10. JavaScript: Arrays

Yea, from the table of my memory I'll wipe away all trivial fond records.

—William Shakespeare

Praise invariably implies a reference to a higher standard.

—Aristotle

With sobs and tears he sorted out Those of the largest size...

—Lewis Carroll

Attempt the end, and never stand to doubt; Nothing's so hard, but search will find it out.

—Robert Herrick

Objectives

In this chapter you'll:

- Declare arrays, initialize arrays and refer to individual elements of arrays.
- Store lists and tables of values in arrays.
- Pass arrays to functions.
- Search and sort arrays.
- Declare and manipulate multidimensional arrays.

Outline

10.1. Introduction

Arrays are data structures consisting of related data items. JavaScript arrays are "dynamic" entities in that they can change size after they're created. Many techniques demonstrated in this chapter are used frequently in Chapters 12–13 when we introduce the collections that allow you to dynamically manipulate all of an HTML5 document's elements.

10.2. Arrays

An array is a group of memory locations that all have the same name and normally are of the same type (although this attribute is *not required* in JavaScript). To refer to a particular location or element in the array, we specify the name of the array and the **position number** of the particular element in the array.

Figure 10.1 shows an array of integer values named `c`. This array contains 12 **elements**. 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 (`[]`). The first element in every array is the **zeroth element**. Thus, the first element of array `c` is referred to as `c[0]`, the second element as `c[1]`, the seventh element as `c[6]` and, in general, the *i*th element of array `c` is referred to as `c[i-1]`. Array names follow the same conventions as other identifiers.

The position number in square brackets is called an **index** and must be an integer or an integer expression. If a program uses an expression as an index, then the expression is evaluated to determine the value of the index. For example, if the variable `a` is equal to `5` and `b` is equal to `6`, then the statement

```
c[ a + b ] += 2;
```

adds `2` to the value of array element `c[11]`. An indexed array name can be used on the left side of an assignment to place a new value into an array element. It can also be used on the right side of an assignment to assign its value to another variable.

Let's examine array `c` in Fig. 10.1 more closely. The array's **name** is `c`. The array's **length** is 12 and can be found by using the array's **`length` property**, as in:

```
c.length
```

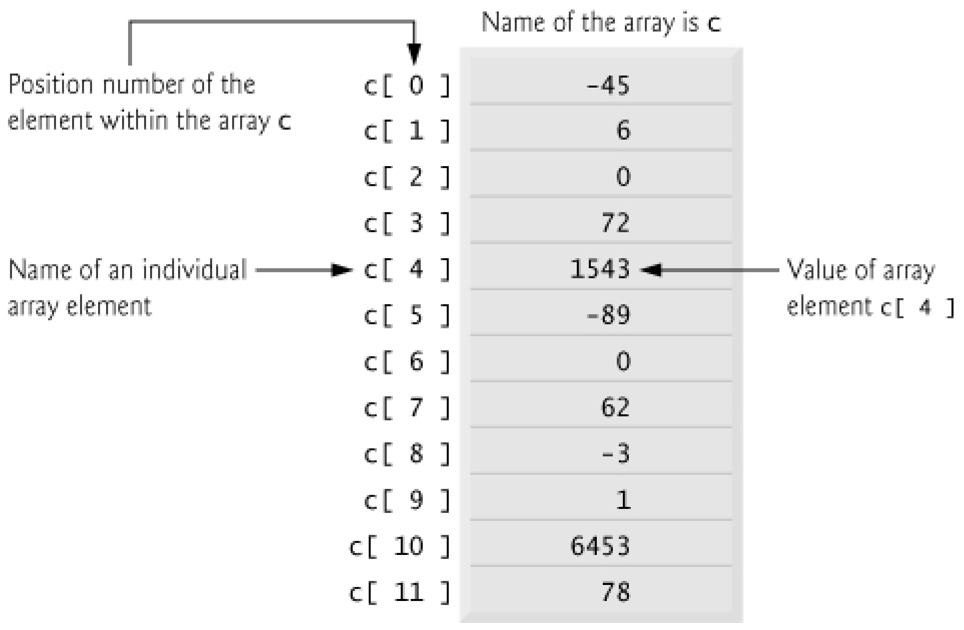
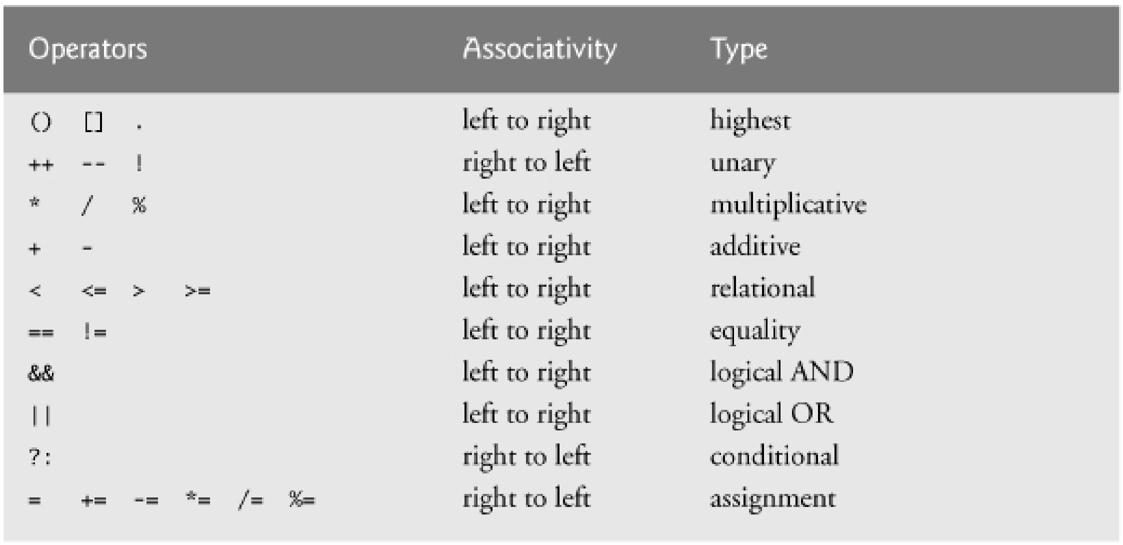


Fig. 10.1. Array with 12 elements.

The array's 12 elements are referred to as `c[0]`, `c[1]`, `c[2]`, ..., `c[11]`. The **value** of `c[0]` is `-45`, the value of `c[1]` is `6`, the value of `c[2]` is `0`, the value of `c[7]` is `62` and the value of `c[11]` is `78`. The following statement calculates the sum of the values contained in the first three elements of array `c` and stores the result in variable `sum`:

sum = c[**0**] + c[**1**] + c[**2**];

The brackets that enclose an array index are a JavaScript operator. Brackets have the same level of precedence as parentheses. Figure 10.2 shows the precedence and associativity of the operators introduced so far in the text. They're shown from top to bottom in decreasing order of precedence.



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

Arrays occupy space in memory. Actually, an array in JavaScript is an **`Array` object**. 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`. To allocate 12 elements for integer array `c`, use a `new` expression like:

```
var c = new Array( 12 );
```

The preceding statement can also be performed in two steps, as follows:

```
var c; // declares a variable that will hold the array
c = new Array( 12 ); // allocates the array
```

When arrays are created, the elements are *not* initialized—they have the value `undefined`.

10.4. Examples Using Arrays

This section presents several examples of creating and manipulating arrays.

10.4.1. Creating, Initializing and Growing Arrays

Our next example (Figs. 10.3–10.4) uses operator `new` to allocate an array of five elements and an empty array. The script demonstrates initializing an array of existing elements and also shows that an array can grow dynamically to accommodate new elements. The array's values are displayed in HTML5 tables.

HTML5 Document for Displaying Results

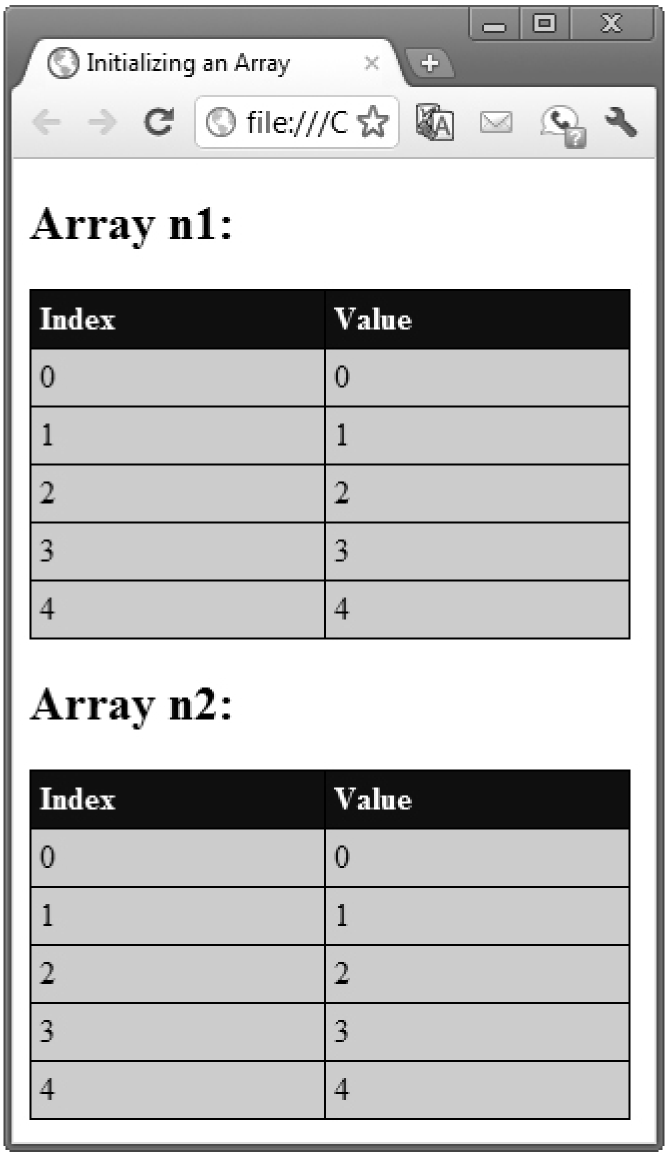
Figure 10.3 shows the HTML5 document in which we display the results. You'll notice that we've placed the CSS styles and JavaScript code into separate files. Line 9 links the CSS file `tablestyle.css` to this document as shown in Chapter 4. (There are no new concepts in the CSS file used in this chapter, so we don't show them in the text.) Line 10 demonstrates how to link a script that's stored in a separate file to this document. To do so, use the `script` element's **`src` attribute** to specify the location of the JavaScript file (named with the **`.js` filename extension**). This document's `body` contains two `div` s in which we'll display the contents of two arrays. When the document finishes loading, the JavaScript function `start` (Fig. 10.4) is called.

SOFTWARE ENGINEERING OBSERVATION 10.1

It's considered good practice to separate your JavaScript scripts into separate files so that they can be reused in multiple web pages.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 10.3: InitArray.html -->
4  <!-- Web page for showing the results of initializing arrays. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
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>
```



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

Fig. 10.3. Web page for showing the results of initializing arrays.

Script that Creates, Initializes and Displays the Contents of Arrays

Figure 10.4 presents the script used by the document in Fig. 10.3. Function `start` (lines 3–24) is called when the `window`'s `load` event occurs.

```
// Fig. 10.4: InitArray.js
// Create two arrays, initialize their elements and display them
function start()
{
  var n1 = new Array( 5 ); // allocate five-element array
  var n2 = new Array(); // allocate empty array

  // assign values to each element of array n1
  var length = n1.length; // get array's length once before the loop

   for ( var i = 0; i < length; ++i )
   {
     n1[ i ] = i;
   } // end for

   // create and initialize five elements in array n2
   for ( i = 0; i < 5; ++i )
   {
     n2[ i ] = i;
   } // end for

   outputArray( "Array n1:", n1, document.getElementById( "output1" )
);
   outputArray( "Array n2:", n2, document.getElementById( "output2" )
);
} // end function start

// output the heading followed by a two-column table
// containing indices and elements of "theArray"
function outputArray( heading, theArray, output )
{
   var content = "<h2>" + heading + "</h2><table>" +
     "<thead><th>Index</th><th>Value</th></thead><tbody>";

   // 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.

Line 5 creates array `n1` with five elements. Line 6 creates array `n2` as an *empty* array. Lines 9–14 use a `for` statement to initialize the elements of `n1` to their index values (0 to 4). With arrays, we use zero-based counting so that the loop can access *every* array element. Line 9 uses the expression `n1.length` to determine the array's length. JavaScript's arrays are dynamically resizable, so it's important to get an array's length once before a loop that processes the array—in case the script changes the array's length. In this example, the array's length is 5, so the loop continues executing as long as the value of control variable `i` is less than `5`. This process is known as **iterating through the array's elements**. For a five-element array, the index values are 0 through 4, so using the less-than operator, `<`, guarantees that the loop does not attempt to access an element beyond the end of the array. Zero-based counting is usually used to iterate through arrays.

Growing an Array Dynamically

Lines 17–20 use a `for` statement to add five elements to the array `n2` and initialize each element to its index value (0 to 4). The array grows dynamically to accommodate each value as it's assigned to each element of the array.

SOFTWARE ENGINEERING OBSERVATION 10.2

JavaScript automatically reallocates an array when a value is assigned to an element that's outside the bounds of the array. Elements between the last element of the original array and the new element are `undefined`.

Lines 22–23 invoke function `outputArray` (defined in lines 28–44) to display the contents of each array in an HTML5 table in a corresponding `div`. Function `outputArray` receives three arguments—a string to be output as an `h2` element before the HTML5 table that displays the contents of the array, the array to output and the `div` in which to place the table. Lines 36–40 use a `for` statement to define each row of the table.

ERROR-PREVENTION TIP 10.1

When accessing array elements, the index values should never go below 0 and should be less than the number of elements in the array (i.e., one less than the array's size), unless it's your explicit intent to grow the array by assigning a value to a nonexistent element.

Using an Initializer List

If an array's element values are known in advance, the elements can be allocated and initialized in the declaration of the array. There are two ways in which the initial values can be specified. The statement

```
var n = [ 10, 20, 30, 40, 50 ];
```

uses a comma-separated **initializer list** enclosed in square brackets (`[` and `]`) to create a five-element array with indices of `0`, `1`, `2`, `3` and `4`. The array size is determined by the number of values in the initializer list. The preceding declaration does *not* require the `new` operator to create the `Array` object—this functionality is provided by the JavaScript interpreter when it encounters an array declaration that includes an initializer list. The statement

```
var n = new Array( 10, 20, 30, 40, 50 );
```

also creates a five-element array with indices of `0`, `1`, `2`, `3` and `4`. In this case, the initial values of the array elements are specified as arguments in the parentheses following `new Array`. The size of the array is determined by the number of values in parentheses. It's also possible to reserve a space in an array for a value to be specified later by using a comma as a **place holder** in the initializer list. For example, the statement

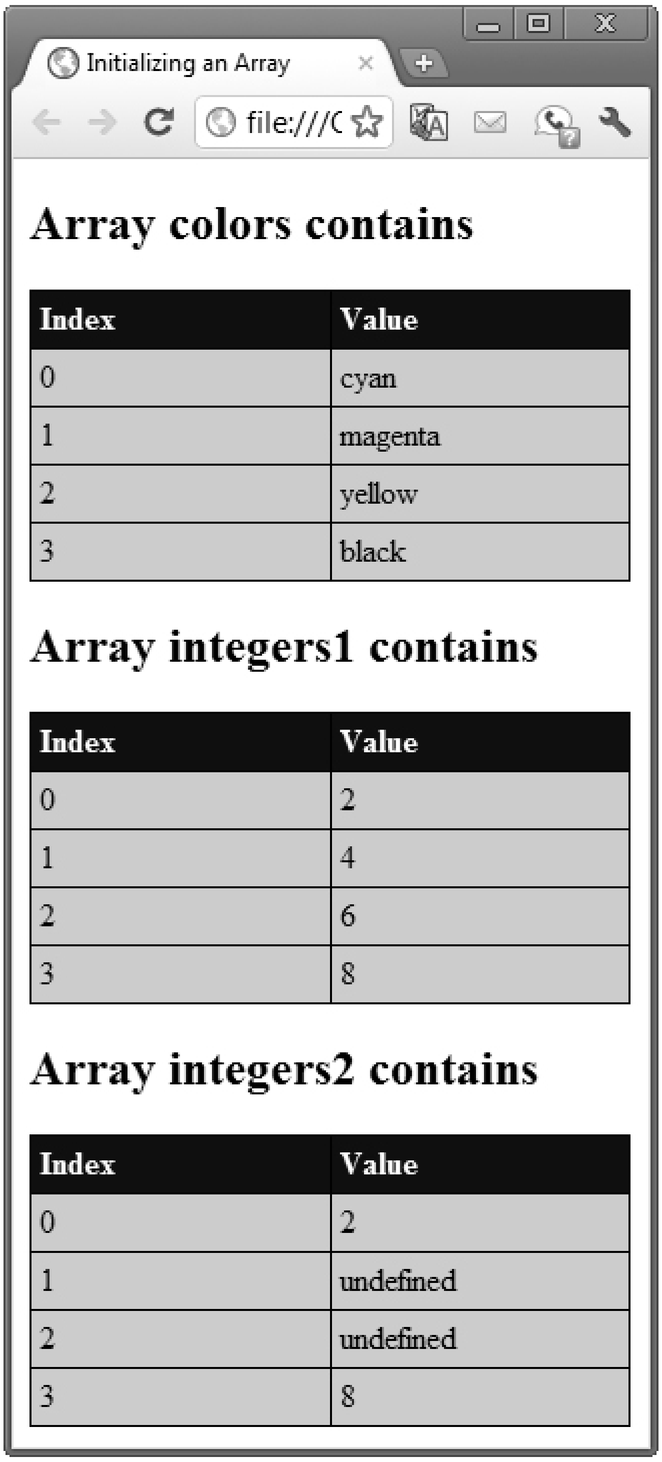
```
var n = [ 10, 20,, 40, 50 ];
```

creates a five-element array in which the third element (`n[2]`) has the value `undefined`.

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>
```



Array colors contains

Index	Value
0	cyan
1	magenta
2	yellow
3	black

Array integers1 contains

Index	Value
0	2
1	4
2	6
3	8

Array integers2 contains

Index	Value
0	2
1	undefined
2	undefined
3	8

Fig. 10.5. Web page for showing the results of initializing arrays.

The `start` function in Fig. 10.6 demonstrates array initializer lists (lines 7–9) and displays each array in an HTML5 table using the same function `outputArray` as Fig. 10.4. Note that when array `integers2` is displayed in the web page, the elements with indices 1 and 2 (the second and third elements of the array) appear in the web page as `undefined`. These are the two elements for which we did not supply values in line 9.

```
// Fig. 10.6: InitArray2.js
// Initializing arrays with initializer lists.
function start()
{
  // Initializer list specifies the number of elements and
  // a value for each element.
  var colors = new Array( "cyan", "magenta","yellow", "black" );
  var integers1 = [ 2, 4, 6, 8 ];
  var integers2 = [ 2,,, 8 ];

   outputArray( "Array colors contains", colors,
     document.getElementById( "output1" ) );
   outputArray( "Array integers1 contains", integers1,
     document.getElementById( "output2" ) );
   outputArray( "Array integers2 contains", integers2,
     document.getElementById( "output3" ) );
} // end function start

// output the heading followed by a two-column table
// containing indices and elements of "theArray"
function outputArray( heading, theArray, output )
{
   var content = "<h2>" + heading + "</h2><table>" +
     "<thead><th>Index</th><th>Value</th></thead><tbody>";

   // output the index and value of each array element
   var length = theArray.length; // get array's length once before loop

   for ( var i = 0; i < length; ++i )
   {
     content += "<tr><td>" + i + "</td><td>" + theArray[ i ] +
       "</td></tr>";
   } // end for

   content += "</tbody></table>";
   output.innerHTML = content; // place the table in the output element
} // end function outputArray

window.addEventListener( "load", start, false );
```

Fig. 10.6. Initializing arrays with initializer lists.

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

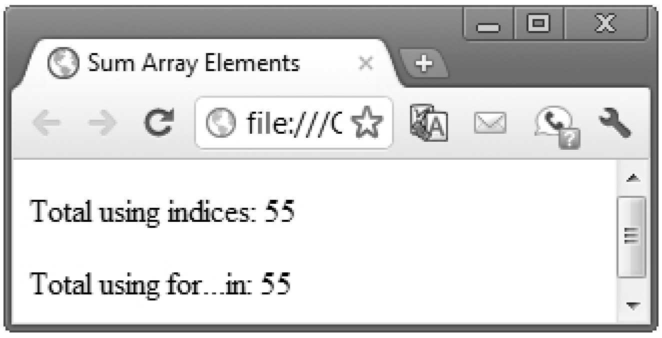
The example in Figs. 10.7–10.8 sums an array’s elements and displays the results. The document in Fig. 10.7 shows the results of the script in Fig. 10.8.

```
<!DOCTYPE html>

<!-- Fig. 10.7: SumArray.html -->
<!-- HTML5 document that displays the sum of an array's elements. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Sum Array Elements</title>
    <script src = "SumArray.js"></script>
  </head>
  <body>
    <div id = "output"></div>
  </body>
</html>
```

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

The script in Fig. 10.8 sums the values contained in `theArray`, the 10-element integer array declared, allocated and initialized in line 5. The statement in line 14 in the body of the first `for` statement does the totaling.

```
// Fig. 10.8: SumArray.js
// Summing the elements of an array with for and for...in
function start()
{
  var theArray = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
  var total1 = 0, total2 = 0;

  // iterates through the elements of the array in order and adds
  // each element's value to total1
   var length = theArray.length; // get array's length once before loop

   for ( var i = 0; i < length; ++i )
   {
     total1 += theArray[ i ];
   } // end for

   var results = "<p>Total using indices: " + total1 + "</p>";

   // iterates through the elements of the array using a for... in
   // statement to add each element's value to total2
   for ( var element in theArray )
   {
     total2 += theArray[ element ];
   } // end for

   results += "<p>Total using for...in: " + total2 + "</p>";
   document.getElementById( "output" ).innerHTML = results;
 } // end function start

 window.addEventListener( "load", start, false );
```



Total using indices: 55

Total using for...in: 55

Fig. 10.8. Summing the elements of an array with `for` and `for` ... `in` .

The `for` ... `in` Repetition Statement

In this example, we introduce JavaScript's **`for` ... `in` statement**, which enables a script to perform a task for each element in an array (or, as we'll see in Chapters 12–13, for each element in a *collection*). Lines 21–24 show the syntax of a `for` ... `in` statement. Inside the parentheses, we declare the `element` variable used to select each element in the object to the right of keyword `in` (`theArray` in this case). When you use `for` ... `in` , JavaScript automatically determines the number of elements in the array. As the JavaScript interpreter iterates over `theArray` 's elements, variable `element` is assigned a value that can be used as an index for `theArray` . In the case of an array, the value assigned is an index in the range from `0` up to, but not including, `theArray.length` . Each value is added to `total2` to produce the sum of the elements in the array.

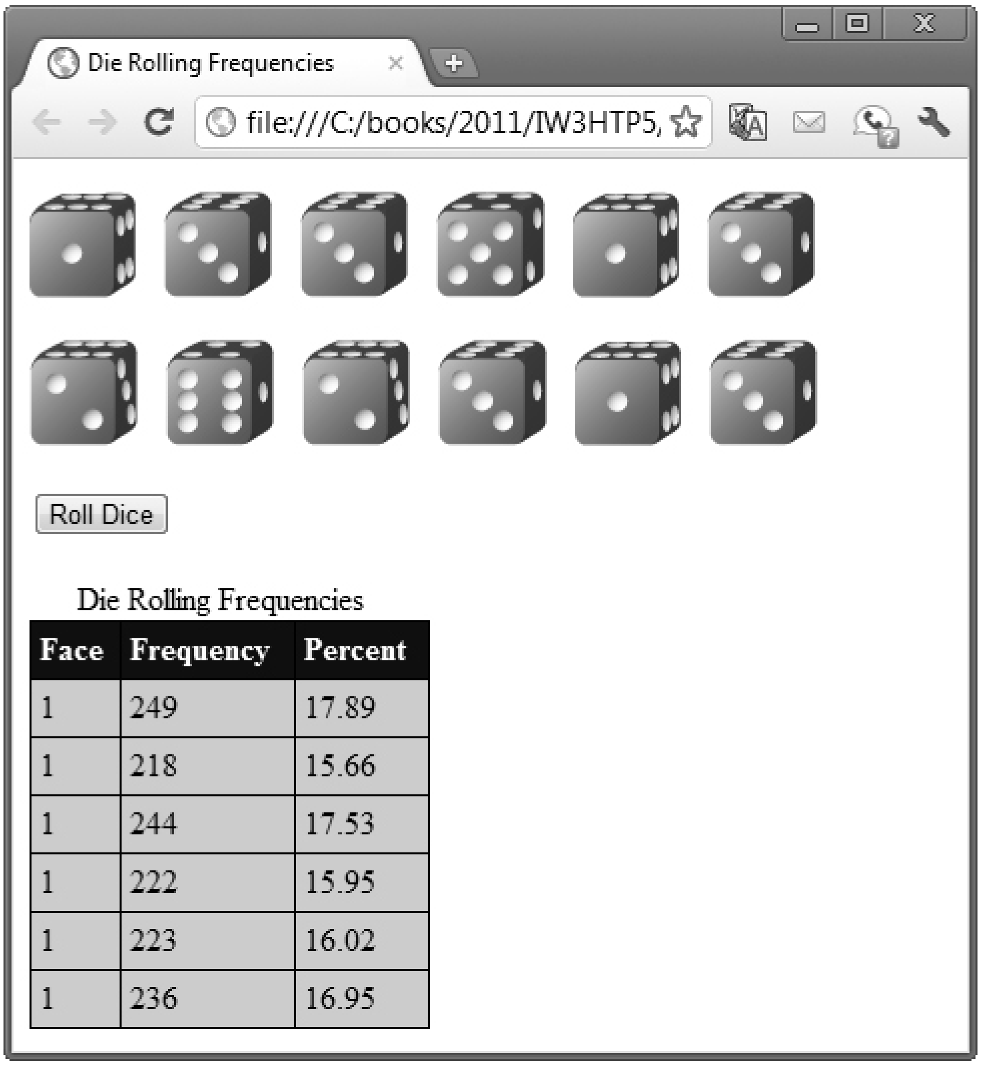
ERROR-PREVENTION TIP 10.2

When iterating over all the elements of an array, use a `for` ... `in` statement to ensure that you manipulate only the existing elements. The `for` ... `in` statement skips any undefined elements in the array.

10.4.4. Using the Elements of an Array as Counters

In Section 9.5.3, we indicated that there's a more elegant way to implement the dice-rolling example presented in that section. The example 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>
     <p><img id = "die7" src = "blank.png" alt = "die 7 image">
       <img id = "die8" src = "blank.png" alt = "die 8 image">
       <img id = "die9" src = "blank.png" alt = "die 9 image">
       <img id = "die10" src = "blank.png" alt = "die 10 image">
       <img id = "die11" src = "blank.png" alt = "die 11 image">
       <img id = "die12" src = "blank.png" alt = "die 12 image"></p>
     <form action = "#">
       <input id = "rollButton" type = "button" value = "Roll Dice">
     </form>
     <div id = "frequencyTableDiv"></div>
  </body>
</html>
```



Die Rolling Frequencies

Face	Frequency	Percent
1	249	17.89
1	218	15.66
1	244	17.53
1	222	15.95
1	223	16.02
1	236	16.95

Fig. 10.9. HTML5 document for the dice-rolling example.

In Fig. 10.10, lines 3–5 declare the scripts global variables. The `frequency` array (line 3) contains seven elements representing the counters we use in this script. We ignore element 0 of the array and use only the elements that correspond to values on the sides of a die (the elements with indices 1–6). Variable `totalDice` tracks the total number of dice rolled. The `dieImages` array contains 12 elements that will refer to the 12 `img` elements in the HTML document (Fig. 10.9).

```
1  // Fig. 10.10: RollDice.js
2  // Summarizing die-rolling frequencies with an array instead of a switch
```

```
var frequency = [, 0, 0, 0, 0, 0, 0 ]; // frequency[0] uninitialized
var totalDice = 0;
var dieImages = new Array(12); // array to store img elements

// get die img elements
function start()
{
   var button = document.getElementById( "rollButton" );
   button.addEventListener( "click", rollDice, false );
   var length = dieImages.length; // get array's length once before loop

   for ( var i = 0; i < length; ++i )
   {
      dieImages[ i ] = document.getElementById( "die" + (i + 1) );
   } // end for
} // end function start

// roll the dice
function rollDice()
{
   var face;  // face rolled
   var length = dieImages.length;

   for ( var i = 0; i < length; ++i )
   {
      face = Math.floor( 1 + Math.random() * 6 );
      tallyRolls( face ); // increment a frequency counter
      setImage( i, face ); // display appropriate die image
      ++totalDice; // increment total
   } // end for

   updateFrequencyTable();
} // end function rollDice

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

```
{
  ++frequency[ face ]; // increment appropriate counte
} // end function tallyRolls

// set image source for a die
function setImage( dieImg )
{
  dieImages[ dieNumber ].setAttribute( "src", "die" + face + ".png" );
  dieImages[ dieNumber ].setAttribute( "alt",
    "die with " + face + " spot(s)" );
} // end function setImage

// update frequency table in the page
function updateFrequencyTable()
{
  var results = "<table><caption>Die Rolling Frequencies</caption>" +
    "<thead><th>Face</th><th>Frequency</th>" +
    "<th>Percent</th></thead><tbody>";
  var length = frequency.length;

  // create table rows for frequencies
  for ( var i = 1; i < length; ++i )
  {
    results += "<tr><td>1</td><td>" + i + "</td><td>" +
      formatPercent(frequency[ i ] / totalDice) + "</td></tr>";
  } // end for

  results += "</tbody></table>";
  document.getElementById( "frequencyTableDiv" ).innerHTML = results;
} // end function updateFrequencyTable

// format percentage
function formatPercent( value )
{
  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`.

When the document finishes loading, the script's `start` function (lines 8–18) is called to register the button's event handler and to get the `img` elements and store them in the global array `dieImages` for use in the rest of the script. Each time the user clicks the **Roll Dice** button, function `rollDice` (lines 21–35) is called to roll 12 dice and update the results on the page.

The `switch` statement in Fig. 9.6 is replaced by line 40 in function `tallyRolls`. This line uses the random `face` value (calculated at line 28) as the index for the array `frequency` to determine which element to increment during each iteration of the loop. Because the random number calculation in line 28 produces numbers from 1 to 6 (the values for a six-sided die), the `frequency` array must have seven elements (index values 0 to 6). Also, lines 60–64 of this program generate the table rows that were written one line at a time in Fig. 9.6. We can loop through array `frequency` to help produce the output, so we do not have to enumerate each HTML5 table row as we did in Fig. 9.6.

10.5. Random Image Generator Using Arrays

In Chapter 9, we created a random image generator that 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`). In this example (Figs. 10.11–10.12), we create a more elegant random image generator that does not require the image filenames to contain integers in sequence. This version uses an array `pictures` to store the names of the image files as strings. Each time you click the image in the document (Fig. 10.11), 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. In addition, we update the `alt` attribute with an appropriate description of the image from the `descriptions` array.

```
<!DOCTYPE html>

<!-- Fig. 10.11: RandomPicture.html -->
<!-- HTML5 document that displays randomly selected images. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Random Image Generator</title>
    <script src = "RandomPicture.js"></script>
  </head>
  <body>
    <img id = "image" src = "CPE.png" alt = "Common Programming Error">
  </body>
</html>
```

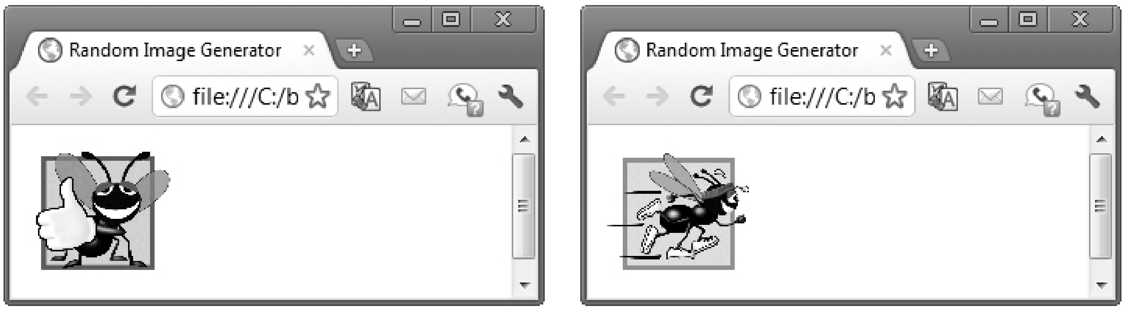


Fig. 10.11. HTML5 document that displays randomly selected images.

The script (Fig. 10.12) declares the array `pictures` in line 4 and initializes it with the names of seven image files—the files contain our bug icons that we associate with our programming tips. Lines 5–8 create a separate array `descriptions` that contains the `alt` text for the corresponding images in the `pictures` array. When the user clicks the `img` element in the document, function `pickImage` (lines 12–17) is called to pick a random integer `index` from 0 to 6 and display the associated image.

Line 15 uses that `index` to get a value from the `pictures` array, appends `".png"` to it, then sets the `img` element's `src` attribute to the new image file name. Similarly, line 16 uses the `index` to get the corresponding text from the `descriptions` array and assigns that text to the `img` element's `alt` attribute.

```
 1  // Fig. 10.12: RandomPicture2.js
 2  // Random image selection using arrays
 3  var iconImg;
 4  var pictures = [ "CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO" ];
 5  var descriptions = [ "Common Programming Error",
 6     "Error-Prevention Tip", "Good Programming Practice",
 7     "Look-and-Feel Observation", "Performance Tip", "Portability Tip",
 8     "Software Engineering Observation" ];
 9
10  // pick a random image and corresponding description, then modify
11  // the img element in the document's body
12  function pickImage()
13  {
14     var index = Math.floor( Math.random() * 7 );
15     iconImg.setAttribute( "src", pictures[ index ] + ".png" );
16     iconImg.setAttribute( "alt", descriptions[ index ] );
17  } // end function pickImage
18
19  // registers iconImg's click event handler
20  function start()
21  {
22     iconImg = document.getElementById( "iconImg" );
23     iconImg.addEventListener( "click", pickImage, false );
24  } // end function start
25
26  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) in many programming languages are **pass-by-value** and **pass-by-reference**. When an argument is passed to a function by value, a *copy* of the argument's value is made and is passed to the called function. In JavaScript, numbers, boolean values and strings are passed to functions by value.

With 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. This procedure is accomplished by passing to the called function the **address in memory** where the data resides. Pass-by-reference 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. In JavaScript, all objects (and thus all arrays) are passed to functions by reference.

ERROR-PREVENTION TIP 10.3

With pass-by-value, changes to the copy of the value received by the called function do not affect the original variable's value in the calling function. This prevents the accidental side effects that hinder the development of correct and reliable software systems.

SOFTWARE ENGINEERING OBSERVATION 10.3

When information is returned from a function via a `return` *statement, numbers and boolean values are returned by value (i.e., a copy is returned), and objects are returned by reference (i.e., a reference to the object is returned). When an object is passed-by-reference, it's not necessary to return the object, because the function operates on the original object in memory.*

The name of an array actually is a *reference* to an object that contains the array elements and the `length` variable. To pass a reference to an object into a function, simply specify the reference name in the function call. The reference name is the identifier that the program uses to manipulate the object. Mentioning the reference by its parameter name in the body of the called function actually refers to the original object in memory, and the original object is accessed directly by the called function.

10.7. Passing Arrays to Functions

To pass an array argument to a function, specify the array's name (a reference to the array) without brackets. For example, if array `hourlyTemperatures` has been declared as

```
var hourlyTemperatures = new Array( 24 );
```

then the function call

```
modifyArray( hourlyTemperatures );
```

passes array `hourlyTemperatures` to function `modifyArray`. As stated in Section 10.2, every array object in JavaScript knows its own size (via the `length` attribute). Thus, when we pass an array object into a function, we do not pass the array's size separately as an argument. Figure 10.4 demonstrated this concept.

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**. Objects referred to by individual array elements are still passed by reference. To pass an array element to a function, use the indexed name of the element as an argument in the function call.

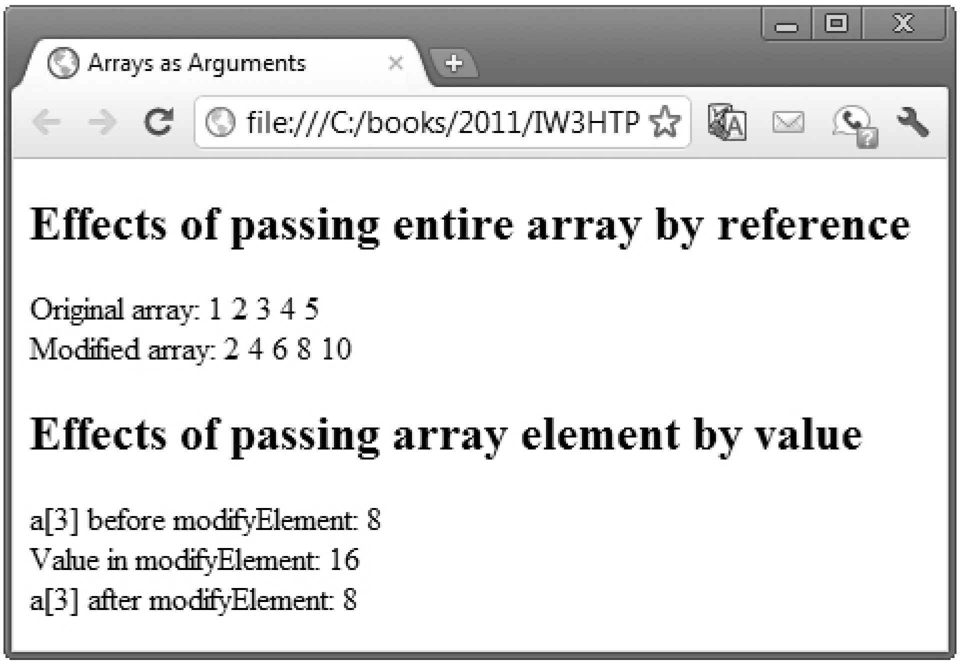
For a function to receive an array through a function call, the function's parameter list must specify a parameter that will refer to the array in the body of the function. JavaScript does not provide a special syntax for this purpose—it simply requires that the identifier for the array be specified in the parameter list. For example, the function header for function `modifyArray` might be written as

```
function modifyArray( b )
```

indicating that `modifyArray` expects to receive a parameter named `b`. Arrays are passed by reference, and therefore when the called function uses the array name `b`, it refers to the actual array in the caller (array `hourlyTemperatures` in the preceding call). The script in Figures 10.13–10.14 demonstrates the difference between passing an entire array and passing an array element. The `body` of the document in Fig. 10.13 contains the `p` elements that the script in Fig. 10.14 uses to display the results.

```
<!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>
```

```
21    </body>
22 </html>
```



Effects of passing entire array by reference

Original array: 1 2 3 4 5
Modified array: 2 4 6 8 10

Effects of passing array element by value

a[3] before modifyElement: 8
Value in modifyElement: 16
a[3] after modifyElement: 8

Fig. 10.13. HTML document that demonstrates passing arrays and individual array elements to functions.

SOFTWARE ENGINEERING OBSERVATION 10.4

JavaScript does not check the number of arguments or types of arguments that are passed to a function. It's possible to pass any number of values to a function.

When the document of Fig. 10.13 loads, function `start` (Fig. 10.14, lines 3–20) is called. Lines 8–9 invoke `outputArray` to display the array `a`'s contents before it's modified. Function `outputArray` (lines 23–26) receives a string to display, the array to display and the element in which to place the content. Line 25 uses `Array` method **`join`** to create a string containing all the elements in `theArray`. Method `join` takes as its argument a string containing the **separator** that should be used to separate the array elements in the string that's returned. If the argument is not specified, the empty string is used as the separator.

Line 10 invokes `modifyArray` (lines 29–35) and passes it array `a`. The function multiplies each element by 2. To illustrate that array `a`'s elements were modified, lines 11–12 invoke `outputArray` again to display the array `a`'s contents after it's modified. As the screen capture in Fig. 10.13 shows, the elements of `a` are indeed modified by `modifyArray`.

```
 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
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.

Lines 15–16 display the value of `a[3]` before the call to `modifyElement`. Line 17 invokes `modifyElement` (lines 38–43), passing `a[3]` as the argument. Remember that `a[3]` actually is one integer value in the array, and that numeric values and boolean values are always passed to functions by value. Therefore, a *copy* of `a[3]` is passed. Function `modifyElement` multiplies its argument by 2, stores the result in its parameter `e`, then displays `e`'s value. A parameter is a local variable in a function, so when the function terminates, the local variable is no longer accessible. Thus, when control is returned to `start`, the unmodified original value of `a[3]` is displayed by the statement in lines 18–19.

10.8. Sorting Arrays with `Array` Method `sort`

Sorting data (putting data in a particular order, such as ascending or descending) is one of the most important computing functions. The `Array` object in JavaScript has a built-in method **`sort`** for sorting arrays. The example in Figs. 10.15–10.16 demonstrates the `Array` object's `sort` method. The unsorted and sorted values are displayed in Figs. 10.15's paragraph elements (lines 14–15).

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 10.15: Sort.html -->
4  <!-- HTML5 document that displays the results of sorting an array. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Array Method sort</title>
9        <link rel = "stylesheet" type = "text/css" href = "style.css">
10       <script src = "Sort.js"></script>
11    </head>
12    <body>
13       <h1>Sorting an Array</h1>
14       <p id = "originalArray"></p>
15       <p id = "sortedArray"></p>
16    </body>
17 </html>
```

Fig. 10.15. HTML5 document that displays the results of sorting an array.

By default, `Array` method `sort` (with no arguments) uses *string* comparisons to determine the sorting order of the array elements. The strings are compared by the ASCII values of their characters. [*Note:* String comparison is discussed in more detail in Chapter 11.] In this script (Fig. 10.16), we'd like to sort an array of *integers*.

Method `sort` (line 9) takes as its argument the name of a **comparator function** that compares its two arguments and returns one of the following:

- a negative value if the first argument is *less than* the second argument,

---

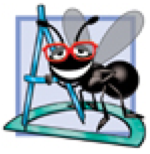
```
 1  // Fig. 10.16: Sort.js
 2  // Sorting an array with sort.
 3  function start()
 4  {
 5     var a = [ 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 ];
 6
 7     outputArray( "Data items in original order: ", a,
 8        document.getElementById( "originalArray" ) );
 9     a.sort( compareIntegers );  // sort 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
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` .

- zero if the arguments are *equal*, or
- a positive value if the first argument is *greater than* the second argument.

This example uses the comparator function `compareIntegers` (defined in lines 21–24). It calculates the difference between the integer values of its two arguments (function `parseInt` ensures that the arguments are handled properly as integers).

Line 9 invokes `Array` object `a` 's `sort` method and passes function `compareIntegers` as an argument. Method `sort` then uses function `compareIntegers` to compare elements of the array `a` to determine their sorting order.



SOFTWARE ENGINEERING OBSERVATION 10.5

Functions in JavaScript are considered to be data. Therefore, functions can be assigned to variables, stored in arrays and passed to functions just like other data types.

10.9. Searching Arrays with `Array` Method `indexOf`

When working with data stored in arrays, 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, and 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. The example in Figs. 10.17–10.18 demonstrates method `indexOf` . You enter the integer search key in the `form` 's `number` input element (Fig. 10.17, line 14) then press the `button` (lines 15–16) to invoke the script’s `buttonPressed` function, which performs the search and displays the results in the paragraph at line 17.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 10.17: search.html -->
4  <!-- HTML5 document for searching an array with indexOf. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Search an Array</title>
9        <script src = "search.js"></script>
10    </head>
11    <body>
12       <form action  = "#">
13          <p><label>Enter integer search key:
14             <input id = "inputVal" type = "number"></label>
15             <input id = "searchButton" type = "button" value = "Search">
16          </p>
17          <p id = "result"></p>
18       </form>
19    </body>
20 </html>
```

Fig. 10.17. HTML5 document for searching an array with `indexOf`.

The script in Fig. 10.18 creates an array containing 100 elements (line 3), then initializes the array's elements with the even integers from 0 to 198 (lines 6–9). When the user presses the `button` in Fig. 10.17, function `buttonPressed` (lines 12–32) performs the search and displays the results. Line 15 gets the `inputVal` `number` `input` element, which contains the key value specified by the user, and line 18 gets the paragraph where the script displays the results. Next, we get the integer value entered by the user (line 21). Every `input` element has a **`value` property** that can be used to get or set the element's value. Finally, line 22 performs the search by calling method `indexOf` on the array `a`, and lines 24–31 display the results.

```
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 );
```

```

   if ( element != -1 )
   {
     result.innerHTML = "Found value in element " + element;
   } // end if
   else
   {
     result.innerHTML = "Value not found";
   } // end else
} // end function buttonPressed

// register searchButton's click event handler
function start()
{
   var searchButton = document.getElementById( "searchButton" );
   searchButton.addEventListener( "click", buttonPressed, false );
} // end function start

window.addEventListener( "load", start, false );
```

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

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. For example, the 100-element array in Fig. 10.18 has indices from 0 to 99. If we pass `-10` as the second argument, the search will begin from index 89. If a negative second argument results in an index value less than 0 as the start point, the entire array will be searched.

10.10. Multidimensional Arrays

Multidimensional arrays with two indices are often used to represent *tables* of values consisting of information arranged in **rows** and **columns**. To identify a particular table element, we must specify the two indices; by convention, the first identifies the element's row and the second the element's column. Arrays that require two indices to identify a particular element are called **two-dimensional arrays**.

Multidimensional arrays can have *more* than two dimensions. JavaScript does not support multidimensional arrays directly, but it does allow you to specify arrays whose elements are also arrays, thus achieving the same effect. When an array contains one-dimensional arrays as its elements, we can imagine these one-dimensional arrays as rows of a table, and the positions in these arrays as columns. Figure 10.19 illustrates a two-dimensional array named `a` that contains three rows and four columns (i.e., a three-by-four array—three one-dimensional arrays, each with four elements). In general, an array with *m* rows and *n* columns is called an ***m*-by-*n* array**.

Every element in array `a` is identified in Fig. 10.19 by an element name of the form `a[row][column]` — `a` is the name of the array, and `row` and `column` are the indices that uniquely identify the row and column, respectively, of each element in `a`. The element names in row 0 all have a first index of `0`; the element names in column 3 all have a second index of `3`.

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

Arrays of One-Dimensional Arrays

Multidimensional arrays can be initialized in declarations like a one-dimensional array. Array `b` with two rows and two columns could be declared and initialized with the statement

```
var b = [ [ 1, 2 ], [ 3, 4 ] ];
```

The values are grouped by row in square brackets. The array `[1, 2]` initializes element `b[0]`, and the array `[3, 4]` initializes element `b[1]`. So `1` and `2` initialize `b[0][0]` and `b[0][1]`, respectively. Similarly, `3` and `4` initialize `b[1][0]` and `b[1][1]`, respectively. The interpreter determines the number of rows by counting the number of subinitializer lists —arrays nested within the outermost array. The interpreter determines the number of columns in each row by counting the number of values in the subarray that initializes the row.

Two-Dimensional Arrays with Rows of Different Lengths

The rows of a two-dimensional array can vary in length. The declaration

```
var b = [ [ 1, 2 ], [ 3, 4, 5 ] ];
```

creates array `b` with row `0` containing two elements (`1` and `2`) and row `1` containing three elements (`3`, `4` and `5`).

Creating Two-Dimensional Arrays with `new`

A multidimensional array in which each row has a *different* number of columns can be allocated dynamically, as follows:

```
var b;
b = new Array( 2 );      // allocate two rows
b[ 0 ] = new Array( 5 ); // allocate columns for row 0
b[ 1 ] = new Array( 3 ); // allocate columns for row 1
```

The preceding code creates a two-dimensional array with two rows. Row `0` has five columns, and row `1` has three columns.

Two-Dimensional Array Example: Displaying Element Values

The example in Figs. 10.20–10.21 initializes two-dimensional arrays in declarations and uses nested `for … in` loops to **traverse the arrays** (i.e., manipulate every element of the array). When the document in Fig. 10.20 loads, the script's `start` function displays the results of initializing the arrays.

The script's `start` function declares and initializes two arrays (Fig. 10.21, lines 5–9). The declaration of `array1` (lines 5–6) provides six initializers in two sublists. The first sublist initializes row 0 of the array to the values 1, 2 and 3; the second sublist initializes row 1 of the array to the values 4, 5 and 6. The declaration of `array2` (lines 7–9) provides six initializers in three sublists. The sublist for row 0 explicitly initializes the row to have two elements, with values 1 and 2, respectively. The sublist for row 1 initializes the row to have one element, with value 3. The sublist for row 2 initializes the third row to the values 4, 5 and 6.

```html
<!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.

```
// Fig. 10.21: InitArray3.js
// Initializing multidimensional arrays.
function start()
{
  var array1 = [ [ 1, 2, 3 ], // row 0
                 [ 4, 5, 6 ] ]; // row 1
  var array2 = [ [ 1, 2 ], // row 0
                 [ 3 ], // row 1
                 [ 4, 5, 6 ] ]; // row 2

   outputArray( "Values in array1 by row", array1,
      document.getElementById( "output1" ) );
   outputArray( "Values in array2 by row", array2,
      document.getElementById( "output2" ) );
} // end function start

// display array contents
function outputArray( heading, theArray, output )
{
```

```
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.

Function `start` calls function `outputArray` twice (lines 11–14) to display each array's elements in the web page. Function `outputArray` (lines 18–37) receives a string `heading` to output before the array, the array to output (called `theArray`) and the element in which to display the array. The function uses a nested `for` … `in` statement (lines 23–34) to output each row of a two-dimensional array as an ordered list. Using CSS, we set each list item's `display` property to `inline` so that the list items appear unnumbered from left to right on the page, rather than numbered and listed vertically (the default). The outer `for` … `in` statement iterates over the rows of the array. The inner `for` … `in` statement iterates over the columns of the current row being processed. The nested `for` … `in` statement in this example could have been written with `for` statements, as follows:

```
var numberOfRows = theArray.length;

for ( var row = 0; row < numberOfRows; ++row )
{
  results += "<ol>"; // start ordered list
  var numberOfcolumns = theArray[ row ].length;

  for ( var column = 0; j < numberOfcolumns; ++j )
  {
    results += "<li>" + theArray[ row ][ column ] + "</li>";
  } // end inner for

  results += "</ol>"; // end ordered list
} // end outer for
```

Just before the outer `for` statement, the expression `theArray.length` determines the number of rows in the array. Just before the inner `for` statement, the expression `theArray[row].length` determines the number of columns in the current row of the array. This enables the loop to determine, for each row, the exact number of columns.

Common Multidimensional-Array Manipulations with `for` and `for … in` Statements

Many common array manipulations use `for` or `for … in` repetition statements. For example, the following `for` statement sets all the elements in row 2 of array `a` in Fig. 10.19 to zero:

```
var columns = a[ 2 ].length;

for ( var col = 0; col < columns; ++col )
{
  a[ 2 ][ col ] = 0;
}
```

We specified row 2; therefore, we know that the first index is always `2`. The `for` loop varies only the second index (i.e., the column index). The preceding `for` statement is equivalent to the assignment statements

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

The following `for … in` statement is also equivalent to the preceding `for` statement:

```
for ( var col in a[ 2 ] )
{
  a[ 2 ][ col ] = 0;
}
```

The following nested `for` statement determines the total of all the elements in array `a`:

```
var total = 0;
var rows = a.length;

for ( var row = 0; row < rows; ++row )
{
  var columns = a[ row ].length;

  for ( var col = 0; col < columns; ++col )
  {
    total += a[ row ][ col ];
  }
}
```

The `for` statement totals the elements of the array, one row at a time. The outer `for` statement begins by setting the `row` index to `0`, so that the elements of row 0 may be totaled by the inner `for` statement. The outer `for` statement then increments `row to 1`, so that the elements of row 1 can be totaled. Then the outer `for` statement increments `row` to `2`, so that the elements of row 2 can be totaled. The result can be displayed when the nested `for` statement terminates. The preceding `for` statement is equivalent to the following `for … in` statement:

```
var total = 0;

for ( var row in a )
{
  for ( var col in a[ row ] )
  {
    total += a[ row ][ col ];
  }
}
```

Summary

Section 10.1 Introduction

• Arrays (p. 325) are data structures consisting of related data items (sometimes called collections of data items.

• JavaScript arrays are “dynamic” entities in that they can change size after they’re created.

Section 10.2 Arrays

• An array is a group of memory locations that 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 (p. 325). Any one of these elements may be referred to by giving the name of the array followed by the position number (an integer normally referred to as the index, p. 325) of the element in square brackets (`[]`).

• The first element in every array is the zeroth element (p. 325). In general, the *i*th element of array `c` is referred to as `c[i-1]`. Array names (p. 325) follow the same conventions as other identifiers.

- An indexed array name can be used on the left side of an assignment to place a new value (p. 326) into an array element. It can also be used on the right side of an assignment operation to assign its value to another variable.

- Every array in JavaScript knows its own length (p. 325), which it stores in its `length` attribute.

Section 10.3 Declaring and Allocating Arrays

- JavaScript arrays are represented by `Array` objects (p. 327).

- Arrays are created with operator `new` (p. 327).

Section 10.4 Examples Using Arrays

- To link a JavaScript file to an HTML document, use the `script` element's `src` attribute (p. 327) to specify the location of the JavaScript file.

- Zero-based counting is usually used to iterate through arrays.

- JavaScript automatically reallocates an array when a value is assigned to an element that's outside the bounds of the original array. Elements between the last element of the original array and the new element have undefined values.

- Arrays can be created using a comma-separated initializer list (p. 330) enclosed in square brackets (`[` and `]`). The array's size is determined by the number of values in the initializer list.

- The initial values of an array can also be specified as arguments in the parentheses following `new Array`. The size of the array is determined by the number of values in parentheses.

- JavaScript's `for` … `in` statement (p. 334) enables a script to perform a task for each element in an array. This process is known as iterating over the elements of an array.

Section 10.5 Random Image Generator Using Arrays

- We create a more elegant random image generator than the one in Chapter 9 that does not require the image filenames to be integers by using a `pictures` array to store the names of the image files as strings and accessing the array using a randomized index.

Section 10.6 References and Reference Parameters

- Two ways to pass arguments to functions (or methods) in many programming languages are pass-by-value and pass-by-reference (p. 339).

- When an argument is passed to a function by value, a *copy* of the argument's value is made and is passed to the called function.

- In JavaScript, numbers, boolean values and strings are passed to functions by value.

- With pass-by-reference, the caller gives the called function access to the caller's data and allows it to modify the data if it so chooses. Pass-by-reference 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.

- In JavaScript, all objects (and thus all arrays) are passed to functions by reference.

- The name of an array is actually a reference to an object that contains the array elements and the `length` variable, which indicates the number of elements in the array.

Section 10.7 Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array (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 (p. 340). To pass an array element to a function, use the indexed name of the element as an argument in the function call.

- Method `join` (p. 341) takes as its argument a string containing the separator (p. 341) that should be used to separate the elements of the array in the string that's returned. If the argument is not specified, the empty string is used as the separator.

Section 10.8 Sorting Arrays with `Array` Method `sort`

- Sorting data (putting data in a particular order, such as ascending or descending, p. 343) is one of the most important computing functions.

- The `Array` object in JavaScript has a built-in method `sort` (p. 343) for sorting arrays.

- By default, `Array` method `sort` (with no arguments) uses string comparisons to determine the sorting order of the array elements.

- Method `sort` takes as its optional argument the name of a function (called the comparator function, p. 343) that compares its two arguments and returns a negative value, zero, or a positive value, if the first argument is less than, equal to, or greater than the second, respectively.

- Functions in JavaScript are considered to be data. Therefore, functions can be assigned to variables, stored in arrays and passed to functions just like other data types.

Section 10.9 Searching Arrays with `Array` Method `indexOf`

- Array method `indexOf` (p. 344) searches for the first occurrence of a value and, if found, returns the value's array index; otherwise, it returns -1. Method `lastIndexOf` searches for the last occurrence.

Section 10.10 Multidimensional Arrays

- To identify a particular two-dimensional multidimensional array element, we must specify the two indices; by convention, the first identifies the element's row (p. 347) and the second the element's column (p. 347).

- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array (p. 347).

- Every element in a two-dimensional array (p. 347) is accessed using an element name of the form `a[row][column]` ; `a` is the name of the array, and `row` and `column` are the indices that uniquely identify the row and column, respectively, of each element in `a` .

- Multidimensional arrays are maintained as arrays of arrays.

Self-Review Exercises

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

a. Lists and tables of values can be stored in ________.

b. The number used to refer to a particular element of an array is called its ________.

c. The process of putting the elements of an array in order is called ________ the array.

d. Determining whether an array contains a certain key value is called ________ the array.

e. An array that uses two indices is referred to as a(n) ________ array.

10.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a. An array can store many different types of values.

b. An array index should normally be a floating-point value.

c. An individual array element that's passed to a function and modified in it will contain the modified value when the called function completes execution.

10.3 Write JavaScript statements (regarding array `fractions`) to accomplish each of the following tasks:

a. Declare an array with 10 elements, and initialize the elements of the array to `0`.

b. Refer to element 3 of the array.

c. Refer to array element 4.

d. Assign the value `1.667` to array element 9.

e. Assign the value `3.333` to the lowest-numbered element of the array.

f. Sum all the elements of the array, using a `for`… `in` statement. Define variable `x` as a control variable for the loop.

10.4 Write JavaScript statements (regarding array `table`) to accomplish each of the following tasks:

a. Declare and create the array with three rows and three columns.

b. Store the total number of elements in variable `totalElements`.

c. Use a `for`… `in` statement to initialize each element of the array to the sum of its row and column indices. Assume that the variables `x` and `y` are declared as control variables.

10.5 Find the error(s) in each of the following program segments, and correct them.

a.

```
var b = new Array( 10 );
var length = b.length;

for ( var i = 0; i <= length; ++i )
{
  b[ i ] = 1;
}
```

b.

```
var a = [ [ 1, 2 ], [ 3, 4 ] ];
a[ 1, 1 ] = 5;
```

Answers to Self-Review Exercises

10.1

a. arrays.

b. index.

c. sorting.

d. searching.

e. two-dimensional.

10.2

a. True.

b. False. An array index must be an integer or an integer expression.

c. False. Individual primitive-data-type elements are passed by value. If a reference to an array is passed, then modifications to the elements of the array are reflected in the original element of the array. Also, an individual element of an object type passed to a function is passed by reference, and changes to the object will be reflected in the original array element.

10.3

a. `var fractions = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];`

b. `fractions[ 3 ]`

c. `fractions[ 4 ]`

d. `fractions[ 9 ] = 1.667;`

e. `fractions[ 6 ] = 3.333;`

f.

```
var total = 0;
for ( var x in fractions )
{
  total += fractions[ x ];
}
```

10.4

a.

```
var table = new Array( new Array( 3 ), new Array( 3 ),
  new Array( 3 ) );
```

b. `var totalElements = table.length * table[ 0 ].length;`

c.

```
for ( var x in table )
{
  for ( var y in table[ x ] )
  {
    table[ x ][ y ] = x + y;
  }
}
```

10.5

a. Error: Referencing an array element outside the bounds of the array (`b[10]`). [*Note:* This is actually a logic error, not a syntax error.] Correction: Change the `<=` operator to `<` .

b. Error: The array indexing is done incorrectly. Correction: Change the statement to `a[1][1] = 5;` .

Exercises

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

a. JavaScript stores lists of values in ________.

b. The names of the four elements of array `p` are ________, ________, ________ and ________.

c. In a two-dimensional array, by convention the first index identifies the ________ of an element, and the second index identifies the ________ of an element.

d. An *m*-by-*n* array contains ________ rows, ________ columns and ________ elements.

e. The name the element in row 3 and column 5 of array `d` is ________.

10.7 State whether each of the following is *true* or *false*. If *false*, explain why.

a. To refer to a particular location or element in an array, we specify the name of the array and the value of the element.

b. A variable declaration reserves space for an array.

c. To indicate that 100 locations should be reserved for integer array `p`, the programmer should write the declaration

p[**100**];

d. A JavaScript program that initializes the elements of a 15-element array to zero must contain at least one `for` statement.

e. A JavaScript program that totals the elements of a two-dimensional array must contain nested `for` statements.

10.8 Write JavaScript statements to accomplish each of the following tasks:

a. Display the value of the seventh element of array `f`.

b. Initialize each of the five elements of one-dimensional array `g` to `8`.

c. Total the elements of array `c`, which contains 100 numeric elements.

d. Copy 11-element array `a` into the first portion of array `b`, which contains 34 elements.

e. Determine and print the smallest and largest values contained in 99-element floating-point array `w`.

10.9 Consider a two-by-three array `t` that will store integers.

a. Write a statement that declares and creates array `t`.

b. How many rows does `t` have?

c. How many columns does `t` have?

d. How many elements does `t` have?

e. Write the names of all the elements in row 1 of `t`.

f. Write the names of all the elements in the third column of `t`.

g. Write a single statement that sets the element of `t` in row 1 and column 2 to zero.

h. Write a series of statements that initializes each element of `t` to zero. Do not use a repetition statement.

i. Write a nested `for` statement that initializes each element of `t` to zero.

j. Write a series of statements that determines and prints the smallest value in array `t`.

k. Write a nonrepetition statement that displays the elements of the first row of `t`.

l. Write a series of statements that prints the array `t` in neat, tabular format. List the column indices as headings across the top, and list the row indices at the left of each row.

10.10 Use a one-dimensional array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who grosses $5000 in sales in a week receives $200 plus 9 percent of $5000, or a total of $650. Write a script (using an array of counters) that obtains the gross sales for each employee through an HTML5 form and determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

a. $200–299

b. $300–399

c. $400–499

d. $500–599

e. $600–699

f. $700–799

g. $800–899

h. $900–999

i. $1000 and over

10.11 Write statements that perform the following operations for a one-dimensional array:

a. Set the 10 elements of array `counts` to zeros.

b. Add 1 to each of the 15 elements of array `bonus` .

c. Display the five values of array `bestScores`, separated by spaces.

10.12 Use a one-dimensional array to solve the following problem: Read in 10 numbers, each of which is between 10 and 100. As each number is read, print it only if it's not a duplicate of a number that has already been read. Provide for the "worst case," in which all 10 numbers are different. Use the smallest possible array to solve this problem.

10.13 Label the elements of three-by-five two-dimensional array `sales` to indicate the order in which they're set to zero by the following program segment:

```
for ( var row in sales )
{
  for ( var col in sales[ row ] )
  {
    sales[ row ][ col ] = 0;
  }
}
```

10.14 Write a script to simulate the rolling of two dice. The script should use `Math.random` to roll the first die and again to roll the second die. The sum of the two values should then be calculated. [*Note:* Since each die can show an integer value from 1 to 6, the sum of the values will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent sums. Figure 10.22 shows the 36 possible combinations of the two dice. Your program should roll the dice 36,000 times. Use a one-dimensional array to tally the number of times each possible sum appears. Display the results in an HTML5 table. Also determine whether the totals are reasonable (e.g., there are six ways to roll a 7, so approximately 1/6 of all the rolls should be 7).]

Fig. 10.22. The 36 possible combinations of the two dice.

10.15 ***(Turtle Graphics)*** The Logo language, which is popular among young computer users, made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a JavaScript program. The turtle holds a pen in one of two positions, up or down. When the pen is down, the turtle traces out shapes as it moves; when the pen is up, the turtle moves about freely without writing anything. In this problem, you'll simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 20-by-20 array `floor` that's initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and of whether the pen is currently up or down. Assume that the turtle always starts at position (0, 0) of the floor, with its pen up. The set of turtle commands your script must process are as in Fig. 10.23.

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square, then leave the pen in the up position:

Fig. 10.23. Turtle-graphics commands.

2
5,12
3
5,12
3
5,12
3
5,12
1
6
9

As the turtle moves with the pen down, set the appropriate elements of array `floor` to `1`s. When the `6` command (print) is given, display an asterisk or some other character of your choosing wherever there's a `1` in the array. Wherever there's a zero, display a blank. Write a script to implement the turtle-graphics capabilities discussed here. Write several turtle-graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle-graphics language.

10.16 ***(The Sieve of Eratosthenes)*** A prime integer is an integer greater than 1 that's evenly divisible only by itself and 1. The Sieve of Eratosthenes is an algorithm for finding prime numbers. It operates as follows:

a. Create an array with all elements initialized to 1 (true). Array elements with prime indices will remain as 1. All other array elements will eventually be set to zero.

b. Set the first two elements to zero, since 0 and 1 are not prime. Starting with array index 2, every time an array element is found whose value is 1, loop through the remainder of the array and set to zero every element whose index is a multiple of the index for the element with value 1. For array index 2, all elements beyond 2 in the array that are multiples of 2 will be set to zero (indices 4, 6, 8, 10, etc.); for array index 3, all elements beyond 3 in the array that are multiples of 3 will be set to zero (indices 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still set to 1 indicate that the index is a prime number. These indices can then be printed. Write a script that uses an array of 1000 elements to determine and print the prime numbers between 1 and 999. Ignore element 0 of the array.

10.17 ***(Simulation: The Tortoise and the Hare)*** In this problem, you'll re-create one of the truly great moments in history, namely the classic race of the tortoise and the hare. You'll use random number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

There's a clock that ticks once per second. With each tick of the clock, your script should adjust the position of the animals according to the rules in Fig. 10.24.

Fig. 10.24. Rules for adjusting the position of the tortoise and the hare.

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the “starting gate”). If an animal slips left before square 1, move the animal back to square 1.

Generate the percentages in Fig. 10.24 by producing a random integer *i* in the range $1 \leq i \leq 10$. For the tortoise, perform a “fast plod” when $1 \leq i \leq 5$, a “slip” when $6 \leq i \leq 7$ and a “slow plod” when $8 \leq i \leq 10$. Use a similar technique to move the hare.

Begin the race by printing

BANG !!!!!
AND THEY'RE OFF !!!!!

Then, for each tick of the clock (i.e., each repetition of a loop), print a 70-position line showing the letter `T` in the position of the tortoise and the letter `H` in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your script should print `OUCH!!!` beginning at that position. All print positions other than the `T`, the `H` or the `OUCH!!!` (in case of a tie) should be blank.

After each line is printed, test whether either animal has reached or passed square 70. If so, print the winner, and terminate the simulation. If the tortoise wins, print `TORTOISE WINS!!! YAY!!!` If the hare wins, print `Hare wins. Yuck!` If both animals win on the same tick of the clock, you may want to favor the turtle (the “underdog”), or you may want to print `It's a tie`. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you’re ready to run your script, assemble a group of fans to watch the race. You’ll be amazed at how involved your audience gets!

Later in the book, we introduce a number of Dynamic HTML capabilities, such as graphics, images, animation and sound. As you study those features, you might enjoy enhancing your tortoise-and-hare contest simulation.