and display them. (Part 2 of 2.)

10.4 Examples Using Arrays (Cont.)

Using an Initializer List

- Arrays can be created using a comma-separated **initializer list** enclosed in square brackets (`[]`)
  - The array's size is determined by the number of values in the initializer list
- The initial values of an array can be specified as arguments in the parentheses following `new Array`
  - The size of the array is determined by the number of values in parentheses

10.4.2 Initializing Arrays with Initializer Lists

- The example in Figs. 10.5–10.6 creates three `Array` objects to demonstrate initializing arrays with initializer lists.
- Figure 10.5 is nearly identical to Fig. 10.3 but provides three `div`s in its body element for displaying this example's arrays.

```
<!DOCTYPE html>

<!-- Fig. 10.5: InitArray2.html -->
<!-- Web page for showing the results of initializing arrays. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Initializing an Array</title>
      <link rel = "stylesheet" type = "text/css" href = "tablestyle.css">
      <script src = "InitArray2.js"></script>
   </head>
   <body>
      <div id = "output1"></div>
      <div id = "output2"></div>
      <div id = "output3"></div>
   </body>
</html>
```

Fig. 10.5 | Web page for showing the results of initializing arrays. (Part 1 of 2.)

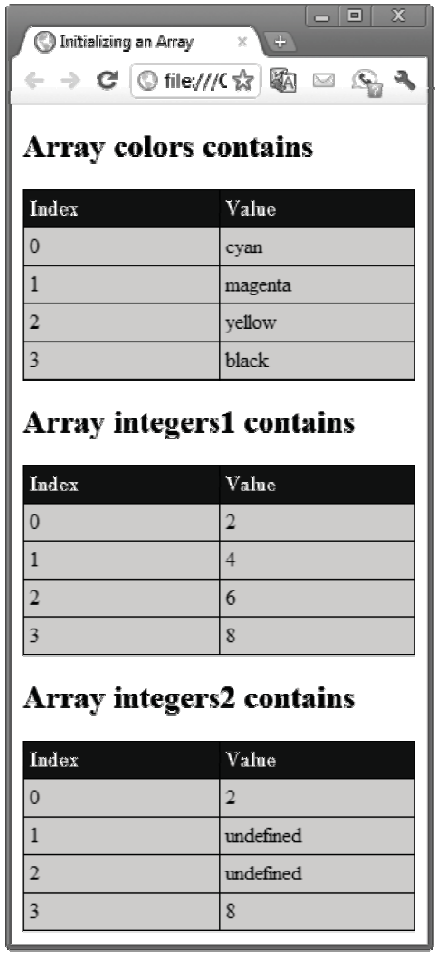


Fig. 10.5 | Web page for showing the results of initializing arrays. (Part 2 of 2.)

```
1  // Fig. 10.6: InitArray2.js
2  // Initializing arrays with initializer lists.
3  function start()
4  {
5     // Initializer list specifies the number of elements and
6     // a value for each element.
7     var colors = new Array( "cyan", "magenta","yellow", "black" );
8     var integers1 = [ 2, 4, 6, 8 ];
9     var integers2 = [ 2, , , 8 ];
10
11    outputArray( "Array colors contains", colors,
12       document.getElementById( "output1" ) );
13    outputArray( "Array integers1 contains", integers1,
14       document.getElementById( "output2" ) );
15    outputArray( "Array integers2 contains", integers2,
16       document.getElementById( "output3" ) );
17 } // end function start
18
```

Fig. 10.6 | Initializing arrays with initializer lists. (Part 1 of 2.)

```
19  // output the heading followed by a two-column table
20  // containing indices and elements of "theArray"
21  function outputArray( heading, theArray, output )
22  {
23     var content = "<h2>" + heading + "</h2><table>" +
24        "<thead><th>Index</th><th>Value</th></thead><tbody>";
25
26     // output the index and value of each array element
27     var length = theArray.length; // get array's length once before loop
28
29     for ( var i = 0; i < length; ++i )
30     {
31        content += "<tr><td>" + i + "</td><td>" + theArray[ i ] +
32           "</td></tr>";
33     } // end for
34
35     content += "</tbody></table>";
36     output.innerHTML = content; // place the table in the output element
37  } // end function outputArray
38
39  window.addEventListener( "load", start, false );
```

Fig. 10.6 | Initializing arrays with initializer lists. (Part 2 of 2.)

10.4.3 Summing the Elements of an `Array` with `for` and `for...in`

- The example in Figs. 10.7–10.8 sums an array's elements and displays the results.
- The document in Fig. 10.7 shows the results of the script in Fig. 10.8.
- JavaScript's `for...in` Repetition Statement
  - Enables a script to perform a task for each element in an array

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 10.7: SumArray.html -->
4  <!-- HTML5 document that displays the sum of an array's elements. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Sum Array Elements</title>
9        <script src = "SumArray.js"></script>
10    </head>
11    <body>
12       <div id = "output"></div>
13    </body>
14 </html>
```

Fig. 10.7 | HTML5 document that displays the sum of an array's elements.

```
1  // Fig. 10.8: SumArray.js
2  // Summing the elements of an array with for and for...in
3  function start()
4  {
5     var theArray = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
6     var total1 = 0, total2 = 0;
7
8     // iterates through the elements of the array in order and adds
9     // each element's value to total1
10    var length = theArray.length; // get array's length once before loop
11
12    for ( var i = 0; i < length; ++i )
13    {
14       total1 += theArray[ i ];
15    } // end for
16
17    var results = "<p>Total using indices: " + total1 + "</p>";
18
```

Fig. 10.8 | Summing the elements of an array with `for` and `for...in`. (Part 1 of 2.)

```
19      // iterates through the elements of the array using a for... in
20      // statement to add each element's value to total2
21      for ( var element in theArray )
22      {
23         total2 += theArray[ element ];
24      } // end for
25
26      results += "<p>Total using for...in: " + total2 + "</p>";
27      document.getElementById( "output" ).innerHTML = results;
28   } // end function start
29
30   window.addEventListener( "load", start, false );
```

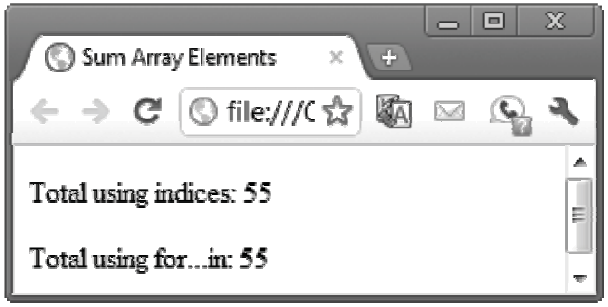


Fig. 10.8 | Summing the elements of an array with `for` and `for...in`. (Part 2 of 2.)

10.4.4 Using the Elements of an Array as Counters

- The example in Section 9.5.3 allowed the user to roll 12 dice at a time and kept statistics showing the number of times and the percentage of the time each face occurred.
- An array version of this example is shown in Figs. 10.9–10.10.
- We divided the example into three files
  - `style.css` contains the styles (not shown here),
  - `RollDice.html` (Fig. 10.9) contains the HTML5 document and
  - `RollDice.js` (Fig. 10.10) contains the JavaScript.

```
<!DOCTYPE html>

<!-- Fig. 10.9: RollDice.html -->
<!-- HTML5 document for the dice-rolling example. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Roll a Six-Sided Die 6000000 Times</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "RollDice.js"></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>
```

Fig. 10.9 | HTML5 document for the dice-rolling example. (Part 1 of 3.)

```
19          <p><img id = "die7" src = "blank.png" alt = "die 7 image">
20             <img id = "die8" src = "blank.png" alt = "die 8 image">
21             <img id = "die9" src = "blank.png" alt = "die 9 image">
22             <img id = "die10" src = "blank.png" alt = "die 10 image">
23             <img id = "die11" src = "blank.png" alt = "die 11 image">
24             <img id = "die12" src = "blank.png" alt = "die 12 image"></p>
25          <form action = "#">
26             <input id = "rollButton" type = "button" value = "Roll Dice">
27          </form>
28          <div id = "frequencyTableDiv"></div>
29       </body>
30    </html>
```

Fig. 10.9 | HTML5 document for the dice-rolling example. (Part 2 of 3.)

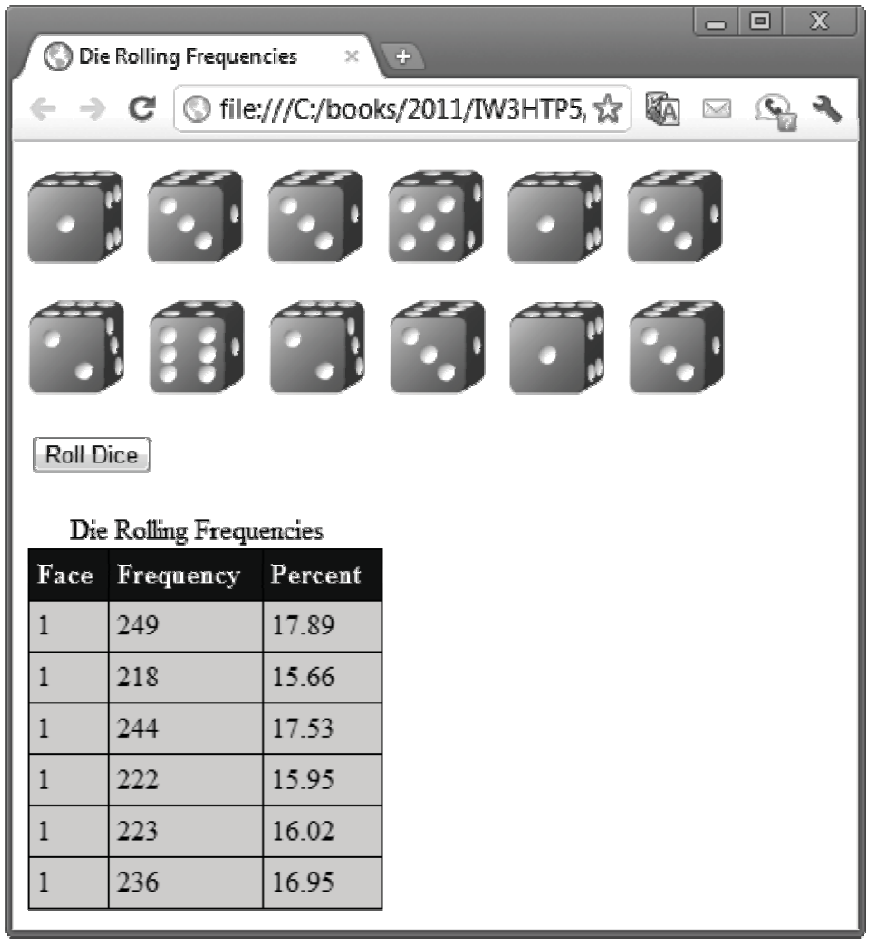


Fig. 10.9 | HTML5 document for the dice-rolling example. (Part 3 of

```
1  // Fig. 10.10: RollDice.js
2  // Summarizing die-rolling frequencies with an array instead of a switch
3  var frequency = [ , 0, 0, 0, 0, 0, 0 ]; // frequency[0] uninitialized
4  var totalDice = 0;
5  var dieImages = new Array(12); // array to store img elements
6
7  // get die img elements
8  function start()
9  {
10    var button = document.getElementById( "rollButton" );
11    button.addEventListener( "click", rollDice, false );
12    var length = dieImages.length; // get array's length once before loop
13
14    for ( var i = 0; i < length; ++i )
15    {
16       dieImages[ i ] = document.getElementById( "die" + (i + 1) );
17    } // end for
18 } // end function start
19
```

Fig. 10.10 | Summarizing die-rolling frequencies with an array instead of a `switch`. (Part 1 of 5.)

```
20  // roll the dice
21  function rollDice()
22  {
23     var face;  // face rolled
24     var length = dieImages.length;
25
26     for ( var i = 0; i < length; ++i )
27     {
28        face = Math.floor( 1 + Math.random() * 6 );
29        tallyRolls( face ); // increment a frequency counter
30        setImage( i, face ); // display appropriate die image
31        ++totalDice; // increment total
32     } // end for
33
34     updateFrequencyTable();
35  } // end function rollDice
36
37  // increment appropriate frequency counter
38  function tallyRolls( face )
39  {
40     ++frequency[ face ]; // increment appropriate counte
41  } // end function tallyRolls
42
```

Fig. 10.10 | Summarizing die-rolling frequencies with an array instead of a `switch`. (Part 2 of 5.)

```
43   // set image source for a die
44   function setImage( dieImg )
45   {
46      dieImages[ dieNumber ].setAttribute( "src", "die" + face + ".png" );
47      dieImages[ dieNumber ].setAttribute( "alt",
48         "die with " + face + " spot(s)" );
49   } // end function setImage
50
```

Fig. 10.10 | Summarizing die-rolling frequencies with an array instead of a `switch`. (Part 3 of 5.)

```
51  // update frequency table in the page
52  function updateFrequencyTable()
53  {
54     var results = "<table><caption>Die Rolling Frequencies</caption>" +
55        "<thead><th>Face</th><th>Frequency</th>" +
56        "<th>Percent</th></thead><tbody>";
57     var length = frequency.length;
58
59     // create table rows for frequencies
60     for ( var i = 1; i < length; ++i )
61     {
62        results += "<tr><td>1</td><td>" + i + "</td><td>" +
63           formatPercent(frequency[ i ] / totalDice) + "</td></tr>";
64     } // end for
65
66     results += "</tbody></table>";
67     document.getElementById( "frequencyTableDiv" ).innerHTML = results;
68  } // end function updateFrequencyTable
69
```

Fig. 10.10 | Summarizing die-rolling frequencies with an array instead of a switch. (Part 4 of 5.)

```
70   // format percentage
71   function formatPercent( value )
72   {
73      value *= 100;
74      return value.toFixed(2);
75   } // end function formatPercent
76
77   window.addEventListener( "load", start, false );
```

Fig. 10.10 | Summarizing die-rolling frequencies with an array instead of a `switch`. (Part 5 of 5.)

10.5 Random Image Generator Using Arrays

- In Chapter 9, the random image generator required image files to be named with the word `die` followed by a number from 1 to 6 and the file extension `.png` (e.g, `die1.png`).
- The example in Figs. 10.11–10.12 does not require the image filenames to contain integers in sequence.

10.5 Random Image Generator Using Arrays

- It uses an array `pictures` to store the names of the image files as strings.
  - Each time you click the image in the document, the script generates a random integer and uses it as an index into the `pictures` array.
  - The script updates the `img` element's `src` attribute with the image filename at the randomly selected position in the `pictures` array.
  - We update the `alt` attribute with an appropriate description of the image from the descriptions array.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 10.11: RandomPicture.html -->
4  <!-- HTML5 document that displays randomly selected images. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Random Image Generator</title>
9        <script src = "RandomPicture.js"></script>
10    </head>
11    <body>
12       <img id = "image" src = "CPE.png" alt = "Common Programming Error">
13    </body>
14 </html>
```

Fig. 10.11 | HTML5 document that displays randomly selected images. (Part 1 of 2.)

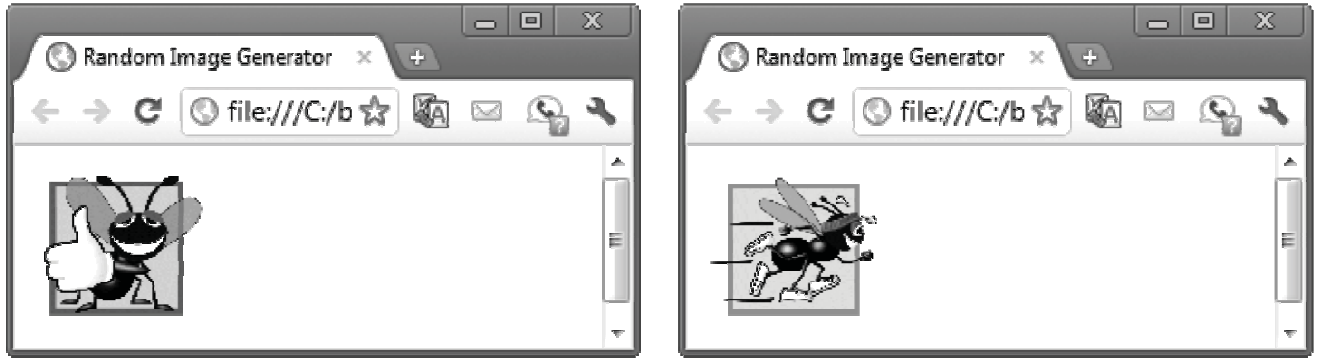


Fig. 10.11 | HTML5 document that displays randomly selected images. (Part 2 of 2.)

```
15  // Fig. 10.12: RandomPicture2.js
16  // Random image selection using arrays
17  var iconImg;
18  var pictures = [ "CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO" ];
19  var descriptions = [ "Common Programming Error",
20     "Error-Prevention Tip", "Good Programming Practice",
21     "Look-and-Feel Observation", "Performance Tip", "Portability Tip",
22     "Software Engineering Observation" ];
23
24  // pick a random image and corresponding description, then modify
25  // the img element in the document's body
26  function pickImage()
27  {
28     var index = Math.floor( Math.random() * 7 );
29     iconImg.setAttribute( "src", pictures[ index ] + ".png" );
30     iconImg.setAttribute( "alt", descriptions[ index ] );
31  } // end function pickImage
32
33  // registers iconImg's click event handler
34  function start()
35  {
36     iconImg = document.getElementById( "iconImg" );
37     iconImg.addEventListener( "click", pickImage, false );
38  } // end function start
39
40  window.addEventListener( "load", start, false );
```

Fig. 10.12 | Random image selection using arrays.

10.6 References and Reference Parameters

- Two ways to pass arguments to functions (or methods)
  - pass-by-value
  - pass-by-reference
- Pass-by-value
  - a *copy* of the argument's value is made and is passed to the called function

10.6 References and Reference Parameters

- In JavaScript, numbers, boolean values and strings are passed to functions by value.
- Pass-by-reference
  - The caller gives the called function access to the caller's data and allows the called function to *modify* the data if it so chooses
  - Can *improve performance* because it can eliminate the overhead of copying large amounts of data, but it can weaken security because the called function can access the caller's data
  - All objects are passed to functions by reference

10.7 Passing Arrays to Functions

- Pass an array as an argument to a function
  - Specify the array's name (a reference to the array) without brackets
- Although entire arrays are passed by reference, *individual numeric and boolean array elements* are passed *by value* exactly as simple numeric and boolean variables are passed
  - Such simple single pieces of data are called scalars, or scalar quantities
  - To pass an array element to a function, use the indexed name of the element as an argument in the function call

10.7 Passing Arrays to Functions (Cont.)

- `join` method of an `Array`
  - Returns a string that contains all of the elements of an array, separated by the string supplied in the function's argument
  - If an argument is not specified, the empty string is used as the separator

```
<!DOCTYPE html>

<!-- Fig. 10.13: PassArray.html -->
<!-- HTML document that demonstrates passing arrays and -->
<!-- individual array elements to functions. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Arrays as Arguments</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "PassArray.js"></script>
   </head>
   <body>
      <h2>Effects of passing entire array by reference</h2>
      <p id = "originalArray"></p>
      <p id = "modifiedArray"></p>
      <h2>Effects of passing array element by value</h2>
      <p id = "originalElement"></p>
      <p id = "inModifyElement"></p>
      <p id = "modifiedElement"></p>
   </body>
</html>
```

Fig. 10.13 | HTML document that demonstrates passing arrays and individual array elements to functions. (Part 1 of 2.)

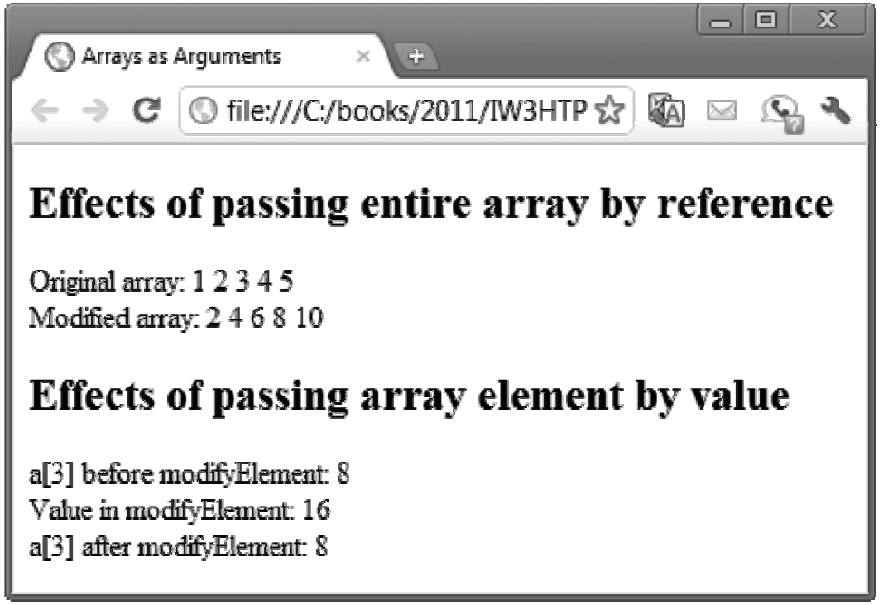


Fig. 10.13 | HTML document that demonstrates passing arrays and individual array elements to functions. (Part 2 of 2.)

```
1  // Fig. 10.14: PassArray.js
2  // Passing arrays and individual array elements to functions.
3  function start()
4  {
5     var a = [ 1, 2, 3, 4, 5 ];
6
7     // passing entire array
8     outputArray( "Original array: ", a,
9        document.getElementById( "originalArray" ) );
10    modifyArray( a );  // array a passed by reference
11    outputArray( "Modified array: ", a,
12       document.getElementById( "modifiedArray" ) );
13
14    // passing individual array element
15    document.getElementById( "originalElement" ).innerHTML =
16       "a[3] before modifyElement: " + a[ 3 ];
17    modifyElement( a[ 3 ] ); // array element a[3] passed by value
18    document.getElementById( "modifiedElement" ).innerHTML =
19       "a[3] after modifyElement: " + a[ 3 ];
20 } // end function start()
21
```

Fig. 10.14 | Passing arrays and individual array elements to functions. (Part 1 of 2.)

```
22  // outputs heading followed by the contents of "theArray"
23  function outputArray( heading, theArray, output )
24  {
25     output.innerHTML = heading + theArray.join( " " );
26  } // end function outputArray
27
28  // function that modifies the elements of an array
29  function modifyArray( theArray )
30  {
31     for ( var j in theArray )
32     {
33        theArray[ j ] *= 2;
34     } // end for
35  } // end function modifyArray
36
37  // function that modifies the value passed
38  function modifyElement( e )
39  {
40     e *= 2; // scales element e only for the duration of the function
41     document.getElementById( "inModifyElement" ).innerHTML =
42        "Value in modifyElement: " + e;
43  } // end function modifyElement
44
45  window.addEventListener( "load", start, false );
```

Fig. 10.14 | Passing arrays and individual array elements to functions. (Part 2 of 2.)

10.8 Sorting Arrays with `Array` Method `Sort`

- **Sorting** data
  - Putting data in a particular order, such as ascending or descending
  - One of the most important computing functions

10.8 Sorting Arrays with `Array` Method `sort` (Cont.)

- `Array` object in JavaScript has a built-in method `sort`
  - With no arguments, the method uses string comparisons to determine the sorting order of the array elements
  - Method `sort` takes as its argument the name of a function that compares its two arguments and returns
    - a negative value if the first argument is less than the second argument,
    - Zero if the arguments are *equal*, or
    - a positive value if the first argument is *greater than* the second

```
<!DOCTYPE html>

<!-- Fig. 10.15: Sort.html -->
<!-- HTML5 document that displays the results of sorting an array. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Array Method sort</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "Sort.js"></script>
   </head>
   <body>
      <h1>Sorting an Array</h1>
      <p id = "originalArray"></p>
      <p id = "sortedArray"></p>
   </body>
</html>
```

Fig. 10.15 | HTML5 document that displays the results of sorting an array. (Part 1 of 2.)

```
1  // Fig. 10.16: Sort.js
2  // Sorting an array with sort.
3  function start()
4  {
5     var a = [ 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 ];
6
7     outputArray( "Data items in original order: ", a,
8        document.getElementById( "originalArray" ) );
9     a.sort( compareIntegers );  // sort the array
10    outputArray( "Data items in ascending order: ", a,
11       document.getElementById( "sortedArray" ) );
12 } // end function start
13
14 // output the heading followed by the contents of theArray
15 function outputArray( heading, theArray, output )
16 {
17    output.innerHTML = heading + theArray.join( " " );
18 } // end function outputArray
19
```

Fig. 10.16 | Sorting an array with sort. (Part 1 of 2.)

```
20   // comparison function for use with sort
21   function compareIntegers( value1, value2 )
22   {
23      return parseInt( value1 ) - parseInt( value2 );
24   } // end function compareIntegers
25
26   window.addEventListener( "load", start, false );
```

Fig. 10.16 | Sorting an array with sort. (Part 2 of 2.)

10.9 Searching Arrays with `Array` Method `indexOf`

- It's often necessary to determine whether an array contains a value that matches a certain *key value*.
- The process of locating a particular element value in an array is called *searching*.
- The `Array` object in JavaScript has built-in methods `indexOf` and `lastIndexOf` for searching arrays.
  - Method `indexOf` searches for the first occurrence of the specified key value
  - Method `lastIndexOf` searches for the last occurrence of the specified key value.
- If the key value is found in the array, each method returns the index of that value; otherwise, `-1` is returned.

10.9 Searching Arrays with `Array` Method `indexOf` (Cont.)

- Every `input` element has a **`value` property** that can be used to get or set the element's value.

Optional Second Argument to `indexOf` and `lastIndexOf`

- You can pass an optional second argument to methods `indexOf` and `lastIndexOf` that represents the index from which to start the search.
- By default, this argument's value is 0 and the methods search the entire array.
- If the argument is greater than or equal to the array's length, the methods simply return -1.
- If the argument's value is negative, it's used as an offset from the end of the array.

```
<!DOCTYPE html>

<!-- Fig. 10.17: search.html -->
<!-- HTML5 document for searching an array with indexOf. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Search an Array</title>
      <script src = "search.js"></script>
   </head>
   <body>
      <form action  = "#">
         <p><label>Enter integer search key:
            <input id = "inputVal" type = "number"></label>
            <input id = "searchButton" type = "button" value = "Search">
         </p>
         <p id = "result"></p>
      </form>
   </body>
</html>
```

Fig. 10.17 | HTML5 document for searching an array with `indexOf`. (Part 1 of 2.)

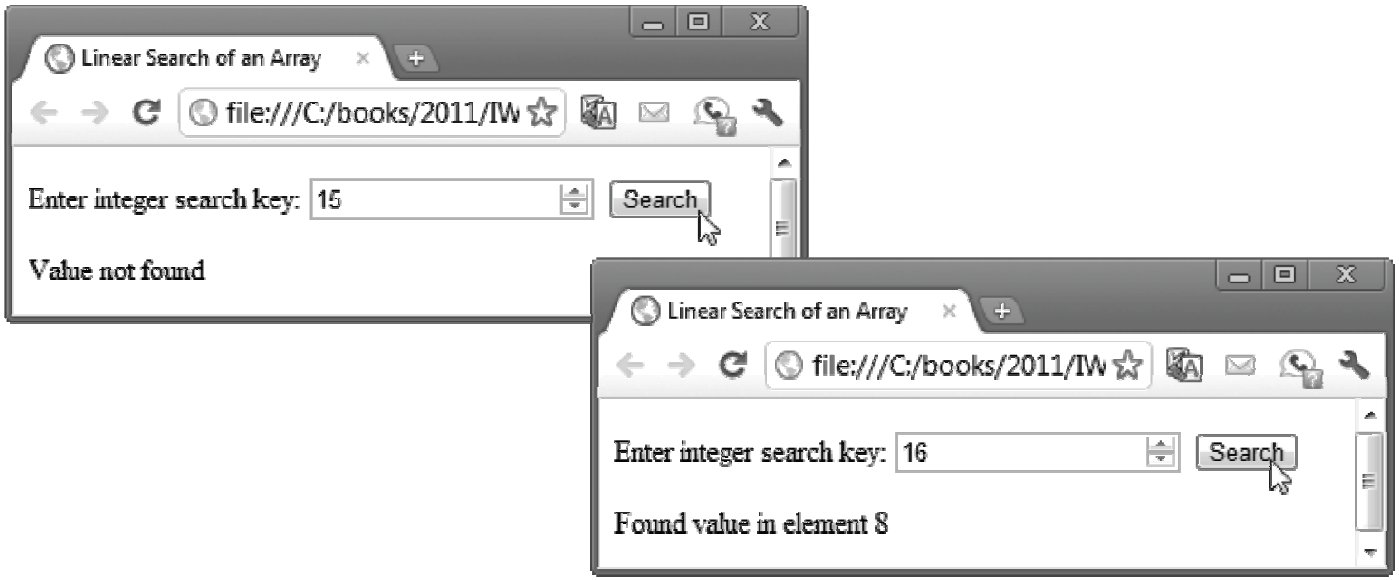


Fig. 10.17 | HTML5 document for searching an array with `indexOf`. (Part 2 of 2.)

```
1  // Fig. 10.18: search.js
2  // Search an array with indexOf.
3  var a = new Array( 100 );  // create an array
4
5  // fill array with even integer values from 0 to 198
6  for ( var i = 0; i < a.length; ++i )
7  {
8     a[ i ] = 2 * i;
9  } // end for
10
11 // function called when "Search" button is pressed
12 function buttonPressed()
13 {
14    // get the input text field
15    var inputVal = document.getElementById( "inputVal" );
16
17    // get the result paragraph
18    var result = document.getElementById( "result" );
19
20    // get the search key from the input text field then perform the search
21    var searchKey = parseInt( inputVal.value );
22    var element = a.indexOf( searchKey );
23
```

Fig. 10.18 | Search an array with `indexOf`.

```
24      if ( element != -1 )
25      {
26         result.innerHTML = "Found value in element " + element;
27      } // end if
28      else
29      {
30         result.innerHTML = "Value not found";
31      } // end else
32   } // end function buttonPressed
33
34   // register searchButton's click event handler
35   function start()
36   {
37      var searchButton = document.getElementById( "searchButton" );
38      searchButton.addEventListener( "click", buttonPressed, false );
39   } // end function start
40
41   window.addEventListener( "load", start, false );
```

Fig. 10.18 | Search an array with `indexOf`.

10.10 Multidimensional Arrays

- To identify a particular two-dimensional multidimensional array element
  - Specify the two indices
  - By convention, the first identifies the element's row, and the second identifies the element's column
- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array
- Two-dimensional array element accessed using an element name of the form `a[ row ][ column ]`
  - a is the name of the array
  - `row` and `column` are the indices that uniquely identify the row and column

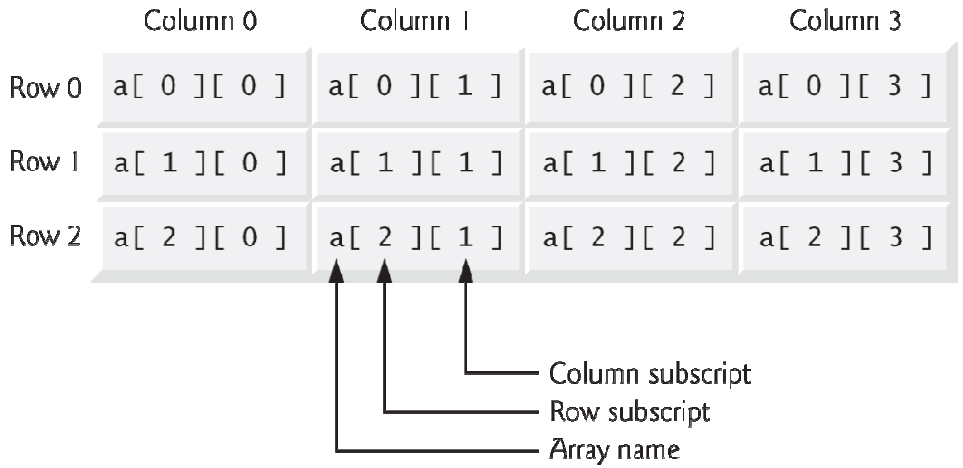


Fig. 10.19 | Two-dimensional array with three rows and four columns.

10.10 Multidimensional Arrays (Cont.)

- Multidimensional arrays can be initialized in declarations like a one-dimensional array, with values grouped by row in square brackets
  - The interpreter determines the number of rows by counting the number of sub initializer
  - The interpreter determines the number of columns in each row by counting the number of values in the sub-array that initializes the row
- The rows of a two-dimensional array can vary in length
- A multidimensional array in which each row has a different number of columns can be allocated dynamically with operator new

```
<!DOCTYPE html>

<!-- Fig. 10.20: InitArray3.html -->
<!-- HTML5 document showing multidimensional array initialization. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Multidimensional Arrays</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "InitArray3.js"></script>
   </head>
   <body>
      <h2>Values in array1 by row</h2>
      <div id = "output1"></div>
      <h2>Values in array2 by row</h2>
      <div id = "output2"></div>
   </body>
</html>
```

Fig. 10.20 | HTML5 document showing multidimensional array initialization. (Part 1 of 2.)

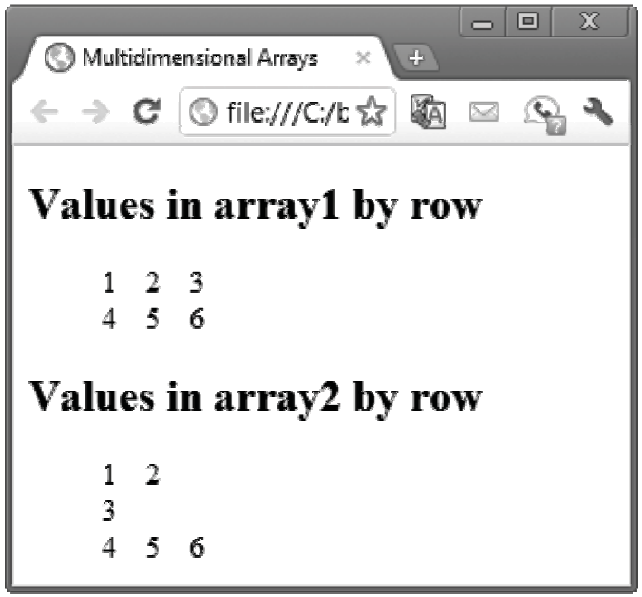


Fig. 10.20 | HTML5 document showing multidimensional array initialization. (Part 2 of 2.)

```
1  // Fig. 10.21: InitArray3.js
2  // Initializing multidimensional arrays.
3  function start()
4  {
5     var array1 = [ [ 1, 2, 3 ], // row 0
6                    [ 4, 5, 6 ] ]; // row 1
7     var array2 = [ [ 1, 2 ], // row 0
8                    [ 3 ], // row 1
9                    [ 4, 5, 6 ] ]; // row 2
10
11    outputArray( "Values in array1 by row", array1,
12       document.getElementById( "output1" ) );
13    outputArray( "Values in array2 by row", array2,
14       document.getElementById( "output2" ) );
15 } // end function start
16
```

Fig. 10.21 | Initializing multidimensional arrays. (Part 1 of 2.)

```
17  // display array contents
18  function outputArray( heading, theArray, output )
19  {
20     var results = "";
21
22     // iterates through the set of one-dimensional arrays
23     for ( var row in theArray )
24     {
25        results += "<ol>"; // start ordered list
26
27        // iterates through the elements of each one-dimensional array
28        for ( var column in theArray[ row ] )
29        {
30           results += "<li>" + theArray[ row ][ column ] + "</li>";
31        } // end inner for
32
33        results += "</ol>"; // end ordered list
34     } // end outer for
35
36     output.innerHTML = results;
37  } // end function outputArray
38
39  window.addEventListener( "load", start, false );
```

Fig. 10.21 | Initializing multidimensional arrays. (Part 2 of 2.)

11

JavaScript: Objects

11.1 Introduction

- This chapter presents a more formal treatment of objects.
- We use HTML5's new web storage capabilities to create a web application that stores a user's favorite Twitter searches on the computer for easy access at a later time.
- We also provide a brief introduction to JSON, a means for creating JavaScript objects—typically for transferring data over the Internet between client-side and server-side programs.

11.2 Math Object

- Math object methods enable you to conveniently perform many common mathematical calculations.
- An object's methods are called by writing the name of the object followed by a dot operator (.) and the name of the method
- In parentheses following the method name is are arguments to the method

Method	Description	Examples

`abs( x )`	Absolute value of x.	`abs( 7.2 )` is `7.2`
		`abs( 0 )` is `0`
		`abs( -5.6 )` is `5.6`

`ceil( x )`	Rounds x to the smallest integer not less than x.	`ceil( 9.2 )` is `10` `ceil( -9.8 )` is `-9.0`
`cos( x )`	Trigonometric cosine of x (x in radians).	`cos( 0 )` is `1`
`exp( x )`	Exponential method e^x.	`exp( 1 )` is `2.71828` `exp( 2 )` is `7.38906`
`floor( x )`	Rounds x to the largest integer not greater than x.	`floor( 9.2 )` is `9` `floor( -9.8 )` is `-10.0`
`log( x )`	Natural logarithm of x (base *e*).	`log( 2.718282 )` is `1` `log( 7.389056 )` is `2`
`max( x, y )`	Larger value of x and y.	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`

Fig. 11.1 | `Math` object methods. (Part 1 of 2.)

Method	Description	Examples
`min( x, y )`	Smaller value of x and y.	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`
`pow( x, y )`	x raised to power y (x^y).	`pow( 2, 7 )` is `128` `pow( 9, .5 )` is `3.0`
`round( x )`	Rounds x to the closest integer.	`round( 9.75 )` is `10` `round( 9.25 )` is `9`
`sin( x )`	Trigonometric sine of x (x in radians).	`sin( 0 )` is `0`
`sqrt( x )`	Square root of x.	`sqrt( 900 )` is `30` `sqrt( 9 )` is `3`
`tan( x )`	Trigonometric tangent of x (x in radians).	`tan( 0 )` is `0`

Fig. 11.1 | `Math` object methods. (Part 2 of 2.)

Constant	Description	Value
`Math.E`	Base of a natural logarithm (*e*).	Approximately 2.718
`Math.LN2`	Natural logarithm of 2.	Approximately 0.693
`Math.LN10`	Natural logarithm of 10.	Approximately 2.302
`Math.LOG2E`	Base 2 logarithm of *e*.	Approximately 1.442
`Math.LOG10E`	Base 10 logarithm of *e*.	Approximately 0.434
`Math.PI`	π—the ratio of a circle's circumference to its diameter.	Approximately 3.141592653589793
`Math.SQRT1_2`	Square root of 0.5.	Approximately 0.707
`Math.SQRT2`	Square root of 2.0.	Approximately 1.414

Fig. 11.2 | Properties of the `Math` object.

11.3 String Object

- Characters are the building blocks of JavaScript programs
- Every program is composed of a sequence of characters grouped together meaningfully that is interpreted by the computer as a series of instructions used to accomplish a task
- A string is a series of characters treated as a single unit
- A string may include letters, digits and various **special characters**, such as +, -, *, /, and $
- JavaScript supports **Unicode**, which represents a large portion of the world's languages
- **String literals** or **string constants** are written as a sequence of characters in double or single quotation marks

11.3.2 Methods of the `String` Object

- Combining strings is called **concatenation**

Method	Description
`charAt( index )`	Returns a string containing the character at the specified *index*. If there's no character at the *index*, `charAt` returns an empty string. The first character is located at *index* 0.
`charCodeAt( index )`	Returns the Unicode value of the character at the specified *index*, or `NaN` (not a number) if there's no character at that *index*.
`concat( string )`	Concatenates its argument to the end of the string on which the method is invoked. The original string is not modified; instead a new `String` is returned. This method is the same as adding two strings with the string-concatenation operator + (e.g., `s1.concat(s2)` is the same as `s1 + s2`).
`fromCharCode( value1, value2, ...)`	Converts a list of Unicode values into a string containing the corresponding characters.
`indexOf( substring, index )`	Searches for the *first* occurrence of *substring* starting from position *index* in the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from index 0 in the source string.

Fig. 11.3 | Some `String`-object methods. (Part 1 of 3.)

Method	Description
`lastIndexOf(` *substring*, *index* `)`	Searches for the *last* occurrence of *substring* starting from position *index* and searching toward the beginning of the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from the *end* of the source string.
`replace(` *searchString*, *replaceString* `)`	Searches for the substring *searchString*, replaces the first occurrence with *replaceString* and returns the modified string, or returns the original string if no replacement was made.
`slice(` *start*, *end* `)`	Returns a string containing the portion of the string from index *start* through index *end*. If the *end* index is not specified, the method returns a string from the *start* index to the end of the source string. A negative *end* index specifies an offset from the end of the string, starting from a position one past the end of the last character (so –1 indicates the last character position in the string).

Fig. 11.3 | Some `String`-object methods. (Part 2 of 3.)

Method	Description
`split( string )`	Splits the source string into an array of strings (tokens), where its *string* argument specifies the delimiter (i.e., the characters that indicate the end of each token in the source string).
`substr( start, length )`	Returns a string containing *length* characters starting from index *start* in the source string. If *length* is not specified, a string containing characters from *start* to the end of the source string is returned.
`substring( start, end )`	Returns a string containing the characters from index *start* up to but not including index *end* in the source string.
`toLowerCase()`	Returns a string in which all uppercase letters are converted to lowercase letters. Non-letter characters are not changed.
`toUpperCase()`	Returns a string in which all lowercase letters are converted to uppercase letters. Non-letter characters are not changed.

Fig. 11.3 | Some `String`-object methods. (Part 3 of 3.)

11.3.3 Character Processing Methods

- `String` method `charAt`
  - Returns the character at a specific position
  - Indices for the characters in a string start at 0 (the first character) and go up to (but do not include) the string's `length`
  - If the index is outside the bounds of the string, the method returns an empty string
- `String` method `charCodeAt`
  - Returns the Unicode value of the character at a specific position
  - If the index is outside the bounds of the string, the method returns `NaN`.

11.3.3 Character Processing Methods (Cont.)

- `String` method `fromCharCode`
  - Returns a string created from a series of Unicode values
- `String` method `toLowerCase`
  - Returns the lowercase version of a string
- `String` method `toUpperCase`
  - Returns the uppercase version of a string

```
<!DOCTYPE html>

<!-- Fig. 11.4: CharacterProcessing.html -->
<!-- HTML5 document to demonstrate String methods charAt, charCodeAt,
   fromCharCode, toLowercase and toUpperCase. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Character Processing</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "CharacterProcessing.js"></script>
   </head>
   <body>
      <div id = "results"></div>
   </body>
</html>
```

Fig. 11.4 | HTML5 document to demonstrate methods charAt, charCodeAt, fromCharCode, toLowercase and toUpperCase. (Part 1 of 2.)

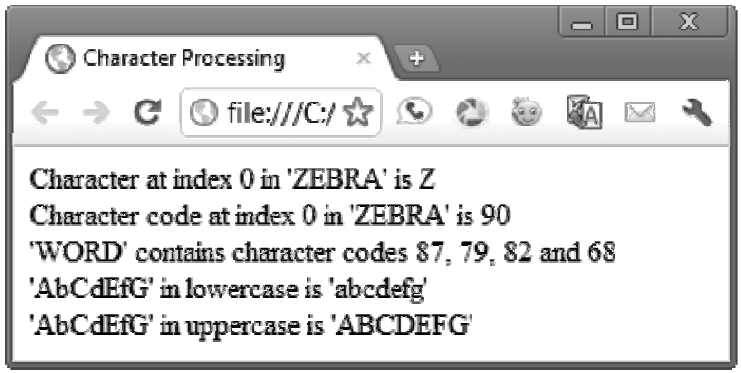


Fig. 11.4 | HTML5 document to demonstrate methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowercase` and `toUpperCase`. (Part 2 of 2.)

```
1  // Fig. 11.5: CharacterProcessing.js
2  // String methods charAt, charCodeAt, fromCharCode,
3  // toLowercase and toUpperCase.
4  function start()
5  {
6     var s = "ZEBRA";
7     var s2 = "AbCdEfG";
8     var result = "";
9
10    result = "<p>Character at index 0 in '" + s + "' is " +
11       s.charAt( 0 ) + "</p>";
12    result += "<p>Character code at index 0 in '" + s + "' is " +
13       s.charCodeAt( 0 ) + "</p>";
14
15    result += "<p>'" + String.fromCharCode( 87, 79, 82, 68 ) +
16       "' contains character codes 87, 79, 82 and 68</p>";
17
```

Fig. 11.5 | String methods charAt, charCodeAt, fromCharCode, toLowercase and toUpperCase. (Part 1 of 2.)

```
18      result += "<p>'" + s2 + "' in lowercase is '" +
19         s2.toLowerCase() + "'</p>";
20      result += "<p>'" + s2 + "' in uppercase is '" +
21         s2.toUpperCase() + "'</p>";
22
23      document.getElementById( "results" ).innerHTML = result;
24   } // end function start
25
26   window.addEventListener( "load", start, false );
```

Fig. 11.5 | String methods charAt, charCodeAt, fromCharCode, toLowercase and toUpperCase. (Part 2 of 2.)

11.3.4. Searching Methods

- String method `indexOf`
  - Determines the location of the first occurrence of its argument in the string used to call the method
  - If the substring is found, the index at which the first occurrence of the substring begins is returned; otherwise, -1 is returned
  - Receives an optional second argument specifying the index from which to begin the search
- String method `lastIndexOf`
  - Determines the location of the last occurrence of its argument in the string used to call the method
  - If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, -1 is returned
  - Receives an optional second argument specifying the index from which to begin the search

```
<!DOCTYPE html>

<!-- Fig. 11.6: SearchingStrings.html -->
<!-- HTML document to demonstrate methods indexOf and lastIndexOf. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Searching Strings</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
   <script src = "SearchingStrings.js"></script>
   </head>
   <body>
      <form id = "searchForm" action = "#">
         <h1>The string to search is:
             abcdefghijklmnopqrstuvwxyzabcdefghijklm</h1>
         <p>Enter the substring to search for
         <input id = "inputField" type = "search">
         <input id = "searchButton" type = "button" value = "Search"></p>
         <div id = "results"></div>
      </form>
   </body>
</html>
```

Fig. 11.6 | HTML document to demonstrate methods `indexOf` and `lastIndexOf`. (Part 1 of 3.)

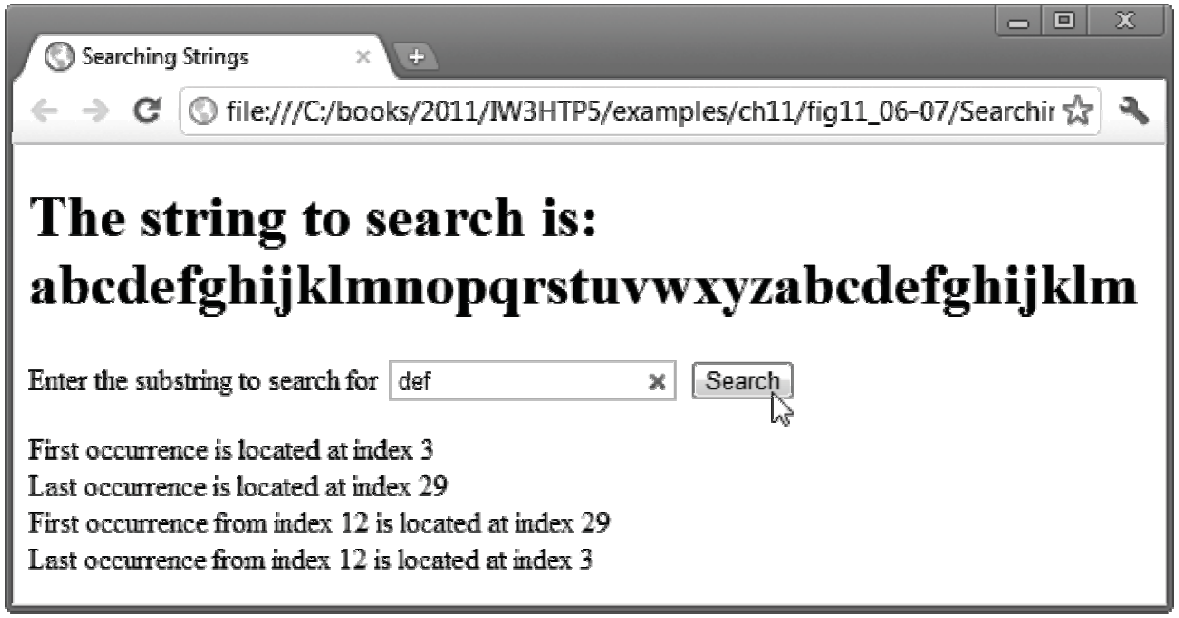


Fig. 11.6 | HTML document to demonstrate methods `indexOf` and `lastIndexOf`. (Part 2 of 3.)

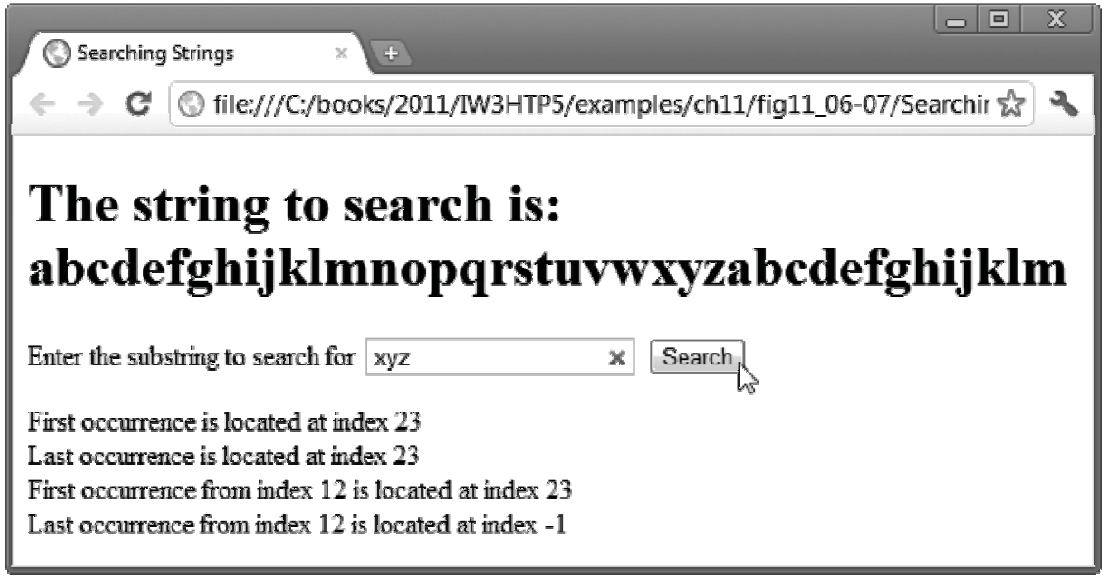


Fig. 11.6 | HTML document to demonstrate methods `indexOf` and `lastIndexOf`. (Part 3 of 3.)

```
1  // Fig. 11.7: SearchingStrings.js
2  // Searching strings with indexOf and lastIndexOf.
3  var letters = "abcdefghijklmnopqrstuvwxyzabcdefghijklm";
4
5  function buttonPressed()
6  {
7     var inputField = document.getElementById( "inputField" );
8
9     document.getElementById( "results" ).innerHTML =
10       "<p>First occurrence is located at index " +
11          letters.indexOf( inputField.value ) + "</p>" +
12       "<p>Last occurrence is located at index " +
13          letters.lastIndexOf( inputField.value ) + "</p>" +
14       "<p>First occurrence from index 12 is located at index " +
15          letters.indexOf( inputField.value, 12 ) + "</p>" +
16       "<p>Last occurrence from index 12 is located at index " +
17          letters.lastIndexOf( inputField.value, 12 ) + "</p>";
18 } // end function buttonPressed
```

Fig. 11.7 | Searching strings with `indexOf` and `lastIndexOf`. (Part 1 of 2.)

```
19
20  // register click event handler for searchButton
21  function start()
22  {
23     var searchButton = document.getElementById( "searchButton" );
24     searchButton.addEventListener( "click", buttonPressed, false );
25  } // end function start
26
27  window.addEventListener( "load", start, false );
```

Fig. 11.7 | Searching strings with `indexOf` and `lastIndexOf`. (Part 2 of 2.)

11.3.5 Splitting Strings and Obtaining Substrings

- Breaking a string into tokens is called tokenization
- Tokens are separated from one another by delimiters, typically white-space characters such as blank, tab, newline and carriage return
  - Other characters may also be used as delimiters to separate tokens
- String method `split`
  - Breaks a string into its component tokens
  - Argument is the delimiter string
  - Returns an array of strings containing the tokens
- String method `substring`
  - Returns the substring from the starting index (its first argument) up to but not including the ending index (its second argument)
  - If the ending index is greater than the length of the string, the substring returned includes the characters from the starting index to the end of the original string

```
<!DOCTYPE html>

<!-- Fig. 11.8: SplitAndSubString.html -->
<!-- HTML document demonstrating String methods split and substring. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>split and substring</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "SplitAndSubString.js"></script>
   </head>
   <body>
      <form action = "#">
         <p>Enter a sentence to split into words:</p>
         <p><input id = "inputField" type = "text">
            <input id = "splitButton" type = "button" value = "Split"></p>
         <div id = "results"></p>
      </form>
   </body>
</html>
```

Fig. 11.8 | HTML document demonstrating `String` methods `split` and `substring`. (Part 1 of 2.)

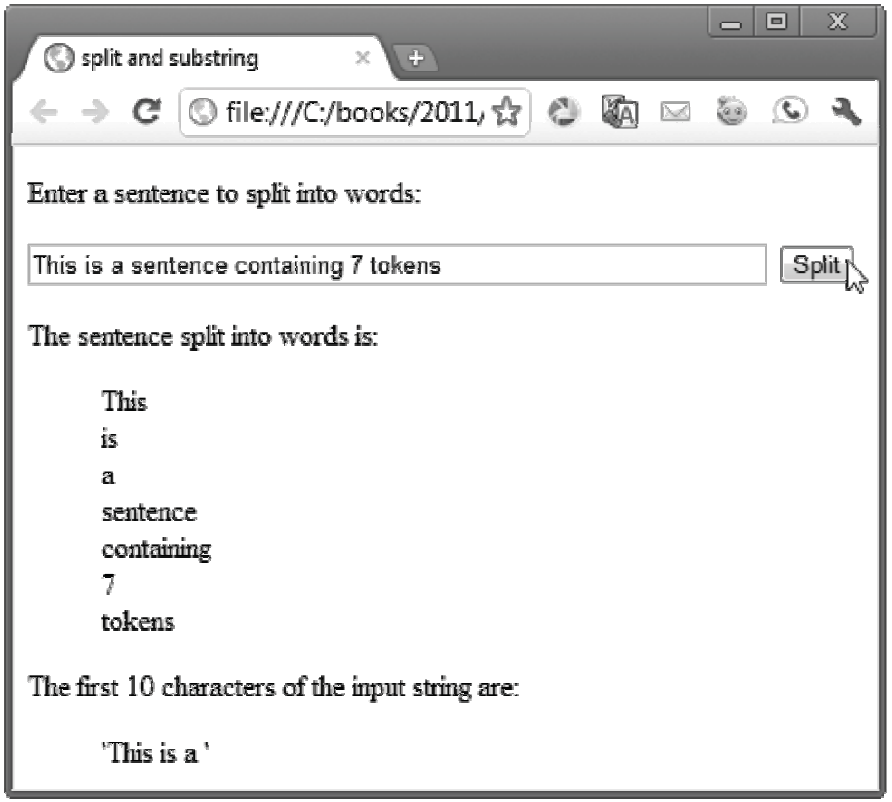


Fig. 11.8 | HTML document demonstrating `String` methods `split` and `substring`. (Part 2 of 2.)

```
1  // Fig. 11.9: SplitAndSubString.js
2  // String object methods split and substring.
3  function splitButtonPressed()
4  {
5     var inputString = document.getElementById( "inputField" ).value;
6     var tokens = inputString.split( " " );
7
8     var results = document.getElementById( "results" );
9     results.innerHTML = "<p>The sentence split into words is: </p>" +
10       "<p class = 'indent'>" +
11       tokens.join( "</p><p class = 'indent'>" ) + "</p>" +
12       "<p>The first 10 characters of the input string are: </p>" +
13       "<p class = 'indent'>'" + inputString.substring( 0, 10 ) + "'</p>";
14 } // end function splitButtonPressed
15
16 // register click event handler for searchButton
17 function start()
18 {
19    var splitButton = document.getElementById( "splitButton" );
20    splitButton.addEventListener( "click", splitButtonPressed, false );
21 } // end function start
22
23 window.addEventListener( "load", start, false );
```

Fig. 11.9 | String-object methods `split` and `substring`.

11.3.5 Splitting Strings and Obtaining Substrings (Cont.)

- delimiter string
  - the string that determines the end of each token in the original string.
- The method returns the substring from the **starting index** (0 in this example) up to but not including the **ending index** (10 in this example).
- If the ending index is greater than the length of the string, the substring returned includes the characters from the starting index to the end of the original string.

11.4 Date Object

- Date object provides methods for date and time manipulations
  - Based either on the computer's local time zone or on World Time Standard's Coordinated Universal Time (abbreviated UTC)
- Most methods have a local time zone and a UTC version
- Empty parentheses after an object name indicate a call to the object's constructor with no arguments
  - A constructor is an initializer method for an object
  - Called automatically when an object is allocated with new
  - The Date constructor with no arguments initializes the Date object with the local computer's current date and time
  - A new Date object can be initialized by passing the number of milliseconds since midnight, January 1, 1970, to the Date constructor
  - Can also create a new Date object by supplying arguments to the Date constructor for year, month, date, hours, minutes, seconds and milliseconds.
    - Hours, minutes, seconds and milliseconds arguments are all optional
    - If any one of these arguments is not specified, a zero is supplied in its place
    - If an argument is specified, all arguments to its left must be specified

Method	Description
`getDate()` `getUTCDate()`	Returns a number from 1 to 31 representing the day of the month in local time or UTC.
`getDay()` `getUTCDay()`	Returns a number from 0 (Sunday) to 6 (Saturday) representing the day of the week in local time or UTC.
`getFullYear()` `getUTCFullYear()`	Returns the year as a four-digit number in local time or UTC.
`getHours()` `getUTCHours()`	Returns a number from 0 to 23 representing hours since midnight in local time or UTC.
`getMilliseconds()` `getUTCMilliSeconds()`	Returns a number from 0 to 999 representing the number of milliseconds in local time or UTC, respectively. The time is stored in hours, minutes, seconds and milliseconds.
`getMinutes()` `getUTCMinutes()`	Returns a number from 0 to 59 representing the minutes for the time in local time or UTC.
`getMonth()` `getUTCMonth()`	Returns a number from 0 (January) to 11 (December) representing the month in local time or UTC.
`getSeconds()` `getUTCSeconds()`	Returns a number from 0 to 59 representing the seconds for the time in local time or UTC.

Fig. 11.10 | Date-object methods. (Part 1 of 4.)

Method	Description
`getTime()`	Returns the number of milliseconds between January 1, 1970, and the time in the `Date` object.
`getTimezoneOffset()`	Returns the difference in minutes between the current time on the local computer and UTC (Coordinated Universal Time).
`setDate( val )` `setUTCDate( val )`	Sets the day of the month (1 to 31) in local time or UTC.
`setFullYear( y, m, d )` `setUTCFullYear( y, m, d )`	Sets the year in local time or UTC. The second and third arguments representing the month and the date are optional. If an optional argument is not specified, the current value in the `Date` object is used.
`setHours( h, m, s, ms )` `setUTCHours( h, m, s, ms )`	Sets the hour in local time or UTC. The second, third and fourth arguments, representing the minutes, seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the `Date` object is used.
`setMilliSeconds( ms )` `setUTCMilliseconds( ms )`	Sets the number of milliseconds in local time or UTC.

Fig. 11.10 | Date-object methods. (Part 2 of 4.)

Method	Description
`setMinutes( m, s, ms )` `setUTCMinutes( m, s, ms )`	Sets the minute in local time or UTC. The second and third arguments, representing the seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the `Date` object is used.
`setMonth( m, d )` `setUTCMonth( m, d )`	Sets the month in local time or UTC. The second argument, representing the date, is optional. If the optional argument is not specified, the current date value in the `Date` object is used.
`setSeconds( s, ms )` `setUTCSeconds( s, ms )`	Sets the seconds in local time or UTC. The second argument, representing the milliseconds, is optional. If this argument is not specified, the current milliseconds value in the `Date` object is used.
`setTime( ms )`	Sets the time based on its argument—the number of elapsed milliseconds since January 1, 1970.
`toLocaleString()`	Returns a string representation of the date and time in a form specific to the computer's locale. For example, September 13, 2007, at 3:42:22 PM is represented as *09/13/07 15:47:22* in the United States and *13/09/07 15:47:22* in Europe.

Fig. 11.10 | `Date`-object methods. (Part 3 of 4.)

Method	Description
`toUTCString()`	Returns a string representation of the date and time in the form: *15 Sep 2007 15:47:22 UTC.*
`toString()`	Returns a string representation of the date and time in a form specific to the locale of the computer (*Mon Sep 17 15:47:22 EDT 2007* in the United States).
`valueOf()`	The time in number of milliseconds since midnight, January 1, 1970. (Same as `getTime`.)

Fig. 11.10 | Date-object methods. (Part 4 of 4.)

11.4 Date Object (Cont.)

- Date method parse
  - Receives as its argument a string representing a date and time and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time
- Date method UTC
  - Returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified as its arguments
  - Arguments include the required year, month and date, and the optional hours, minutes, seconds and milliseconds
  - If an argument is not specified, a 0 is supplied in its place
  - For hours, minutes and seconds, if the argument to the right of any of these arguments is specified, that argument must also be specified

```
<!DOCTYPE html>

<!-- Fig. 11.11: DateTime.html -->
<!-- HTML document to demonstrate Date-object methods. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Date and Time Methods</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "DateTime.js"></script>
   </head>
   <body>
      <h1>String representations and valueOf</h1>
      <section id = "strings"></section>
      <h1>Get methods for local time zone</h1>
      <section id = "getMethods"></section>
      <h1>Specifying arguments for a new Date</h1>
      <section id = "newArguments"></section>
      <h1>Set methods for local time zone</h1>
      <section id = "setMethods"></section>
   </body>
</html>
```

Fig. 11.11 | HTML document to demonstrate Date-object methods. (Part 1 of 3.)

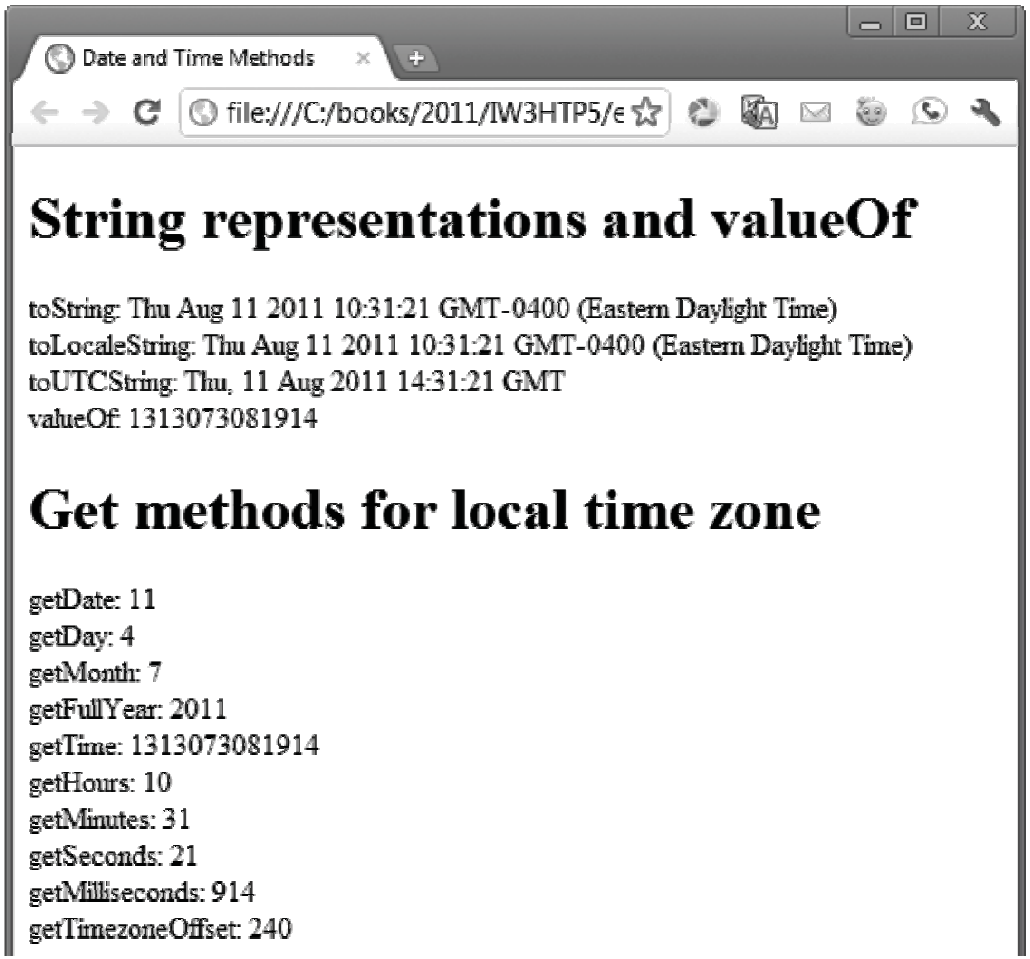


Fig. 11.11 | HTML document to demonstrate `Date`-object methods. (Part 2 of 2.)

Fig. 11.11 | HTML document to demonstrate `Date`-object methods. (Part 3 of 3.)

```
1  // Fig. 11.12: DateTime.js
2  // Date and time methods of the Date object.
3  function start()
4  {
5     var current = new Date();
6
7     // string-formatting methods and valueOf
8     document.getElementById( "strings" ).innerHTML =
9        "<p>toString: " + current.toString() + "</p>" +
10       "<p>toLocaleString: " + current.toLocaleString() + "</p>" +
11       "<p>toUTCString: " + current.toUTCString() + "</p>" +
12       "<p>valueOf: " + current.valueOf() + "</p>";
13
14    // get methods
15    document.getElementById( "getMethods" ).innerHTML =
16       "<p>getDate: " + current.getDate() + "</p>" +
17       "<p>getDay: " + current.getDay() + "</p>" +
18       "<p>getMonth: " + current.getMonth() + "</p>" +
19       "<p>getFullYear: " + current.getFullYear() + "</p>" +
20       "<p>getTime: " + current.getTime() + "</p>" +
21       "<p>getHours: " + current.getHours() + "</p>" +
22       "<p>getMinutes: " + current.getMinutes() + "</p>" +
23       "<p>getSeconds: " + current.getSeconds() + "</p>" +
24       "<p>getMilliseconds: " + current.getMilliseconds() + "</p>" +
25       "<p>getTimezoneOffset: " + current.getTimezoneOffset() + "</p>";
```

Fig. 11.12 | Date and time methods of the Date object. (Part 1 of 2.)

```
26
27       // creating a Date
28       var anotherDate = new Date( 2011, 2, 18, 1, 5, 0, 0 );
29       document.getElementById( "newArguments" ).innerHTML =
30          "<p>Date: " + anotherDate + "</p>";
31
32       // set methods
33       anotherDate.setDate( 31 );
34       anotherDate.setMonth( 11 );
35       anotherDate.setFullYear( 2011 );
36       anotherDate.setHours( 23 );
37       anotherDate.setMinutes( 59 );
38       anotherDate.setSeconds( 59 );
39       document.getElementById( "setMethods" ).innerHTML =
40          "<p>Modified date: " + anotherDate + "</p>";
41    } // end function start
42
43    window.addEventListener( "load", start, false );
```

Fig. 11.12 | Date and time methods of the `Date` object. (Part 2 of 2.)

11.5 Boolean and Number Objects

- The **Boolean** and **Number** objects are **object wrappers** for boolean `true`/`false` values and numbers, respectively
- When a boolean value is required in a JavaScript program, JavaScript automatically creates a `Boolean` object to store the value
- JavaScript programmers can create `Boolean` objects explicitly

```
var b = new Boolean( booleanValue );
```

booleanValue specifies whether the `Boolean` object should contain `true` or `false`.

  - If *booleanValue* is `false`, `0`, `null`, `Number.NaN` or the empty string (`""`), or if no argument is supplied, the new `Boolean` object contains `false`
  - Otherwise, the new `Boolean` object contains `true`

Method	Description
`toString()`	Returns the string `"true"` if the value of the `Boolean` object is `true`; otherwise, returns the string `"false"`.
`valueOf()`	Returns the value `true` if the `Boolean` object is `true`; otherwise, returns `false`.

Fig. 11.13 | `Boolean`-object methods.

11.5 Boolean and Number Objects (Cont.)

- JavaScript automatically creates `Number` objects to store numeric values in a script
- You can create a `Number` object with the statement

```
var n = new Number( numericValue );
```

  numericValue is the number to store in the object
- Although you can explicitly create `Number` objects, normally they are created when needed by the JavaScript interpreter

Method or property	Description
`toString(` *radix* `)`	Returns the string representation of the number. The optional *radix* argument (a number from 2 to 36) specifies the number's base. Radix 2 results in the *binary* representation, 8 in the *octal* representation, 10 in the *decimal* representation and 16 in the *hexadecimal* representation. See Appendix E, Number Systems, for an explanation of the binary, octal, decimal and hexadecimal number systems.
`valueOf()`	Returns the numeric value.
`Number.MAX_VALUE`	The largest value that can be stored in a JavaScript program.
`Number.MIN_VALUE`	The smallest value that can be stored in a JavaScript program.
`Number.NaN`	*Not a number*—a value returned from an arithmetic expression that doesn't result in a number (e.g., `parseInt("hello")` cannot convert the string `"hello"` to a number, so `parseInt` would return `Number.NaN`.) To determine whether a value is `NaN`, test the result with function `isNaN`, which returns `true` if the value is `NaN`; otherwise, it returns `false`.
`Number.NEGATIVE_INFINITY`	A value less than `-Number.MAX_VALUE`.
`Number.POSITIVE_INFINITY`	A value greater than `Number.MAX_VALUE`.

Fig. 11.14 | Number-object methods and properties.

11.6 document Object

- document object
  - Provided by the browser and allows JavaScript code to manipulate the current document in the browser

Method	Description
`getElementById( id )`	Returns the HTML5 element whose `id` attribute matches *id*.
`getElementByTagName( tagName )`	Returns an array of the HTML5 elements with the specified *tagName*.

Fig. 11.15 | document-object methods.

11.7 Favorite Twitter Searches

- Before HTML5, websites could store only small amounts of text-based information on a user's computer using cookies.
- A **cookie** is a *key/value pair* in which each *key* has a corresponding *value*.
- The key and value are both strings.
- Cookies are stored by the browser on the user's computer to maintain client-specific information during and between browser sessions.
- A website might use a cookie to record user preferences or other information that it can retrieve during the client's subsequent visits.
- When a user visits a website, the browser locates any cookies written by that website and sends them to the server.
- *Cookies may be accessed only by the web server and scripts of the website from which the cookies originated*

11.7 Favorite Twitter Searches (Cont.)

Problems with Cookies

- They're extremely limited in size.
- Cookies cannot store entire documents.
- If the user browses the same site from multiple tabs, all of the site's cookies are shared by the pages in each tab.
  - This could be problematic in web applications that allow the user to purchase items.

11.7 Favorite Twitter Searches (Cont.)

Introducing `localStorage` and `sessionStorage`

- As of HTML5, there are two new mechanisms for storing key/value pairs that help eliminate some of the problems with cookies.
  - Web applications can use the window object's **`localStorage` property** to store up to several megabytes of key/value-pair string data on the user's computer and can access that data across browsing sessions and browser tabs.
  - Web applications that need access to data for *only* a browsing session and that must keep that data separate among multiple tabs can use the window object's **`sessionStorage` property**. There's a separate `sessionStorage` object for every browsing session, including separate tabs that are accessing the same website.

11.7 Favorite Twitter Searches (Cont.)

Favorite Twitter Searches App Using `localStorage` and `sessionStorage`

- This app allows users to save their favorite (possibly lengthy) Twitter search strings with easy-to-remember, user-chosen, short tag names. Users can then conveniently follow the tweets on their favorite topics by visiting this web page and clicking the link for a saved search.
- The user's favorite searches are saved using `localStorage`, so they're immediately available each time the user browses the app's web page.
- The app uses `sessionStorage` to determine whether the user has visited the page previously during the current browsing session. If not, the app displays a welcome message.

a) **Favorite Twitter Searches** app when it's loaded for the first time in this browsing session and there are no tagged searches

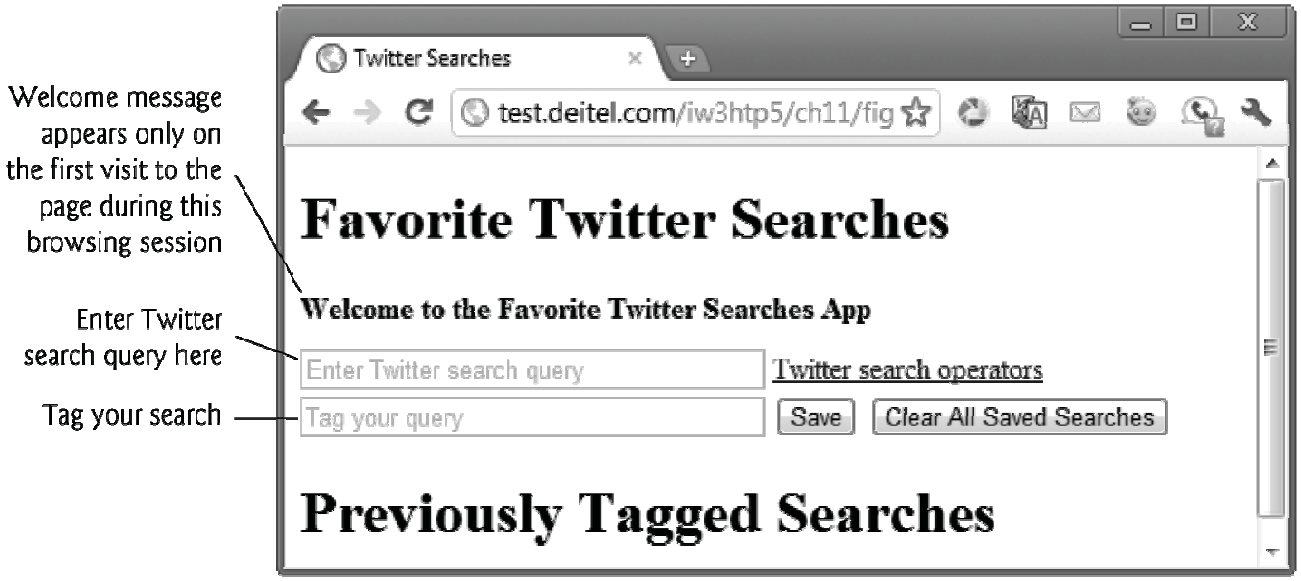


Fig. 11.16 | Sample outputs from the **Favorite Twitter Searches** web application. (Part 1 of 4.)

b) App with several saved searches and the user saving a new search

Fig. 11.16 | Sample outputs from the **Favorite Twitter Searches** web application. (Part 2 of 4.)

c) App after new search is saved—the user is about to click the Deitel search

Fig. 11.16 | Sample outputs from the **Favorite Twitter Searches** web application. (Part 3 of 4.)

d) Results of touching the **Deitel** link

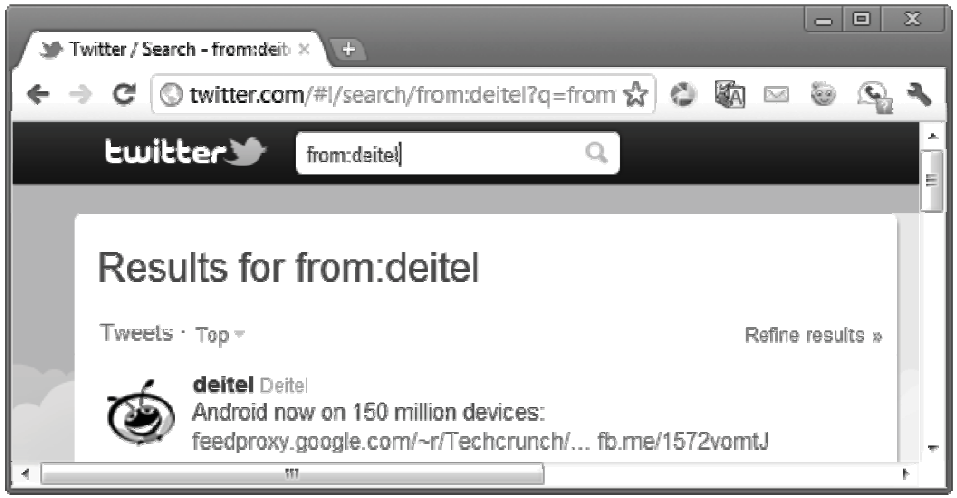


Fig. 11.16 | Sample outputs from the **Favorite Twitter Searches** web application. (Part 4 of 4.)

11.7 Favorite Twitter Searches (Cont.)

Favorite Twitter Searches HTML5 Document

- The Favorite Twitter Searches application contains three files
  - `FavoriteTwitterSearches.html`
  - `styles.css`
  - `FavoriteTwitterSearches.js`
- The HTML5 document provides a form that allows the user to enter new searches. Previously tagged searches are displayed in the `div` named `searches`.

```
<!DOCTYPE html>

<!-- Fig. 11.17: FavoriteTwitterSearchs.html -->
<!-- Favorite Twitter Searches web application. -->
<html>
<head>
   <title>Twitter Searches</title>
   <link rel = "stylesheet" type = "text/css" href = "style.css">
   <script src = "FavoriteTwitterSearches.js"></script>
</head>
<body>
   <h1>Favorite Twitter Searches</h1>
   <p id = "welcomeMessage"></p>
   <form action = "#">
      <p><input id = "query" type = "text"
         placeholder = "Entery Twitter search query">
         <a href = "https://dev.twitter.com/docs/using-search">
            Twitter search operators</a></p>
      <p><input id = "tag" type = "text" placeholder = "Tag your query">
         <input type = "button" value = "Save"
            id = "saveButton">
         <input type = "button" value = "Clear All Saved Searches"
            id = "clearButton"></p>
   </form>
```

Fig. 11.17 | Favorite Twitter Searches web application. (Part 1 of 2.)

```
25      <h1>Previously Tagged Searches</h1>
26      <div id = "searches"></div>
27   </body>
28   </html>
```

Fig. 11.17 | Favorite Twitter Searches web application. (Part 2 of 2.)

11.7 Favorite Twitter Searches (Cont.)

CSS for Favorite Twitter Searches

- `styles.css` contains the CSS styles for this app.

```
1  p { margin: 0px; }
2  #welcomeMessage { margin-bottom: 10px; font-weight: bold; }
3  input[type = "text"] { width: 250px; }
4
5  /* list item styles */
6  span { margin-left: 10px; display: inline-block; width: 100px; }
7  li { list-style-type: none; width: 220px;}
8  li:first-child { border-top: 1px solid grey; }
9  li:nth-child(even) { background-color: lightyellow;
10    border-bottom: 1px solid grey; }
11 li:nth-child(odd) { background-color: lightblue;
12    border-bottom: 1px solid grey; }
```

Fig. 11.18 | Styles used in the Favorite Twitter Searches app.

11.7 Favorite Twitter Searches (Cont.)

Script for Favorite Twitter Searches

- `FavoriteTwitterSearches.js` presents the JavaScript for the app.
- When the HTML5 document loads, function `start` is called to register event handlers and call function `loadSearches`.
- The `sessionStorage` object is used to determine whether the user has already visited the page during this browsing session.
- The **`getItem` method** receives a name of a key as an argument.
  - If the key exists, the method returns the corresponding string value; otherwise, it returns null.
- If this is the user's first visit to the page during this browsing session, the **`setItem` method** is used to set the key `"herePreviously"` to the string `"true"`, then the app displays a welcome message.

11.7 Favorite Twitter Searches (Cont.)

- The `localStorage` object's **length** represents the number of key/value pairs stored.
- Method **key** receives an index as an argument and returns the corresponding key.
- For simplicity, we use the `onclick` attributes of the dynamically generated Edit and Delete buttons to set the buttons' event handlers—this is an older mechanism for registering event handlers.
- To register these with the elements' `addEventListener` method, we'd have to dynamically locate the buttons in the page after we've created them, then register the event handlers, which would require significant additional code.
- Separately, notice that each event handler is receiving the button input element's id as an argument—this enables the event handler to use the id value when handling the event.
- [Note: The `localStorage` and `sessionStorage` properties and methods we discuss throughout this section apply to both objects.]

11.7 Favorite Twitter Searches (Cont.)

- Function `clearAllSearches` is called when the user clicks the Clear All Saved Searches button.
- The `clear` method of the `localStorage` object removes all key/value pairs from the object.
- `loadSearches` is called to refresh the list of saved searches in the web page.
- Function `saveSearch` is called when the user clicks Save to save a search.
- The `setItem` method stores a key/value pair in the `localStorage` object.
  - If the key already exits, `setItem` replaces the corresponding value;
  - otherwise, it creates a new key/value pair.
- `loadSearches` is called to refresh the list of saved searches in the web page.
- `removeItem` method is called to remove a key/value pair from the localStorage object.

```
1  // Fig. 11.19: FavoriteTwitterSearchs.js
2  // Storing and retrieving key/value pairs using
3  // HTML5 localStorage and sessionStorage
4  var tags; // array of tags for queries
5
6  // loads previously saved searches and displays them in the page
7  function loadSearches()
8  {
9     if ( !sessionStorage.getItem( "herePreviously" ) )
10    {
11       sessionStorage.setItem( "herePreviously", "true" );
12       document.getElementById( "welcomeMessage" ).innerHTML =
13          "Welcome to the Favorite Twitter Searches App";
14    } // end if
15
16    var length = localStorage.length; // number of key/value pairs
17    tags = []; // create empty array
18
19    // load all keys
20    for (var i = 0; i < length; ++i)
21    {
22       tags[i] = localStorage.key(i);
23    } // end for
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 1 of 4.)

```
24
25      tags.sort(); // sort the keys
26
27      var markup = "<ul>"; // used to store search link markup
28      var url = "http://search.twitter.com/search?q=";
29
30      // build list of links
31      for (var tag in tags)
32      {
33         var query = url + localStorage.getItem(tags[tag]);
34         markup += "<li><span><a href = '" + query + "'>" + tags[tag] +
35            "</a></span>" +
36            "<input id = '" + tags[tag] + "' type = 'button' " +
37               "value = 'Edit' onclick = 'editTag(id)'>" +
38            "<input id = '" + tags[tag] + "' type = 'button' " +
39               "value = 'Delete' onclick = 'deleteTag(id)'>";
40      } // end for
41
42      markup += "</ul>";
43      document.getElementById("searches").innerHTML = markup;
44   } // end function loadSearches
45
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 2 of 4.)

```
46  // deletes all key/value pairs from localStorage
47  function clearAllSearches()
48  {
49     localStorage.clear();
50     loadSearches(); // reload searches
51  } // end function clearAllSearches
52
53  // saves a newly tagged search into localStorage
54  function saveSearch()
55  {
56     var query = document.getElementById("query");
57     var tag = document.getElementById("tag");
58     localStorage.setItem(tag.value, query.value);
59     tag.value = ""; // clear tag input
60     query.value = ""; // clear query input
61     loadSearches(); // reload searches
62  } // end function saveSearch
63
64  // deletes a specific key/value pair from localStorage
65  function deleteTag( tag )
66  {
67     localStorage.removeItem( tag );
68     loadSearches(); // reload searches
69  } // end function deleteTag
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 3 of 4.)

```
70
71  // display existing tagged query for editing
72  function editTag( tag )
73  {
74     document.getElementById("query").value = localStorage[ tag ];
75     document.getElementById("tag").value = tag;
76     loadSearches(); // reload searches
77  } // end function editTag
78
79  // register event handlers then load searches
80  function start()
81  {
82     var saveButton = document.getElementById( "saveButton" );
83     saveButton.addEventListener( "click", saveSearch, false );
84     var clearButton = document.getElementById( "clearButton" );
85     clearButton.addEventListener( "click", clearAllSearches, false );
86     loadSearches(); // load the previously saved searches
87  } // end function start
88
89  window.addEventListener( "load", start, false );
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 4 of 4.)

11.8 Using JSON to Represent Objects

- JSON (JavaScript Object Notation)
  - A simple way to represent JavaScript objects as strings
  - introduced as an alternative to XML as a data-exchange technique
- JSON has gained acclaim due to its simple format, making objects easy to read, create and parse.
- Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

  { *propertyName1 : value1, propertyName2 : value2* }
- Arrays are represented in JSON with square brackets in the following format:

  [*value0, value1, value2*]
- Each value can be a string, a number, a JSON object, `true`, `false` or `null`.

11.8 Using JSON to Represent Objects (Cont.)

- To appreciate the simplicity of JSON data, examine this representation of an array of address-book entries that we'll use in Chapter 16:

```
[ { first: 'Cheryl', last: 'Black' },
  { first: 'James', last: 'Blue' },
  { first: 'Mike', last: 'Brown' },
  { first: 'Meg', last: 'Gold' } ]
```

- JSON provides a straightforward way to manipulate objects in JavaScript, and many other programming languages now support this format.
- In addition to simplifying object creation, JSON allows programs to easily extract data and efficiently transmit it across the Internet.