

- 8.3**
- a) Error: The semicolon after the `while` header causes an infinite loop, and there is a missing left brace.
Correction: Replace the semicolon by a `{`, or remove both the `;` and the `}`.
 - b) Error: Using a floating-point number to control a `for` repetition statement may not work, because floating-point numbers are represented approximately by most computers.
Correction: Use an integer, and perform the proper calculation to get the values you desire:
- ```
for (y = 1; y != 10; y++)
 document.writeln((y / 10) + " ");
```
- c) Error: Missing `break` statement in the statements for the first `case`.  
Correction: Add a `break` statement at the end of the statements for the first `case`. Note that this missing statement is not necessarily an error if the programmer wants the statement of `case 2`: to execute every time the `case 1:` statement executes.
  - d) Error: Improper relational operator used in the `while` continuation condition.  
Correction: Use `<=` rather than `<`, or change 10 to 11.

## Exercises

- 8.4** Find the error in each of the following segments of code. [Note: There may be more than one error.]

- a) `For ( x = 100, x >= 1, x++ )`  
 `document.writeln( x );`
- b) The following code should print whether integer value is odd or even:  

```
switch (value % 2) {
 case 0:
 document.writeln("Even integer");
 case 1:
 document.writeln("Odd integer");
}
```
- c) The following code should output the odd integers from 19 to 1:  

```
for (x = 19; x >= 1; x += 2)
 document.writeln(x);
```
- d) The following code should output the even integers from 2 to 100:  

```
counter = 2;
do {
 document.writeln(counter);
 counter += 2;
} while (counter < 100);
```

- 8.5** What does the following script do?

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Exercise 8.5: ex08_05.html -->
6 <html xmlns = "http://www.w3.org/1999/xhtml">
7 <head><title>Mystery</title>
8 <script type = "text/javascript">
9 <!--
10 document.writeln("<table>");
11

```

```

12 for (var i = 1; i <= 10; i++)
13 {
14 document.writeln("<tr>");
15
16 for (var j = 1; j <= 5; j++)
17 document.writeln("<td>" + i + ", " + j + "</td>");
18
19 document.writeln("</tr>");
20 } // end for
21
22 document.writeln("</table>");
23 -->
24 </script>
25 </head><body />
26 </html>
```

**8.6** Write a script that finds the smallest of several non-negative integers. Assume that the first value read specifies the number of values to be input from the user.

**8.7** Write a script that calculates the product of the odd integers from 1 to 15 then outputs XHTML text that displays the results.

**8.8** Modify the compound interest program in Fig. 8.6 to repeat its steps for interest rates of 5, 6, 7, 8, 9 and 10 percent. Use a `for` statement to vary the interest rate. Use a separate table for each rate.

**8.9** Write a script that outputs XHTML to display the given patterns separately, one below the other. Use `for` statements to generate the patterns. All asterisks (\*) should be printed by a single statement of the form `document.write("*)");` (this causes the asterisks to print side by side). A statement of the form `document.writeln("<br />");` can be used to position to the next line. A statement of the form `document.write(" ");` can be used to display a space (needed for the last two patterns). There should be no other output statements in the program. [Hint: The last two patterns require that each line begin with an appropriate number of blanks. You may need to use the XHTML `<pre></pre>` tags.]

| (a)   | (b)   | (c)   | (d)   |
|-------|-------|-------|-------|
| *     | ***** | ***** | *     |
| **    | ***** | ***** | **    |
| ***   | ***** | ***** | ***   |
| ****  | ****  | ****  | ****  |
| ***** | ***   | ***   | ***** |
| ***** | **    | **    | ***** |
| ***** | *     | *     | ***** |

**8.10** One interesting application of computers is the drawing of graphs and bar charts (sometimes called histograms). Write a script that reads five numbers between 1 and 30. For each number read, output XHTML text that displays a line containing the same number of adjacent asterisks. For example, if your program reads the number 7, it should output XHTML text that displays \*\*\*\*\*.

**8.11** (*"The Twelve Days of Christmas" Song*) Write a script that uses repetition and a `switch` structures to print the song "The Twelve Days of Christmas." You can find the words at the site

**8.12** A mail-order house sells five different products whose retail prices are as follows: product 1, \$2.98; product 2, \$4.50; product 3, \$9.98; product 4, \$4.49; and product 5, \$6.87. Write a script that reads a series of pairs of numbers as follows:

- Product number
- Quantity sold for one day

Your program should use a switch statement to determine each product's retail price and should calculate and output XHTML that displays the total retail value of all the products sold last week. Use a prompt dialog to obtain the product number and quantity from the user. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

**8.13** Assume that  $i = 1$ ,  $j = 2$ ,  $k = 3$  and  $m = 2$ . What does each of the given statements print? Are the parentheses necessary in each case?

- `document.writeln( i == 1 );`
- `document.writeln( j == 3 );`
- `document.writeln( i >= 1 && j < 4 );`
- `document.writeln( m <= 99 && k < m );`
- `document.writeln( j >= i || k == m );`
- `document.writeln( k + m < j || 3 - j >= k );`
- `document.writeln( !( k > m ) );`

**8.14** Modify Exercise 8.9 to combine your code from the four separate triangles of asterisks into a single script that prints all four patterns side by side, making clever use of nested for statements.

```

* ***** **** *
** ***** **** **
*** ***** **** ***
**** ***** **** ****
***** ***** **** ****
***** ***** **** ****
***** ***** *** ****
***** *** *** ****
***** ** ** ****
***** * * ****

```

**8.15** (*De Morgan's Laws*) In this chapter, we have discussed the logical operators `&&`, `||` and `!`. De Morgan's Laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression `!(condition1 && condition2)` is logically equivalent to the expression `(!condition1 || !condition2)`. Also, the expression `!(condition1 || condition2)` is logically equivalent to the expression `(!condition1 && !condition2)`. Use De Morgan's Laws to write equivalent expressions for each of the following, then write a program to show that the original expression and the new expression are equivalent in each case:

- `!( x < 5 ) && !( y >= 7 )`
- `!( a == b ) || !( g != 5 )`
- `!( ( x <= 8 ) && ( y > 4 ) )`
- `!( ( i > 4 ) || ( j <= 6 ) )`

**8.16** Write a script that prints the following diamond shape:

```

*

 *

```

You may use output statements that print a single asterisk (\*), a single space or a single newline character. Maximize your use of repetition (with nested `for` statements), and minimize the number of output statements.

**8.17** Modify the program you wrote in Exercise 8.16 to read an odd number in the range 1 to 19. This number specifies the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

**8.18** A criticism of the `break` statement and the `continue` statement is that each is unstructured. Actually, `break` statements and `continue` statements can always be replaced by structured statements, although coding the replacement can be awkward. Describe in general how you would remove any `break` statement from a loop in a program and replace it with some structured equivalent. [Hint: The `break` statement “jumps out of” a loop from the body of that loop. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates “early exit because of a ‘break’ condition.”] Use the technique you develop here to remove the `break` statement from the program in Fig. 8.11.

**8.19** What does the following script do?

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Exercise 8.19: ex08_19.html -->
6 <html xmlns = "http://www.w3.org/1999/xhtml">
7 <head><title>Mystery</title>
8 <script type = "text/javascript">
9 <!--
10 for (var i = 1; i <= 5; i++)
11 {
12 for (var j = 1; j <= 3; j++)
13 {
14 for (var k = 1; k <= 4; k++)
15 document.write("*");
16 document.writeln("
");
17 } // end for
18 document.writeln("
");
19 } // end for
20 // --
21 </script>
22 </head><body></body>
23 </html>
```

**8.20** Describe in general how you would remove any `continue` statement from a loop in a program and replace it with some structured equivalent. Use the technique you develop to remove the `continue` statement from the program in Fig. 8.12.

**8.21** Given the following `switch` statement:

```

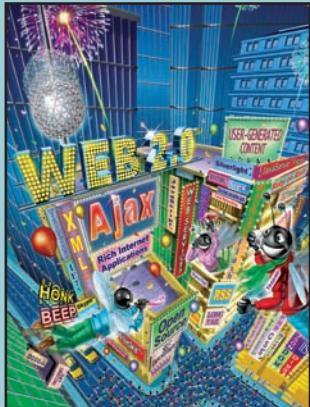
1 switch (k)
2 {
3 case 1:
4 break;
5 case 2:
6 case 3:
7 ++k;
8 break;
```

```
1 case 4:
2 --k;
3 break;
4 default:
5 k *= 3;
6 } //end switch
7
8 x = k;
```

What values are assigned to x when k has values of 1, 2, 3, 4 and 10?

# 9

# JavaScript: Functions



*Form ever follows function.*

—Louis Sullivan

*E pluribus unum.*

(One composed of many.)

—Virgil

*O! call back yesterday, bid  
time return.*

—William Shakespeare

*Call me Ishmael.*

—Herman Melville

*When you call me that,  
smile.*

—Owen Wister

## OBJECTIVES

In this chapter you will learn:

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

## Outline

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

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 9.1 Introduction

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

## 9.2 Program Modules in JavaScript

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

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



### Good Programming Practice 9.1

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



### Software Engineering Observation 9.1

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



### Portability Tip 9.1

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

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

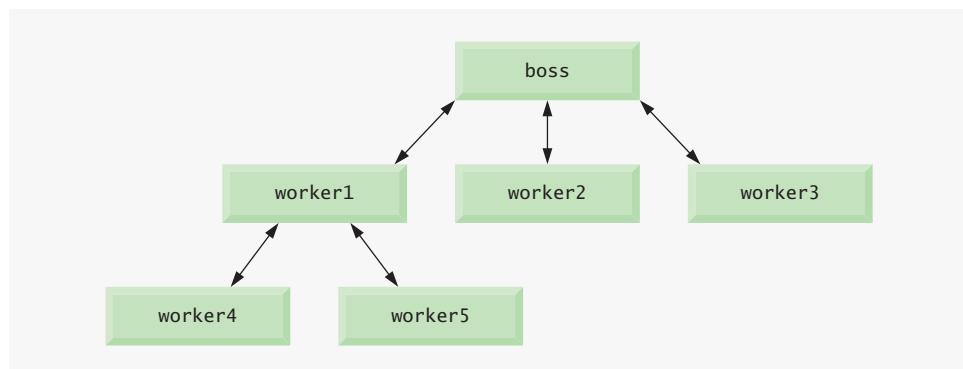
A function is **invoked** (i.e., made to perform its designated task) by a **function call**. The function call specifies the function name and provides information (as **arguments**) that the called function needs to perform its task. A common analogy for this structure is the hierarchical form of management.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

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

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

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

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



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

## 9.3 Programmer-Defined Functions

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

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



### Software Engineering Observation 9.2

---

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

## 9.4 Function Definitions

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

### Programmer-Defined Function `square`

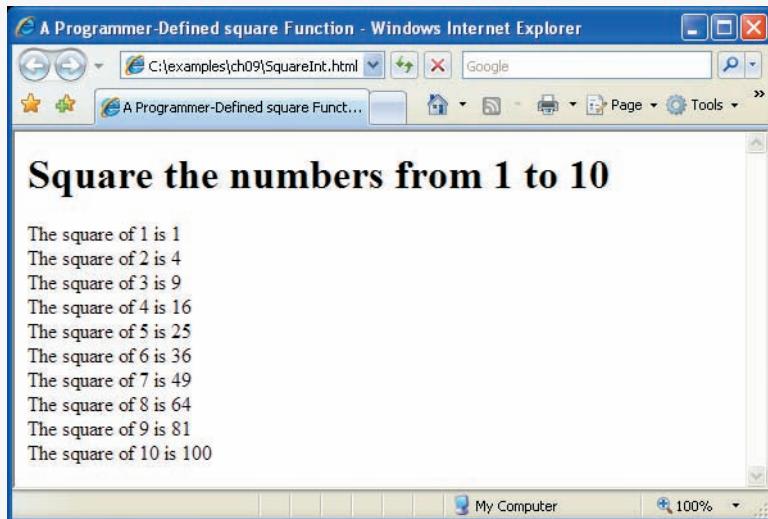
Consider a script (Fig. 9.2) that uses a function `square` to calculate the squares of the integers from 1 to 10. [Note: We continue to show many examples in which the body element of the XHTML document is empty and the document is created directly by JavaScript. In later chapters, we show many examples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

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

```

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

```



**Fig. 9.2** | Programmer-defined function square.

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

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



### Common Programming Error 9.1

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

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



### Good Programming Practice 9.2

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



### Software Engineering Observation 9.3

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

The format of a function definition is

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

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



### Common Programming Error 9.2

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



### Common Programming Error 9.3

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



### Common Programming Error 9.4

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



### Common Programming Error 9.5

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



### Good Programming Practice 9.3

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



### Software Engineering Observation 9.4

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



### Error-Prevention Tip 9.1

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

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

```
return;
```

If the function does return a result, the statement

```
return expression;
```

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

#### *Programmer-Defined Function maximum*

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

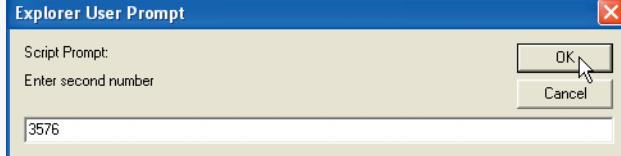
The three floating-point values are input by the user via `prompt` dialogs (lines 12–14). Lines 16–18 use function `parseFloat` to convert the strings entered by the user to floating-point values. The statement in line 20 passes the three floating-point values to function `maximum` (defined in lines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

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

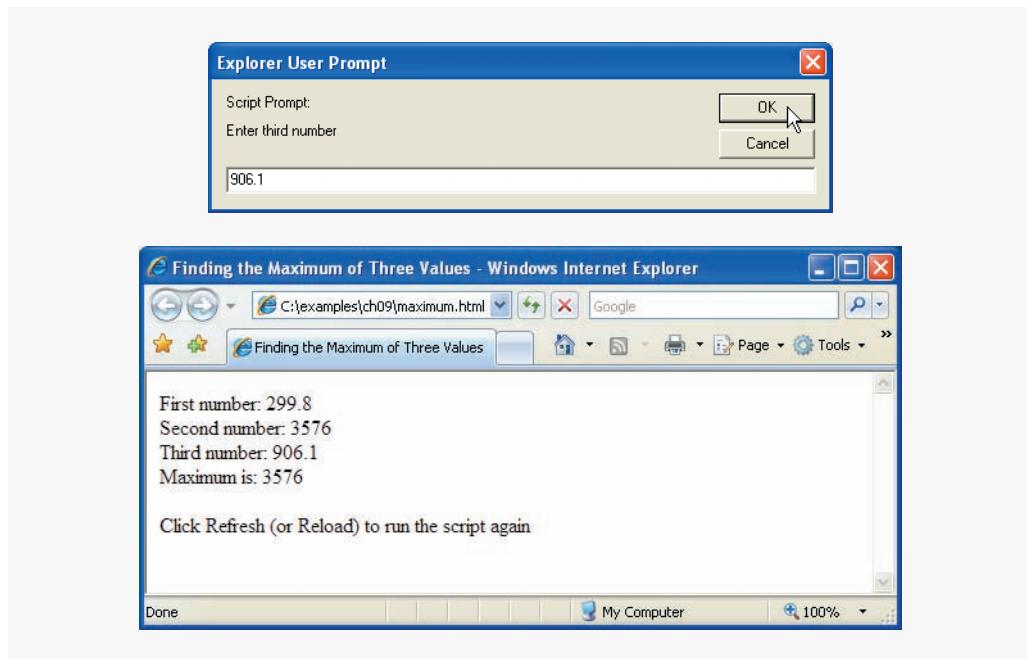
```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 9.3: maximum.html -->
6 <!-- Programmer-Defined maximum function. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Finding the Maximum of Three Values</title>
10 <script type = "text/javascript">
11 <!--
12 var input1 = window.prompt("Enter first number", "0");
13 var input2 = window.prompt("Enter second number", "0");
14 var input3 = window.prompt("Enter third number", "0");
15
16 var value1 = parseFloat(input1);
17 var value2 = parseFloat(input2);
18 var value3 = parseFloat(input3);
19
20 var maxValue = maximum(value1, value2, value3);
21
22 document.writeln("First number: " + value1 +
23 "
Second number: " + value2 +
24 "
Third number: " + value3 +
25 "
Maximum is: " + maxValue);
26
27 // maximum function definition (called from line 20)
28 function maximum(x, y, z)
29 {
30 return Math.max(x, Math.max(y, z));
31 } // end function maximum
32 // --
33 </script>
34 </head>
35 <body>
36 <p>Click Refresh (or Reload) to run the script again</p>
37 </body>
38 </html>

```



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



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

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

## 9.5 Random Number Generation

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

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

Consider the following statement:

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

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

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

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

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

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

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

```

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

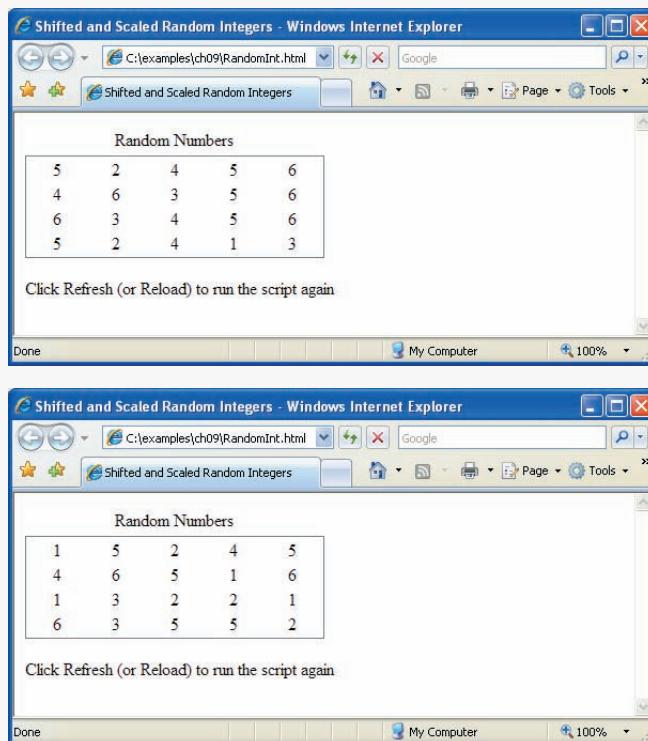
```

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

```

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

```



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

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

```

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

```

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

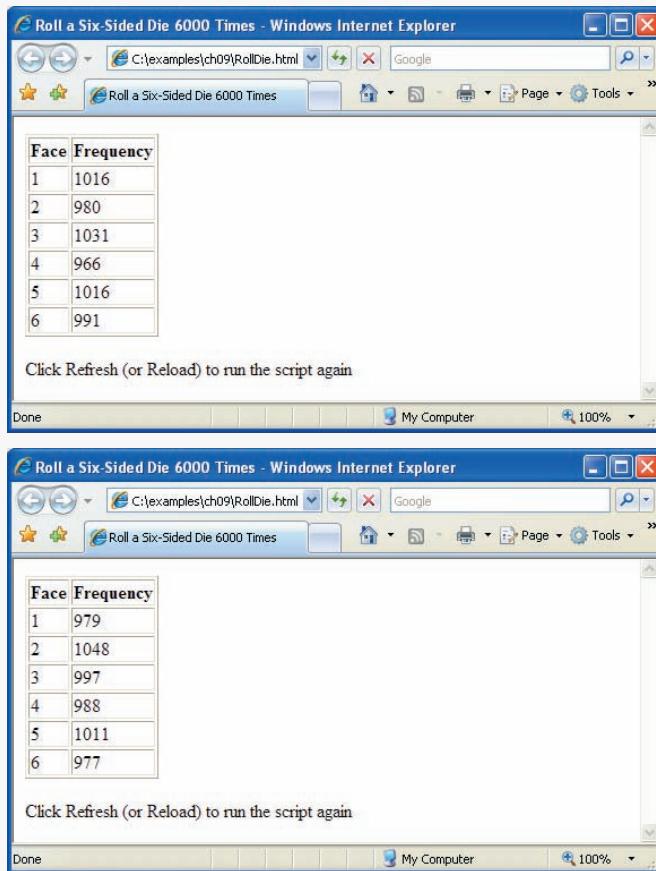
```
8 <head>
9 <title>Roll a Six-Sided Die 6000 Times</title>
10 <script type = "text/javascript">
11 <!--
12 var frequency1 = 0;
13 var frequency2 = 0;
14 var frequency3 = 0;
15 var frequency4 = 0;
16 var frequency5 = 0;
17 var frequency6 = 0;
18 var face;
19
20 // roll die 6000 times and accumulate results
21 for (var roll = 1; roll <= 6000; roll++)
22 {
23 face = Math.floor(1 + Math.random() * 6);
24
25 switch (face)
26 {
27 case 1:
28 ++frequency1;
29 break;
30 case 2:
31 ++frequency2;
32 break;
33 case 3:
34 ++frequency3;
35 break;
36 case 4:
37 ++frequency4;
38 break;
39 case 5:
40 ++frequency5;
41 break;
42 case 6:
43 ++frequency6;
44 break;
45 } // end switch
46 } // end for
47
48 document.writeln("<table border = \"1\">");
49 document.writeln("<thead><th>Face</th>" +
50 "<th>Frequency</th></thead>");
51 document.writeln("<tbody><tr><td>1</td><td>" +
52 frequency1 + "</td></tr>");
53 document.writeln("<tr><td>2</td><td>" + frequency2 +
54 "</td></tr>");
55 document.writeln("<tr><td>3</td><td>" + frequency3 +
56 "</td></tr>");
57 document.writeln("<tr><td>4</td><td>" + frequency4 +
58 "</td></tr>");
59 document.writeln("<tr><td>5</td><td>" + frequency5 +
60 "</td></tr>");
```

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

```

61 document.writeln("<tr><td>6</td><td>" + frequency6 +
62 "</td></tr></tbody></table>");
63 // -->
64 </script>
65 </head>
66 <body>
67 <p>Click Refresh (or Reload) to run the script again</p>
68 </body>
69 </html>

```



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

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

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

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

The values returned by `random` are always in the range

```
0.0 ≤ Math.random() < 1.0
```

Previously, we demonstrated the statement

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

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

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

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

## 9.6 Example: Game of Chance

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

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

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

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

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

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

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

```
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 9.6: Craps.html -->
6 <!-- Craps game simulation. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Program that Simulates the Game of Craps</title>
10 <style type = "text/css">
11 table { text-align: right }
12 body { font-family: arial, sans-serif }
13 div.red { color: red }
14 </style>
15 <script type = "text/javascript">
16 <!--
17 // variables used to test the state of the game
18 var WON = 0;
19 var LOST = 1;
20 var CONTINUE_ROLLING = 2;
21
22 // other variables used in program
23 var firstRoll = true; // true if current roll is first
24 var sumOfDice = 0; // sum of the dice
25 var myPoint = 0; // point if no win/loss on first roll
26 var gameStatus = CONTINUE_ROLLING; // game not over yet
27
28 // process one roll of the dice
29 function play()
30 {
31 // get the point field on the page
32 var point = document.getElementById("pointfield");
33
34 // get the status div on the page
35 var statusDiv = document.getElementById("status");
36 if (firstRoll) // first roll of the dice
37 {
```

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

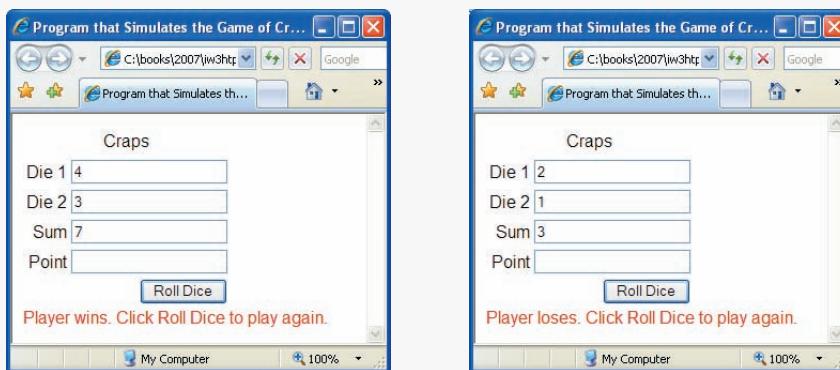
```
38 sumOfDice = rollDice();
39
40 switch (sumOfDice)
41 {
42 case 7: case 11: // win on first roll
43 gameStatus = WON;
44 // clear point field
45 point.value = "";
46 break;
47 case 2: case 3: case 12: // lose on first roll
48 gameStatus = LOST;
49 // clear point field
50 point.value = "";
51 break;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

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

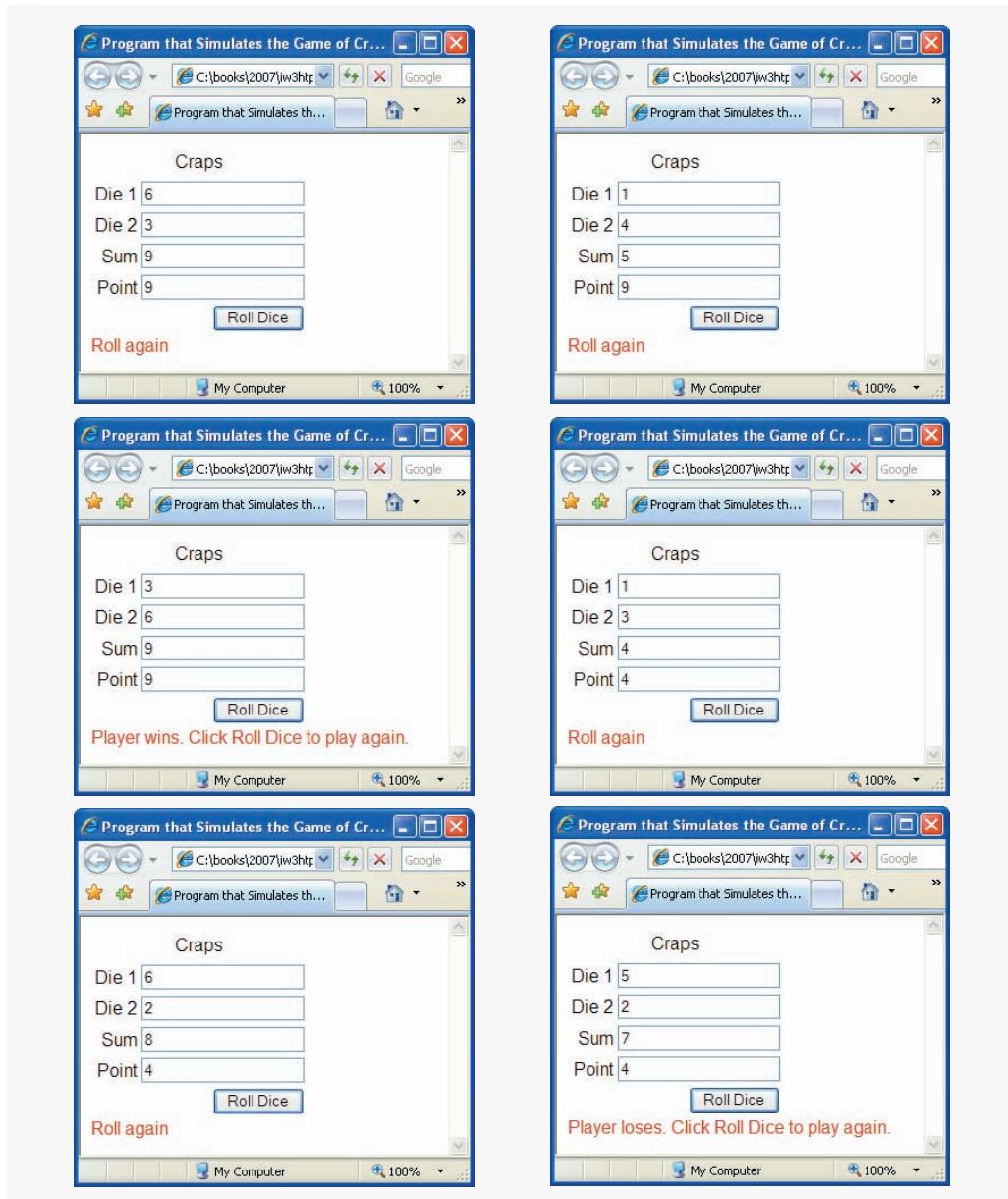
```

91
92 die1 = Math.floor(1 + Math.random() * 6);
93 die2 = Math.floor(1 + Math.random() * 6);
94 workSum = die1 + die2;
95
96 document.getElementById("die1field").value = die1;
97 document.getElementById("die2field").value = die2;
98 document.getElementById("sumfield").value = workSum;
99
100 return workSum;
101 } // end function rollDice
102 // -->
103 </script>
104 </head>
105 <body>
106 <form action = "">
107 <table>
108 <caption>Craps</caption>
109 <tr><td>Die 1</td>
110 <td><input id = "die1field" type = "text" />
111 </td></tr>
112 <tr><td>Die 2</td>
113 <td><input id = "die2field" type = "text" />
114 </td></tr>
115 <tr><td>Sum</td>
116 <td><input id = "sumfield" type = "text" />
117 </td></tr>
118 <tr><td>Point</td>
119 <td><input id = "pointfield" type = "text" />
120 </td></tr>
121 <tr><td><td><input type = "button" value = "Roll Dice"
122 onclick = "play()" /></td></tr>
123 </table>
124 <div id = "status" class = "red">
125 Click the Roll Dice button to play</div>
126 </form>
127 </body>
128 </html>

```



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



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

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

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

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

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

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

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

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

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



#### Good Programming Practice 9.4

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



#### Good Programming Practice 9.5

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

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

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

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

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

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

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

Roll again.

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

Player wins. Click Roll Dice to play again.

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

Player loses. Click Roll Dice to play again.

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

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

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



### Software Engineering Observation 9.5

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

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



### Error-Prevention Tip 9.2

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

## 9.7 Another Example: Random Image Generator

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

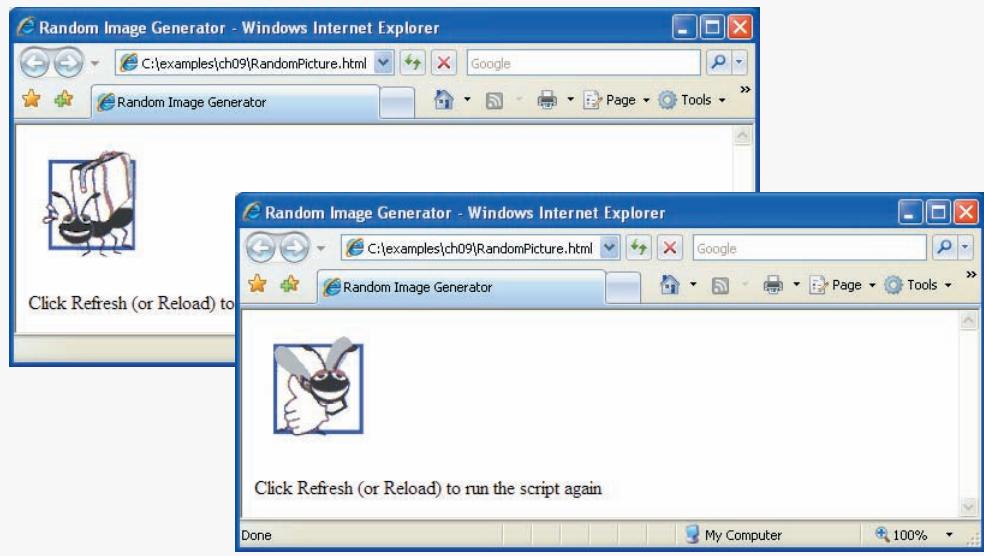
For the script in Fig. 9.7 to function properly, the directory containing the file `RandomPicture.html` must also contain seven images with integer filenames (i.e., `1.gif`, `2.gif`, ..., `7.gif`). The web page containing this script displays one of these seven images, selected at random, each time the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

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

```

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

```



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

## 9.8 Scope Rules

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

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

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



### Good Programming Practice 9.6

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

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

```

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

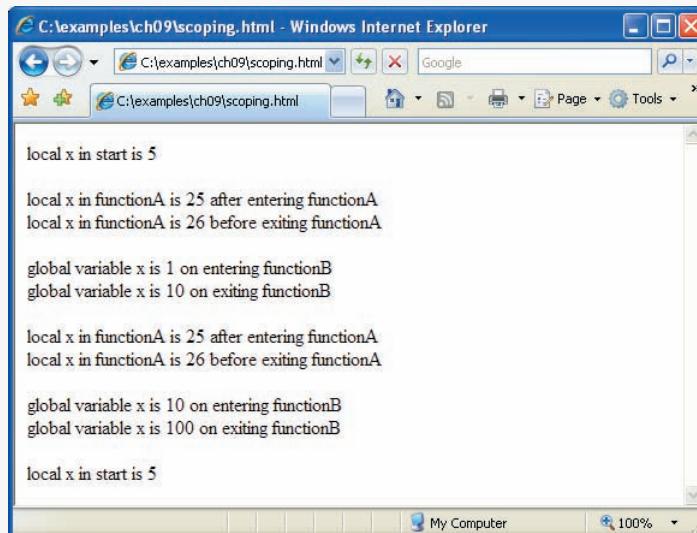
```

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

```

36 ++x;
37 document.writeln("
local x in functionA is " +
38 " x + " before exiting functionA" + "</p>");
39 } // end functionA
40
41 function functionB()
42 {
43 document.writeln("<p>global variable x is " + x +
44 " on entering functionB");
45 x *= 10;
46 document.writeln("
global variable x is " +
47 " x + " on exiting functionB" + "</p>");
48 } // end functionB
49 // -->
50 </script>
51 </head>
52 <body onload = "start()"></body>
53 </html>

```



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

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

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

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

## 9.9 JavaScript Global Functions

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

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

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

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

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

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

## 9.10 Recursion

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

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

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

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

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

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

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

```
var factorial = 1;

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

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

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

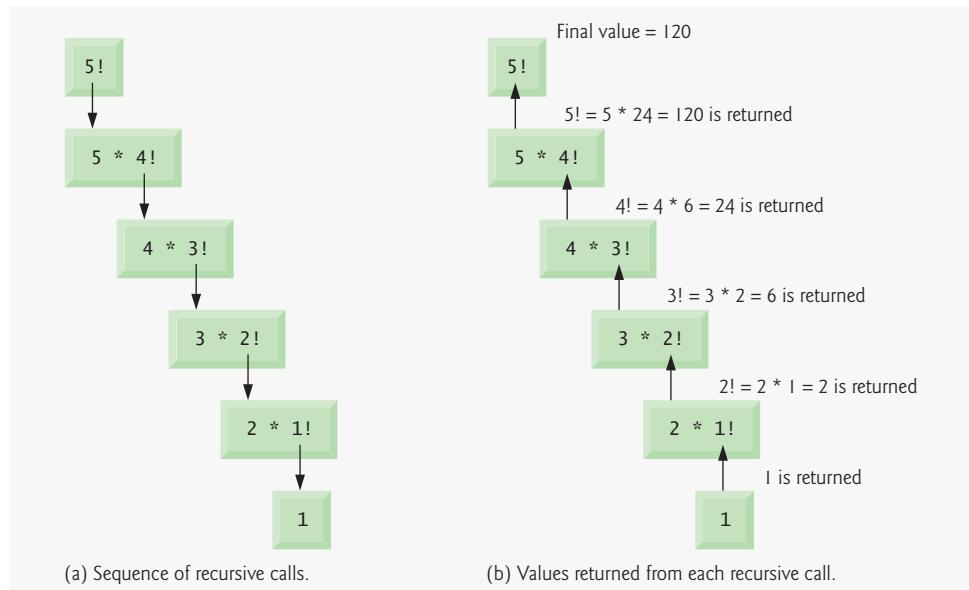
For example,  $5!$  is clearly equal to  $5 * 4!$ , as is shown by the following equations:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

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

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

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

**Fig. 9.10** | Recursive evaluation of  $5!$ .

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 9.11: FactorialTest.html -->
6 <!-- Factorial calculation with a recursive function. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Recursive Factorial Function</title>
10 <script type = "text/javascript">
11 <!--
12 document.writeln("<h1>Factorials of 1 to 10</h1>");
13 document.writeln("<table>");
14
15 for (var i = 0; i <= 10; i++)
16 document.writeln("<tr><td>" + i + "!</td><td>" +
17 factorial(i) + "</td></tr>");
18
19 document.writeln("</table>");
20
21 // Recursive definition of function factorial
22 function factorial(number)
23 {
24 if (number <= 1) // base case
25 return 1;
26 else
27 return number * factorial(number - 1);
28 } // end function factorial

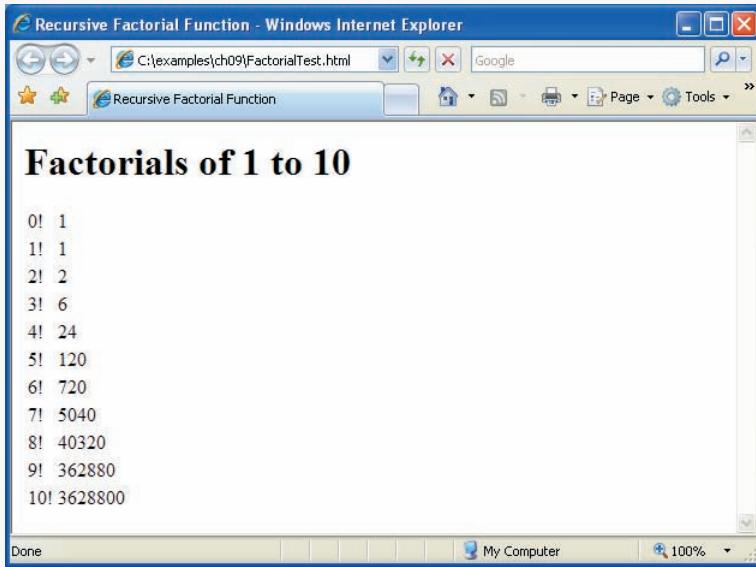
```

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

```

29 // -->
30 </script>
31 </head><body></body>
32 </html>

```



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



### Common Programming Error 9.6

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



### Common Programming Error 9.7

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



### Error-Prevention Tip 9.3

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

## 9.11 Recursion vs. Iteration

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

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

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

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



### Software Engineering Observation 9.6

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



### Performance Tip 9.1

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



### Common Programming Error 9.8

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

In addition to the Factorial function example (Fig. 9.11), we also provide several recursion exercises—raising an integer to an integer power (Exercise 9.34), visualizing recursion (Exercise 9.35) and sum of two integers (Exercise 9.36). Also, Fig. 14.26 uses recursion to traverse an XML document tree.

## 9.12 Wrap-Up

This chapter introduced JavaScript functions, which allow you to modularize your programs. We showed how to call functions and methods and how to define your own functions that accomplish tasks. We also showed how parameters are used to pass data into a function, while return values are used to pass a result back to the caller. We discussed how to get a range of random numbers and built a craps game and a random image generator using these concepts. Finally, we introduced recursion, which provides an alternative method for solving problems that involve repetitive calculations. In the next chapter, we introduce arrays, which allow you to store lists of data in a single variable.

## 9.13 Web Resources

[www.deitel.com/javascript/](http://www.deitel.com/javascript/)

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

## Summary

### Section 9.1 Introduction

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

### Section 9.2 Program Modules in JavaScript

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

### Section 9.3 Programmer-Defined Functions

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

### Section 9.4 Function Definitions

- The return statement passes information from inside a function back to the point in the program where it was called.

- A function must be called explicitly for the code in its body to execute.
- The format of a function definition is

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

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

### Section 9.5 Random Number Generation

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

### Section 9.6 Example: Game of Chance

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

### Section 9.7 Another Example: Random Image Generator

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

### Section 9.8 Scope Rules

- Each identifier in a program has a scope. The scope of an identifier for a variable or function is the portion of the program in which the identifier can be referenced.
- Global variables or script-level variables (i.e., variables declared in the head element of the XHTML document) are accessible in any part of a script and are said to have global scope. Thus every function in the script can potentially use the variables.
- Identifiers declared inside a function have function (or local) scope and can be used only in that function. Function scope begins with the opening left brace (`{`) of the function in which the identifier is declared.

tifier is declared and ends at the terminating right brace (}) of the function. Local variables of a function and function parameters have function scope.

- If a local variable in a function has the same name as a global variable, the global variable is “hidden” from the body of the function.
- The `onload` property of the `body` element calls an event handler when the body of the XHTML document is completely loaded into the browser window.

### ***Section 9.9 JavaScript Global Functions***

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

### ***Section 9.10 Recursion***

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

### ***Section 9.11 Recursion vs. Iteration***

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

## **Terminology**

argument in a function call

called function

base case

caller

binary format

calling function

block

computer-assisted instruction (CAI)

|                                              |                                       |
|----------------------------------------------|---------------------------------------|
| constant                                     | local scope                           |
| converge on the base case                    | local variable                        |
| copy of a value                              | max method of the Math object         |
| divide and conquer                           | method                                |
| dot (.)                                      | modularize a program                  |
| dot notation                                 | module                                |
| element of chance                            | object                                |
| escape function                              | octal                                 |
| eval function                                | onclick event                         |
| element of chance                            | onload event                          |
| event                                        | parameter in a function definition    |
| event handler                                | parseFloat function                   |
| event-handling function                      | parseInt function                     |
| event-driven programming                     | programmer-defined function           |
| floor method of the Math object              | probability                           |
| function                                     | programmer-defined function           |
| function (local) scope                       | radix                                 |
| function argument                            | random method of the Math object      |
| function body                                | random-number generation              |
| function call                                | recursion                             |
| function definition                          | recursive function                    |
| function name                                | recursive step                        |
| function parameter                           | registering an event handler          |
| function-call operator ()                    | respond to an event                   |
| getElementById method of the document object | return statement                      |
| Global object                                | scaling                               |
| global scope                                 | scaling factor                        |
| global variable                              | scope                                 |
| hexadecimal                                  | script-level variable                 |
| innerHTML property                           | shifting value                        |
| invoke a function                            | simulation                            |
| isFinite function                            | software engineering                  |
| isNaN function                               | software reusability                  |
| iterative solution                           | unescape function                     |
| listen for events                            | value property of an XHTML text field |

## Self-Review Exercises

- 9.1** Fill in the blanks in each of the following statements:
- Program modules in JavaScript are called \_\_\_\_\_.
  - A function is invoked using a(n) \_\_\_\_\_.
  - A variable known only inside the function in which it is defined is called a(n) \_\_\_\_\_.
  - The \_\_\_\_\_ statement in a called function can be used to pass the value of an expression back to the calling function.
  - The keyword \_\_\_\_\_ indicates the beginning of a function definition.
- 9.2** For the given program, state the scope (either global scope or function scope) of each of the following elements:
- The variable x.
  - The variable y.
  - The function cube.
  - The function output.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5 <!-- Exercise 9.2: cube.html -->
6 <html xmlns = "http://www.w3.org/1999/xhtml">
7 <head>
8 <title>Scoping</title>
9 <script type = "text/javascript">
10 <!--
11 var x;
12
13 function output()
14 {
15 for (x = 1; x <= 10; x++)
16 document.writeln(cube(x) + "
");
17 } // end function output
18
19 function cube(y)
20 {
21 return y * y * y;
22 } // end function cube
23 // -->
24 </script>
25 </head><body onload = "output()"></body>
26 </html>

```

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

- Programmer-defined functions, global variables and JavaScript's global functions are all part of the \_\_\_\_\_ object.
- Function \_\_\_\_\_ determines if its argument is or is not a number.
- Function \_\_\_\_\_ takes a string argument and returns a string in which all spaces, punctuation, accent characters and any other character that is not in the ASCII character set are encoded in a hexadecimal format.
- Function \_\_\_\_\_ takes a string argument representing JavaScript code to execute.
- Function \_\_\_\_\_ takes a string as its argument and returns a string in which all characters that were previously encoded with escape are decoded.

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

- An identifier's \_\_\_\_\_ is the portion of the program in which it can be used.
- The three ways to return control from a called function to a caller are \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.
- The \_\_\_\_\_ function is used to produce random numbers.
- Variables declared in a block or in a function's parameter list are of \_\_\_\_\_ scope.

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

- `method g()`  
`{`  
 `document.writeln( "Inside method g" );`  
`}`
- // This function should return the sum of its arguments  
`function sum( x, y )`  
`{`  
 `var result;`  
 `result = x + y;`  
`}`

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

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

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

to calculate the volume. The user should input the radius through an XHTML text field in a `<form>` and click an XHTML button to initiate the calculation.

## Answers to Self-Review Exercises

- 9.1** a) functions. b) function call. c) local variable. d) return. e) function.
- 9.2** a) Global scope. b) Function scope. c) Global scope. d) Global scope.
- 9.3** a) `Global`. b) `isNaN`. c) `escape`. d) `eval`. e) `unescape`.
- 9.4** a) scope. b) `return`; or `return expression`; or encountering the closing right brace of a function. c) `Math.random`. d) local.
- 9.5** a) Error: `method` is not the keyword used to begin a function definition.  
Correction: Change `method` to `function`.
- b) Error: The function is supposed to return a value, but does not.  
Correction: Either delete variable `result` and place the statement  
`return x + y;`  
in the function or add the following statement at the end of the function body:  
`return result;`
- c) Error: The semicolon after the right parenthesis that encloses the parameter list.  
Correction: Delete the semicolon after the right parenthesis of the parameter list.

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

```

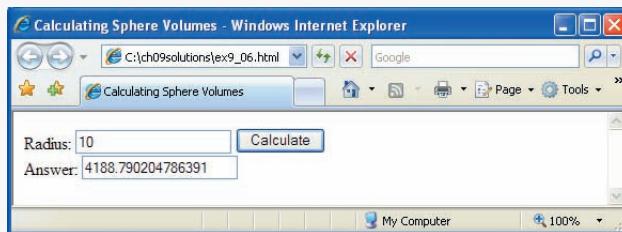
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Exercise 9.6: volume.html -->
6 <html xmlns = "http://www.w3.org/1999/xhtml">
7 <head>
8 <title>Calculating Sphere Volumes</title>
9 <script type = "text/javascript">
10 <!--
11 function displayVolume()
12 {
13 var inputField = document.getElementById("radiusField");
14 var radius = parseFloat(inputField.value);
15 var answerField = document.getElementById("answer");
16 answerField.value = sphereVolume(radius);
17 } // end function displayVolume
18
19 function sphereVolume(radius)
20 {
21 return (4.0 / 3.0) * Math.PI * Math.pow(radius, 3);
22 } // end function sphereVolume
23 // -->
```

```

24 </script>
25 </head>
26 <body>
27 <form action = "">
28 <div>
29 <label>Radius:
30 <input id = "radiusField" type = "text" /></label>
31 <input type = "button" value = "Calculate"
32 onclick = "displayVolume()" />
33

34 <label>Answer:
35 <input id = "answer" type = "text" /></label>
36 </div>
37 </form>
38 </body>
39 </html>

```



## Exercises

**9.7** Write a script that prompts the user for the radius of a circle, uses a function `circleArea` to calculate the area of the circle, and prints the area of the circle.

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

**9.9** Write function `distance` that calculates the distance between two points ( $x_1, y_1$ ) and ( $x_2, y_2$ ). All numbers and return values should be floating-point values. Incorporate this function into a script that enables the user to enter the coordinates of the points through an XHTML form.

**9.10** Answer each of the following questions:

- What does it mean to choose numbers “at random”?
- Why is the `Math.random` function useful for simulating games of chance?
- Why is it often necessary to scale and/or shift the values produced by `Math.random`?
- Why is computerized simulation of real-world situations a useful technique?

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

- $1 \leq n \leq 2$
- $1 \leq n \leq 100$
- $0 \leq n \leq 9$
- $1000 \leq n \leq 1112$

- e)  $-1 \leq n \leq 1$
- f)  $-3 \leq n \leq 11$

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

- a) 2, 4, 6, 8, 10.
- b) 3, 5, 7, 9, 11.
- c) 6, 10, 14, 18, 22.

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

*base exponent*

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

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

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

**9.16** Write a function `squareOfAsterisks` that displays a solid square of asterisks whose side is specified in integer parameter `side`. For example, if `side` is 4, the function displays

```



```

Incorporate this function into a script that reads an integer value for `side` from the user at the keyboard and performs the drawing with the `squareOfAsterisks` function.

**9.17** Modify the script created in Exercise 9.16 to also prompt the user for a character which will be used to create the square. Thus, if `side` is 5 and `fillCharacter` is `#`, the function should print

```
#####
#####
#####
#####
#####
```

**9.18** Write program segments that accomplish each of the following tasks:

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

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

**9.19** Implement the following functions:

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

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

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

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

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

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

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

**9.21** An integer number is said to be a **perfect number** if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number, because  $6 = 1 + 2 + 3$ . Write a function `perfect` that determines whether parameter `number` is a perfect number. Use this function in a script that determines and displays all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the computing power of your computer by testing numbers much larger than 1000. Display the results in a `<textarea>`.

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

- Write a function that determines whether a number is prime.
- Use this function in a script that determines and prints all the prime numbers between 1 and 10,000. How many of these 10,000 numbers do you really have to test before being sure that you have found all the primes? Display the results in a `<textarea>`.
- Initially, you might think that  $n/2$  is the upper limit for which you must test to see whether a number is prime, but you only need go as high as the square root of  $n$ . Why? Rewrite the program using the `Math.sqrt` method to calculate the square root, and run it both ways. Estimate the performance improvement.

**9.23** Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367. Incorporate the function into a script that reads a value from the user. Display the result of the function in the status bar.

**9.24** The **greatest common divisor** (GCD) of two integers is the largest integer that evenly divides each of the two numbers. Write a function `gcd` that returns the greatest common divisor of two integers. Incorporate the function into a script that reads two values from the user.

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

**9.26** Write a script that simulates coin tossing. Let the program toss the coin each time the user clicks the **Toss** button. Count the number of times each side of the coin appears. Display the results.

The program should call a separate function `flip` that takes no arguments and returns `false` for tails and `true` for heads. [Note: If the program realistically simulates the coin tossing, each side of the coin should appear approximately half the time.]

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

How much is 6 times 7?

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

**9.28** The use of computers in education is referred to as **computer-assisted instruction** (CAI). One problem that develops in CAI environments is student fatigue. This problem can be eliminated by varying the computer's dialogue to hold the student's attention. Modify the program in Exercise 9.27 to print one of a variety of comments for each correct answer and each incorrect answer. The set of responses for correct answers is as follows:

Very good!  
Excellent!  
Nice work!  
Keep up the good work!

The set of responses for incorrect answers is as follows:

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

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

**9.29** More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program in Exercise 9.28 to count the number of correct and incorrect responses typed by the student. After the student answers 10 questions, your program should calculate the percentage of correct responses. If the percentage is lower than 75 percent, print `Please ask your instructor for extra help`, and reset the program so another student can try it.

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

**9.31** Modify the program of Exercise 9.30 to count the number of guesses the player makes. If the number is 10 or fewer, display `Either you know the secret or you got lucky!` If the player guesses the number in 10 tries, display `Ahah! You know the secret!` If the player makes more than 10

guesses, display You should be able to do better! Why should it take no more than 10 guesses? Well, with each good guess, the player should be able to eliminate half of the numbers. Now show why any number 1 to 1000 can be guessed in 10 or fewer tries.

**9.32** Exercises 9.27 through 9.29 developed a computer-assisted instruction program to teach an elementary-school student multiplication. This exercise suggests enhancements to that program.

- Modify the program to allow the user to enter a grade-level capability. A grade level of 1 means to use only single-digit numbers in the problems, a grade level of 2 means to use numbers as large as two digits, and so on.
- Modify the program to allow the user to pick the type of arithmetic problems he or she wishes to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only and 5 means to intermix randomly problems of all these types.

**9.33** Modify the craps program in Fig. 9.6 to allow wagering. Initialize variable bankBalance to 1000 dollars. Prompt the player to enter a wager. Check that the wager is less than or equal to bankBalance and, if not, have the user reenter wager until a valid wager is entered. After a valid wager is entered, run one game of craps. If the player wins, increase bankBalance by wager, and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new bankBalance, check whether bankBalance has become zero and, if so, print the message Sorry. You busted! As the game progresses, print various messages to create some chatter, such as Oh, you're going for broke, huh? or Aw c'mon, take a chance! or You're up big. Now's the time to cash in your chips!. Implement the chatter as a separate function that randomly chooses the string to display.

**9.34** Write a recursive function power( base, exponent ) that, when invoked, returns

$$\text{base}^{\text{exponent}}$$

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

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

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

$$\text{base}^1 = \text{base}$$

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

**9.35** (*Visualizing Recursion*) It is interesting to watch recursion in action. Modify the factorial function in Fig. 9.11 to display its local variable and recursive-call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

**9.36** What does the following function do?

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

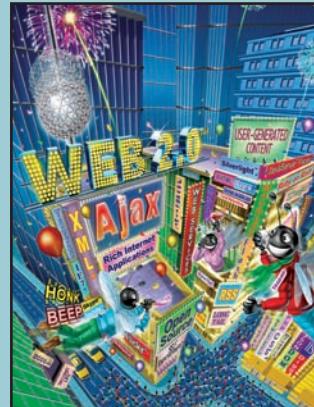
# 10

# JavaScript: Arrays

## OBJECTIVES

In this chapter you will learn:

- To use arrays to store lists and tables of values.
- To declare an array, initialize an array and refer to individual elements of an array.
- To pass arrays to functions.
- To search and sort an array.
- To declare and manipulate multidimensional arrays.



*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

*Now go, write it before them  
in a table,  
and note it in a book.*

—Isaiah 30:8

*'Tis in my memory lock'd,  
And you yourself shall keep  
the key of it.*

—William Shakespeare

**Outline**

- 10.1** Introduction
- 10.2** Arrays
- 10.3** Declaring and Allocating Arrays
- 10.4** Examples Using Arrays
- 10.5** Random Image Generator Using Arrays
- 10.6** References and Reference Parameters
- 10.7** Passing Arrays to Functions
- 10.8** Sorting Arrays
- 10.9** Searching Arrays: Linear Search and Binary Search
- 10.10** Multidimensional Arrays
- 10.11** Building an Online Quiz
- 10.12** Wrap-Up
- 10.13** Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 10.1 Introduction

This chapter serves as an introduction to the important topic of data structures. **Arrays** 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 are created. Many of the techniques demonstrated in this chapter are used frequently in Chapters 12–13 as we introduce the collections that allow a script programmer to manipulate every element of an XHTML document dynamically.

## 10.2 Arrays

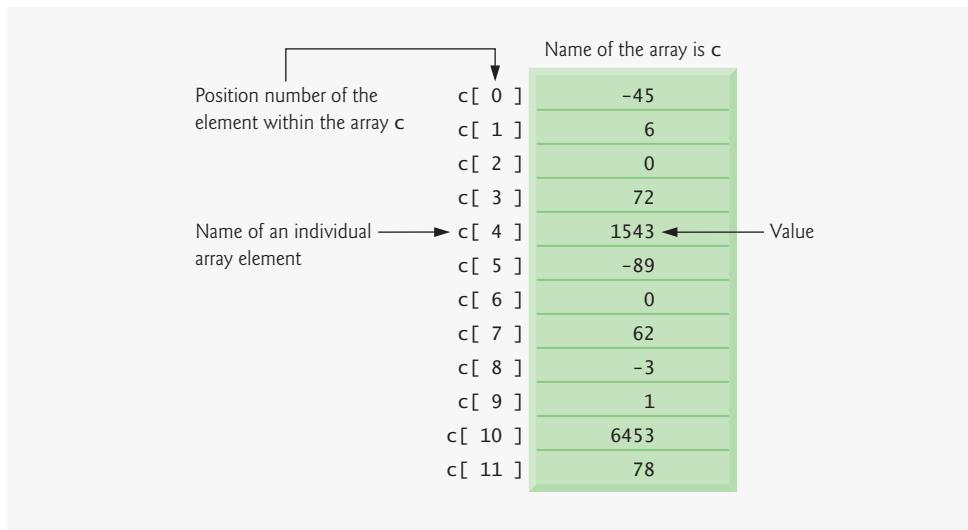
An array is a group of memory locations that all have the same name and normally are of the same type (although this attribute is not required in JavaScript). To refer to a particular location or element in the array, we specify the name of the array and the **position number** of the particular element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

Figure 10.1 shows an array of integer values named `c`. This array contains 12 **elements**. Any one of these elements may be referred to by giving the name of the array 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 of array `c` is referred to as `c[1]`, the seventh element of array `c` is referred to 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 a **subscript** (or an **index**). A subscript must be an integer or an integer expression. If a program uses an expression as a subscript, then the expression is evaluated to determine the value of the subscript. For example, if we assume that variable `a` is equal to 5 and that variable `b` is equal to 6, then the statement

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

adds 2 to array element `c[ 11 ]`. Note that a subscripted array name is a left-hand-side expression—it can be used on the left side of an assignment to place a new value into an array



**Fig. 10.1** | Array with 12 elements.

element. It can also be used on the right side of an assignment to assign its value to another left-hand side expression.

Let us examine array `c` in Fig. 10.1 more closely. The array's **name** is `c`. The **length** of array `c` is 12 and can be found using by the following expression:

`c.length`

Every array in JavaScript knows its own length. 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. To calculate the sum of the values contained in the first three elements of array `c` and store the result in variable `sum`, we would write

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

To divide the value of the seventh element of array `c` by 2 and assign the result to the variable `x`, we would write

```
x = c[6] / 2;
```



### Common Programming Error 10.1

*It is important to note the difference between the “seventh element of the array” and “array element seven.” Because array subscripts begin at 0, the seventh element of the array has a subscript of 6, while array element seven has a subscript of 7 and is actually the eighth element of the array. This confusion is a source of “off-by-one” errors.*

The brackets that enclose the array subscript are a JavaScript operator. Brackets have the same level of precedence as parentheses. The chart in Fig. 10.2 shows the precedence and associativity of the operators introduced so far in the text. They are shown from top to bottom in decreasing order of precedence, alongside their associativity and type.

| Operators        | Associativity | Type           |
|------------------|---------------|----------------|
| o [] .           | 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**. The programmer uses **operator new** to allocate dynamically (request memory for) the number of elements required by each array. Operator new creates an object as the program executes by obtaining enough memory to store an object of the type specified to the right of new. The process of creating new objects is also known as **creating an instance** or **instantiating an object**, and operator new is known as the **dynamic memory allocation operator**. Arrays are allocated with new because arrays are considered to be objects, and all objects must be created with new. To allocate 12 elements for integer array c, use the statement

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

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

```
var c; // declares the array
c = new Array(12); // allocates the array
```

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



### Common Programming Error 10.2

Assuming that the elements of an array are initialized when the array is allocated may result in logic errors.

## 10.4 Examples Using Arrays

This section presents several examples of creating and manipulating arrays.

### Creating and Initializing Arrays

The script in Fig. 10.3 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 XHTML tables.

Line 17 creates Array n1 as an array of five elements. Line 18 creates Array n2 as an empty array. Lines 21–22 use a for statement to initialize the elements of n1 to their sub-

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.3: InitArray.html -->
6 <!-- Initializing the elements of an array. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Initializing an Array</title>
10 <style type = "text/css">
11 table { width: 10em }
12 th { text-align: left }
13 </style>
14 <script type = "text/javascript">
15 <!--
16 // create (declare) two new arrays
17 var n1 = new Array(5); // allocate five-element Array
18 var n2 = new Array(); // allocate empty Array
19
20 // assign values to each element of Array n1
21 for (var i = 0; i < n1.length; ++i)
22 n1[i] = i;
23
24 // create and initialize five elements in Array n2
25 for (i = 0; i < 5; ++i)
26 n2[i] = i;
27
28 outputArray("Array n1:", n1);
29 outputArray("Array n2:", n2);
30
31 // output the heading followed by a two-column table
32 // containing subscripts and elements of "theArray"
33 function outputArray(heading, theArray)
34 {
35 document.writeln("<h2>" + heading + "</h2>");
36 document.writeln("<table border = \"1\" ");
37 document.writeln("<thead><th>Subscript</th>" +
38 "<th>Value</th></thead><tbody>");
39
40 // output the subscript and value of each array element
41 for (var i = 0; i < theArray.length; i++)
42 document.writeln("<tr><td>" + i + "</td><td>" +
43 theArray[i] + "</td></tr>");
44
45 document.writeln("</tbody></table>");
46 } // end function outputArray
47 // -->
```

**Fig. 10.3** | Initializing the elements of an array. (Part I of 2.)

```

48 </script>
49 </head><body></body>
50 </html>

```

The screenshot shows a Windows Internet Explorer window titled "Initializing an Array - Windows Internet Explorer". The address bar shows the URL "C:\examples\ch10\InitArray.html". The page content displays two tables labeled "Array n1:" and "Array n2:". Both arrays have 5 rows, indexed from 0 to 4, with values 0, 1, 2, 3, and 4 respectively.

| Subscript | Value |
|-----------|-------|
| 0         | 0     |
| 1         | 1     |
| 2         | 2     |
| 3         | 3     |
| 4         | 4     |

| Subscript | Value |
|-----------|-------|
| 0         | 0     |
| 1         | 1     |
| 2         | 2     |
| 3         | 3     |
| 4         | 4     |

**Fig. 10.3** | Initializing the elements of an array. (Part 2 of 2.)

script numbers (0 to 4). Note also the use of zero-based counting (remember, array subscripts start at 0) so that the loop can access every element of the array. Note too the use of the expression `n1.length` in the condition for the `for` statement to determine the length of the array. In this example, the length of the array is 5, so the loop continues executing as long as the value of control variable `i` is less than 5. For a five-element array, the subscript 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.

Lines 25–26 use a `for` statement to add five elements to the `Array n2` and initialize each element to its subscript number (0 to 4). Note that `Array n2` grows dynamically to accommodate the values assigned to each element of the array.



### Software Engineering Observation 10.1

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

Lines 28–29 invoke function `outputArray` (defined in lines 33–46) to display the contents of each array in an XHTML table. Function `outputArray` receives two arguments—a string to be output before the XHTML table that displays the contents of the array and the array to output. Lines 41–43 use a `for` statement to output XHTML text

that defines each row of the table. Once again, note the use of zero-based counting so that the loop can access every element of the array.



### Common Programming Error 10.3

*Referring to an element outside the Array bounds is normally a logic error.*



### Error-Prevention Tip 10.1

*When using subscripts to loop through an Array, the subscript should never go below 0 and should always be less than the number of elements in the Array (i.e., one less than the size of the Array). Make sure that the loop-terminating condition prevents the access of elements outside this range.*

If the values of an Array's elements 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 subscripts of 0, 1, 2, 3 and 4. The array size is determined by the number of values in the initializer list. Note that the preceding declaration does not require the `new` operator to create the Array object—this functionality is provided by the 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 subscripts 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 is 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 with no value specified for the third element (`n[ 2 ]`).

#### *Initializing Arrays with Initializer Lists*

The script in Fig. 10.4 creates three Array objects to demonstrate initializing arrays with initializer lists (lines 18–20) and displays each array in an XHTML table using the same function `outputArray` discussed in Fig. 10.3. Note that when Array `integers2` is displayed in the web page, the elements with subscripts 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 the declaration in line 20 in the script.

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

The script in Fig. 10.5 sums the values contained in `theArray`, the 10-element integer array declared, allocated and initialized in line 13. The statement in line 19 in the body of the first `for` statement does the totaling. Note that the values supplied as initializers for array `theArray` could be read into the program using an XHTML form.

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 will see in Chapters 12–13, for each element in a collection). This process is also known as **iterating over the elements of an array**. Lines 25–26 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 key-

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.4: InitArray2.html -->
6 <!-- Declaring and initializing arrays. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Initializing an Array with a Declaration</title>
10 <style type = "text/css">
11 table { width: 15em }
12 th { text-align: left }
13 </style>
14 <script type = "text/javascript">
15 <!--
16 // Initializer list specifies the number of elements and
17 // a value for each element.
18 var colors = new Array("cyan", "magenta", "yellow", "black");
19 var integers1 = [2, 4, 6, 8];
20 var integers2 = [2, , , 8];
21
22 outputArray("Array colors contains", colors);
23 outputArray("Array integers1 contains", integers1);
24 outputArray("Array integers2 contains", integers2);
25
26 // output the heading followed by a two-column table
27 // containing the subscripts and elements of theArray
28 function outputArray(heading, theArray)
29 {
30 document.writeln("<h2>" + heading + "</h2>");
31 document.writeln("<table border = \"1\" ");
32 document.writeln("<thead><th>Subscript</th>" +
33 "<th>Value</th></thead><tbody>");
34
35 // output the subscript and value of each array element
36 for (var i = 0; i < theArray.length; i++)
37 document.writeln("<tr><td>" + i + "</td><td>" +
38 theArray[i] + "</td></tr>");
39
40 document.writeln("</tbody></table>");
41 } // end function outputArray
42 // --
43 </script>
44 </head><body></body>
45 </html>
```

**Fig. 10.4** | Declaring and initializing arrays. (Part I of 2.)

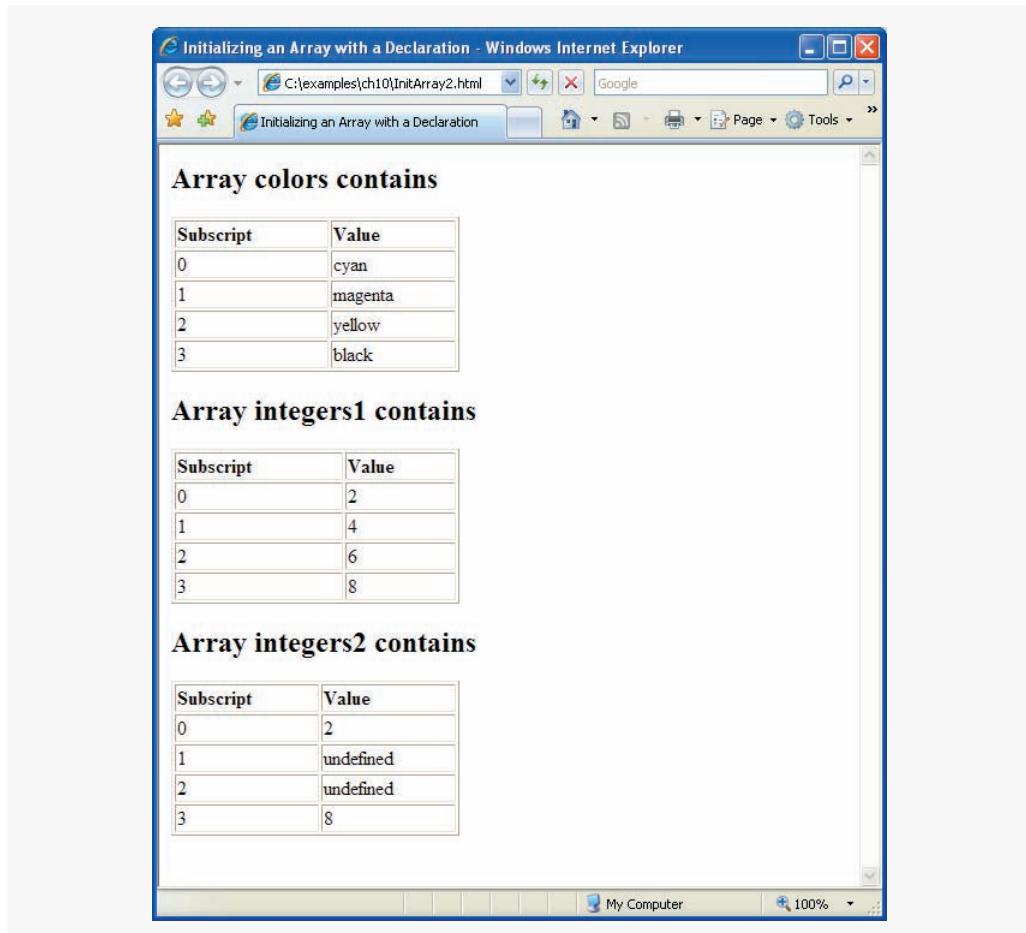


Fig. 10.4 | Declaring and initializing arrays. (Part 2 of 2.)

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.5: SumArray.html -->
6 <!-- Summing elements of an array. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Sum the Elements of an Array</title>
10
11 <script type = "text/javascript">
12 <!--
13 var theArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
14 var total1 = 0, total2 = 0;
15

```

Fig. 10.5 | Summing elements of an array. (Part 1 of 2.)

```

16 // iterates through the elements of the array in order and adds
17 // each element's value to total1
18 for (var i = 0; i < theArray.length; i++)
19 total1 += theArray[i];
20
21 document.writeln("Total using subscripts: " + total1);
22
23 // iterates through the elements of the array using a for...in
24 // statement to add each element's value to total2
25 for (var element in theArray)
26 total2 += theArray[element];
27
28 document.writeln("
Total using for...in: " + total2);
29 // -->
30
```

</script>

</head><body></body>

</html>



**Fig. 10.5** | Summing elements of an array. (Part 2 of 2.)

word `in` (`theArray` in this case). When using `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 a subscript for `theArray`. In the case of an Array, the value assigned is a subscript 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 of the Array. Note that a `for...in` statement skips any undefined elements in the array.*

### Using the Elements of an Array as Counters

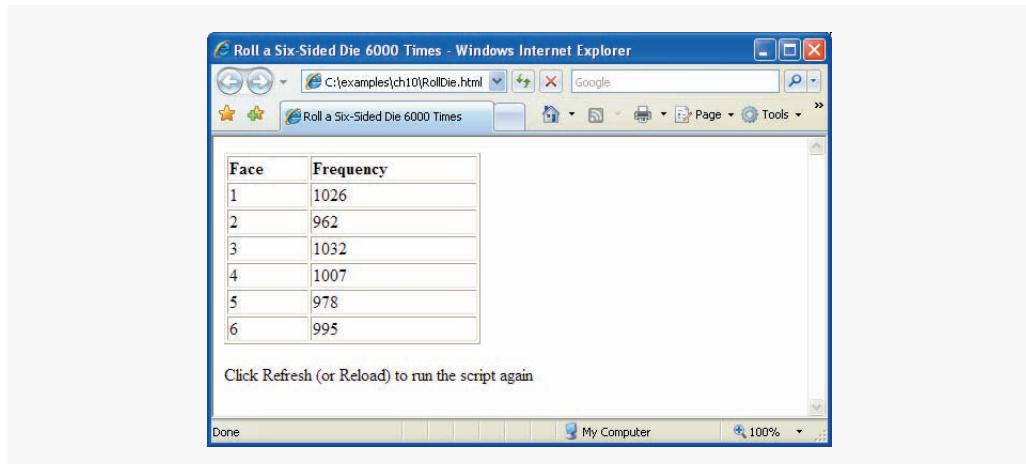
In Chapter 9, we indicated that there is a more elegant way to implement the dice-rolling program in Fig. 9.5. The program rolled a single six-sided die 6000 times and used a `switch` statement to total the number of times each value was rolled. An array version of this script is shown in Fig. 10.6. The `switch` statement in Fig. 9.5 is replaced by line 24 of this program. This line uses the random face value as the subscript for the array `frequency` to determine which element to increment during each iteration of the loop. Because the random number calculation in line 23 produces numbers from 1 to 6 (the values

for a six-sided die), the frequency array must be large enough to allow subscript values of 1 to 6. The smallest number of elements required for an array to have these subscript values is seven elements (subscript values from 0 to 6). In this program, we ignore element 0 of array frequency and use only the elements that correspond to values on the sides of a die. Also, lines 32–34 of this program use a loop to generate the table that was written one line at a time in Fig. 9.5. Because we can loop through array frequency to help produce the output, we do not have to enumerate each XHTML table row as we did in Fig. 9.5.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.6: RollDie.html -->
6 <!-- Dice-rolling program using an array instead of a switch. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Roll a Six-Sided Die 6000 Times</title>
10 <style type = "text/css">
11 table { width: 15em }
12 th { text-align: left }
13 </style>
14 <script type = "text/javascript">
15 <!--
16 var face;
17 var frequency = [, 0, 0, 0, 0, 0, 0]; // leave frequency[0]
18 // uninitialized
19
20 // summarize results
21 for (var roll = 1; roll <= 6000; ++roll)
22 {
23 face = Math.floor(1 + Math.random() * 6);
24 ++frequency[face];
25 } // end for
26
27 document.writeln("<table border = \"1\"><thead>");
28 document.writeln("<th>Face</th> +
29 "<th>Frequency</th></thead><tbody>");
30
31 // generate entire table of frequencies for each face
32 for (face = 1; face < frequency.length; ++face)
33 document.writeln("<tr><td>" + face + "</td><td>" +
34 frequency[face] + "</td></tr>");
35
36 document.writeln("</tbody></table>");
37 // --
38 </script>
39 </head>
40 <body>
41 <p>Click Refresh (or Reload) to run the script again</p>
42 </body>
43 </html>
```

**Fig. 10.6** | Dice-rolling program using an array instead of a switch. (Part I of 2.)



**Fig. 10.6** | Dice-rolling program using an array instead of a switch. (Part 2 of 2.)

## 10.5 Random Image Generator Using Arrays

In Chapter 9, we created a random image generator that required image files to be named 1.gif, 2.gif, ..., 7.gif. In this example (Fig. 10.7), we create a more elegant random image generator that does not require the image filenames to be integers. This version of the random image generator uses an array `pictures` to store the names of the image files as strings. The script generates a random integer and uses it as a subscript into the `pictures` array. The script outputs an XHTML `img` element whose `src` attribute contains the image filename located in the randomly selected position in the `pictures` array.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.7: RandomPicture2.html -->
6 <!-- Random image generation using arrays. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Random Image Generator</title>
10 <style type = "text/css">
11 table { width: 15em }
12 th { text-align: left }
13 </style>
14 <script type = "text/javascript">
15 <!--
16 var pictures =
17 ["CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO"];
18
19 // pick a random image from the pictures array and displays by
20 // creating an img tag and appending the src attribute to the
21 // filename

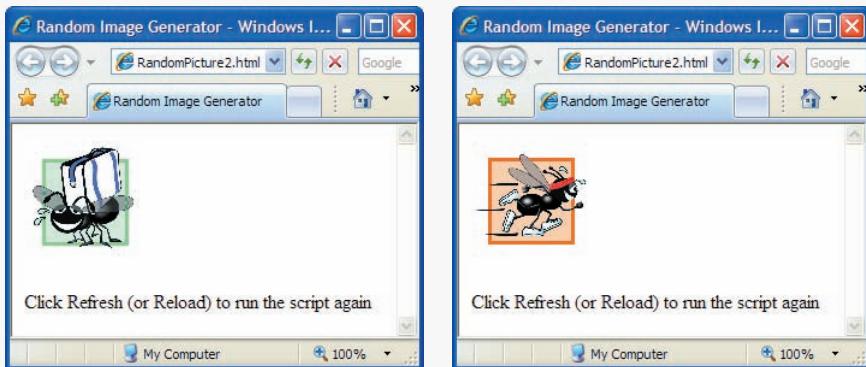
```

**Fig. 10.7** | Random image generation using arrays. (Part 1 of 2.)

```

22 document.write ("<img src = \\" +
23 pictures[Math.floor(Math.random() * 7)] + ".gif\\" />");
24 // -->
25 </script>
26 </head>
27 <body>
28 <p>Click Refresh (or Reload) to run the script again</p>
29 </body>
30 </html>

```



**Fig. 10.7 |** Random image generation using arrays. (Part 2 of 2.)

The script declares the array `pictures` in lines 16–17 and initializes it with the names of seven image files. Lines 22–23 create the `img` tag that displays the random image on the web page. Line 22 opens the `img` tag and begins the `src` attribute. Line 23 generates a random integer from 0 to 6 as an index into the `pictures` array, the result of which is a randomly selected image filename. The expression

```
pictures[Math.floor(Math.random() * 7)]
```

evaluates to a string from the `pictures` array, which then is written to the document (line 23). Line 23 completes the `img` tag with the extension of the image file (`.gif`).

## 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 direct access to the caller's data and allows it to modify the data if it so chooses. This procedure is accomplished by passing to the called function the actual **location in memory** (also called the **address**) 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 called function do not affect the original variable's value in the calling function. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems.*



### Software Engineering Observation 10.2

*Unlike some other languages, JavaScript does not allow the programmer to choose whether to pass each argument by value or by reference. Numbers, boolean values and strings are passed by value. Objects are passed to functions by reference. When a function receives a reference to an object, the function can manipulate the object directly.*



### Software Engineering Observation 10.3

*When returning information from a function via a return statement, numbers and boolean values are always returned by value (i.e., a copy is returned), and objects are always returned by reference (i.e., a reference to the object is returned). Note that, in the pass-by-reference case, it is not necessary to return the new value, since the object is already modified.*

To pass a reference to an object into a function, simply specify the reference name in the function call. Normally, 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 can be accessed directly by the called function.

Arrays are objects in JavaScript, so Arrays are passed to a function by reference—a called function can access the elements of the caller's original Arrays. The name of an array actually is a reference to an object that contains the array elements and the `length` variable, which indicates the number of elements in the array. In the next section, we demonstrate pass-by-value and pass-by-reference, using arrays.

## 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. 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 size of the array separately as an argument. Figure 10.3 illustrated this concept when we passed Arrays `n1` and `n2` to function `outputArray` to display each Array's contents.

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 (the objects referred to by individual elements of an Array of objects are still passed by reference). Such simple single pieces of data are called **scalars**, or **scalar quantities**. To pass an array element to a function, use the subscripted name of the element as an argument in the function call.

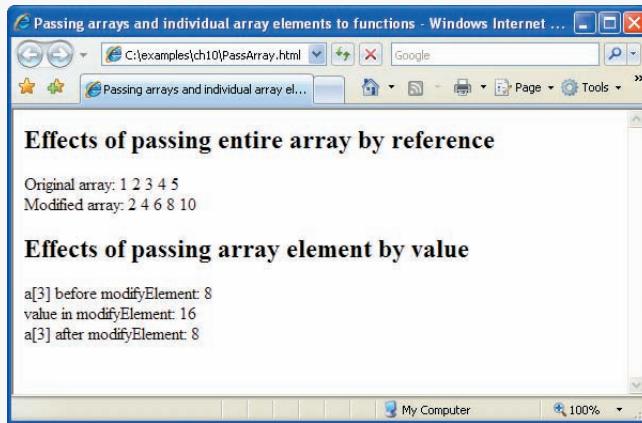
```
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.8: PassArray.html -->
6 <!-- Passing arrays and individual array elements to functions. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Passing arrays and individual array
10 elements to functions</title>
11 <script type = "text/javascript">
12 <!--
13 var a = [1, 2, 3, 4, 5];
14
15 document.writeln("<h2>Effects of passing entire " +
16 "array by reference</h2>");
17 outputArray("Original array: ", a);
18
19 modifyArray(a); // array a passed by reference
20
21 outputArray("Modified array: ", a);
22
23 document.writeln("<h2>Effects of passing array " +
24 "element by value</h2>" +
25 "a[3] before modifyElement: " + a[3]);
26
27 modifyElement(a[3]); // array element a[3] passed by value
28
29 document.writeln("
a[3] after modifyElement: " + a[3]);
30
31 // outputs heading followed by the contents of "theArray"
32 function outputArray(heading, theArray)
33 {
34 document.writeln(
35 heading + theArray.join(" ") + "
");
36 } // end function outputArray
37
38 // function that modifies the elements of an array
39 function modifyArray(theArray)
40 {
41 for (var j in theArray)
42 theArray[j] *= 2;
43 } // end function modifyArray
44
45 // function that modifies the value passed
46 function modifyElement(e)
47 {
48 e *= 2; // scales element e only for the duration of the
49 // function
50 document.writeln("
value in modifyElement: " + e);
51 } // end function modifyElement
52 // -->
53 </script>
```

Fig. 10.8 | Passing arrays and individual array elements to functions. (Part I of 2.)

```

54 </head><body></body>
55 </html>

```



**Fig. 10.8** | Passing arrays and individual array elements to functions. (Part 2 of 2.)

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. Unlike other programming languages, JavaScript does not provide a special syntax for this purpose. JavaScript 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` (the argument supplied in the calling function must be an Array). 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 Fig. 10.8 demonstrates the difference between passing an entire array and passing an array element.



#### Software Engineering Observation 10.4

*JavaScript does not check the number of arguments or types of arguments that are passed to a function. It is possible to pass any number of values to a function. JavaScript will attempt to perform conversions when the values are used.*

The statement in line 17 invokes function `outputArray` to display the contents of array `a` before it is modified. Function `outputArray` (defined in lines 32–36) receives a string to output and the array to output. The statement in lines 34–35 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 elements of the array in the string that is returned. If the argument is not specified, the empty string is used as the separator.

Line 19 invokes function `modifyArray` (lines 39–43) and passes it array `a`. The `modifyArray` function multiplies each element by 2. To illustrate that array `a`'s elements were modified, the statement in line 21 invokes function `outputArray` again to display the

contents of array `a` after it is modified. As the screen capture shows, the elements of `a` are indeed modified by `modifyArray`.

To show the value of `a[ 3 ]` before the call to `modifyElement`, line 25 outputs the value of `a[ 3 ]`. Line 27 invokes `modifyElement` (lines 46–51) and passes `a[ 3 ]` as the argument. Remember that `a[ 3 ]` actually is one integer value in the array `a`. Also remember 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 and stores the result in its parameter `e`. The parameter of function `modifyElement` is a local variable in that function, so when the function terminates, the local variable is no longer accessible. Thus, when control is returned to the main script, the unmodified value of `a[ 3 ]` is displayed by the statement in line 29.

## 10.8 Sorting Arrays

**Sorting** data (putting data in a particular order, such as ascending or descending) is one of the most important computing functions. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name, to make it easy to find phone numbers. Virtually every organization must sort some data—in many cases, massive amounts of data. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science.

The `Array` object in JavaScript has a built-in method `sort` for sorting arrays. Figure 10.9 demonstrates the `Array` object's `sort` method.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.9: Sort.html -->
6 <!-- Sorting an array with sort. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Sorting an Array with Array Method sort</title>
10 <script type = "text/javascript">
11 <!--
12 var a = [10, 1, 9, 2, 8, 3, 7, 4, 6, 5];
13
14 document.writeln("<h1>Sorting an Array</h1>");
15 outputArray("Data items in original order: ", a);
16 a.sort(compareIntegers); // sort the array
17 outputArray("Data items in ascending order: ", a);
18
19 // output the heading followed by the contents of theArray
20 function outputArray(heading, theArray)
21 {
22 document.writeln("<p>" + heading +
23 theArray.join(" ") + "</p>");
24 } // end function outputArray

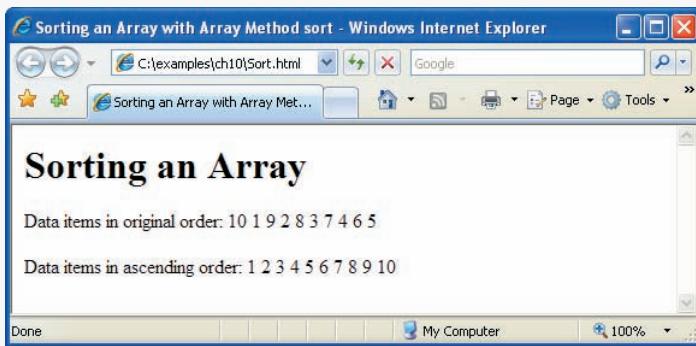
```

Fig. 10.9 | Sorting an array with `sort`. (Part I of 2.)

```

25 // comparison function for use with sort
26 function compareIntegers(value1, value2)
27 {
28 return parseInt(value1) - parseInt(value2);
29 } // end function compareIntegers
30 // -->
31 </script>
32 </head><body></body>
33
34 </html>

```



**Fig. 10.9** | Sorting an array with `sort`. (Part 2 of 2.)

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, *JavaScript: Objects*.] In this example, we'd like to sort an array of integers.

Method `sort` takes as its optional argument the name of a function (called the **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
- zero if the arguments are equal, or
- a positive value if the first argument is greater than the second argument

This example uses function `compareIntegers` (defined in lines 27–30) as the comparator function for method `sort`. It calculates the difference between the integer values of its two arguments (function `parseInt` ensures that the arguments are handled properly as integers). If the first argument is less than the second argument, the difference will be a negative value. If the arguments are equal, the difference will be zero. If the first argument is greater than the second argument, the difference will be a positive value.

Line 16 invokes `Array` object `a`'s `sort` method and passes function `compareIntegers` as an argument. Method `sort` receives function `compareIntegers` as an argument, then uses the function 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: Linear Search and Binary Search

Often, a programmer will be working with large amounts of data stored in arrays. It may be 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. In this section we discuss two searching techniques—the simple linear search technique (Fig. 10.10) and the more efficient binary search technique (Fig. 10.11).

### *Searching an Array with Linear Search*

The script in Fig. 10.10 performs a **linear search** on an array. Function `linearSearch` (defined in lines 42–50) uses a `for` statement containing an `if` statement to compare each element of an array with a search key (lines 45–47). If the search key is found, the function returns the subscript value (line 47) of the element to indicate the exact position of the search key in the array. [Note: The loop (lines 45–47) in the `linearSearch` function terminates, and the function returns control to the caller as soon as the `return` statement in its body executes.] If the search key is not found, the function returns a value of `-1`. The function returns the value `-1` because it is not a valid subscript number.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.10: LinearSearch.html -->
6 <!-- Linear search of an array. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Linear Search of an Array</title>
10 <script type = "text/javascript">
11 <!--
12 var a = new Array(100); // create an Array
13
14 // fill Array with even integer values from 0 to 198
15 for (var i = 0; i < a.length; ++i)
16 a[i] = 2 * i;
17
18 // function called when "Search" button is pressed
19 function buttonPressed()
20 {
21 // get the input text field
22 var inputVal = document.getElementById("inputVal");
23
24 // get the result text field
25 var result = document.getElementById("result");
26
27 // get the search key from the input text field
28 var searchKey = inputVal.value;
29
30 // Array a is passed to linearSearch even though it
31 // is a global variable. Normally an array will
32 // be passed to a method for searching.

```

**Fig. 10.10** | Linear search of an array. (Part I of 2.)

```
33 var element = linearSearch(a, parseInt(searchKey));
34
35 if (element != -1)
36 result.value = "Found value in element " + element;
37 else
38 result.value = "Value not found";
39 } // end function buttonPressed
40
41 // Search "theArray" for the specified "key" value
42 function linearSearch(theArray, key)
43 {
44 // iterates through each element of the array in order
45 for (var n = 0; n < theArray.length; ++n)
46 if (theArray[n] == key)
47 return n;
48
49 return -1;
50 } // end function linearSearch
51 // -->
52
```

```
</script>
```

```
</head>
```

```
<body>
```

```
 <form action = "">
```

```
 <p>Enter integer search key

```

```
 <input id = "inputVal" type = "text" />
```

```
 <input type = "button" value = "Search"
```

```
 onclick = "buttonPressed()" />
</p>
```

```
 <p>Result

```

```
 <input id = "result" type = "text" size = "30" /></p>
```

```
 </form>
```

```
</body>
```

```
</html>
```

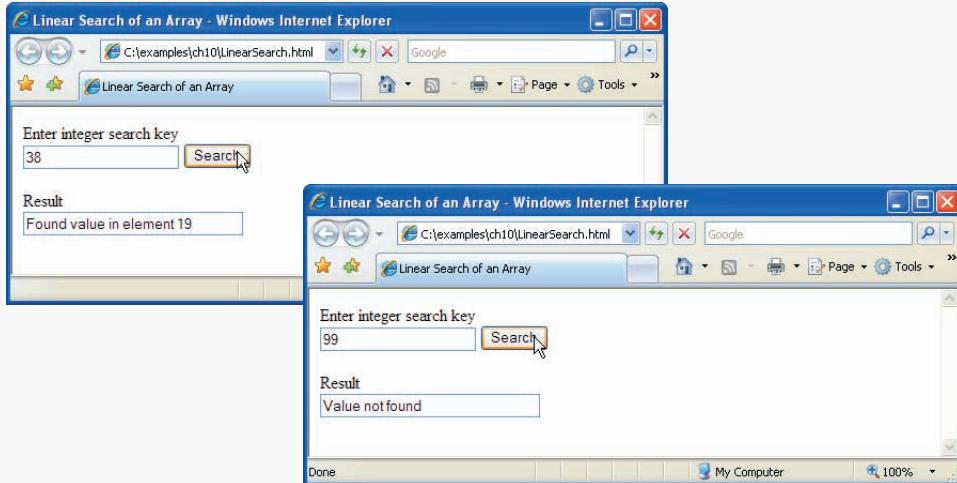


Fig. 10.10 | Linear search of an array. (Part 2 of 2.)

If the array being searched is not in any particular order, the value is just as likely to be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array.

The program contains a 100-element array (defined in line 12) filled with the even integers from 0 to 198. The user types the search key in a text field (defined in the XHTML form in lines 56–63) and clicks the **Search** button to start the search. [Note: The array is passed to `linearSearch` even though the array is a global script variable. We do this because arrays are normally passed to functions for searching.]

### *Searching an Array with Binary Search*

The linear search method works well for small arrays or for unsorted arrays. However, for large arrays, linear searching is inefficient. The **binary search algorithm** is more efficient, but it requires that the array be sorted. The first iteration of this algorithm tests the middle element in the array. If this matches the search key, the algorithm ends. Assuming the array is sorted in ascending order, then if the search key is less than the middle element, it cannot match any element in the second half of the array and the algorithm continues with only the first half of the array (i.e., the first element up to, but not including, the middle element). If the search key is greater than the middle element, it cannot match any element in the first half of the array and the algorithm continues with only the second half of the array (i.e., the element after the middle element through the last element). Each iteration tests the middle value of the remaining portion of the array. If the search key does not match the element, the algorithm eliminates half of the remaining elements. The algorithm ends by either finding an element that matches the search key or reducing the subarray to zero size.

As an example consider the sorted 15-element array

2    3    5    10    27    30    34    51    56    65    77    81    82    93    99

and a search key of 65. A program implementing the binary search algorithm would first check whether 51 is the search key (because 51 is the middle element of the array). The search key (65) is larger than 51, so 51 is discarded along with the first half of the array (all elements smaller than 51.) Next, the algorithm checks whether 81 (the middle element of the remainder of the array) matches the search key. The search key (65) is smaller than 81, so 81 is discarded along with the elements larger than 81. After just two tests, the algorithm has narrowed the number of values to check to three (56, 65 and 77). The algorithm then checks 65 (which indeed matches the search key), and returns the index of the array element containing 65. This algorithm required just three comparisons to determine whether the search key matched an element of the array. Using a linear search algorithm would have required 10 comparisons. [Note: In this example, we have chosen to use an array with 15 elements so that there will always be an obvious middle element in the array. With an even number of elements, the middle of the array lies between two elements. We implement the algorithm to choose the lower of those two elements.]

Figure 10.11 presents the iterative version of function `binarySearch` (lines 40–64). Function `binarySearch` is called from line 31 of function `buttonPressed` (lines 18–37)—the event handler for the **Search** button in the XHTML form. Function `binarySearch` receives two arguments—an array called `theArray` (the array to search) and `key` (the search key). The array is passed to `binarySearch` even though the array is a global variable. Once

again, we do this because an array is normally passed to a function for searching. If key matches the middle element of a subarray (line 55), middle (the subscript of the current element) is returned, to indicate that the value was found and the search is complete. If key does not match the middle element of a subarray, the low subscript or the high subscript (both declared in the function) is adjusted, so that a smaller subarray can be searched. If key is less than the middle element (line 57), the high subscript is set to middle - 1 and the search is continued on the elements from low to middle - 1. If key is greater than the middle element (line 59), the low subscript is set to middle + 1 and the search is continued on the elements from middle + 1 to high. These comparisons are performed by the nested if...else statement in lines 55–60.

```
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.11: BinarySearch.html -->
6 <!-- Binary search of an array. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Binary Search</title>
10 <script type = "text/javascript">
11 <!--
12 var a = new Array(15);
13
14 for (var i = 0; i < a.length; ++i)
15 a[i] = 2 * i;
16
17 // function called when "Search" button is pressed
18 function buttonPressed()
19 {
20 var inputVal = document.getElementById("inputVal");
21 var result = document.getElementById("result");
22 var searchKey = inputVal.value;
23
24 result.value = "Portions of array searched\n";
25
26 // Array a is passed to binarySearch even though it
27 // is a global variable. This is done because
28 // normally an array is passed to a method
29 // for searching.
30 var element =
31 binarySearch(a, parseInt(searchKey));
32
33 if (element != -1)
34 result.value += "\nFound value in element " + element;
35 else
36 result.value += "\nValue not found";
37 } // end function buttonPressed
38
```

**Fig. 10.11** | Binary search of an array. (Part I of 3.)

```
39 // binary search function
40 function binarySearch(theArray, key)
41 {
42 var low = 0; // low subscript
43 var high = theArray.length - 1; // high subscript
44 var middle; // middle subscript
45
46 while (low <= high) {
47 middle = (low + high) / 2;
48
49 // The following line is used to display the
50 // part of theArray currently being manipulated
51 // during each iteration of the binary
52 // search loop.
53 buildOutput(theArray, low, middle, high);
54
55 if (key == theArray[middle]) // match
56 return middle;
57 else if (key < theArray[middle])
58 high = middle - 1; // search low end of array
59 else
60 low = middle + 1; // search high end of array
61 } // end while
62
63 return -1; // searchKey not found
64 } // end function binarySearch
65
66 // Build one row of output showing the current
67 // part of the array being processed.
68 function buildOutput(theArray, low, mid, high)
69 {
70 var result = document.getElementById("result");
71
72 for (var i = 0; i < theArray.length; i++)
73 {
74 if (i < low || i > high)
75 result.value += " ";
76 else if (i == mid) // mark middle element in output
77 result.value += theArray[i] +
78 (theArray[i] < 10 ? "* " : "* ");
79 else
80 result.value += theArray[i] +
81 (theArray[i] < 10 ? " " : " ");
82 } // end for
83
84 result.value += "\n";
85 } // end function buildOutput
86 // -->
87 </script>
88 </head>
89
```

Fig. 10.11 | Binary search of an array. (Part 2 of 3.)

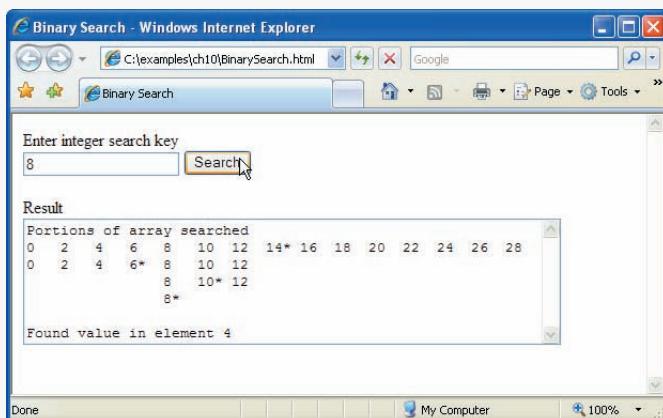
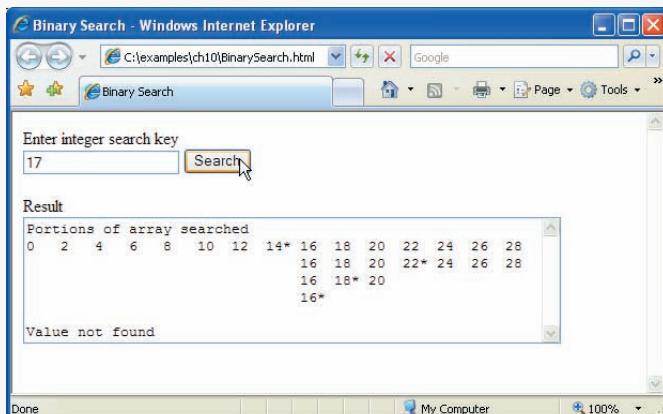
```

90 <body>
91 <form action = "">
92 <p>Enter integer search key

93 <input id = "inputVal" type = "text" />
94 <input type = "button" value = "Search"
95 onclick = "buttonPressed()" />
</p>
96 <p>Result

97 <textarea id = "result" rows = "7" cols = "60">
98 </textarea></p>
99 </form>
100 </body>
101 </html>

```



**Fig. 10.11** | Binary search of an array. (Part 3 of 3.)

In a worst-case scenario, searching an array of 1023 elements will take only 10 comparisons using a binary search. Repeatedly dividing 1024 by 2 (because after each comparison we are able to eliminate half of the array) yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 ( $2^{10}$ ) is divided by 2 only ten times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of

about one million elements takes a maximum of 20 comparisons to find the key. An array of about one billion elements takes a maximum of 30 comparisons to find the key. When searching a sorted array, this is a tremendous increase in performance over the linear search that required comparing the search key to an average of half the elements in the array. For a one-billion-element array, this is the difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

Our program uses a 15-element array (defined at line 12). The first power of 2 greater than the number of elements is 16 ( $2^4$ ), so `binarySearch` requires at most four comparisons to find the key. To illustrate this, line 53 calls method `buildOutput` (declared in lines 68–85) to output each subarray during the binary search process. Method `buildOutput` marks the middle element in each subarray with an asterisk (\*) to indicate the element with which the key is compared. No matter what search key is entered, each search in this example results in a maximum of four lines of output—one per comparison.

## 10.10 Multidimensional Arrays

Multidimensional arrays with two subscripts 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 subscripts; by convention, the first identifies the element's row, and the second identifies the element's column. Arrays that require two subscripts 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 does allow the programmer 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 10.12 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 4 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. 10.12 by an element name of the form `a[ i ][ j ]`; `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify

|       | Column 0                 | Column 1                 | Column 2                 | Column 3                 |
|-------|--------------------------|--------------------------|--------------------------|--------------------------|
| Row 0 | <code>a[ 0 ][ 0 ]</code> | <code>a[ 0 ][ 1 ]</code> | <code>a[ 0 ][ 2 ]</code> | <code>a[ 0 ][ 3 ]</code> |
| Row 1 | <code>a[ 1 ][ 0 ]</code> | <code>a[ 1 ][ 1 ]</code> | <code>a[ 1 ][ 2 ]</code> | <code>a[ 1 ][ 3 ]</code> |
| Row 2 | <code>a[ 2 ][ 0 ]</code> | <code>a[ 2 ][ 1 ]</code> | <code>a[ 2 ][ 2 ]</code> | <code>a[ 2 ][ 3 ]</code> |

**Fig. 10.12** | Two-dimensional array with three rows and four columns.

the row and column, respectively, of each element in `a`. Note that the names of the elements in the first row all have a first subscript of 0; the names of the elements in the fourth column all have a second subscript of 3.

### *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 sub initializer 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 sub-array 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 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*

Figure 10.13 initializes two-dimensional arrays in declarations and uses nested `for...in` loops to **traverse the arrays** (i.e., manipulate every element of the array).

```
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.13: InitArray3.html -->
6 <!-- Initializing multidimensional arrays. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
```

**Fig. 10.13** | Initializing multidimensional arrays. (Part I of 2.)

```

8 <head>
9 <title>Initializing Multidimensional Arrays</title>
10 <script type = "text/javascript">
11 <!--
12 var array1 = [[1, 2, 3], // first row
13 [4, 5, 6]]; // second row
14 var array2 = [[1, 2], // first row
15 [3], // second row
16 [4, 5, 6]]; // third row
17
18 outputArray("Values in array1 by row", array1);
19 outputArray("Values in array2 by row", array2);
20
21 function outputArray(heading, theArray)
22 {
23 document.writeln("<h2>" + heading + "</h2><pre>");
24
25 // iterates through the set of one-dimensional arrays
26 for (var i in theArray)
27 {
28 // iterates through the elements of each one-dimensional
29 // array
30 for (var j in theArray[i])
31 document.write(theArray[i][j] + " ");
32
33 document.writeln("
");
34 } // end for
35
36 document.writeln("</pre>");
37 } // end function outputArray
38 // -->
39 </script>
40 </head><body></body>
41 </html>

```

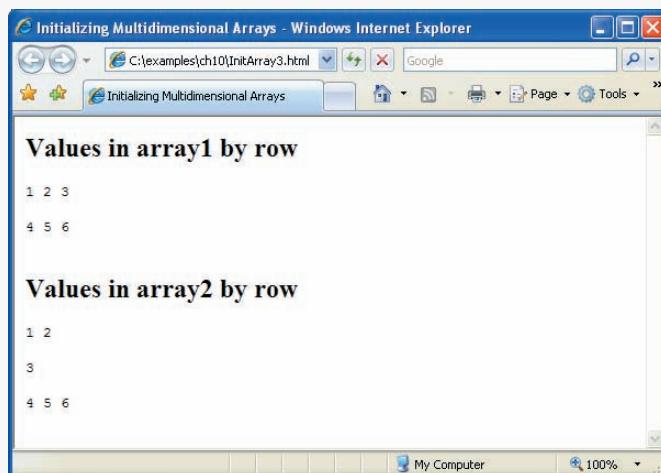


Fig. 10.13 | Initializing multidimensional arrays. (Part 2 of 2.)

The program declares two arrays in main script (in the XHTML head element). The declaration of array1 (lines 12–13 provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3; the second sublist initializes the second row of the array to the values 4, 5 and 6. The declaration of array2 (lines 14–16) provides six initializers in three sublists. The sublist for the first row explicitly initializes the first row to have two elements, with values 1 and 2, respectively. The sublist for the second row initializes the second row to have one element, with value 3. The sublist for the third row initializes the third row to the values 4, 5 and 6.

The script calls function outputArray from lines 18–19 to display each array's elements in the web page. Function outputArray (lines 21–37) receives two arguments—a string heading to output before the array and the array to output (called theArray). Note the use of a nested for...in statement to output the rows of each two-dimensional array. 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:

```
for (var i = 0; i < theArray.length; ++i) {
 for (var j = 0; j < theArray[i].length; ++j)
 document.write(theArray[i][j] + " ");
 document.writeln("
");
}
```

In the outer for statement, the expression theArray.length determines the number of rows in the array. In the inner for statement, the expression theArray[i].length determines the number of columns in each row of the array. This condition 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 the third row of array a in Fig. 10.12 to zero:

```
for (var col = 0; col < a[2].length; ++col)
 a[2][col] = 0;
```

We specified the *third* row; therefore, we know that the first subscript is always 2 (0 is the first row and 1 is the second row). The for loop varies only the second subscript (i.e., the column subscript). 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;
for (var row = 0; row < a.length; ++row)
 for (var col = 0; col < a[row].length; ++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` subscript to 0, so that the elements of the first row may be totaled by the inner `for` statement. The outer `for` statement then increments `row` to 1, so that the elements of the second row can be totaled. Then the outer `for` statement increments `row` to 2, so that the elements of the third row 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];
```

## 10.11 Building an Online Quiz

Online quizzes and polls are popular web applications often used for educational purposes or just for fun. Web developers typically build quizzes using simple XHTML forms and process the results with JavaScript. Arrays allow a programmer to represent several possible answer choices in a single data structure. Figure 10.14 contains an online quiz consisting of one question. The quiz page contains one of the tip icons used throughout this book and an XHTML form in which the user identifies the type of tip the image represents by selecting one of four radio buttons. After the user selects one of the radio button choices and submits the form, the script determines whether the user selected the correct type of tip to match the mystery image. The JavaScript function that checks the user's answer combines several of the concepts from the current chapter and previous chapters in a concise and useful script.

Before we discuss the script code, we first discuss the `body` element (lines 25–48) of the XHTML document. The `body`'s GUI components play an important role in the script.

Lines 26–47 define the `form` that presents the quiz to users. Line 26 begins the `form` element and specifies the `onsubmit` attribute to "checkAnswers()", indicating that the interpreter should execute the JavaScript function `checkAnswers` (lines 12–21) when the user submits the form (i.e., clicks the `Submit` button or presses `Enter`).

Line 29 adds the tip image to the page. Lines 32–42 display the radio buttons and corresponding `labels` that display possible answer choices. Lines 44–45 add the `submit` and `reset` buttons to the page.

We now examine the script used to check the answer submitted by the user. Lines 12–21 declare the function `checkAnswers` that contains all the JavaScript required to grade the quiz. The `if...else` statement in lines 17–20 determines whether the user answered the question correctly. The image that the user is asked to identify is the Error-Prevention Tip icon. Thus the correct answer to the quiz corresponds to the second radio button.

An XHTML form's elements can be accessed individually using `getElementById` or through the `elements` property of the containing `form` object. The `elements` property

```

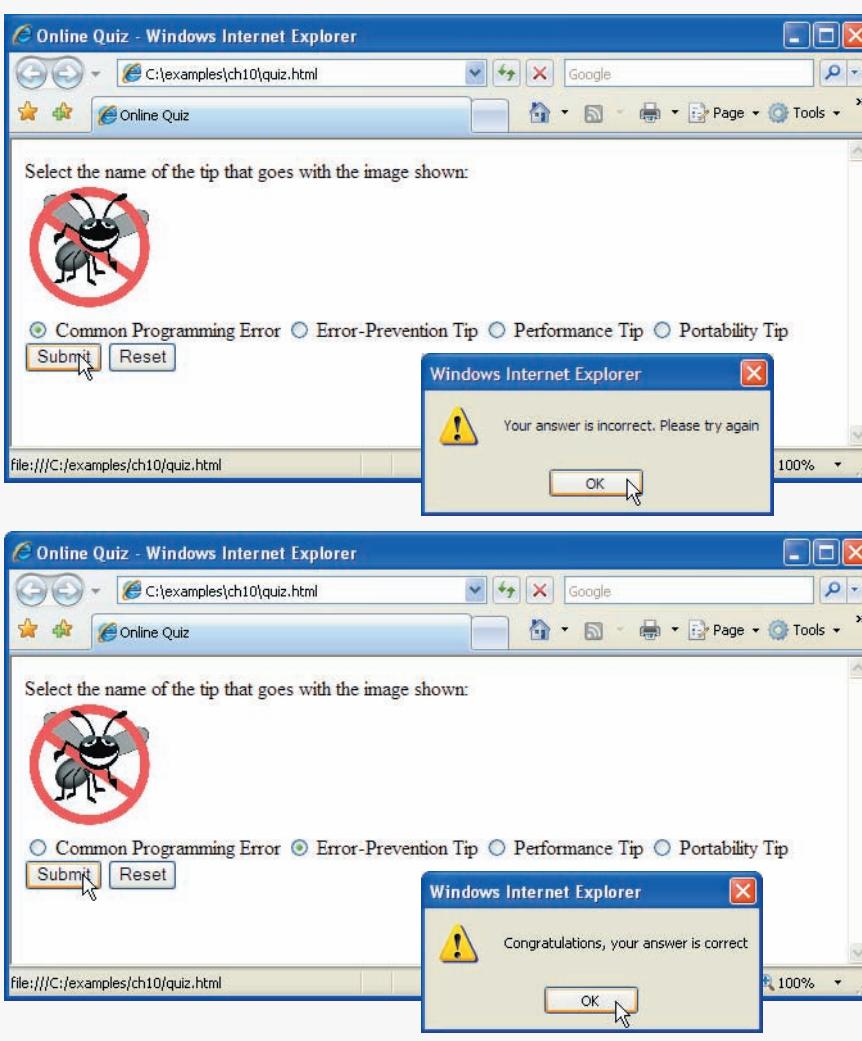
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 10.14: quiz.html -->
6 <!-- Online quiz graded with JavaScript. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Online Quiz</title>
10 <script type = "text/JavaScript">
11 <!--
12 function checkAnswers()
13 {
14 var myQuiz = document.getElementById("myQuiz");
15
16 // determine whether the answer is correct
17 if (myQuiz.elements[1].checked)
18 alert("Congratulations, your answer is correct");
19 else // if the answer is incorrect
20 alert("Your answer is incorrect. Please try again");
21 } // end function checkAnswers
22 -->
23 </script>
24 </head>
25 <body>
26 <form id = "myQuiz" onsubmit = "checkAnswers()" action = "">
27 <p>Select the name of the tip that goes with the
28 image shown:

29
30

31
32 <input type = "radio" name = "radiobutton" value = "CPE" />
33 <label>Common Programming Error</label>
34
35 <input type = "radio" name = "radiobutton" value = "EPT" />
36 <label>Error-Prevention Tip</label>
37
38 <input type = "radio" name = "radiobutton" value = "PERF" />
39 <label>Performance Tip</label>
40
41 <input type = "radio" name = "radiobutton" value = "PORT" />
42 <label>Portability Tip</label>

43
44 <input type = "submit" name = "submit" value = "Submit" />
45 <input type = "reset" name = "reset" value = "Reset" />
46 </p>
47 </form>
48 </body>
49 </html>
```

**Fig. 10.14** | Online quiz graded with JavaScript. (Part I of 2.)



**Fig. 10.14** | Online quiz graded with JavaScript. (Part 2 of 2.)

contains an array of all the form's controls. The radio buttons are part of the XHTML form `myQuiz`, so we access the `elements` array in line 17 using dot notation (`myQuiz.elements[ 1 ]`). The array element `myQuiz.elements[ 1 ]` corresponds to the correct answer (i.e., the second radio button). Finally, line 17 determines whether the property `checked` of the second radio button is `true`. Property `checked` of a radio button is `true` when the radio button is selected, and it is `false` when the radio button is not selected. Recall that only one radio button may be selected at any given time. If property `myQuiz.elements[ 1 ].checked` is `true`, indicating that the correct answer is selected, the script alerts a congratulatory message. If property `checked` of the radio button is `false`, then the script alerts an alternate message (line 20).

## 10.12 Wrap-Up

In this chapter, we discussed how to store related data items into an array structure. After demonstrating the declaration and initialization of an array, we described how to access and manipulate individual elements of the array using subscripts. We then introduced the concept of an `Array` object, and learned that objects are passed to a method by reference instead of by value. We explored the sorting of arrays, and described two different methods of searching them. Finally, we discussed multidimensional arrays and how to use them to organize arrays in other arrays.

## 10.13 Web Resources

[www.deitel.com/javascript/](http://www.deitel.com/javascript/)

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

---

## Summary

### *Section 10.1 Introduction*

- Arrays 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 are 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. 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 subscript or index) of the element in square brackets (`[]`).
- The first element in every array is the zeroth element. 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.
- A subscripted array name is a left-hand-side expression—it 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 operation to assign its value to another left-hand-side expression.
- Every array in JavaScript knows its own length, which it stores in its `length` attribute.

### *Section 10.3 Declaring and Allocating Arrays*

- JavaScript arrays are represented by `Array` objects.
- The process of creating new objects using the `new` operator is known as creating an instance or instantiating an object, and operator `new` is known as the dynamic memory allocation operator.

### *Section 10.4 Examples Using Arrays*

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

- JavaScript automatically reallocates an Array when a value is assigned to an element that is 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 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 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 the previous chapter 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.
- 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 direct 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.
- Arrays are objects in JavaScript, so `Arrays` are passed to a function by reference—a called function can access the elements of the caller's original `Arrays`. 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. To pass an array element to a function, use the subscripted name of the element as an argument in the function call.
- The `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.

### Section 10.8 Sorting Arrays

- 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.
- 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) 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: Linear Search and Binary Search**

- The linear search algorithm iterates through the elements of an array until it finds an element that matches a search key.
- If the array being searched is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array.
- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the array be sorted.
- The binary search algorithm tests the middle element in the array and returns the index if it matches the search key. If not, it cuts the list in half, depending on whether the key is greater than or less than the middle element, and repeats the process on the remaining half of the sorted list. The algorithm ends by either finding an element that matches the search key or reducing the subarray to zero size.
- When searching a sorted array, the binary search provides a tremendous increase in performance over the linear search. For a one-billion-element array, this is the difference between an average of 500 million comparisons and a maximum of 30 comparisons.
- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

### **Section 10.10 Multidimensional Arrays**

- To identify a particular two-dimensional multidimensional array element, we must specify the two subscripts; 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.
- Every element in a two-dimensional array is accessed using an element name of the form `a[ i ][ j ]`; `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify the row and column, respectively, of each element in `a`.
- Multidimensional arrays are maintained as arrays of arrays.

### **Section 10.11 Building an Online Quiz**

- An XHTML form's elements can be accessed individually using `getElementById` or through the `elements` property of the containing `form` object. The `elements` property contains an array of all the `form`'s controls.
- Property `checked` of a radio button is `true` when the radio button is selected, and it is `false` when the radio button is not selected.

## **Terminology**

|                      |                        |
|----------------------|------------------------|
| <code>a[i]</code>    | array initializer list |
| <code>a[i][j]</code> | Array object           |
| address in memory    | binary search          |
| array data structure | bounds of an array     |

|                                                          |                                                           |
|----------------------------------------------------------|-----------------------------------------------------------|
| checked property of a radio button                       | name of an array                                          |
| collections of data items                                | new operator                                              |
| column subscript                                         | off-by-one error                                          |
| comma-separated initializer list                         | one-dimensional array                                     |
| comparator function                                      | pass-by-reference                                         |
| creating an instance                                     | pass-by-value                                             |
| data structure                                           | passing arrays to functions                               |
| declare an array                                         | place holder in an initializer list (,)                   |
| dynamic memory allocation operator ( <code>new</code> )  | position number of an element                             |
| element of an array                                      | reserve a space in an <code>Array</code>                  |
| <code>elements</code> property of a form object          | row subscript                                             |
| <code>for...in</code> repetition statement               | scalar quantities                                         |
| index of an element                                      | separator                                                 |
| initialize an array                                      | search key                                                |
| initializer                                              | searching an array                                        |
| initializer list                                         | <code>sort</code> method of the <code>Array</code> object |
| instantiating an object                                  | sorting an array                                          |
| iterating over an array's elements                       | square brackets []                                        |
| <code>join</code> method of an <code>Array</code> object | subscript                                                 |
| left-hand-side expression                                | table of values                                           |
| <code>length</code> of an <code>Array</code> object      | tabular format                                            |
| linear search of an array                                | traverse an array                                         |
| location in an array                                     | two-dimensional array                                     |
| <i>m</i> -by- <i>n</i> array                             | value of an element                                       |
| multidimensional array                                   | zeroth element                                            |

## 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 elements of an array are related by the fact that they normally have the same \_\_\_\_\_.
- c) The number used to refer to a particular element of an array is called its \_\_\_\_\_.
- d) The process of putting the elements of an array in order is called \_\_\_\_\_ the array.
- e) Determining whether an array contains a certain key value is called \_\_\_\_\_ the array.
- f) An array that uses two subscripts 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 subscript should normally be a floating-point value.
- c) An individual array element that is 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 the fourth element 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 seventh 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:

- Declare and create the array with three rows and three columns.
- Display the number of elements.
- Use a `for...in` statement to initialize each element of the array to the sum of its subscripts. 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.

- ```
var b = new Array( 10 );
for ( var i = 0; i <= b.length; ++i )
    b[ i ] = 1;
```
- ```
var a = [[1, 2], [3, 4]];
a[1, 1] = 5;
```

## Answers to Self-Review Exercises

**10.1** a) arrays. b) type. c) subscript. d) sorting. e) searching. f) two-dimensional.

**10.2** a) True. b) False. An array subscript 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, 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) `document.write( "total: " + ( 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 error is actually a logic error, not a syntax error.] Correction: Change the `<=` operator to `<`. b) Error: The array subscripting 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:

- JavaScript stores lists of values in \_\_\_\_\_.
- The names of the four elements of array `p` are \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.
- In a two-dimensional array, the first subscript identifies the \_\_\_\_\_ of an element, and the second subscript identifies the \_\_\_\_\_ of an element.
- An  $m$ -by- $n$  array contains \_\_\_\_\_ rows, \_\_\_\_\_ columns and \_\_\_\_\_ elements.
- The name of the element in row 3 and column 5 of array `d` is \_\_\_\_\_.

f) The name of the element in the third row and fifth column 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 the second row of t.

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

g) Write a single statement that sets the elements 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 structure.

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 statement that displays the elements of the first row of t.

l) Write a statement that totals the elements of the fourth column of t.

m) Write a series of statements that prints the array t in neat, tabular format. List the column subscripts as headings across the top, and list the row subscripts 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 XHTML 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 20 numbers, each of which is between 10 and 100. As each number is read, print it only if it is not a duplicate of a number that has already been read. Provide for the “worst case,” in which all 20 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 are 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 10.15 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 numbers of times each possible sum appears. Display the results in an XHTML 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).]

**10.15** Write a script that runs 1000 games of craps and answers the following questions:

- a) How many games are won on the first roll, second roll, ..., twentieth roll and after the twentieth roll?
- b) How many games are lost on the first roll, second roll, ..., twentieth roll and after the twentieth roll?
- c) What are the chances of winning at craps? [Note: You should discover that craps is one of the fairest casino games. What do you suppose this means?]
- d) What is the average length of a game of craps?
- e) Do the chances of winning improve with the length of the game?

**10.16** (*Airline Reservations System*) A small airline has just purchased a computer for its new automated reservations system. You have been asked to program the new system. You are to write a program to assign seats on each flight of the airline’s only plane (capacity: 10 seats).

|   | 1 | 2 | 3 | 4  | 5  | 6  |
|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5  | 6  | 7  |
| 2 | 3 | 4 | 5 | 6  | 7  | 8  |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  |
| 4 | 5 | 6 | 7 | 8  | 9  | 10 |
| 5 | 6 | 7 | 8 | 9  | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Fig. 10.15** | Thirty-six possible outcomes of rolling two dice.

Your program should display the following menu of alternatives: Please type 1 for "First Class" and Please type 2 for "Economy". If the person types 1, your program should assign a seat in the first-class section (seats 1–5). If the person types 2, your program should assign a seat in the economy section (seats 6–10). Your program should print a boarding pass indicating the person's seat number and whether it is in the first-class or economy section of the plane.

Use a one-dimensional array to represent the seating chart of the plane. Initialize all the elements of the array to 0 to indicate that all the seats are empty. As each seat is assigned, set the corresponding elements of the array to 1 to indicate that the seat is no longer available.

Your program should, of course, never assign a seat that has already been assigned. When the first-class section is full, your program should ask the person if it is acceptable to be placed in the economy section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message "Next flight leaves in 3 hours."

**10.17** Use a two-dimensional array to solve the following problem: A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product actually sold. Each slip contains

- a) the salesperson number,
- b) the product number, and
- c) the total dollar value of the product sold that day.

Thus, each salesperson passes in between zero and five sales slips per day. Assume that the information from all of the slips for last month is available. Write a script that will read all this information for last month's sales and summarize the total sales by salesperson by product. All totals should be stored in the two-dimensional array `sales`. After processing all the information for last month, display the results in an XHTML table format, with each of the columns representing a different salesperson and each of the rows representing a different product. Cross-total each row to get the total sales of each product for last month; cross-total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross-totals to the right of the totaled rows and to the bottom of the totaled columns.

**10.18** (*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 will simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 20-by-20 array `floor` that is 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. 10.16.

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:

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

| Command | Meaning                                            |
|---------|----------------------------------------------------|
| 1       | Pen up                                             |
| 2       | Pen down                                           |
| 3       | Turn right                                         |
| 4       | Turn left                                          |
| 5,10    | Move forward 10 spaces (or a number other than 10) |
| 6       | Print the 20-by-20 array                           |
| 9       | End of data (sentinel)                             |

**Fig. 10.16** | Turtle graphics commands.

As the turtle moves with the pen down, set the appropriate elements of array `floor` to 1s. When the 6 command (print) is given, display an asterisk or some other character of your choosing wherever there is a 1 in the array. Wherever there is 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.

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

- Create an array with all elements initialized to 1 (true). Array elements with prime subscripts will remain as 1. All other array elements will eventually be set to zero.
- Set the first two elements to zero, since 0 and 1 are not prime. Starting with array subscript 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 subscript is a multiple of the subscript for the element with value 1. For array subscript 2, all elements beyond 2 in the array that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements beyond 3 in the array that are multiples of 3 will be set to zero (subscripts 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 subscript is a prime number. These subscripts 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.

**10.20** (*Simulation: The Tortoise and the Hare*) In this problem, you will re-create one of the truly great moments in history, namely the classic race of the tortoise and the hare. You will 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 is 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. 10.17.

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.

| Animal   | Move type  | Percentage of the time | Actual move            |
|----------|------------|------------------------|------------------------|
| Tortoise | Fast plod  | 50%                    | 3 squares to the right |
|          | Slip       | 20%                    | 6 squares to the left  |
|          | Slow plod  | 30%                    | 1 square to the right  |
| Hare     | Sleep      | 20%                    | No move at all         |
|          | Big hop    | 20%                    | 9 squares to the right |
|          | Big slip   | 10%                    | 12 squares to the left |
|          | Small hop  | 30%                    | 1 square to the right  |
|          | Small slip | 20%                    | 2 squares to the left  |

**Fig. 10.17** | Rules for adjusting the position of the tortoise and the hare.

Generate the percentages in Fig. 10.17 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.

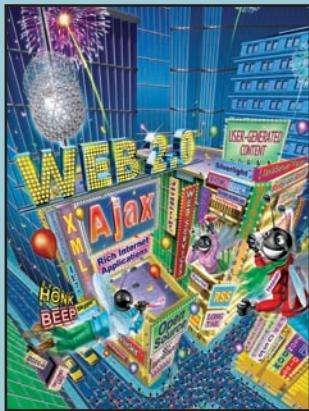
Begin the race by printing

```
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 are 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.



*My object all sublime  
I shall achieve in time.*

—W. S. Gilbert

*Is it a world to hide virtues  
in?*

—William Shakespeare

*Good as it is to inherit a  
library, it is better to collect  
one.*

—Augustine Birrell

*A philosopher of imposing  
stature doesn't think in a  
vacuum. Even his most  
abstract ideas are, to some  
extent, conditioned by what  
is or is not known in the time  
when he lives.*

—Alfred North Whitehead

# JavaScript: Objects

## OBJECTIVES

In this chapter you will learn:

- Object-based programming terminology and concepts.
- The concepts of encapsulation and data hiding.
- The value of object orientation.
- To use the JavaScript objects `Math`, `String`, `Date`, `Boolean` and `Number`.
- To use the browser's `document` and `window` objects.
- To use cookies.
- To represent objects simply using JSON.

## Outline

- 11.1 Introduction
- 11.2 Introduction to Object Technology
- 11.3 Math Object
- 11.4 String Object
  - 11.4.1 Fundamentals of Characters and Strings
  - 11.4.2 Methods of the String Object
  - 11.4.3 Character-Processing Methods
  - 11.4.4 Searching Methods
  - 11.4.5 Splitting Strings and Obtaining Substrings
  - 11.4.6 XHTML Markup Methods
- 11.5 Date Object
- 11.6 Boolean and Number Objects
- 11.7 document Object
- 11.8 window Object
- 11.9 Using Cookies
- 11.10 Final JavaScript Example
- 11.11 Using JSON to Represent Objects
- 11.12 Wrap-Up
- 11.13 Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercise](#) | [Answers to Self-Review Exercise](#) | [Exercises](#)  
[Special Section: Challenging String Manipulation Exercises](#)

## 11.1 Introduction

Most of the JavaScript programs we've demonstrated illustrate basic programming concepts. These programs provide you with the foundation you need to build powerful and complex scripts as part of your web pages. As you proceed beyond this chapter, you will use JavaScript to manipulate every element of an XHTML document from a script.

This chapter presents a more formal treatment of **objects**. We begin by giving a brief introduction to the concepts behind object-orientation. The remainder of the chapter overviews—and serves as a reference for—several of JavaScript's built-in objects and demonstrates many of their capabilities. We also provide a brief introduction to JSON, a means for creating JavaScript objects. In the chapters on the Document Object Model and Events that follow this chapter, you will be introduced to many objects provided by the browser that enable scripts to interact with the elements of an XHTML document.

## 11.2 Introduction to Object Technology

This section provides a general introduction to object orientation. The terminology and technologies discussed here support various chapters that come later in the book. Here, you'll learn that objects are a natural way of thinking about the world and about scripts that manipulate XHTML documents. In Chapters 6–10, we used built-in JavaScript objects—Math and Array—and objects provided by the web browser—document and win-

dow—to perform tasks in our scripts. JavaScript uses objects to perform many tasks and therefore is referred to as an **object-based programming language**. As we have seen, JavaScript also uses constructs from the “conventional” structured programming methodology supported by many other programming languages. The first five JavaScript chapters concentrated on these conventional parts of JavaScript because they are important components of all JavaScript programs. Our goal here is to help you develop an object-oriented way of thinking. Many concepts in this book, including CSS, JavaScript, Ajax, Ruby on Rails, ASP.NET, and JavaServer Faces are based on at least some of the concepts introduced in this section.

### ***Basic Object-Technology Concepts***

We begin our introduction to object technology with some key terminology. Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers, monitors and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects we see around us every day.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are “alive” in some sense—they move around and do things. Inanimate objects do not move on their own. Objects of both types, however, have some things in common. They all have **attributes** (e.g., size, shape, color and weight), and they all exhibit **behaviors** (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We’ll study the kinds of attributes and behaviors that software objects have.

Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults, and between humans and chimpanzees.

**Object-oriented design (OOD)** models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics—cars, trucks, little red wagons and roller skates have much in common. OOD takes advantage of **inheritance** relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class “convertible” certainly has the characteristics of the more general class “automobile,” but more specifically, the roof goes up and down.

Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

OOD **encapsulates** (i.e., wraps) attributes and **operations** (behaviors) into objects—an object’s attributes and operations are intimately tied together. Objects have the property of **information hiding**. This means that objects may know how to communicate with one another across well-defined **interfaces**, but normally they are not allowed to know how

other objects are implemented—implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on. Information hiding, as we'll see, is crucial to good software engineering.

Like the designers of an automobile, the designers of web browsers have defined a set of objects that encapsulate an XHTML document's elements and expose to a JavaScript programmer the attributes and behaviors that enable a JavaScript program to interact with (or script) those elements (objects). You'll soon see that the browser's document object contains attributes and behaviors that provide access to every element of an XHTML document. Similarly, JavaScript provides objects that encapsulate various capabilities in a script. For example, the JavaScript `Array` object provides attributes and behaviors that enable a script to manipulate a collection of data. The `Array` object's `length` property (attribute) contains the number of elements in the `Array`. The `Array` object's `sort` method (behavior) orders the elements of the `Array`.

Some programming languages—like Java, Visual Basic, C# and C++—are **object oriented**. Programming in such a language is called **object-oriented programming (OOP)**, and it allows computer programmers to implement object-oriented designs as working software systems. Languages like C, on the other hand, are **procedural**, so programming tends to be **action oriented**. In procedural languages, the unit of programming is the **function**. In object-oriented languages, the unit of programming is the **class** from which objects are eventually **instantiated** (an OOP term for “created”). Classes contain functions that implement operations and data that comprises attributes.

Procedural programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs. Data is certainly important in procedural languages, but the view is that data exists primarily in support of the actions that functions perform. The **verbs** in a system specification help a procedural programmer determine the set of functions that work together to implement the system.

### *Classes, Properties and Methods*

Object-oriented programmers concentrate on creating their own **user-defined types** called **classes**. Each class contains data as well as the set of functions that manipulate that data and provide services to **clients** (i.e., other classes or functions that use the class). The data components of a class are called properties. For example, a bank account class might include an account number and a balance. The function components of a class are called methods. For example, a bank account class might include methods to make a deposit (increasing the balance), make a withdrawal (decreasing the balance) and inquire what the current balance is. You use built-in types (and other user-defined types) as the “building blocks” for constructing new user-defined types (classes). The **nouns** in a system specification help you determine the set of classes from which objects are created that work together to implement the system.

Classes are to objects as blueprints are to houses—a class is a “plan” for building an object of the class. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class. You cannot cook meals in the kitchen of a blueprint; you can cook meals in the kitchen of a house. You cannot sleep in the bedroom of a blueprint; you can sleep in the bedroom of a house.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the “bank teller” class needs to relate to other classes, such as the “customer” class, the “cash drawer” class, the “safe” class, and so on. These relationships are called **associations**.

Packaging software as classes makes it possible for future software systems to **reuse** the classes. Groups of related classes are often packaged as reusable **components**. Just as realtors often say that the three most important factors affecting the price of real estate are “location, location and location,” some people in the software development community say that the three most important factors affecting the future of software development are “reuse, reuse and reuse.”

Indeed, with object technology, you can build much of the new software you’ll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can reuse to speed and enhance the quality of future software development efforts. Now that we’ve introduced the terminology associated with object-orientation, you’ll see it used in the upcoming discussions of some of JavaScript’s objects.

## 11.3 Math Object

The **Math** object’s methods allow you to perform many common mathematical calculations. As shown previously, an object’s methods are called by writing the name of the object followed by a dot (.) and the name of the method. In parentheses following the method name is the argument (or a comma-separated list of arguments) to the method. For example, to calculate and display the square root of 900.0 you might write

```
document.writeln(Math.sqrt(900.0));
```

which calls method **Math.sqrt** to calculate the square root of the number contained in the parentheses (900.0), then outputs the result. The number 900.0 is the argument of the **Math.sqrt** method. The preceding statement would display 30.0. Some **Math** object methods are summarized in Fig. 11.1.

| Method           | Description                                      | Examples                                                                                                                 |
|------------------|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <b>abs( x )</b>  | absolute value of x                              | <code>abs( 7.2 )</code> is <b>7.2</b><br><code>abs( 0.0 )</code> is <b>0.0</b><br><code>abs( -5.6 )</code> is <b>5.6</b> |
| <b>ceil( x )</b> | rounds x to the smallest integer not less than x | <code>ceil( 9.2 )</code> is <b>10.0</b><br><code>ceil( -9.8 )</code> is <b>-9.0</b>                                      |
| <b>cos( x )</b>  | trigonometric cosine of x (x in radians)         | <code>cos( 0.0 )</code> is <b>1.0</b>                                                                                    |
| <b>exp( x )</b>  | exponential method $e^x$                         | <code>exp( 1.0 )</code> is <b>2.71828</b><br><code>exp( 2.0 )</code> is <b>7.38906</b>                                   |

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

| Method                   | Description                                        | Examples                                                                                                   |
|--------------------------|----------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>floor( x )</code>  | rounds x to the largest integer not greater than x | <code>floor( 9.2 )</code> is <code>9.0</code><br><code>floor( -9.8 )</code> is <code>-10.0</code>          |
| <code>log( x )</code>    | natural logarithm of x (base $e$ )                 | <code>log( 2.718282 )</code> is <code>1.0</code><br><code>log( 7.389056 )</code> is <code>2.0</code>       |
| <code>max( x, y )</code> | larger value of x and y                            | <code>max( 2.3, 12.7 )</code> is <code>12.7</code><br><code>max( -2.3, -12.7 )</code> is <code>-2.3</code> |
| <code>min( x, y )</code> | smaller value of x and y                           | <code>min( 2.3, 12.7 )</code> is <code>2.3</code><br><code>min( -2.3, -12.7 )</code> is <code>-12.7</code> |
| <code>pow( x, y )</code> | x raised to power y ( $x^y$ )                      | <code>pow( 2.0, 7.0 )</code> is <code>128.0</code><br><code>pow( 9.0, .5 )</code> is <code>3.0</code>      |
| <code>round( x )</code>  | rounds x to the closest integer                    | <code>round( 9.75 )</code> is <code>10</code><br><code>round( 9.25 )</code> is <code>9</code>              |
| <code>sin( x )</code>    | trigonometric sine of x (x in radians)             | <code>sin( 0.0 )</code> is <code>0.0</code>                                                                |
| <code>sqrt( x )</code>   | square root of x                                   | <code>sqrt( 900.0 )</code> is <code>30.0</code><br><code>sqrt( 9.0 )</code> is <code>3.0</code>            |
| <code>tan( x )</code>    | trigonometric tangent of x (x in radians)          | <code>tan( 0.0 )</code> is <code>0.0</code>                                                                |

**Fig. 11.1** | Math object methods. (Part 2 of 2.)**Common Programming Error 11.1**

Forgetting to invoke a Math method by preceding the method name with the object name Math and a dot (.) is an error.

**Software Engineering Observation 11.1**

The primary difference between invoking a standalone function and invoking a method of an object is that an object name and a dot are not required to call a standalone function.

The Math object defines several commonly used mathematical constants, summarized in Fig. 11.2. [Note: By convention, the names of constants are written in all uppercase letters so they stand out in a program.]

| Constant               | Description                          | Value               |
|------------------------|--------------------------------------|---------------------|
| <code>Math.E</code>    | Base of a natural logarithm ( $e$ ). | Approximately 2.718 |
| <code>Math.LN2</code>  | Natural logarithm of 2               | Approximately 0.693 |
| <code>Math.LN10</code> | Natural logarithm of 10              | Approximately 2.302 |

**Fig. 11.2** | Properties of the Math object. (Part 1 of 2.)

| Constant     | Description                                                  | Value                           |
|--------------|--------------------------------------------------------------|---------------------------------|
| Math.LOG2E   | Base 2 logarithm of $e$                                      | Approximately 1.442             |
| Math.LOG10E  | Base 10 logarithm of $e$                                     | Approximately 0.434             |
| Math.PI      | $\pi$ —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. (Part 2 of 2.)



### Good Programming Practice 11.1

*Use the mathematical constants of the Math object rather than explicitly typing the numeric value of the constant.*

## 11.4 String Object

In this section, we introduce JavaScript’s string- and character-processing capabilities. The techniques discussed here are appropriate for processing names, addresses, telephone numbers, and similar items.

### 11.4.1 Fundamentals of Characters and Strings

Characters are the fundamental 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 the set of characters called **Unicode®**, which represents a large portion of the world’s languages. (We discuss Unicode in detail in Appendix F.) A string is an object of type **String**. **String literals** or **string constants** (often called **anonymous String objects**) are written as a sequence of characters in double quotation marks or single quotation marks, as follows:

|                          |                      |
|--------------------------|----------------------|
| "John Q. Doe"            | (a name)             |
| '9999 Main Street'       | (a street address)   |
| "Waltham, Massachusetts" | (a city and state)   |
| '(201) 555-1212'         | (a telephone number) |

A **String** may be assigned to a variable in a declaration. The declaration

```
var color = "blue";
```

initializes variable **color** with the **String** object containing the string "blue". Strings can be compared via the relational (<, <=, > and >=) and equality operators (== and !=). Strings are compared using the Unicode values of the corresponding characters. For example, the expression "hello" < "Hello" evaluates to false because lowercase letters have higher Unicode values.

### 11.4.2 Methods of the String Object

The `String` object encapsulates the attributes and behaviors of a string of characters. It provides many methods (behaviors) that accomplish useful tasks such as selecting characters from a string, combining strings (called **concatenation**), obtaining substrings of a string, searching for substrings within a string, tokenizing strings (i.e., splitting strings into individual words) and converting strings to all uppercase or lowercase letters. The `String` object also provides several methods that generate XHTML tags. Figure 11.3 summarizes many `String` methods. Figures 11.4–11.7 demonstrate some of these methods.

| Method                                              | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>charAt( index )</code>                        | Returns a string containing the character at the specified <i>index</i> . If there is no character at the <i>index</i> , <code>charAt</code> returns an empty string. The first character is located at <i>index</i> 0.                                                                                                                                                                                                                                           |
| <code>charCodeAt( index )</code>                    | Returns the Unicode value of the character at the specified <i>index</i> , or <code>NaN</code> (not a number) if there is no character at that <i>index</i> .                                                                                                                                                                                                                                                                                                     |
| <code>concat( string )</code>                       | Concatenates its argument to the end of the string that invokes the method. The string invoking this method is not modified; instead a new <code>String</code> is returned. This method is the same as adding two strings with the string-concatenation operator <code>+</code> (e.g., <code>s1.concat(s2)</code> is the same as <code>s1 + s2</code> ).                                                                                                          |
| <code>fromCharCode( value1, value2, ... )</code>    | Converts a list of Unicode values into a string containing the corresponding characters.                                                                                                                                                                                                                                                                                                                                                                          |
| <code>indexOf( substring, index )</code>            | Searches for the first occurrence of <i>substring</i> starting from position <i>index</i> in the string that invokes the method. The method returns the starting index of <i>substring</i> in the source string or <code>-1</code> if <i>substring</i> is not found. If the <i>index</i> argument is not provided, the method begins searching from index 0 in the source string.                                                                                 |
| <code>lastIndexOf( substring, index )</code>        | Searches for the last occurrence of <i>substring</i> starting from position <i>index</i> and searching toward the beginning of the string that invokes the method. The method returns the starting index of <i>substring</i> in the source string or <code>-1</code> if <i>substring</i> is not found. If the <i>index</i> argument is not provided, the method begins searching from the end of the source string.                                               |
| <code>replace( searchString, replaceString )</code> | Searches for the substring <i>searchString</i> , and replaces the first occurrence with <i>replaceString</i> and returns the modified string, or the original string if no replacement was made.                                                                                                                                                                                                                                                                  |
| <code>slice( start, end )</code>                    | Returns a string containing the portion of the string from index <i>start</i> through index <i>end</i> . If the <i>end</i> index is not specified, the method returns a string from the <i>start</i> index to the end of the source string. A negative <i>end</i> index specifies an offset from the end of the string, starting from a position one past the end of the last character (so <code>-1</code> indicates the last character position in the string). |

**Fig. 11.3** | Some `String` object methods. (Part 1 of 2.)

| Method                                  | Description                                                                                                                                                                                                                                  |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>split( string )</code>            | Splits the source string into an array of strings (tokens), where its <i>string</i> argument specifies the delimiter (i.e., the characters that indicate the end of each token in the source string).                                        |
| <code>substr( start, length )</code>    | Returns a string containing <i>length</i> characters starting from index <i>start</i> in the source string. If <i>length</i> is not specified, a string containing characters from <i>start</i> to the end of the source string is returned. |
| <code>substring( start, end )</code>    | Returns a string containing the characters from index <i>start</i> up to but not including index <i>end</i> in the source string.                                                                                                            |
| <code>toLowerCase()</code>              | Returns a string in which all uppercase letters are converted to lowercase letters. Nonletter characters are not changed.                                                                                                                    |
| <code>toUpperCase()</code>              | Returns a string in which all lowercase letters are converted to uppercase letters. Nonletter characters are not changed.                                                                                                                    |
| <i>Methods that generate XHTML tags</i> |                                                                                                                                                                                                                                              |
| <code>anchor( name )</code>             | Wraps the source string in an anchor element ( <code>&lt;a&gt;&lt;/a&gt;</code> ) with <i>name</i> as the anchor name.                                                                                                                       |
| <code>fixed()</code>                    | Wraps the source string in a <code>&lt;tt&gt;&lt;/tt&gt;</code> element (same as <code>&lt;pre&gt;&lt;/pre&gt;</code> ).                                                                                                                     |
| <code>link( url )</code>                | Wraps the source string in an anchor element ( <code>&lt;a&gt;&lt;/a&gt;</code> ) with <i>url</i> as the hyperlink location.                                                                                                                 |
| <code>strike()</code>                   | Wraps the source string in a <code>&lt;strike&gt;&lt;/strike&gt;</code> element.                                                                                                                                                             |
| <code>sub()</code>                      | Wraps the source string in a <code>&lt;sub&gt;&lt;/sub&gt;</code> element.                                                                                                                                                                   |
| <code>sup()</code>                      | Wraps the source string in a <code>&lt;sup&gt;&lt;/sup&gt;</code> element.                                                                                                                                                                   |

**Fig. 11.3** | Some String object methods. (Part 2 of 2.)

### 11.4.3 Character-Processing Methods

The script in Fig. 11.4 demonstrates some of the String object's character-processing methods, including `charAt` (returns the character at a specific position), `charCodeAt` (returns the Unicode value of the character at a specific position), `fromCharCode` (returns a string created from a series of Unicode values), `toLowerCase` (returns the lowercase version of a string) and `toUpperCase` (returns the uppercase version of a string).

```

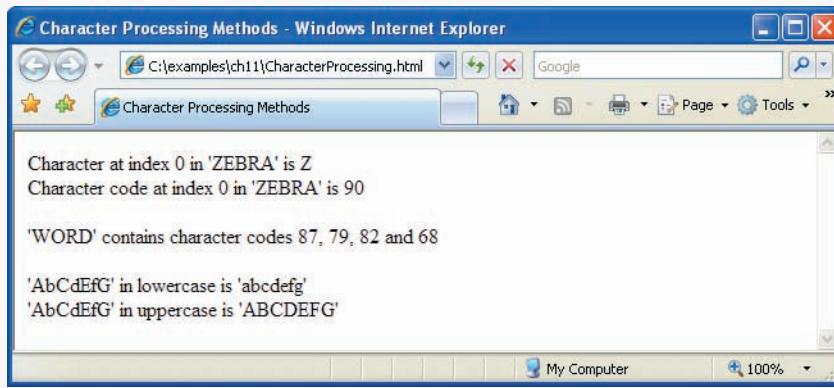
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.4: CharacterProcessing.html -->
6 <!-- String methods charAt, charCodeAt, fromCharCode, toLowercase and
7 toUpperCase. -->
```

**Fig. 11.4** | String methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowercase` and `toUpperCase`. (Part 1 of 2.)

```

8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Character Processing Methods</title>
11 <script type = "text/javascript">
12 <!--
13 var s = "ZEBRA";
14 var s2 = "AbCdEfG";
15
16 document.writeln("<p>Character at index 0 in '" +
17 s + "' is " + s.charAt(0));
18 document.writeln("
Character code at index 0 in '" +
19 + s + "' is " + s.charCodeAt(0) + "</p>");
20
21 document.writeln("<p>'"
22 String.fromCharCode(87, 79, 82, 68) +
23 "' contains character codes 87, 79, 82 and 68</p>")
24
25 document.writeln("<p>'"
26 s2 + "' in lowercase is " +
27 s2.toLowerCase() + "'");
28 document.writeln("
'"
29 + s2 + "' in uppercase is " +
30 + s2.toUpperCase() + "'</p>");
31 // -->
32 </script>
33 </head><body></body>
34 </html>

```



**Fig. 11.4** | String methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowerCase` and `toUpperCase`. (Part 2 of 2.)

Lines 16–17 display the first character in String `s` ("ZEBRA") using String method `charAt`. Method `charAt` returns a string containing the character at the specified index (0 in this example). 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` (i.e., if the string contains five characters, the indices are 0 through 4). If the index is outside the bounds of the string, the method returns an empty string.

Lines 18–19 display the character code for the first character in String `s` ("ZEBRA") by calling String method `charCodeAt`. Method `charCodeAt` returns the Unicode value of

the character at the specified index (0 in this example). If the index is outside the bounds of the string, the method returns NaN.

`String` method `fromCharCode` receives as its argument a comma-separated list of Unicode values and builds a string containing the character representation of those Unicode values. Lines 21–23 display the string "WORD", which consists of the character codes 87, 79, 82 and 68. Note that the `String` object calls method `fromCharCode`, rather than a specific `String` variable. Appendix D, ASCII Character Set, contains the character codes for the ASCII character set—a subset of the Unicode character set (Appendix F) that contains only Western characters.

The statements in lines 25–26 and 27–28 use `String` methods `toLowerCase` and `toUpperCase` to display versions of `String` `s2` ("AbCdEfG") in all lowercase letters and all uppercase letters, respectively.

#### 11.4.4 Searching Methods

Being able to search for a character or a sequence of characters in a string is often useful. For example, if you are creating your own word processor, you may want to provide a capability for searching through the document. The script in Fig. 11.5 demonstrates the `String` object methods `indexOf` and `lastIndexOf` that search for a specified substring in a string. All the searches in this example are performed on the global string `letters` (initialized in line 14 with "abcdefghijklmнопqrstuvwxyzабцдѓхјкљм" in the script).

The user types a substring in the XHTML form `searchForm`'s `inputVal` text field and presses the `Search` button to search for the substring in `letters`. Clicking the `Search` button calls function `buttonPressed` (defined in lines 16–29) to respond to the `onclick` event and perform the searches. The results of each search are displayed in the appropriate text field of `searchForm`.

Lines 21–22 use `String` method `indexOf` to determine the location of the first occurrence in string `letters` of the string `inputVal.value` (i.e., the string the user typed in the

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.5: SearchingStrings.html -->
6 <!-- String searching with indexOf and lastIndexOf. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>
10 Searching Strings with indexOf and lastIndexOf
11 </title>
12 <script type = "text/javascript">
13 <!--
14 var letters = "абцдѓхјкљмнопqrstuvwxyzабцдѓхјкљм";
15
16 function buttonPressed()
17 {
18 var searchForm = document.getElementById("searchForm");
19 var inputVal = document.getElementById("inputVal");

```

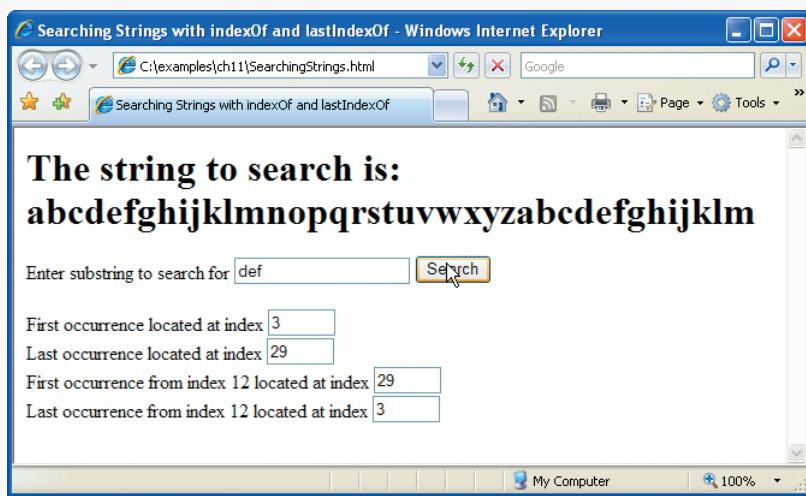
Fig. 11.5 | String searching with `indexOf` and `lastIndexOf`. (Part 1 of 3.)

```

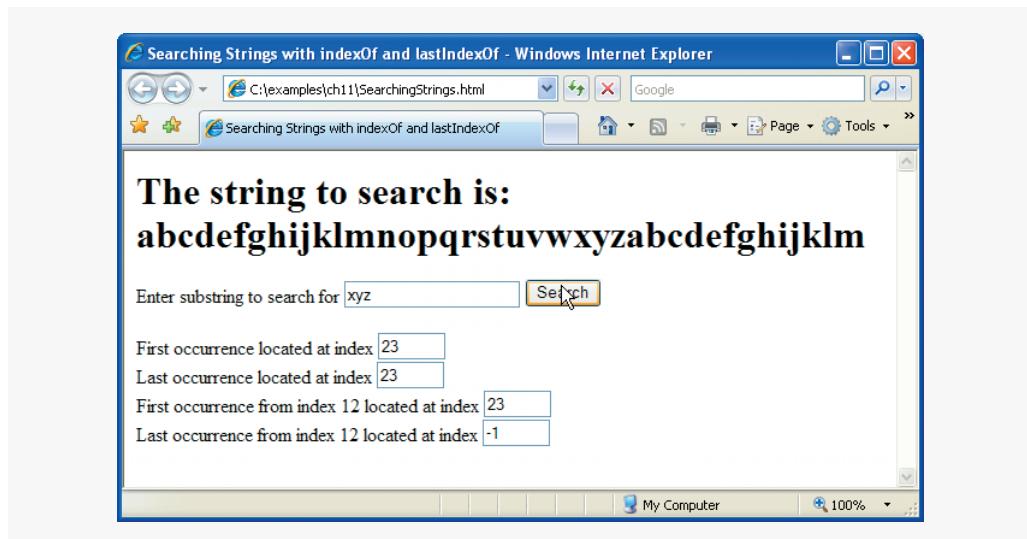
20 searchForm.elements[2].value =
21 letters.indexOf(inputVal.value);
22 searchForm.elements[3].value =
23 letters.lastIndexOf(inputVal.value);
24 searchForm.elements[4].value =
25 letters.indexOf(inputVal.value, 12);
26 searchForm.elements[5].value =
27 letters.lastIndexOf(inputVal.value, 12);
28 } // end function buttonPressed
29 // -->
30 </script>
31 </head>
32 <body>
33 <form id = "searchForm" action = "">
34 <h1>The string to search is:

35 abcdefghijklmnopqrstuvwxyzabcdefghijklm</h1>
36 <p>Enter substring to search for
37 <input id = "inputVal" type = "text" />
38 <input id = "search" type = "button" value = "Search"
39 onclick = "buttonPressed()" />
</p>
40
41 <p>First occurrence located at index
42 <input id = "first" type = "text" size = "5" />
43
Last occurrence located at index
44 <input id = "last" type = "text" size = "5" />
45
First occurrence from index 12 located at index
46 <input id = "first12" type = "text" size = "5" />
47
Last occurrence from index 12 located at index
48 <input id = "last12" type = "text" size = "5" /></p>
49
50 </form>
51 </body>
52 </html>

```



**Fig. 11.5** | String searching with `indexOf` and `lastIndexOf`. (Part 2 of 3.)



**Fig. 11.5 |** String searching with `indexOf` and `lastIndexOf`. (Part 3 of 3.)

`inputVal` text field). If the substring is found, the index at which the first occurrence of the substring begins is returned; otherwise, `-1` is returned.

Lines 23–24 use `String` method `lastIndexOf` to determine the location of the last occurrence in `letters` of the string in `inputVal`. If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, `-1` is returned.

Lines 25–26 use `String` method `indexOf` to determine the location of the first occurrence in string `letters` of the string in the `inputVal` text field, starting from index 12 in `letters`. If the substring is found, the index at which the first occurrence of the substring (starting from index 12) begins is returned; otherwise, `-1` is returned.

Lines 27–28 use `String` method `lastIndexOf` to determine the location of the last occurrence in `letters` of the string in the `inputVal` text field, starting from index 12 in `letters` and moving toward the beginning of the input. If the substring is found, the index at which the first occurrence of the substring (if one appears before index 12) begins is returned; otherwise, `-1` is returned.



### Software Engineering Observation 11.2

*String methods `indexOf` and `lastIndexOf`, with their optional second argument (the starting index from which to search), are particularly useful for continuing a search through a large amount of text.*

#### 11.4.5 Splitting Strings and Obtaining Substrings

When you read a sentence, your mind breaks it into individual words, or **tokens**, each of which conveys meaning to you. The process of breaking a string into tokens is called **tokenization**. Interpreters also perform tokenization. They break up statements into such individual pieces as keywords, identifiers, operators and other elements of a programming language. Figure 11.6 demonstrates `String` method `split`, which breaks a string into its component tokens. Tokens are separated from one another by **delimiters**, typically white-

space characters such as blanks, tabs, newlines and carriage returns. Other characters may also be used as delimiters to separate tokens. The XHTML document displays a form containing a text field where the user types a sentence to tokenize. The results of the tokenization process are displayed in an XHTML `textarea` GUI component. The script also demonstrates `String` method `substring`, which returns a portion of a string.

The user types a sentence into the text field with id `inputVal` text field and presses the `Split` button to tokenize the string. Function `splitButtonPressed` (lines 12–21) handles the button's `onclick` event.

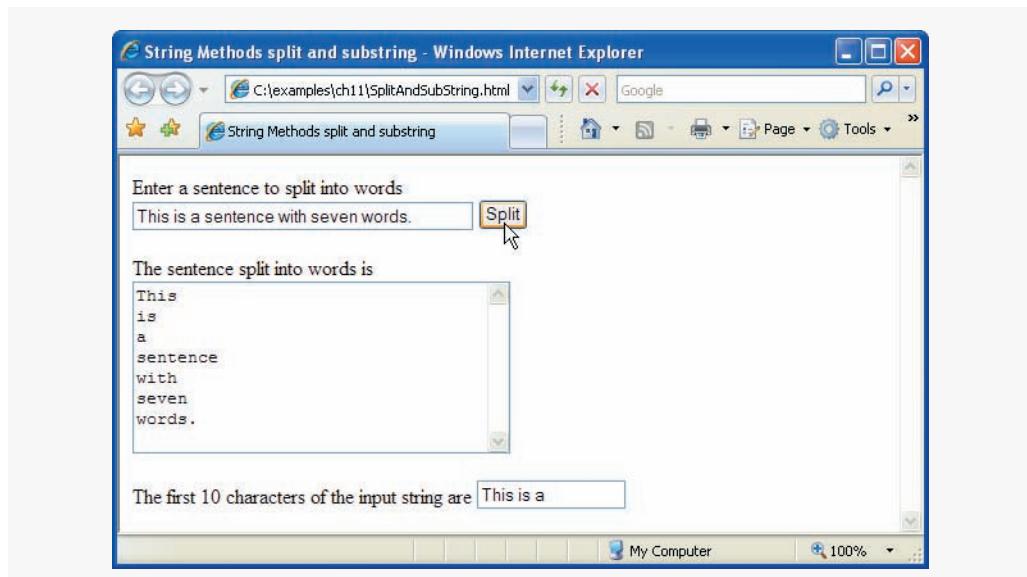
```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.6: SplitAndSubString.html -->
6 <!-- String object methods split and substring. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>String Methods split and substring</title>
10 <script type = "text/javascript">
11 <!--
12 function splitButtonPressed()
13 {
14 var inputString = document.getElementById("inputVal").value;
15 var tokens = inputString.split(" ");
16 document.getElementById("output").value =
17 tokens.join("\n");
18
19 document.getElementById("outputSubstring").value =
20 inputString.substring(0, 10);
21 } // end function splitButtonPressed
22 // --
23 </script>
24 </head>
25 <body>
26 <form action = "">
27 <p>Enter a sentence to split into words

28 <input id = "inputVal" type = "text" size = "40" />
29 <input type = "button" value = "Split"
30 onclick = "splitButtonPressed()" /></p>
31
32 <p>The sentence split into words is

33 <textarea id = "output" rows = "8" cols = "34">
34 </textarea></p>
35
36 <p>The first 10 characters of the input string are
37 <input id = "outputSubstring" type = "text"
38 size = "15" /></p>
39 </form>
40 </body>
41 </html>
```

**Fig. 11.6 |** String object methods `split` and `substring`. (Part I of 2.)



**Fig. 11.6 |** String object methods `split` and `substring`. (Part 2 of 2.)

Line 14 gets the value of the input field and stores it in variable `inputString`. Line 15 calls `String` method `split` to tokenize `inputString`. The argument to method `split` is the **delimiter string**—the string that determines the end of each token in the original string. In this example, the space character delimits the tokens. The delimiter string can contain multiple characters that should be used as delimiters. Method `split` returns an array of strings containing the tokens. Line 17 uses `Array` method `join` to combine the tokens in array `tokens` and separate each token with a newline character (`\n`). The resulting string is assigned to the `value` property of the XHTML form's `output` GUI component (an XHTML `textarea`).

Lines 19–20 use `String` method `substring` to obtain a string containing the first 10 characters of the string the user entered (still stored in `inputString`). 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.

#### 11.4.6 XHTML Markup Methods

The script in Fig. 11.7 demonstrates the `String` object's methods that generate XHTML markup tags. When a `String` object invokes a markup method, the method wraps the `String`'s contents in the appropriate XHTML tag. These methods are particularly useful for generating XHTML dynamically during script processing.

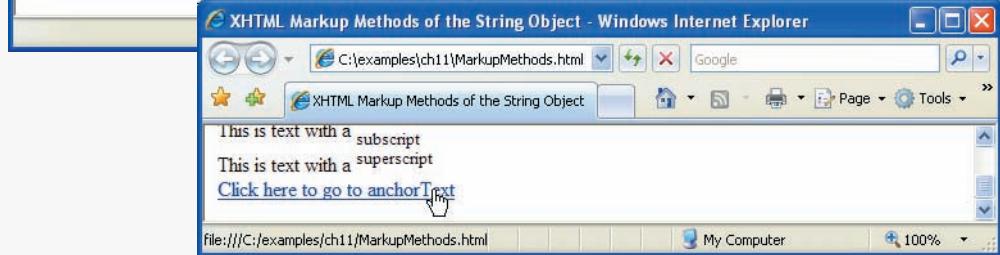
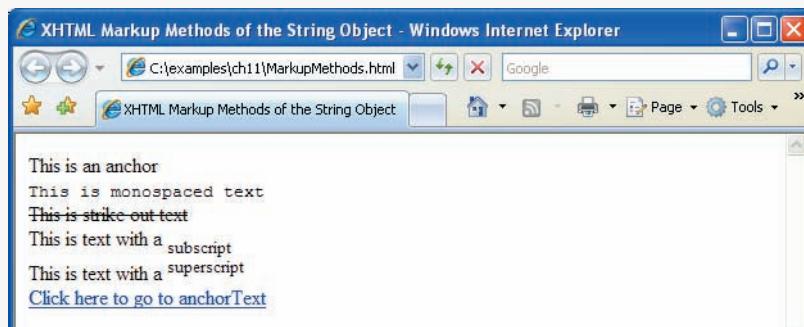
Lines 12–17 define the strings that call each of the XHTML markup methods of the `String` object. Line 19 uses `String` method `anchor` to format the string in variable `anchorText` ("This is an anchor") as

```
This is an anchor
```

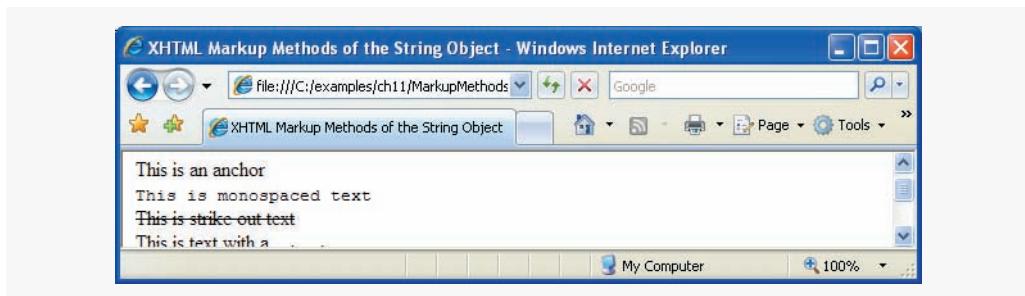
```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.7: MarkupMethods.html -->
6 <!-- String object XHTML markup methods. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>XHTML Markup Methods of the String Object</title>
10 <script type = "text/javascript">
11 <!--
12 var anchorText = "This is an anchor";
13 var fixedText = "This is monospaced text";
14 var linkText = "Click here to go to anchorText";
15 var strikeText = "This is strike out text";
16 var subText = "subscript";
17 var supText = "superscript";
18
19 document.writeln(anchorText.anchor("top"));
20 document.writeln("
" + fixedText.fixed());
21 document.writeln("
" + strikeText.strike());
22 document.writeln(
23 "
This is text with a " + subText.sub());
24 document.writeln(
25 "
This is text with a " + supText.sup());
26 document.writeln("
" + linkText.link("#top"));
27 // --
28 </script>
29 </head><body></body>
30 </html>

```



**Fig. 11.7** | String object XHTML markup methods. (Part 1 of 2.)



**Fig. 11.7** | String object XHTML markup methods. (Part 2 of 2.)

The name of the anchor is the argument to the method. This anchor will be used later in the example as the target of a hyperlink.

Line 20 uses `String` method `fixed` to display text in a fixed-width font by formatting the string in variable `fixedText` ("This is monospaced text") as

```
<tt>This is monospaced text</tt>
```

Line 21 uses `String` method `strike` to display text with a line through it by formatting the string in variable `strikeText` ("This is strike out text") as

```
<strike>This is strike out text</strike>
```

Lines 22–23 use `String` method `sub` to display subscript text by formatting the string in variable `subText` ("subscript") as

```
_{subscript}
```

Note that the resulting line in the XHTML document displays the word `subscript` smaller than the rest of the line and slightly below the line.

Lines 24–25 call `String` method `sup` to display superscript text by formatting the string in variable `supText` ("superscript") as

```
^{superscript}
```

Note that the resulting line in the XHTML document displays the word `superscript` smaller than the rest of the line and slightly above the line.

Line 26 uses `String` method `link` to create a hyperlink by formatting the string in variable `linkText` ("Click here to go to anchorText") as

```
Click here to go to anchorText
```

The target of the hyperlink (#top in this example) is the argument to the method and can be any URL. In this example, the hyperlink target is the anchor created in line 19. If you make your browser window short and scroll to the bottom of the web page, then click this link, the browser will reposition to the top of the web page.

## 11.5 Date Object

JavaScript's `Date` object provides methods for date and time manipulations. Date and time processing can be performed based on the computer's `local time zone` or based on World Time Standard's `Coordinated Universal Time` (abbreviated `UTC`)—formerly called

**Greenwich Mean Time (GMT).** Most methods of the Date object have a local time zone and a UTC version. The methods of the Date object are summarized in Fig. 11.8.

| Method                                                                          | Description                                                                                                                                                                                                                           |
|---------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getDate()</code><br><code>getUTCDate()</code>                             | Returns a number from 1 to 31 representing the day of the month in local time or UTC.                                                                                                                                                 |
| <code>getDay()</code><br><code>getUTCDay()</code>                               | Returns a number from 0 (Sunday) to 6 (Saturday) representing the day of the week in local time or UTC.                                                                                                                               |
| <code>getFullYear()</code><br><code>getUTCFullYear()</code>                     | Returns the year as a four-digit number in local time or UTC.                                                                                                                                                                         |
| <code>getHours()</code><br><code>getUTCHours()</code>                           | Returns a number from 0 to 23 representing hours since midnight in local time or UTC.                                                                                                                                                 |
| <code>getMilliseconds()</code><br><code>getUTCMilliseconds()</code>             | 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.                                                            |
| <code>getMinutes()</code><br><code>getUTCMinutes()</code>                       | Returns a number from 0 to 59 representing the minutes for the time in local time or UTC.                                                                                                                                             |
| <code>getMonth()</code><br><code>getUTCMonth()</code>                           | Returns a number from 0 (January) to 11 (December) representing the month in local time or UTC.                                                                                                                                       |
| <code>getSeconds()</code><br><code>getUTCSeconds()</code>                       | Returns a number from 0 to 59 representing the seconds for the time in local time or UTC.                                                                                                                                             |
| <code>getTime()</code>                                                          | Returns the number of milliseconds between January 1, 1970, and the time in the Date object.                                                                                                                                          |
| <code>getTimezoneOffset()</code>                                                | Returns the difference in minutes between the current time on the local computer and UTC (Coordinated Universal Time).                                                                                                                |
| <code> setDate( val )</code><br><code>setUTCDate( val )</code>                  | Sets the day of the month (1 to 31) in local time or UTC.                                                                                                                                                                             |
| <code>setFullYear( y, m, d )</code><br><code>setUTCFullYear( y, m, d )</code>   | 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.                          |
| <code>setHours( h, m, s, ms )</code><br><code>setUTCHours( h, m, s, ms )</code> | 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. |
| <code>setMilliseconds( ms )</code><br><code>setUTCMilliseconds( ms )</code>     | Sets the number of milliseconds in local time or UTC.                                                                                                                                                                                 |

**Fig. 11.8** | Date object methods. (Part 1 of 2.)

| Method                                                                        | Description                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>setMinutes( m, s, ms )</code><br><code>setUTCMinutes( m, s, ms )</code> | 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.                                   |
| <code>setMonth( m, d )</code><br><code>setUTCMonth( m, d )</code>             | 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.                                                              |
| <code>setSeconds( s, ms )</code><br><code>setUTCSeconds( s, ms )</code>       | Sets the second in local time or UTC. The second argument, representing the milliseconds, is optional. If this argument is not specified, the current millisecond value in the Date object is used.                                                      |
| <code>getTime( ms )</code>                                                    | Sets the time based on its argument—the number of elapsed milliseconds since January 1, 1970.                                                                                                                                                            |
| <code>toLocaleString()</code>                                                 | 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 <i>09/13/07 15:47:22</i> in the United States and <i>13/09/07 15:47:22</i> in Europe. |
| <code>toUTCString()</code>                                                    | Returns a string representation of the date and time in the form: <i>15 Sep 2007 15:47:22 UTC</i>                                                                                                                                                        |
| <code>toString()</code>                                                       | Returns a string representation of the date and time in a form specific to the locale of the computer ( <i>Mon Sep 17 15:47:22 EDT 2007</i> in the United States).                                                                                       |
| <code>valueOf()</code>                                                        | The time in number of milliseconds since midnight, January 1, 1970. (Same as <code>getTime()</code> .)                                                                                                                                                   |

**Fig. 11.8 |** Date object methods. (Part 2 of 2.)

The script of Fig. 11.9 demonstrates many of the local time zone methods in Fig. 11.8. Line 12 creates a new Date object. The new operator allocates the memory for the Date object. The empty parentheses indicate a call to the Date object's **constructor** with no arguments. A constructor is an initializer method for an object. Constructors are 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.



### Software Engineering Observation 11.3

---

When an object is allocated with new, the object's constructor is called automatically to initialize the object before it is used in the program.

Lines 16–19 demonstrate the methods `toString`, `toLocaleString`, `toUTCString` and `valueOf`. Note that method `valueOf` returns a large integer value representing the total number of milliseconds between midnight, January 1, 1970, and the date and time stored in Date object `current`.

Lines 23–32 demonstrate the `Date` object's *get* methods for the local time zone. Note that method `getFullYear` returns the year as a four-digit number. Note as well that method `getTimeZoneOffset` returns the difference in minutes between the local time zone and UTC time (i.e., a difference of four hours in our time zone when this example was executed).

```

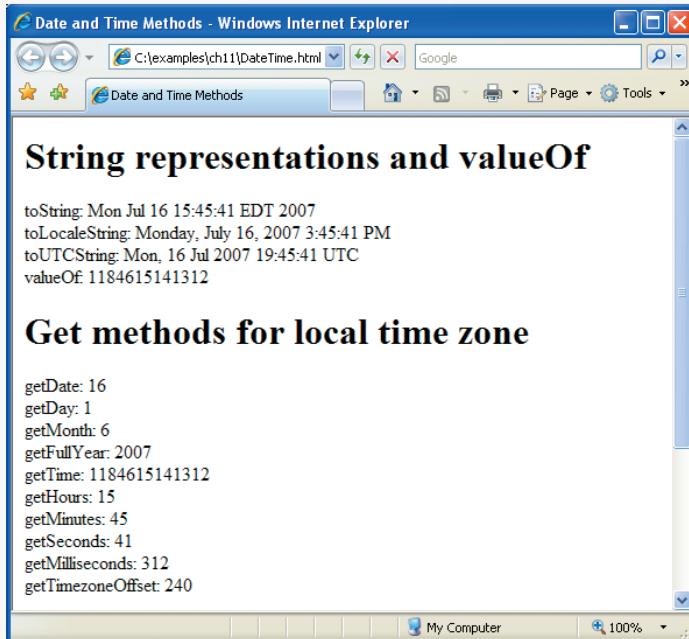
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.9: DateTime.html -->
6 <!-- Date and time methods of the Date object. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Date and Time Methods</title>
10 <script type = "text/javascript">
11 <!--
12 var current = new Date();
13
14 document.writeln(
15 "<h1>String representations and valueOf</h1>");
16 document.writeln("toString: " + current.toString() +
17 "
toLocaleString: " + current.toLocaleString() +
18 "
toUTCString: " + current.toUTCString() +
19 "
valueOf: " + current.valueOf());
20
21 document.writeln(
22 "<h1>Get methods for local time zone</h1>");
23 document.writeln("getDate: " + current.getDate() +
24 "
getDay: " + current.getDay() +
25 "
getMonth: " + current.getMonth() +
26 "
getFullYear: " + current.getFullYear() +
27 "
getTime: " + current.getTime() +
28 "
getHours: " + current.getHours() +
29 "
getMinutes: " + current.getMinutes() +
30 "
getSeconds: " + current.getSeconds() +
31 "
getMilliseconds: " + current.getMilliseconds() +
32 "
getTimezoneOffset: " + current.getTimezoneOffset());
33
34 document.writeln(
35 "<h1>Specifying arguments for a new Date</h1>");
36 var anotherDate = new Date(2007, 2, 18, 1, 5, 0, 0);
37 document.writeln("Date: " + anotherDate);
38
39 document.writeln("<h1>Set methods for local time zone</h1>");
40 anotherDate.setDate(31);
41 anotherDate.setMonth(11);
42 anotherDate.setFullYear(2007);
43 anotherDate.setHours(23);
44 anotherDate.setMinutes(59);
45 anotherDate.setSeconds(59);
46 document.writeln("Modified date: " + anotherDate);

```

**Fig. 11.9** | Date and time methods of the `Date` object. (Part I of 2.)

```

47 // -->
48 </script>
49 </head><body></body>
50 </html>
```



**Fig. 11.9** | Date and time methods of the Date object. (Part 2 of 2.)

Line 36 demonstrates creating a new Date object and supplying arguments to the Date constructor for *year*, *month*, *date*, *hours*, *minutes*, *seconds* and *milliseconds*. Note that the *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. For the *hours*, *minutes* and *seconds* arguments, if the argument to the right of any of these arguments is specified, it too must be specified (e.g., if the *minutes* argument is specified, the *hours* argument must be specified; if the *milliseconds* argument is specified, all the arguments must be specified).

Lines 40–45 demonstrate the `Date` object *set* methods for the local time zone. `Date` objects represent the month internally as an integer from 0 to 11. These values are off by one from what you might expect (i.e., 1 for January, 2 for February, ..., and 12 for December). When creating a `Date` object, you must specify 0 to indicate January, 1 to indicate February, ..., and 11 to indicate December.



### Common Programming Error 11.2

*Assuming that months are represented as numbers from 1 to 12 leads to off-by-one errors when you are processing Dates.*

The `Date` object provides two other methods that can be called without creating a new `Date` object—`Date.parse` and `Date.UTC`. Method `Date.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. This value can be converted to a `Date` object with the statement

```
var theDate = new Date(numberOfMilliseconds);
```

which passes to the `Date` constructor the number of milliseconds since midnight, January 1, 1970, for the `Date` object.

Method `parse` converts the string using the following rules:

- Short dates can be specified in the form MM-DD-YY, MM-DD-YYYY, MM/DD/YY or MM/DD/YYYY. The month and day are not required to be two digits.
- Long dates that specify the complete month name (e.g., “January”), date and year can specify the month, date and year in any order.
- Text in parentheses within the string is treated as a comment and ignored. Commas and white-space characters are treated as delimiters.
- All month and day names must have at least two characters. The names are not required to be unique. If the names are identical, the name is resolved as the last match (e.g., “Ju” represents “July” rather than “June”).
- If the name of the day of the week is supplied, it is ignored.
- All standard time zones (e.g., EST for Eastern Standard Time), Coordinated Universal Time (UTC) and Greenwich Mean Time (GMT) are recognized.
- When specifying hours, minutes and seconds, separate each by colons.
- When using a 24-hour-clock format, “PM” should not be used for times after 12 noon.

`Date` method `UTC` returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified as its arguments. The arguments to the `UTC` method include the required *year*, *month* and *date*, and the optional *hours*, *minutes*, *seconds* and *milliseconds*. If any of the *hours*, *minutes*, *seconds* or *milliseconds* arguments is not specified, a zero is supplied in its place. For the *hours*, *minutes* and *seconds* arguments, if the argument to the right of any of these arguments in the argument list is specified, that argument must also be specified (e.g., if the *minutes* argument is specified, the *hours* argument must be specified; if the *milliseconds* argument is specified, all the arguments must be specified). As

with the result of `Date.parse`, the result of `Date.UTC` can be converted to a `Date` object by creating a new `Date` object with the result of `Date.UTC` as its argument.

## 11.6 Boolean and Number Objects

JavaScript provides the `Boolean` and `Number` objects as object **wrappers** for boolean `true`/`false` values and numbers, respectively. These wrappers define methods and properties useful in manipulating boolean values and numbers. Wrappers provide added functionality for working with simple data types.

When a JavaScript program requires a boolean value, JavaScript automatically creates a `Boolean` object to store the value. JavaScript programmers can create `Boolean` objects explicitly with the statement

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

The constructor argument `booleanValue` specifies whether the value of the `Boolean` object should be `true` or `false`. If `booleanValue` is `false`, `0`, `null`, `Number.NaN` or an empty string (""), or if no argument is supplied, the new `Boolean` object contains `false`. Otherwise, the new `Boolean` object contains `true`. Figure 11.10 summarizes the methods of the `Boolean` object.

JavaScript automatically creates `Number` objects to store numeric values in a JavaScript program. JavaScript programmers can create a `Number` object with the statement

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

The constructor argument `numericValue` is the number to store in the object. Although you can explicitly create `Number` objects, normally the JavaScript interpreter creates them as needed. Figure 11.11 summarizes the methods and properties of the `Number` object.

| Method                  | Description                                                                                                                             |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <code>toString()</code> | Returns the string "true" if the value of the <code>Boolean</code> object is <code>true</code> ; otherwise, returns the string "false". |
| <code>valueOf()</code>  | Returns the value <code>true</code> if the <code>Boolean</code> object is <code>true</code> ; otherwise, returns <code>false</code> .   |

**Fig. 11.10** | Boolean object methods.

| Method or property             | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>toString( radix )</code> | Returns the string representation of the number. The optional <code>radix</code> argument (a number from 2 to 36) specifies the number's base. For example, <code>radix 2</code> results in the binary representation of the number, <code>8</code> results in the octal representation, <code>10</code> results in the decimal representation and <code>16</code> results in the hexadecimal representation. See Appendix E, Number Systems, for a review of the binary, octal, decimal and hexadecimal number systems. |

**Fig. 11.11** | Number object methods and properties. (Part 1 of 2.)

| Method or property                    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>valueOf()</code>                | Returns the numeric value.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>Number.MAX_VALUE</code>         | This property represents the largest value that can be stored in a JavaScript program—approximately 1.79E+308.                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>Number.MIN_VALUE</code>         | This property represents the smallest value that can be stored in a JavaScript program—approximately 5.00E-324.                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>Number.NaN</code>               | This property represents <i>not a number</i> —a value returned from an arithmetic expression that does not result in a number (e.g., the expression <code>parseInt("hello")</code> cannot convert the string "hello" into a number, so <code>parseInt</code> would return <code>Number.NaN</code> ). To determine whether a value is <code>NaN</code> , test the result with function <code>isNaN</code> , which returns <code>true</code> if the value is <code>NaN</code> ; otherwise, it returns <code>false</code> . |
| <code>Number.NEGATIVE_INFINITY</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|                                       | This property represents a value less than <code>-Number.MAX_VALUE</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>Number.POSITIVE_INFINITY</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|                                       | This property represents a value greater than <code>Number.MAX_VALUE</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                            |

**Fig. 11.11** | Number object methods and properties. (Part 2 of 2.)

## 11.7 document Object

The `document` object is used to manipulate the document that is currently visible in the browser window. The `document` object has many properties and methods, such as methods `document.write` and `document.writeln`, which have both been used in prior JavaScript examples. Figure 11.12 shows the methods and properties of the `document` objects that are used in this chapter. You can learn more about the properties and methods of the `document` object in our JavaScript Resource Center ([www.deitel.com/javascript](http://www.deitel.com/javascript)).

| Method or property                | Description                                                                                                                                |
|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getElementById( id )</code> | Returns the DOM node representing the XHTML element whose <code>id</code> attribute matches <code>id</code> .                              |
| <code>write( string )</code>      | Writes the string to the XHTML document as XHTML code.                                                                                     |
| <code>writeln( string )</code>    | Writes the string to the XHTML document as XHTML code and adds a newline character at the end.                                             |
| <code>cookie</code>               | A string containing the values of all the cookies stored on the user's computer for the current document. See Section 11.9, Using Cookies. |
| <code>lastModified</code>         | The date and time that this document was last modified.                                                                                    |

**Fig. 11.12** | Important `document` object methods and properties.

## 11.8 window Object

The `window` object provides methods for manipulating browser windows. The following script shows many of the commonly used properties and methods of the `window` object and uses them to create a website that spans multiple browser windows. Figure 11.13 allows the user to create a new, fully customized browser window by completing an XHTML form and clicking the `Submit` button. The script also allows the user to add text to the new window and navigate the window to a different URL.

The script starts in line 10. Line 12 declares a variable to refer to the new window. We refer to the new window as the **child window** because it is created and controlled by the main, or **parent**, window in this script. Lines 14–50 define the `createChildWindow` function, which determines the features that have been selected by the user and creates a child window with those features (but does not add any content to the window). Lines 18–20 declare several variables to store the status of the checkboxes on the page. Lines 23–38 set each variable to "yes" or "no" based on whether the corresponding checkbox is checked or unchecked.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.13: window.html -->
6 <!-- Using the window object to create and modify child windows. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Using the Window Object</title>
10 <script type = "text/javascript">
11 <!--
12 var childWindow; // variable to control the child window
13
14 function createChildWindow()
15 {
16 // these variables all contain either "yes" or "no"
17 // to enable or disable a feature in the child window
18 var toolBar;
19 var menuBar;
20 var scrollBars;
21
22 // determine whether the Tool Bar checkbox is checked
23 if (document.getElementById("toolBarCheckBox").checked)
24 toolBar = "yes";
25 else
26 toolBar = "no";
27
28 // determine whether the Menu Bar checkbox is checked
29 if (document.getElementById("menuBarCheckBox").checked)
30 menuBar = "yes";
31 else
32 menuBar = "no";
33

```

**Fig. 11.13** | Using the `window` object to create and modify child windows. (Part 1 of 4.)

```
34 // determine whether the Scroll Bar checkbox is checked
35 if (document.getElementById("scrollBarsCheckBox").checked)
36 scrollBars = "yes";
37 else
38 scrollBars = "no";
39
40 //display window with selected features
41 childWindow = window.open("", "",
42 ",toolbar = " + toolbar +
43 ",menubar = " + menuBar +
44 ",scrollbars = " + scrollBars);
45
46 // disable buttons
47 document.getElementById("closeButton").disabled = false;
48 document.getElementById("modifyButton").disabled = false;
49 document.getElementById("setURLButton").disabled = false;
50 } // end function createChildWindow
51
52 // insert text from the textbox in the child window
53 function modifyChildWindow()
54 {
55 if (childWindow.closed)
56 alert("You attempted to interact with a closed window");
57 else
58 childWindow.document.write(
59 document.getElementById("textForChild").value);
60 } // end function modifyChildWindow
61
62 // close the child window
63 function closeChildWindow()
64 {
65 if (childWindow.closed)
66 alert("You attempted to interact with a closed window");
67 else
68 childWindow.close();
69
70 document.getElementById("closeButton").disabled = true;
71 document.getElementById("modifyButton").disabled = true;
72 document.getElementById("setURLButton").disabled = true;
73 } // end function closeChildWindow
74
75 // set the URL of the child window to the URL
76 // in the parent window's myChildURL
77 function setChildWindowURL()
78 {
79 if (childWindow.closed)
80 alert("You attempted to interact with a closed window");
81 else
82 childWindow.location =
83 document.getElementById("myChildURL").value;
84 } // end function setChildWindowURL
85 //-->
86 </script>
```

Fig. 11.13 | Using the window object to create and modify child windows. (Part 2 of 4.)

```

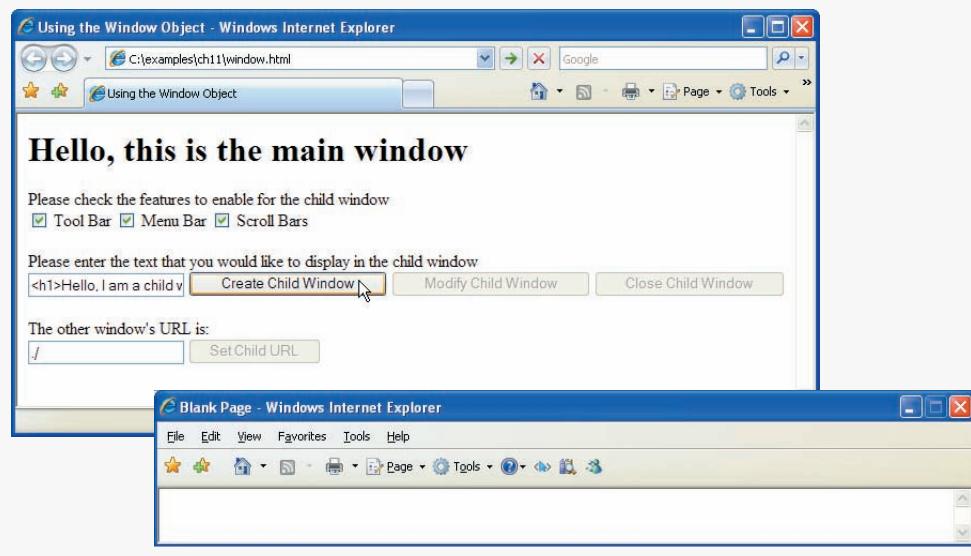
87 </head>
88 <body>
89 <h1>Hello, this is the main window</h1>
90 <p>Please check the features to enable for the child window

91 <input id = "toolBarCheckBox" type = "checkbox" value = ""
92 checked = "checked" />
93 <label>Tool Bar</label>
94 <input id = "menuBarCheckBox" type = "checkbox" value = ""
95 checked = "checked" />
96 <label>Menu Bar</label>
97 <input id = "scrollBarsCheckBox" type = "checkbox" value = ""
98 checked = "checked" />
99 <label>Scroll Bars</label></p>
100
101 <p>Please enter the text that you would like to display
102 in the child window

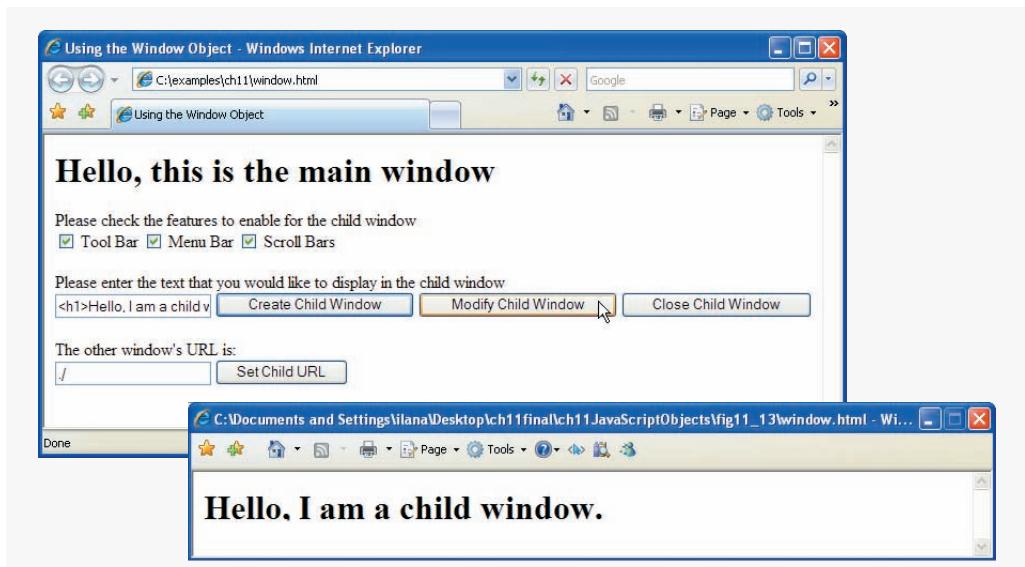
103 <input id = "textForChild" type = "text"
104 value = "<h1>Hello, I am a child window.</h1> " />
105 <input id = "createButton" type = "button"
106 value = "Create Child Window" onclick = "createChildWindow()" />
107 <input id = "modifyButton" type = "button" value = "Modify Child Window"
108 onclick = "modifyChildWindow()" disabled = "disabled" />
109 <input id = "closeButton" type = "button" value = "Close Child Window"
110 onclick = "closeChildWindow()" disabled = "disabled" /></p>
111
112 <p>The other window's URL is:

113 <input id = "myChildURL" type = "text" value = "./" />
114 <input id = "setURLButton" type = "button" value = "Set Child URL"
115 onclick = "setChildWindowURL()" disabled = "disabled" /></p>
116 </body>
117 </html>

```



**Fig. 11.13** | Using the window object to create and modify child windows. (Part 3 of 4.)



**Fig. 11.13** | Using the window object to create and modify child windows. (Part 4 of 4.)

The statement in lines 41–44 uses the `window` object’s `open` method to create the requested child window. Method `open` has three parameters. The first parameter is the URL of the page to open in the new window, and the second parameter is the name of the window. If you specify the `target` attribute of an `a` (anchor) element to correspond to the name of a window, the `href` of the link will be opened in the window. In our example, we pass `window.open` empty strings as the first two parameter values because we want the new window to open a blank page, and we use a different method to manipulate the child window’s URL.

The third parameter of the `open` method is a string of comma-separated, all-lowercase feature names, each followed by an `=` sign and either "yes" or "no" to determine whether that feature should be displayed in the new window. If these parameters are omitted, the browser defaults to a new window containing an empty page, no title and all features visible. [Note: If your menu bar is normally hidden in IE7, it will not appear in the child window. Press the `Alt` key to display it.] Lines 47–49 enable the buttons for manipulating the child window—these are initially disabled when the page loads.

Lines 53–60 define the function `modifyChildWindow`, which adds a line of text to the content of the child window. In line 55, the script determines whether the child window is closed. Function `modifyChildWindow` uses property `childWindow.closed` to obtain a boolean value that is `true` if `childWindow` is closed and `false` if the window is still open. If the window is closed, an alert box is displayed notifying the user that the window is currently closed and cannot be modified. If the child window is open, lines 58–59 obtain text from the `textForChild` input (lines 103–104) in the XHTML form in the parent window and uses the child’s `document.write` method to write this text to the child window.

Function `closeChildWindow` (lines 63–73) also determines whether the child window is closed before proceeding. If the child window is closed, the script displays an alert box telling the user that the window is already closed. If the child window is open, line 68

closes it using the `childWindow.close` method. Lines 70–72 disable the buttons that interact with the child window.



### Look-and-Feel Observation 11.1

*Popup windows should be used sparingly. Many users dislike websites that open additional windows, or that resize or reposition the browser. Some users have popup blockers that will prevent new windows from opening.*



### Software Engineering Observation 11.4

`window.location` is a property that always contains a string representation of the URL displayed in the current window. Typically, web browsers will allow a script to retrieve the `window.location` property of another window only if the script belongs to the same website as the page in the other window.

Function `setChildWindowURL` (lines 77–84) copies the contents of the `myChildURL` text field to the `location` property of the child window. If the child window is open, lines 81–82 set property `location` of the child window to the string in the `myChildURL` textbox. This action changes the URL of the child window and is equivalent to typing a new URL into the window's address bar and clicking `Go` (or pressing `Enter`).

The script ends in line 86. Lines 88–116 contain the body of the XHTML document, comprising a form that contains checkboxes, buttons, textboxes and form field labels. The script uses the form elements defined in the body to obtain input from the user. Lines 106, 108, 110, and 115 specify the `onclick` attributes of XHTML buttons. Each button is set to call a corresponding JavaScript function when clicked.

Figure 11.14 contains a list of some commonly used methods and properties of the `window` object.

| Method or property                      | Description                                                                                                                                                                                                             |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>open( url, name, options )</code> | Creates a new window with the URL of the window set to <code>url</code> , the name set to <code>name</code> to refer to it in the script, and the visible features set by the string passed in as <code>option</code> . |
| <code>prompt( prompt, default )</code>  | Displays a dialog box asking the user for input. The text of the dialog is <code>prompt</code> , and the default value is set to <code>default</code> .                                                                 |
| <code>close()</code>                    | Closes the current window and deletes its object from memory.                                                                                                                                                           |
| <code>focus()</code>                    | This method gives focus to the window (i.e., puts the window in the foreground, on top of any other open browser windows).                                                                                              |
| <code>blur()</code>                     | This method takes focus away from the window (i.e., puts the window in the background).                                                                                                                                 |
| <code>window.document</code>            | This property contains the <code>document</code> object representing the document currently inside the window.                                                                                                          |
| <code>window.closed</code>              | This property contains a boolean value that is set to true if the window is closed, and false if it is not.                                                                                                             |

**Fig. 11.14** | Important `window` object methods and properties. (Part 1 of 2.)

| Method or property         | Description                                                                                                                  |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>window.opener</code> | This property contains the <code>window</code> object of the window that opened the current window, if such a window exists. |

**Fig. 11.14** | Important `window` object methods and properties. (Part 2 of 2.)

## 11.9 Using Cookies

Cookies provide web developers with a tool for personalizing web pages. A **cookie** is a piece of data that is stored on the user's computer to maintain information about the client during and between browser sessions. A website may store a cookie on the client's computer to record user preferences or other information that the website can retrieve during the client's subsequent visits. For example, a website can retrieve the user's name from a cookie and use it to display a personalized greeting.

Microsoft Internet Explorer and Mozilla Firefox store cookies as small text files on the client's hard drive. When a user visits a website, the browser locates any cookies written by scripts on that site and makes them available to any scripts located on the site. Note that cookies may be accessed only by scripts belonging to the same website from which they originated (i.e., a cookie set by a script on `amazon.com` can be read only by other scripts on `amazon.com`).

Cookies are accessible in JavaScript through the `document` object's **cookie** property. JavaScript treats a cookie as a string of text. Any standard string function or method can manipulate a cookie. A cookie has the syntax "`identifier=value`," where `identifier` is any valid JavaScript variable identifier, and `value` is the value of the cookie variable. When multiple cookies exist for one website, `identifier-value` pairs are separated by semicolons in the `document.cookie` string.

Cookies differ from ordinary strings in that each cookie has an expiration date, after which the web browser deletes it. This date can be defined by setting the **expires** property in the cookie string. If a cookie's expiration date is not set, then the cookie expires by default after the user closes the browser window. A cookie can be deleted immediately by setting the `expires` property to a date and time in the past.

The assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it. Thus, if we set two cookies

```
document.cookie = "name1=value1;";
document.cookie = "name2=value2;";
```

`document.cookie` will contain "`name1=value1; name2=value2`".

Figure 11.15 uses a cookie to store the user's name and displays a personalized greeting. This example improves upon the functionality in the dynamic welcome page example of Fig. 6.17 by requiring the user to enter a name only during the first visit to the web page. On each subsequent visit, the script can display the user name that is stored in the cookie.

Line 10 begins the script. Lines 12–13 declare the variables needed to obtain the time, and line 14 declares the variable that stores the name of the user. Lines 16–27 contain the same `if...else` statement used in Fig. 6.17 to display a time-sensitive greeting.

Lines 30–66 contain the code used to manipulate the cookie. Line 30 determines whether a cookie exists on the client computer. The expression `document.cookie` evaluates to true if a cookie exists. If a cookie does not exist, then the script prompts the user to enter a name (line 45). The script creates a cookie containing the string "name=", followed by a copy of the user's name produced by the built-in JavaScript function `escape` (line 49). The function `escape` converts any non-alphanumeric characters, such as spaces

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.15: cookie.html -->
6 <!-- Using cookies to store user identification data. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Using Cookies</title>
10 <script type = "text/javascript">
11 <!--
12 var now = new Date(); // current date and time
13 var hour = now.getHours(); // current hour (0-23)
14 var name;
15
16 if (hour < 12) // determine whether it is morning
17 document.write("<h1>Good Morning, ");
18 else
19 {
20 hour = hour - 12; // convert from 24-hour clock to PM time
21
22 // determine whether it is afternoon or evening
23 if (hour < 6)
24 document.write("<h1>Good Afternoon, ");
25 else
26 document.write("<h1>Good Evening, ");
27 } // end else
28
29 // determine whether there is a cookie
30 if (document.cookie)
31 {
32 // convert escape characters in the cookie string to their
33 // English notation
34 var myCookie = unescape(document.cookie);
35
36 // split the cookie into tokens using = as delimiter
37 var cookieTokens = myCookie.split("=");
38
39 // set name to the part of the cookie that follows the = sign
40 name = cookieTokens[1];
41 } // end if
42 else
43 {
44 // if there was no cookie, ask the user to input a name
45 name = window.prompt("Please enter your name", "Paul");

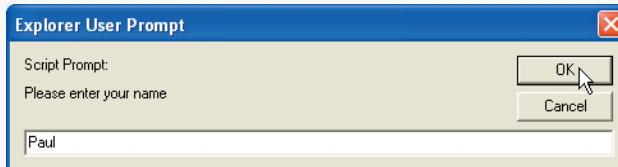
```

**Fig. 11.15** | Using cookies to store user identification data. (Part I of 3.)

```

46 // escape special characters in the name string
47 // and add name to the cookie
48 document.cookie = "name=" + escape(name);
49 } // end else
50
51
52 document.writeln(
53 name + ", welcome to JavaScript programming!</h1>");
54 document.writeln(" " +
55 "Click here if you are not " + name + "");
56
57 // reset the document's cookie if wrong person
58 function wrongPerson()
59 {
60 // reset the cookie
61 document.cookie= "name=null;" +
62 " expires=Thu, 01-Jan-95 00:00:01 GMT";
63
64 // reload the page to get a new name after removing the cookie
65 location.reload();
66 } // end function wrongPerson
67
68 // -->
69 </script>
70 </head>
71 <body>
72 <p>Click Refresh (or Reload) to run the script again</p>
73 </body>
74 </html>

```



**Fig. 11.15** | Using cookies to store user identification data. (Part 2 of 3.)



**Fig. 11.15** | Using cookies to store user identification data. (Part 3 of 3.)

and semicolons, in a string to their equivalent **hexadecimal escape sequences** of the form "%XX," where XX is the two-digit hexadecimal ASCII value of a special character. For example, if name contains the value "David Green", the statement `escape( name )` evaluates to "David%20Green", because the hexadecimal ASCII value of a blank space is 20. It is a good idea to always escape cookie values before writing them to the client. This conversion prevents any special characters in the cookie from being misinterpreted as having a special meaning in the code, rather than being a character in a cookie value. For instance, a semicolon in a cookie value could be misinterpreted as a semicolon separating two adjacent *identifier-value* pairs. Applying the function **unescape** to cookies when they are read out of the `document.cookie` string converts the hexadecimal escape sequences back to English characters for display in a web page.



### Good Programming Practice 11.2

*Always store values in cookies with self-documenting identifiers. Do not forget to include the identifier followed by an = sign before the value being stored.*

If a cookie exists (i.e., the user has been to the page before), then the script **parses** the user name out of the cookie string and stores it in a local variable. Parsing generally refers to the act of splitting a string into smaller, more useful components. Line 34 uses the JavaScript function `unescape` to replace all the escape sequences in the cookie with their equivalent English-language characters. The script stores the unescaped cookie value in the variable `myCookie` (line 34) and uses the JavaScript function `split` (line 37), introduced in Section 11.4.5, to break the cookie into identifier and value tokens. At this point in the script, `myCookie` contains a string of the form "name=value". We call `split` on `myCookie` with = as the delimiter to obtain the `cookieTokens` array, with the first element equal to the name of the identifier and the second element equal to the value of the identifier. Line 40 assigns the value of the second element in the `cookieTokens` array (i.e., the actual value stored in the cookie) to the variable `name`. Lines 52–53 add the personalized greeting to the web page, using the user's name stored in the cookie.

The script allows the user to reset the cookie, which is useful in case someone new is using the computer. Lines 54–55 create a hyperlink that, when clicked, calls the JavaScript function `wrongPerson` (lines 58–66). Lines 61–62 set the cookie name to `null` and the `expires` property to January 1, 1995 (though any date in the past will suffice). Internet Explorer detects that the `expires` property is set to a date in the past and deletes the cookie from the user's computer. The next time this page loads, no cookie will be found. The `reload` method of the `location` object forces the page to refresh (line 65), and, unable to find an existing cookie, the script prompts the user to enter a new name.

## 11.10 Final JavaScript Example

The past few chapters have explored many JavaScript concepts and how they can be applied on the web. The next JavaScript example combines many of these concepts into a single web page. Figure 11.16 uses functions, cookies, arrays, loops, the `Date` object, the `window` object and the `document` object to create a sample welcome screen containing a personalized greeting, a short quiz, a random image and a random quotation. We have seen all of these concepts before, but this example illustrates how they work together on one web page.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.16: final.html -->
6 <!-- Rich welcome page using several JavaScript concepts. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Putting It All Together</title>
10 <script type = "text/javascript">
11 <!--
12 var now = new Date(); // current date and time
13 var hour = now.getHours(); // current hour
14
15 // array with names of the images that will be randomly selected
16 var pictures =
17 ["CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO"];
18
19 // array with the quotes that will be randomly selected
20 var quotes = [
21 "Form ever follows function.
" +
22 " Louis Henri Sullivan", "E pluribus unum." +
23 "(One composed of many.)
 Virgil", "Is it a" +
24 " world to hide virtues in?
 William Shakespeare"];
25
26 // write the current date and time to the web page
27 document.write("<p>" + now.toLocaleString() + "
</p>");
28
29 // determine whether it is morning
30 if (hour < 12)
 document.write("<h2>Good Morning, ");

```

**Fig. 11.16** | Rich welcome page using several JavaScript concepts. (Part I of 5.)

```

31 else
32 {
33 hour = hour - 12; // convert from 24-hour clock to PM time
34
35 // determine whether it is afternoon or evening
36 if (hour < 6)
37 document.write("<h2>Good Afternoon, ");
38 else
39 document.write("<h2>Good Evening, ");
40 } // end else
41
42 // determine whether there is a cookie
43 if (document.cookie)
44 {
45 // convert escape characters in the cookie string to their
46 // English notation
47 var myCookie = unescape(document.cookie);
48
49 // split the cookie into tokens using = as delimiter
50 var cookieTokens = myCookie.split("=");
51
52 // set name to the part of the cookie that follows the = sign
53 name = cookieTokens[1];
54 } // end if
55 else
56 {
57 // if there was no cookie, ask the user to input a name
58 name = window.prompt("Please enter your name", "Paul");
59
60 // escape special characters in the name string
61 // and add name to the cookie
62 document.cookie = "name =" + escape(name);
63 } // end else
64
65 // write the greeting to the page
66 document.writeln(
67 name + ", welcome to JavaScript programming!</h2>");
68
69 // write the link for deleting the cookie to the page
70 document.writeln(" " +
71 "Click here if you are not " + name + "
");
72
73 // write the random image to the page
74 document.write ("<img src = \""
75 pictures[Math.floor(Math.random() * 7)] +
76 ".gif\" />
");
77
78 // write the random quote to the page
79 document.write (quotes[Math.floor(Math.random() * 3)]);
80
81 // create a window with all the quotes in it
82 function allQuotes()
83 {

```

**Fig. 11.16** | Rich welcome page using several JavaScript concepts. (Part 2 of 5.)

```

84 // create the child window for the quotes
85 var quoteWindow = window.open("", "", "resizable=yes, " +
86 "toolbar=no, menubar=no, status=no, location=no," +
87 " scrollBars=yes");
88 quoteWindow.document.write("<p>")
89
90 // loop through all quotes and write them in the new window
91 for (var i = 0; i < quotes.length; i++)
92 quoteWindow.document.write((i + 1) + ". " +
93 quotes[i] + "

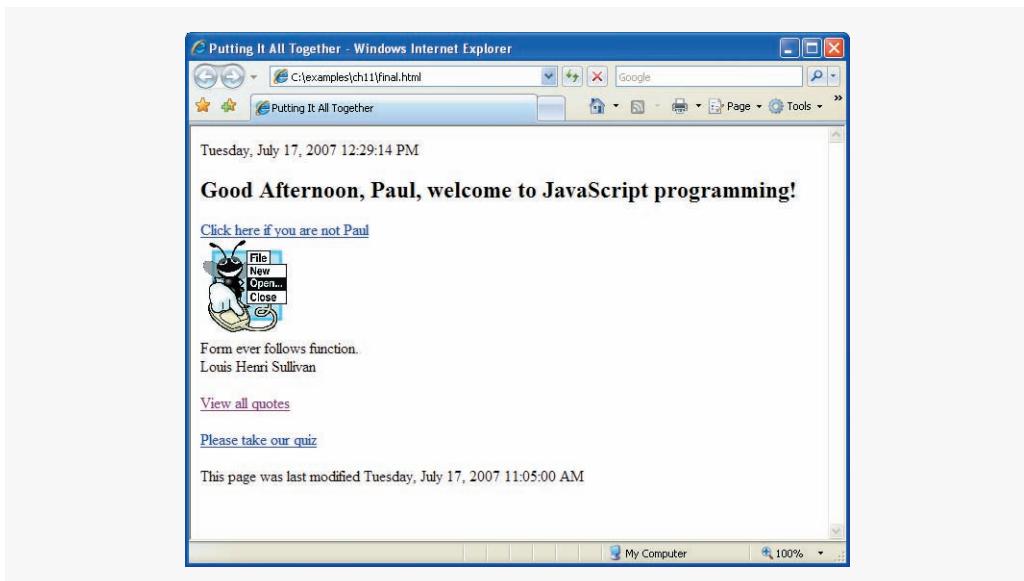
");
94
95 // write a close link to the new window
96 quoteWindow.document.write("</p>
<a href = " +
97 "\"/javascript:window.close()\">Close this window");
98 } // end function allQuotes
99
100 // reset the document's cookie if wrong person
101 function wrongPerson()
102 {
103 // reset the cookie
104 document.cookie= "name=null;" +
105 " expires=Thu, 01-Jan-95 00:00:01 GMT";
106
107 // reload the page to get a new name after removing the cookie
108 location.reload();
109 } // end function wrongPerson
110
111 // open a new window with the quiz2.html file in it
112 function openQuiz()
113 {
114 window.open("quiz2.html", "", "toolbar = no, " +
115 "menubar = no, scrollBars = no");
116 } // end function openQuiz
117 // -->
118 </script>
119 </head>
120 <body>
121 <p>View all quotes</p>
122
123 <p id = "quizSpot">
124 Please take our quiz</p>
125
126 <script type = "text/javascript">
127 // variable that gets the last modification date and time
128 var modDate = new Date(document.lastModified);
129
130 // write the last modified date and time to the page
131 document.write ("This page was last modified " +
132 modDate.toLocaleString());
133 </script>
134 </body>
135 </html>

```

**Fig. 11.16** | Rich welcome page using several JavaScript concepts. (Part 3 of 5.)

The screenshot shows a Windows Internet Explorer window titled "Putting It All Together - Windows Internet Explorer". The address bar displays "C:\examples\ch11\final.html". The main content area of the browser shows a welcome message: "Good Afternoon, Paul, welcome to JavaScript programming!". Below this, there is a link "Click here if you are not Paul". A small cartoon character icon with a menu is visible. The text "Form ever follows function.  
Louis Henri Sullivan" is displayed. There are links "View all quotes" and "Please take our quiz". A timestamp at the bottom indicates the page was last modified on Tuesday, July 17, 2007, at 11:05:00 AM. The status bar at the bottom of the browser window shows "javascript:allQuotes()". Above the browser window, a separate "Explorer User Prompt" dialog box is open, titled "Script Prompt". It contains the message "Please enter your name" and a text input field containing "Paul". It includes "OK" and "Cancel" buttons.

**Fig. 11.16** | Rich welcome page using several JavaScript concepts. (Part 4 of 5.)



**Fig. 11.16** | Rich welcome page using several JavaScript concepts. (Part 5 of 5.)

The script that builds most of this page starts in line 10. Lines 12–13 declare variables needed for determining the time of day. Lines 16–23 create two arrays from which content is randomly selected. This web page contains both an image (whose filename is randomly selected from the pictures array) and a quote (whose text is randomly selected from the quotes array). Line 26 writes the user's local date and time to the web page using the Date object's `toLocaleString` method. Lines 29–40 display a time-sensitive greeting using the same code as Fig. 6.17. The script either uses an existing cookie to obtain the user's name (lines 43–54) or prompts the user for a name, which the script then stores in a new cookie (lines 55–63). Lines 66–67 write the greeting to the web page, and lines 70–71 produce the link for resetting the cookie. This is the same code used in Fig. 11.15 to manipulate cookies. Lines 74–79 write the random image and random quote to the web page. The script chooses each by randomly selecting an index into each array. This code is similar to the code used in Fig. 10.7 to display a random image using an array.

Function `allQuotes` (lines 82–98) uses the `window` object and a `for` loop to open a new window containing all the quotes in the `quotes` array. Lines 85–87 create a new window called `quoteWindow`. The script does not assign a URL or a name to this window, but it does specify the window features to display. Line 88 opens a new paragraph in `quoteWindow`. A `for` loop (lines 91–93) traverses the `quotes` array and writes each quote to `quoteWindow`. Lines 96–97 close the paragraph in `quoteWindow`, insert a new line and add a link at the bottom of the page that allows the user to close the window. Note that `allQuotes` generates a web page and opens it in an entirely new window with JavaScript.

Function `wrongPerson` (lines 101–109) resets the cookie storing the user's name. This function is identical to function `wrongPerson` in Fig. 11.15.

Function `openQuiz` (lines 112–116) opens a new window to display a sample quiz. Using the `window.open` method, the script creates a new window containing `quiz2.html` (lines 114–115). We discuss `quiz2.html` later in this section.

The primary script ends in line 118, and the body of the XHTML document begins in line 120. Line 121 creates the link that calls function `a11Quotes` when clicked. Lines 123–124 create a paragraph element containing the attribute `id = "quizSpot"`. This paragraph contains a link that calls function `openQuiz`.

Lines 126–133 contain a second script. This script appears in the XHTML document's body because it adds a dynamic footer to the page, which must appear after the static XHTML content contained in the first part of the body. This script creates another instance of the `Date` object, but the date is set to the last modified date and time of the XHTML document, rather than the current date and time (line 128). The script obtains the last modified date and time using property `document.lastModified`. Lines 131–132 add this information to the web page. Note that the last modified date and time appear at the bottom of the page, after the rest of the body content. If this script were in the `head` element, this information would be displayed before the entire body of the XHTML document. Lines 133–135 close the `script`, the `body` and the XHTML document.

### *The Quiz Page*

The quiz used in this example is in a separate XHTML document named `quiz2.html` (Fig. 11.17). This document is similar to `quiz.html` in Fig. 10.14. The quiz in this example differs from the quiz in Fig. 10.14 in that it shows the result in the main window in the example, whereas the earlier quiz example alerts the result. After the **Submit** button in the quiz window is clicked, the main window changes to reflect that the quiz was taken, and the quiz window closes.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 11.17: quiz2.html -->
6 <!-- Online quiz in a child window. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Online Quiz</title>
10 <script type = "text/JavaScript">
11 <!--
12 function checkAnswers()
13 {
14 // determine whether the answer is correct
15 if (document.getElementById("myQuiz").elements[1].checked)
16 window.opener.document.getElementById("quizSpot").
17 innerHTML = "Congratulations, your answer is correct";
18 else // if the answer is incorrect
19 window.opener.document.getElementById("quizSpot").
20 innerHTML = "Your answer is incorrect. " +
21 "Please try again
 <a href = " +
22 "\\\'javascript:openQuiz()\\\'>Please take our quiz";
23
24 window.opener.focus();
25 window.close();
26 } // end function checkAnswers

```

**Fig. 11.17** | Online quiz in a child window. (Part I of 3.)

```

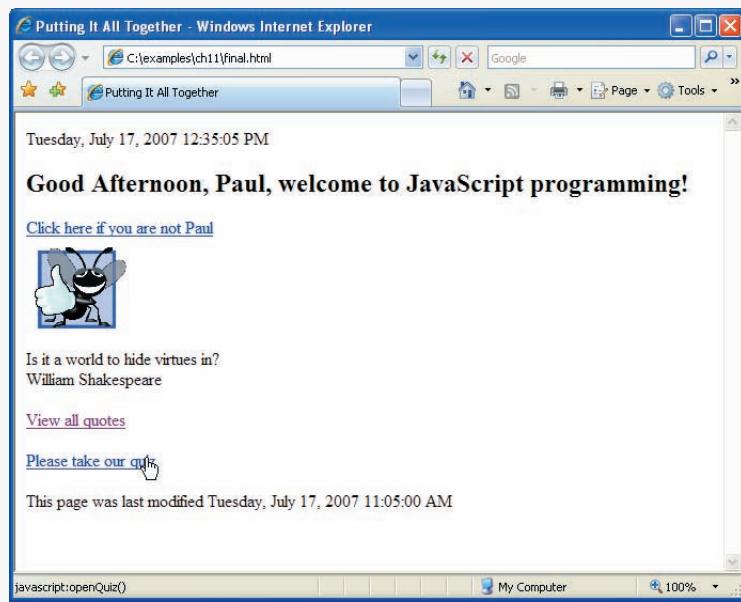
27 //-->
28 </script>
29 </head>
30 <body>
31 <form id = "myQuiz" action = "javascript:checkAnswers()">
32 <p>Select the name of the tip that goes with the
33 image shown:

34
35

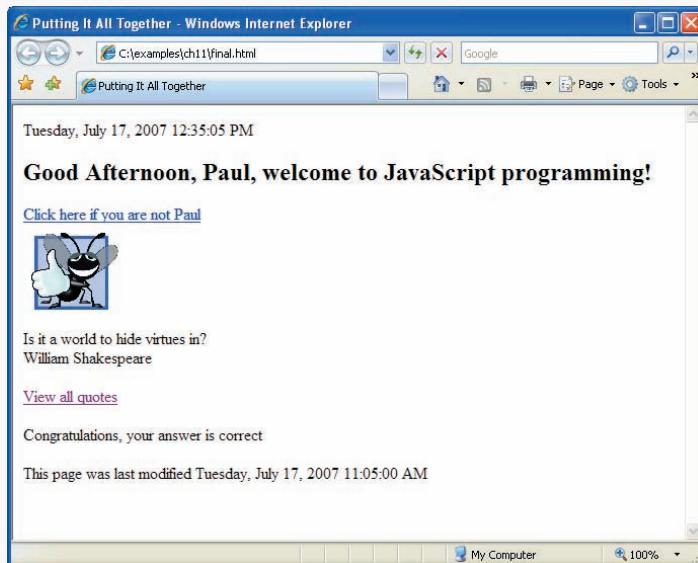
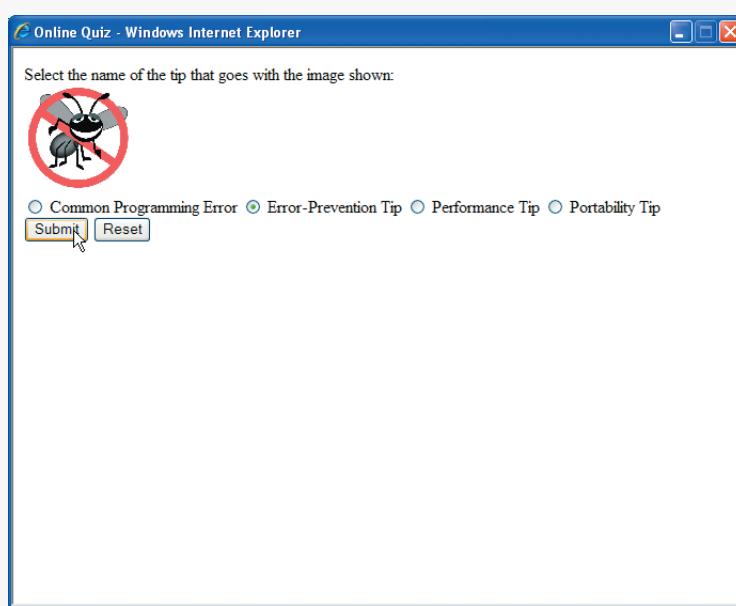
36
37 <input type = "radio" name = "radiobutton" value = "CPE" />
38 <label>Common Programming Error</label>
39
40 <input type = "radio" name = "radiobutton" value = "EPT" />
41 <label>Error-Prevention Tip</label>
42
43 <input type = "radio" name = "radiobutton" value = "PERF" />
44 <label>Performance Tip</label>
45
46 <input type = "radio" name = "radiobutton" value = "PORT" />
47 <label>Portability Tip</label>

48
49 <input type = "submit" name = "Submit" value = "Submit" />
50 <input type = "reset" name = "reset" value = "Reset" />
51 </p>
52 </form>
53 </body>
54 </html>

```



**Fig. 11.17** | Online quiz in a child window. (Part 2 of 3.)



**Fig. 11.17** | Online quiz in a child window. (Part 3 of 3.)

Lines 15–22 of this script check the user's answer and output the result to the main window. Lines 16–17 use `window.opener` to write to the main window. The property `window.opener` always contains a reference to the window that opened the current window, if such a window exists. Lines 16–17 write to property `window.opener.document.getElementById("quizSpot").innerHTML`. Recall that `quizSpot` is the id of the

paragraph in the main window that contains the link to open the quiz. Property `innerHTML` refers to the HTML code inside the `quizSpot` paragraph (i.e., the code between `<p>` and `</p>`). Modifying the `innerHTML` property dynamically changes the XHTML code in the paragraph. Thus, when lines 16–17 execute, the link in the main window disappears, and the string "Congratulations, your answer is correct." appears. Lines 19–22 modify `window.opener.document.getElementById("quizSpot").innerHTML`. Lines 19–22 use the same technique to display "Your answer is incorrect. Please try again", followed by a link to try the quiz again.

After checking the quiz answer, the script gives focus to the main window (i.e., puts the main window in the foreground, on top of any other open browser windows), using the method `focus` of the main window's `window` object. The property `window.opener` references the main window, so `window.opener.focus()` (line 24) gives the main window focus, allowing the user to see the changes made to the text of the main window's `quizSpot` paragraph. Finally, the script closes the quiz window, using method `window.close` (line 25).

Lines 28–29 close the `script` and `head` elements of the XHTML document. Line 30 opens the `body` of the XHTML document. The `body` contains the `form`, `image`, `text` labels and `radio` buttons that comprise the quiz. Lines 52–54 close the `form`, the `body` and the XHTML document.

## 11.11 Using JSON to Represent Objects

In 1999, [JSON \(JavaScript Object Notation\)](#)—a simple way to represent JavaScript objects as strings—was 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:

```
[value1, value2, value3]
```

Each value can be a string, a number, a JSON object, `true`, `false` or `null`. To appreciate the simplicity of JSON data, examine this representation of an array of address-book entries from Chapter 15:

```
[{ 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 extract data easily and to efficiently transmit data across the Internet. JSON integrates especially well with Ajax applications, discussed in Chapter 15. See Section 15.7 for a more detailed discussion of JSON, as well as an Ajax-specific example. For more information on JSON, visit our JSON Resource Center at [www.deitel.com/json](http://www.deitel.com/json).

## 11.12 Wrap-Up

This chapter provided an introduction to Object Technology, many of JavaScript's built-in objects, and a brief introduction to JSON, a simple way to represent new JavaScript objects. We introduced vocabulary and concepts integral to object-oriented and object-based programming. We took a closer look at some methods and properties of the `Math`, `Date`, `Boolean`, `Number`, `String`, `document`, and `window` objects. In the next chapter, we introduce the Document Object Model, a set of JavaScript objects that represent the elements in a web page that allows you to dynamically modify the page's content.

## 11.13 Web Resources

[www.deitel.com/javascript/](http://www.deitel.com/javascript/)

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

---

## Summary

### *Section 11.1 Introduction*

- The chapter describes several of JavaScript's built-in objects, which will serve as a basis for understanding browser objects in the chapters on Dynamic HTML.

### *Section 11.2 Introduction to Object Technology*

- Objects are a natural way of thinking about the world and about scripts that manipulate XHTML documents.
- JavaScript uses objects to perform many tasks and therefore is referred to as an object-based programming language.
- Objects have attributes and exhibit behaviors.
- Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics.
- OOD takes advantage of inheritance relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects.
- OOD also models communication between objects.
- OOD encapsulates attributes and operations (behaviors) into objects.
- Objects have the property of information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves.
- The designers of web browsers have defined a set of objects that encapsulate an XHTML document's elements and expose to a JavaScript programmer the attributes and behaviors that enable a JavaScript program to interact with (or script) those elements (objects).

- Some programming languages—like Java, Visual Basic, C# and C++—are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement object-oriented designs as working software systems.
- Languages like C are procedural, so programming tends to be action oriented.
- In procedural languages, the unit of programming is the function.
- In object-oriented languages, the unit of programming is the class from which objects are eventually instantiated. Classes contain functions that implement operations and data that implements attributes.
- Procedural programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs.
- Object-oriented programmers concentrate on creating their own user-defined types called classes. Each class contains data as well as the set of functions that manipulate that data and provide services to clients.
- The data components of a class are called properties.
- The function components of a class are called methods.
- The nouns in a system specification help you determine the set of classes from which objects are created that work together to implement the system.
- Classes are to objects as blueprints are to houses.
- Classes can have relationships with other classes. These relationships are called associations.
- Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components.
- With object technology, you can build much of the new software you'll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can reuse to speed and enhance the quality of future software development efforts.

### **Section 11.3 Math Object**

- Math object methods allow you to 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 the argument (or a comma-separated list of arguments) to the method.

### **Section 11.4 String Object**

- Characters are the fundamental 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 (often called anonymous String objects) are written as a sequence of characters in double quotation marks or single quotation marks.
- Combining strings is called concatenation.
- String method `charAt` returns the character at a specific index in a string. 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` (i.e., if the string contains five characters, the indices are 0 through 4). 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 index in a string. If the index is outside the bounds of the string, the method returns `Nan`. String method `fromCharCode` creates a string from a list of Unicode values.
- String method `toLowerCase` returns the lowercase version of a string. String method `toUpperCase` returns the uppercase version of a string.
- 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. This method 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. This method receives an optional second argument specifying the index from which to begin the search.
- The process of 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. The argument to method `split` is the delimiter string—the string that determines the end of each token in the original string. Method `split` 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.
- String method `anchor` wraps the string that calls the method in XHTML element `<a></a>` with the name of the anchor supplied as the argument to the method.
- String method `fixed` displays text in a fixed-width font by wrapping the string that calls the method in a `<tt></tt>` XHTML element.
- String method `strike` displays struck-out text (i.e., text with a line through it) by wrapping the string that calls the method in a `<strike></strike>` XHTML element.
- String method `sub` displays subscript text by wrapping the string that calls the method in a `<sub></sub>` XHTML element.
- String method `sup` displays superscript text by wrapping the string that calls the method in a `<sup></sup>` XHTML element.
- String method `link` creates a hyperlink by wrapping the string that calls the method in XHTML element `<a></a>`. The target of the hyperlink (i.e., value of the `href` property) is the argument to the method and can be any URL.

### **Section 11.5 Date Object**

- JavaScript's Date object provides methods for date and time manipulations.
- Date and time processing can be performed based either on the computer's local time zone or on World Time Standard's Coordinated Universal Time (abbreviated UTC)—formerly called Greenwich Mean Time (GMT).
- Most methods of the Date object have a local time zone and a UTC version.
- 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. The arguments to the `UTC` method include the required year, month and date, and the optional hours, minutes, seconds and milliseconds. If any of the hours, minutes, seconds or milliseconds arguments is not specified, a zero is supplied in its place. For the hours, minutes and seconds arguments, if the argument to the right of any of these arguments is specified, that argument must also be specified (e.g., if the minutes argument is specified, the hours argument must be specified; if the milliseconds argument is specified, all the arguments must be specified).

### **Section 11.6 Boolean and Number Objects**

- JavaScript provides the `Boolean` and `Number` objects as 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 with the statement

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

The argument `booleanValue` specifies the value of the `Boolean` object (`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`.

- JavaScript automatically creates `Number` objects to store numeric values in a JavaScript program.
- JavaScript programmers can create a `Number` object with the statement

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

The argument `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.

### **Section 11.7 document Object**

- JavaScript provides the `document` object for manipulating the document that is currently visible in the browser window.

### **Section 11.8 window Object**

- JavaScript's `window` object provides methods for manipulating browser windows.

### **Section 11.9 Using Cookies**

- A cookie is a piece of data that is stored on the user's computer to maintain information about the client during and between browser sessions.
- Cookies are accessible in JavaScript through the `document` object's `cookie` property.
- A cookie has the syntax "`identifier=value`," where `identifier` is any valid JavaScript variable identifier, and `value` is the value of the cookie variable. When multiple cookies exist for one website, `identifier-value` pairs are separated by semicolons in the `document.cookie` string.
- The `expires` property in a cookie string sets an expiration date, after which the web browser deletes the cookie. If a cookie's expiration date is not set, then the cookie expires by default after the user closes the browser window. A cookie can be deleted immediately by setting the `expires` property to a date and time in the past.
- The assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it.

### Section 11.10 Final JavaScript Example

- `window.opener` always contains a reference to the window that opened the current window.
- The property `innerHTML` refers to the HTML code inside the current paragraph element.
- Method `focus` puts the window it references on top of all the others.
- The `window` object's `close` method closes the browser window represented by the `window` object.

### Section 11.11 Using JSON to Represent Objects

- JSON (JavaScript Object Notation) is a simple way to represent JavaScript objects as strings.
- JSON was introduced in 1999 as an alternative to XML for data exchange.
- 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:

[ *value1*, *value2*, *value3* ]

- Values in JSON can be strings, numbers, JSON objects, `true`, `false` or `null`.

## Terminology

|                                                       |                                                             |
|-------------------------------------------------------|-------------------------------------------------------------|
| <code>abs</code> method of <code>Math</code>          | <code>exp</code> method of <code>Math</code>                |
| abstraction                                           | <code>fixed</code> method of <code>String</code>            |
| action-oriented programming language                  | <code>floor</code> method of <code>Math</code>              |
| <code>anchor</code> method of <code>String</code>     | <code>focus</code> method of <code>window</code>            |
| anonymous <code>String</code> object                  | <code>fromCharCode</code> method of <code>String</code>     |
| association                                           | <code>getDate</code> method of <code>Date</code>            |
| attribute (property)                                  | <code>getDay</code> method of <code>Date</code>             |
| behavior (method)                                     | <code>getFullYear</code> method of <code>Date</code>        |
| <code>blink</code> method of <code>String</code>      | <code>getHours</code> method of <code>Date</code>           |
| <code>Boolean</code> object                           | <code>getMilliseconds</code> method of <code>Date</code>    |
| <code>ceil</code> method of <code>Math</code>         | <code>getMinutes</code> method of <code>Date</code>         |
| character                                             | <code>getMonth</code> method of <code>Date</code>           |
| <code>charAt</code> method of <code>String</code>     | <code>getSeconds</code> method of <code>Date</code>         |
| <code>charCodeAt</code> method of <code>String</code> | <code>getTime</code> method of <code>Date</code>            |
| class                                                 | <code>getTimezoneOffset</code> method of <code>Date</code>  |
| <code>close</code> method of <code>window</code>      | <code>getUTCDate</code> method of <code>Date</code>         |
| code reuse                                            | <code>getUTCDay</code> method of <code>Date</code>          |
| components                                            | <code>getUTCFullYear</code> method of <code>Date</code>     |
| <code>concat</code> method of <code>String</code>     | <code>getUTCHours</code> method of <code>Date</code>        |
| cookie                                                | <code>getUTCMilliseconds</code> method of <code>Date</code> |
| Coordinated Universal Time (UTC)                      | <code>getUTCMinutes</code> method of <code>Date</code>      |
| <code>cos</code> method of <code>Math</code>          | <code>getUTCMonth</code> method of <code>Date</code>        |
| date                                                  | <code>getUTCSeconds</code> method of <code>Date</code>      |
| <code>Date</code> object                              | Greenwich Mean Time (GMT)                                   |
| delimiter                                             | hexadecimal escape sequences                                |
| <code>document</code> object                          | hiding                                                      |
| <code>E</code> property of <code>Math</code>          | index in a string                                           |
| empty string                                          | <code>indexOf</code> method of <code>String</code>          |
| encapsulation                                         | information hiding                                          |
| <code>escape</code> function                          | inheritance                                                 |

innerHTML property  
 instantiation  
 interface  
 lastIndexOf method of String  
 link method of String  
 LN10 property of Math  
 LN2 property of Math  
 local time zone  
 log method of Math  
 LOG10E property of Math  
 LOG2E property of Math  
 Math object  
 max method of Math  
 MAX\_SIZE property of Number  
 method  
 min method of Math  
 MIN\_SIZE property of Number  
 NaN property of Number  
 NEGATIVE\_INFINITY property of Number  
 Number object  
 object  
 object wrapper  
 object-based programming language  
 object-oriented design (OOD)  
 object-oriented programming (OOP)  
 open method of window  
 opener property of window  
 operation  
 parse method of Date  
 PI property of Math  
 POSITIVE\_INFINITY property of Number  
 pow method of Math  
 property  
 round method of Math  
 search a string  
 sending a message to an object  
 setDate method of Date  
 setFullYear method of Date  
 setHours method of Date  
 setMilliSeconds method of Date  
 setMinutes method of Date  
 setMonth method of Date  
 setSeconds method of Date  
 setTime method of Date  
 setUTCDate method of Date  
 setUTCFullYear method of Date  
 setUTCHours method of Date  
 setUTCMilliseconds method of Date  
 setUTCMinutes method of Date  
 setUTCMonth method of Date  
 setUTCSeconds method of Date  
 sin method of Math  
 slice method of String  
 special characters  
 split method of String  
 sqrt method of Math  
 SQRT1\_2 property of Math  
 SQRT2 property of Math  
 strike method of String  
 string  
 string constant  
 string literal  
 String object  
 sub method of String  
 substr method of String  
 substring  
 substring method of String  
 sup method of String  
 tan method of Math  
 time  
 token  
 tokenization  
 toLocaleString method of Date  
 toLowerCase method of String  
 toString method of Date  
 toString method of String  
 toUpperCase method of String  
 toUTCString method of Date  
 unescape function  
 Unicode  
 user-defined type  
 UTC (Coordinated Universal Time)  
 UTC method of Date  
 valueOf method of Boolean  
 valueOf method of Date  
 valueOf method of Number  
 valueOf method of String  
 well-defined interfaces  
 window object  
 wrap in XHTML tags

## Self-Review Exercise

- 11.1** Fill in the blanks in each of the following statements:
- Because JavaScript uses objects to perform many tasks, JavaScript is commonly referred to as a(n) \_\_\_\_\_.

- b) All objects have \_\_\_\_\_ and exhibit \_\_\_\_\_.
- c) The methods of the \_\_\_\_\_ object allow you to perform many common mathematical calculations.
- d) Invoking (or calling) a method of an object is referred to as \_\_\_\_\_.
- e) String literals or string constants are written as a sequence of characters in \_\_\_\_\_ or \_\_\_\_\_.
- f) Indices for the characters in a string start at \_\_\_\_\_.
- g) String methods \_\_\_\_\_ and \_\_\_\_\_ search for the first and last occurrences of a substring in a `String`, respectively.
- h) The process of breaking a string into tokens is called \_\_\_\_\_.
- i) `String` method \_\_\_\_\_ formats a `String` as a hyperlink.
- j) Date and time processing can be performed based on the \_\_\_\_\_ or on World Time Standard's \_\_\_\_\_.
- k) Date method \_\_\_\_\_ 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.

## Answers to Self-Review Exercise

**11.1** a) object-based programming language. b) attributes, behaviors. c) Math. d) sending a message to the object. e) double quotation marks, single quotation marks. f) 0. g) `indexOf`, `lastIndexOf`. h) tokenization. i) link. j) computer's local time zone, Coordinated Universal Time (UTC). k) `parse`.

## Exercises

**11.2** Create a web page that contains four XHTML buttons. Each button, when clicked, should cause an alert dialog to display a different time or date in relation to the current time. Create a Now button that alerts the current time and date and a Yesterday button that alerts the time and date 24 hours ago. The other two buttons should alert the time and date ten years ago and one week from today.

**11.3** Write a script that tests as many of the `Math` library functions in Fig. 11.1 as you can. Exercise each of these functions by having your program display tables of return values for several argument values in an XHTML `textarea`.

**11.4** `Math` method `floor` may be used to round a number to a specific decimal place. For example, the statement

```
y = Math.floor(x * 10 + .5) / 10;
```

rounds `x` to the tenths position (the first position to the right of the decimal point). The statement

```
y = Math.floor(x * 100 + .5) / 100;
```

rounds `x` to the hundredths position (i.e., the second position to the right of the decimal point). Write a script that defines four functions to round a number `x` in various ways:

- a) `roundToInteger( number )`
- b) `roundToTenths( number )`
- c) `roundToHundredths( number )`
- d) `roundToThousandths( number )`

For each value read, your program should display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

- 11.5** Modify the solution to Exercise 11.4 to use `Math` method `round` instead of method `floor`.
- 11.6** Write a script that uses relational and equality operators to compare two `String`s input by the user through an XHTML form. Output in an XHTML `textarea` whether the first string is less than, equal to or greater than the second.
- 11.7** Write a script that uses random number generation to create sentences. Use four arrays of strings called `article`, `noun`, `verb` and `preposition`. Create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, concatenate it to the previous words in the sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period.
- The arrays should be filled as follows: the `article` array should contain the articles "the", "a", "one", "some" and "any"; the `noun` array should contain the nouns "boy", "girl", "dog", "town" and "car"; the `verb` array should contain the verbs "drove", "jumped", "ran", "walked" and "skipped"; the `preposition` array should contain the prepositions "to", "from", "over", "under" and "on".
- The program should generate 20 sentences to form a short story and output the result to an XHTML `textarea`. The story should begin with a line reading "Once upon a time..." and end with a line reading "THE END".
- 11.8** (*Limericks*) A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 11.7, write a script that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!
- 11.9** (*Pig Latin*) Write a script that encodes English-language phrases in pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm:
- To form a pig Latin phrase from an English-language phrase, tokenize the phrase into an array of words using `String` method `split`. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters "ay." Thus the word "jump" becomes "umpjay," the word "the" becomes "hetay" and the word "computer" becomes "omputercay." Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function `printLatinWord` should display each word. Each token (i.e., word in the sentence) is passed to method `printLatinWord` to print the pig Latin word. Enable the user to input the sentence through an XHTML form. Keep a running display of all the converted sentences in an XHTML `textarea`.
- 11.10** Write a script that inputs a telephone number as a string in the form (555) 555-5555. The script should use `String` method `split` to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. Display the area code in one text field and the seven-digit phone number in another text field.
- 11.11** Write a script that inputs a line of text, tokenizes it with `String` method `split` and outputs the tokens in reverse order.
- 11.12** Write a script that inputs text from an XHTML form and outputs it in uppercase and lowercase letters.
- 11.13** Write a script that inputs several lines of text and a search character and uses `String` method `indexOf` to determine the number of occurrences of the character in the text.
- 11.14** Write a script based on the program in Exercise 11.13 that inputs several lines of text and uses `String` method `indexOf` to determine the total number of occurrences of each letter of the

alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array, and print the values in tabular format in an XHTML `textarea` after the totals have been determined.

**11.15** Write a script that reads a series of strings and outputs in an XHTML `textarea` only those strings beginning with the character “b.”

**11.16** Write a script that reads a series of strings and outputs in an XHTML `textarea` only those strings ending with the characters “ed.”

**11.17** Write a script that inputs an integer code for a character and displays the corresponding character.

**11.18** Modify your solution to Exercise 11.17 so that it generates all possible three-digit codes in the range 000 to 255 and attempts to display the corresponding characters. Display the results in an XHTML `textarea`.

**11.19** Write your own version of the `String` method `indexOf` and use it in a script.

**11.20** Write your own version of the `String` method `lastIndexOf` and use it in a script.

**11.21** Write a program that reads a five-letter word from the user and produces all possible three-letter words that can be derived from the letters of the five-letter word. For example, the three-letter words produced from the word “bathe” include the commonly used words “ate,” “bat,” “bet,” “tab,” “hat,” “the” and “tea.” Output the results in an XHTML `textarea`.

**11.22** (*Printing Dates in Various Formats*) Dates are printed in several common formats. Write a script that reads a date from an XHTML form and creates a `Date` object in which to store it. Then use the various methods of the `Date` object that convert `Dates` into strings to display the date in several formats.

## Special Section: Challenging String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test the reader's understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. The reader should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

**11.23** (*Text Analysis*) The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare really wrote the works attributed to him. Some scholars believe there is substantial evidence indicating that Christopher Marlowe actually penned these masterpieces. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

- a) Write a script that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

To be, or not to be: that is the question:

contains one “a,” two “b’s,” no “c’s,” etc.

- b) Write a script that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc., appearing in the text. For example, the phrase

Whether 'tis nobler in the mind to suffer

contains

| Word length | Occurrences        |
|-------------|--------------------|
| 1           | 0                  |
| 2           | 2                  |
| 3           | 1                  |
| 4           | 2 (including 'tis) |
| 5           | 0                  |
| 6           | 2                  |
| 7           | 1                  |

- c) Write a script that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contain the word “to” three times, the word “be” twice, and the word “or” once. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

**11.24** (*Check Protection*) Computers are frequently employed in check-writing systems such as payroll and accounts payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of \$1 million. Incorrect amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers build controls into their systems to prevent erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection*.

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example:

```
1,230.60 (check amount)

12345678 (position numbers)
```

On the other hand, if the amount is less than \$1000, then several of the spaces would ordinarily be left blank. For example,

```
99.87

12345678
```

contains three blank spaces. If a check is printed with blank spaces, it is easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```
***99.87

12345678
```

Write a script that inputs a dollar amount to be printed on a check, then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing the amount.

**11.25** (*Writing the Word Equivalent of a Check Amount*) Continuing the discussion in the preceding exercise, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be written both in numbers and spelled out in words. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words.

Many computerized check-writing systems do not print the amount of the check in words. Perhaps the main reason for this omission is the fact that most high-level languages used in commercial applications do not contain adequate string-manipulation features. Another reason is that the logic for writing word equivalents of check amounts is somewhat involved.

Write a script that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 112.43 should be written as

ONE HUNDRED TWELVE and 43/100

**11.26** (*Morse Code*) Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit and a few special characters (e.g., period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space or, quite simply, by the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 11.18.

Write a script that reads an English-language phrase and encodes it in Morse code. Also write a program that reads a phrase in Morse code and converts the phrase into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

**11.27** (*Metric Conversion Program*) Write a script that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (e.g., centimeters, liters, grams, for the metric system and inches, quarts, pounds, for the English system) and should respond to simple questions such as

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Your program should recognize invalid conversions. For example, the question

```
"How many feet in 5 kilograms?"
```

is not a meaningful question because "feet" is a unit of length whereas "kilograms" is a unit of mass.

**11.28** (*Project: A Spell Checker*) Many popular word-processing software packages have built-in spell checkers.

In this project, you are asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. Use a computerized dictionary (if you have access to one) as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it is because we simply do not know the correct spelling, so we make a best guess. In some cases, it is because we transpose two letters (e.g., "defualt" instead of "default"). Sometimes we double-type a letter

| Character | Code  | Character     | Code   |
|-----------|-------|---------------|--------|
| A         | .-    | T             | -      |
| B         | -...  | U             | ..-    |
| C         | -.-.  | V             | ...-   |
| D         | -..   | W             | .--    |
| E         | .     | X             | -...   |
| F         | ...-. | Y             | -.--   |
| G         | --.   | Z             | --..   |
| H         | ....  |               |        |
| I         | ..    | <i>Digits</i> |        |
| J         | ----  | 1             | .----- |
| K         | -.-   | 2             | ...--- |
| L         | -.-.  | 3             | ....-  |
| M         | --    | 4             | .....  |
| N         | -.    | 5             | .....  |
| O         | ---   | 6             | -....  |
| P         | -.-.  | 7             | --...  |
| Q         | -.-.  | 8             | ---..  |
| R         | -.-   | 9             | ----.  |
| S         | ...   | 0             | ----   |

**Fig. 11.18** | Letters of the alphabet as expressed in international Morse code.

accidentally (e.g., “hanndy” instead of “handy”). Sometimes we type a nearby key instead of the one we intended (e.g., “biryhday” instead of “birthday”). And so on.

Design and implement a spell-checker application in JavaScript. Your program should maintain an array `wordList` of strings. Enable the user to enter these strings.

Your program should ask a user to enter a word. The program should then look up the word in the `wordList` array. If the word is present in the array, your program should print “Word is spelled correctly.”

If the word is not present in the array, your program should print “word is not spelled correctly.” Then your program should try to locate other words in `wordList` that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word “default” is a direct match to a word in `wordList`. Of course, this implies that your program will check all other single transpositions, such as “edfault,” “dfeault,”

“deafult,” “defalut” and “default.” When you find a new word that matches one in `wordList`, print that word in a message, such as “Did you mean “default?””

Implement other tests, such as replacing each double letter with a single letter and any other tests you can develop, to improve the value of your spell checker.

**11.29** (*Project: Crossword Puzzle Generator*) Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is suggested here as a string-manipulation project requiring substantial sophistication and effort.

There are many issues you must resolve to get even the simplest crossword puzzle generator program working. For example, how does one represent the grid of a crossword puzzle in the computer? Should one use a series of strings, or use double-subscripted arrays?

You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program?

The really ambitious reader will want to generate the clues portion of the puzzle, in which the brief hints for each across word and each down word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

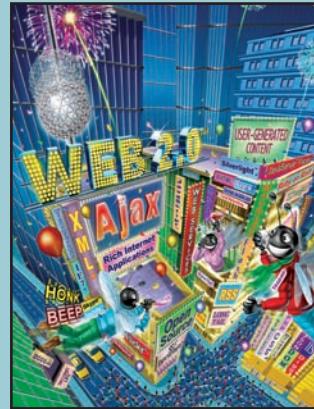
# 12

# Document Object Model (DOM): Objects and Collections

## OBJECTIVES

In this chapter you will learn:

- How to use JavaScript and the W3C Document Object Model to create dynamic web pages.
- The concept of DOM nodes and DOM trees.
- How to traverse, edit and modify elements in an XHTML document.
- How to change CSS styles dynamically.
- To create JavaScript animations.



*Our children may learn about heroes of the past. Our task is to make ourselves architects of the future.*

—Jomo Mzee Kenyatta

*Though leaves are many, the root is one.*

—William Butler Yeats

*The thing that impresses me most about America is the way parents obey their children.*

—Duke of Windsor

*Most of us become parents long before we have stopped being children.*

—Mignon McLaughlin

*To write it, it took three months; to conceive it three minutes; to collect the data in it—all my life.*

—F. Scott Fitzgerald

*Sibling rivalry is inevitable. The only sure way to avoid it is to have one child.*

—Nancy Samalin

**Outline**

- 12.1** Introduction
- 12.2** Modeling a Document: DOM Nodes and Trees
- 12.3** Traversing and Modifying a DOM Tree
- 12.4** DOM Collections
- 12.5** Dynamic Styles
- 12.6** Summary of the DOM Objects and Collections
- 12.7** Wrap-Up
- 12.8** Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 12.1 Introduction

In this chapter we introduce the **Document Object Model (DOM)**. The DOM gives you access to all the elements on a web page. Inside the browser, the whole web page—paragraphs, forms, tables, etc.—is represented in an **object hierarchy**. Using JavaScript, you can create, modify and remove elements in the page dynamically.

Previously, both Internet Explorer and Netscape had different versions of Dynamic HTML, which provided similar functionality to the DOM. However, while they provided many of the same capabilities, these two models were incompatible with each other. In an effort to encourage cross-browser websites, the W3C created the standardized Document Object Model. Firefox 2, Internet Explorer 7, and most other major browsers implement *most* of the features of the W3C DOM.

This chapter begins by formally introducing the concept of DOM nodes and DOM trees. We then discuss properties and methods of DOM nodes and cover additional methods of the `document` object. We also discuss how to dynamically change style properties, which enables you to create many types of effects, such as user-defined background colors and animations. Then, we present a diagram of the extensive object hierarchy, with explanations of the various objects and properties, and we provide links to websites with further information on the topic.



### Software Engineering Observation 12.1

*With the DOM, XHTML elements can be treated as objects, and many attributes of XHTML elements can be treated as properties of those objects. Then, objects can be scripted (through their id attributes) with JavaScript to achieve dynamic effects.*

## 12.2 Modeling a Document: DOM Nodes and Trees

As we saw in previous chapters, the document's `getElementById` method is the simplest way to access a specific element in a page. In this section and the next, we discuss more thoroughly the objects returned by this method.

The `getElementById` method returns objects called **DOM nodes**. Every element in an XHTML page is modeled in the web browser by a DOM node. All the nodes in a document make up the page's **DOM tree**, which describes the relationships among elements. Nodes are related to each other through child-parent relationships. An XHTML element inside another element is said to be a **child** of the containing element. The containing element is known as the **parent**. A node may have multiple children, but only one parent. Nodes with the same parent node are referred to as **siblings**.

Some browsers have tools that allow you to see a visual representation of the DOM tree of a document. When installing Firefox, you can choose to install a tool called the **DOM Inspector**, which allows you to view the DOM tree of an XHTML document. To inspect a document, Firefox users can access the **DOM Inspector** from the **Tools** menu of Firefox. If the DOM inspector is not in the menu, run the Firefox installer and choose **Custom** in the **Setup Type** screen, making sure the **DOM Inspector** box is checked in the **Optional Components** window.

Microsoft provides a **Developer Toolbar** for Internet Explorer that allows you to inspect the DOM tree of a document. The toolbar can be downloaded from Microsoft at [go.microsoft.com/fwlink/?LinkId=92716](http://go.microsoft.com/fwlink/?LinkId=92716). Once the toolbar is installed, restart the browser, then click the » icon at the right of the toolbar and choose **IE Developer Toolbar** from the menu. Figure 12.1 shows an XHTML document and its DOM tree displayed in Firefox's DOM Inspector and in IE's Web Developer Toolbar.

The XHTML document contains a few simple elements. We explain the example based on the Firefox DOM Inspector—the IE Toolbar displays the document with only minor differences. A node can be expanded and collapsed using the + and - buttons next to the node's name. Figure 12.1(b) shows all the nodes in the document fully expanded. The document node (shown as **#document**) at the top of the tree is called the **root node**, because it has no parent. Below the document node, the **HTML** node is indented from the document node to signify that the **HTML** node is a child of the **#document** node. The **HTML** node represents the **html** element (lines 7–24).

The **HEAD** and **BODY** nodes are siblings, since they are both children of the **HTML** node. The **HEAD** contains two **#comment** nodes, representing lines 5–6. The **TITLE** node

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 12.1: domtree.html -->
6 <!-- Demonstration of a document's DOM tree. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>DOM Tree Demonstration</title>
10 </head>
11 <body>
12 <h1>An XHTML Page</h1>
13 <p>This page contains some basic XHTML elements. We use the Firefox
14 DOM Inspector and the IE Developer Toolbar to view the DOM tree
15 of the document, which contains a DOM node for every element in
16 the document.</p>
17 <p>Here's a list:</p>
18
19 One
20 Two
21 Three
22
23 </body>
24 </html>
```

**Fig. 12.1** | Demonstration of a document's DOM tree. (Part I of 3.)

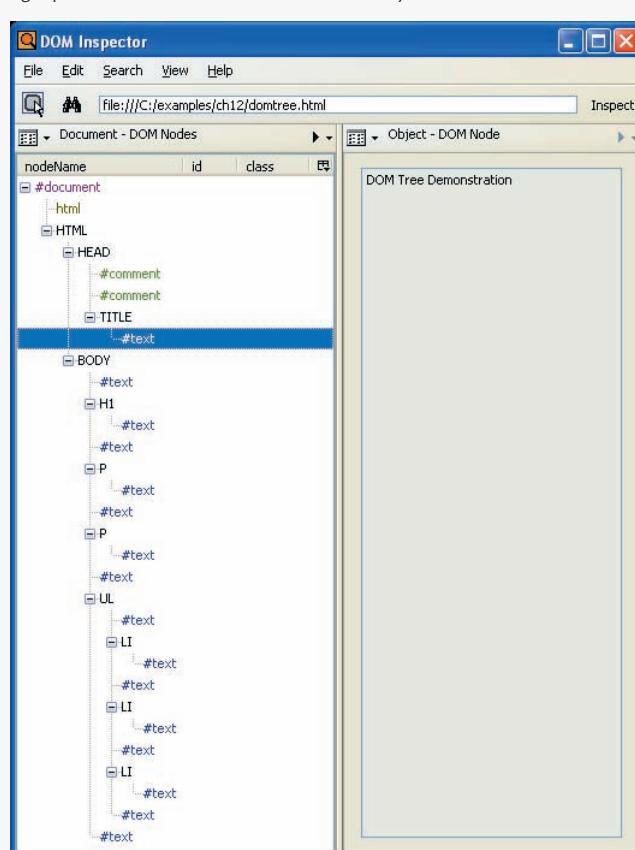
a) The XHTML document is rendered in Firefox.



Here's a list:

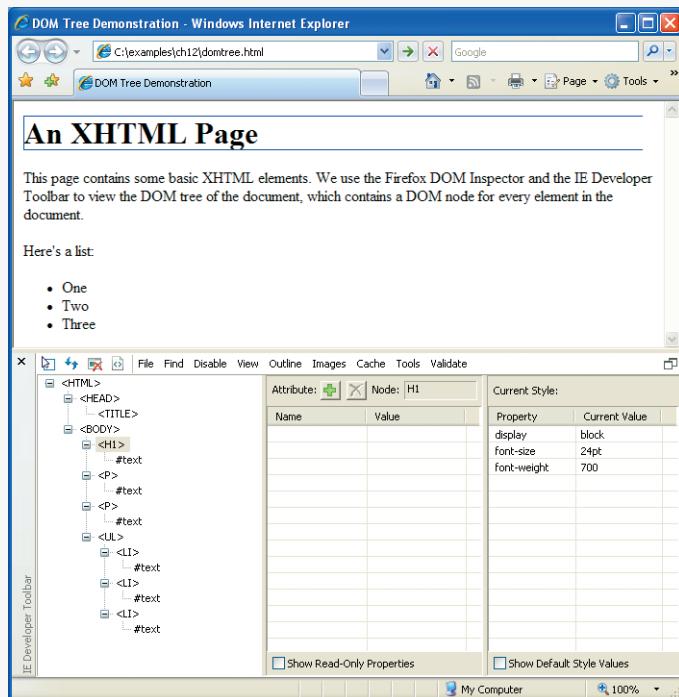
- ◆ One
- ◆ Two
- ◆ Three

b) The Firefox DOM inspector displays the document tree in the left panel. The right panel shows information about the currently selected node.



**Fig. 12.1** | Demonstration of a document's DOM tree. (Part 2 of 3.)

- c) The Internet Explorer Web Developer Toolbar displays much of the same information as the DOM inspector in Firefox in a panel at the bottom of the browser window.



**Fig. 12.1 |** Demonstration of a document's DOM tree. (Part 3 of 3.)

has a child text node (`#text`) containing the text **DOM Tree Demonstration**, visible in the right pane of the DOM inspector when the text node is selected. The **BODY** node contains nodes representing each of the elements in the page. Note that the **LI** nodes are children of the **UL** node, since they are nested inside it.

Also, notice that, in addition to the text nodes representing the text inside the body, paragraphs and list elements, a number of other text nodes appear in the document. These text nodes contain nothing but white space. When Firefox parses an XHTML document into a DOM tree, the white space between sibling elements is interpreted as text and placed inside text nodes. Internet Explorer ignores white space and does not convert it into empty text nodes. If you run this example on your own computer, you will notice that the **BODY** node has a `#comment` child node not present above in both the Firefox and Internet Explorer DOM trees. This is a result of the copyright line at the end of the posted file.

This section introduced the concept of DOM nodes and DOM trees. The next section discusses DOM nodes in more detail, discussing methods and properties of DOM nodes that allow you to modify the DOM tree of a document using JavaScript.

## 12.3 Traversing and Modifying a DOM Tree

The DOM gives you access to the elements of a document, allowing you to modify the contents of a page dynamically using event-driven JavaScript. This section introduces

properties and methods of all DOM nodes that enable you to traverse the DOM tree, modify nodes and create or delete content dynamically.

Figure 12.2 showcases some of the functionality of DOM nodes, as well as two additional methods of the `document` object. The program allows you to highlight, modify, insert and remove elements.

Lines 117–132 contain basic XHTML elements and content. Each element has an `id` attribute, which is also displayed at the beginning of the element in square brackets. For example, the `id` of the `h1` element in lines 117–118 is set to `bigheading`, and the heading text begins with `[bigheading]`. This allows the user to see the `id` of each element in the page. The body also contains an `h3` heading, several `p` elements, and an unordered list.

A `div` element (lines 133–162) contains the remainder of the XHTML body. Line 134 begins a `form` element, assigning the empty string to the required `action` attribute (because we're not submitting to a server) and returning `false` to the `onsubmit` attribute. When a form's `onsubmit` handler returns `false`, the navigation to the address specified in the `action` attribute is aborted. This allows us to modify the page using JavaScript event handlers without reloading the original, unmodified XHTML.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 12.2: dom.html -->
6 <!-- Basic DOM functionality. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Basic DOM Functionality</title>
10 <style type = "text/css">
11 h1, h3 { text-align: center;
12 font-family: tahoma, geneva, sans-serif }
13 p { margin-left: 5%;
14 margin-right: 5%;
15 font-family: arial, helvetica, sans-serif }
16 ul { margin-left: 10% }
17 a { text-decoration: none }
18 a:hover { text-decoration: underline }
19 .nav { width: 100%;
20 border-top: 3px dashed blue;
21 padding-top: 10px }
22 .highlighted { background-color: yellow }
23 .submit { width: 120px }
24 </style>
25 <script type = "text/javascript">
26 <!--
27 var currentNode; // stores the currently highlighted node
28 var idcount = 0; // used to assign a unique id to new elements
29
30 // get and highlight an element by its id attribute
31 function getById()
32 {

```

**Fig. 12.2** | Basic DOM functionality. (Part 1 of 8.)

```
33 var id = document.getElementById("gbi").value;
34 var target = document.getElementById(id);
35
36 if (target)
37 switchTo(target);
38 } // end function byId
39
40 // insert a paragraph element before the current element
41 // using the insertBefore method
42 function insert()
43 {
44 var newNode = createNewNode(
45 document.getElementById("ins").value);
46 currentNode.parentNode.insertBefore(newNode, currentNode);
47 switchTo(newNode);
48 } // end function insert
49
50 // append a paragraph node as the child of the current node
51 function appendNode()
52 {
53 var newNode = createNewNode(
54 document.getElementById("append").value);
55 currentNode.appendChild(newNode);
56 switchTo(newNode);
57 } // end function appendNode
58
59 // replace the currently selected node with a paragraph node
60 function replaceCurrent()
61 {
62 var newNode = createNewNode(
63 document.getElementById("replace").value);
64 currentNode.parentNode.replaceChild(newNode, currentNode);
65 switchTo(newNode);
66 } // end function replaceCurrent
67
68 // remove the current node
69 function remove()
70 {
71 if (currentNode.parentNode == document.body)
72 alert("Can't remove a top-level element.");
73 else
74 {
75 var oldNode = currentNode;
76 switchTo(oldNode.parentNode);
77 currentNode.removeChild(oldNode);
78 }
79 } // end function remove
80
81 // get and highlight the parent of the current node
82 function parent()
83 {
84 var target = currentNode.parentNode;
```

Fig. 12.2 | Basic DOM functionality. (Part 2 of 8.)

```
86 if (target != document.body)
87 switchTo(target);
88 else
89 alert("No parent.");
90 } // end function parent
91
92 // helper function that returns a new paragraph node containing
93 // a unique id and the given text
94 function createNewNode(text)
95 {
96 var newNode = document.createElement("p");
97 nodeId = "new" + idcount;
98 ++idcount;
99 newNode.id = nodeId;
100 text = "[" + nodeId + "] " + text;
101 newNode.appendChild(document.createTextNode(text));
102 return newNode;
103 } // end function createNewNode
104
105 // helper function that switches to a new currentNode
106 function switchTo(newNode)
107 {
108 currentNode.className = ""; // remove old highlighting
109 currentNode = newNode;
110 currentNode.className = "highlighted"; // highlight new node
111 document.getElementById("gbi").value = currentNode.id;
112 } // end function switchTo
113 // -->
114 </script>
115</head>
116<body onload = "currentNode = document.getElementById('bigheading')">
117 <h1 id = "bigheading" class = "highlighted">
118 [bigheading] DHTML Object Model</h1>
119 <h3 id = "smallheading">[smallheading] Element Functionality</h3>
120 <p id = "para1">[para1] The Document Object Model (DOM) allows for
121 quick, dynamic access to all elements in an XHTML document for
122 manipulation with JavaScript.</p>
123 <p id = "para2">[para2] For more information, check out the
124 "JavaScript and the DOM" section of Deitel's
125
126 [link] JavaScript Resource Center.</p>
127 <p id = "para3">[para3] The buttons below demonstrate:(list)</p>
128 <ul id = "list">
129 <li id = "item1">[item1] getElementById and parentNode
130 <li id = "item2">[item2] insertBefore and appendChild
131 <li id = "item3">[item3] replaceChild and removeChild
132
133 <div id = "nav" class = "nav">
134 <form onsubmit = "return false" action = "">
135 <table>
136 <tr>
137 <td><input type = "text" id = "gbi"
138 value = "bigheading" /></td>
```

**Fig. 12.2** | Basic DOM functionality. (Part 3 of 8.)

```

139 <td><input type = "submit" value = "Get By id"
140 onclick = "byId()" class = "submit" /></td>
141 </tr><tr>
142 <td><input type = "text" id = "ins" /></td>
143 <td><input type = "submit" value = "Insert Before"
144 onclick = "insert()" class = "submit" /></td>
145 </tr><tr>
146 <td><input type = "text" id = "append" /></td>
147 <td><input type = "submit" value = "Append Child"
148 onclick = "appendNode()" class = "submit" /></td>
149 </tr><tr>
150 <td><input type = "text" id = "replace" /></td>
151 <td><input type = "submit" value = "Replace Current"
152 onclick = "replaceCurrent()" class = "submit" /></td>
153 </tr><tr><td />
154 <td><input type = "submit" value = "Remove Current"
155 onclick = "remove()" class = "submit" /></td>
156 </tr><tr><td />
157 <td><input type = "submit" value = "Get Parent"
158 onclick = "parent()" class = "submit" /></td>
159 </tr>
160 </table>
161 </form>
162 </div>
163 </body>
164 </html>

```

a) This is the page when it first loads. It begins with the large heading highlighted.

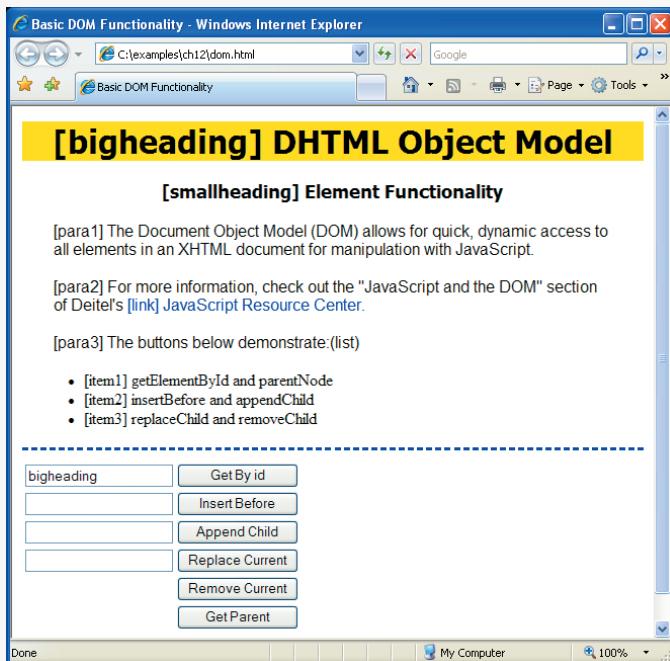
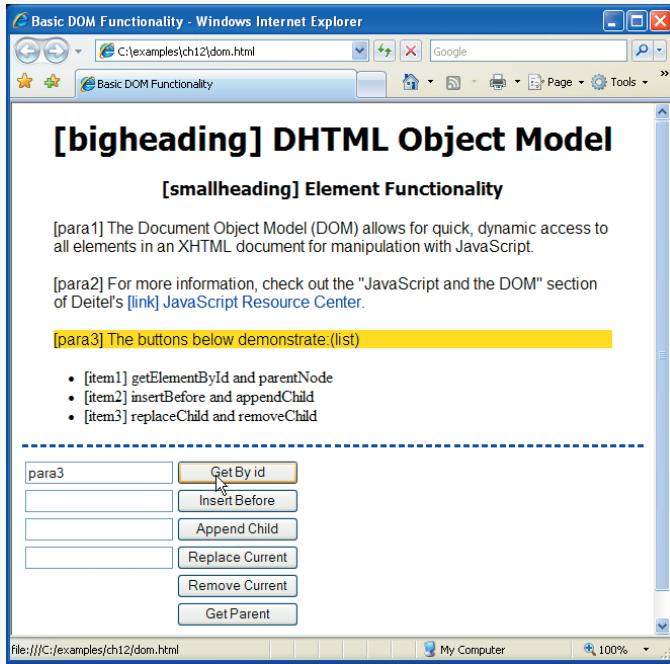


Fig. 12.2 | Basic DOM functionality. (Part 4 of 8.)

b) This is the document after using the Get By id button to select para3.



c) This is the document after inserting a new paragraph before the selected one.

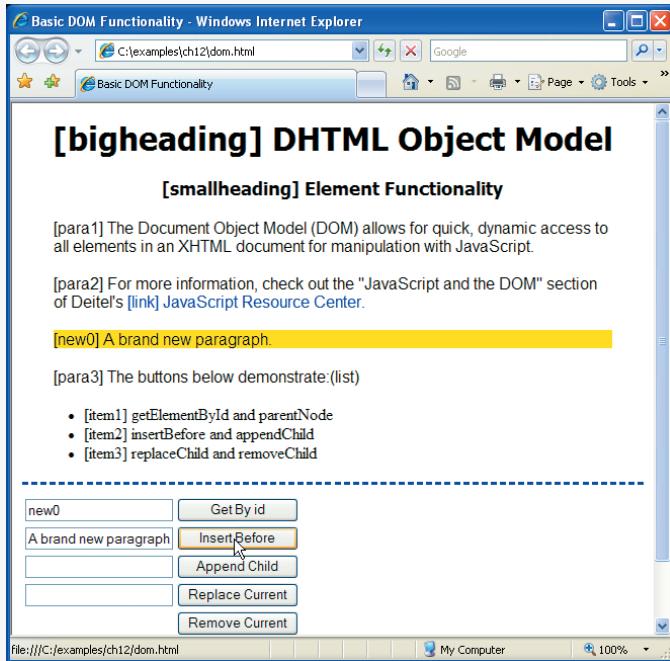
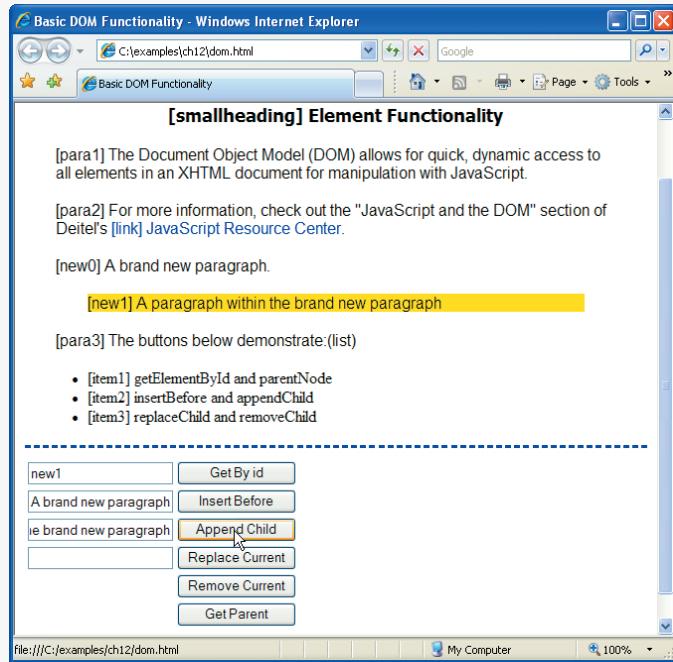
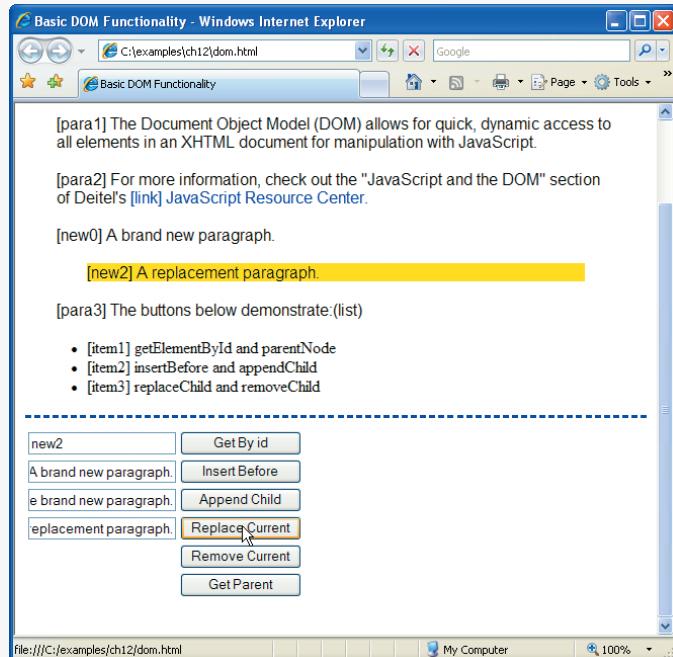


Fig. 12.2 | Basic DOM functionality. (Part 5 of 8.)

d) Using the Append Child button, a child paragraph is created.

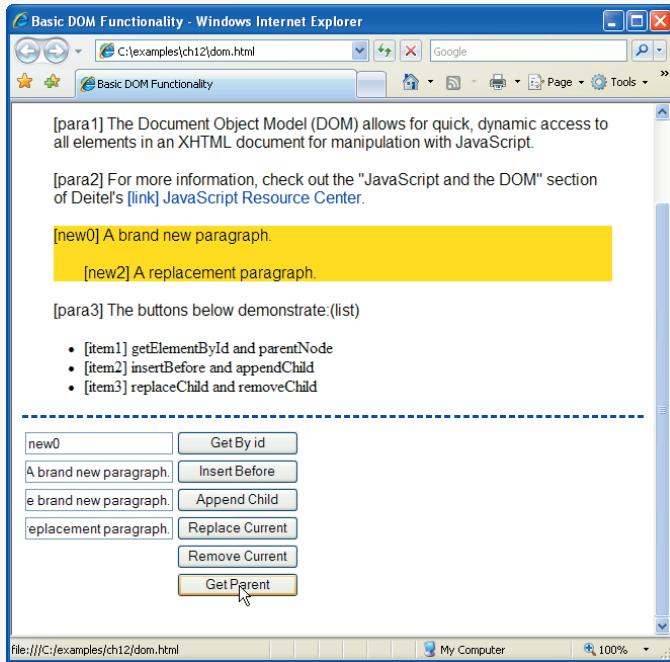


e) The selected paragraph is replaced with a new one.

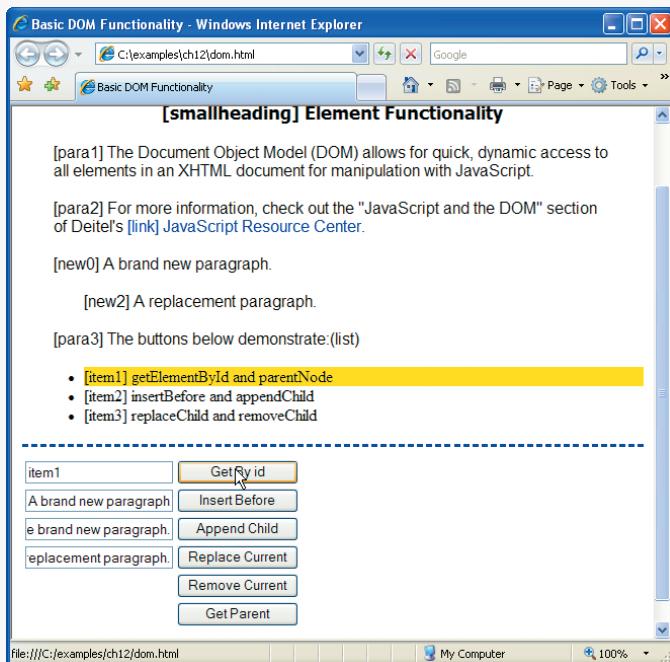


**Fig. 12.2** | Basic DOM functionality. (Part 6 of 8.)

f) The Get Parent button gets the parent of the selected node.

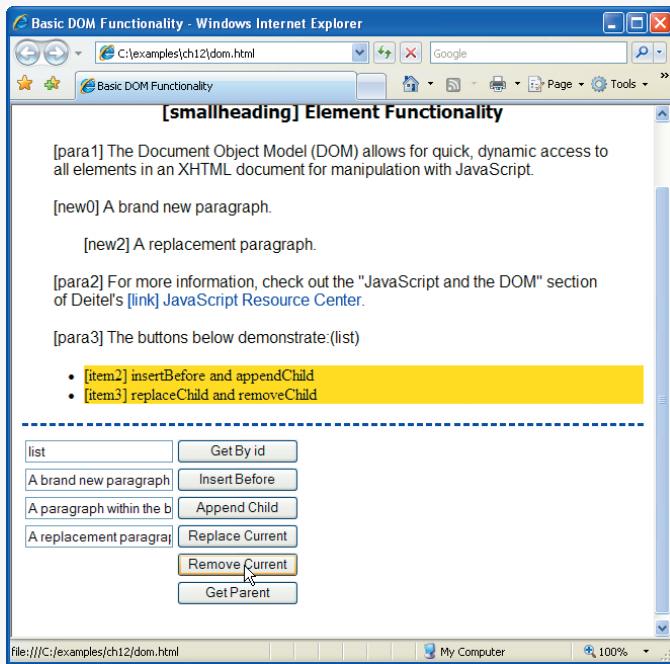


g) Now we select the first list item.



**Fig. 12.2** | Basic DOM functionality. (Part 7 of 8.)

h) The Remove Current button removes the current node and selects its parent.



**Fig. 12.2 |** Basic DOM functionality. (Part 8 of 8.)

A table (lines 135–160) contains the controls for modifying and manipulating the elements on the page. Each of the six buttons calls its own event-handling function to perform the action described by its `value`.

The JavaScript code begins by declaring two variables. The variable `currentNode` (line 27) keeps track of the currently highlighted node, because the functionality of the buttons depends on which node is currently selected. The body's `onload` attribute (line 116) initializes `currentNode` to the `h1` element with `id bigheading`. Variable `idcount` (line 28) is used to assign a unique `id` to any new elements that are created. The remainder of the JavaScript code contains event handling functions for the XHTML buttons and two helper functions that are called by the event handlers. We now discuss each button and its corresponding event handler in detail.

#### ***Finding and Highlighting an Element Using `getElementById` and `className`***

The first row of the table (lines 136–141) allows the user to enter the `id` of an element into the text field (lines 137–138) and click the `Get By Id` button (lines 139–140) to find and highlight the element, as shown in Fig. 12.2(b) and (g). The `onclick` attribute sets the button's event handler to `byId`.

The `byId` function is defined in lines 31–38. Line 33 uses `getElementById` to assign the contents of the text field to variable `id`. Line 34 uses `getElementById` again to find the element whose `id` attribute matches the contents of variable `id`, and assign it to variable `target`. If an element is found with the given `id`, `getElementById` returns an object rep-

resenting that element. If no element is found, `getElementById` returns `null`. Line 36 checks whether `target` is an object—recall that any object used as a boolean expression is `true`, while `null` is `false`. If `target` evaluates to `true`, line 37 calls the `switchTo` function with `target` as its argument.

The `switchTo` function, defined in lines 106–112, is used throughout the program to highlight a new element in the page. The current element is given a yellow background using the style class `highlighted`, defined in line 22. Line 108 sets the current node's `className` property to the empty string. The `className` property allows you to change an XHTML element's `class` attribute. In this case, we clear the `class` attribute in order to remove the `highlighted` class from the `currentNode` before we highlight the new one.

Line 109 assigns the `newNode` object (passed into the function as a parameter) to variable `currentNode`. Line 110 adds the `highlighted` style class to the new `currentNode` using the `className` property.

Finally, line 111 uses the `id` property to assign the current node's `id` to the input field's `value` property. Just as `className` allows access to an element's `class` attribute, the `id` property controls an element's `id` attribute. While this isn't necessary when `switchTo` is called by `byId`, we will see shortly that other functions call `switchTo`. This line makes sure that the text field's value is consistent with the currently selected node's `id`. Having found the new element, removed the highlighting from the old element, updated the `currentNode` variable and highlighted the new element, the program has finished selecting a new node by a user-entered `id`.

### *Creating and Inserting Elements Using `insertBefore` and `appendChild`*

The next two table rows allow the user to create a new element and insert it before the current node or as a child of the current node. The second row (lines 141–145) allows the user to enter text into the text field and click the *Insert Before* button. The text is placed in a new paragraph element, which is then inserted into the document before the currently selected element, as in Fig. 12.2(c). The button in lines 143–144 calls the `insert` function, defined in lines 42–48.

Lines 44–45 call the function `createNewNode`, passing it the value of the input field (whose `id` is `ins`) as an argument. Function `createNewNode`, defined in lines 94–103, creates a paragraph node containing the text passed to it. Line 96 creates a `p` element using the `document` object's `createElement` method. The `createElement` method creates a new DOM node, taking the tag name as an argument. Note that while `createElement` *creates* an element, it does not *insert* the element on the page.

Line 97 creates a unique `id` for the new element by concatenating "new" and the value of `idcount` before incrementing `idcount` in line 98. Line 99 assigns the `id` to the new element. Line 100 concatenates the element's `id` in square brackets to the beginning of `text` (the parameter containing the paragraph's text).

Line 101 introduces two new methods. The `document`'s `createTextNode` method creates a node that can contain only text. Given a string argument, `createTextNode` inserts the string into the text node. In line 101, we create a new text node containing the contents of variable `text`. This new node is then used (still in line 101) as the argument to the `appendChild` method, which is called on the paragraph node. Method `appendChild` is called on a parent node to insert a child node (passed as an argument) after any existing children.

After the `p` element is created, line 102 returns the node to the calling function `insert`, where it is assigned to variable `newNode` in lines 44–45. Line 46 inserts the newly created node before the currently selected node. The `parentNode` property of any DOM node contains the node's parent. In line 46, we use the `parentNode` property of `currentNode` to get its parent.

We call the `insertBefore` method (line 46) on the parent with `newNode` and `currentNode` as its arguments to insert `newNode` as a child of the parent directly before `currentNode`. The general syntax of the `insertBefore` method is

```
parent.insertBefore(newChild, existingChild);
```

The method is called on a parent with the new child and an existing child as arguments. The node `newChild` is inserted as a child of `parent` directly before `existingChild`. Line 47 uses the `switchTo` function (discussed earlier in this section) to update the `currentNode` to the newly inserted node and highlight it in the XHTML page.

The third table row (lines 145–149) allows the user to append a new paragraph node as a child of the current element, demonstrated in Fig. 12.2(d). This features uses a similar procedure to the `insertBefore` functionality. Lines 53–54 in function `appendNode` create a new node, line 55 inserts it as a child of the current node, and line 56 uses `switchTo` to update `currentNode` and highlight the new node.

### ***Replacing and Removing Elements Using `replaceChild` and `removeChild`***

The next two table rows (lines 149–156) allow the user to replace the current element with a new `p` element or simply remove the current element. Lines 150–152 contain a text field and a button that replaces the currently highlighted element with a new paragraph node containing the text in the text field. This feature is demonstrated in Fig. 12.2(e).

The button in lines 151–152 calls function `replaceCurrent`, defined in lines 60–66. Lines 62–63 call `createNewNode`, in the same way as in `insert` and `appendNode`, getting the text from the correct input field. Line 64 gets the parent of `currentNode`, then calls the `replaceChild` method on the parent. The `replaceChild` method works as follows:

```
parent.replaceChild(newChild, oldChild);
```

The `parent`'s `replaceChild` method inserts `newChild` into its list of children in place of `oldChild`.

The Remove Current feature, shown in Fig. 12.2(h), removes the current element entirely and highlights the parent. No text field is required because a new element is not being created. The button in lines 154–155 calls the `remove` function, defined in lines 69–79. If the node's parent is the body element, line 72 alerts an error—the program does not allow the entire body element to be selected. Otherwise, lines 75–77 remove the current element. Line 75 stores the old `currentNode` in variable `oldNode`. We do this to maintain a reference to the node to be removed after we've changed the value of `currentNode`. Line 76 calls `switchTo` to highlight the parent node.

Line 77 uses the `removeChild` method to remove the `oldNode` (a child of the new `currentNode`) from its place in the XHTML document. In general,

```
parent.removeChild(child);
```

looks in `parent`'s list of children for `child` and removes it.

The final button (lines 157–158) selects and highlights the parent element of the currently highlighted element by calling the `parent` function, defined in lines 82–90. Function `parent` simply gets the parent node (line 84), makes sure it is not the body element, (line 86) and calls `switchTo` to highlight it (line 87). Line 89 alerts an error if the parent node is the body element. This feature is shown in Fig. 12.2(f).

This section introduced the basics of DOM tree traversal and manipulation. Next, we introduce the concept of collections, which give you access to multiple elements in a page.

## 12.4 DOM Collections

Included in the Document Object Model is the notion of **collections**, which are groups of related objects on a page. DOM collections are accessed as properties of DOM objects such as the document object or a DOM node. The document object has properties containing the **images collection**, **links collection**, **forms collection** and **anchors collection**. These collections contain all the elements of the corresponding type on the page. Figure 12.3 gives an example that uses the `links` collection to extract all of the links on a page and display them together at the bottom of the page.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 12.3: collections.html -->
6 <!-- Using the links collection. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Using Links Collection</title>
10 <style type = "text/css">
11 body { font-family: arial, helvetica, sans-serif }
12 h1 { font-family: tahoma, geneva, sans-serif;
13 text-align: center }
14 p { margin: 5% }
15 p a { color: #aa0000 }
16 .links { font-size: 14px;
17 text-align: justify;
18 margin-left: 10%;
19 margin-right: 10% }
20 .link a { text-decoration: none }
21 .link a:hover { text-decoration: underline }
22 </style>
23 <script type = "text/javascript">
24 <!--
25 function processlinks()
26 {
27 var linkslist = document.links; // get the document's links
28 var contents = "Links in this page:\n
| ";
29
30 // concatenate each link to contents
31 for (var i = 0; i < linkslist.length; i++)
32 {

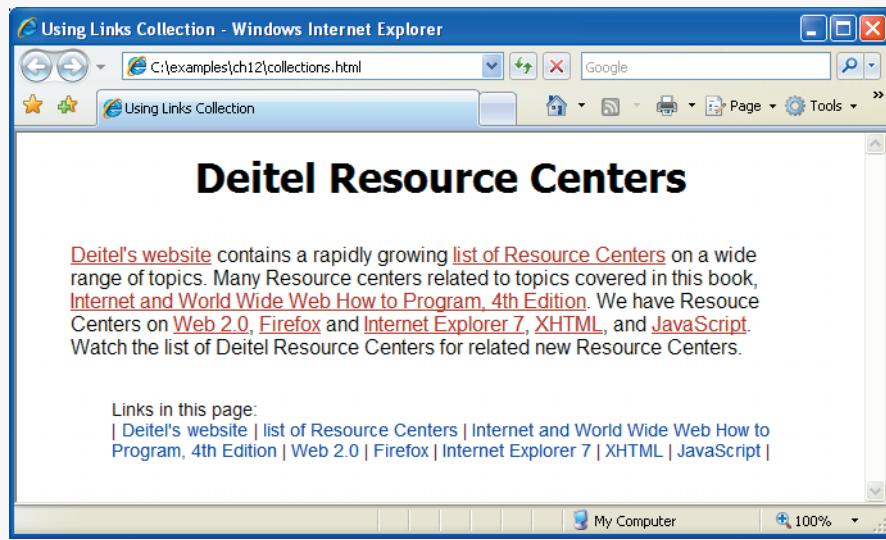
```

**Fig. 12.3** | Using the `links` collection. (Part 1 of 2.)

```

33 var currentLink = linkslist[i];
34 contents += "" +
35 currentLink.innerHTML.link(currentLink.href) +
36 " | ";
37 } // end for
38
39 document.getElementById("links").innerHTML = contents;
40 } // end function processlinks
41 // --
42 </script>
43 </head>
44 <body onload = "processlinks()">
45 <h1>Deitel Resource Centers</h1>
46 <p>Deitel's website contains
47 a rapidly growing
48 list of
49 Resource Centers on a wide range of topics. Many Resource
50 centers related to topics covered in this book,
51 Internet and World Wide
52 Web How to Program, 4th Edition. We have Resouce Centers on
53 Web 2.0,
54 Firefox and
55 Internet Explorer 7,
56 XHTML, and
57 JavaScript.
58 Watch the list of Deitel Resource Centers for related new
59 Resource Centers.</p>
60 <div id = "links" class = "links"></div>
61 </body>
62 </html>

```



**Fig. 12.3** | Using the `links` collection. (Part 2 of 2.)

The XHTML body contains a paragraph (lines 46–59) with links at various places in the text and an empty div (line 60) with id `links`. The body's `onload` attribute specifies that the `processlinks` method is called when the body finishes loading.

Method `processlinks` declares variable `linkslist` (line 27) to store the document's `links` collection, which is accessed as the `links` property of the document object. Line 28 creates the string (`contents`) that will contain all the document's links, to be inserted into the `links` div later. Line 31 begins a `for` statement to iterate through each link. To find the number of elements in the collection, we use the collection's **`length` property**.

Line 33 inside the `for` statement creates a variable (`currentlink`) that stores the current link. Note that we can access the collection stored in `linkslist` using indices in square brackets, just as we did with arrays. DOM collections are stored in objects which have only one property and two methods—the `length` property, the **`item` method** and the **`namedItem` method**. The `item` method—an alternative to the square bracketed indices—can be used to access specific elements in a collection by taking an index as an argument. The `namedItem` method takes a name as a parameter and finds the element in the collection, if any, whose `id` attribute or `name` attribute matches it.

Lines 34–36 add a `span` element to the `contents` string containing the current link. Recall that the `link` method of a string object returns the string as a link to the URL passed to the method. Line 35 uses the `link` method to create an `a` (anchor) element containing the proper text and `href` attribute.

Notice that variable `currentLink` (a DOM node representing an `a` element) has a specialized **`href` property** to refer to the link's `href` attribute. Many types of XHTML elements are represented by special types of nodes that extend the functionality of a basic DOM node. Line 39 inserts the contents into the empty div with id "links" (line 60) in order to show all the links on the page in one location.

Collections allow easy access to all elements of a single type in a page. This is useful for gathering elements into one place and for applying changes across an entire page. For example, the `forms` collection could be used to disable all form inputs after a submit button has been pressed to avoid multiple submissions while the next page loads. The next section discusses how to dynamically modify CSS styles using JavaScript and DOM nodes.

## 12.5 Dynamic Styles

An element's style can be changed dynamically. Often such a change is made in response to user events, which we discuss in Chapter 13. Such style changes can create many effects, including mouse hover effects, interactive menus, and animations. Figure 12.4 is a simple example that changes the `background-color` style property in response to user input.

```

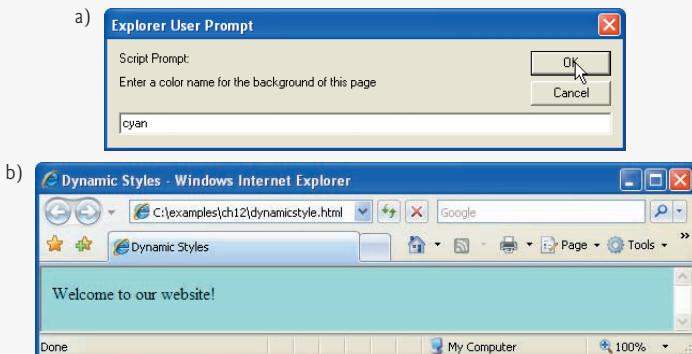
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 12.4: dynamicstyle.html -->
6 <!-- Dynamic styles. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
```

Fig. 12.4 | Dynamic styles. (Part I of 2.)

```

9 <title>Dynamic Styles</title>
10 <script type = "text/javascript">
11 <!--
12 function start()
13 {
14 var inputColor = prompt("Enter a color name for the " +
15 "background of this page", "");
16 document.body.style.backgroundColor = inputColor;
17 } // end function start
18 // -->
19 </script>
20 </head>
21 <body id = "body" onload = "start()">
22 <p>Welcome to our website!</p>
23 </body>
24 </html>

```



**Fig. 12.4** | Dynamic styles. (Part 2 of 2.)

Function `start` (lines 12–17) prompts the user to enter a color name, then sets the background color to that value. [Note: An error occurs if the value entered is not a valid color. See Appendix B, XHTML Colors, for further information.] We refer to the background color as `document.body.style.backgroundColor`—the **body** property of the document object refers to the body element. We then use the `style` property (a property of most XHTML elements) to set the `background-color` CSS property. This is referred to as `backgroundColor` in JavaScript—the hyphen is removed to avoid confusion with the subtraction (-) operator. This naming convention is consistent for most CSS properties. For example, `borderWidth` correlates to the `border-width` CSS property, and `fontFamily` correlates to the `font-family` CSS property. In general, CSS properties are accessed in the format `node.style.property`.

Figure 12.5 introduces the `setInterval` and `clearInterval` methods of the `window` object, combining them with dynamic styles to create animated effects. This example is a basic image viewer that allows you to select a Deitel book cover and view it in a larger size. When one of the thumbnail images on the right is clicked, the larger version grows from the top-left corner of the main image area.

The `body` (lines 66–85) contains two `div` elements, both floated `left` using styles defined in lines 14 and 17 in order to present them side by side. The left `div` contains the

full-size image `iw3htp4.jpg`, the cover of this book, which appears when the page loads. The right `div` contains six thumbnail images which respond to the click event by calling the `display` method and passing it the filename of the corresponding full-size image.

The `display` function (lines 46–62) dynamically updates the image in the left `div` to the one corresponding to the user's click. Lines 48–49 prevent the rest of the function from executing if `interval` is defined (i.e., an animation is in progress.) Line 51 gets the left `div` by its `id`, `imgCover`. Line 52 creates a new `img` element. Lines 53–55 set its `id` to `imgCover`, set its `src` to the correct image file in the `fullsize` directory, and set its required `alt` attribute. Lines 56–59 do some additional initialization before beginning the animation in line 61. To create the growing animation effect, lines 57–58 set the image `width` and `height` to 0. Line 59 replaces the current `bigImage` node with `newNode` (created in line 52), and line 60 sets `count`, the variable that controls the animation, to 0.

Line 61 introduces the `window` object's **`setInterval` method**, which starts the animation. This method takes two parameters—a statement to execute repeatedly, and an integer specifying how often to execute it, in milliseconds. We use `setInterval` to call

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 12.5: coverviewer.html -->
6 <!-- Dynamic styles used for animation. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Deitel Book Cover Viewer</title>
10 <style type = "text/css">
11 .thumbs { width: 192px;
12 height: 370px;
13 padding: 5px;
14 float: left }
15 .mainimg { width: 289px;
16 padding: 5px;
17 float: left }
18 .imgCover { height: 373px }
19 img { border: 1px solid black }
20 </style>
21 <script type = "text/javascript">
22 <!--
23 var interval = null; // keeps track of the interval
24 var speed = 6; // determines the speed of the animation
25 var count = 0; // size of the image during the animation
26
27 // called repeatedly to animate the book cover
28 function run()
29 {
30 count += speed;
31
32 // stop the animation when the image is large enough
33 if (count >= 375)
34 {

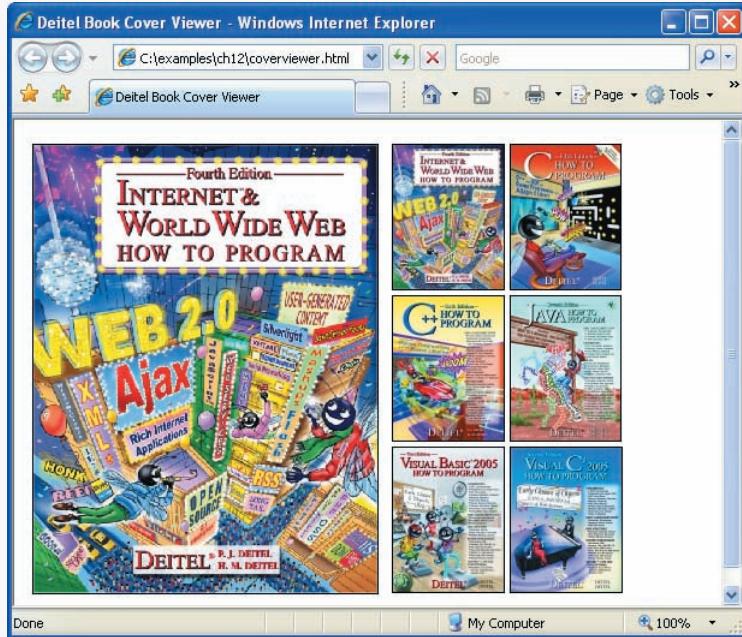
```

Fig. 12.5 | Dynamic styles used for animation. (Part I of 4.)

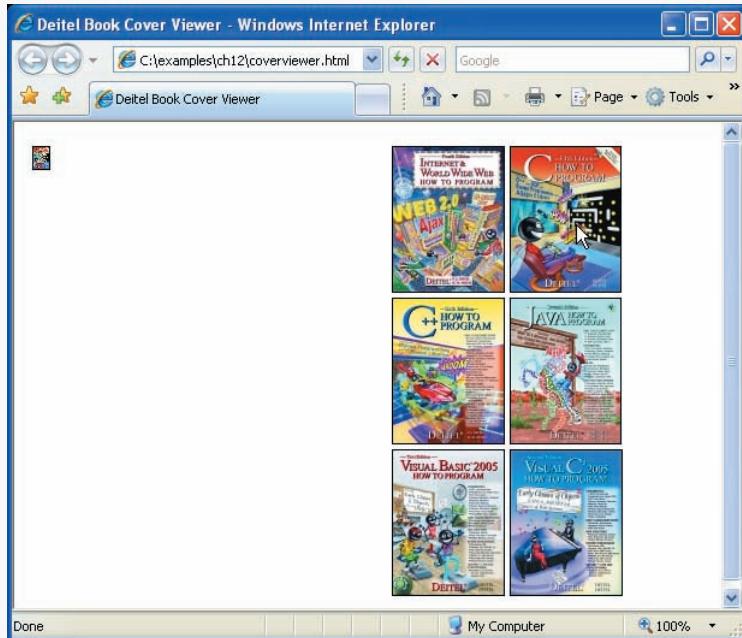
```
35 window.clearInterval(interval);
36 interval = null;
37 } // end if
38
39 var bigImage = document.getElementById("imgCover");
40 bigImage.style.width = .7656 * count + "px";
41 bigImage.style.height = count + "px";
42 } // end function run
43
44 // inserts the proper image into the main image area and
45 // begins the animation
46 function display(imgfile)
47 {
48 if (interval)
49 return;
50
51 var bigImage = document.getElementById("imgCover");
52 var newNode = document.createElement("img");
53 newNode.id = "imgCover";
54 newNode.src = "fullsize/" + imgfile;
55 newNode.alt = "Large image";
56 newNode.className = "imgCover";
57 newNode.style.width = "0px";
58 newNode.style.height = "0px";
59 bigImage.parentNode.replaceChild(newNode, bigImage);
60 count = 0; // start the image at size 0
61 interval = window.setInterval("run()", 10); // animate
62 } // end function display
63 // -->
64 </script>
65 </head>
66 <body>
67 <div id = "mainimg" class = "mainimg">
68 <img id = "imgCover" src = "fullsize/iw3htp4.jpg"
69 alt = "Full cover image" class = "imgCover" />
70 </div>
71 <div id = "thumbs" class = "thumbs" >
72 <img src = "thumbs/iw3htp4.jpg" alt = "iw3htp4"
73 onclick = "display('iw3htp4.jpg')" />
74 <img src = "thumbs/chtp5.jpg" alt = "chtp5"
75 onclick = "display('chtp5.jpg')" />
76 <img src = "thumbs/cpphtp6.jpg" alt = "cpphtp6"
77 onclick = "display('cpphtp6.jpg')" />
78 <img src = "thumbs/jhtp7.jpg" alt = "jhtp7"
79 onclick = "display('jhtp7.jpg')" />
80 <img src = "thumbs/vbhtp3.jpg" alt = "vbhtp3"
81 onclick = "display('vbhtp3.jpg')" />
82 <img src = "thumbs/vcsharphtp2.jpg" alt = "vcsharphtp2"
83 onclick = "display('vcsharphtp2.jpg')" />
84 </div>
85 </body>
86 </html>
```

Fig. 12.5 | Dynamic styles used for animation. (Part 2 of 4.)

a) The cover viewer page loads with the cover of this book.

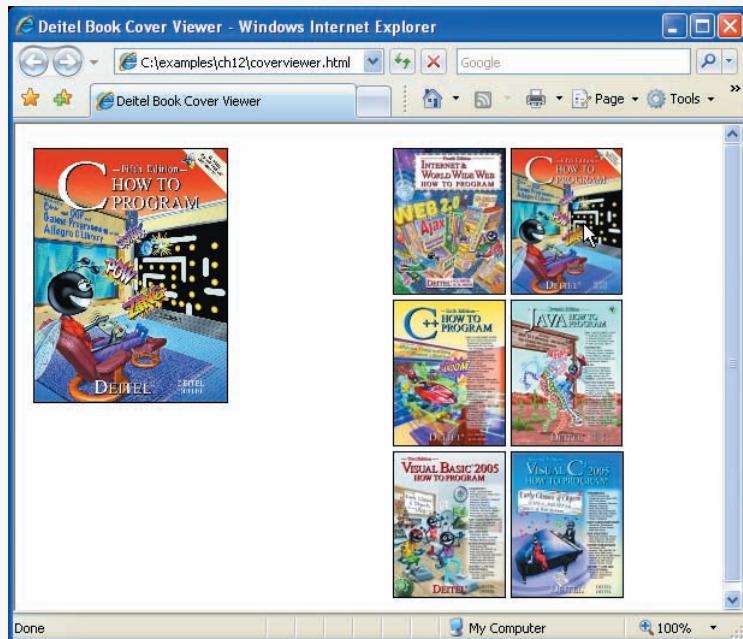


b) When the user clicks the thumbnail of C How to Program, the full-size image begins growing from the top-left corner of the window.

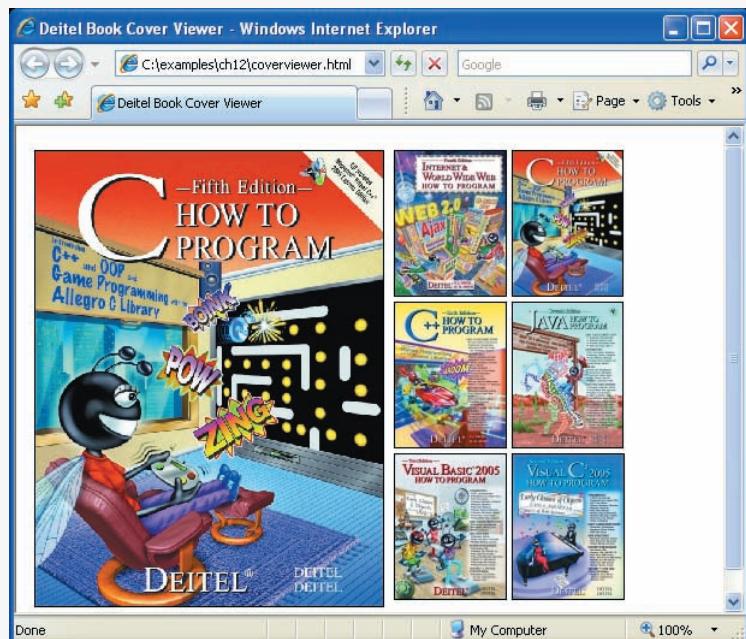


**Fig. 12.5** | Dynamic styles used for animation. (Part 3 of 4.)

c) The cover continues to grow.



d) The animation finishes when the cover reaches its full size.



**Fig. 12.5** | Dynamic styles used for animation. (Part 4 of 4.)

function run every 10 milliseconds. The `setInterval` method returns a unique identifier to keep track of that particular interval—we assign this identifier to the variable `interval`. We use this identifier to stop the animation when the image has finished growing.

The `run` function, defined in lines 28–42, increases the height of the image by the value of `speed` and updates its width accordingly to keep the aspect ratio consistent. Because the `run` function is called every 10 milliseconds, this increase happens repeatedly to create an animated growing effect. Line 30 adds the value of `speed` (declared and initialized to 6 in line 24) to `count`, which keeps track of the animation's progress and dictates the current size of the image. If the image has grown to its full height (375), line 35 uses the window's `clearInterval` method to stop the repetitive calls of the `run` method. We pass to `clearInterval` the interval identifier (stored in `interval`) that `setInterval` created in line 61. Although it seems unnecessary in this script, this identifier allows the script to keep track of multiple intervals running at the same time and to choose which interval to stop when calling `clearInterval`.

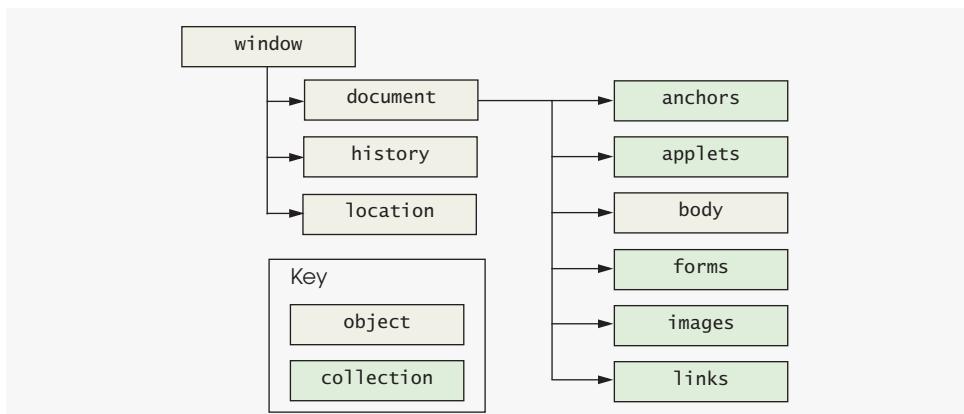
Line 39 gets the image and lines 40–41 set its `width` and `height` CSS properties. Note that line 40 multiplies `count` by a scaling factor of .7656 in order to keep the ratio of the image's dimensions consistent with the actual dimensions of the image. Run the code example and click on a thumbnail image to see the full animation effect.

This section demonstrated the concept of dynamically changing CSS styles using JavaScript and the DOM. We also discussed the basics of how to create scripted animations using `setInterval` and `clearInterval`.

## 12.6 Summary of the DOM Objects and Collections

As you have seen in the preceding sections, the objects and collections in the W3C DOM give you flexibility in manipulating the elements of a web page. We have shown how to access the objects in a page, how to access the objects in a collection, and how to change element styles dynamically.

The W3C DOM allows you to access every element in an XHTML document. Each element in a document is represented by a separate object. The diagram in Fig. 12.6 shows many of the important objects and collections provided by the W3C DOM. Figure 12.7 provides a brief description of each object and collection in Fig. 12.6.



**Fig. 12.6** | W3C Document Object Model.

| Object or collection | Description                                                                                                                                                                                                                                                    |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Objects</i>       |                                                                                                                                                                                                                                                                |
| window               | Represents the browser window and provides access to the document object contained in the window. Also contains history and location objects.                                                                                                                  |
| document             | Represents the XHTML document rendered in a window. The document object provides access to every element in the XHTML document and allows dynamic modification of the XHTML document. Contains several collections for accessing all elements of a given type. |
| body                 | Provides access to the body element of an XHTML document.                                                                                                                                                                                                      |
| history              | Keeps track of the sites visited by the browser user. The object provides a script programmer with the ability to move forward and backward through the visited sites.                                                                                         |
| location             | Contains the URL of the rendered document. When this object is set to a new URL, the browser immediately navigates to the new location.                                                                                                                        |
| <i>Collections</i>   |                                                                                                                                                                                                                                                                |
| anchors              | Collection contains all the anchor elements ( <code>a</code> ) that have a name or id attribute. The elements appear in the collection in the order in which they were defined in the XHTML document.                                                          |
| forms                | Contains all the <code>form</code> elements in the XHTML document. The elements appear in the collection in the order in which they were defined in the XHTML document.                                                                                        |
| images               | Contains all the <code>img</code> elements in the XHTML document. The elements appear in the collection in the order in which they were defined in the XHTML document.                                                                                         |
| links                | Contains all the anchor elements ( <code>a</code> ) with an <code>href</code> property. The elements appear in the collection in the order in which they were defined in the XHTML document.                                                                   |

**Fig. 12.7** | Objects and collections in the W3C Document Object Model.

For a complete reference on the W3C Document Object Model, see the DOM Level 3 recommendation from the W3C at <http://www.w3.org/TR/DOM-Level-3-Core/>. The DOM Level 2 HTML Specification (the most recent HTML DOM standard), available at <http://www.w3.org/TR/DOM-Level-2-HTML/>, describes additional DOM functionality specific to HTML, such as objects for various types of XHTML elements. Keep in mind that not all web browsers implement all features included in the specification.

## 12.7 Wrap-Up

This chapter discussed the Document Object Model, which provides access to a web page's elements. We described the child-parent relationships that exist between a docu-

ment's nodes, and provided visual representations of a DOM tree to illustrate how elements in a web page are related. We introduced several properties and methods that allow us to manipulate nodes, and described groups of related objects on a page, called collections. Finally, we discussed how to dynamically modify content in a web page, adding interactivity and animation to the existing site. In the next chapter, we discuss JavaScript events, which enable JavaScript programs to process user interactions with a web page.

## 12.8 Web Resources

[www.deitel.com/javascript/](http://www.deitel.com/javascript/)

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks, tutorials and more. Check out the section specifically dedicated to the Document Object Model. Be sure to visit the related Resource Centers on XHTML ([www.deitel.com/xhtml/](http://www.deitel.com/xhtml/)) and CSS 2.1 ([www.deitel.com/css21/](http://www.deitel.com/css21/)).

---

## Summary

### Section 12.1 Introduction

- The Document Object Model gives you access to all the elements on a web page. Using JavaScript, you can create, modify and remove elements in the page dynamically.

### Section 12.2 Modeling a Document: DOM Nodes and Trees

- The `getElementById` method returns objects called DOM nodes. Every element in an XHTML page is modeled in the web browser by a DOM node.
- All the nodes in a document make up the page's DOM tree, which describes the relationships among elements.
- Nodes are related to each other through child-parent relationships. An XHTML element inside another element is said to be a child of the containing element. The containing element is known as the parent. A node may have multiple children, but only one parent. Nodes with the same parent node are referred to as siblings.
- Firefox's DOM Inspector and the IE Web Developer Toolbar allow you to see a visual representation of a document's DOM tree and information about each node.
- The document node in a DOM tree is called the root node, because it has no parent.

### Section 12.3 Traversing and Modifying a DOM Tree

- The `className` property of a DOM node allows you to change an XHTML element's `class` attribute.
- The `id` property of a DOM node controls an element's `id` attribute.
- The `document` object's `createElement` method creates a new DOM node, taking the tag name as an argument. Note that while `createElement` creates an element, it does not *insert* the element on the page.
- The `document`'s `createTextNode` method creates a DOM node that can contain only text. Given a string argument, `createTextNode` inserts the string into the text node.
- Method `appendChild` is called on a parent node to insert a child node (passed as an argument) after any existing children.

- The `parentNode` property of any DOM node contains the node's parent.
- The `insertBefore` method is called on a parent with a new child and an existing child as arguments. The new child is inserted as a child of the parent directly before the existing child.
- The `replaceChild` method is called on a parent, taking a new child and an existing child as arguments. The method inserts the new child into its list of children in place of the existing child.
- The `removeChild` method is called on a parent with a child to be removed as an argument.

#### Section 12.4 DOM Collections

- Included in the Document Object Model is the notion of collections, which are groups of related objects on a page. DOM collections are accessed as properties of DOM objects such as the `document` object or a DOM node.
- The `document` object has properties containing the `images` collection, `links` collection, `forms` collection and `anchors` collection. These collections contain all the elements of the corresponding type on the page.
- To find the number of elements in the collection, use the collection's `length` property.
- To access items in a collection, use square brackets the same way you would with an array, or use the `item` method. The `item` method of a DOM collection is used to access specific elements in a collection, taking an index as an argument. The `namedItem` method takes a name as a parameter and finds the element in the collection, if any, whose `id` attribute or `name` attribute matches it.
- The `href` property of a DOM link node refers to the link's `href` attribute. Many types of XHTML elements are represented by special types of nodes that extend the functionality of a basic DOM node.
- Collections allow easy access to all elements of a single type in a page. This is useful for gathering elements into one place and for applying changes across an entire page.

#### Section 12.5 Dynamic Styles

- An element's style can be changed dynamically. Often such a change is made in response to user events, which are discussed in the next chapter. Such style changes can create many effects, including mouse hover effects, interactive menus, and animations.
- The `body` property of the `document` object refers to the `body` element in the XHTML page.
- The `style` property can access a CSS property in the format `node.style.property`.
- A CSS property with a hyphen (-), such as `background-color`, is referred to as `backgroundColor` in JavaScript, to avoid confusion with the subtraction (-) operator. Removing the hyphen and capitalizing the first letter of the following word is the convention for most CSS properties.
- The `setInterval` method of the `window` object repeatedly executes a statement on a certain interval. It takes two parameters—a statement to execute repeatedly, and an integer specifying how often to execute it, in milliseconds. The `setInterval` method returns a unique identifier to keep track of that particular interval.
- The `window` object's `clearInterval` method stops the repetitive calls of object's `setInterval` method. We pass to `clearInterval` the interval identifier that `setInterval` returned.

#### Section 12.6 Summary of the DOM Objects and Collections

- The objects and collections in the W3C DOM give you flexibility in manipulating the elements of a web page.
- The W3C DOM allows you to access every element in an XHTML document. Each element in a document is represented by a separate object.

- For a reference on the W3C Document Object Model, see the DOM Level 3 recommendation from the W3C at <http://www.w3.org/TR/DOM-Level-3-Core/>. The DOM Level 2 HTML Specification, available at <http://www.w3.org/TR/DOM-Level-2-HTML/>, describes additional DOM functionality specific to HTML, such as objects for various types of XHTML elements.
- Not all web browsers implement all features included in the DOM specification.

## Terminology

anchors collection of the document object  
 body property of the document object  
 appendChild method of a DOM node  
 child  
 className property of a DOM node  
 clearInterval method of the window object  
 collection  
 createElement method of the document object  
 createTextNode method of the document object  
 document object  
 Document Object Model  
 DOM collection  
 DOM Inspector  
 DOM node  
 DOM tree  
 dynamic style  
 forms collection of the document object  
 href property of an a (anchor) node

Internet Explorer Web Developer Toolbar  
 id property of a DOM node  
 images collection  
 innerHTML property of a DOM node  
 insertBefore method of a DOM node  
 item method of a DOM collection  
 length property of a DOM collection  
 links collection of the document object  
 namedItem method of a DOM collection  
 object hierarchy  
 parent  
 removeChild method of a DOM node  
 replaceChild method of a DOM node  
 root node  
 setInterval method of the window object  
 sibling  
 style property of a DOM node  
 W3C Document Object Model

## Self-Review Exercises

- 12.1** State whether each of the following is *true* or *false*. If *false*, explain why.
- Every XHTML element in a page is represented by a DOM tree.
  - A text node cannot have child nodes.
  - The document node in a DOM tree cannot have child nodes.
  - You can change an element's style class dynamically with the `style` property.
  - The `createElement` method creates a new node and inserts it into the document.
  - The `setInterval` method calls a function repeatedly at a set time interval.
  - The `insertBefore` method is called on the document object, taking a new node and an existing one to insert the new one before.
  - The most recently started interval is stopped when the `clearInterval` method is called.
  - The collection `links` contains all the links in a document with specified name or id attributes.
- 12.2** Fill in the blanks for each of the following statements.
- The \_\_\_\_\_ property refers to the text inside an element, including XHTML tags.
  - A document's DOM \_\_\_\_\_ represents all of the nodes in a document, as well as their relationships to each other.
  - The \_\_\_\_\_ property contains the number of elements in a collection.
  - The \_\_\_\_\_ method allows access to an individual element in a collection.
  - The \_\_\_\_\_ collection contains all the `img` elements on a page.
  - The \_\_\_\_\_ object contains information about the sites that a user previously visited.
  - CSS properties may be accessed using the \_\_\_\_\_ object.

## Answers to Self-Review Exercises

**12.1** a) False. Every element is represented by a DOM *node*. Each node is a member of the document's DOM tree. b) True. c) False. The document is the root node, therefore has no parent node. d) False. The style class is changed with the `className` property. e) False. The `createElement` method creates a node, but does not insert it into the DOM tree. f) True. g) False. `insertBefore` is called on the parent. h) False. `clearInterval` takes an interval identifier as an argument to determine which interval to end. i) False. The `links` collection contains all links in a document.

**12.2** a) `innerHTML`. b) `tree`. c) `length`. d) `item`. e) `images`. f) `history`. g) `style`.

## Exercises

**12.3** Modify Fig. 12.3 to use a background color to highlight all the links in the page instead of displaying them in a box at the bottom.

**12.4** Use the Firefox DOM Inspector or the IE Web Developer Toolbar to view the DOM tree of the document in Fig. 12.2. Look at the document tree of your favorite website. Notice the information these tools give you in the right panel(s) about an element when you click it.

**12.5** Write a script that contains a button and a counter in a `div`. The button should increment the counter each time it is clicked.

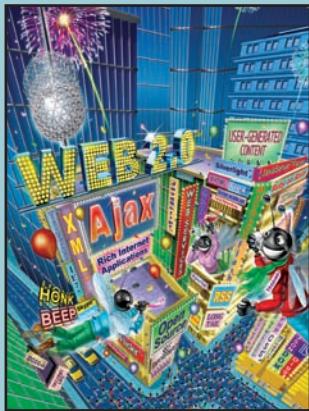
**12.6** Write a script that prints out the length of all the JavaScript collections on a page.

**12.7** Create a web page in which users are allowed to select their favorite layout and formatting through the use of the `className` property.

**12.8** (*15 Puzzle*) Write a web page that enables the user to play the game of 15. There is a 4-by-4 board (implemented as an XHTML table) for a total of 16 slots. One of the slots is empty. The other slots are occupied by 15 tiles, randomly numbered from 1 through 15. Any tile next to the currently empty slot can be moved into the currently empty slot by clicking on the tile. Your program should create the board with the tiles out of order. The user's goal is to arrange the tiles in sequential order row by row. Using the DOM and the `onClick` event, write a script that allows the user to swap the positions of the open position and an adjacent tile. [*Hint:* The `onClick` event should be specified for each table cell.]

**12.9** Modify your solution to Exercise 12.8 to determine when the game is over, then prompt the user to determine whether to play again. If so, scramble the numbers using the `Math.random` method.

**12.10** Modify your solution to Exercise 12.9 to use an image that is split into 16 equally sized pieces. Discard one of the pieces and randomly place the other 15 pieces in the XHTML table.



*The wisest prophets make  
sure of the event first.*

—Horace Walpole

*Do you think I can listen all  
day to such stuff?*

—Lewis Carroll

*The user should feel in  
control of the computer; not  
the other way around. This  
is achieved in applications  
that embody three qualities:  
responsiveness,  
permissiveness, and  
consistency.*

—Inside Macintosh, Volume 1  
Apple Computer, Inc., 1985

*We are responsible for  
actions performed in  
response to circumstances for  
which we are not  
responsible.*

—Allan Massie

# JavaScript: Events

## OBJECTIVES

In this chapter you will learn:

- The concepts of events, event handlers and event bubbling.
- To create and register event handlers that respond to mouse and keyboard events.
- To use the `event` object to get information about an event.
- To recognize and respond to many common events.

**Outline**

- 13.1** Introduction
- 13.2** Registering Event Handlers
- 13.3** Event `onload`
- 13.4** Event `onmousemove`, the `event` Object, and `this`
- 13.5** Rollovers with `onmouseover` and `onmouseout`
- 13.6** Form Processing with `onfocus` and `onblur`
- 13.7** More Form Processing with `onsubmit` and `onreset`
- 13.8** Event Bubbling
- 13.9** More Events
- 13.10** Wrap-Up
- 13.11** Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 13.1 Introduction

We've seen that XHTML pages can be controlled via scripting, and we've already used a few events to trigger scripts, such as the `onclick` and `onsubmit` events. This chapter goes into more detail on [JavaScript events](#), which allow scripts to respond to user interactions and modify the page accordingly. Events allow scripts to respond to a user who is moving the mouse, entering form data or pressing keys. Events and event handling help make web applications more responsive, dynamic and interactive.

In this chapter, we discuss how to set up functions to react when an event [fires](#) (occurs). We give examples of event handling for nine common events, including mouse events and form-processing events. At the end of the chapter, we provide a table of the events covered in this chapter and other useful events.

## 13.2 Registering Event Handlers

Functions that handle events are called [event handlers](#). Assigning an event handler to an event on a DOM node is called [registering an event handler](#). Previously, we have registered event handlers using the [inline model](#), treating events as attributes of XHTML elements (e.g., `<p onclick = "myfunction()">`). Another model, known as the [traditional model](#), for registering event handlers is demonstrated alongside the inline model in Fig. 13.1.

In the earliest event-capable browsers, the inline model was the only way to handle events. Later, Netscape developed the traditional model and Internet Explorer adopted it. Since then, both Netscape and Microsoft have developed separate (incompatible)

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4

```

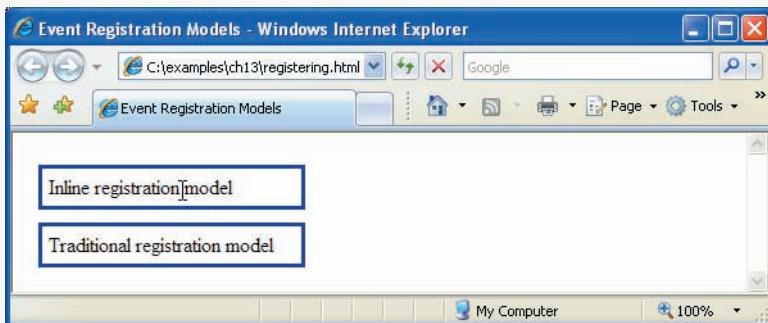
**Fig. 13.1** | Event registration models. (Part 1 of 3.)

```

5 <!-- Fig. 13.1: registering.html -->
6 <!-- Event registration models. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Event Registration Models</title>
10 <style type = "text/css">
11 div { padding: 5px;
12 margin: 10px;
13 border: 3px solid #0000BB;
14 width: 12em }
15 </style>
16 <script type = "text/javascript">
17 <!--
18 // handle the onclick event regardless of how it was registered
19 function handleEvent()
20 {
21 alert("The event was successfully handled.");
22 } // end function handleEvent
23
24 // register the handler using the traditional model
25 function registerHandler()
26 {
27 var traditional = document.getElementById("traditional");
28 traditional.onclick = handleEvent;
29 } // end function registerHandler
30 // --
31 </script>
32 </head>
33 <body onload = "registerHandler()">
34 <!-- The event handler is registered inline -->
35 <div id = "inline" onclick = "handleEvent()">
36 Inline registration model</div>
37
38 <!-- The event handler is registered by function registerHandler -->
39 <div id = "traditional">Traditional registration model</div>
40 </body>
41 </html>

```

- a) The user clicks the **div** for which the event handler was registered using the inline model.



**Fig. 13.1** | Event registration models. (Part 2 of 3.)

- b) The event handler displays an alert dialog.



- c) The user clicks the `div` for which the event handler was registered using the traditional model.



- d) The event handler displays an alert dialog..



**Fig. 13.1 |** Event registration models. (Part 3 of 3.)

advanced event models with more functionality than either the inline or the traditional model. Netscape's advanced model was adapted by the W3C to create a DOM Events Specification. Most browsers support the W3C model, but Internet Explorer 7 does not. This means that to create cross-browser websites, we are mostly limited to the traditional and inline event models. While the advanced models provide more convenience and functionality, most of the features can be implemented with the traditional model.

Line 35 assigns "handleEvent()" to the `onClick` attribute of the `div` in lines 35–36. This is the inline model for event registration we've seen in previous examples. The `div` in line 39 is assigned an event handler using the traditional model. When the `body` element (lines 33–40) loads, the `registerHandler` function is called.

Function `registerHandler` (lines 25–29) uses JavaScript to register the function `handleEvent` as the event handler for the `onClick` event of the `div` with the `id` "traditional". Line 27 gets the `div`, and line 28 assigns the function `handleEvent` to the `div`'s `onClick` property.

Notice that in line 28, we do not put `handleEvent` in quotes or include parentheses at the end of the function name, as we do in the inline model in line 35. In the inline

model, the value of the XHTML attribute is a *JavaScript statement* to execute when the event occurs. The value of the `onclick` property of a DOM node is not an executable statement, but the name of a *function* to be called when the event occurs. Recall that JavaScript functions can be treated as data (i.e., passed into methods, assigned to variables, etc.).



### Common Programming Error 13.1

*Putting quotes around the function name when registering it using the inline model would assign a string to the onclick property of the node—a string cannot be called.*



### Common Programming Error 13.2

*Putting parentheses after the function name when registering it using the inline model would call the function immediately and assign its return value to the onclick property.*

Once the event handler is registered in line 28, the `div` in line 39 has the same behavior as the `div` in lines 35–36, because `handleEvent` (lines 19–22) is set to handle the `onclick` event for both `div`s. When either `div` is clicked, an alert will display "The event was successfully handled."

The traditional model allows us to register event handlers in JavaScript code. This has important implications for what we can do with JavaScript events. For example, traditional event-handler registration allows us to assign event handlers to many elements quickly and easily using repetition statements, instead of adding an inline event handler to each XHTML element. In the remaining examples in this chapter, we use both the inline and traditional registration models depending on which is more convenient.

## 13.3 Event onload

The `onload` event fires whenever an element finishes loading successfully (i.e., all its children are loaded). Frequently, this event is used in the `body` element to initiate a script after the page loads in the client's browser. Figure 13.2 uses the `onload` event for this purpose. The script called by the `onload` event updates a timer that indicates how many seconds have elapsed since the document was loaded.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 13.2: onload.html -->
6 <!-- Demonstrating the onload event. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>onload Event</title>
10 <script type = "text/javascript">
11 <!--
12 var seconds = 0;
13
14 // called when the page loads to begin the timer
15 function startTimer()
16 {

```

Fig. 13.2 | Demonstrating the `onload` event. (Part 1 of 2.)

```

17 // 1000 milliseconds = 1 second
18 window.setInterval("updateTime()", 1000);
19 } // end function startTimer
20
21 // called every 1000 ms to update the timer
22 function updateTime()
23 {
24 ++seconds;
25 document.getElementById("soFar").innerHTML = seconds;
26 } // end function updateTime
27 // -->
28 </script>
29 </head>
30 <body onload = "startTimer()">
31 <p>Seconds you have spent viewing this page so far:
32 <strong id = "soFar">0</p>
33 </body>
34 </html>

```



**Fig. 13.2 |** Demonstrating the `onload` event. (Part 2 of 2.)

Our use of the `onload` event occurs in line 30. After the `body` section loads, the browser triggers the `onload` event. This calls function `startTimer` (lines 15–19), which in turn uses method `window.setInterval` to specify that function `updateTime` (lines 22–26) should be called every 1000 milliseconds. The `updateTime` function increments variable `seconds` and updates the counter on the page.

Note that we could not have created this program without the `onload` event, because elements in the XHTML page cannot be accessed until the page has loaded. If a script in the head attempts to get a DOM node for an XHTML element in the body, `getElementById` returns `null` because the body has not yet loaded. Other uses of the `onload` event include opening a pop-up window once a page has loaded and triggering a script when an image or Java applet loads.



### Common Programming Error 13.3

*Trying to get an element in a page before the page has loaded is a common error. Avoid this by putting your script in a function using the `onload` event to call the function.*

## 13.4 Event `onmousemove`, the `event` Object and `this`

This section introduces the `onmousemove` event, which fires repeatedly whenever the user moves the mouse over the web page. We also discuss the `event` object and the keyword

this, which permit more advanced event-handling capabilities. Figure 13.3 uses onmousemove and this to create a simple drawing program that allows the user to draw inside a box in red or blue by holding down the Shift or Ctrl keys.

The XHTML body has a table with a tbody containing one row that gives the user instructions on how to use the program. The body's onload attribute (line 61) calls function createCanvas, which initializes the program by filling in the table.

The createCanvas function (lines 23–41) fills in the table with a grid of cells. The CSS rule in lines 14–15 sets the width and height of every td element to 4px. Line 11

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 13.3: draw.html -->
6 <!-- A simple drawing program. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Simple Drawing Program</title>
10 <style type = "text/css">
11 #canvas { width: 400px;
12 border: 1px solid #999999;
13 border-collapse: collapse }
14 td { width: 4px;
15 height: 4px }
16 th.key { font-family: arial, helvetica, sans-serif;
17 font-size: 12px;
18 border-bottom: 1px solid #999999 }
19 </style>
20 <script type = "text/javascript">
21 <!--
22 //initialization function to insert cells into the table
23 function createCanvas ()
24 {
25 var side = 100;
26 var tbody = document.getElementById("tbody");
27
28 for (var i = 0; i < side; i++)
29 {
30 var row = document.createElement("tr");
31
32 for (var j = 0; j < side; j++)
33 {
34 var cell = document.createElement("td");
35 cell.onmousemove = processMouseMove;
36 row.appendChild(cell);
37 } // end for
38
39 tbody.appendChild(row);
40 } // end for
41 } // end function createCanvas
42

```

**Fig. 13.3** | Simple drawing program. (Part I of 3.)

```
43 // processes the onmousemove event
44 function processMouseMove(e)
45 {
46 // get the event object from IE
47 if (!e)
48 var e = window.event;
49
50 // turn the cell blue if the Ctrl key is pressed
51 if (e.ctrlKey)
52 this.style.backgroundColor = "blue";
53
54 // turn the cell red if the Shift key is pressed
55 if (e.shiftKey)
56 this.style.backgroundColor = "red";
57 } // end function processMouseMove
58 // -->
59 </script>
60</head>
61<body onload = "createCanvas()">
62 <table id = "canvas" class = "canvas"><tbody id = "tablebody">
63 <tr><th class = "key" colspan = "100">Hold <tt>ctrl</tt>
64 to draw blue. Hold <tt>shift</tt> to draw red.</th></tr>
65 </tbody></table>
66</body>
67</html>
```

- a) The page loads and fills with white cells. With no keys held down, moving the mouse does not draw anything.

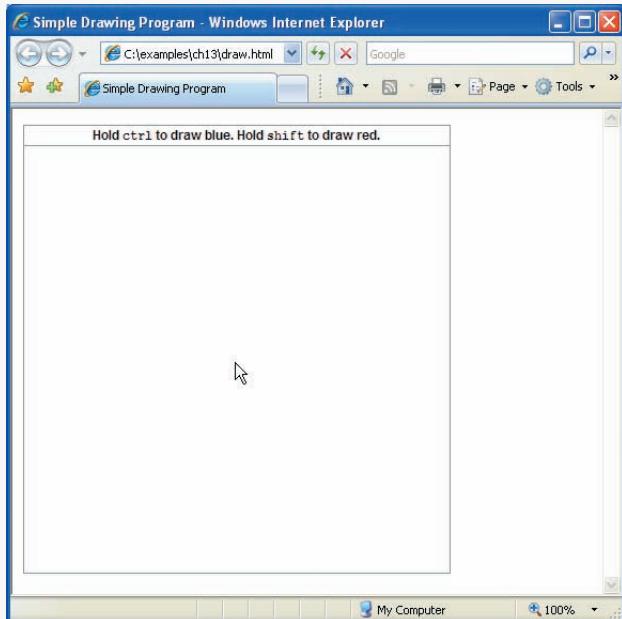
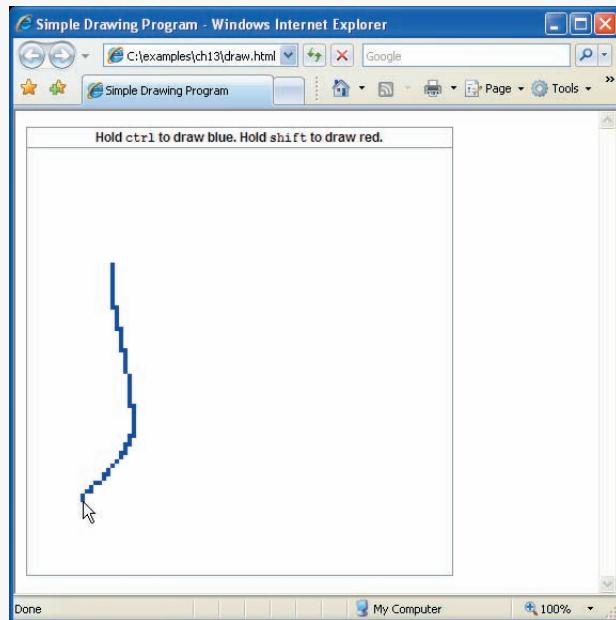
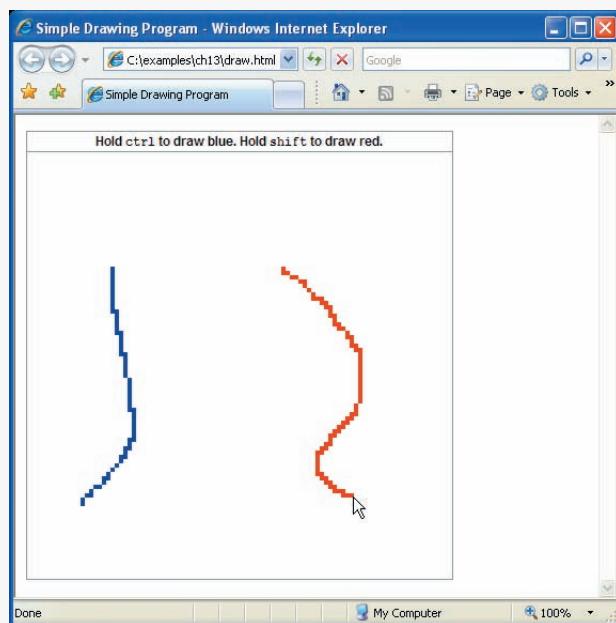


Fig. 13.3 | Simple drawing program. (Part 2 of 3.)

b) The user holds the *Ctrl* key and moves the mouse to draw a blue line.



c) The user holds the *Shift* key and moves the mouse to draw a red line.



**Fig. 13.3** | Simple drawing program. (Part 3 of 3.)

dictates that the table is 400px wide. Line 13 uses the `border-collapse` CSS property to eliminate space between the table cells.

Line 25 defines variable `side`, which determines the number of cells in each row and the number of rows created by the nested `for` statements in lines 28–40. We set `side` to 100 in order to fill the table with 10,000 4px cells. Line 26 stores the `tbody` element so that we can append rows to it as they are generated.



### Common Programming Error 13.4

*Although you can omit the `tbody` element in an XHTML table, without it you cannot append `tr` elements as children of a `table` using JavaScript. While Firefox treats appended rows as members of the table body, Internet Explorer will not render any table cells that are dynamically added to a table outside a `thead`, `tbody` or `tfoot` element.*

The nested `for` statements in lines 28–40 fill the table with a  $100 \times 100$  grid of cells. The outer loop creates each table row, while the inner loop creates each cell. The inner loop uses the `createElement` method to create a table cell, assigns function `processMouseMove` as the event handler for the cell's `onmousemove` event and appends the cell as a child of the row. The `onmousemove` event of an element fires whenever the user moves the mouse over that element.

At this point, the program is initialized and simply calls `processMouseMove` whenever the mouse moves over any table cell. The function `processMouseMove` (lines 44–57) colors the cell the mouse moves over, depending on the key that is pressed when the event occurs. Lines 44–48 get the `event object`, which stores information about the event that called the event-handling function.

Internet Explorer and Firefox do not implement the same event models, so we need to account for some differences in how the event object can be handled and used. Firefox and other W3C-compliant browsers (e.g., Safari, Opera) pass the event object as an argument to the event-handling function. Internet Explorer, on the other hand, stores the event object in the `event` property of the `window` object. To get the event object regardless of the browser, we use a two-step process. Function `processMouseMove` takes the parameter `e` in line 44 to get the event object from Firefox. Then, if `e` is undefined (i.e., if the client is Internet Explorer), we assign the object in `window.event` to `e` in line 48.

In addition to providing different ways to access the event object, Firefox and Internet Explorer also implement different functionality in the event object itself. However, there are several event properties that both browsers implement with the same name, and some that both browsers implement with different names. In this book, we use properties that are implemented in both event models, or we write our code to use the correct property depending on the browser—all of our code runs properly in IE7 and Firefox 2.

Once `e` contains the event object, we can use it to get information about the event. Lines 51–56 do the actual drawing. The event object's `ctrlKey` property contains a boolean which reflects whether the `Ctrl` key was pressed during the event. If `ctrlKey` is true, line 52 executes, changing the color of a table cell.

To determine which table cell to color, we introduce the `this keyword`. The meaning of `this` depends on its context. In an event-handling function, `this` refers to the DOM object on which the event occurred. Our function uses `this` to refer to the table cell over which the mouse moved. The `this` keyword allows us to use one event handler to apply a change to one of many DOM elements, depending on which one received the event.

Lines 51–52 change the background color of this table cell to blue if the *Ctrl* key is pressed during the event. Similarly, lines 55–56 color the cell red if the *Shift* key is pressed. To determine this, we use the **shiftKey** property of the event object. This simple function allows the user to draw inside the table on the page in red and blue. You'll add more functionality to this example in the exercises at the end of this chapter.

This example demonstrated the **ctrlKey** and **shiftKey** properties of the event object. Figure 13.4 provides a table of some important cross-browser properties of the event object.

This section introduced the event **onmousemove** and the keyword **this**. We also discussed more advanced event handling using the event object to get information about the event. The next section continues our introduction of events with the **onmouseover** and **onmouseout** events.

| Property                                      | Description                                                                                                                                                         |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>altKey</code>                           | This value is <code>true</code> if the <i>Alt</i> key was pressed when the event fired.                                                                             |
| <code>cancelBubble</code>                     | Set to <code>true</code> to prevent the event from bubbling. Defaults to <code>false</code> . (See Section 14.9, Event Bubbling.)                                   |
| <code>clientX</code> and <code>clientY</code> | The coordinates of the mouse cursor inside the client area (i.e., the active area where the web page is displayed, excluding scrollbars, navigation buttons, etc.). |
| <code>ctrlKey</code>                          | This value is <code>true</code> if the <i>Ctrl</i> key was pressed when the event fired.                                                                            |
| <code>keyCode</code>                          | The ASCII code of the key pressed in a keyboard event. See Appendix D for more information on the ASCII character set.                                              |
| <code>screenX</code> and <code>screenY</code> | The coordinates of the mouse cursor on the screen coordinate system.                                                                                                |
| <code>shiftKey</code>                         | This value is <code>true</code> if the <i>Shift</i> key was pressed when the event fired.                                                                           |
| <code>type</code>                             | The name of the event that fired, without the prefix "on".                                                                                                          |

**Fig. 13.4** | Some event object properties.

## 13.5 Rollovers with onmouseover and onmouseout

Two more events fired by mouse movements are **onmouseover** and **onmouseout**. When the mouse cursor moves into an element, an **onmouseover event** occurs for that element. When the cursor leaves the element, an **onmouseout event** occurs. Figure 13.5 uses these events to achieve a **rollover effect** that updates text when the mouse cursor moves over it. We also introduce a technique for creating rollover images.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4

```

**Fig. 13.5** | Events onmouseover and onmouseout. (Part 1 of 5.)

```

5 <!-- Fig. 13.5: onmouseoverout.html -->
6 <!-- Events onmouseover and onmouseout. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Events onmouseover and onmouseout</title>
10 <style type = "text/css">
11 body { background-color: wheat }
12 table { border-style: groove;
13 text-align: center;
14 font-family: monospace;
15 font-weight: bold }
16 td { width: 6em }
17 </style>
18 <script type = "text/javascript">
19 <!--
20 image1 = new Image();
21 image1.src = "heading1.gif";
22 image2 = new Image();
23 image2.src = "heading2.gif";
24
25 function mouseOver(e)
26 {
27 if (!e)
28 var e = window.event;
29
30 var target = getTarget(e);
31
32 // swap the image when the mouse moves over it
33 if (target.id == "heading")
34 {
35 target.src = image2.src;
36 return;
37 } // end if
38
39 // if an element's id is defined, assign the id to its color
40 // to turn hex code's text the corresponding color
41 if (target.id)
42 target.style.color = target.id;
43 } // end function mouseOver
44
45 function mouseOut(e)
46 {
47 if (!e)
48 var e = window.event;
49
50 var target = getTarget(e);
51
52 // put the original image back when the mouse moves away
53 if (target.id == "heading")
54 {
55 target.src = image1.src;
56 return;
57 } // end if

```

Fig. 13.5 | Events onmouseover and onmouseout. (Part 2 of 5.)

```
58 // if an element's id is defined, assign id to innerHTML
59 // to display the color name
60 if (target.id)
61 target.innerHTML = target.id;
62 } // end function mouseOut
63
64
65 // return either e.srcElement or e.target, whichever exists
66 function getTarget(e)
67 {
68 if (e.srcElement)
69 return e.srcElement;
70 else
71 return e.target;
72 } // end function getTarget
73
74 document.onmouseover = mouseOver;
75 document.onmouseout = mouseOut;
76 // -->
77 </script>
78 </head>
79 <body>
80
81 <p>Can you tell a color from its hexadecimal RGB code
82 value? Look at the hex code, guess its color. To see
83 what color it corresponds to, move the mouse over the
84 hex code. Moving the mouse out of the hex code's table
85 cell will display the color name.</p>
86 <table>
87 <tr>
88 <td id = "Black">#000000</td>
89 <td id = "Blue">#0000FF</td>
90 <td id = "Magenta">#FF00FF</td>
91 <td id = "Gray">#808080</td>
92 </tr>
93 <tr>
94 <td id = "Green">#008000</td>
95 <td id = "Lime">#00FF00</td>
96 <td id = "Maroon">#800000</td>
97 <td id = "Navy">#000080</td>
98 </tr>
99 <tr>
100 <td id = "Olive">#808000</td>
101 <td id = "Purple">#800080</td>
102 <td id = "Red">#FF0000</td>
103 <td id = "Silver">#C0C0C0</td>
104 </tr>
105 <tr>
106 <td id = "Cyan">#00FFFF</td>
107 <td id = "Teal">#008080</td>
108 <td id = "Yellow">#FFFF00</td>
109 <td id = "White">#FFFFFF</td>
110 </tr>
```

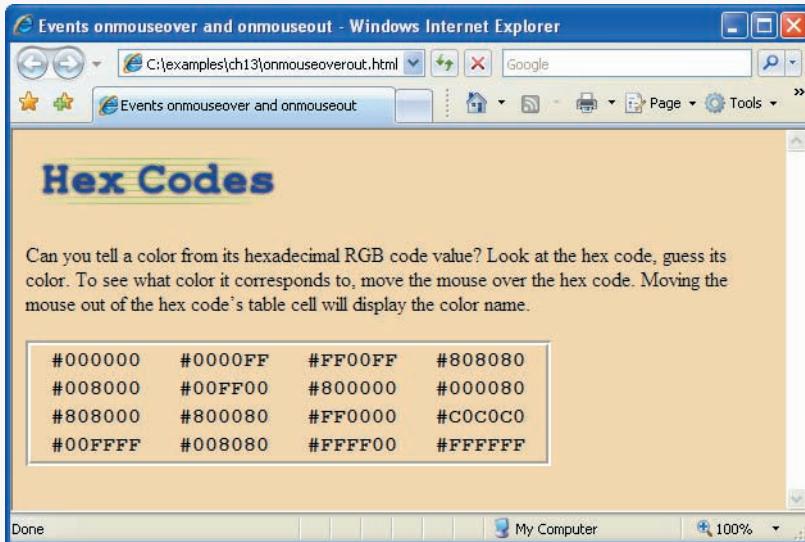
**Fig. 13.5** | Events onmouseover and onmouseout. (Part 3 of 5.)

```

111 </table>
112 </body>
113 </html>

```

- a) The page loads with the blue heading image and all the hex codes in black.



- b) The heading image switches to an image with green text when the mouse rolls over it.

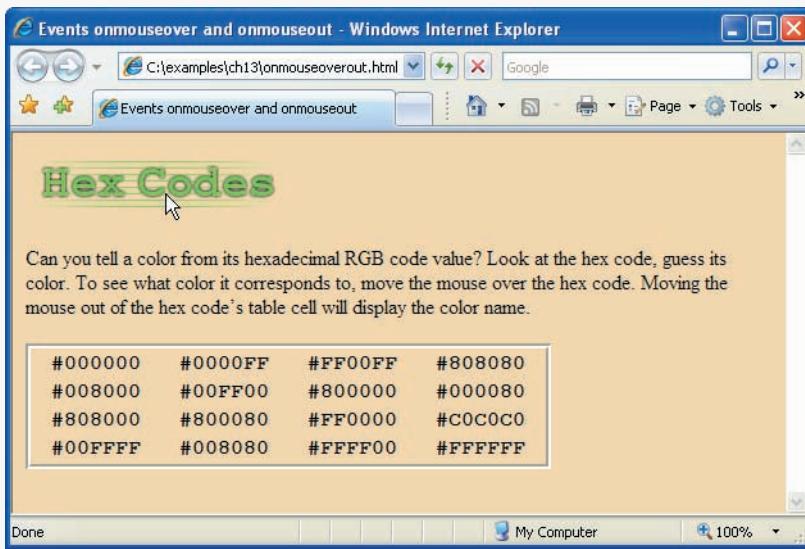
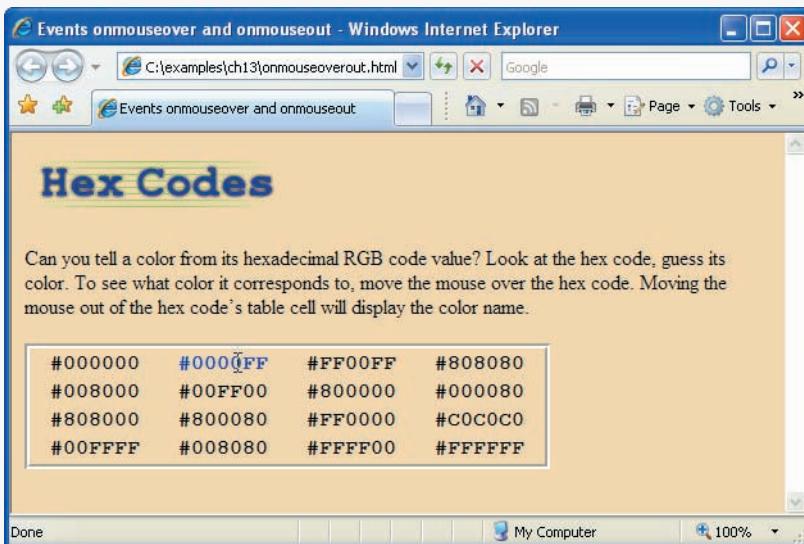
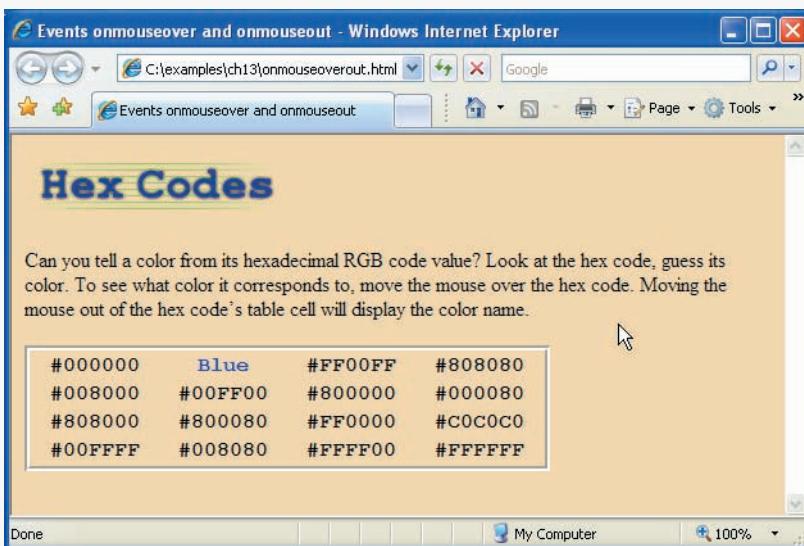


Fig. 13.5 | Events onmouseover and onmouseout. (Part 4 of 5.)

- c) When mouse rolls over a hex code, the text color changes to the color represented by the hex code. Notice that the heading image has become blue again because the mouse is no longer over it.



- d) When the mouse leaves the hex code's table cell, the text changes to the name of the color.



**Fig. 13.5** | Events onmouseover and onmouseout. (Part 5 of 5.)

To create a rollover effect for the image in the heading, lines 20–23 create two new JavaScript `Image` objects—`image1` and `image2`. Image `image2` displays when the mouse hovers over the image. Image `image1` displays when the mouse is outside the image. The script sets the `src` properties of each `Image` in lines 21 and 23, respectively. Creating `Image`

objects preloads the images (i.e., loads the images in advance), so the browser does not need to download the rollover image the first time the script displays the image. If the image is large or the connection is slow, downloading would cause a noticeable delay in the image update.



### Performance Tip 13.1

*Preloading images used in rollover effects prevents a delay the first time an image is displayed.*

Functions `mouseOver` and `mouseout` are set to process the `onmouseover` and `onmouseout` events, respectively, in lines 74–75. Both functions begin (lines 25–28 and 45–48) by getting the event object and using function `getTarget` to find the element that received the action. Because of browser event model differences, we need `getTarget` (defined in lines 66–72) to return the DOM node targeted by the action. In Internet Explorer, this node is stored in the event object's `srcElement` property. In Firefox, it is stored in the event object's `target` property. Lines 68–71 return the node using the correct property to hide the browser differences from the rest of our program. We must use function `getTarget` instead of this because we do not define an event handler for each specific element in the document. In this case, using `this` would return the entire document. In both `mouseOver` and `mouseout`, we assign the return value of `getTarget` to variable `target` (lines 30 and 50).

Lines 33–37 in the `mouseOver` function handle the `onmouseover` event for the heading image by setting its `src` attribute (`target.src`) to the `src` property of the appropriate `Image` object (`image2.src`). The same task occurs with `image1` in the `mouseout` function (lines 53–57).

The script handles the `onmouseover` event for the table cells in lines 41–42. This code tests whether an `id` is specified, which is true only for our hex code table cells and the heading image in this example. If the element receiving the action has an `id`, the code changes the color of the element to match the color name stored in the `id`. As you can see in the code for the `table` (lines 86–111), each `td` element containing a color code has an `id` attribute set to one of the 16 basic XHTML colors. Lines 61–62 handle the `onmouseout` event by changing the text in the table cell the mouse cursor just left to match the color that it represents.

## 13.6 Form Processing with `onfocus` and `onblur`

The `onfocus` and `onblur` events are particularly useful when dealing with form elements that allow user input (Fig. 13.6). The `onfocus` event fires when an element gains focus (i.e., when the user clicks a form field or uses the *Tab* key to move between form elements), and `onblur` fires when an element loses focus, which occurs when another control gains the focus. In lines 31–32, the script changes the text inside the `div` below the form (line 58) based on the `messageNum` passed to function `helpText` (lines 29–33). Each of the elements of the form, such as the `name` input in lines 40–41, passes a different value to the `helpText` function when it gains focus (and its `onfocus` event fires). These values are used as indices for `helpArray`, which is declared and initialized in lines 17–27 and stores help messages. When elements lose focus, they all pass the value 6 to `helpText` to clear the `tip` `div` (note that the empty string "" is stored in the last element of the array).

```
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 13.6: onfocusblur.html -->
6 <!-- Demonstrating the onFocus and onBlur events. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>A Form Using onFocus and onBlur</title>
10 <style type = "text/css">
11 .tip { font-family: sans-serif;
12 color: blue;
13 font-size: 12px }
14 </style>
15 <script type = "text/javascript">
16 <!--
17 var helpArray =
18 ["Enter your name in this input box.", // element 0
19 "Enter your e-mail address in this input box, " +
20 "in the format user@domain.", // element 1
21 "Check this box if you liked our site.", // element 2
22 "In this box, enter any comments you would " +
23 "like us to read.", // element 3
24 "This button submits the form to the " +
25 "server-side script.", // element 4
26 "This button clears the form.", // element 5
27 ""]; // element 6
28
29 function helpText(messageNum)
30 {
31 document.getElementById("tip").innerHTML =
32 helpArray[messageNum];
33 } // end function helpText
34 // --
35 </script>
36 </head>
37 <body>
38 <form id = "myForm" action = "">
39 <div>
40 Name: <input type = "text" name = "name"
41 onFocus = "helpText(0)" onBlur = "helpText(6)" />

42 E-mail: <input type = "text" name = "e-mail"
43 onFocus = "helpText(1)" onBlur = "helpText(6)" />

44 Click here if you like this site
45 <input type = "checkbox" name = "like" onFocus =
46 "helpText(2)" onBlur = "helpText(6)" />
<hr />
47
48 Any comments?

49 <textarea name = "comments" rows = "5" cols = "45"
50 onFocus = "helpText(3)" onBlur = "helpText(6)"></textarea>
51

52 <input type = "submit" value = "Submit" onFocus =
53 "helpText(4)" onBlur = "helpText(6)" />
```

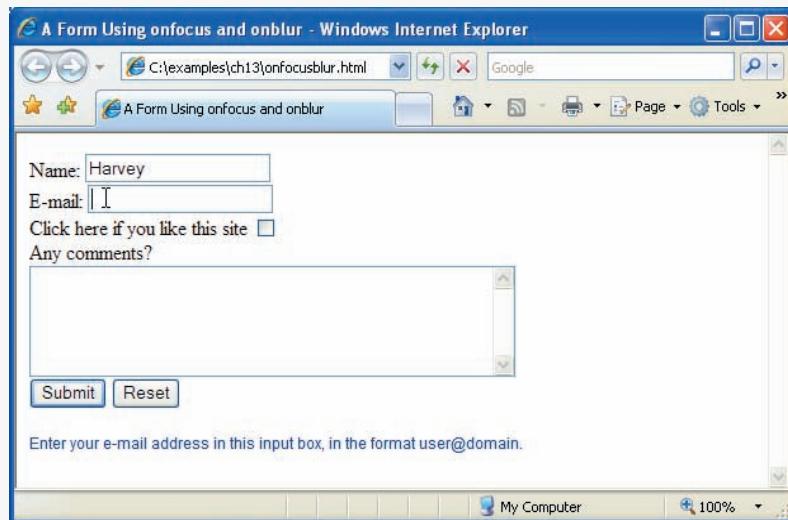
Fig. 13.6 | Demonstrating the onFocus and onBlur events. (Part I of 2.)

```

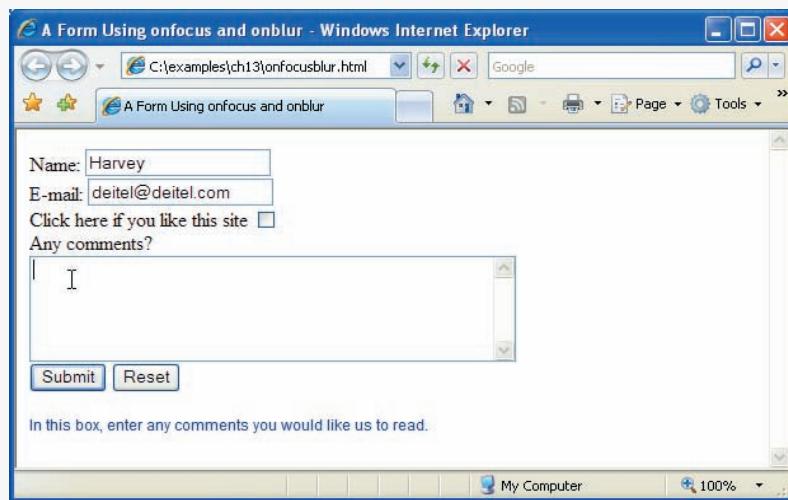
54 <input type = "reset" value = "Reset" onfocus =
55 "helpText(5)" onblur = "helpText(6)" />
56 </div>
57 </form>
58 <div id = "tip" class = "tip"></div>
59 </body>
60 </html>

```

- a) The blue message at the bottom of the page instructs the user to enter an e-mail when the e-mail field has focus.



- b) The message changes depending on which field has focus. Now it gives instructions for the comments box.



**Fig. 13.6** | Demonstrating the onfocus and onblur events. (Part 2 of 2.)

## 13.7 More Form Processing with onsubmit and onreset

Two more useful events for processing forms are `onsubmit` and `onreset`. These events fire when a form is submitted or reset, respectively (Fig. 13.7). Function `registerEvents` (lines 35–46) registers the event handlers for the form after the body has loaded.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 13.7: onsubmitreset.html -->
6 <!-- Demonstrating the onsubmit and onreset events. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>A Form Using onsubmit and onreset</title>
10 <style type = "text/css">
11 .tip { font-family: sans-serif;
12 color: blue;
13 font-size: 12px }
14 </style>
15 <script type = "text/javascript">
16 <!--
17 var helpArray =
18 ["Enter your name in this input box.",
19 "Enter your e-mail address in this input box, " +
20 "in the format user@domain.",
21 "Check this box if you liked our site.",
22 "In this box, enter any comments you would " +
23 "like us to read.",
24 "This button submits the form to the " +
25 "server-side script.",
26 "This button clears the form.",
27 ""];
28
29 function helpText(messageNum)
30 {
31 document.getElementById("tip").innerHTML =
32 helpArray[messageNum];
33 } // end function helpText
34
35 function registerEvents()
36 {
37 document.getElementById("myForm").onsubmit = function()
38 {
39 return confirm("Are you sure you want to submit?");
40 } // end anonymous function
41
42 document.getElementById("myForm").onreset = function()
43 {
44 return confirm("Are you sure you want to reset?");
45 } // end anonymous function
46 } // end function registerEvents

```

**Fig. 13.7** | Demonstrating the `onsubmit` and `onreset` events. (Part I of 2.)

```

47 // -->
48 </script>
49</head>
50<body onload = "registerEvents()">
51 <form id = "myForm" action = "">
52 <div>
53 Name: <input type = "text" name = "name"
54 onfocus = "helpText(0)" onblur = "helpText(6)" />

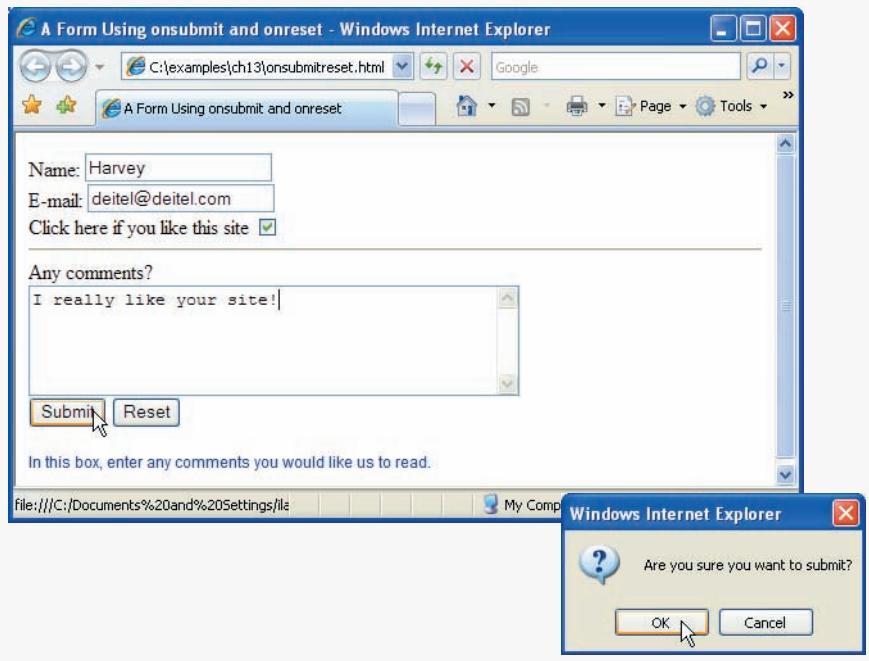
55 E-mail: <input type = "text" name = "e-mail"
56 onfocus = "helpText(1)" onblur = "helpText(6)" />

57 Click here if you like this site
58 <input type = "checkbox" name = "like" onfocus =
59 "helpText(2)" onblur = "helpText(6)" />
<hr />
60
61 Any comments?

62 <textarea name = "comments" rows = "5" cols = "45"
63 onfocus = "helpText(3)" onblur = "helpText(6)"></textarea>
64

65 <input type = "submit" value = "Submit" onfocus =
66 "helpText(4)" onblur = "helpText(6)" />
67 <input type = "reset" value = "Reset" onfocus =
68 "helpText(5)" onblur = "helpText(6)" />
69 </div>
70 </form>
71 <div id = "tip" class = "tip"></div>
72</body>
73</html>

```



**Fig. 13.7** | Demonstrating the `onsubmit` and `onreset` events. (Part 2 of 2.)

Lines 37–40 and 42–45 introduce several new concepts. Line 37 gets the `form` element ("myForm", lines 51–70), then lines 37–40 assign an **anonymous function** to its `onsubmit` property. An anonymous function is defined with no name—it is created in nearly the same way as any other function, but with no identifier after the keyword `function`. This notation is useful when creating a function for the sole purpose of assigning it to an event handler. We never call the function ourselves, so we don't need to give it a name, and it's more concise to create the function and register it as an event handler at the same time.

The anonymous function (lines 37–40) assigned to the `onsubmit` property of `myForm` executes in response to the user submitting the form (i.e., clicking the `Submit` button or pressing the `Enter` key). Line 39 introduces the **`confirm` method** of the `window` object. As with `alert`, we do not need to prefix the call with the object name `window` and the dot (.) operator. The `confirm` dialog asks the users a question, presenting them with an `OK` button and a `Cancel` button. If the user clicks `OK`, `confirm` returns `true`; otherwise, `confirm` returns `false`.

Our event handlers for the form's `onsubmit` and `onreset` events simply return the value of the `confirm` dialog, which asks the users if they are sure they want to submit or reset (lines 39 and 44, respectively). By returning either `true` or `false`, the event handlers dictate whether the default action for the event—in this case submitting or resetting the form—is taken. (Recall that we also returned `false` from some event-handling functions to prevent forms from submitting in Chapter 12.) Other default actions, such as following a hyperlink, can be prevented by returning `false` from an `onclick` event handler on the link. If an event handler returns `true` or does not return a value, the default action is taken once the event handler finishes executing.

## 13.8 Event Bubbling

**Event bubbling** is the process by which events fired in child elements “bubble” up to their parent elements. When an event is fired on an element, it is first delivered to the element's event handler (if any), then to the parent element's event handler (if any). This might result in event handling that was not intended. If you intend to handle an event in a child element alone, you should cancel the bubbling of the event in the child element's event-handling code by using the **`cancelBubble` property** of the `event` object, as shown in Fig. 13.8.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 13.8: bubbling.html -->
6 <!-- Canceling event bubbling. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Event Bubbling</title>
10 <script type = "text/javascript">
11 <!--

```

Fig. 13.8 | Canceling event bubbling. (Part 1 of 3.)

```

12 function documentClick()
13 {
14 alert("You clicked in the document.");
15 } // end function documentClick
16
17 function bubble(e)
18 {
19 if (!e)
20 var e = window.event;
21
22 alert("This will bubble.");
23 e.cancelBubble = false;
24 } // end function bubble
25
26 function noBubble(e)
27 {
28 if (!e)
29 var e = window.event;
30
31 alert("This will not bubble.");
32 e.cancelBubble = true;
33 } // end function noBubble
34
35 function registerEvents()
36 {
37 document.onclick = documentClick;
38 document.getElementById("bubble").onclick = bubble;
39 document.getElementById("noBubble").onclick = noBubble;
40 } // end function registerEvents
41 // -->
42 </script>
43 </head>
44 <body onload = "registerEvents()">
45 <p id = "bubble">Bubbling enabled.</p>
46 <p id = "noBubble">Bubbling disabled.</p>
47 </body>
48 </html>

```

- a) The user clicks the first paragraph, for which bubbling is enabled.



**Fig. 13.8** | Canceling event bubbling. (Part 2 of 3.)

- b) The paragraph's event handler causes an alert.



- c) The document's event handler causes another alert, because the event bubbles up to the document.



- d) The user clicks the second paragraph, for which bubbling is disabled.



- b) The paragraph's event handler causes an alert. The document's event handler is not called.



**Fig. 13.8 |** Canceling event bubbling. (Part 3 of 3.)

Clicking the first p element (line 45) triggers a call to bubble. Then, because line 37 registers the document's onclick event, documentClick is also called. This occurs because the onclick event bubbles up to the document. This is probably not the desired result. Clicking the second p element (line 46) calls noBubble, which disables the event bubbling for this event by setting the cancelBubble property of the event object to true. [Note: The default value of cancelBubble is false, so the statement in line 23 is unnecessary.]



### Common Programming Error 13.5

*Forgetting to cancel event bubbling when necessary may cause unexpected results in your scripts.*

## 13.9 More Events

The events we covered in this chapter are among the most commonly used. A list of some events supported by both Firefox and Internet Explorer is given with descriptions in Fig. 13.9.

| Event       | Description                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------|
| onabort     | Fires when image transfer has been interrupted by user.                                                                        |
| onchange    | Fires when a new choice is made in a <code>select</code> element, or when a text input is changed and the element loses focus. |
| onclick     | Fires when the user clicks using the mouse.                                                                                    |
| ondblclick  | Fires when the mouse is double clicked.                                                                                        |
| onfocus     | Fires when a form element gains focus.                                                                                         |
| onkeydown   | Fires when the user pushes down a key.                                                                                         |
| onkeypress  | Fires when the user presses then releases a key.                                                                               |
| onkeyup     | Fires when the user releases a key.                                                                                            |
| onload      | Fires when an element and all its children have loaded.                                                                        |
| onmousedown | Fires when a mouse button is pressed down.                                                                                     |
| onmousemove | Fires when the mouse moves.                                                                                                    |
| onmouseout  | Fires when the mouse leaves an element.                                                                                        |
| onmouseover | Fires when the mouse enters an element.                                                                                        |
| onmouseup   | Fires when a mouse button is released.                                                                                         |
| onreset     | Fires when a form resets (i.e., the user clicks a reset button).                                                               |
| onresize    | Fires when the size of an object changes (i.e., the user resizes a window or frame).                                           |
| onselect    | Fires when a text selection begins (applies to <code>input</code> or <code>textarea</code> ).                                  |
| onsubmit    | Fires when a form is submitted.                                                                                                |
| onunload    | Fires when a page is about to unload.                                                                                          |

**Fig. 13.9** | Cross-browser events.

## 13.10 Wrap-Up

This chapter introduced JavaScript events, which allow scripts to respond to user interactions and make web pages more dynamic. We described event handlers and how to register them to specific events on DOM nodes in the script. We introduced the `event` object and the keyword `this`, which allows us to use one event handler to apply a change to the recipient of an event. We discussed various events that fire from mouse actions, as well as the `onfocus`, `onblur`, `onsubmit` and `onreset` events. Finally, we learned about event bubbling, which can produce unexpected effects in scripts that use events. The next chapter introduces XML, an open technology used for data exchange.

## 13.11 Web Resources

<http://www.quirksmode.org/js/introevents.html>

An introduction and reference site for JavaScript events. Includes comprehensive information on history of events, the different event models, and making events work across multiple browsers.

[wsabstract.com/dhtmltutors/domevent1.shtml](http://wsabstract.com/dhtmltutors/domevent1.shtml)

This *JavaScript Kit* tutorial introduces event handling and discusses the W3C DOM advanced event model.

[http://www.w3schools.com/jsref/jsref\\_events.asp](http://www.w3schools.com/jsref/jsref_events.asp)

The W3 School's JavaScript Event Reference site has a comprehensive list of JavaScript events, a description of their usage and their browser compatibilities.

<http://www.brainjar.com/dhtml/events/>

BrainJar.com's DOM Event Model site provides a comprehensive introduction to the DOM event model, and has example code to demonstrate several different ways of assigning and using events.

## Summary

### Section 13.1 Introduction

- JavaScript events allow scripts to respond to user interactions and modify the page accordingly.
- Events and event handling help make web applications more responsive, dynamic and interactive.

### Section 13.2 Registering Event Handlers

- Functions that handle events are called event handlers. Assigning an event handler to an event on a DOM node is called registering an event handler.
- We discuss two models for registering event handlers. The inline model treats events as attributes of XHTML elements.
- To register events using the traditional model, we assign the name of the function to the event property of a DOM node.
- In the inline model, the value of the XHTML attribute is a *JavaScript statement* to be executed when the event occurs.
- In the traditional model, the value of the event property of a DOM node is the name of a *function* to be called when the event occurs.
- Traditional registration of event handlers allows us to quickly and easily assign event handlers to many elements using repetition statements, instead of adding an inline event handler to each XHTML element.

### Section 13.3 Event onload

- The `onload` event fires whenever an element finishes loading successfully.
- If a script in the head attempts to get a DOM node for an XHTML element in the body, `getElementById` returns `null` because the body has not yet loaded.

### Section 13.4 Event onmousemove, the event Object and this

- The `onmousemove` event fires whenever the user moves the mouse.
- The event object stores information about the event that called the event-handling function.
- The event object's `ctrlKey` property contains a boolean which reflects whether the *Ctrl* key was pressed during the event.

- The event object's `shiftKey` property reflects whether the *Shift* key was pressed during the event.
- In an event-handling function, `this` refers to the DOM object on which the event occurred.
- The `this` keyword allows us to use one event handler to apply a change to one of many DOM elements, depending on which one received the event.

### **Section 13.5 Rollovers with `onmouseover` and `onmouseout`**

- When the mouse cursor enters an element, an `onmouseover` event occurs for that element. When the mouse cursor leaves the element, an `onmouseout` event occurs for that element.
- Creating an `Image` object and setting its `src` property preloads the image.
- The event object stores the node on which the action occurred. In Internet Explorer, this node is stored in the event object's `srcElement` property. In Firefox, it is stored in the event object's `target` property.

### **Section 13.6 Form Processing with `onfocus` and `onblur`**

- The `onfocus` event fires when an element gains focus (i.e., when the user clicks a form field or uses the *Tab* key to move between form elements).
- `onblur` fires when an element loses focus, which occurs when another control gains the focus.

### **Section 13.7 More Form Processing with `onsubmit` and `onreset`**

- The `onsubmit` and `onreset` events fire when a form is submitted or reset, respectively.
- An anonymous function is a function that is defined with no name—it is created in nearly the same way as any other function, but with no identifier after the keyword `function`.
- Anonymous functions are useful when creating a function for the sole purpose of assigning it to an event handler.
- The `confirm` method asks the users a question, presenting them with an **OK** button and a **Cancel** button. If the user clicks **OK**, `confirm` returns `true`; otherwise, `confirm` returns `false`.
- By returning either `true` or `false`, event handlers dictate whether the default action for the event is taken.
- If an event handler returns `true` or does not return a value, the default action is taken once the event handler finishes executing.

### **Section 13.8 Event Bubbling**

- Event bubbling is the process whereby events fired in child elements “bubble” up to their parent elements. When an event is fired on an element, it is first delivered to the element's event handler (if any), then to the parent element's event handler (if any).
- If you intend to handle an event in a child element alone, you should cancel the bubbling of the event in the child element's event-handling code by using the `cancelBubble` property of the event object.

## **Terminology**

|                                                           |                             |
|-----------------------------------------------------------|-----------------------------|
| <code>altKey</code> property of event object              | default action for an event |
| anonymous function                                        | event bubbling              |
| <code>cancelBubble</code> property of event object        | event handler               |
| <code>clientX</code> property of event object             | event models                |
| <code>clientY</code> property of event object             | event object                |
| <code>confirm</code> method of <code>window</code> object | event registration models   |
| <code>ctrlKey</code> property of event object             | events in JavaScript        |

|                                     |                                            |
|-------------------------------------|--------------------------------------------|
| fire an event                       | onreset event                              |
| inline model of event registration  | onresize event                             |
| keyboard event                      | onselect event                             |
| keyCode property of an event object | onsubmit event                             |
| mouse event                         | onunload event                             |
| onabort event                       | registering an event handler               |
| onblur event                        | return value of an event handler           |
| onchange event                      | rollover effect                            |
| onclick event                       | screenX property of event object           |
| ondblclick event                    | screenY property of event object           |
| onfocus event                       | setInterval method of window object        |
| onkeydown event                     | shiftkey property of event object          |
| onkeypress event                    | srcElement property of event object        |
| onkeyup event                       | Tab key to switch between fields on a form |
| onload event                        | target property of event object            |
| onmousedown event                   | this keyword                               |
| onmousemove event                   | traditional model of event registration    |
| onmouseout event                    | trigger an event                           |
| onmouseover event                   | type property of event object              |
| onmouseup event                     |                                            |

## Self-Review Exercises

**13.1** Fill in the blanks in each of the following statements:

- Event handlers can be registered in XHTML using the \_\_\_\_\_ model or in JavaScript using the \_\_\_\_\_ model.
- The state of three keys can be retrieved by using the event object. These keys are \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.
- If a child element does not handle an event, \_\_\_\_\_ lets the event rise through the object hierarchy.
- The \_\_\_\_\_ of an event-handling function specifies whether to perform the default action for the event.
- In an event handler, the reference for the id of an element that fired an event is \_\_\_\_\_ in Firefox and \_\_\_\_\_ in Internet Explorer.
- Three events that fire when the user clicks the mouse are \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.

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

- The onload event fires whenever an element starts loading.
- The onclick event fires when the user clicks the mouse on an element.
- The onfocus event fires when an element loses focus.
- When using the rollover effect with images, it is a good practice to create Image objects that preload the desired images.
- Returning true in an event handler on an a (anchor) element prevents the browser from following the link when the event handler finishes.

## Answers to Self-Review Exercises

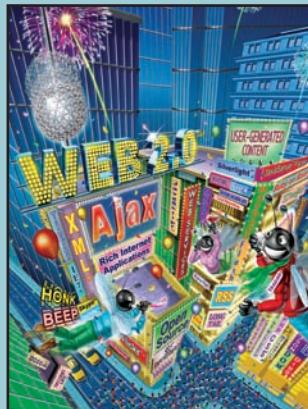
**13.1** a) inline, traditional. b) *Ctrl*, *Alt* and *Shift*. c) event bubbling. d) return value. e) event.target.id, event.srcElement.id. f) onclick, onmousedown, onmouseup.

**13.2** a) False. The onload event fires when an element *finishes* loading. b) True. c) False. It fires when an element gains focus. d) True. e) False. Returning false prevents the default action.

## Exercises

- 13.3** Add an erase feature to the drawing program in Fig. 13.3. Try setting the background color of the table cell over which the mouse moved to `white` when the `Alt` key is pressed.
- 13.4** Add a button to your program from Exercise 13.3 to erase the entire drawing window.
- 13.5** You have a server-side script that cannot handle any ampersands (&) in the form data. Write a function that converts all ampersands in a form field to " and " when the field loses focus (`onblur`).
- 13.6** Write a function that responds to a click anywhere on the page by displaying an `alert` dialog. Display the event name if the user held `Shift` during the mouse click. Display the element name that triggered the event if the user held `Ctrl` during the mouse click.
- 13.7** Use CSS absolute positioning, `onmousedown`, `onmousemove`, `onmouseup` and the `clientX/clientY` properties of the event object to create a program that allows you to drag and drop an image. When the user clicks the image, it should follow the cursor until the mouse button is released.
- 13.8** Modify Exercise 13.7 to allow multiple images to be dragged and dropped in the same page.

# 14



*Knowing trees, I understand  
the meaning of patience.  
Knowing grass, I can  
appreciate persistence.*

—Hal Borland

*Like everything  
metaphysical, the harmony  
between thought and reality  
is to be found in the  
grammar of the language.*

—Ludwig Wittgenstein

*I played with an idea, and  
grew willful; tossed it into  
the air; transformed it; let it  
escape and recaptured it;  
made it iridescent with  
fancy, and winged it with  
paradox.*

—Oscar Wilde

## XML and RSS

### OBJECTIVES

In this chapter you will learn:

- To mark up data using XML.
- How XML namespaces help provide unique XML element and attribute names.
- To create DTDs and schemas for specifying and validating the structure of an XML document.
- To create and use simple XSL style sheets to render XML document data.
- To retrieve and manipulate XML data programmatically using JavaScript.
- RSS and how to programmatically apply an XSL transformation to an RSS document using JavaScript.

## Outline

- 14.1 Introduction
- 14.2 XML Basics
- 14.3 Structuring Data
- 14.4 XML Namespaces
- 14.5 Document Type Definitions (DTDs)
- 14.6 W3C XML Schema Documents
- 14.7 XML Vocabularies
  - 14.7.1 MathML™
  - 14.7.2 Other Markup Languages
- 14.8 Extensible Stylesheet Language and XSL Transformations
- 14.9 Document Object Model (DOM)
- 14.10 RSS
- 14.11 Wrap-Up
- 14.12 Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 14.1 Introduction

The [Extensible Markup Language \(XML\)](#) was developed in 1996 by the [World Wide Web Consortium's \(W3C's\)](#) XML Working Group. XML is a widely supported [open technology](#) (i.e., nonproprietary technology) for describing data that has become the standard format for data exchanged between applications over the Internet.

Web applications use XML extensively and web browsers provide many XML-related capabilities. Sections 14.2–14.7 introduce XML and XML-related technologies—XML namespaces for providing unique XML element and attribute names, and Document Type Definitions (DTDs) and XML Schemas for validating XML documents. These sections support the use of XML in many subsequent chapters. Sections 14.8–14.9 present additional XML technologies and key JavaScript capabilities for loading and manipulating XML documents programmatically—this material is optional but is recommended if you plan to use XML in your own applications. Finally, Section 14.10 introduces RSS—an XML format used to syndicate simple website content—and shows how to format RSS elements using JavaScript and other technologies presented in this chapter.

## 14.2 XML Basics

XML permits document authors to create [markup](#) (i.e., a text-based notation for describing data) for virtually any type of information. This enables document authors to create entirely new markup languages for describing any type of data, such as mathematical formulas, software-configuration instructions, chemical molecular structures, music, news, recipes and financial reports. XML describes data in a way that both human beings and computers can understand.

Figure 14.1 is a simple XML document that describes information for a baseball player. We focus on lines 5–9 to introduce basic XML syntax. You will learn about the other elements of this document in Section 14.3.

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.1: player.xml -->
4 <!-- Baseball player structured with XML -->
5 <player>
6 <firstName>John</firstName>
7 <lastName>Doe</lastName>
8 <battingAverage>0.375</battingAverage>
9 </player>

```

**Fig. 14.1** | XML that describes a baseball player's information.

XML documents contain text that represents content (i.e., data), such as John (line 6 of Fig. 14.1), and **elements** that specify the document's structure, such as `firstName` (line 6 of Fig. 14.1). XML documents delimit elements with **start tags** and **end tags**. A start tag consists of the element name in **angle brackets** (e.g., `<player>` and `<firstName>` in lines 5 and 6, respectively). An end tag consists of the element name preceded by a **forward slash (/)** in angle brackets (e.g., `</firstName>` and `</player>` in lines 6 and 9, respectively). An element's start and end tags enclose text that represents a piece of data (e.g., the player's `firstName`—John—in line 6, which is enclosed by the `<firstName>` start tag and `</firstName>` end tag). Every XML document must have exactly one **root element** that contains all the other elements. In Fig. 14.1, the root element is `player` (lines 5–9).

XML-based markup languages—called XML **vocabularies**—provide a means for describing particular types of data in standardized, structured ways. Some XML vocabularies include XHTML (Extensible HyperText Markup Language), MathML (for mathematics), VoiceXML™ (for speech), CML (Chemical Markup Language—for chemistry), XBRL (Extensible Business Reporting Language—for financial data exchange) and others that we discuss in Section 14.7.

Massive amounts of data are currently stored on the Internet in many formats (e.g., databases, web pages, text files). Much of this data, especially that which is passed between systems, will soon take the form of XML. Organizations see XML as the future of data encoding. Information technology groups are planning ways to integrate XML into their systems. Industry groups are developing custom XML vocabularies for most major industries that will allow business applications to communicate in common languages. For example, many web services allow web-based applications to exchange data seamlessly through standard protocols based on XML. We discuss web services in Chapter 28.

The next generation of the web is being built on an XML foundation, enabling you to develop more sophisticated web-based applications. XML allows you to assign meaning to what would otherwise be random pieces of data. As a result, programs can “understand” the data they manipulate. For example, a web browser might view a street address in a simple web page as a string of characters without any real meaning. In an XML document, however, this data can be clearly identified (i.e., marked up) as an address. A program that uses the document can recognize this data as an address and provide links to a map of that location, driving directions from that location or other location-specific information. Likewise, an application can recognize names of people, dates, ISBN numbers and any other type of XML-encoded data. The application can then present users with other related information, providing a richer, more meaningful user experience.

### *Viewing and Modifying XML Documents*

XML documents are highly portable. Viewing or modifying an XML document—which is a text file that usually ends with the `.xml` filename extension—does not require special software, although many software tools exist, and new ones are frequently released that make it more convenient to develop XML-based applications. Any text editor that supports ASCII/Unicode characters can open XML documents for viewing and editing. Also, most web browsers can display XML documents in a formatted manner that shows the XML’s structure. Section 14.3 demonstrates this in Internet Explorer and Firefox. An important characteristic of XML is that it is both human and machine readable.

### *Processing XML Documents*

Processing an XML document requires software called an **XML parser** (or **XML processor**). A parser makes the document’s data available to applications. While reading an XML document’s contents, a parser checks that the document follows the syntax rules specified by the W3C’s XML Recommendation ([www.w3.org/XML](http://www.w3.org/XML)). XML syntax requires a single root element, a start tag and end tag for each element, and properly nested tags (i.e., the end tag for a nested element must appear before the end tag of the enclosing element). Furthermore, XML is case sensitive, so the proper capitalization must be used in elements. A document that conforms to this syntax is a **well-formed XML document** and is syntactically correct. We present fundamental XML syntax in Section 14.3. If an XML parser can process an XML document successfully, that XML document is well-formed. Parsers can provide access to XML-encoded data in well-formed documents only.

Often, XML parsers are built into software or available for download over the Internet. Some popular parsers include **Microsoft XML Core Services (MSXML)**—which is included with Internet Explorer, the Apache Software Foundation’s **Xerces** ([xml.apache.org](http://xml.apache.org)) and the open-source **Expat XML Parser** ([expat.sourceforge.net](http://expat.sourceforge.net)).

### *Validating XML Documents*

An XML document can reference a **Document Type Definition (DTD)** or a **schema** that defines the proper structure of the XML document. When an XML document references a DTD or a schema, some parsers (called **validating parsers**) can read the DTD/schema and check that the XML document follows the structure defined by the DTD/schema. If the XML document conforms to the DTD/schema (i.e., the document has the appropriate structure), the XML document is **valid**. For example, if in Fig. 14.1 we were referencing a DTD that specified that a `player` element must have `firstName`, `lastName` and `battin-gAverage` elements, then omitting the `lastName` element (line 7 in Fig. 14.1) would invalidate the XML document `player.xml`. However, the XML document would still be well-formed, because it follows proper XML syntax (i.e., it has one root element, each element has a start tag and an end tag, and the elements are nested properly). By definition, a valid XML document is well-formed. Parsers that cannot check for document conformity against DTDs/schemas are **nonvalidating parsers**—they determine only whether an XML document is well-formed, not whether it is valid.

We discuss validation, DTDs and schemas, as well as the key differences between these two types of structural specifications, in Sections 14.5–14.6. For now, note that schemas are XML documents themselves, whereas DTDs are not. As you will learn in Section 14.6, this difference presents several advantages in using schemas over DTDs.



## Software Engineering Observation 14.1

*DTDs and schemas are essential for business-to-business (B2B) transactions and mission-critical systems. Validating XML documents ensures that disparate systems can manipulate data structured in standardized ways and prevents errors caused by missing or malformed data.*

### Formatting and Manipulating XML Documents

Most XML documents contain only data, not formatting instructions, so applications that process XML documents must decide how to manipulate or display the data. For example, a PDA (personal digital assistant) may render an XML document differently than a wireless phone or a desktop computer. You can use [Extensible Stylesheet Language \(XSL\)](#) to specify rendering instructions for different platforms. We discuss XSL in Section 14.8.

XML-processing programs can also search, sort and manipulate XML data using XSL. Some other XML-related technologies are XPath (XML Path Language—a language for accessing parts of an XML document), XSL-FO (XSL Formatting Objects—an XML vocabulary used to describe document formatting) and XSLT (XSL Transformations—a language for transforming XML documents into other documents). We present XSLT and XPath in Section 14.8.

## 14.3 Structuring Data

In this section and throughout this chapter, we create our own XML markup. XML allows you to describe data precisely in a well-structured format.

### XML Markup for an Article

In Fig. 14.2, we present an XML document that marks up a simple article using XML. The line numbers shown are for reference only and are not part of the XML document.

This document begins with an [XML declaration](#) (line 1), which identifies the document as an XML document. The [version attribute](#) specifies the XML version to which the document conforms. The current XML standard is version 1.0. Though the W3C released a version 1.1 specification in February 2004, this newer version is not yet widely supported. The W3C may continue to release new versions as XML evolves to meet the requirements of different fields.

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.2: article.xml -->
4 <!-- Article structured with XML -->
5 <article>
6 <title>Simple XML</title>
7 <date>July 4, 2007</date>
8 <author>
9 <firstName>John</firstName>
10 <lastName>Doe</lastName>
11 </author>
12 <summary>XML is pretty easy.</summary>
13 <content>This chapter presents examples that use XML.</content>
14 </article>
```

**Fig. 14.2** | XML used to mark up an article.



### Portability Tip 14.1

*Documents should include the XML declaration to identify the version of XML used. A document that lacks an XML declaration might be assumed to conform to the latest version of XML—when it does not, errors could result.*

As in most markup languages, blank lines (line 2), white spaces and indentation help improve readability. Blank lines are normally ignored by XML parsers. XML comments (lines 3–4), which begin with `<!--` and end with `-->`, can be placed almost anywhere in an XML document and can span multiple lines. There must be exactly one end marker (`-->`) for each begin marker (`<!--`).



### Common Programming Error 14.1

*Placing any characters, including white space, before the XML declaration is an error.*



### Common Programming Error 14.2

*In an XML document, each start tag must have a matching end tag; omitting either tag is an error. Soon, you will learn how such errors are detected.*



### Common Programming Error 14.3

*XML is case sensitive. Using different cases for the start tag and end tag names for the same element is a syntax error.*

In Fig. 14.2, `article` (lines 5–14) is the root element. The lines that precede the root element (lines 1–4) are the XML **prolog**. In an XML prolog, the XML declaration must appear before the comments and any other markup.

The elements we use in the example do not come from any specific markup language. Instead, we chose the element names and markup structure that best describe our particular data. You can invent elements to mark up your data. For example, element `title` (line 6) contains text that describes the article’s title (e.g., `Simple XML`). Similarly, `date` (line 7), `author` (lines 8–11), `firstName` (line 9), `lastName` (line 10), `summary` (line 12) and `content` (line 13) contain text that describes the date, author, the author’s first name, the author’s last name, a summary and the content of the document, respectively. XML element names can be of any length and may contain letters, digits, underscores, hyphens and periods. However, they must begin with either a letter or an underscore, and they should not begin with “`xml`” in any combination of uppercase and lowercase letters (e.g., `XML`, `Xm1`, `xM1`), as this is reserved for use in the XML standards.



### Common Programming Error 14.4

*Using a white-space character in an XML element name is an error.*



### Good Programming Practice 14.1

*XML element names should be meaningful to humans and should not use abbreviations.*

XML elements are **nested** to form hierarchies—with the root element at the top of the hierarchy. This allows document authors to create parent/child relationships between data. For example, elements `title`, `date`, `author`, `summary` and `content` are nested within

`article`. Elements `firstName` and `lastName` are nested within `author`. We discuss the hierarchy of Fig. 14.2 later in this chapter (Fig. 14.25).



### Common Programming Error 14.5

*Nesting XML tags improperly is a syntax error. For example, <x><y>hello</x></y> is an error, because the </y> tag must precede the </x> tag.*

Any element that contains other elements (e.g., `article` or `author`) is a **container element**. Container elements also are called **parent elements**. Elements nested inside a container element are **child elements** (or children) of that container element. If those child elements are at the same nesting level, they are **siblings** of one another.

### Viewing an XML Document in Internet Explorer and Firefox

The XML document in Fig. 14.2 is simply a text file named `article.xml`. This document does not contain formatting information for the article. This is because XML is a technology for describing the structure of data. Formatting and displaying data from an XML document are application-specific issues. For example, when the user loads `article.xml` in Internet Explorer, MSXML (Microsoft XML Core Services) parses and displays the document's data. Firefox has a similar capability. Each browser has a built-in **style sheet** to format the data. Note that the resulting format of the data (Fig. 14.3) is similar to the format of the listing in Fig. 14.2. In Section 14.8, we show how to create style sheets to transform your XML data into various formats suitable for display.

Note the minus sign (-) and plus sign (+) in the screen shots of Fig. 14.3. Although these symbols are not part of the XML document, both browsers place them next to every container element. A minus sign indicates that the browser is displaying the container element's child elements. Clicking the minus sign next to an element collapses that element

a) `article.xml` in Internet Explorer.

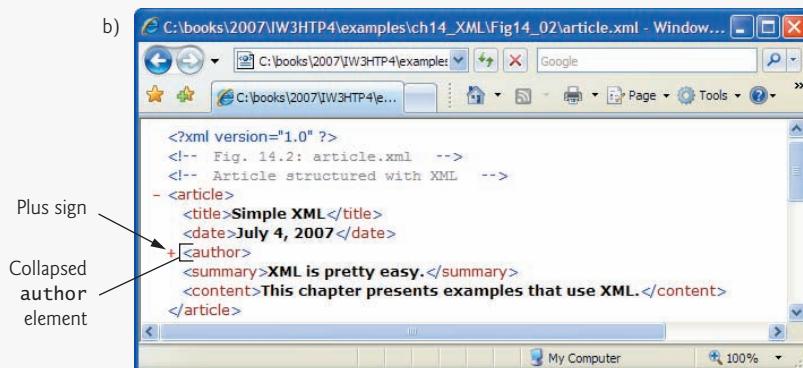
```

<?xml version="1.0" ?>
<!-- Fig. 14.2: article.xml -->
<!-- Article structured with XML -->
- <article>
 <title>Simple XML</title>
 <date>July 4, 2007</date>
 - <author>
 <firstName>John</firstName>
 <lastName>Doe</lastName>
 </author>
 <summary>XML is pretty easy.</summary>
 <content>This chapter presents examples that use XML.</content>
</article>

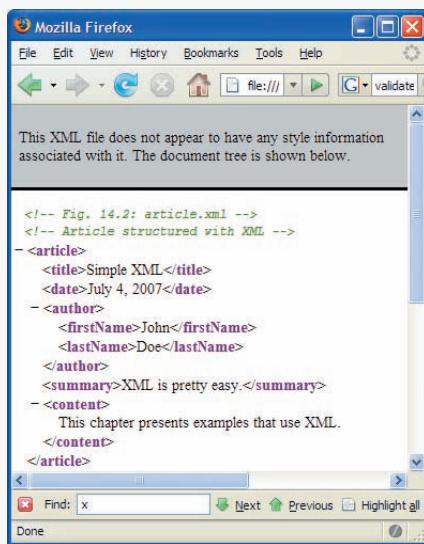
```

Fig. 14.3 | `article.xml` displayed by Internet Explorer 7 and Firefox 2. (Part 1 of 2.)

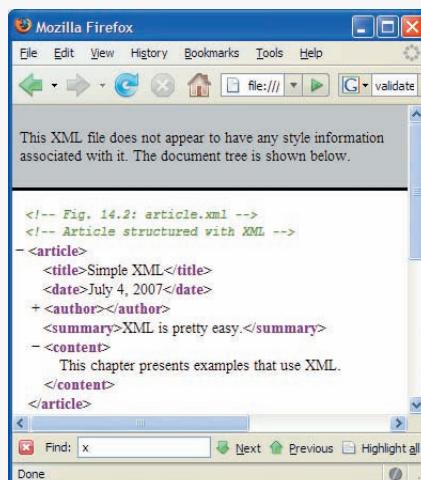
b) article.xml in Internet Explorer with author element collapsed.



c) article.xml in Firefox.



d) article.xml in Firefox with author element collapsed.



**Fig. 14.3 |** article.xml displayed by Internet Explorer 7 and Firefox 2. (Part 2 of 2.)

(i.e., causes the browser to hide the container element's children and replace the minus sign with a plus sign). Conversely, clicking the plus sign next to an element expands that element (i.e., causes the browser to display the container element's children and replace the plus sign with a minus sign). This behavior is similar to viewing the directory structure on your system in Windows Explorer or another similar directory viewer. In fact, a directory structure often is modeled as a series of tree structures, in which the **root** of a tree represents a disk drive (e.g., C:), and **nodes** in the tree represent directories. Parsers often store XML data as tree structures to facilitate efficient manipulation, as discussed in Section 14.9.

[*Note:* In Windows XP Service Pack 2 and Windows Vista, by default Internet Explorer displays all the XML elements in expanded view, and clicking the minus sign (Fig. 14.3(a)) does not do anything. To enable collapsing and expanding, right click the *Information Bar* that appears just below the **Address** field and select **Allow Blocked Content....** Then click **Yes** in the pop-up window that appears.]

### *XML Markup for a Business Letter*

Now that you've seen a simple XML document, let's examine a more complex XML document that marks up a business letter (Fig. 14.4). Again, we begin the document with the XML declaration (line 1) that states the XML version to which the document conforms.

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.4: letter.xml -->
4 <!-- Business letter marked up as XML -->
5 <!DOCTYPE letter SYSTEM "letter.dtd">
6
7 <letter>
8 <contact type = "sender">
9 <name>Jane Doe</name>
10 <address1>Box 12345</address1>
11 <address2>15 Any Ave.</address2>
12 <city>Othertown</city>
13 <state>Otherstate</state>
14 <zip>67890</zip>
15 <phone>555-4321</phone>
16 <flag gender = "F" />
17 </contact>
18
19 <contact type = "receiver">
20 <name>John Doe</name>
21 <address1>123 Main St.</address1>
22 <address2></address2>
23 <city>Anytown</city>
24 <state>Anystate</state>
25 <zip>12345</zip>
26 <phone>555-1234</phone>
27 <flag gender = "M" />
28 </contact>
29
30 <salutation>Dear Sir:</salutation>
31
32 <paragraph>It is our privilege to inform you about our new database
33 managed with XML. This new system allows you to reduce the
34 load on your inventory list server by having the client machine
35 perform the work of sorting and filtering the data.
36 </paragraph>
37
38 <paragraph>Please visit our website for availability and pricing.
39 </paragraph>
40

```

**Fig. 14.4** | Business letter marked up as XML. (Part I of 2.)

```

41 <closing>Sincerely,</closing>
42 <signature>Ms. Jane Doe</signature>
43 </letter>
```

**Fig. 14.4** | Business letter marked up as XML. (Part 2 of 2.)

Line 5 specifies that this XML document references a DTD. Recall from Section 14.2 that DTDs define the structure of the data for an XML document. For example, a DTD specifies the elements and parent/child relationships between elements permitted in an XML document.



### Error-Prevention Tip 14.1

*An XML document is not required to reference a DTD, but validating XML parsers can use a DTD to ensure that the document has the proper structure.*



### Portability Tip 14.2

*Validating an XML document helps guarantee that independent developers will exchange data in a standardized form that conforms to the DTD.*

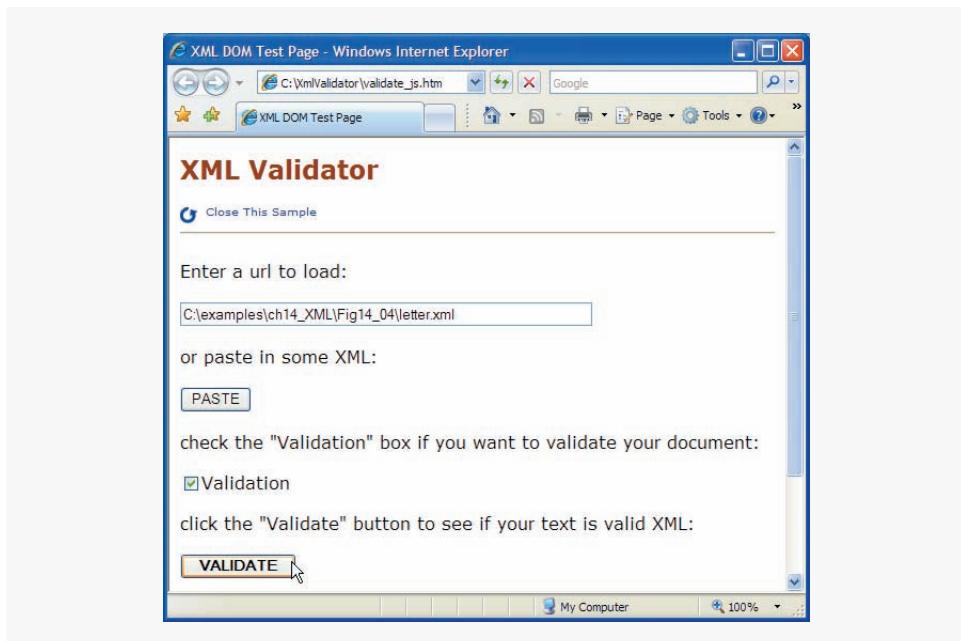
The DOCTYPE reference (line 5) contains three items, the name of the root element that the DTD specifies (`letter`); the keyword `SYSTEM` (which denotes an `external DTD`—a DTD declared in a separate file, as opposed to a DTD declared locally in the same file); and the DTD’s name and location (i.e., `letter.dtd` in the current directory; this could also be a fully qualified URL). DTD document filenames typically end with the `.dtd` extension. We discuss DTDs and `letter.dtd` in detail in Section 14.5.

Several tools (many of which are free) validate documents against DTDs (discussed in Section 14.5) and schemas (discussed in Section 14.6). Microsoft’s [XML Validator](#) is available free of charge from the [Download sample](#) link at

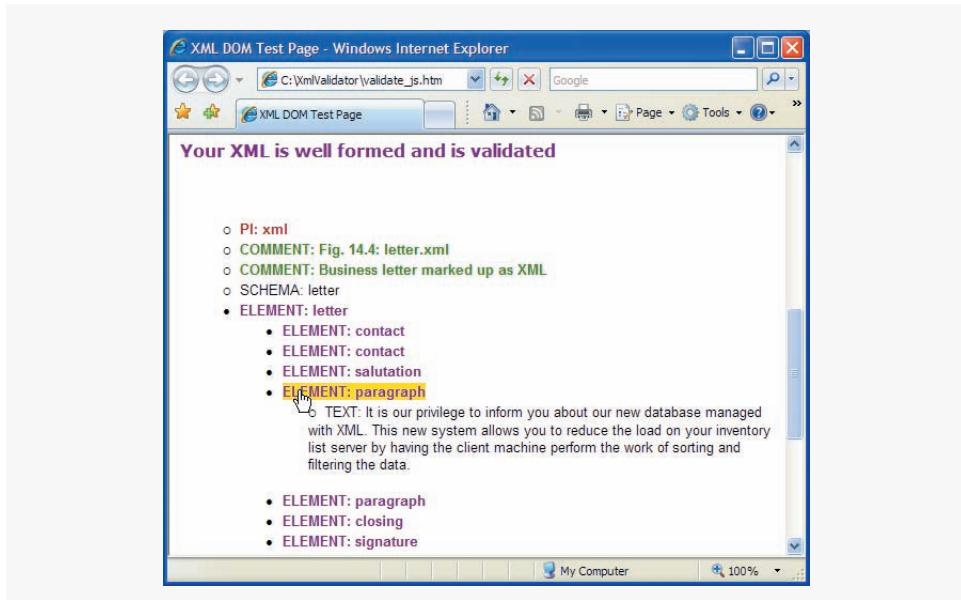
```
msdn.microsoft.com/archive/en-us/samples/internet/
xml/xml_validator/default.asp
```

This validator can validate XML documents against both DTDs and schemas. To install it, run the downloaded executable file `xml_validator.exe` and follow the steps to complete the installation. Once the installation is successful, open the `validate_js.htm` file located in your XML Validator installation directory in IE to validate your XML documents. We installed the XML Validator at `C:\XMLValidator` (Fig. 14.5). The output (Fig. 14.6) shows the results of validating the document using Microsoft’s XML Validator. You can click a node to expand it and see its contents. Visit [www.w3.org/XML/Schema](#) for a list of additional validation tools.

Root element `letter` (lines 7–43 of Fig. 14.4) contains the child elements `contact`, `contact`, `salutation`, `paragraph`, `paragraph`, `closing` and `signature`. Data can be placed between an elements’ tags or as `attributes`—name/value pairs that appear within the angle brackets of an element’s start tag. Elements can have any number of attributes (separated by spaces) in their start tags. The first `contact` element (lines 8–17) has an attribute named `type` with `attribute value "sender"`, which indicates that this `contact` element identifies the letter’s sender. The second `contact` element (lines 19–28) has attribute `type` with value `"receiver"`, which indicates that this `contact` element identifies the letter’s recipient. Like element names, attribute names are case sensitive, can be any



**Fig. 14.5** | Validating an XML document with Microsoft's XML Validator.



**Fig. 14.6** | Validation result using Microsoft's XML Validator.

length, may contain letters, digits, underscores, hyphens and periods, and must begin with either a letter or an underscore character. A **contact** element stores various items of information about a contact, such as the contact's name (represented by element **name**), address

(represented by elements `address1`, `address2`, `city`, `state` and `zip`), phone number (represented by element `phone`) and gender (represented by attribute `gender` of element `flag`). Element `salutation` (line 30) marks up the letter's salutation. Lines 32–39 mark up the letter's body using two paragraph elements. Elements `closing` (line 41) and `signature` (line 42) mark up the closing sentence and the author's "signature," respectively.



### Common Programming Error 14.6

*Failure to enclose attribute values in double ("") or single (' ') quotes is a syntax error.*

Line 16 introduces the `empty element` flag. An empty element is one that does not have any content. Instead, an empty element sometimes places data in attributes. Empty element flag has one attribute that indicates the gender of the contact (represented by the parent `contact` element). Document authors can close an empty element either by placing a slash immediately preceding the right angle bracket, as shown in line 16, or by explicitly writing an end tag, as in line 22

```
<address2></address2>
```

Note that the `address2` element in line 22 is empty because there is no second part to this contact's address. However, we must include this element to conform to the structural rules specified in the XML document's DTD—`letter.dtd` (which we present in Section 14.5). This DTD specifies that each `contact` element must have an `address2` child element (even if it is empty). In Section 14.5, you will learn how DTDs indicate required and optional elements.

## 14.4 XML Namespaces

XML allows document authors to create custom elements. This extensibility can result in **naming collisions** among elements in an XML document that each have the same name. For example, we may use the element `book` to mark up data about a Deitel publication. A stamp collector may use the element `book` to mark up data about a book of stamps. Using both of these elements in the same document could create a naming collision, making it difficult to determine which kind of data each element contains.

An XML **namespace** is a collection of element and attribute names. XML namespaces provide a means for document authors to unambiguously refer to elements with the same name (i.e., prevent collisions). For example,

```
<subject>Geometry</subject>
```

and

```
<subject>Cardiology</subject>
```

use element `subject` to mark up data. In the first case, the subject is something one studies in school, whereas in the second case, the subject is a field of medicine. Namespaces can differentiate these two `subject` elements—for example:

```
<highschool:subject>Geometry</highschool:subject>
```

and

```
<medicalschool:subject>Cardiology</medicalschool:subject>
```

Both `highschool` and `medicalschool` are **namespace prefixes**. A document author places a namespace prefix and colon (:) before an element name to specify the namespace to which that element belongs. Document authors can create their own namespace prefixes using virtually any name except the reserved namespace prefix `xml`. In the next subsections, we demonstrate how document authors ensure that namespaces are unique.



### Common Programming Error 14.7

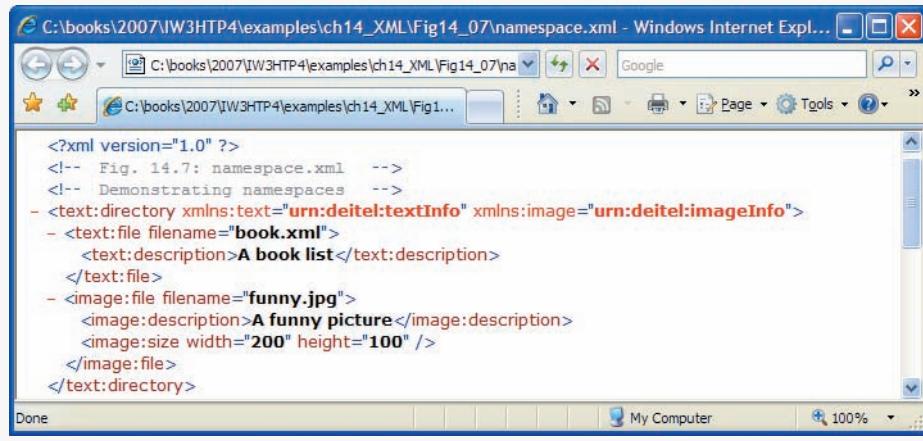
*Attempting to create a namespace prefix named `xml` in any mixture of uppercase and lowercase letters is a syntax error—the `xml` namespace prefix is reserved for internal use by XML itself.*

#### Differentiating Elements with Namespaces

Figure 14.7 demonstrates namespaces. In this document, namespaces differentiate two distinct elements—the `file` element related to a text file and the `file` document related to an image file.

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.7: namespace.xml -->
4 <!-- Demonstrating namespaces -->
5 <text:directory
6 xmlns:text = "urn:deitel:textInfo"
7 xmlns:image = "urn:deitel:imageInfo">
8
9 <text:file filename = "book.xml">
10 <text:description>A book list</text:description>
11 </text:file>
12
13 <image:file filename = "funny.jpg">
14 <image:description>A funny picture</image:description>
15 <image:size width = "200" height = "100" />
16 </image:file>
17 </text:directory>
```



**Fig. 14.7** | XML namespaces demonstration.

Lines 6–7 use the XML-namespace reserved attribute `xmlns` to create two namespace prefixes—`text` and `image`. Each namespace prefix is bound to a series of characters called a **Uniform Resource Identifier (URI)** that uniquely identifies the namespace. Document authors create their own namespace prefixes and URIs. A URI is a way to identify a resource, typically on the Internet. Two popular types of URI are **Uniform Resource Name (URN)** and **Uniform Resource Locator (URL)**.

To ensure that namespaces are unique, document authors must provide unique URIs. In this example, we use the `text urn:deitel:textInfo` and `urn:deitel:imageInfo` as URIs. These URIs employ the URN scheme frequently used to identify namespaces. Under this naming scheme, a URI begins with "urn:", followed by a unique series of additional names separated by colons.

Another common practice is to use URLs, which specify the location of a file or a resource on the Internet. For example, `www.deitel.com` is the URL that identifies the home page of the Deitel & Associates website. Using URLs guarantees that the namespaces are unique because the domain names (e.g., `www.deitel.com`) are guaranteed to be unique. For example, lines 5–7 could be rewritten as

```
<text:directory
 xmlns:text = "http://www.deitel.com/xmlns-text"
 xmlns:image = "http://www.deitel.com/xmlns-image">
```

where URLs related to the `deitel.com` domain name serve as URIs to identify the `text` and `image` namespaces. The parser does not visit these URLs, nor do these URLs need to refer to actual web pages. They each simply represent a unique series of characters used to differentiate URI names. In fact, any string can represent a namespace. For example, our `image` namespace URI could be `hgjfkdlsa4556`, in which case our prefix assignment would be

```
xmlns:image = "hgjfkdlsa4556"
```

Lines 9–11 use the `text` namespace prefix for elements `file` and `description`. Note that the end tags must also specify the namespace prefix `text`. Lines 13–16 apply namespace prefix `image` to the elements `file`, `description` and `size`. Note that attributes do not require namespace prefixes (although they can have them), because each attribute is already part of an element that specifies the namespace prefix. For example, attribute `filename` (line 9) is implicitly part of namespace `text` because its element (i.e., `file`) specifies the `text` namespace prefix.

### *Specifying a Default Namespace*

To eliminate the need to place namespace prefixes in each element, document authors may specify a **default namespace** for an element and its children. Figure 14.8 demonstrates using a default namespace (`urn:deitel:textInfo`) for element `directory`.

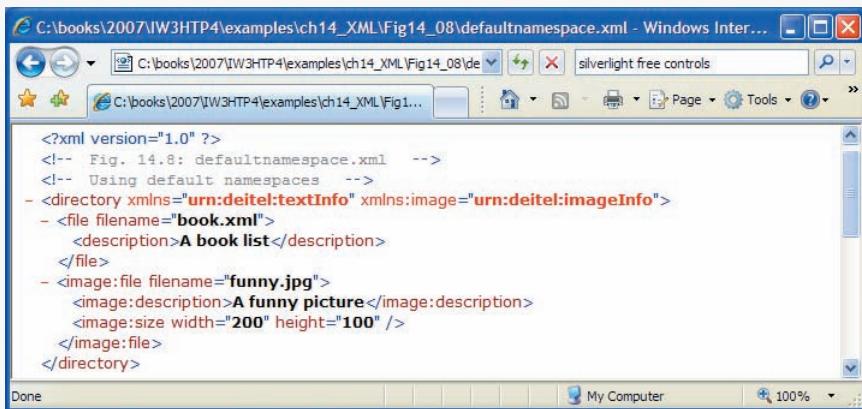
Line 5 defines a default namespace using attribute `xmlns` with no prefix specified, but with a URI as its value. Once we define this default namespace, child elements belonging to the namespace need not be qualified by a namespace prefix. Thus, element `file` (lines 8–10) is in the default namespace `urn:deitel:textInfo`. Compare this to lines 9–10 of Fig. 14.7, where we had to prefix the `file` and `description` element names with the namespace prefix `text`.

The default namespace applies to the `directory` element and all elements that are not qualified with a namespace prefix. However, we can use a namespace prefix to specify a

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.8: defaultnamespace.xml -->
4 <!-- Using default namespaces -->
5 <directory xmlns = "urn:deitel:textInfo"
6 xmlns:image = "urn:deitel:imageInfo">
7
8 <file filename = "book.xml">
9 <description>A book list</description>
10 </file>
11
12 <image:file filename = "funny.jpg">
13 <image:description>A funny picture</image:description>
14 <image:size width = "200" height = "100" />
15 </image:file>
16 </directory>

```



**Fig. 14.8** | Default namespace demonstration.

different namespace for a particular element. For example, the `file` element in lines 12–15 includes the `image` namespace prefix, indicating that this element is in the `urn:deitel:imageInfo` namespace, not the default namespace.

#### *Namespaces in XML Vocabularies*

XML-based languages, such as XML Schema (Section 14.6) and Extensible Stylesheet Language (XSL) (Section 14.8), often use namespaces to identify their elements. Each of these vocabularies defines special-purpose elements that are grouped in namespaces. These namespaces help prevent naming collisions between predefined elements and user-defined elements.

## 14.5 Document Type Definitions (DTDs)

Document Type Definitions (DTDs) are one of two main types of documents you can use to specify XML document structure. Section 14.6 presents W3C XML Schema documents, which provide an improved method of specifying XML document structure.



## Software Engineering Observation 14.2

*XML documents can have many different structures, and for this reason an application cannot be certain whether a particular document it receives is complete, ordered properly, and not missing data. DTDs and schemas (Section 14.6) solve this problem by providing an extensible way to describe XML document structure. Applications should use DTDs or schemas to confirm whether XML documents are valid.*



## Software Engineering Observation 14.3

*Many organizations and individuals are creating DTDs and schemas for a broad range of applications. These collections—called **repositories**—are available free for download from the web (e.g., [www.xml.org](http://www.xml.org), [www.oasis-open.org](http://www.oasis-open.org)).*

### Creating a Document Type Definition

Figure 14.4 presented a simple business letter marked up with XML. Recall that line 5 of `letter.xml` references a DTD—`letter.dtd` (Fig. 14.9). This DTD specifies the business letter’s element types and attributes, and their relationships to one another.

A DTD describes the structure of an XML document and enables an XML parser to verify whether an XML document is valid (i.e., whether its elements contain the proper attributes and appear in the proper sequence). DTDs allow users to check document structure and to exchange data in a standardized format. A DTD expresses the set of rules for document structure using an EBNF (Extended Backus-Naur Form) grammar. DTDs are not themselves XML documents. [Note: EBNF grammars are commonly used to define programming languages. To learn more about EBNF grammars, visit [en.wikipedia.org/wiki/EBNF](https://en.wikipedia.org/wiki/EBNF) or [www.garshol.priv.no/download/text/bnf.html](http://www.garshol.priv.no/download/text/bnf.html).]

```

1 <!-- Fig. 14.9: letter.dtd -->
2 <!-- DTD document for letter.xml -->
3
4 <!ELEMENT letter (contact+, salutation, paragraph+,
5 closing, signature)>
6
7 <!ELEMENT contact (name, address1, address2, city, state,
8 zip, phone, flag)>
9 <!ATTLIST contact type CDATA #IMPLIED>
10
11 <!ELEMENT name (#PCDATA)>
12 <!ELEMENT address1 (#PCDATA)>
13 <!ELEMENT address2 (#PCDATA)>
14 <!ELEMENT city (#PCDATA)>
15 <!ELEMENT state (#PCDATA)>
16 <!ELEMENT zip (#PCDATA)>
17 <!ELEMENT phone (#PCDATA)>
18 <!ELEMENT flag EMPTY>
19 <!ATTLIST flag gender (M | F) "M">
20
21 <!ELEMENT salutation (#PCDATA)>
22 <!ELEMENT closing (#PCDATA)>
23 <!ELEMENT paragraph (#PCDATA)>
24 <!ELEMENT signature (#PCDATA)>
```

**Fig. 14.9** | Document Type Definition (DTD) for a business letter.



## Common Programming Error 14.8

*For documents validated with DTDs, any document that uses elements, attributes or nesting relationships not explicitly defined by a DTD is an invalid document.*

### Defining Elements in a DTD

The **ELEMENT element type declaration** in lines 4–5 defines the rules for element `letter`. In this case, `letter` contains one or more `contact` elements, one `salutation` element, one or more `paragraph` elements, one `closing` element and one `signature` element, in that sequence. The **plus sign (+) occurrence indicator** specifies that the DTD requires one or more occurrences of an element. Other occurrence indicators include the **asterisk (\*)**, which indicates an optional element that can occur zero or more times, and the **question mark (?)**, which indicates an optional element that can occur at most once (i.e., zero or one occurrence). If an element does not have an occurrence indicator, the DTD requires exactly one occurrence.

The `contact` element type declaration (lines 7–8) specifies that a `contact` element contains child elements `name`, `address1`, `address2`, `city`, `state`, `zip`, `phone` and `flag`—in that order. The DTD requires exactly one occurrence of each of these elements.

### Defining Attributes in a DTD

Line 9 uses the **ATTLIST attribute-list declaration** to define an attribute named `type` for the `contact` element. Keyword **#IMPLIED** specifies that if the parser finds a `contact` element without a `type` attribute, the parser can choose an arbitrary value for the attribute or can ignore the attribute. Either way the document will still be valid (if the rest of the document is valid)—a missing `type` attribute will not invalidate the document. Other keywords that can be used in place of `#IMPLIED` in an ATTLIST declaration include **#REQUIRED** and **#FIXED**. Keyword **#REQUIRED** specifies that the attribute must be present in the element, and keyword **#FIXED** specifies that the attribute (if present) must have the given fixed value. For example,

```
<!ATTLIST address zip CDATA #FIXED "01757">
```

indicates that attribute `zip` (if present in element `address`) must have the value `01757` for the document to be valid. If the attribute is not present, then the parser, by default, uses the fixed value that the ATTLIST declaration specifies.

### Character Data vs. Parsed Character Data

Keyword **CDATA** (line 9) specifies that attribute `type` contains **character data** (i.e., a string). A parser will pass such data to an application without modification.



## Software Engineering Observation 14.4

*DTD syntax cannot describe an element's or attribute's data type. For example, a DTD cannot specify that a particular element or attribute can contain only integer data.*

Keyword **#PCDATA** (line 11) specifies that an element (e.g., `name`) may contain **parsed character data** (i.e., data that is processed by an XML parser). Elements with parsed character data cannot contain markup characters, such as less than (`<`), greater than (`>`) or ampersand (`&`). The document author should replace any markup character in a `#PCDATA` element with the character's corresponding **character entity reference**. For example, the

character entity reference &lt; should be used in place of the less-than symbol (<), and the character entity reference &gt; should be used in place of the greater-than symbol (>). A document author who wishes to use a literal ampersand should use the entity reference & instead—parsed character data can contain ampersands (&) only for inserting entities. See Appendix A, XHTML Special Characters, for a list of other character entity references.



### Common Programming Error 14.9

*Using markup characters (e.g., <, > and &) in parsed character data is an error. Use character entity references (e.g., &lt;, &gt; and &amp;) instead.*

#### Defining Empty Elements in a DTD

Line 18 defines an empty element named flag. Keyword EMPTY specifies that the element does not contain any data between its start and end tags. Empty elements commonly describe data via attributes. For example, flag's data appears in its gender attribute (line 19). Line 19 specifies that the gender attribute's value must be one of the enumerated values (M or F) enclosed in parentheses and delimited by a vertical bar (|) meaning “or.” Note that line 19 also indicates that gender has a default value of M.

#### Well-Formed Documents vs. Valid Documents

In Section 14.3, we demonstrated how to use the Microsoft XML Validator to validate an XML document against its specified DTD. The validation revealed that the XML document letter.xml (Fig. 14.4) is well-formed and valid—it conforms to letter.dtd (Fig. 14.9). Recall that a well-formed document is syntactically correct (i.e., each start tag has a corresponding end tag, the document contains only one root element, etc.), and a valid document contains the proper elements with the proper attributes in the proper sequence. An XML document cannot be valid unless it is well-formed.

When a document fails to conform to a DTD or a schema, the Microsoft XML Validator displays an error message. For example, the DTD in Fig. 14.9 indicates that a contact element must contain the child element name. A document that omits this child element is still well-formed, but is not valid. In such a scenario, Microsoft XML Validator displays the error message shown in Fig. 14.10.

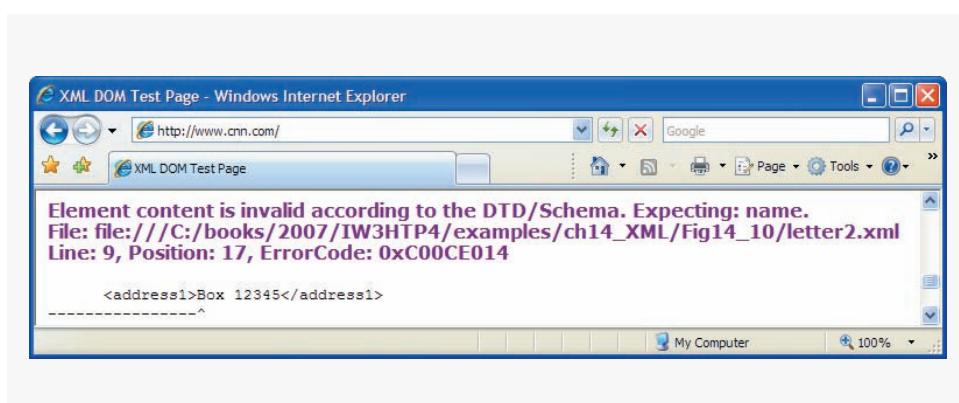


Fig. 14.10 | XML Validator displaying an error message.

## 14.6 W3C XML Schema Documents

In this section, we introduce schemas for specifying XML document structure and validating XML documents. Many developers in the XML community believe that DTDs are not flexible enough to meet today's programming needs. For example, DTDs lack a way of indicating what specific type of data (e.g., numeric, text) an element can contain, and DTDs are not themselves XML documents, forcing developers to learn multiple grammars and developers to create multiple types of parsers. These and other limitations have led to the development of schemas.

Unlike DTDs, schemas do not use EBNF grammar. Instead, schemas use XML syntax and are actually XML documents that programs can manipulate. Like DTDs, schemas are used by validating parsers to validate documents.

In this section, we focus on the W3C's **XML Schema** vocabulary (note the capital "S" in "Schema"). We use the term XML Schema in the rest of the chapter whenever we refer to W3C's XML Schema vocabulary. For the latest information on XML Schema, visit [www.w3.org/XML/Schema](http://www.w3.org/XML/Schema). For tutorials on XML Schema concepts beyond what we present here, visit [www.w3schools.com/schema/default.asp](http://www.w3schools.com/schema/default.asp).

Recall that a DTD describes an XML document's structure, not the content of its elements. For example,

```
<quantity>5</quantity>
```

contains character data. If the document that contains element `quantity` references a DTD, an XML parser can validate the document to confirm that this element indeed does contain PCDATA content. However, the parser cannot validate that the content is numeric; DTDs do not provide this capability. So, unfortunately, the parser also considers

```
<quantity>hello</quantity>
```

to be valid. An application that uses the XML document containing this markup should test that the data in element `quantity` is numeric and take appropriate action if it is not.

XML Schema enables schema authors to specify that element `quantity`'s data must be numeric or, even more specifically, an integer. A parser validating the XML document against this schema can determine that 5 conforms and hello does not. An XML document that conforms to a schema document is **schema valid**, and one that does not conform is **schema invalid**. Schemas are XML documents and therefore must themselves be valid.

### *Validating Against an XML Schema Document*

Figure 14.11 shows a schema-valid XML document named `book.xml`, and Fig. 14.12 shows the pertinent XML Schema document (`book.xsd`) that defines the structure for `book.xml`. By convention, schemas use the `.xsd` extension. We used an online XSD schema validator provided at

[www.xmlforasp.net/SchemaValidator.aspx](http://www.xmlforasp.net/SchemaValidator.aspx)

to ensure that the XML document in Fig. 14.11 conforms to the schema in Fig. 14.12. To validate the schema document itself (i.e., `book.xsd`) and produce the output shown in Fig. 14.12, we used an online XSV (XML Schema Validator) provided by the W3C at

[www.w3.org/2001/03/webdata/xsv](http://www.w3.org/2001/03/webdata/xsv)

These tools are free and enforce the W3C's specifications regarding XML Schemas and schema validation.

```

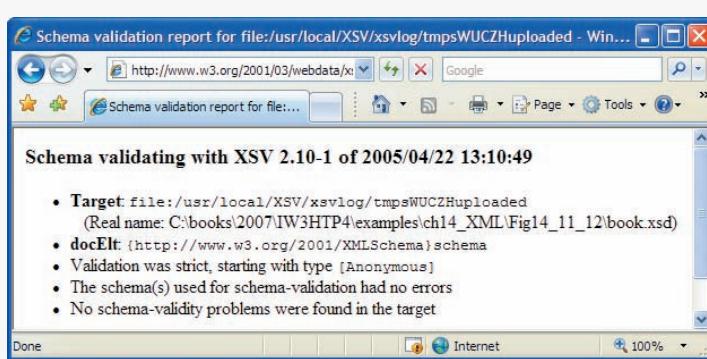
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.11: book.xml -->
4 <!-- Book list marked up as XML -->
5 <deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
6 <book>
7 <title>Visual Basic 2005 How to Program, 3/e</title>
8 </book>
9 <book>
10 <title>Visual C# 2005 How to Program, 2/e</title>
11 </book>
12 <book>
13 <title>Java How to Program, 7/e</title>
14 </book>
15 <book>
16 <title>C++ How to Program, 6/e</title>
17 </book>
18 <book>
19 <title>Internet and World Wide Web How to Program, 4/e</title>
20 </book>
21 </deitel:books>
```

**Fig. 14.11** | Schema-valid XML document describing a list of books.

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.12: book.xsd -->
4 <!-- Simple W3C XML Schema document -->
5 <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6 xmlns:deitel = "http://www.deitel.com/booklist"
7 targetNamespace = "http://www.deitel.com/booklist">
8
9 <element name = "books" type = "deitel:BooksType"/>
10
11 <complexType name = "BooksType">
12 <sequence>
13 <element name = "book" type = "deitel:SingleBookType"
14 minOccurs = "1" maxOccurs = "unbounded"/>
15 </sequence>
16 </complexType>
17
18 <complexType name = "SingleBookType">
19 <sequence>
20 <element name = "title" type = "string"/>
21 </sequence>
22 </complexType>
23 </schema>
```

**Fig. 14.12** | XML Schema document for book.xml. (Part 1 of 2.)



**Fig. 14.12** | XML Schema document for book.xml. (Part 2 of 2.)

Figure 14.11 contains markup describing several Deitel books. The books element (line 5) has the namespace prefix `deitel`, indicating that the books element is a part of the `http://www.deitel.com/booklist` namespace.

#### *Creating an XML Schema Document*

Figure 14.12 presents the XML Schema document that specifies the structure of book.xml (Fig. 14.11). This document defines an XML-based language (i.e., a vocabulary) for writing XML documents about collections of books. The schema defines the elements, attributes and parent/child relationships that such a document can (or must) include. The schema also specifies the type of data that these elements and attributes may contain.

Root element `schema` (Fig. 14.12, lines 5–23) contains elements that define the structure of an XML document such as book.xml. Line 5 specifies as the default namespace the standard W3C XML Schema namespace URI—<http://www.w3.org/2001/XMLSchema>. This namespace contains predefined elements (e.g., root-element `schema`) that comprise the XML Schema vocabulary—the language used to write an XML Schema document.



#### **Portability Tip 14.3**

*W3C XML Schema authors specify URI <http://www.w3.org/2001/XMLSchema> when referring to the XML Schema namespace. This namespace contains predefined elements that comprise the XML Schema vocabulary. Specifying this URI ensures that validation tools correctly identify XML Schema elements and do not confuse them with those defined by document authors.*

Line 6 binds the URI `http://www.deitel.com/booklist` to namespace prefix `deitel`. As we discuss momentarily, the schema uses this namespace to differentiate names created by us from names that are part of the XML Schema namespace. Line 7 also specifies `http://www.deitel.com/booklist` as the `targetNamespace` of the schema. This attribute identifies the namespace of the XML vocabulary that this schema defines. Note that the `targetNamespace` of book.xsd is the same as the namespace referenced in line 5 of book.xml (Fig. 14.11). This is what “connects” the XML document with the schema that defines its structure. When an XML schema validator examines book.xml and book.xsd, it will recognize that book.xml uses elements and attributes from the `http://www.deitel.com/booklist` namespace. The validator also will recognize that this namespace is the namespace defined in book.xsd (i.e., the schema’s `targetNamespace`).

Thus the validator knows where to look for the structural rules for the elements and attributes used in `book.xml`.

### *Defining an Element in XML Schema*

In XML Schema, the `element` tag (line 9) defines an element to be included in an XML document that conforms to the schema. In other words, `element` specifies the actual *elements* that can be used to mark up data. Line 9 defines the `books` element, which we use as the root element in `book.xml` (Fig. 14.11). Attributes `name` and `type` specify the element's name and type, respectively. An element's type indicates the data that the element may contain. Possible types include XML Schema-defined types (e.g., `string`, `double`) and user-defined types (e.g., `BooksType`, which is defined in lines 11–16). Figure 14.13 lists several of XML Schema's many built-in types. For a complete list of built-in types, see Section 3 of the specification found at [www.w3.org/TR/xmlschema-2](http://www.w3.org/TR/xmlschema-2).

| XML Schema type      | Description             | Ranges or structures                                                                                                                                                                                                                         | Examples                          |
|----------------------|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| <code>string</code>  | A character string      |                                                                                                                                                                                                                                              | "hello"                           |
| <code>boolean</code> | True or false           | <code>true</code> , <code>false</code>                                                                                                                                                                                                       | <code>true</code>                 |
| <code>decimal</code> | A decimal numeral       | $i * (10^n)$ , where $i$ is an integer and $n$ is an integer that is less than or equal to zero.                                                                                                                                             | 5, -12, -45.78                    |
| <code>float</code>   | A floating-point number | $m * (2^e)$ , where $m$ is an integer whose absolute value is less than $2^{24}$ and $e$ is an integer in the range -149 to 104. Plus three additional numbers: positive infinity, negative infinity and not-a-number ( <code>NaN</code> ).  | 0, 12, -109.375, <code>NaN</code> |
| <code>double</code>  | A floating-point number | $m * (2^e)$ , where $m$ is an integer whose absolute value is less than $2^{53}$ and $e$ is an integer in the range -1075 to 970. Plus three additional numbers: positive infinity, negative infinity and not-a-number ( <code>NaN</code> ). | 0, 12, -109.375, <code>NaN</code> |
| <code>long</code>    | A whole number          | -9223372036854775808 to 9223372036854775807, inclusive.                                                                                                                                                                                      | 1234567890, -1234567890           |
| <code>int</code>     | A whole number          | -2147483648 to 2147483647, inclusive.                                                                                                                                                                                                        | 1234567890, -1234567890           |

**Fig. 14.13** | Some XML Schema types. (Part 1 of 2.)

| XML Schema type | Description                                     | Ranges or structures                                                                                                     | Examples       |
|-----------------|-------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|----------------|
| short           | A whole number                                  | -32768 to 32767, inclusive.                                                                                              | 12, -345       |
| date            | A date consisting of a year, month and day      | yyyy-mm with an optional dd and an optional time zone, where yyyy is four digits long and mm and dd are two digits long. | 2005-05-10     |
| time            | A time consisting of hours, minutes and seconds | hh:mm:ss with an optional time zone, where hh, mm and ss are two digits long.                                            | 16:30:25-05:00 |

**Fig. 14.13** | Some XML Schema types. (Part 2 of 2.)

In this example, books is defined as an element of type `deitel:BooksType` (line 9). BooksType is a user-defined type (lines 11–16) in the `http://www.deitel.com/booklist` namespace and therefore must have the namespace prefix `deitel`. It is not an existing XML Schema type.

Two categories of type exist in XML Schema—**simple types** and **complex types**. Simple and complex types differ only in that simple types cannot contain attributes or child elements and complex types can.

A user-defined type that contains attributes or child elements must be defined as a complex type. Lines 11–16 use element `complexType` to define BooksType as a complex type that has a child element named book. The `sequence` element (lines 12–15) allows you to specify the sequential order in which child elements must appear. The `element` (lines 13–14) nested within the `complexType` element indicates that a BooksType element (e.g., books) can contain child elements named book of type `deitel:SingleBookType` (defined in lines 18–22). Attribute `minOccurs` (line 14), with value 1, specifies that elements of type BooksType must contain a minimum of one book element. Attribute `maxOccurs` (line 14), with value `unbounded`, specifies that elements of type BooksType may have any number of book child elements.

Lines 18–22 define the complex type SingleBookType. An element of this type contains a child element named title. Line 20 defines element title to be of simple type string. Recall that elements of a simple type cannot contain attributes or child elements. The schema end tag (`</schema>`, line 23) declares the end of the XML Schema document.

### *A Closer Look at Types in XML Schema*

Every element in XML Schema has a type. Types include the built-in types provided by XML Schema (Fig. 14.13) or user-defined types (e.g., SingleBookType in Fig. 14.12).

Every simple type defines a **restriction** on an XML Schema-defined type or a restriction on a user-defined type. Restrictions limit the possible values that an element can hold.

Complex types are divided into two groups—those with **simple content** and those with **complex content**. Both can contain attributes, but only complex content can contain

child elements. Complex types with simple content must extend or restrict some other existing type. Complex types with complex content do not have this limitation. We demonstrate complex types with each kind of content in the next example.

The schema document in Fig. 14.14 creates both simple types and complex types. The XML document in Fig. 14.15 (`laptop.xml`) follows the structure defined in Fig. 14.14 to describe parts of a laptop computer. A document such as `laptop.xml` that conforms to a schema is known as an **XML instance document**—the document is an instance (i.e., example) of the schema.

Line 5 declares the default namespace to be the standard XML Schema namespace—any elements without a prefix are assumed to be in the XML Schema namespace. Line 6 binds the namespace prefix `computer` to the namespace `http://www.deitel.com/computer`. Line 7 identifies this namespace as the `targetNamespace`—the namespace being defined by the current XML Schema document.

To design the XML elements for describing laptop computers, we first create a simple type in lines 9–13 using the `simpleType` element. We name this `simpleType` `gigahertz`

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 14.14: computer.xsd -->
3 <!-- W3C XML Schema document -->
4
5 <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6 xmlns:computer = "http://www.deitel.com/computer"
7 targetNamespace = "http://www.deitel.com/computer">
8
9 <simpleType name = "gigahertz">
10 <restriction base = "decimal">
11 <minInclusive value = "2.1"/>
12 </restriction>
13 </simpleType>
14
15 <complexType name = "CPU">
16 <simpleContent>
17 <extension base = "string">
18 <attribute name = "model" type = "string"/>
19 </extension>
20 </simpleContent>
21 </complexType>
22
23 <complexType name = "portable">
24 <all>
25 <element name = "processor" type = "computer:CPU"/>
26 <element name = "monitor" type = "int"/>
27 <element name = "CPUSpeed" type = "computer:gigahertz"/>
28 <element name = "RAM" type = "int"/>
29 </all>
30 <attribute name = "manufacturer" type = "string"/>
31 </complexType>
32
33 <element name = "laptop" type = "computer:portable"/>
34 </schema>
```

**Fig. 14.14** | XML Schema document defining simple and complex types.

because it will be used to describe the clock speed of the processor in gigahertz. Simple types are restrictions of a type typically called a **base type**. For this **simpleType**, line 10 declares the base type as **decimal**, and we restrict the value to be at least 2.1 by using the **minInclusive** element in line 11.

Next, we declare a **complexType** named **CPU** that has **simpleContent** (lines 16–20). Remember that a complex type with simple content can have attributes but not child elements. Also recall that complex types with simple content must extend or restrict some XML Schema type or user-defined type. The **extension** element with attribute **base** (line 17) sets the base type to **string**. In this **complexType**, we extend the base type **string** with an attribute. The **attribute** element (line 18) gives the **complexType** an attribute of type **string** named **model**. Thus an element of type **CPU** must contain **string** text (because the base type is **string**) and may contain a **model** attribute that is also of type **string**.

Last, we define type **portable**, which is a **complexType** with complex content (lines 23–31). Such types are allowed to have child elements and attributes. The element **all** (lines 24–29) encloses elements that must each be included once in the corresponding XML instance document. These elements can be included in any order. This complex type holds four elements—**processor**, **monitor**, **CPUSpeed** and **RAM**. They are given types **CPU**, **int**, **gigahertz** and **int**, respectively. When using types **CPU** and **gigahertz**, we must include the namespace prefix **computer**, because these user-defined types are part of the **computer** namespace (<http://www.deitel.com/computer>)—the namespace defined in the current document (line 7). Also, **portable** contains an attribute defined in line 30. The **attribute** element indicates that elements of type **portable** contain an attribute of type **string** named **manufacturer**.

Line 33 declares the actual element that uses the three types defined in the schema. The element is called **laptop** and is of type **portable**. We must use the namespace prefix **computer** in front of **portable**.

We have now created an element named **laptop** that contains child elements **processor**, **monitor**, **CPUSpeed** and **RAM**, and an attribute **manufacturer**. Figure 14.15 uses the **laptop** element defined in the **computer.xsd** schema. Once again, we used an online XSD schema validator ([www.xmlforasp.net/SchemaValidator.aspx](http://www.xmlforasp.net/SchemaValidator.aspx)) to ensure that this XML instance document adheres to the schema's structural rules.

Line 5 declares namespace prefix **computer**. The **laptop** element requires this prefix because it is part of the <http://www.deitel.com/computer> namespace. Line 6 sets the

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.15: laptop.xml -->
4 <!-- Laptop components marked up as XML -->
5 <computer:laptop xmlns:computer = "http://www.deitel.com/computer"
6 manufacturer = "IBM">
7
8 <processor model = "Centrino">Intel</processor>
9 <monitor>17</monitor>
10 <CPUSpeed>2.4</CPUSpeed>
11 <RAM>256</RAM>
12 </computer:laptop>
```

**Fig. 14.15** | XML document using the **laptop** element defined in **computer.xsd**.

laptop's `manufacturer` attribute, and lines 8–11 use the elements defined in the schema to describe the laptop's characteristics.

This section introduced W3C XML Schema documents for defining the structure of XML documents, and we validated XML instance documents against schemas using an online XSD schema validator. Section 14.7 discusses several XML vocabularies and demonstrates the MathML vocabulary. Section 14.10 demonstrates the RSS vocabulary.

## 14.7 XML Vocabularies

XML allows authors to create their own tags to describe data precisely. People and organizations in various fields of study have created many different kinds of XML for structuring data. Some of these markup languages are: [MathML \(Mathematical Markup Language\)](#), [Scalable Vector Graphics \(SVG\)](#), [Wireless Markup Language \(WML\)](#), [Extensible Business Reporting Language \(XBRL\)](#), [Extensible User Interface Language \(XUL\)](#) and [Product Data Markup Language \(PDML\)](#). Two other examples of XML vocabularies are W3C XML Schema and the Extensible Stylesheet Language (XSL), which we discuss in Section 14.8. The following subsections describe MathML and other custom markup languages.

### 14.7.1 MathML™

Until recently, computers typically required specialized software packages such as TeX and LaTeX for displaying complex mathematical expressions. This section introduces MathML, which the W3C developed for describing mathematical notations and expressions. One application that can parse, render and edit MathML is the W3C's [Amaya™](#) browser/editor, which can be downloaded from

[www.w3.org/Amaya/User/BinDist.html](http://www.w3.org/Amaya/User/BinDist.html)

This page contains download links for several platforms. Amaya documentation and installation notes also are available at the W3C website. Firefox also can render MathML, but it requires additional fonts. Instructions for downloading and installing these fonts are available at [www.mozilla.org/projects/mathml/fonts/](http://www.mozilla.org/projects/mathml/fonts/). You can download a plug-in ([www.dessci.com/en/products/mathplayer/](http://www.dessci.com/en/products/mathplayer/)) to render MathML in Internet Explorer .

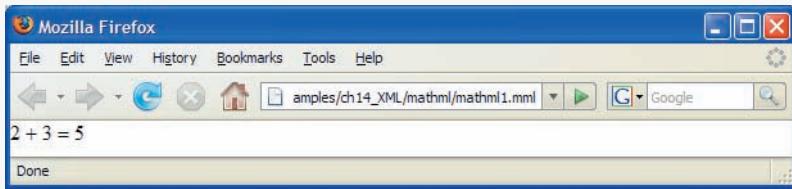
MathML markup describes mathematical expressions for display. MathML is divided into two types of markup—[content](#) markup and [presentation](#) markup. Content markup provides tags that embody mathematical concepts. Content MathML allows programmers to write mathematical notation specific to different areas of mathematics. For instance, the multiplication symbol has one meaning in set theory and another meaning in linear algebra. Content MathML distinguishes between different uses of the same symbol. Programmers can take content MathML markup, discern mathematical context and evaluate the marked-up mathematical operations. Presentation MathML is directed toward formatting and displaying mathematical notation. We focus on Presentation MathML in the MathML examples.

#### *Simple Equation in MathML*

Figure 14.16 uses MathML to mark up a simple expression. For this example, we show the expression rendered in Firefox.

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3 "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5 <!-- Fig. 14.16: mathml1.mml -->
6 <!-- MathML equation. -->
7 <math xmlns="http://www.w3.org/1998/Math/MathML">
8 <mn>2</mn>
9 <mo>+</mo>
10 <mn>3</mn>
11 <mo>=</mo>
12 <mn>5</mn>
13 </math>
```



**Fig. 14.16** | Expression marked up with MathML and displayed in the Firefox browser.

By convention, MathML files end with the `.mml` filename extension. A MathML document's root node is the `math` element, and its default namespace is `http://www.w3.org/1998/Math/MathML` (line 7). The `mn` element (line 8) marks up a number. The `mo` element (line 9) marks up an operator (e.g., `+`). Using this markup, we define the expression  $2 + 3 = 5$ , which any MathML capable browser can display.

### Algebraic Equation in MathML

Let's consider using MathML to mark up an algebraic equation containing exponents and arithmetic operators (Fig. 14.17). For this example, we again show the expression rendered in Firefox.

```

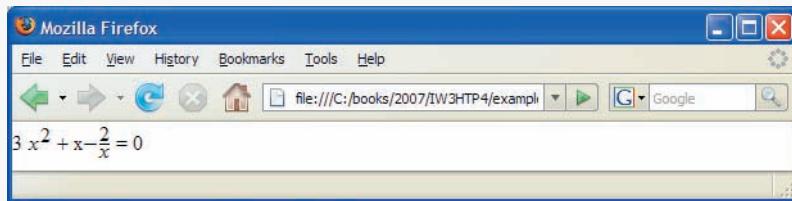
1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3 "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5 <!-- Fig. 14.17: mathml2.html -->
6 <!-- MathML algebraic equation. -->
7 <math xmlns="http://www.w3.org/1998/Math/MathML">
8 <mn>3</mn>
9 <mo>⁢</mo>
10 <msup>
11 <mi>x</mi>
12 <mn>2</mn>
13 </msup>
```

**Fig. 14.17** | Algebraic equation marked up with MathML and displayed in the Firefox browser.  
(Part 1 of 2.)

```

14 <mo>+</mo>
15 <mn>x</mn>
16 <mo>−</mo>
17 <mfrac>
18 <mn>2</mn>
19 <mi>x</mi>
20 </mfrac>
21 <mo>=</mo>
22 <mn>0</mn>
23 </math>

```



**Fig. 14.17** | Algebraic equation marked up with MathML and displayed in the Firefox browser. (Part 2 of 2.)

Line 9 uses **entity reference &InvisibleTimes;** to indicate a multiplication operation without explicit **symbolic representation** (i.e., the multiplication symbol does not appear between the 3 and x). For exponentiation, lines 10–13 use the **m<sup>sup</sup>** element, which represents a superscript. This **m<sup>sup</sup> element** has two children—the expression to be superscripted (i.e., the base) and the superscript (i.e., the exponent). Correspondingly, the **m<sub>sub</sub>** element represents a subscript. To display variables such as x, line 11 uses **identifier element mi**.

To display a fraction, lines 17–20 uses the **mfrac element**. Lines 18–19 specify the numerator and the denominator for the fraction. If either the numerator or the denominator contains more than one element, it must appear in an **mrow** element.

### *Calculus Expression in MathML*

Figure 14.18 marks up a calculus expression that contains an integral symbol and a square-root symbol.

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3 "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5 <!-- Fig. 14.18 mathml3.html -->
6 <!-- Calculus example using MathML -->
7 <math xmlns="http://www.w3.org/1998/Math/MathML">
8 <mrow>
9 <msubsup>
10 <mo>∫</mo>

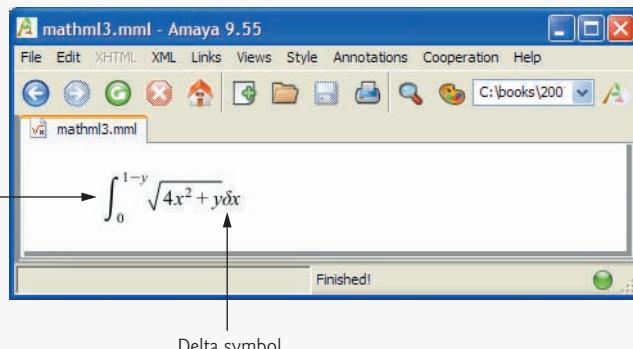
```

**Fig. 14.18** | Calculus expression marked up with MathML and displayed in the Amaya browser. [Courtesy of World Wide Web Consortium (W3C).] (Part 1 of 2.)

```

11 <mn>0</mn>
12 <mrow>
13 <mn>1</mn>
14 <mo>−</mo>
15 <mi>y</mi>
16 </mrow>
17 </msubsup>
18 <msqrt>
19 <mn>4</mn>
20 <mo>⁢</mo>
21 <msup>
22 <mi>x</mi>
23 <mn>2</mn>
24 </msup>
25 <mo>+</mo>
26 <mi>y</mi>
27 </msqrt>
28 <mo>δ</mo>
29 <mi>x</mi>
30 </mrow>
31 </math>

```



**Fig. 14.18** | Calculus expression marked up with MathML and displayed in the Amaya browser. [Courtesy of World Wide Web Consortium (W3C).] (Part 2 of 2.)

Lines 8–30 group the entire expression in an **mrow element**, which is used to group elements that are positioned horizontally in an expression. The entity reference **&int;** (line 10) represents the integral symbol, while the **msubsup element** (lines 9–17) specifies the subscript and superscript a base expression (e.g., the integral symbol). Element **mo** marks up the integral operator. The **msubsup** element requires three child elements—an operator (e.g., the integral entity, line 10), the subscript expression (line 11) and the superscript expression (lines 12–16). Element **mn** (line 11) marks up the number (i.e., 0) that represents the subscript. Element **mrow** (lines 12–16) marks up the superscript expression (i.e.,  $1-y$ ).

Element **msqrt** (lines 18–27) represents a square-root expression. Line 28 introduces entity reference **&delta;** for representing a lowercase delta symbol. Delta is an operator, so line 28 places this entity in element **mo**. To see other operations and symbols in MathML, visit [www.w3.org/Math](http://www.w3.org/Math).

### 14.7.2 Other Markup Languages

Literally hundreds of markup languages derive from XML. Every day developers find new uses for XML. Figure 14.20 summarizes a few of these markup languages. The website

[www.service-architecture.com/xml/articles/index.html](http://www.service-architecture.com/xml/articles/index.html)

provides a nice list of common XML vocabularies and descriptions.

| Markup language                                      | Description                                                                                                                                                                                                                                                                                                                                                                            |
|------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Chemical Markup Language (CML)                       | Chemical Markup Language (CML) is an XML vocabulary for representing molecular and chemical information. Many previous methods for storing this type of information (e.g., special file types) inhibited document reuse. CML takes advantage of XML's portability to enable document authors to use and reuse molecular information without corrupting important data in the process.  |
| VoiceXML™                                            | The VoiceXML Forum founded by AT&T, IBM, Lucent and Motorola developed VoiceXML. It provides interactive voice communication between humans and computers through a telephone, PDA (personal digital assistant) or desktop computer. IBM's VoiceXML SDK can process VoiceXML documents. Visit <a href="http://www.voicexml.org">www.voicexml.org</a> for more information on VoiceXML. |
| Synchronous Multimedia Integration Language (SMIL™)  | SMIL is an XML vocabulary for multimedia presentations. The W3C was the primary developer of SMIL, with contributions from some companies. Visit <a href="http://www.w3.org/AudioVideo">www.w3.org/AudioVideo</a> for more on SMIL.                                                                                                                                                    |
| Research Information Exchange Markup Language (RXML) | RIXML, developed by a consortium of brokerage firms, marks up investment data. Visit <a href="http://www.rixml.org">www.rixml.org</a> for more information on RIXML.                                                                                                                                                                                                                   |
| Geography Markup Language (GML)                      | OpenGIS developed the Geography Markup Language to describe geographic information. Visit <a href="http://www.opengis.org">www.opengis.org</a> for more information on GML.                                                                                                                                                                                                            |
| Extensible User Interface Language (XUL)             | The Mozilla Project created the Extensible User Interface Language for describing graphical user interfaces in a platform-independent way.                                                                                                                                                                                                                                             |

**Fig. 14.19** | Various markup languages derived from XML.

## 14.8 Extensible Stylesheet Language and XSL Transformations

**Extensible Stylesheet Language (XSL)** documents specify how programs are to render XML document data. XSL is a group of three technologies—**XSL-FO (XSL Formatting Objects)**, **XPath (XML Path Language)** and **XSLT (XSL Transformations)**. XSL-FO is

a vocabulary for specifying formatting, and XPath is a string-based language of expressions used by XML and many of its related technologies for effectively and efficiently locating structures and data (such as specific elements and attributes) in XML documents.

The third portion of XSL—XSL Transformations (XSLT)—is a technology for transforming XML documents into other documents—i.e., transforming the structure of the XML document data to another structure. XSLT provides elements that define rules for transforming one XML document to produce a different XML document. This is useful when you want to use data in multiple applications or on multiple platforms, each of which may be designed to work with documents written in a particular vocabulary. For example, XSLT allows you to convert a simple XML document to an XHTML document that presents the XML document's data (or a subset of the data) formatted for display in a web browser.

Transforming an XML document using XSLT involves two tree structures—the **source tree** (i.e., the XML document to be transformed) and the **result tree** (i.e., the XML document to be created). XPath is used to locate parts of the source-tree document that match **templates** defined in an **XSL style sheet**. When a match occurs (i.e., a node matches a template), the matching template executes and adds its result to the result tree. When there are no more matches, XSLT has transformed the source tree into the result tree. The XSLT does not analyze every node of the source tree; it selectively navigates the source tree using XPath's **select** and **match** attributes. For XSLT to function, the source tree must be properly structured. Schemas, DTDs and validating parsers can validate document structure before using XPath and XSLTs.

### *A Simple XSL Example*

Figure 14.20 lists an XML document that describes various sports. The output shows the result of the transformation (specified in the XSLT template of Fig. 14.21) rendered by Internet Explorer.

To perform transformations, an XSLT processor is required. Popular XSLT processors include Microsoft's MSXML and the Apache Software Foundation's **Xalan 2** ([xml.apache.org](http://xml.apache.org)). The XML document in Fig. 14.20 is transformed into an XHTML document by MSXML when the document is loaded in Internet Explorer. MSXML is both an XML parser and an XSLT processor. Firefox also includes an XSLT processor.

```

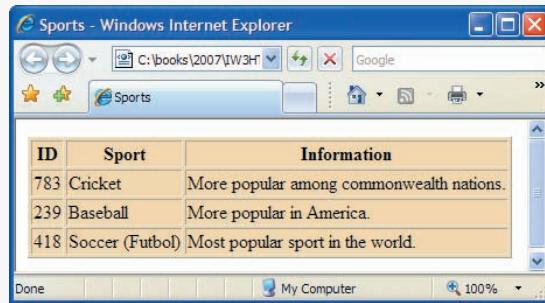
1 <?xml version = "1.0"?>
2 <?xml-stylesheet type = "text/xsl" href = "sports.xsl"?>
3
4 <!-- Fig. 14.20: sports.xml -->
5 <!-- Sports Database -->
6
7 <sports>
8 <game id = "783">
9 <name>Cricket</name>
10
11 <paragraph>
12 More popular among commonwealth nations.
13 </paragraph>
14 </game>
```

**Fig. 14.20** | XML document that describes various sports. (Part I of 2.)

```

15 <game id = "239">
16 <name>Baseball</name>
17
18 <paragraph>
19 More popular in America.
20 </paragraph>
21 </game>
22
23 <game id = "418">
24 <name>Soccer (Futbol)</name>
25
26 <paragraph>
27 Most popular sport in the world.
28 </paragraph>
29 </game>
30
31 </sports>

```



The screenshot shows a Microsoft Internet Explorer window with the title bar 'Sports - Windows Internet Explorer'. The address bar shows 'C:\books\2007\W3H'. The page content displays an XML document with a table:

| ID  | Sport           | Information                              |
|-----|-----------------|------------------------------------------|
| 783 | Cricket         | More popular among commonwealth nations. |
| 239 | Baseball        | More popular in America.                 |
| 418 | Soccer (Futbol) | Most popular sport in the world.         |

**Fig. 14.20** | XML document that describes various sports. (Part 2 of 2.)

Line 2 (Fig. 14.20) is a **processing instruction (PI)** that references the XSL style sheet *sports.xsl* (Fig. 14.21). A processing instruction is embedded in an XML document and provides application-specific information to whichever XML processor the application uses. In this particular case, the processing instruction specifies the location of an XSLT document with which to transform the XML document. The `<?` and `?>` (line 2, Fig. 14.20) delimit a processing instruction, which consists of a **PI target** (e.g., `xmlstylesheet`) and a **PI value** (e.g., `type = "text/xsl" href = "sports.xsl"`). The PI value's `type` attribute specifies that *sports.xsl* is a `text/xsl` file (i.e., a text file containing XSL content). The `href` attribute specifies the name and location of the style sheet to apply—in this case, *sports.xsl* in the current directory.



### Software Engineering Observation 14.5

XSL enables document authors to separate data presentation (specified in XSL documents) from data description (specified in XML documents).



### Common Programming Error 14.10

You will sometimes see the XML processing instruction `<?xmlstylesheet?>` written as `<?xml:stylesheet?>` with a colon rather than a dash. The version with a colon results in an XML parsing error in Firefox.

Figure 14.21 shows the XSL document for transforming the structured data of the XML document of Fig. 14.20 into an XHTML document for presentation. By convention, XSL documents have the filename extension **.xsl**.

Lines 6–7 begin the XSL style sheet with the **stylesheet** start tag. Attribute **version** specifies the XSLT version to which this document conforms. Line 7 binds namespace prefix **xsl** to the W3C's XSLT URI (i.e., <http://www.w3.org/1999/XSL/Transform>).

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 14.21: sports.xsl -->
3 <!-- A simple XSLT transformation -->
4
5 <!-- reference XSL style sheet URI -->
6 <xslstylesheet version = "1.0"
7 xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9 <xsl:output method = "html" omit-xml-declaration = "no"
10 doctype-system =
11 "http://www.w3c.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
12 doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14 <xsl:template match = "/"> <!-- match root element -->
15
16 <html xmlns = "http://www.w3.org/1999/xhtml">
17 <head>
18 <title>Sports</title>
19 </head>
20
21 <body>
22 <table border = "1" bgcolor = "wheat">
23 <thead>
24 <tr>
25 <th>ID</th>
26 <th>Sport</th>
27 <th>Information</th>
28 </tr>
29 </thead>
30
31 <!-- insert each name and paragraph element value -->
32 <!-- into a table row. -->
33 <xsl:for-each select = "/sports/game">
34 <tr>
35 <td><xsl:value-of select = "@id"/></td>
36 <td><xsl:value-of select = "name"/></td>
37 <td><xsl:value-of select = "paragraph"/></td>
38 </tr>
39 </xsl:for-each>
40 </table>
41 </body>
42 </html>
43
44 </xsl:template>
45 </xslstylesheet>
```

**Fig. 14.21** | XSLT that creates elements and attributes in an XHTML document.

Lines 9–12 use element `xsl:output` to write an XHTML document type declaration (DOCTYPE) to the result tree (i.e., the XML document to be created). The DOCTYPE identifies XHTML as the type of the resulting document. Attribute `method` is assigned "html", which indicates that HTML is being output to the result tree. Attribute `omit-xml-declaration` specifies whether the transformation should write the XML declaration to the result tree. In this case, we do not want to omit the XML declaration, so we assign to this attribute the value "no". Attributes `doctype-system` and `doctype-public` write the DOCTYPE DTD information to the result tree.

XSLT uses `templates` (i.e., `xsl:template` elements) to describe how to transform particular nodes from the source tree to the result tree. A template is applied to nodes that are specified in the required `match` attribute. Line 14 uses the `match` attribute to select the `document root` (i.e., the conceptual part of the document that contains the root element and everything below it) of the XML source document (i.e., `sports.xml`). The XPath character `/` (a forward slash) always selects the document root. Recall that XPath is a string-based language used to locate parts of an XML document easily. In XPath, a leading forward slash specifies that we are using `absolute addressing` (i.e., we are starting from the root and defining paths down the source tree). In the XML document of Fig. 14.20, the child nodes of the document root are the two processing instruction nodes (lines 1–2), the two comment nodes (lines 4–5) and the `sports` element node (lines 7–31). The template in Fig. 14.21, line 14, matches a node (i.e., the root node), so the contents of the template are now added to the result tree.

The MSXML processor writes the XHTML in lines 16–29 (Fig. 14.21) to the result tree exactly as it appears in the XSL document. Now the result tree consists of the DOCTYPE definition and the XHTML code from lines 16–29. Lines 33–39 use element `xsl:for-each` to iterate through the source XML document, searching for `game` elements. Attribute `select` is an XPath expression that specifies the nodes (called the `node set`) on which the `xsl:for-each` operates. Again, the first forward slash means that we are using absolute addressing. The forward slash between `sports` and `game` indicates that `game` is a child node of `sports`. Thus, the `xsl:for-each` finds `game` nodes that are children of the `sports` node. The XML document `sports.xml` contains only one `sports` node, which is also the document root node. After finding the elements that match the selection criteria, the `xsl:for-each` processes each element with the code in lines 34–38 (these lines produce one row in a table each time they execute) and places the result of lines 34–38 in the result tree.

Line 35 uses element `value-of` to retrieve attribute `id`'s value and place it in a `td` element in the result tree. The XPath symbol `@` specifies that `id` is an attribute node of the context node `game`. Lines 36–37 place the `name` and `paragraph` element values in `td` elements and insert them in the result tree. When an XPath expression has no beginning forward slash, the expression uses `relative addressing`. Omitting the beginning forward slash tells the `xsl:value-of` `select` statements to search for `name` and `paragraph` elements that are children of the context node, not the root node. Due to the last XPath expression selection, the current context node is `game`, which indeed has an `id` attribute, a `name` child element and a `paragraph` child element.

### *Using XSLT to Sort and Format Data*

Figure 14.22 presents an XML document (`sorting.xml`) that marks up information about a book. Note that several elements of the markup describing the book appear out of

```
1 <?xml version = "1.0"?>
2 <?xml-stylesheet type = "text/xsl" href = "sorting.xsl"?>
3
4 <!-- Fig. 14.22: sorting.xml -->
5 <!-- XML document containing book information -->
6 <book isbn = "999-99999-9-X">
7 <title>Deitel's XML Primer</title>
8
9 <author>
10 <firstName>Jane</firstName>
11 <lastName>Blue</lastName>
12 </author>
13
14 <chapters>
15 <frontMatter>
16 <preface pages = "2" />
17 <contents pages = "5" />
18 <illustrations pages = "4" />
19 </frontMatter>
20
21 <chapter number = "3" pages = "44">Advanced XML</chapter>
22 <chapter number = "2" pages = "35">Intermediate XML</chapter>
23 <appendix number = "B" pages = "26">Parsers and Tools</appendix>
24 <appendix number = "A" pages = "7">Entities</appendix>
25 <chapter number = "1" pages = "28">XML Fundamentals</chapter>
26 </chapters>
27
28 <media type = "CD" />
29 </book>
```

**Fig. 14.22** | XML document containing book information.

order (e.g., the element describing Chapter 3 appears before the element describing Chapter 2). We arranged them this way purposely to demonstrate that the XSL style sheet referenced in line 2 (*sorting.xsl*) can sort the XML file’s data for presentation purposes.

Figure 14.23 presents an XSL document (*sorting.xsl*) for transforming *sorting.xml* (Fig. 14.22) to XHTML. Recall that an XSL document navigates a source tree and builds a result tree. In this example, the source tree is XML, and the output tree is XHTML. Line 14 of Fig. 14.23 matches the root element of the document in Fig. 14.22. Line 15 outputs an *html* start tag to the result tree. In line 16, the *<xsl:apply-templates/>* element specifies that the XSLT processor is to apply the *xsl:templates* defined in this XSL document to the current node’s (i.e., the document root’s) children. The content from the applied templates is output in the *html* element that ends at line 17.

Lines 21–84 specify a template that matches element *book*. The template indicates how to format the information contained in *book* elements of *sorting.xml* (Fig. 14.22) as XHTML.

Lines 23–24 create the title for the XHTML document. We use the *book*’s ISBN (from attribute *isbn*) and the contents of element *title* to create the string that appears in the browser window’s title bar (**ISBN 999-99999-9-X - Deitel’s XML Primer**).

Line 28 creates a header element that contains the *book*’s title. Lines 29–31 create a header element that contains the *book*’s author. Because the context node (i.e., the current

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.23: sorting.xsl -->
4 <!-- Transformation of book information into XHTML -->
5 <xsl:stylesheet version = "1.0"
6 xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
7
8 <!-- write XML declaration and DOCTYPE DTD information -->
9 <xsl:output method = "html" omit-xml-declaration = "no"
10 doctype-system = "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"
11 doctype-public = "-//W3C//DTD XHTML 1.1//EN"/>
12
13 <!-- match document root -->
14 <xsl:template match = "/>
15 <html xmlns = "http://www.w3.org/1999/xhtml">
16 <xsl:apply-templates/>
17 </html>
18 </xsl:template>
19
20 <!-- match book -->
21 <xsl:template match = "book">
22 <head>
23 <title>ISBN <xsl:value-of select = "@isbn"/> -
24 <xsl:value-of select = "title"/></title>
25 </head>
26
27 <body>
28 <h1 style = "color: blue"><xsl:value-of select = "title"/></h1>
29 <h2 style = "color: blue">by
30 <xsl:value-of select = "author/lastName"/>,
31 <xsl:value-of select = "author/firstName"/></h2>
32
33 <table style = "border-style: groove; background-color: wheat">
34
35 <xsl:for-each select = "chapters/frontMatter/*">
36 <tr>
37 <td style = "text-align: right">
38 <xsl:value-of select = "name()"/>
39 </td>
40
41 <td>
42 (<xsl:value-of select = "@pages"/> pages)
43 </td>
44 </tr>
45 </xsl:for-each>
46
47 <xsl:for-each select = "chapters/chapter">
48 <xsl:sort select = "@number" data-type = "number"
49 order = "ascending"/>
50 <tr>
51 <td style = "text-align: right">
52 Chapter <xsl:value-of select = "@number"/>
53 </td>
54 </xsl:for-each>
```

Fig. 14.23 | XSL document that transforms `sorting.xml` into XHTML. (Part I of 2.)

```
54 <td>
55 <xsl:value-of select = "text()"/>
56 (<xsl:value-of select = "@pages"/> pages)
57 </td>
58 </tr>
59 </xsl:for-each>
60
61
62 <xsl:for-each select = "chapters/appendix">
63 <xsl:sort select = "@number" data-type = "text"
64 order = "ascending"/>
65 <tr>
66 <td style = "text-align: right">
67 Appendix <xsl:value-of select = "@number"/>
68 </td>
69
70 <td>
71 <xsl:value-of select = "text()"/>
72 (<xsl:value-of select = "@pages"/> pages)
73 </td>
74 </tr>
75 </xsl:for-each>
76 </table>
77
78
<p style = "color: blue">Pages:
79 <xsl:variable name = "pagecount"
80 select = "sum(chapters//*/@pages)"/>
81 <xsl:value-of select = "$pagecount"/>
82
Media Type: <xsl:value-of select = "media/@type"/></p>
83 </body>
84 </xsl:template>
85 </xsl:stylesheet>
```

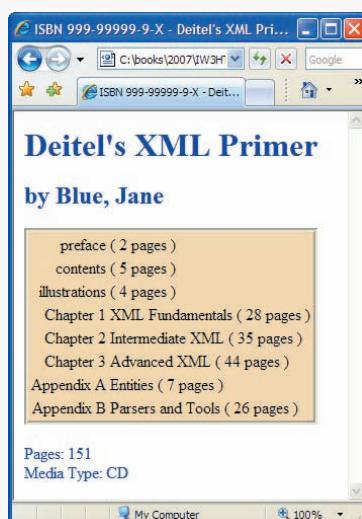


Fig. 14.23 | XSL document that transforms `sorting.xml` into XHTML. (Part 2 of 2.)

node being processed) is book, the XPath expression author/lastName selects the author's last name, and the expression author/firstName selects the author's first name.

Line 35 selects each element (indicated by an asterisk) that is a child of element frontMatter. Line 38 calls **node-set function name** to retrieve the current node's element name (e.g., preface). The current node is the context node specified in the xs1:for-each (line 35). Line 42 retrieves the value of the pages attribute of the current node.

Line 47 selects each chapter element. Lines 48–49 use element **xs1:sort** to sort chapters by number in ascending order. Attribute **select** selects the value of attribute number in context node chapter. Attribute **data-type**, with value "number", specifies a numeric sort, and attribute **order**, with value "ascending", specifies ascending order. Attribute **data-type** also accepts the value "text" (line 63), and attribute **order** also accepts the value "descending". Line 56 uses **node-set function text** to obtain the text between the chapter start and end tags (i.e., the name of the chapter). Line 57 retrieves the value of the pages attribute of the current node. Lines 62–75 perform similar tasks for each appendix.

Lines 79–80 use an **XSL variable** to store the value of the book's total page count and output the page count to the result tree. Attribute **name** specifies the variable's name (i.e., **pagecount**), and attribute **select** assigns a value to the variable. Function **sum** (line 80) totals the values for all page attribute values. The two slashes between **chapters** and **\*** indicate a **recursive descent**—the MSXML processor will search for elements that contain an attribute named **pages** in all descendant nodes of **chapters**. The XPath expression

```
//*
```

selects all the nodes in an XML document. Line 81 retrieves the value of the newly created XSL variable **pagecount** by placing a dollar sign in front of its name.

### *Summary of XSL Style-Sheet Elements*

This section's examples used several predefined XSL elements to perform various operations. Figure 14.24 lists these elements and several other commonly used XSL elements. For more information on these elements and XSL in general, see [www.w3.org/Style/XSL](http://www.w3.org/Style/XSL).

| Element                                       | Description                                                                                                                                                                                              |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <xsl:apply-templates>                         | Applies the templates of the XSL document to the children of the current node.                                                                                                                           |
| <xsl:apply-templates<br>match = "expression"> | Applies the templates of the XSL document to the children of <i>expression</i> . The value of the attribute <b>match</b> (i.e., <i>expression</i> ) must be an XPath expression that specifies elements. |
| <xsl:template>                                | Contains rules to apply when a specified node is matched.                                                                                                                                                |
| <xsl:value-of select =<br>"expression">       | Selects the value of an XML element and adds it to the output tree of the transformation. The required <b>select</b> attribute contains an XPath expression.                                             |

**Fig. 14.24** | XSL style-sheet elements. (Part 1 of 2.)

| Element                                                 | Description                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;xsl:for-each select = "expression"&gt;</code> | Applies a template to every node selected by the XPath specified by the <code>select</code> attribute.                                                                                                                                                                                     |
| <code>&lt;xsl:sort select = "expression"&gt;</code>     | Used as a child element of an <code>&lt;xsl:apply-templates&gt;</code> or <code>&lt;xsl:for-each&gt;</code> element. Sorts the nodes selected by the <code>&lt;xsl:apply-template&gt;</code> or <code>&lt;xsl:for-each&gt;</code> element so that the nodes are processed in sorted order. |
| <code>&lt;xsl:output&gt;</code>                         | Has various attributes to define the format (e.g., XML, XHTML), version (e.g., 1.0, 2.0), document type and media type of the output document. This tag is a top-level element—it can be used only as a child element of an <code>xsl:stylesheet</code> .                                  |
| <code>&lt;xsl:copy&gt;</code>                           | Adds the current node to the output tree.                                                                                                                                                                                                                                                  |

**Fig. 14.24** | XSL style-sheet elements. (Part 2 of 2.)

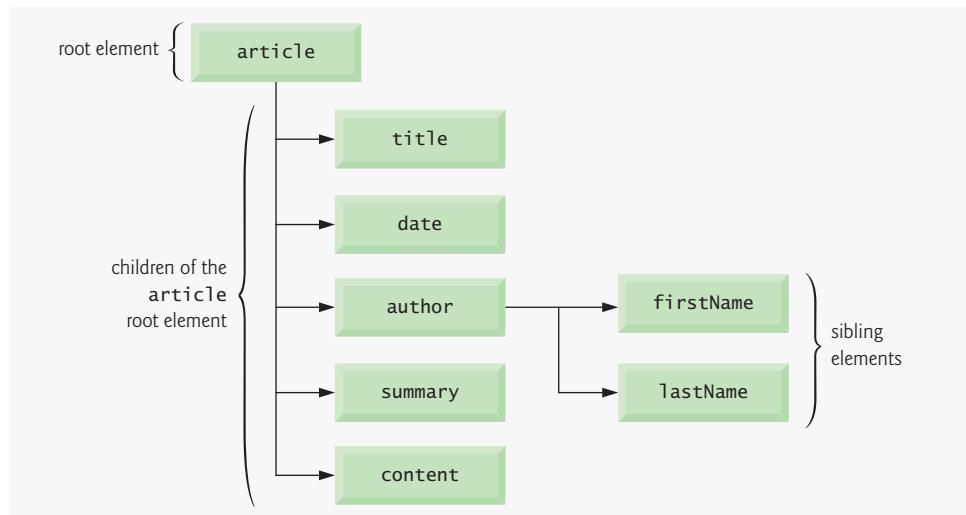
This section introduced Extensible Stylesheet Language (XSL) and showed how to create XSL transformations to convert XML documents from one format to another. We showed how to transform XML documents to XHTML documents for display in a web browser. Recall that these transformations are performed by MSXML, Internet Explorer's built-in XML parser and XSLT processor. In most business applications, XML documents are transferred between business partners and are transformed to other XML vocabularies programmatically. Section 14.9 discusses the XML Document Object Model (DOM) and demonstrates how to manipulate the DOM of an XML document using JavaScript.

## 14.9 Document Object Model (DOM)

Although an XML document is a text file, retrieving data from the document using traditional sequential file processing techniques is neither practical nor efficient, especially for adding and removing elements dynamically.

Upon successfully parsing a document, some XML parsers store document data as tree structures in memory. Figure 14.25 illustrates the tree structure for the root element of the document `article.xml` (Fig. 14.2). This hierarchical tree structure is called a **Document Object Model (DOM) tree**, and an XML parser that creates this type of structure is known as a **DOM parser**. Each element name (e.g., `article`, `date`, `firstName`) is represented by a node. A node that contains other nodes (called **child nodes** or **children**) is called a **parent node** (e.g., `author`). A parent node can have many children, but a child node can have only one parent node. Nodes that are peers (e.g., `firstName` and `lastName`) are called **sibling nodes**. A node's **descendant nodes** include its children, its children's children and so on. A node's **ancestor nodes** include its parent, its parent's parent and so on. Many of the XML DOM capabilities you'll see in this section are similar or identical to those of the XHTML DOM you learned in Chapter 12.

The DOM tree has a single **root node**, which contains all the other nodes in the document. For example, the root node of the DOM tree that represents `article.xml` contains a node for the XML declaration (line 1), two nodes for the comments (lines 3–4) and a node for the XML document's root element `article` (line 5).



**Fig. 14.25** | Tree structure for the document `article.xml` of Fig. 14.2.

To introduce document manipulation with the XML Document Object Model, we provide a scripting example (Fig. 14.26) that uses JavaScript and XML. This example loads the XML document `article.xml` (Fig. 14.2) and uses the XML DOM API to display the document's element names and values. The example also provides buttons that enable you to navigate the DOM structure. As you click each button, an appropriate part of the document is highlighted. All of this is done in a manner that enables the example to execute in both Internet Explorer 7 and Firefox 2. Figure 14.26 lists the JavaScript code that manipulates this XML document and displays its content in an XHTML page.

### *Overview of the `body` Element*

Lines 203–217 create the XHTML document's `body`. When the `body` loads, its `onload` event calls our JavaScript function `loadXMLDocument` to load and display the contents of `article.xml` in the `div` at line 216 (`outputDiv`). Lines 204–215 define a form consisting of five buttons. When each button is pressed, it invokes one of our JavaScript functions to navigate `article.xml`'s DOM structure.

### *Global Script Variables*

Lines 16–21 in the `script` element (lines 14–201) declare several variables used throughout the script. Variable `doc` references a DOM object representation of `article.xml`. Variable `outputHTML` stores the markup that will be placed in `outputDiv`. Variable `idCounter` is used to track the unique `id` attributes that we assign to each element in the `outputHTML` markup. These `ids` will be used to dynamically highlight parts of the document when the user clicks the buttons in the form. Variable `depth` determines the indentation level for the content in `article.xml`. We use this to structure the output using the nesting of the elements in `article.xml`. Variables `current` and `previous` track the current and previous nodes in `article.xml`'s DOM structure as the user navigates it.

```
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 14.26: XMLDOMTraversal.html -->
6 <!-- Traversing an XML document using the XML DOM. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Traversing an XML document using the XML DOM</title>
10 <style type = "text/css">
11 .highlighted { background-color: yellow }
12 #outputDiv { font: 10pt "Lucida Console", monospace; }
13 </style>
14 <script type="text/javascript">
15 <!--
16 var doc; // variable to reference the XML document
17 var outputHTML = ""; // stores text to output in outputDiv
18 var idCounter = 1; // used to create div IDs
19 var depth = -1; // tree depth is -1 to start
20 var current = null; // represents the current node for traversals
21 var previous = null; // represent prior node in traversals
22
23 // load XML document based on whether the browser is IE7 or Firefox 2
24 function LoadXMLDocument(url)
25 {
26 if (window.ActiveXObject) // IE7
27 {
28 // create IE7-specific XML document object
29 doc = new ActiveXObject("Msxml2.DOMDocument.6.0");
30 doc.async = false; // specifies synchronous loading of XML doc
31 doc.load(url); // load the XML document specified by url
32 buildHTML(doc.childNodes); // display the nodes
33 displayDoc();
34 } // end if
35 else if (document.implementation &&
36 document.implementation.createDocument) // other browsers
37 {
38 // create XML document object
39 doc = document.implementation.createDocument("", "", null);
40 doc.load(url); // load the XML document specified by url
41 doc.onload = function() // function to execute when doc loads
42 {
43 buildHTML(doc.childNodes); // called by XML doc onload event
44 displayDoc(); // display the HTML
45 } // end XML document's onload event handler
46 } // end else
47 else // not supported
48 alert('This script is not supported by your browser');
49 } // end function LoadXMLDocument
50
51 // traverse xmlDocument and build XHTML representation of its content
52 function buildHTML(childList)
53 {
```

**Fig. 14.26** | Traversing an XML document using the XML DOM. (Part I of 8.)

```

54 ++depth; // increase tab depth
55
56 // display each node's content
57 for (var i = 0; i < childList.length; i++)
58 {
59 switch (childList[i].nodeType)
60 {
61 case 1: // Node.ELEMENT_NODE; value used for portability
62 outputHTML += "<div id=\"id" + idCounter + "\">>";
63 spaceOutput(depth); // insert spaces
64 outputHTML += childList[i].nodeName; // show node's name
65 ++idCounter; // increment the id counter
66
67 // if current node has children, call buildHTML recursively
68 if (childList[i].childNodes.length != 0)
69 buildHTML(childList[i].childNodes);
70
71 outputHTML += "</div>";
72 break;
73 case 3: // Node.TEXT_NODE; value used for portability
74 case 8: // Node.COMMENT_NODE; value used for portability
75 // if nodeValue is not 3 or 6 spaces (Firefox issue),
76 // include nodeValue in HTML
77 if (childList[i].nodeValue.indexOf(" ") == -1 &&
78 childList[i].nodeValue.indexOf(" ") == -1)
79 {
80 outputHTML += "<div id=\"id" + idCounter + "\">>";
81 spaceOutput(depth); // insert spaces
82 outputHTML += childList[i].nodeValue + "</div>";
83 ++idCounter; // increment the id counter
84 } // end if
85 } // end switch
86 } // end for
87
88 --depth; // decrease tab depth
89 } // end function buildHTML
90
91 // display the XML document and highlight the first child
92 function displayDoc()
93 {
94 document.getElementById("outputDiv").innerHTML = outputHTML;
95 current = document.getElementById('id1');
96 setCurrentNodeStyle(current.id, true);
97 } // end function displayDoc
98
99 // insert non-breaking spaces for indentation
100 function spaceOutput(number)
101 {
102 for (var i = 0; i < number; i++)
103 {
104 outputHTML += " ";
105 } // end for
106 } // end function spaceOutput

```

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 2 of 8.)

```

107
108 // highlight first child of current node
109 function processFirstChild()
110 {
111 if (current.childNodes.length == 1 && // only one child
112 current.firstChild.nodeType == 3) // and it's a text node
113 {
114 alert("There is no child node");
115 } // end if
116 else if (current.childNodes.length > 1)
117 {
118 previous = current; // save currently highlighted node
119
120 if (current.firstChild.nodeType != 3) // if not text node
121 current = current.firstChild; // get new current node
122 else // if text node, use firstChild's nextSibling instead
123 current = current.firstChild.nextSibling; // get first sibling
124
125 setCurrentNodeStyle(previous.id, false); // remove highlight
126 setCurrentNodeStyle(current.id, true); // add highlight
127 } // end if
128 else
129 alert("There is no child node");
130 } // end function processFirstChild
131
132 // highlight next sibling of current node
133 function processNextSibling()
134 {
135 if (current.id != "outputDiv" && current.nextSibling)
136 {
137 previous = current; // save currently highlighted node
138 current = current.nextSibling; // get new current node
139 setCurrentNodeStyle(previous.id, false); // remove highlight
140 setCurrentNodeStyle(current.id, true); // add highlight
141 } // end if
142 else
143 alert("There is no next sibling");
144 } // end function processNextSibling
145
146 // highlight previous sibling of current node if it is not a text node
147 function processPreviousSibling()
148 {
149 if (current.id != "outputDiv" && current.previousSibling &&
150 current.previousSibling.nodeType != 3)
151 {
152 previous = current; // save currently highlighted node
153 current = current.previousSibling; // get new current node
154 setCurrentNodeStyle(previous.id, false); // remove highlight
155 setCurrentNodeStyle(current.id, true); // add highlight
156 } // end if
157 else
158 alert("There is no previous sibling");
159 } // end function processPreviousSibling

```

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 3 of 8.)

```

160
161 // highlight last child of current node
162 function processLastChild()
163 {
164 if (current.childNodes.length == 1 &&
165 current.lastChild.nodeType == 3)
166 {
167 alert("There is no child node");
168 } // end if
169 else if (current.childNodes.length != 0)
170 {
171 previous = current; // save currently highlighted node
172 current = current.lastChild; // get new current node
173 setCurrentNodeStyle(previous.id, false); // remove highlight
174 setCurrentNodeStyle(current.id, true); // add highlight
175 } // end if
176 else
177 alert("There is no child node");
178 } // end function processLastChild
179
180 // highlight parent of current node
181 function processParentNode()
182 {
183 if (current.parentNode.id != "body")
184 {
185 previous = current; // save currently highlighted node
186 current = current.parentNode; // get new current node
187 setCurrentNodeStyle(previous.id, false); // remove highlight
188 setCurrentNodeStyle(current.id, true); // add highlight
189 } // end if
190 else
191 alert("There is no parent node");
192 } // end function processParentNode
193
194 // set style of node with specified id
195 function setCurrentNodeStyle(id, highlight)
196 {
197 document.getElementById(id).className =
198 (highlight ? "highlighted" : "");
199 } // end function setCurrentNodeStyle
200 // -->
201 </script>
202 </head>
203 <body id = "body" onload = "loadXMLDocument('article.xml');">
204 <form action = "" onsubmit = "return false;">
205 <input type = "submit" value = "firstChild"
206 onclick = "processFirstChild()"/>
207 <input type = "submit" value = "nextSibling"
208 onclick = "processNextSibling()"/>
209 <input type = "submit" value = "previousSibling"
210 onclick = "processPreviousSibling()"/>
211 <input type = "submit" value = "lastChild"
212 onclick = "processLastChild()"/>

```

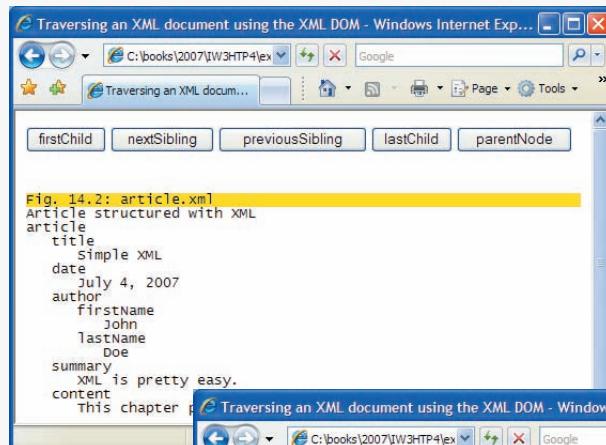
Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 4 of 8.)

```

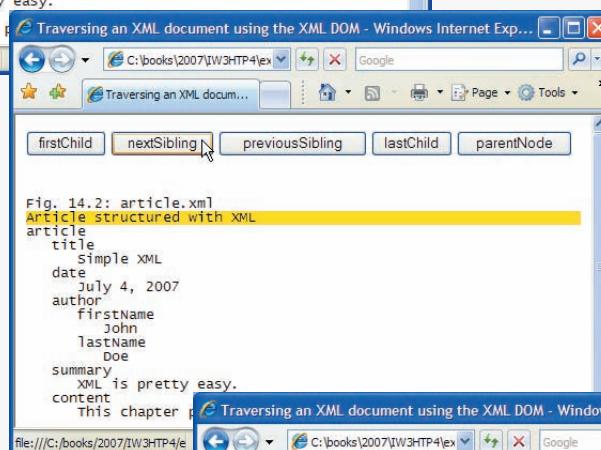
213 <input type = "submit" value = "parentNode"
214 onclick = "process.parentNode()"/>
215 </form>

216 <div id = "outputDiv"></div>
217 </body>
218 </html>

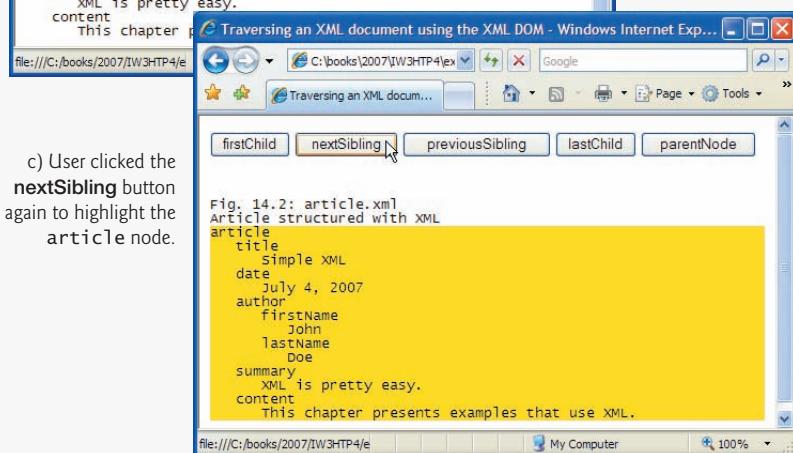
```



a) The comment node at the beginning of `article.xml` is highlighted when the XML document first loads.

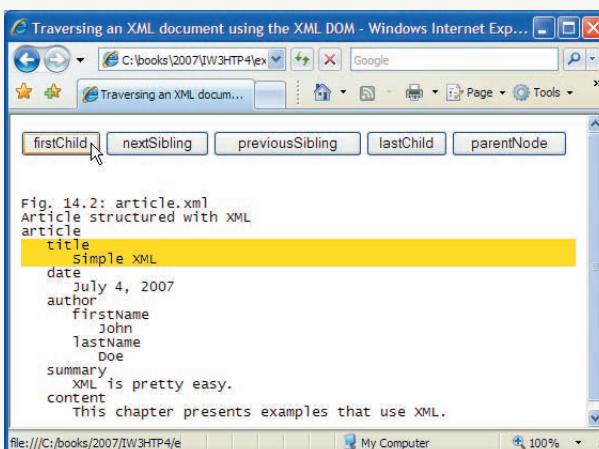


b) User clicked the `nextSibling` button to highlight the second comment node.

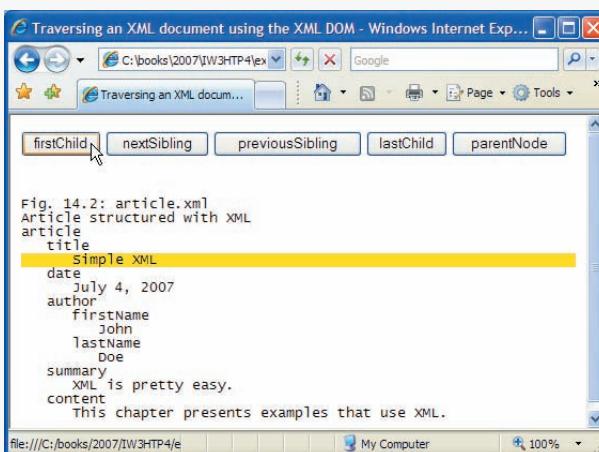


c) User clicked the `nextSibling` button again to highlight the `article` node.

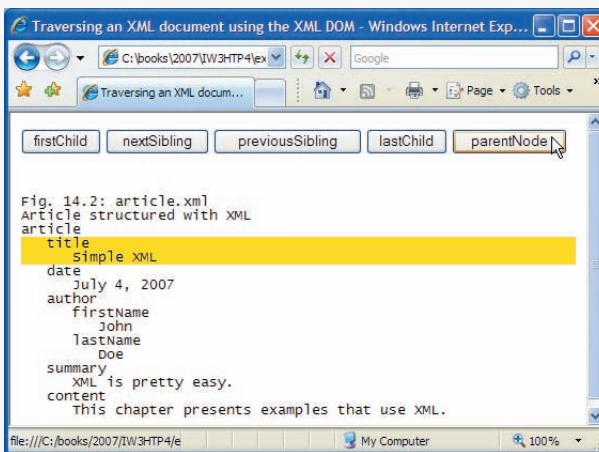
**Fig. 14.26** | Traversing an XML document using the XML DOM. (Part 5 of 8.)



- d) User clicked the **firstChild** button to highlight the **article** node's **title** child node.

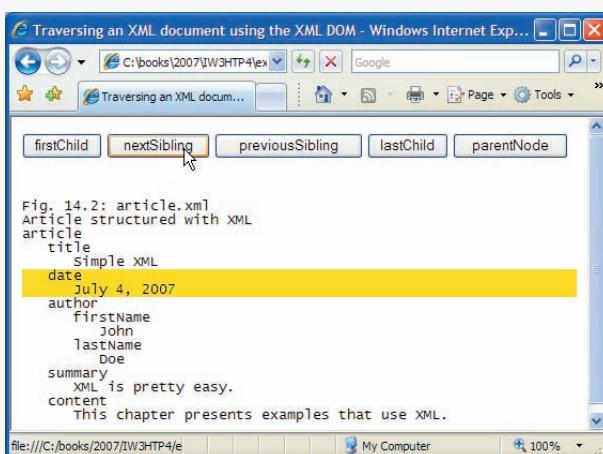


- e) User clicked the **firstChild** button again to highlight the **title** node's text child node.

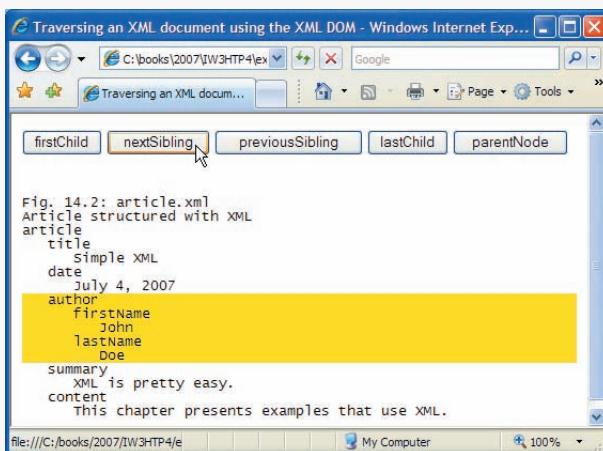


- f) User clicked the **parentNode** button to highlight the text node's parent **title** node.

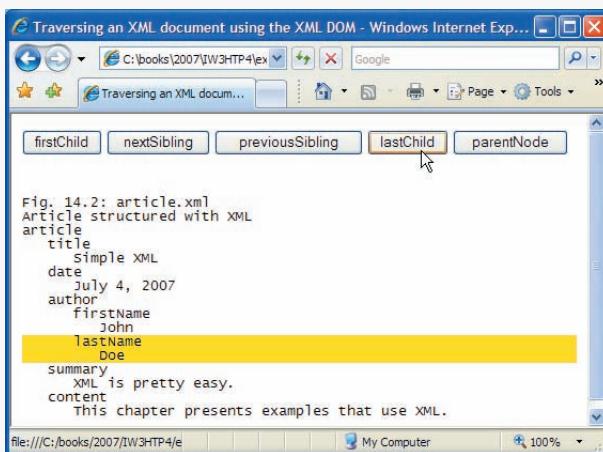
**Fig. 14.26** | Traversing an XML document using the XML DOM. (Part 6 of 8.)



g) User clicked the **nextSibling** button to highlight the **title** node's **date** sibling node.

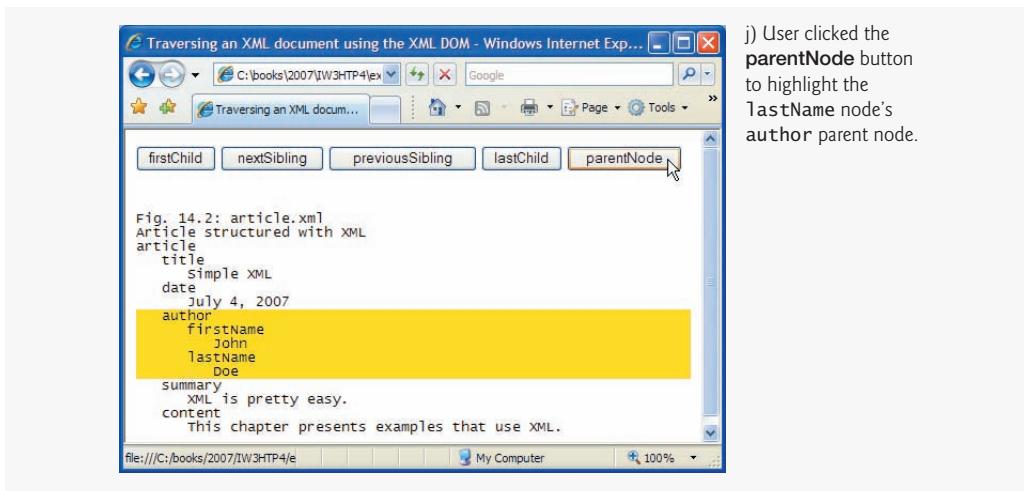


h) User clicked the **nextSibling** button to highlight the **date** node's **author** sibling node.



i) User clicked the **lastChild** button to highlight the **author** node's last child node (**lastName**).

**Fig. 14.26** | Traversing an XML document using the XML DOM. (Part 7 of 8.)



**Fig. 14.26** | Traversing an XML document using the XML DOM. (Part 8 of 8.)

### Function `loadXMLDocument`

Function `loadXMLDocument` (lines 24–49) receives the URL of an XML document to load, then loads the document based on whether the browser is Internet Explorer 7 (26–34) or Firefox 2 (lines 35–46)—the code for Firefox 2 works in several other browsers as well. Line 26 determines whether `window.ActiveXObject` exists. If so, this indicates that the browser is Internet Explorer. Line 29 creates a Microsoft ActiveXObject that loads Microsoft’s **MSXML parser**, which provides capabilities for manipulating XML documents. Line 30 indicates that we’d like the XML document to be loaded synchronously, then line 31 uses the ActiveXObject’s **Load method** to load `article.xml`. When this completes, we call our `buildHTML` method (defined in lines 52–89) to construct an XHTML representation of the XML document. The expression `doc.childNodes` is a list of the XML document’s top-level nodes. Line 33 calls our `displayDoc` function (lines 92–97) to display the contents of `article.xml` in `outputDiv`.

If the browser is Firefox 2, then the `document` object’s **implementation** property and the `implementation` property’s **createDocument** method will exist (lines 35–36). In this case, line 39 uses the `createDocument` method to create an empty XML document object. If necessary, you can specify the XML document’s namespace as the first argument and its root element as the second argument. We used empty strings for both in this example. According to the site [www.w3schools.com/xml/xml\\_parser.asp](http://www.w3schools.com/xml/xml_parser.asp), the third argument is not implemented yet, so it should always be `null`. Line 40 calls its `load` method to load `article.xml`. Firefox loads the XML document asynchronously, so you must use the XML document’s `onload` property to specify a function to call (an anonymous function in this example) when the document finishes loading. When this event occurs, lines 43–44 call `buildHTML` and `displayDoc` just as we did in lines 32–33.



### Common Programming Error 14.11

Attempting to process the contents of a dynamically loaded XML document in Firefox before the document’s `onload` event fires is a logic error. The document’s contents are not available until the `onload` event fires.

### **Function buildHTML**

Function `buildHTML` (lines 52–89) is a recursive function that receives a list of nodes as an argument. Line 54 increments the depth for indentation purposes. Lines 57–86 iterate through the nodes in the list. The `switch` statement (lines 59–85) uses the current node's `nodeType` property to determine whether the current node is an element (line 61), a text node (i.e., the text content of an element; line 73) or a comment node (line 74). If it is an element, then we begin a new `div` element in our XHTML (line 62) and give it a unique `id`. Then function `spaceOutput` (defined in lines 100–106) appends `nonbreaking spaces (&nbsp;)`—i.e., spaces that the browser is not allowed to collapse or that can be used to keep words together—to indent the current element to the correct level. Line 64 appends the name of the current element using the node's `nodeName` property. If the current element has children, the length of the current node's `childNodes` list is nonzero and line 69 recursively calls `buildHTML` to append the current element's child nodes to the markup. When that recursive call completes, line 71 completes the `div` element that we started at line 62.

If the current element is a text node, lines 77–78 obtain the node's value with the `nodeValue` property and use the string method `index0f` to determine whether the node's value starts with three or six spaces. Unfortunately, unlike MSMXL, Firefox's XML parser does not ignore the white space used for indentation in XML documents. Instead it creates text nodes containing just the space characters. The condition in lines 77–78 enables us to ignore these nodes in Firefox. If the node contains text, lines 80–82 append a new `div` to the markup and use the node's `nodeValue` property to insert that text in the `div`. Line 88 in `buildHTML` decrements the depth counter.



### **Portability Tip 14.4**

*FIREFOX'S XML PARSER DOES NOT IGNORE WHITE SPACE USED FOR INDENTATION IN XML DOCUMENTS. INSTEAD, IT CREATES TEXT NODES CONTAINING THE WHITE-SPACE CHARACTERS.*

### **Function displayDoc**

In function `displayDoc` (lines 92–97), line 94 uses the DOM's `getElementById` method to obtain the `outputDiv` element and set its `innerHTML` property to the new markup generated by `buildHTML`. Then, line 95 sets variable `current` to refer to the `div` with `id 'id1'` in the new markup, and line 96 uses our `setCurrentNodeStyle` method (defined at lines 195–199) to highlight that `div`.

### **Functions processFirstChild and processLastChild**

Function `processFirstChild` (lines 109–130) is invoked by the `onClick` event of the button at lines 205–206. If the current node has only one child and it's a text node (lines 111–112), line 114 displays an alert dialog indicating that there is no child node—we navigate only to nested XML elements in this example. If there are two or more children, line 118 stores the value of `current` in `previous`, and lines 120–123 set `current` to refer to its `firstChild` (if this child is not a text node) or its `firstChild`'s `nextSibling` (if the `firstChild` is a text node)—again, this is to ensure that we navigate only to nodes that represent XML elements. Then lines 125–126 unhighlight the previous node and highlight the new current node. Function `processLastChild` (lines 162–178) works similarly, using the current node's `lastChild` property.

### ***Functions processNextSibling and processPreviousSibling***

Function `processNextSibling` (lines 133–144) first ensures that the current node is not the `outputDiv` and that `nextSibling` exists. If so, lines 137–140 adjust the previous and current nodes accordingly and update their highlighting. Function `processPreviousSibling` (lines 147–159) works similarly, ensuring first that the current node is not the `outputDiv`, that `previousSibling` exists and that `previousSibling` is not a text node.

### ***Function processParentNode***

Function `processParentNode` (lines 181–192) first checks whether the current node's `parentNode` is the XHTML page's body. If not, lines 185–188 adjust the previous and current nodes accordingly and update their highlighting.

### ***Common DOM Properties***

The tables in Figs. 14.27–14.32 describe many common DOM properties and methods. Some of the key DOM objects are `Node` (a node in the tree), `NodeList` (an ordered set of `Nodes`), `Document` (the document), `Element` (an element node), `Attr` (an attribute node) and `Text` (a text node). There are many more objects, properties and methods than we can possibly list here. Our XML Resource Center ([www.deitel.com/XML/](http://www.deitel.com/XML/)) includes links to various DOM reference websites.

| Property/Method              | Description                                                                                                                                                                                                                                   |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>nodeType</code>        | An integer representing the node type.                                                                                                                                                                                                        |
| <code>nodeName</code>        | The name of the node.                                                                                                                                                                                                                         |
| <code>nodeValue</code>       | A string or null depending on the node type.                                                                                                                                                                                                  |
| <code>parentNode</code>      | The parent node.                                                                                                                                                                                                                              |
| <code>childNodes</code>      | A <code>NodeList</code> (Fig. 14.28) with all the children of the node.                                                                                                                                                                       |
| <code>firstChild</code>      | The first child in the Node's <code>NodeList</code> .                                                                                                                                                                                         |
| <code>lastChild</code>       | The last child in the Node's <code>NodeList</code> .                                                                                                                                                                                          |
| <code>previousSibling</code> | The node preceding this node; <code>null</code> if there is no such node.                                                                                                                                                                     |
| <code>nextSibling</code>     | The node following this node; <code>null</code> if there is no such node.                                                                                                                                                                     |
| <code>attributes</code>      | A collection of <code>Attr</code> objects (Fig. 14.31) containing the attributes for this node.                                                                                                                                               |
| <code>insertBefore</code>    | Inserts the node (passed as the first argument) before the existing node (passed as the second argument). If the new node is already in the tree, it is removed before insertion. The same behavior is true for other methods that add nodes. |
| <code>replaceChild</code>    | Replaces the second argument node with the first argument node.                                                                                                                                                                               |
| <code>removeChild</code>     | Removes the child node passed to it.                                                                                                                                                                                                          |
| <code>appendChild</code>     | Appends the node it receives to the list of child nodes.                                                                                                                                                                                      |

**Fig. 14.27** | Common Node properties and methods.

| Property/Method     | Description                                                                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>item</code>   | Method that receives an index number and returns the element node at that index. Indices range from 0 to <code>length</code> – 1. You can also access the nodes in a <code>NodeList</code> via array indexing. |
| <code>length</code> | The total number of nodes in the list.                                                                                                                                                                         |

**Fig. 14.28** | `NodeList` property and method.

| Property/Method                | Description                                                                                                                                                                                                                                                                 |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>documentElement</code>   | The root node of the document.                                                                                                                                                                                                                                              |
| <code>createElement</code>     | Creates and returns an element node with the specified tag name.                                                                                                                                                                                                            |
| <code>createAttribute</code>   | Creates and returns an <code>Attr</code> node (Fig. 14.31) with the specified name and value.                                                                                                                                                                               |
| <code>createTextNode</code>    | Creates and returns a text node that contains the specified text.                                                                                                                                                                                                           |
| <code>getElementsByName</code> | Returns a <code>NodeList</code> of all the nodes in the subtree with the name specified as the first argument, ordered as they would be encountered in a preorder traversal. An optional second argument specifies either the direct child nodes (0) or any descendant (1). |

**Fig. 14.29** | Document properties and methods.

| Property/Method               | Description                                                                                                 |
|-------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>tagName</code>          | The name of the element.                                                                                    |
| <code>getAttribute</code>     | Returns the value of the specified attribute.                                                               |
| <code>setAttribute</code>     | Changes the value of the attribute passed as the first argument to the value passed as the second argument. |
| <code>removeAttribute</code>  | Removes the specified attribute.                                                                            |
| <code>getAttributeNode</code> | Returns the specified attribute node.                                                                       |
| <code>setAttributeNode</code> | Adds a new attribute node with the specified name.                                                          |

**Fig. 14.30** | Element property and methods.

| Property           | Description                      |
|--------------------|----------------------------------|
| <code>value</code> | The specified attribute's value. |
| <code>name</code>  | The name of the attribute.       |

**Fig. 14.31** | `Attr` properties.

| Property | Description                                     |
|----------|-------------------------------------------------|
| data     | The text contained in the node.                 |
| length   | The number of characters contained in the node. |

**Fig. 14.32** | Text methods.

### Locating Data in XML Documents with XPath

Although you can use XML DOM capabilities to navigate through and manipulate nodes, this is not the most efficient means of locating data in an XML document's DOM tree. A simpler way to locate nodes is to search for lists of nodes matching search criteria that are written as XPath expressions. Recall that XPath (XML Path Language) provides a syntax for locating specific nodes in XML documents effectively and efficiently. XPath is a string-based language of expressions used by XML and many of its related technologies (such as XSLT, discussed in Section 14.8).

Figure 14.33 enables the user to enter XPath expressions in an XHTML form. When the user clicks the **Get Matches** button, the script applies the XPath expression to the XML DOM and displays the matching nodes. Figure 14.34 shows the XML document `sports.xml` that we use in this example. [Note: The versions of `sports.xml` presented in Fig. 14.34 and Fig. 14.20 are nearly identical. In the current example, we do not want to apply an XSLT, so we omit the processing instruction found in line 2 of Fig. 14.20. We also removed extra blank lines to save space.]

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 14.33: xpath.html -->
6 <!-- Using XPath to locate nodes in an XML document. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Using XPath to Locate Nodes in an XML Document</title>
10 <style type = "text/css">
11 #outputDiv { font: 10pt "Lucida Console", monospace; }
12 </style>
13 <script type = "text/javascript">
14 <!--
15 var doc; // variable to reference the XML document
16 var outputHTML = ""; // stores text to output in outputDiv
17 var browser = ""; // used to determine which browser is being used
18
19 // load XML document based on whether the browser is IE7 or Firefox 2
20 function LoadXMLDocument(url)
21 {
22 if (window.ActiveXObject) // IE
23 {

```

**Fig. 14.33** | Using XPath to locate nodes in an XML document. (Part 1 of 3.)

```

24 // create IE7-specific XML document object
25 doc = new ActiveXObject("Msxml2.DOMDocument.6.0");
26 doc.async = false; // specifies synchronous loading of XML doc
27 doc.load(url); // load the XML document specified by url
28 browser = "IE7"; // set browser
29 } // end if
30 else if (document.implementation &&
31 document.implementation.createDocument) // other browsers
32 {
33 // create XML document object
34 doc = document.implementation.createDocument("", "", null);
35 doc.load(url); // load the XML document specified by url
36 browser = "FF2"; // set browser
37 } // end else
38 else // not supported
39 alert('This script is not supported by your browser');
40 } // end function loadXMLDocument
41
42 // display the XML document
43 function displayDoc()
44 {
45 document.getElementById("outputDiv").innerHTML = outputHTML;
46 } // end function displayDoc
47
48 // obtain and apply XPath expression
49 function processXPathExpression()
50 {
51 var xpathExpression = document.getElementById("inputField").value;
52 outputHTML = "";
53
54 if (browser == "IE7")
55 {
56 var result = doc.selectNodes(xpathExpression);
57
58 for (var i = 0; i < result.length; i++)
59 outputHTML += "<div style='clear: both'>" +
60 result.item(i).text + "</div>";
61 } // end if
62 else // browser == "FF2"
63 {
64 var result = document.evaluate(xpathExpression, doc, null,
65 XPathResult.ANY_TYPE, null);
66 var current = result.iterateNext();
67
68 while (current)
69 {
70 outputHTML += "<div style='clear: both'>" +
71 current.textContent + "</div>";
72 current = result.iterateNext();
73 } // end while
74 } // end else
75 }

```

Fig. 14.33 | Using XPath to locate nodes in an XML document. (Part 2 of 3.)

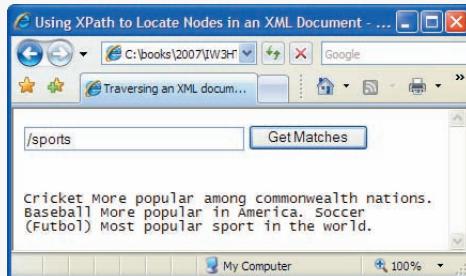
```

76 displayDoc();
77 } // end function processXPathExpression
78 // -->
79 </script>
80 </head>
81 <body id = "body" onLoad = "loadXMLDocument('sports.xml');">
82 <form action = "" onSubmit = "return false;">
83 <input id = "inputField" type = "text" style = "width: 200px;"/>
84 <input type = "submit" value = "Get Matches"
85 onClick = "processXPathExpression()"/>
86 </form>

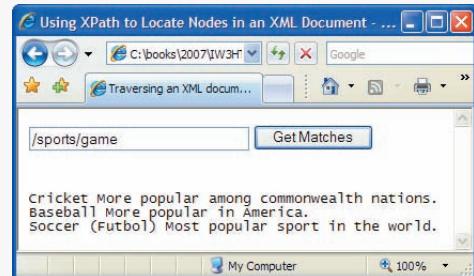
87 <div id = "outputDiv"></div>
88 </body>
89 </html>

```

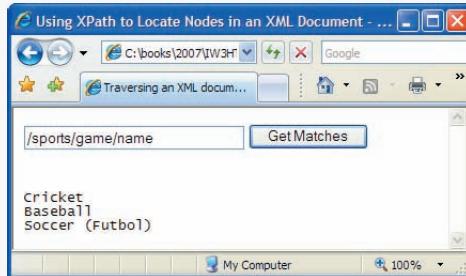
a)



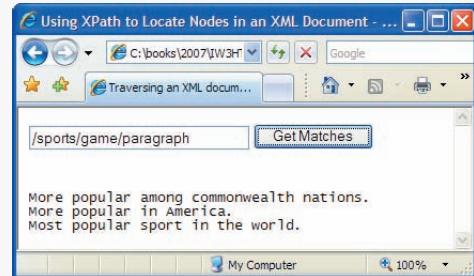
b)



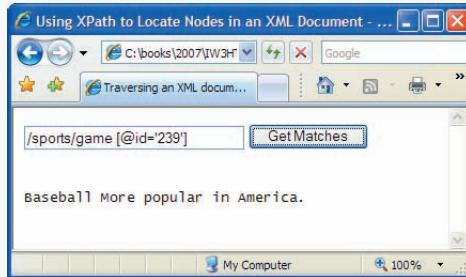
c)



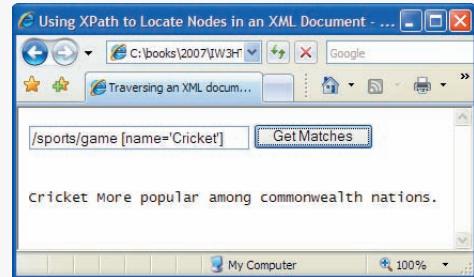
d)



e)



f)



**Fig. 14.33** | Using XPath to locate nodes in an XML document. (Part 3 of 3.)

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 14.34: sports.xml -->
4 <!-- Sports Database -->
5 <sports>
6 <game id = "783">
7 <name>Cricket</name>
8 <paragraph>
9 More popular among commonwealth nations.
10 </paragraph>
11 </game>
12 <game id = "239">
13 <name>Baseball</name>
14 <paragraph>
15 More popular in America.
16 </paragraph>
17 </game>
18 <game id = "418">
19 <name>Soccer (Futbol)</name>
20 <paragraph>
21 Most popular sport in the world.
22 </paragraph>
23 </game>
24 </sports>

```

**Fig. 14.34** | XML document that describes various sports.

The program of Fig. 14.33 loads the XML document `sports.xml` (Fig. 14.34) using the same techniques we presented in Fig. 14.26, so we focus on only the new features in this example. Internet Explorer 7 (MSXML) and Firefox 2 handle XPath processing differently, so this example declares the variable `browser` (line 17) to store the browser that loaded the page. In function `loadDocument` (lines 20–40), lines 28 and 36 assign a string to variable `browser` indicating the appropriate browser.

When the body of this XHTML document loads, its `onLoad` event calls `loadDocument` (line 81) to load the `sports.xml` file. The user specifies the XPath expression in the `input` element at line 83. When the user clicks the **Get Matches** button (lines 84–85), its `onClick` event handler invokes our `processXPathExpression` function to locate any matches and display the results in `outputDiv` (line 87).

Function `processXPathExpression` (lines 49–77) first obtains the XPath expression (line 51). The document object's `getElementsById` method returns the element with the `id` "inputField"; then we use its `value` property to get the XPath expression. Lines 54–61 apply the XPath expression in Internet Explorer 7, and lines 62–74 apply the XPath expression in Firefox 2. In IE7, the XML document object's `selectNodes` method receives an XPath expression as an argument and returns a collection of elements that match the expression. Lines 58–60 iterate through the results and mark up each one in a separate `div` element. After this loop completes, line 76 displays the generated markup in `outputDiv`.

For Firefox 2, lines 64–65 invoke the XML document object's `evaluate` method, which receives five arguments—the XPath expression, the document to apply the expression to, a namespace resolver, a result type and an `XPathResult` object into which to place the results. If the last argument is `null`, the function simply returns a new `XPathResult`

`object` containing the matches. The namespace resolver argument can be null if you are not using XML namespace prefixes in the XPath processing. Lines 66–73 iterate through the `XPathResult` and mark up the results. Line 66 invokes the `XPathResult`'s `iterateNext` method to position to the first result. If there is a result, the condition in line 68 will be true, and lines 70–71 create a `div` for that result. Line 72 then positions to the next result. After this loop completes, line 76 displays the generated markup in `outputDiv`.

Figure 14.35 summarizes the XPath expressions that we demonstrate in Fig. 14.33's sample outputs. For more information on using XPath in Firefox, visit the site [developer.mozilla.org/en/docs/XPath](https://developer.mozilla.org/en/docs/XPath). For more information on using XPath in Internet Explorer, visit [msdn.microsoft.com/msdnmag/issues/0900/xml/](https://msdn.microsoft.com/msdnmag/issues/0900/xml/).

| Expression                                 | Description                                                                                                                                                                                            |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>/sports</code>                       | Matches all <code>sports</code> nodes that are child nodes of the document root node.                                                                                                                  |
| <code>/sports/game</code>                  | Matches all <code>game</code> nodes that are child nodes of <code>sports</code> , which is a child of the document root.                                                                               |
| <code>/sports/game/name</code>             | Matches all <code>name</code> nodes that are child nodes of <code>game</code> . The <code>game</code> is a child of <code>sports</code> , which is a child of the document root.                       |
| <code>/sports/game/paragraph</code>        | Matches all <code>paragraph</code> nodes that are child nodes of <code>game</code> . The <code>game</code> is a child of <code>sports</code> , which is a child of the document root.                  |
| <code>/sports/game [@id='239']</code>      | Matches the <code>game</code> node with the <code>id</code> number 239. The <code>game</code> is a child of <code>sports</code> , which is a child of the document root.                               |
| <code>/sports/game [name='Cricket']</code> | Matches all <code>game</code> nodes that contain a child element whose name is <code>Cricket</code> . The <code>game</code> is a child of <code>sports</code> , which is a child of the document root. |

**Fig. 14.35** | XPath expressions and descriptions.

## 14.10 RSS

RSS stands for **RDF (Resource Description Framework) Site Summary** and is also known as **Rich Site Summary** and **Really Simple Syndication**. RSS is an XML format used to syndicate website content, such as news articles, blog entries, product reviews, podcasts, vodcasts and more for inclusion on other websites. An RSS feed contains an **rss root element** with a `version` attribute and a **channel child element** with **item subelements**. Depending on the RSS version, the `channel` and `item` elements have certain required and optional child elements. The `item` elements provide the feed subscriber with a link to a web page or file, a title and description of the page or file. The most commonly used RSS feed versions are 0.91, 1.0, and 2.0, with RSS 2.0 being the most popular version. We discuss only RSS version 2.0 in this section.

RSS version 2.0, introduced in 2002, builds upon the RSS 0.9x versions. Version 2.0 does not contain length limitations or `item` element limitations of earlier versions, makes some formerly required elements optional, and adds new `channel` and `item` subelements. Removing length limitations on `item` descriptions allows RSS feeds to contain entire articles, blog entries and other web content. You can also have partial feeds that provide only a summary of the syndicated content. Partial feeds require the RSS subscriber to visit a website to view the complete content. RSS 2.0 allows `item` elements to contain an `enclosure` element providing the location of a media file that is related to the `item`. Such enclosures enable syndication of audio and video (such as podcasts and vodcasts) via RSS feeds.

By providing up-to-date, linkable content for anyone to use, RSS enables website developers to draw more traffic. It also allows users to get news and information from many sources easily and reduces content development time. RSS simplifies importing information from portals, weblogs and news sites. Any piece of information can be syndicated via RSS, not just news. After putting information in RSS format, an RSS program, such as a feed reader or aggregator, can check the feed for changes and react to them. For more details on RSS and for links to many RSS sites, visit our RSS Resource Center at [www.deitel.com/RSS](http://www.deitel.com/RSS).

### **RSS 2.0 `channel` and `item` Elements**

In RSS 2.0, the required child elements of `channel` are `description`, `link` and `title`, and the required child element of an `item` is either `title` or `description`. Figures 14.36–14.37 overview the child elements of `channels` and `items`, respectively.

| Element                     | Description                                                                                                                             |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <code>title</code>          | The name of the <code>channel</code> or feed.                                                                                           |
| <code>link</code>           | The URL to the website of the <code>channel</code> or feed the RSS is coming from.                                                      |
| <code>description</code>    | A description of the <code>channel</code> or feed.                                                                                      |
| <code>language</code>       | The language the <code>channel</code> is in, using W3C language values.                                                                 |
| <code>copyright</code>      | The copyright material of the <code>channel</code> or feed.                                                                             |
| <code>managingEditor</code> | The e-mail address of the editor of the <code>channel</code> or feed.                                                                   |
| <code>webMaster</code>      | The e-mail address for the webmaster of the <code>channel</code> or feed.                                                               |
| <code>pubDate</code>        | The date of the <code>channel</code> or feed release, using the RFC 822 Date and Time Specification—e.g., Sun, 14 Jan 2007 8:00:00 EST. |
| <code>lastBuildDate</code>  | The last date the <code>channel</code> or feed was changed, using the RFC 822 Date and Time Specification.                              |
| <code>category</code>       | The category (or several categories) of the <code>channel</code> or feed. This element has an optional <code>attribute</code> tag.      |
| <code>generator</code>      | Indicates the program that was used to generate the <code>channel</code> or feed.                                                       |

**Fig. 14.36** | `channel` elements and descriptions. (Part 1 of 2.)

| Element   | Description                                                                                                                                                                                                             |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| docs      | The URL of the documentation for the format used in the RSS file.                                                                                                                                                       |
| cloud     | Specifies a SOAP web service that supports the rssCloud interface ( <a href="http://cyber.law.harvard.edu/rss/soapMeetsRss.html#rsscloudInterface">cyber.law.harvard.edu/rss/soapMeetsRss.html#rsscloudInterface</a> ). |
| ttl       | (Time To Live) A number of minutes for how long the channel or feed can be cached before refreshing from the source.                                                                                                    |
| image     | The GIF, JPEG or PNG image that can be displayed with the channel or feed. This element contains the required children title, link and url, and the optional children description, height and width.                    |
| rating    | The PICS (Platform for Internet Content Selection) rating for the channel or feed.                                                                                                                                      |
| textInput | Specifies a text input box to display with the channel or feed. This element contains the required children title, name, link and description.                                                                          |
| skipHours | Tells aggregators which hours they can skip checking for new content.                                                                                                                                                   |
| skipDays  | Tells aggregators which days they can skip checking for new content.                                                                                                                                                    |

**Fig. 14.36** | channel elements and descriptions. (Part 2 of 2.)

| Element     | Description                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| title       | The title of the item.                                                                                              |
| link        | The URL of the item.                                                                                                |
| description | The description of the item.                                                                                        |
| author      | The e-mail address of the author of the item.                                                                       |
| category    | The category (or several categories) of the item. This element has an optional attribute tag.                       |
| comments    | The URL of a page for comments related to the item.                                                                 |
| enclosure   | The location of a media object attached to the item. This element has the required attributes type, url and length. |
| guid        | (Globally Unique Identifier) A string that uniquely identifies the item.                                            |
| pubDate     | The date the item was published, using the RFC 822 Date and Time Specification—e.g., Sun, 14 Jan 2007 8:00:00 EST.  |
| source      | The RSS channel the item came from. This element has a required attribute url.                                      |

**Fig. 14.37** | item elements and descriptions.

### *Browsers and RSS Feeds*

Many of the latest web browsers can now view RSS feeds, determine whether a website offers feeds, allow you to subscribe to feeds and create feed lists. An [RSS aggregator](#) keeps tracks of many RSS feeds and brings together information from the separate feeds. There are many RSS aggregators available, including Bloglines, BottomFeeder, FeedDemon, Microsoft Internet Explorer 7, Mozilla Firefox 2.0, My Yahoo, NewsGator and Opera 9.

To allow browsers and search engines to determine whether a web page contains an RSS feed, a `link` element can be added to the head of a page as follows:

```
<link rel = "alternate" type = "application/rss+xml" title = "RSS"
 href = "file">
```

Many sites provide RSS feed validators. Some examples of RSS feed validators are [validator.w3.org/feed](http://validator.w3.org/feed), [feedvalidator.org](http://feedvalidator.org), and [www.validome.org/rss-atom/](http://www.validome.org/rss-atom/).

### *Creating a Feed Aggregator*

The DOM and XSL can be used to create RSS aggregators. A simple RSS aggregator uses an XSL stylesheet to format RSS feeds as XHTML. Figure 14.38 loads two XML documents—an RSS feed (a small portion of which is shown in Fig. 14.39) and an XSL style sheet—then uses JavaScript to apply an XSL transformation to the RSS content and render it on the page. You'll notice as we discuss this program that there is little commonality between Internet Explorer 7 and Firefox with regard to programmatically applying XSL transformations. This is one of the reasons that JavaScript libraries have become popular in web development—they tend to hide such browser-specific issues from you. We discuss the Dojo toolkit—one of many popular JavaScript libraries—in Section 15.8. For more information on JavaScript libraries, see our JavaScript and Ajax Resource Centers ([www.deitel.com/JavaScript/](http://www.deitel.com/JavaScript/) and [www.deitel.com/Ajax/](http://www.deitel.com/Ajax/), respectively).

### *Determining the Browser Type and Loading the Documents*

When this page first loads, lines 19–23 (Fig. 14.38) determine whether the browser is Internet Explorer 7 or Firefox 2 and store the result in variable `browser` for use throughout the script. After the body of this XHTML document loads, its `onload` event calls function `start` (lines 26–48) to load RSS and XSL files as XML documents, and to transform the RSS. Since Internet Explorer 7 can download the files synchronously, lines 30–33 perform the loading, transformation and display steps sequentially. As mentioned previously, Firefox 2 loads the files asynchronously. For this reason, line 37 starts loading the `rss.xsl` document (included with this example's code), and lines 38–46 register an `onload` event handler for that document. When the document finishes loading, line 40 begins loading the `deitel-20.xml` RSS document. Lines 41–45 register an `onload` event handler for this second document. When it finishes loading, lines 43–44 perform the transformation and display the results.

### *Transforming the RSS to XHTML*

Function `applyTransform` (Fig. 14.38, lines 75–96) performs the browser-specific XSL transformations using the RSS document and XSL document it receives as arguments. Line 81 uses the MSXML object's built-in XSLT capabilities to apply the transformations. Method `transformNode` is invoked on the `rssDocument` object and receives the `xslDocument` object as an argument.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 14.38: RssViewer.html -->
6 <!-- Simple RSS viewer. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Simple RSS Viewer</title>
10 <style type = "text/css">
11 #outputDiv { font: 12px Verdana, Geneva, Arial,
12 Helvetica, sans-serif; }
13 </style>
14 <script type = "text/javascript">
15 <!--
16 var browser = ""; // used to determine which browser is being used
17
18 // is the browser Internet Explorer 7 or Firefox 2?
19 if (window.ActiveXObject) // IE7
20 browser = "IE7";
21 else if (document.implementation &&
22 document.implementation.createDocument) // FF2 and other browsers
23 browser = "FF2";
24
25 // load both the RSS feed and the XSL file to process it
26 function start()
27 {
28 if (browser == "IE7")
29 {
30 var xsl = loadXMLDocument('rss.xsl'); // load XSL file
31 var rss = loadXMLDocument('deitel-20.xml'); // load RSS feed
32 var result = applyTransform(rss, xsl); // apply transform
33 displayTransformedRss(result); // display feed info
34 } // end if
35 else if (browser == "FF2")
36 {
37 var xsl = loadXMLDocument('rss.xsl'); // load XSL file
38 xsl.onload = function() // function to execute when xsl loads
39 {
40 var rss = loadXMLDocument('deitel-20.xml'); // load RSS feed
41 rss.onload = function() // function to execute when rss loads
42 {
43 var result = applyTransform(rss, xsl); // apply transform
44 displayTransformedRss(result); // display feed info
45 } // end onload event handler for rss
46 } // end onload event handler for xsl
47 } // end else
48 } // end function start
49
50 // load XML document based on whether the browser is IE7 or Firefox 2
51 function loadXMLDocument(url)
52 {
53 var doc = ""; // variable to manage loading file

```

**Fig. 14.38** | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part I of 3.)

```

54 if (browser == "IE7") // IE7
55 {
56 // create IE7-specific XML document object
57 doc = new ActiveXObject("Msxml2.DOMDocument.6.0");
58 doc.async = false; // specifies synchronous loading of XML doc
59 doc.load(url); // load the XML document specified by url
60 } // end if
61 else if (browser == "FF2") // other browsers
62 {
63 // create XML document object
64 doc = document.implementation.createDocument("", "", null);
65 doc.load(url); // load the XML document specified by url
66 } // end else
67 else // not supported
68 alert('This script is not supported by your browser');
69
70 return doc; // return the loaded document
71 } // end function loadXMLDocument
72
73
74 // apply XSL transformation and show results
75 function applyTransform(rssDocument, xslDocument)
76 {
77 var result; // stores transformed RSS
78
79 // transform the RSS feed to XHTML
80 if (browser == "IE7")
81 result = rssDocument.transformNode(xslDocument);
82 else // browser == "FF2"
83 {
84 // create Firefox object to perform transformation
85 var xsltProcessor = new XSLTProcessor();
86
87 // specify XSL stylesheet to use in transformation
88 xsltProcessor.importStylesheet(xslDocument);
89
90 // apply the transformation
91 result =
92 xsltProcessor.transformToFragment(rssDocument, document);
93 } // end else
94
95 return result; // return the transformed RSS
96 } // end function applyTransform
97
98 // display the XML document and highlight the first child
99 function displayTransformedRss(resultXHTML)
100 {
101 if (browser == "IE7")
102 document.getElementById("outputDiv").innerHTML = resultXHTML;
103 else // browser == "FF2"
104 document.getElementById("outputDiv").appendChild(
105 resultXHTML);
106 } // end function displayTransformedRss

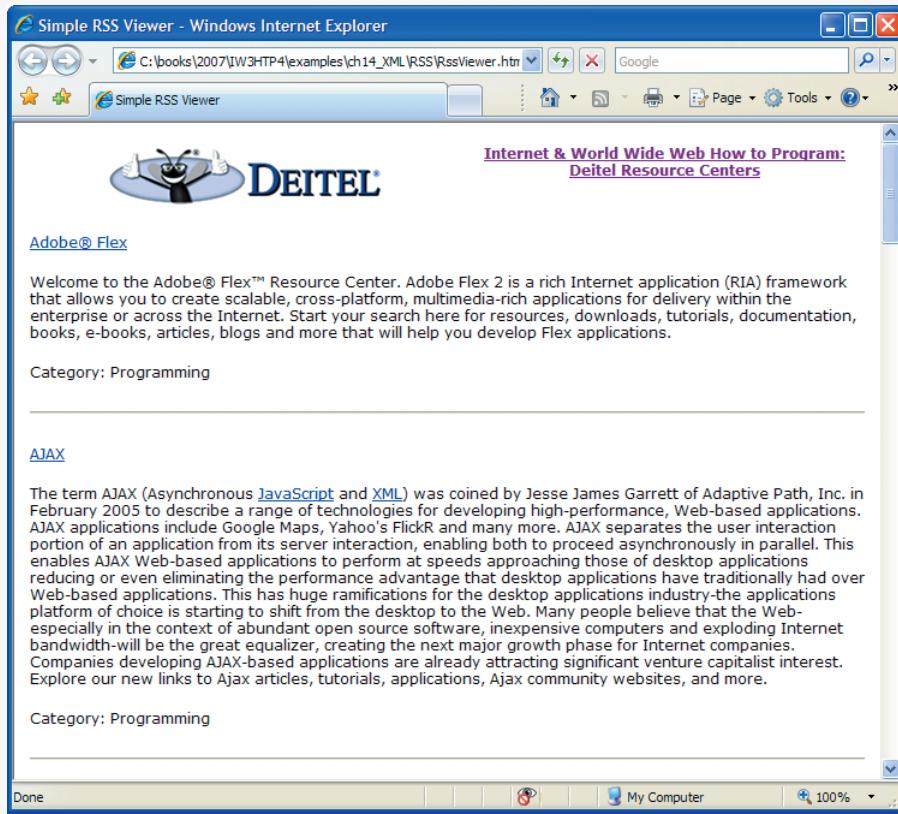
```

**Fig. 14.38** | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 2 of 3.)

```

107 // -->
108 </script>
109 </head>
110 <body id = "body" onload = "start();">
111 <div id = "outputDiv"></div>
112 </body>
113 </html>

```



**Fig. 14.38** | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 3 of 3.)

```

1 <?xml version="1.0" encoding="utf-8"?>
2
3 <!-- Fig. 14.39: deitel-20.xml -->
4 <!-- RSS 2.0 feed of Deitel Resource Centers -->
5 <rss version="2.0">
6 <channel>
7 <title>
8 Internet & World Wide Web How to Program:
9 Deitel Resource Centers
10 </title>

```

**Fig. 14.39** | RSS 2.0 sample feed. (Part 1 of 2.)

```

11 <link>http://www.deitel.com/ResourceCenters.html</link>
12 <description>
13 Check out our growing network of Resource Centers that focus on
14 many of today's hottest programming, Web 2.0 and technology
15 topics. Start your search here for downloads, tutorials,
16 documentation, books, e-books, blogs, RSS feeds, journals,
17 articles, training, webcasts, podcasts, videos and more.
18 </description>
19 <language>en-us</language>
20 <image>
21 <url>
22 http://www.deitel.com/Portals/0/deitel_transparent_smaller.png
23 </url>
24 <title>Deitel.com</title>
25 <link>http://www.deitel.com/</link>
26 </image>
27
28 <item>
29 <title>Adobe® Flex</title>
30 <link>http://www.deitel.com/Flex/</link>
31 <description>
32 <p>
33 Welcome to the Adobe® Flex™ Resource Center. Adobe Flex 2 is a
34 rich Internet application (RIA) framework that allows you to
35 create scalable, cross-platform, multimedia-rich applications
36 for delivery within the enterprise or across the Internet.
37 Start your search here for resources, downloads, tutorials,
38 documentation, books, e-books, articles, blogs and more that
39 will help you develop Flex applications.
40 </p>
41 </description>
42 <category>Programming</category>
43 </item>
44 </channel>
45 </rss>

```

**Fig. 14.39** | RSS 2.0 sample feed. (Part 2 of 2.)

Firefox provides built-in XSLT processing in the form of the `XSLTProcessor` object (created at line 85). After creating this object, you use its `importStylesheet` method to specify the XSL stylesheet you'd like to apply (line 88). Finally, lines 91–92 apply the transformation by invoking the `XSLTProcessor` object's `transformToFragment` method, which returns a document fragment—i.e., a piece of a document. In our case, the `rss.xsl` document transforms the RSS into an XHTML `table` element that we'll append to the `outputDiv` element in our XHTML page. The arguments to `transformToFragment` are the document to transform and the document object to which the transformed fragment will belong. To learn more about `XSLTProcessor`, visit [developer.mozilla.org/en/docs/The\\_XSLT/JavaScript\\_Interface\\_in\\_Gecko](https://developer.mozilla.org/en/docs/The_XSLT/JavaScript_Interface_in_Gecko).

In each browser's case, after the transformation, the resulting XHTML markup is assigned to variable `result` and returned from function `applyTransform`. Then function `displayTransformedRss` is called.

### *Displaying the XHTML Markup*

Function `displayTransformedRss` (lines 99–106) displays the transformed RSS in the `outputDiv` element (line 111 in the body). In both Internet Explorer 7 and Firefox 2, we use the DOM method `getElementById` to obtain the `outputDiv` element. In Internet Explorer 7, the node's `innerHTML` property is used to add the table as a child of the `outputDiv` element (line 102). In Firefox, the node's `appendChild` method must be used to append the table (a document fragment) to the `outputDiv` element.

## 14.11 Wrap-Up

In this chapter, we studied Extensible Markup Language and several of its related technologies. We began by discussing some basic XML terminology, introducing the concepts of markup, XML vocabularies and XML parsers (validating and nonvalidating). We then demonstrated how to describe and structure data in XML, illustrating these points with examples marking up an article and a business letter.

The chapter discussed the concept of an XML namespace. You learned that each namespace has a unique name that provides a means for document authors to unambiguously refer to elements with the same name (i.e., prevent naming collisions). We presented examples of defining two namespaces in the same document, as well as setting the default namespace for a document.

We also discussed how to create DTDs and schemas for specifying and validating the structure of an XML document. We showed how to use various tools to confirm whether XML documents are valid (i.e., conform to a DTD or schema).

The chapter demonstrated how to create and use XSL documents to specify rules for converting XML documents between formats. Specifically, you learned how to format and sort XML data as XHTML for display in a web browser.

The final sections of the chapter presented more advanced uses of XML. We demonstrated how to retrieve and display data from an XML document using JavaScript. We illustrated how a Document Object Model (DOM) tree represents each element of an XML document as a node in the tree. You also learned how to traverse the DOM tree, interact with individual nodes in the DOM tree from JavaScript code, search for nodes using XPath and apply XSL transformations.

Chapter 15 begins our discussion of Rich Internet Applications (RIAs)—web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and rich GUI. You'll learn about Ajax (Asynchronous JavaScript and XML), which uses all the concepts you've learned so far to build rich web applications. You'll see that Ajax techniques are key to the responsiveness of RIAs.

## 14.12 Web Resources

[www.deitel.com/XML/](http://www.deitel.com/XML/)

The Deitel XML Resource Center focuses on the vast amount of free XML content available online, plus some for-sale items. Start your search here for tools, downloads, tutorials, podcasts, wikis, documentation, conferences, FAQs, books, e-books, sample chapters, articles, newsgroups, forums, downloads from CNET's download.com, jobs and contract opportunities, and more that will help you develop XML applications.

## Summary

### Section 14.1 Introduction

- XML is a portable, widely supported, open (i.e., nonproprietary) technology for data storage and exchange.

### Section 14.2 XML Basics

- XML documents are readable by both humans and machines.
- XML permits document authors to create custom markup for any type of information. This enables document authors to create entirely new markup languages that describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes.
- An XML parser is responsible for identifying components of XML documents (typically files with the .xml extension) and then storing those components in a data structure for manipulation.
- An XML document can optionally reference a Document Type Definition (DTD) or schema that defines the XML document's structure.
- An XML document that conforms to a DTD/schema (i.e., has the appropriate structure) is valid.
- If an XML parser (validating or nonvalidating) can process an XML document successfully, that XML document is well-formed.

### Section 14.3 Structuring Data

- An XML document begins with an optional XML declaration, which identifies the document as an XML document. The version attribute specifies the version of XML syntax used in the document.
- XML comments begin with <!-- and end with -->.
- An XML document contains text that represents its content (i.e., data) and elements that specify its structure. XML documents delimit an element with start and end tags.
- The root element of an XML document encompasses all its other elements.
- XML element names can be of any length and can contain letters, digits, underscores, hyphens and periods. However, they must begin with either a letter or an underscore, and they should not begin with “xml” in any combination of uppercase and lowercase letters, as this is reserved for use in the XML standards.
- When a user loads an XML document in a browser, a parser parses the document, and the browser uses a style sheet to format the data for display.
- IE and Firefox each display minus (–) or plus (+) signs next to all container elements. A minus sign indicates that all child elements are being displayed. When clicked, a minus sign becomes a plus sign (which collapses the container element and hides all the children), and vice versa.
- Data can be placed between tags or in attributes (name/value pairs that appear within the angle brackets of start tags). Elements can have any number of attributes.

### Section 14.4 XML Namespaces

- XML allows document authors to create their own markup, and as a result, naming collisions (i.e., two different elements that have the same name) can occur. XML namespaces provide a means for document authors to prevent collisions.
- Each namespace prefix is bound to a uniform resource identifier (URI) that uniquely identifies the namespace. A URI is a series of characters that differentiate names. Document authors create their own namespace prefixes. Any name can be used as a namespace prefix, but the namespace prefix xml is reserved for use in XML standards.

- To eliminate the need to place a namespace prefix in each element, authors can specify a default namespace for an element and its children. We declare a default namespace using keyword `xmlns` with a URI (Uniform Resource Identifier) as its value.
- Document authors commonly use URLs (Uniform Resource Locators) for URIs, because domain names (e.g., `deitel.com`) in URLs must be unique.

### ***Section 14.5 Document Type Definitions (DTDs)***

- DTDs and schemas specify documents' element types and attributes, and their relationships to one another.
- DTDs and schemas enable an XML parser to verify whether an XML document is valid (i.e., its elements contain the proper attributes and appear in the proper sequence).
- A DTD expresses the set of rules for document structure using an EBNF (Extended Backus-Naur Form) grammar.
- In a DTD, an `ELEMENT` element type declaration defines the rules for an element. An `ATTLIST` attribute-list declaration defines attributes for a particular element.

### ***Section 14.6 W3C XML Schema Documents***

- Unlike DTDs, schemas do not use EBNF grammar. Instead, they use XML syntax and are themselves XML documents that programs can manipulate.
- Unlike DTDs, XML Schema documents can specify what type of data (e.g., numeric, text) an element can contain.
- An XML document that conforms to a schema document is schema valid.
- Two categories of types exist in XML Schema: simple types and complex types. Simple types cannot contain attributes or child elements; complex types can.
- Every simple type defines a restriction on an XML Schema-defined schema type or on a user-defined type.
- Complex types can have either simple content or complex content. Both simple and complex content can contain attributes, but only complex content can contain child elements.
- Whereas complex types with simple content must extend or restrict some other existing type, complex types with complex content do not have this limitation.

### ***Section 14.7 XML Vocabularies***

- XML allows authors to create their own tags to describe data precisely.
- Some of these XML vocabularies includ MathML (Mathematical Markup Language), Scalable Vector Graphics (SVG), Wireless Markup Language (WML), Extensible Business Reporting Language (XBRL), Extensible User Interface Language (XUL), Product Data Markup Language (PDML), W3C XML Schema and Extensible Stylesheet Language (XSL).
- MathML markup describes mathematical expressions for display. MathML is divided into two types of markup—content markup and presentation markup.
- Content markup provides tags that embody mathematical concepts. Content MathML allows programmers to write mathematical notation specific to different areas of mathematics.
- Presentation MathML is directed toward formatting and displaying mathematical notation. We focus on Presentation MathML in the MathML examples.
- By convention, MathML files end with the `.mml` filename extension.
- A MathML document's root node is the `math` element and its default namespace is `http://www.w3.org/1998/Math/MathML`.

- The  $\text{mn}$  element marks up a number. The  $\text{mo}$  element marks up an operator.
- Entity reference  $\&\text{InvisibleTimes};$  indicates a multiplication operation without explicit symbolic representation.
- The  $\text{msup}$  element represents a superscript. It has two children—the expression to be superscripted (i.e., the base) and the superscript (i.e., the exponent). Correspondingly, the  $\text{msub}$  element represents a subscript.
- To display variables, use identifier element  $\text{mi}.$
- The  $\text{mfrac}$  element displays a fraction. If either the numerator or the denominator contains more than one element, it must appear in an  $\text{mrow}$  element.
- An  $\text{mrow}$  element is used to group elements that are positioned horizontally in an expression.
- The entity reference  $\&\text{int};$  represents the integral symbol.
- The  $\text{msupsub}$  element specifies the subscript and superscript of a symbol. It requires three child elements—an operator, the subscript expression and the superscript expression.
- Element  $\text{msqrt}$  represents a square-root expression.
- Entity reference  $\&\text{delta};$  represents a lowercase delta symbol.

### ***Section 14.8 Extensible Stylesheet Language and XSL Transformations***

- XSL can convert XML into any text-based document. XSL documents have the extension `.xsl`.
- XPath is a string-based language of expressions used by XML and many of its related technologies for effectively and efficiently locating structures and data (such as specific elements and attributes) in XML documents.
- XPath is used to locate parts of the source-tree document that match templates defined in an XSL style sheet. When a match occurs (i.e., a node matches a template), the matching template executes and adds its result to the result tree. When there are no more matches, XSLT has transformed the source tree into the result tree.
- The XSLT does not analyze every node of the source tree; it selectively navigates the source tree using XPath's `select` and `match` attributes.
- For XSLT to function, the source tree must be properly structured. Schemas, DTDs and validating parsers can validate document structure before using XPath and XSLTs.
- XSL style sheets can be connected directly to an XML document by adding an `xml:stylesheet` processing instruction to the XML document.
- Two tree structures are involved in transforming an XML document using XSLT—the source tree (the document being transformed) and the result tree (the result of the transformation).
- The XPath character `/` (a forward slash) always selects the document root. In XPath, a leading forward slash specifies that we are using absolute addressing.
- An XPath expression with no beginning forward slash uses relative addressing.
- XSL element `value-of` retrieves an attribute's value. The `@` symbol specifies an attribute node.
- XSL node-set function `name` retrieves the current node's element name.
- XSL node-set function `text` retrieves the text between an element's start and end tags.
- The XPath expression `/*` selects all the nodes in an XML document.

### ***Section 14.9 Document Object Model (DOM)***

- Although an XML document is a text file, retrieving data from the document using traditional sequential file processing techniques is neither practical nor efficient, especially for adding and removing elements dynamically.

- Upon successfully parsing a document, some XML parsers store document data as tree structures in memory. This hierarchical tree structure is called a Document Object Model (DOM) tree, and an XML parser that creates this type of structure is known as a DOM parser.
- Each element name is represented by a node. A node that contains other nodes is called a parent node. A parent node can have many children, but a child node can have only one parent node.
- Nodes that are peers are called sibling nodes.
- A node's descendant nodes include its children, its children's children and so on. A node's ancestor nodes include its parent, its parent's parent and so on.
- Many of the XML DOM capabilities are similar or identical to those of the XHTML DOM.
- The DOM tree has a single root node, which contains all the other nodes in the document.
- If `window.ActiveXObject` exists, the browser is Internet Explorer. An `ActiveXObject` that loads Microsoft's MSXML parser is used to manipulate XML documents in Internet Explorer.
- MSXML's `load` method loads an XML document.
- A document's `childNodes` property contains a list of the XML document's top-level nodes.
- If the browser is Firefox 2, then the `document` object's `implementation` property and the `implementation` property's `createDocument` method will exist.
- Firefox loads each XML document asynchronously, so you must use the XML document's `onLoad` property to specify a function to call when the document finishes loading to ensure that you can access the document's contents.
- A node's `nodeType` property contains the type of the node.
- Nonbreaking spaces (`&ampnbsp`) are spaces that the browser is not allowed to collapse or that can be used to keep words together.
- The name of an element can be obtained by the node's `nodeName` property.
- If the current node has children, the length of the node's `childNodes` list is nonzero.
- The `nodeValue` property returns the value of an element.
- Node property `firstChild` refers to the first child of a given node. Similarly, `lastChild` refers to the last child of a given node.
- Node property `nextSibling` refers to the next sibling in a list of children of a particular node. Similarly, `previousSibling` refers to the current node's previous sibling.
- Property `parentNode` refers to the current node's parent node.
- A simpler way to locate nodes is to search for lists of node-matching search criteria that are written as XPath expressions.
- In IE7, the XML document object's `selectNodes` method receives an XPath expression as an argument and returns a collection of elements that match the expression.
- Firefox 2 searches for XPath matches using the XML document object's `evaluate` method, which receives five arguments—the XPath expression, the document to apply the expression to, a namespace resolver, a result type and an `XPathResult` object into which to place the results. If the last argument is `null`, the function simply returns a new `XPathResult` object containing the matches. The namespace resolver argument can be `null` if you are not using XML namespace prefixes in the XPath processing.

### Section 14.10 RSS

- RSS stands for RDF (Resource Description Framework) Site Summary and is also known as Rich Site Summary and Really Simple Syndication.

- RSS is an XML format used to syndicate simple website content, such as news articles, blog entries, product reviews, podcasts, vodcasts and more.
- An RSS feed contains an `rss` root element with a `version` attribute and a `channel` child element with `item` subelements. Depending on the RSS version, the `channel` and `item` elements have certain required and optional child elements.
- The `item` elements provide the feed subscriber with a link to a web page or file, a title and description of the page or file.
- By providing up-to-date, free and linkable content for anyone to use, RSS enables website developers to draw more traffic.
- In RSS 2.0, the required child elements of `channel` are `description`, `link` and `title`, and the required child element of an `item` is either `title` or `description`.
- An RSS aggregator keeps tracks of many RSS feeds and brings together information from the separate feeds.
- Many sites provide RSS feed validators. Some examples of RSS feed validators are `validator.w3.org/feed`, `feedvalidator.org`, and `www.validome.org/rss-atom/`.
- The DOM and XSL can be used to create RSS aggregators. A simple RSS aggregator uses an XSL style sheet to format RSS feeds as XHTML.
- MSXML's built-in XSLT capabilities include method `transformNode` to apply an XSLT transformation. It is invoked on an RSS document object and receives the XSL document object as an argument.
- Firefox provides built-in XSLT processing in the form of the `XSLTProcessor` object. After creating this object, you use its `importStylesheet` method to specify the XSL style sheet you'd like to apply. Finally, you apply the transformation by invoking the `XSLTProcessor` object's `transformToFragment` method, which returns a document fragment.

## Terminology

|                                                                    |                                                                                                  |
|--------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| <code>&amp;delta;</code> entity reference (MathML)                 | attribute element                                                                                |
| <code>&amp;InvisibleTimes;</code> entity reference (MathML)        | attribute in XML                                                                                 |
| <code>&amp;int;</code> entity reference (MathML)                   | attribute-list declaration                                                                       |
| <code>&amp;nbsp;</code>                                            | attribute value in XML                                                                           |
| <code>.mml</code> filename extension for MathML documents          | base attribute of element extension                                                              |
| <code>/</code> , forward slash in end tags                         | base type (XML Schema)                                                                           |
| <code>/</code> , XPath root selector                               | CDATA keyword (DTD)                                                                              |
| <code>&lt;!--...--&gt;</code> , XML comment tags                   | channel child element of an <code>rss</code> element                                             |
| <code>&lt;? and ?&gt;</code> XML processing instruction delimiters | character data in XML                                                                            |
| <code>@</code> , XPath attribute symbol                            | child element                                                                                    |
| absolute addressing (XPath)                                        | child node (DOM tree)                                                                            |
| <code>ActiveXObject</code> from Internet Explorer                  | childNodes property of a Node                                                                    |
| <code>a11</code> XML Schema element                                | complex content in XML Schema                                                                    |
| Amaya (W3C browser)                                                | <code>complexType</code> XML Schema element                                                      |
| ancestor node                                                      | container element                                                                                |
| <code>appendChild</code> method of a Node                          | content                                                                                          |
| asterisk (*) occurrence indicator                                  | context node (XPath)                                                                             |
| <code>ATTLIST</code> attribute-list declaration (DTD)              | <code>createDocument</code> method of the document object's <code>implementation</code> property |
| <code>Attr</code> object                                           | data-type attribute (XPath)                                                                      |
|                                                                    | default namespace                                                                                |

descendant node  
 DOCTYPE parts  
 document object  
 Document Object Model (DOM) tree  
 document root  
 Document Type Definition (DTD)  
 DOM parser  
 .dtd filename extension  
 element (XML)  
 ELEMENT element type declaration (DTD)  
 Element object  
 element type declaration  
 element XML Schema element  
 EMPTY keyword (DTD)  
 enclosure element (RSS)  
 end tag  
 evaluate method of a Firefox 2 XML document  
     object  
 Expat XML Parser  
 Extensible Stylesheet Language (XSL)  
 Extensible User Interface Language (XUL)  
 base attribute  
 extension XML Schema element  
 external DTD  
 firstChild property of a DOM node  
 #FIXED keyword (DTD)  
 forward slash character (/) in end tags  
 getElementById method of the document object  
 getElementsByTagName method  
 identifier element (MathML)  
 implementation property  
 implementation property of the document object  
 #IMPLIED keyword (DTD)  
 importStylesheet method of the XSLTProcessor object (Firefox)  
 item subelement of channel child element of an  
     rss element  
 lastChild property of a DOM node  
 load method of the ActiveXObject object  
 markup in XML  
 match attribute  
 Mathematical Markup Language (MathML)  
 maxOccurs XML Schema attribute  
 mfrac MathML element  
 mi MathML element  
 Microsoft XML Core Services (MSXML)  
 minInclusive XML Schema element  
 minOccurs XML Schema attribute  
 mn MathML element  
     mo MathML element  
     mrow MathML element  
     msqrt MathML element  
     msupsub MathML element  
     msup MathML element  
     MSXML (Microsoft XML Core Services)  
     MSXML parser  
     name attribute (XPath)  
     name node-set function  
     name XML Schema attribute  
     namespace prefix  
     naming collision  
     nested element  
     nextSibling property of a DOM node  
     Node object  
     node-set function  
     for-each element  
     NodeList object  
     nodeName property of a DOM node  
     nodeType property of a DOM node  
     nodeValue property of a DOM node  
     nonbreaking space (&nbsp;)  
     nonvalidating XML parser  
     occurrence indicator  
     omit-xml-declaration attribute  
     open technology  
     order attribute  
     parent element  
     parent node  
     parentNode property of a DOM node  
     parsed character data  
     parser  
     partial RSS feed  
     #PCDATA keyword (DTD)  
     plus sign (+) occurrence indicator  
     presentation  
     previousSibling property of a Node  
     processing instruction (PI)  
     processing instruction target  
     processing instruction value  
     Product Data Markup Language (PDML)  
     prolog (XML)  
     question mark (?) occurrence indicator  
     RDF (Resource Description Framework)  
     RDF Site Summary (RSS)  
     Really Simple Syndication (RSS)  
     recursive descent  
     relative addressing (XPath)  
     replaceChild method of a Node  
     #REQUIRED keyword (DTD)

Research Information Exchange Markup Language (RIXML) 863  
 Resource Description Framework (RDF)  
 restriction on built-in XML Schema data type  
 result tree (XSLT)  
 Rich Site Summary (RSS)  
 root element (XML)  
 root node  
 RSS (RDF Site Summary)  
 RSS 2.0 sample feed  
 RSS aggregator  
 rss root element  
 Scalable Vector Graphics (SVG)  
 schema invalid document  
 schema repository  
 schema valid XML document  
 schema XML Schema element  
 select attribute (XPath)  
 select attribute of `xsl:for-each` element  
`selectNodes` method of MSXML  
 sibling node  
 siblings  
 simple content in XML Schema  
`simpleContent` XML Schema element  
`simpleType` XML Schema element  
 source tree (XSLT)  
 square-root symbol (MathML)  
 start tag  
 string XML Schema data type  
 style sheet  
`stylesheet` start tag  
`sum` function (XSL)  
 symbolic representation (MathML)  
 SYSTEM keyword in XML  
 targetNamespace XML Schema attribute  
 text node-set function  
 Text object  
`transformNode` method of an MSXML document object  
`transformToFragment` method of the XSLTProcessor object (Firefox)  
 type attribute in a processing instruction  
 type XML Schema attribute  
 validating XML parser  
 version attribute (XSL)  
 version in `xml` declaration  
 well-formed XML document  
 Wireless Markup Language (WML)  
 Xalan XSLT processor  
 XBRL (Extensible Business Reporting Language)  
 Xerces parser from Apache  
 XML (Extensible Markup Language)  
`.xml` file extension  
 XML instance document  
 XML Path Language (XPath)  
 XML processor  
 XML Schema  
 XML vocabulary  
`xmlns` attribute in XML  
 XPath (XML Path Language)  
 XPath expression  
`XPathResult` object  
`.xsd` filename extension  
 XSL (Extensible Stylesheet Language)  
`.xsl` filename extension  
 XSL-FO (XSL Formatting Objects)  
 XSL Formatting Objects (XSL-FO)  
 XSL style sheet  
 XSL template  
 XSL variable  
`xsl:for-each` element  
`xsl:output` element  
`xsl:template` element  
`xsl:value-of` element  
 XSLTProcessor object (Firefox)  
 XUL (Extensible User Interface Language)

## Self-Review Exercises

- 14.1** Which of the following are valid XML element names? (Select all that apply.)
- yearBorn
  - year.Born
  - year Born
  - year-Born1
  - 2\_year\_born
  - \_year\_born\_

- 14.2** State which of the following statements are *true* and which are *false*. If *false*, explain why.
- XML is a technology for creating markup languages.
  - XML markup is delimited by forward and backward slashes (/ and \).
  - All XML start tags must have corresponding end tags.
  - Parsers check an XML document's syntax.
  - XML does not support namespaces.
  - When creating XML elements, document authors must use the set of XML tags provided by the W3C.
  - The pound character (#), dollar sign (\$), ampersand (&) and angle brackets (< and >) are examples of XML reserved characters.
  - XML is not case sensitive.
  - XML Schemas are better than DTDs, because DTDs lack a way of indicating what specific type of data (e.g., numeric, text) an element can contain and DTDs are not themselves XML documents.
  - DTDs are written using an XML vocabulary.
  - Schema is a technology for locating information in an XML document.
- 14.3** Fill in the blanks for each of the following:
- \_\_\_\_\_ help prevent naming collisions.
  - \_\_\_\_\_ embed application-specific information into an XML document.
  - \_\_\_\_\_ is Microsoft's XML parser.
  - XSL element \_\_\_\_\_ writes a DOCTYPE to the result tree.
  - XML Schema documents have root element \_\_\_\_\_.
  - XSL element \_\_\_\_\_ is the root element in an XSL document.
  - XSL element \_\_\_\_\_ selects specific XML elements using repetition.
  - Nodes that contain other nodes are called \_\_\_\_\_ nodes.
  - Nodes that are peers are called \_\_\_\_\_ nodes.
- 14.4** In Fig. 14.2, we subdivided the author element into more detailed pieces. How might you subdivide the date element? Use the date May 5, 2005, as an example.
- 14.5** Write a processing instruction that includes style sheet `wap.xsl`.
- 14.6** Write an XPath expression that locates `contact` nodes in `letter.xml` (Fig. 14.4).

## Answers to Self-Review Exercises

- 14.1** a, b, d, f. [Choice c is incorrect because it contains a space. Choice e is incorrect because the first character is a number.]
- 14.2** a) True. b) False. In an XML document, markup text is delimited by tags enclosed in angle brackets (< and >) with a forward slash just after the < in the end tag. c) True. d) True. e) False. XML does support namespaces. f) False. When creating tags, document authors can use any valid name but should avoid ones that begin with the reserved word `xml` (also `XML`, `Xm1`, etc.). g) False. XML reserved characters include the ampersand (&), the left angle bracket (<) and the right angle bracket (>), but not # and \$. h) False. XML is case sensitive. i) True. j) False. DTDs use EBNF grammar, which is not XML syntax. k) False. XPath is a technology for locating information in an XML document. XML Schema provides a means for type checking XML documents and verifying their validity.
- 14.3** a) Namespaces. b) Processing instructions. c) MSXML. d) `xsl:output`. e) schema. f) `xsl:stylesheet`. g) `xsl:for-each`. h) parent. i) sibling.

- 14.4**    <date>  
          <month>May</month>  
          <day>5</day>  
          <year>2005</year>  
      </date>.
- 14.5**    <xsl:stylesheet type = "text/xsl" href = "wap.xsl"?>
- 14.6**    /letter/contact.

## Exercises

**14.7** (*Nutrition Information XML Document*) Create an XML document that marks up the nutrition facts for a package of Grandma White's cookies. A package of cookies has a serving size of 1 package and the following nutritional value per serving: 260 calories, 100 fat calories, 11 grams of fat, 2 grams of saturated fat, 5 milligrams of cholesterol, 210 milligrams of sodium, 36 grams of total carbohydrates, 2 grams of fiber, 15 grams of sugars and 5 grams of protein. Name this document *nutrition.xml*. Load the XML document into Internet Explorer. [Hint: Your markup should contain elements describing the product name, serving size/amount, calories, sodium, cholesterol, proteins, etc. Mark up each nutrition fact/ingredient listed above.]

**14.8** (*Nutrition Information XML Schema*) Write an XML Schema document (*nutrition.xsd*) specifying the structure of the XML document created in Exercise 14.7.

**14.9** (*Nutrition Information XSL Style Sheet*) Write an XSL style sheet for your solution to Exercise 14.7 that displays the nutritional facts in an XHTML table. Modify Fig. 14.38 to output the results.

**14.10** (*Sorting XSLT Modification*) Modify Fig. 14.23 (*sorting.xsl*) to sort by the number of pages rather than by chapter number. Save the modified document as *sorting\_byPage.xsl*.

**14.11** Modify Fig. 14.38 to use *sorting.xml* (Fig. 14.22), *sorting.xsl* (Fig. 14.23) and *sorting\_byPage.xsl* (from Exercise 14.10). Display the result of transforming *sorting.xml* using each style sheet. [Hint: Remove the *xm1:stylesheet* processing instruction from line 2 of *sort-ing.xml* before attempting to transform the file programmatically.]

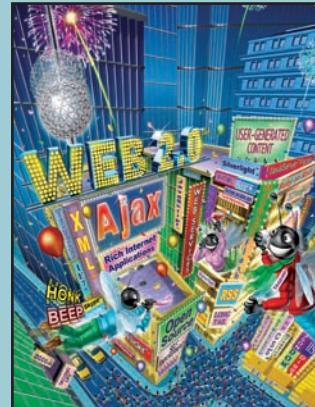
# 15

# Ajax-Enabled Rich Internet Applications

## OBJECTIVES

In this chapter you will learn:

- What Ajax is and why it is important for building Rich Internet Applications.
- What asynchronous requests are and how they help give web applications the feel of desktop applications.
- What the XMLHttpRequest object is and how it's used to create and manage asynchronous requests to servers and to receive asynchronous responses from servers.
- Methods and properties of the XMLHttpRequest object.
- How to use XHTML, JavaScript, CSS, XML, JSON and the DOM in Ajax applications.
- How to use Ajax frameworks and toolkits, specifically Dojo, to conveniently create robust Ajax-enabled Rich Internet Applications.
- About resources for studying Ajax-related issues such as security, performance, debugging, the “back-button problem” and more.



*... the challenges are for the designers of these applications: to forget what we think we know about the limitations of the Web, and begin to imagine a wider, richer range of possibilities. It's going to be fun.*

—Jesse James Garrett

*Dojo is the standard library JavaScript never had.*

—Alex Russell

*To know how to suggest is the great art of teaching. To attain it we must be able to guess what will interest ...*

—Henri-Fredreic Amiel

*It is characteristic of the epistemological tradition to present us with partial scenarios and then to demand whole or categorical answers as it were.*

—Avrum Stroll

*O! call back yesterday, bid time return.*

—William Shakespeare

**Outline**

- 15.1 Introduction
- 15.2 Traditional Web Applications vs. Ajax Applications
- 15.3 Rich Internet Applications (RIAs) with Ajax
- 15.4 History of Ajax
- 15.5 “Raw” Ajax Example Using the XMLHttpRequest Object
- 15.6 Using XML and the DOM
- 15.7 Creating a Full-Scale Ajax-Enabled Application
- 15.8 Dojo Toolkit
- 15.9 Wrap-Up
- 15.10 Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 15.1 Introduction

Despite the tremendous technological growth of the Internet over the past decade, the usability of web applications has lagged behind compared to that of desktop applications. Every significant interaction in a web application results in a waiting period while the application communicates over the Internet with a server. **Rich Internet Applications (RIAs)** are web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and a rich GUI.

RIA performance comes from [Ajax \(Asynchronous JavaScript and XML\)](#), which uses client-side scripting to make web applications more responsive. Ajax applications separate client-side user interaction and server communication, and run them in parallel, reducing the delays of server-side processing normally experienced by the user.

There are many ways to implement Ajax functionality. “[Raw](#)” Ajax uses JavaScript to send asynchronous requests to the server, then updates the page using the DOM (see Section 15.5). “Raw” Ajax is best suited for creating small Ajax components that asynchronously update a section of the page. However, when writing “raw” Ajax you need to deal directly with cross-browser portability issues, making it impractical for developing large-scale applications. These portability issues are hidden by [Ajax toolkits](#), such as [Dojo](#) (Section 15.8), [Prototype](#), [Script.aculo.us](#) and ASP.NET Ajax, which provide powerful ready-to-use controls and functions that enrich web applications, and simplify JavaScript coding by making it cross-browser compatible.

Traditional web applications use XHTML forms (Chapter 4) to build simple and thin GUIs compared to the rich GUIs of Windows, Macintosh and desktop systems in general. We achieve rich GUI in RIAs with Ajax toolkits and with RIA environments such as Adobe’s Flex (Chapter 18), Microsoft’s Silverlight (Chapter 19) and JavaServer Faces (Chapters 26–27). Such toolkits and environments provide powerful ready-to-use controls and functions that enrich web applications.

Previous chapters discussed XHTML, CSS, JavaScript, dynamic HTML, the DOM and XML. This chapter uses these technologies to build Ajax-enabled web applications. The client-side of Ajax applications is written in XHTML and CSS, and uses JavaScript to add functionality to the user interface. XML is used to structure the data passed between the server and the client. We’ll also use JSON (JavaScript Object Notation) for this purpose. The Ajax component that manages interaction with the server is usually imple-

mented with JavaScript's **XMLHttpRequest object**—commonly abbreviated as XHR. The server processing can be implemented using any server-side technology, such as PHP, ASP.NET, JavaServer Faces and Ruby on Rails—each of which we cover in later chapters.

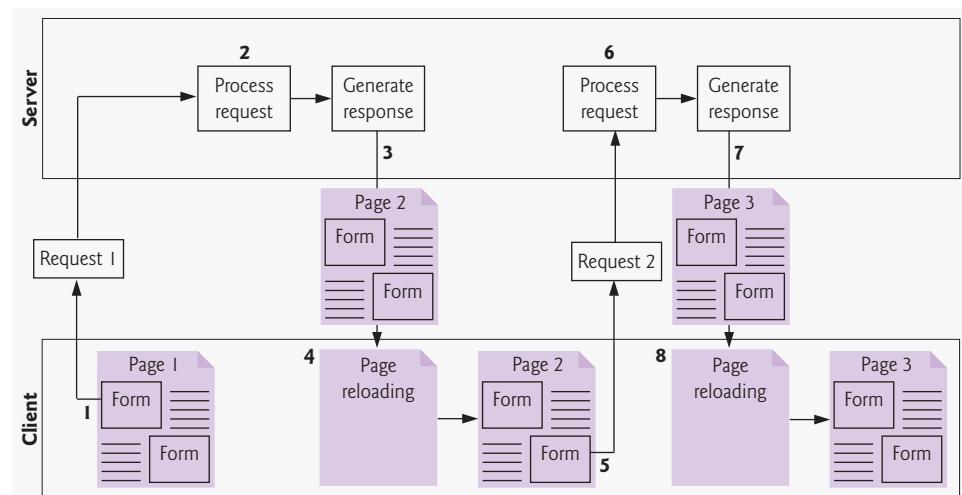
This chapter begins with several examples that build basic Ajax applications using JavaScript and the XMLHttpRequest object. We then build an Ajax application with a rich calendar GUI using the Dojo Ajax toolkit. In subsequent chapters, we use tools such as Adobe Flex, Microsoft Silverlight and JavaServer Faces to build RIAs using Ajax. In Chapter 24, we'll demonstrate features of the Prototype and Script.aculo.us Ajax libraries, which come with the Ruby on Rails framework (and can be downloaded separately). Prototype provides capabilities similar to Dojo. Script.aculo.us provides many “eye candy” effects that enable you to beautify your Ajax applications and create rich interfaces. In Chapter 27, we present Ajax-enabled JavaServer Faces (JSF) components. JSF uses Dojo to implement many of its client-side Ajax capabilities.

## 15.2 Traditional Web Applications vs. Ajax Applications

In this section, we consider the key differences between traditional web applications and Ajax-based web applications.

### *Traditional Web Applications*

Figure 15.1 presents the typical interactions between the client and the server in a traditional web application, such as one that uses a user registration form. First, the user fills in the form’s fields, then submits the form (Fig. 15.1, Step 1). The browser generates a request to the server, which receives the request and processes it (Step 2). The server generates and sends a response containing the exact page that the browser will render (Step 3), which causes the browser to load the new page (Step 4) and temporarily makes the browser window blank. Note that the client *waits* for the server to respond and  *reloads the entire page* with the data from the response (Step 4). While such a **synchronous request** is being pro-



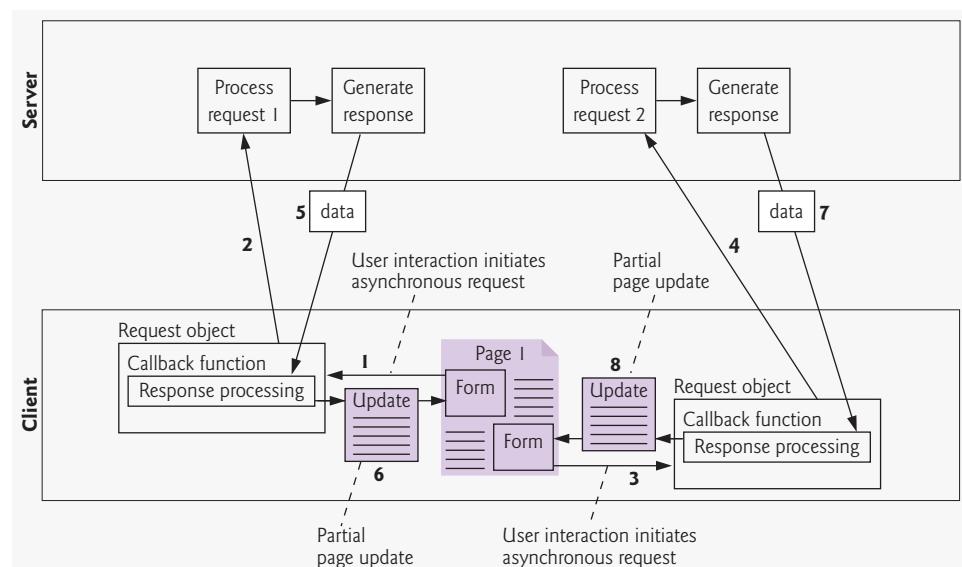
**Fig. 15.1** | Classic web application reloading the page for every user interaction.

cessed on the server, the user cannot interact with the client web page. Frequent long periods of waiting, due perhaps to Internet congestion, have led some users to refer to the World Wide Web as the “World Wide Wait.” If the user interacts with and submits another form, the process begins again (*Steps 5–8*).

This model was originally designed for a web of hypertext documents—what some people call the “brochure web.” As the web evolved into a full-scale applications platform, the model shown in Fig. 15.1 yielded “choppy” application performance. Every full-page refresh required users to re-establish their understanding of the full-page contents. Users began to demand a model that would yield the responsive feel of desktop applications.

### Ajax Web Applications

Ajax applications add a layer between the client and the server to manage communication between the two (Fig. 15.2). When the user interacts with the page, the client creates an XMLHttpRequest object to manage a request (*Step 1*). The XMLHttpRequest object sends the request to the server (*Step 2*) and awaits the response. The requests are **asynchronous**, so the user can continue interacting with the application on the client-side while the server processes the earlier request concurrently. Other user interactions could result in additional requests to the server (*Steps 3 and 4*). Once the server responds to the original request (*Step 5*), the XMLHttpRequest object that issued the request calls a client-side function to process the data returned by the server. This function—known as a **callback function**—uses **partial page updates** (*Step 6*) to display the data in the existing web page *without reloading the entire page*. At the same time, the server may be responding to the second request (*Step 7*) and the client-side may be starting to do another partial page update (*Step 8*). The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications. The web application does not load a new page while the user interacts with it.



**Fig. 15.2** | Ajax-enabled web application interacting with the server asynchronously.

### 15.3 Rich Internet Applications (RIAs) with Ajax

Ajax improves the user experience by making interactive web applications more responsive. Consider a registration form with a number of fields (e.g., first name, last name e-mail address, telephone number, etc.) and a **Register** (or **Submit**) button that sends the entered data to the server. Usually each field has rules that the user's entries have to follow (e.g., valid e-mail address, valid telephone number, etc.).

When the user clicks **Register**, a classic XHTML form sends the server all of the data to be validated (Fig. 15.3). While the server is validating the data, the user cannot interact with the page. The server finds invalid data, generates a new page identifying the errors in the form and sends it back to the client—which renders the page in the browser. Once the user fixes the errors and clicks the **Register** button, the cycle repeats until no errors are found, then the data is stored on the server. The entire page reloads every time the user submits invalid data.

Ajax-enabled forms are more interactive. Rather than sending the entire form to be validated, entries are validated dynamically as the user enters data into the fields. For example, consider a website registration form that requires a unique e-mail address. When the user enters an e-mail address into the appropriate field, then moves to the next form field to continue entering data, an asynchronous request is sent to the server to validate the e-mail address. If the e-mail address is not unique, the server sends an error message that is displayed on the page informing the user of the problem (Fig. 15.4). By sending each entry asynchronously, the user can address each invalid entry quickly, versus making edits and resubmitting the entire form repeatedly until all entries are valid. Asynchronous

- a) A sample registration form in which the user has not filled in the required fields, but attempts to submit the form anyway by clicking **Register**.

**Fig. 15.3** | Classic XHTML form: User submits entire form to server, which validates the data entered (if any). Server responds indicating fields with invalid or missing data. (Part 1 of 2.)

b) The server responds by indicating all the form fields with missing or invalid data. The user must correct the problems and resubmit the entire form repeatedly until all errors are corrected.

**Fig. 15.3** | Classic XHTML form: User submits entire form to server, which validates the data entered (if any). Server responds indicating fields with invalid or missing data. (Part 2 of 2.)

**Fig. 15.4** | Ajax-enabled form shows errors asynchronously when user moves to another field.

requests could also be used to fill some fields based on previous fields (e.g., automatically filling in the “city” and “state” fields based on the zip code entered by the user).

## 15.4 History of Ajax

The term Ajax was coined by Jesse James Garrett of Adaptive Path in February 2005, when he was presenting the previously unnamed technology to a client. The technologies of Ajax (XHTML, JavaScript, CSS, the DOM and XML) have all existed for many years.

Asynchronous page updates can be traced back to earlier browsers. In the 1990s, Netscape’s LiveScript made it possible to include scripts in web pages (e.g., web forms) that could run on the client. LiveScript evolved into JavaScript. In 1998, Microsoft introduced the XMLHttpRequest object to create and manage asynchronous requests and responses. Popular applications like Flickr and Google’s Gmail use the XMLHttpRequest object to update pages dynamically. For example, Flickr uses the technology for its text editing, tagging and organizational features; Gmail continuously checks the server for new e-mail; and Google Maps allows you to drag a map in any direction, downloading the new areas on the map without reloading the entire page.

The name Ajax immediately caught on and brought attention to its component technologies. Ajax has become one of the hottest web-development technologies, enabling webtop applications to challenge the dominance of established desktop applications.

## 15.5 “Raw” Ajax Example Using the XMLHttpRequest Object

In this section, we use the XMLHttpRequest object to create and manage asynchronous requests. The XMLHttpRequest object (which resides on the client) is the layer between the client and the server that manages asynchronous requests in Ajax applications. This object is supported on most browsers, though they may implement it differently—a common issue in JavaScript programming. To initiate an asynchronous request (shown in Fig. 15.5), you create an instance of the XMLHttpRequest object, then use its open method to set up the request and its send method to initiate the request. We summarize the XMLHttpRequest properties and methods in Figs. 15.6–15.7.

Figure 15.5 presents an Ajax application in which the user interacts with the page by moving the mouse over book-cover images. We use the onmouseover and onmouseout events (discussed in Chapter 13) to trigger events when the user moves the mouse over and out of an image, respectively. The onmouseover event calls function `getContent` with the URL of the document containing the book’s description. The function makes this request asynchronously using an XMLHttpRequest object. When the XMLHttpRequest object receives the response, the book description is displayed below the book images. When the user moves the mouse out of the image, the onmouseout event calls function `clearContent` to clear the display box. These tasks are accomplished without reloading the page on the client. You can test-drive this example at [test.deitel.com/examples/iw3htp4/ajax/fig15\\_05/SwitchContent.html](http://test.deitel.com/examples/iw3htp4/ajax/fig15_05/SwitchContent.html).

### Performance Tip 15.1



When an Ajax application requests a file from a server, such as an XHTML document or an image, the browser typically caches that file. Subsequent requests for the same file can load it from the browser’s cache rather than making the round trip to the server again.



## Software Engineering Observation 15.1

For security purposes, the XMLHttpRequest object doesn't allow a web application to request resources from domain names other than the one that served the application. For this reason, the web application and its resources must reside on the same web server (this could be a web server on your local computer). This is commonly known as the *same origin policy (SOP)*. SOP aims to close a vulnerability called *cross-site scripting*, also known as *XSS*, which allows an attacker to compromise a website's security by injecting a malicious script onto the page from another domain. To learn more about XSS visit [en.wikipedia.org/wiki/XSS](https://en.wikipedia.org/wiki/XSS). To get content from another domain securely, you can implement a server-side proxy—an application on the web application's web server—that can make requests to other servers on the web application's behalf.

### Asynchronous Requests

The function `getContent` (lines 19–35) sends the asynchronous request. Line 24 creates the XMLHttpRequest object, which manages the asynchronous request. We store the object in the global variable `asyncRequest` (declared at line 16) so that it can be accessed anywhere in the script.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 15.5: SwitchContent.html -->
6 <!-- Asynchronously display content without reloading the page. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <style type="text/css">
10 .box { border: 1px solid black;
11 padding: 10px }
12 </style>
13 <title>Switch Content Asynchronously</title>
14 <script type = "text/javascript" language = "JavaScript">
15 <!--
16 var asyncRequest; // variable to hold XMLHttpRequest object
17
18 // set up and send the asynchronous request
19 function getContent(url)
20 {
21 // attempt to create the XMLHttpRequest and make the request
22 try
23 {
24 asyncRequest = new XMLHttpRequest(); // create request object
25
26 // register event handler
27 asyncRequest.onreadystatechange = stateChange;
28 asyncRequest.open('GET', url, true); // prepare the request
29 asyncRequest.send(null); // send the request
30 } // end try

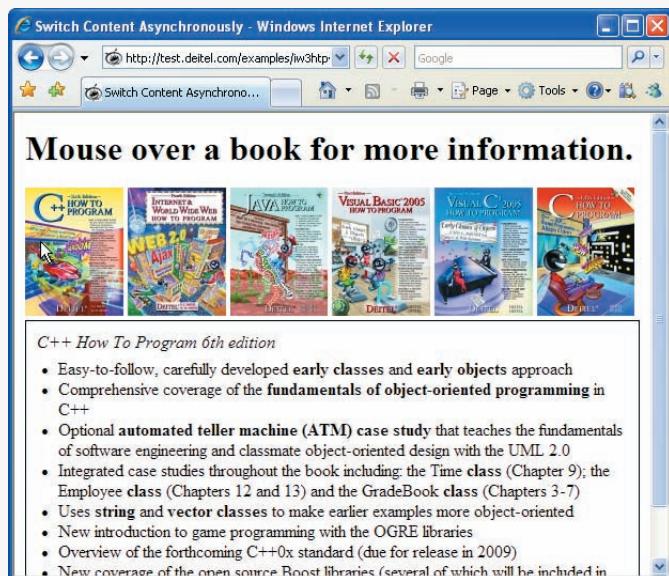
```

**Fig. 15.5** | Asynchronously display content without reloading the page. (Part 1 of 3.)

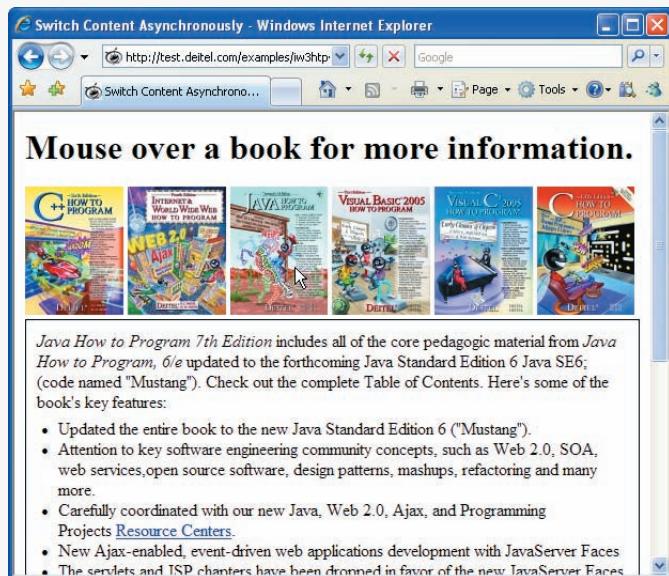
```
31 catch (exception)
32 {
33 alert('Request failed.');
34 } // end catch
35 } // end function getContent
36
37 // displays the response data on the page
38 function stateChange()
39 {
40 if (asyncRequest.readyState == 4 && asyncRequest.status == 200)
41 {
42 document.getElementById('contentArea').innerHTML =
43 asyncRequest.responseText; // places text in contentArea
44 } // end if
45 } // end function stateChange
46
47 // clear the content of the box
48 function clearContent()
49 {
50 document.getElementById('contentArea').innerHTML = '';
51 } // end function clearContent
52 // -->
53 </script>
54 </head>
55 <body>
56 <h1>Mouse over a book for more information.</h1>
57 <img src =
58 "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/cpphttp6.jpg"
59 onmouseover = 'getContent("cpphttp6.html")'
60 onmouseout = 'clearContent()' />
61 <img src =
62 "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/iw3htp4.jpg"
63 onmouseover = 'getContent("iw3htp4.html")'
64 onmouseout = 'clearContent()' />
65 <img src =
66 "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/jhttp7.jpg"
67 onmouseover = 'getContent("jhttp7.html")'
68 onmouseout = 'clearContent()' />
69 <img src =
70 "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/vbhttp3.jpg"
71 onmouseover = 'getContent("vbhttp3.html")'
72 onmouseout = 'clearContent()' />
73 <img src =
74 "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/vcsharphtp2.jpg"
75 onmouseover = 'getContent("vcsharphtp2.html")'
76 onmouseout = 'clearContent()' />
77 <img src =
78 "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/chtp5.jpg"
79 onmouseover = 'getContent("chtp5.html")'
80 onmouseout = 'clearContent()' />
81 <div class = "box" id = "contentArea"> </div>
82 </body>
83 </html>
```

Fig. 15.5 | Asynchronously display content without reloading the page. (Part 2 of 3.)

- a) User hovers over *C++ How to Program* book cover image, causing an asynchronous request to the server to obtain the book's description. When the response is received, the application performs a partial page update to display the description.



- b) User hovers over *Java How to Program* book cover image, causing the process to repeat.



**Fig. 15.5** | Asynchronously display content without reloading the page. (Part 3 of 3.)

Line 28 calls the XMLHttpRequest open method to prepare an asynchronous GET request. In this example, the url parameter specifies the address of an HTML document containing the description of a particular book. When the third argument is true, the

request is asynchronous. The URL is passed to function `getContent` in response to the `onmouseover` event for each image. Line 29 sends the asynchronous request to the server by calling `XMLHttpRequest` `send` method. The argument `null` indicates that this request is not submitting data in the body of the request.

### *Exception Handling*

Lines 22–34 introduce **exception handling**. An **exception** is an indication of a problem that occurs during a program’s execution. The name “exception” implies that the problem occurs infrequently—if the “rule” is that a statement normally executes correctly, then the “exception to the rule” is that a problem occurs. Exception handling enables you to create applications that can resolve (or handle) exceptions—in some cases allowing a program to continue executing as if no problem had been encountered.

Lines 22–30 contain a **try block**, which encloses the code that might cause an exception and the code that should not execute if an exception occurs (i.e., if an exception occurs in a statement of the `try` block, the remaining code in the `try` block is skipped). A `try` block consists of the keyword `try` followed by a block of code enclosed in curly braces (`{}`). If there is a problem sending the request—e.g., if a user tries to access the page using an older browser that does not support `XMLHttpRequest`—the `try` block terminates immediately and a **catch block** (also called a **catch clause** or **exception handler**) catches (i.e., receives) and handles an exception. The `catch` block (lines 31–34) begins with the keyword `catch` and is followed by a parameter in parentheses (called the exception parameter) and a block of code enclosed in curly braces. The exception parameter’s name (`exception` in this example) enables the `catch` block to interact with a caught exception object (for example, to obtain the name of the exception or an exception-specific error message via the exception object’s `name` and `message` properties). In this case, we simply display our own error message ‘Request Failed’ and terminate the `getContent` function. The request can fail because a user accesses the web page with an older browser or the content that is being requested is located on a different domain.

### *Callback Functions*

The `stateChange` function (lines 38–45) is the callback function that is called when the client receives the response data. Line 27 registers function `stateChange` as the event handler for the `XMLHttpRequest` object’s `onreadystatechange` event. Whenever the request makes progress, the `XMLHttpRequest` calls the `onreadystatechange` event handler. This progress is monitored by the `readyState` property, which has a value from 0 to 4. The value 0 indicates that the request is not initialized and the value 4 indicates that the request is complete—all the values for this property are summarized in Fig. 15.6. If the request completes successfully (line 40), lines 42–43 use the `XMLHttpRequest` object’s `responseText` property to obtain the response data and place it in the `div` element named `contentArea` (defined at line 81). We use the DOM’s `getById` method to get this `div` element, and use the element’s `innerHTML` property to place the content in the `div`.

### *XMLHttpRequest Object Properties and Methods*

Figures 15.6 and 15.7 summarize some of the `XMLHttpRequest` object’s properties and methods, respectively. The properties are crucial to interacting with asynchronous requests. The methods initialize, configure and send asynchronous requests.

| Property                        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>onreadystatechange</code> | Stores the callback function—the event handler that gets called when the server responds.                                                                                                                                                                                                                                                                                                                                                                |
| <code>readyState</code>         | Keeps track of the request’s progress. It is usually used in the callback function to determine when the code that processes the response should be launched. The <code>readyState</code> value 0 signifies that the request is uninitialized; 1 signifies that the request is loading; 2 signifies that the request has been loaded; 3 signifies that data is actively being sent from the server; and 4 signifies that the request has been completed. |
| <code>responseText</code>       | Text that is returned to the client by the server.                                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>responseXML</code>        | If the server’s response is in XML format, this property contains the XML document; otherwise, it is empty. It can be used like a <code>document</code> object in JavaScript, which makes it useful for receiving complex data (e.g. populating a table).                                                                                                                                                                                                |
| <code>status</code>             | HTTP status code of the request. A <code>status</code> of 200 means that request was successful. A <code>status</code> of 404 means that the requested resource was not found. A <code>status</code> of 500 denotes that there was an error while the server was processing the request.                                                                                                                                                                 |
| <code>statusText</code>         | Additional information on the request’s status. It is often used to display the error to the user when the request fails.                                                                                                                                                                                                                                                                                                                                |

**Fig. 15.6** | XMLHttpRequest object properties.

| Method                        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>open</code>             | Initializes the request and has two mandatory parameters—method and URL. The method parameter specifies the purpose of the request—typically <code>GET</code> if the request is to take data from the server or <code>POST</code> if the request will contain a body in addition to the headers. The URL parameter specifies the address of the file on the server that will generate the response. A third optional boolean parameter specifies whether the request is asynchronous—it’s set to <code>true</code> by default. |
| <code>send</code>             | Sends the request to the sever. It has one optional parameter, <code>data</code> , which specifies the data to be POSTed to the server—it’s set to <code>null</code> by default.                                                                                                                                                                                                                                                                                                                                               |
| <code>setRequestHeader</code> | Alters the header of the request. The two parameters specify the header and its new value. It is often used to set the <code>content-type</code> field.                                                                                                                                                                                                                                                                                                                                                                        |

**Fig. 15.7** | XMLHttpRequest object methods. (Part 1 of 2.)

| Method                | Description                                                                                                                                                                                                     |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| getResponseHeader     | Returns the header data that precedes the response body. It takes one parameter, the name of the header to retrieve. This call is often used to determine the response's type, to parse the response correctly. |
| getAllResponseHeaders | Returns an array that contains all the headers that precede the response body.                                                                                                                                  |
| abort                 | Cancels the current request.                                                                                                                                                                                    |

**Fig. 15.7** | XMLHttpRequest object methods. (Part 2 of 2.)

## 15.6 Using XML and the DOM

When passing structured data between the server and the client, Ajax applications often use XML because it is easy to generate and parse. When the XMLHttpRequest object receives XML data, it parses and stores the data as an XML DOM object in the responseXML property. The example in Fig. 15.8 asynchronously requests from a server XML documents containing URLs of book-cover images, then displays the images in an HTML table. The code that configures the asynchronous request is the same as in Fig. 15.5. You can test-drive this application at [test.deitel.com/examples/iw3htp4/ajax/fig15\\_08/PullImagesOntoPage.html](http://test.deitel.com/examples/iw3htp4/ajax/fig15_08/PullImagesOntoPage.html) (the book-cover images will be easier to see on the screen).

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 15.8: PullImagesOntoPage.html -->
6 <!-- Image catalog that uses Ajax to request XML data asynchronously. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title> Pulling Images onto the Page </title>
10 <style type = "text/css">
11 td { padding: 4px }
12 img { border: 1px solid black }
13 </style>
14 <script type = "text/javascript" language = "Javascript">
15 var asyncRequest; // variable to hold XMLHttpRequest object
16
17 // set up and send the asynchronous request to the XML file
18 function getImages(url)
19 {
20 // attempt to create the XMLHttpRequest and make the request
21 try
22 {
23 asyncRequest = new XMLHttpRequest(); // create request object
24

```

**Fig. 15.8** | Image catalog that uses Ajax to request XML data asynchronously. (Part 1 of 4.)

```

25 // register event handler
26 asyncRequest.onreadystatechange = processResponse;
27 asyncRequest.open('GET', url, true); // prepare the request
28 asyncRequest.send(null); // send the request
29 } // end try
30 catch (exception)
31 {
32 alert('Request Failed');
33 } // end catch
34 } // end function getImages
35
36 // parses the XML response; dynamically creates a table using DOM and
37 // populates it with the response data; displays the table on the page
38 function processResponse()
39 {
40 // if request completed successfully and responseXML is non-null
41 if (asyncRequest.readyState == 4 && asyncRequest.status == 200 &&
42 asyncRequest.responseXML)
43 {
44 clearTable(); // prepare to display a new set of images
45
46 // get the covers from the responseXML
47 var covers = asyncRequest.responseXML.getElementsByTagName(
48 "cover")
49
50 // get base URL for the images
51 var baseUrl = asyncRequest.responseXML.getElementsByTagName(
52 "baseurl").item(0).firstChild.nodeValue;
53
54 // get the placeholder div element named covers
55 var output = document.getElementById("covers");
56
57 // create a table to display the images
58 var imageTable = document.createElement('table');
59
60 // create the table's body
61 var tableBody = document.createElement('tbody');
62
63 var rowCount = 0; // tracks number of images in current row
64 var imageRow = document.createElement("tr"); // create row
65
66 // place images in row
67 for (var i = 0; i < covers.length; i++)
68 {
69 var cover = covers.item(i); // get a cover from covers array
70
71 // get the image filename
72 var image = cover.getElementsByTagName("image").
73 item(0).firstChild.nodeValue;
74
75 // create table cell and img element to display the image
76 var imageCell = document.createElement("td");

```

**Fig. 15.8** | Image catalog that uses Ajax to request XML data asynchronously. (Part 2 of 4.)

```

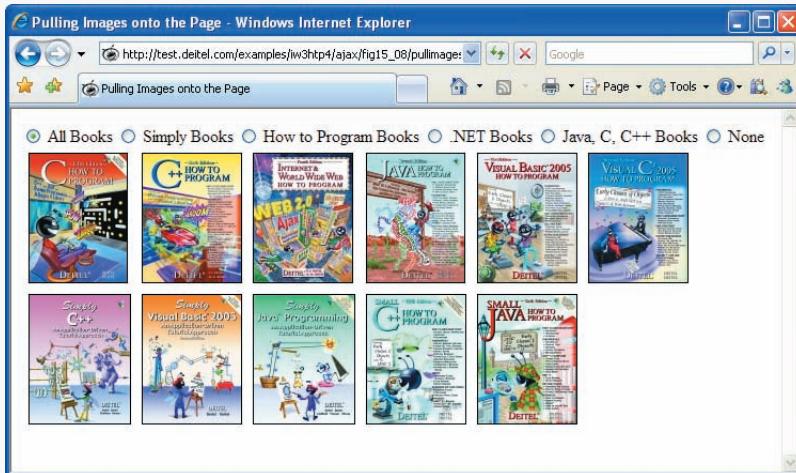
77 var imageTag = document.createElement("img");
78
79 // set img element's src attribute
80 imageTag.setAttribute("src", baseUrl + escape(image));
81 imageCell.appendChild(imageTag); // place img in cell
82 imageRow.appendChild(imageCell); // place cell in row
83 rowCount++; // increment number of images in row
84
85 // if there are 6 images in the row, append the row to
86 // table and start a new row
87 if (rowCount == 6 && i + 1 < covers.length)
88 {
89 tableBody.appendChild(imageRow);
90 imageRow = document.createElement("tr");
91 rowCount = 0;
92 } // end if statement
93 } // end for statement
94
95 tableBody.appendChild(imageRow); // append row to table body
96 imageTable.appendChild(tableBody); // append body to table
97 output.appendChild(imageTable); // append table to covers div
98 } // end if
99 } // end function processResponse
100
101 // deletes the data in the table.
102 function clearTable()
103 {
104 document.getElementById("covers").innerHTML = '';
105 } // end function clearTable
106 </script>
107 </head>
108 <body>
109 <input type = "radio" checked = "unchecked" name ="Books" value = "all"
110 onclick = 'getImages("all.xml")'> All Books
111 <input type = "radio" checked = "unchecked"
112 name = "Books" value = "simply"
113 onclick = 'getImages("simply.xml")'> Simply Books
114 <input type = "radio" checked = "unchecked"
115 name = "Books" value = "howto"
116 onclick = 'getImages("howto.xml")'> How to Program Books
117 <input type = "radio" checked = "unchecked"
118 name = "Books" value = "dotnet"
119 onclick = 'getImages("dotnet.xml")'> .NET Books
120 <input type = "radio" checked = "unchecked"
121 name = "Books" value = "javaccpp"
122 onclick = 'getImages("javaccpp.xml")'> Java, C, C++ Books
123 <input type = "radio" checked = "checked" name = "Books" value = "none"
124 onclick = 'clearTable()'> None
125

126 <div id = "covers"></div>
127 </body>
128 </html>

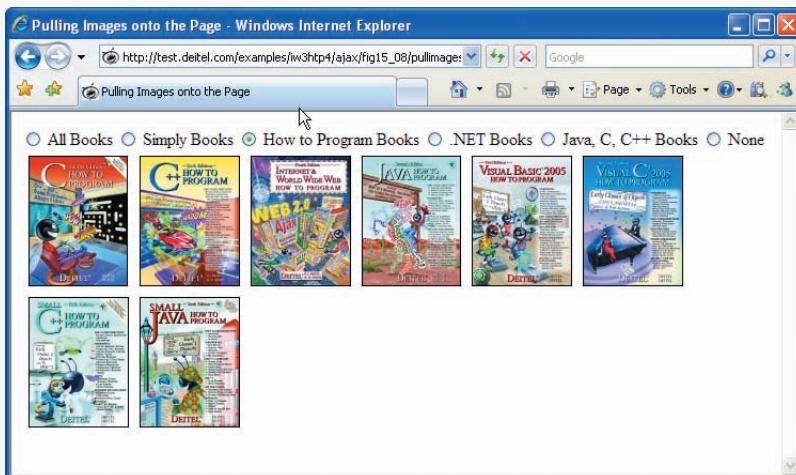
```

**Fig. 15.8** | Image catalog that uses Ajax to request XML data asynchronously. (Part 3 of 4.)

- a) User clicks the **All Books** radio button to display all the book covers. The application sends an asynchronous request to the server to obtain an XML document containing the list of book-cover filenames. When the response is received, the application performs a partial page update to display the set of book covers.



- b) User clicks the **How to Program Books** radio button to select a subset of book covers to display. Application sends an asynchronous request to the server to obtain an XML document containing the appropriate subset of book-cover filenames. When the response is received, the application performs a partial page update to display the subset of book covers.



**Fig. 15.8** | Image catalog that uses Ajax to request XML data asynchronously. (Part 4 of 4.)

When the XMLHttpRequest object receives the response, it invokes the callback function `processResponse` (lines 38–99). We use XMLHttpRequest object's `responseXML` property to access the XML returned by the server. Lines 41–42 check that the request was successful, and that the `responseXML` property is not empty. The XML file that we requested includes a `baseURL` node that contains the address of the image directory and a collection of `cover` nodes that contain image filenames. `responseXML` is a document

object, so we can extract data from it using the XML DOM functions. Lines 47–52 use the DOM's method `getElementsByName` to extract all the image filenames from cover nodes and the URL of the directory from the `baseURL` node. Since the `baseURL` has no child nodes, we use `item(0).firstChild.nodeValue` to obtain the directory's address and store it in variable `baseURL`. The image filenames are stored in the `covers` array.

As in Fig. 15.5 we have a placeholder `div` element (line 126) to specify where the image table will be displayed on the page. Line 55 stores the `div` in variable `output`, so we can fill it with content later in the program.

Lines 58–93 generate an XHTML table dynamically, using the `createElement`, `setAttribute` and `appendChild` DOM methods. Method `createElement` creates an XHTML element of the specified type. Method `setAttribute` adds or changes an attribute of an XHTML element. Method `appendChild` inserts one XHTML element into another. Lines 58 and 61 create the `table` and `tbody` elements, respectively. We restrict each row to no more than six images, which we track with variable `rowCount` variable. Each iteration of the `for` statement (lines 67–93) obtains the filename of the image to be inserted (lines 69–73), creates a table cell element where the image will be inserted (line 76) and creates an `<img>` element (line 77). Line 80 sets the image's `src` attribute to the image's URL, which we build by concatenating the filename to the base URL of the XHTML document. Lines 81–82 insert the `<img>` element into the cell and the cell into the table row. When the row has six cells, it is inserted into the table and a new row is created (lines 87–92). Once all the rows have been inserted into the table, the table is inserted into the placeholder element `covers` that is referenced by variable `output` (line 97). This element is located on the bottom of the web page.

Function `clearTable` (lines 102–105) is called to clear images when the user switches radio buttons. The text is cleared by setting the `innerHTML` property of the placeholder element to the empty string.

## 15.7 Creating a Full-Scale Ajax-Enabled Application

Our next example demonstrates additional Ajax capabilities. The web application interacts with a web service to obtain data and to modify data in a server-side database. The web application and server communicate with a data format called JSON (JavaScript Object Notation). In addition, the application demonstrates server-side validation that occurs in parallel with the user interacting with the web application. You can test the application at [test.deitel.com/examples/iw3htp4/ajax/fig15\\_09\\_10/AddressBook.html](http://test.deitel.com/examples/iw3htp4/ajax/fig15_09_10/AddressBook.html).

### *Using JSON*

**JSON (JavaScript Object Notation)**—a simple way to represent JavaScript objects as strings—is an alternative way (to XML) for passing data between the client and the server. Each object in JSON 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:

```
[value1, value2, value3]
```

Each value can be a string, a number, a JSON representation of an object, `true`, `false` or `null`. You can convert JSON strings into JavaScript objects with JavaScript's `eval` func-

tion. To evaluate a JSON string properly, a left parenthesis should be placed at the beginning of the string and a right parenthesis at the end of the string before the string is passed to the eval function.

The eval function creates a potential security risk—it executes any embedded JavaScript code in its string argument, possibly allowing a harmful script to be injected into JSON. A more secure way to process JSON is to use a JSON parser. In our examples, we use the open source parser from [www.json.org/js.html](http://www.json.org/js.html). When you download its JavaScript file, place it in the same folder as your application. Then, link the json.js file into your XHTML file with the following statement in the head section:

```
<script type = "text/javascript" src = "json.js">
```

You can now call function parseJSON on a JSON string to convert it to a JavaScript object.

JSON strings are easier to create and parse than XML, and require fewer bytes. For these reasons, JSON is commonly used to communicate in client/server interaction. For more information on JSON, visit our JSON Resource Center at [www.deitel.com/json](http://www.deitel.com/json).

### *Rich Functionality*

The previous examples in this chapter requested data from static files on the server. The example in Fig. 15.9 is an address-book application that communicates with a server-side application. The application uses server-side processing to give the page the functionality and usability of a desktop application. We use JSON to encode server-side responses and to create objects on the fly.

Initially the address book loads a list of entries, each containing a first and last name (Fig. 15.9(a)). Each time the user clicks a name, the address book uses Ajax functionality to load the person's address from the server and expand the entry *without reloading the page* (Fig. 15.9(b))—and it does this *in parallel* with allowing the user to click other names. The application allows the user to search the address book by typing a last name. As the user enters each keystroke, the application asynchronously displays the list of names in which the last name starts with the characters the user has entered so far (Fig. 15.9(c), Fig. 15.9(d) and Fig. 15.9(e))—a popular feature called **type ahead**.

```
1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 15.9 addressbook.html -->
6 <!-- Ajax enabled address book application. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <title>Address Book</title>
10 <link rel = "stylesheet" type = "text/css" href = "address.css" />
11 <script type = "text/javascript" src = "json.js"></script>
12 <script type = "text/javascript">
13 <!--
14 // URL of the web service
15 var webServiceUrl = '/AddressBookWebService/AddressService.asmx' ;
```

**Fig. 15.9** | Ajax-enabled address-book application. (Part 1 of 10.)

```

16
17 var phoneValid = false; // indicates if the telephone is valid
18 var zipValid = false; //indicates if the zip code is valid
19
20 // get a list of names from the server and display them
21 function showAddressBook()
22 {
23 // hide the "addEntry" form and show the address book
24 document.getElementById('addEntry').style.display = 'none';
25 document.getElementById('addressBook').style.display = 'block';
26
27 var params = "[]"; // create an empty object
28 callWebService('getAllNames', params, parseData);
29 } // end function showAddressBook
30
31 // send the asynchronous request to the web service
32 function callWebService(method, paramString, callBack)
33 {
34 // build request URL string
35 var requestUrl = webServiceUrl + "/" + method;
36 var params = paramString.parseJSON();
37
38 // build the parameter string to add to the url
39 for (var i = 0; i < params.length; i++)
40 {
41 // checks whether it is the first parameter and builds
42 // the parameter string accordingly
43 if (i == 0)
44 requestUrl = requestUrl + "?" + params[i].param +
45 "=" + params[i].value; // add first parameter to url
46 else
47 requestUrl = requestUrl + "&" + params[i].param +
48 "=" + params[i].value; // add other parameters to url
49 } // end for
50
51 // attempt to send the asynchronous request
52 try
53 {
54 var asyncRequest = new XMLHttpRequest(); // create request
55
56 // set up callback function and store it
57 asyncRequest.onreadystatechange = function()
58 {
59 callBack(asyncRequest);
60 }; // end anonymous function
61
62 // send the asynchronous request
63 asyncRequest.open('GET', requestUrl, true);
64 asyncRequest.setRequestHeader("Accept",
65 "application/json; charset=utf-8");
66 asyncRequest.send(); // send request
67 } // end try

```

Fig. 15.9 | Ajax-enabled address-book application. (Part 2 of 10.)

```

68 catch (exception)
69 {
70 alert ('Request Failed');
71 } // end catch
72 } // end function callWebService
73
74 // parse JSON data and display it on the page
75 function parseData(asyncRequest)
76 {
77 // if request has completed successfully process the response
78 if (asyncRequest.readyState == 4 && asyncRequest.status == 200)
79 {
80 // convert the JSON string to an Object
81 var data = asyncRequest.responseText.parseJSON();
82 displayNames(data); // display data on the page
83 } // end if
84 } // end function parseData
85
86 // use the DOM to display the retrieved address book entries
87 function displayNames(data)
88 {
89 // get the placeholder element from the page
90 var listBox = document.getElementById('Names');
91 listBox.innerHTML = ''; // clear the names on the page
92
93 // iterate over retrieved entries and display them on the page
94 for (var i = 0; i < data.length; i++)
95 {
96 // dynamically create a div element for each entry
97 // and a fieldset element to place it in
98 var entry = document.createElement('div');
99 var field = document.createElement('fieldset');
100 entry.onclick = handleOnClick; // set onclick event handler
101 entry.id = i; // set the id
102 entry.innerHTML = data[i].First + ' ' + data[i].Last;
103 field.appendChild(entry); // insert entry into the field
104 listBox.appendChild(field); // display the field
105 } // end for
106 } // end function displayAll
107
108 // event handler for entry's onclick event
109 function handleOnClick()
110 {
111 // call getAddress with the element's content as a parameter
112 getAddress(eval('this'), eval('this.innerHTML'));
113 } // end function handleOnClick
114
115 // search the address book for input
116 // and display the results on the page
117 function search(input)
118 {
119 // get the placeholder element and delete its content
120 var listBox = document.getElementById('Names');

```

**Fig. 15.9** | Ajax-enabled address-book application. (Part 3 of 10.)

```

I21 listBox.innerHTML = ''; // clear the display box
I22
I23 // if no search string is specified all the names are displayed
I24 if (input == "") // if no search value specified
I25 {
I26 showAddressBook(); // Load the entire address book
I27 } // end if
I28 else
I29 {
I30 var params = '[{"param": "input", "value": "' + input + '"}]';
I31 callWebService("search", params, parseData);
I32 } // end else
I33 } // end function search
I34
I35 // Get address data for a specific entry
I36 function getAddress(entry, name)
I37 {
I38 // find the address in the JSON data using the element's id
I39 // and display it on the page
I40 var firstLast = name.split(" "); // convert string to array
I41 var requestUrl = webServiceUrl + "/getAddress?first="
I42 + firstLast[0] + "&last=" + firstLast[1];
I43
I44 // attempt to send an asynchronous request
I45 try
I46 {
I47 // create request object
I48 var asyncRequest = new XMLHttpRequest();
I49
I50 // create a callback function with 2 parameters
I51 asyncRequest.onreadystatechange = function()
I52 {
I53 displayAddress(entry, asyncRequest);
I54 }; // end anonymous function
I55
I56 asyncRequest.open('GET', requestUrl, true);
I57 asyncRequest.setRequestHeader("Accept",
I58 "application/json; charset=utf-8"); // set response datatype
I59 asyncRequest.send(); // send request
I60 } // end try
I61 catch (exception)
I62 {
I63 alert ('Request Failed.');
I64 } // end catch
I65 } // end function getAddress
I66
I67 // clear the entry's data.
I68 function displayAddress(entry, asyncRequest)
I69 {
I70 // if request has completed successfully, process the response
I71 if (asyncRequest.readyState == 4 && asyncRequest.status == 200)
I72 {

```

Fig. 15.9 | Ajax-enabled address-book application. (Part 4 of 10.)

```
173 // convert the JSON string to an object
174 var data = asyncRequest.responseText.parseJSON();
175 var name = entry.innerHTML // save the name string
176 entry.innerHTML = name + '
' + data.Street +
177 '
' + data.City + ', ' + data.State
178 + ', ' + data.Zip + '
' + data.Telephone;
179
180 // clicking on the entry removes the address
181 entry.onclick = function()
182 {
183 clearField(entry, name);
184 }; // end anonymous function
185
186 } // end if
187 } // end function displayAddress
188
189 // clear the entry's data
190 function clearField(entry, name)
191 {
192 entry.innerHTML = name; // set the entry to display only the name
193 entry.onclick = function() // set onclick event
194 {
195 getAddress(entry, name); // retrieve address and display it
196 }; // end function
197 } // end function clearField
198
199 // display the form that allows the user to enter more data
200 function addEntry()
201 {
202 document.getElementById('addressBook').style.display = 'none';
203 document.getElementById('addEntry').style.display = 'block';
204 } // end function addEntry
205
206 // send the zip code to be validated and to generate city and state
207 function validateZip(zip)
208 {
209 // build parameter array
210 var params = '[{"param": "zip", "value": "' + zip + '"}]';
211 callWebService("validateZip", params, showCityState);
212 } // end function validateZip
213
214 // get city and state that were generated using the zip code
215 // and display them on the page
216 function showCityState(asyncRequest)
217 {
218 // display message while request is being processed
219 document.getElementById('validateZip').
220 innerHTML = "Checking zip...";
221
222 // if request has completed successfully, process the response
223 if (asyncRequest.readyState == 4)
224 {
```

**Fig. 15.9** | Ajax-enabled address-book application. (Part 5 of 10.)

```

225 if (asyncRequest.status == 200)
226 {
227 // convert the JSON string to an object
228 var data = asyncRequest.responseText.parseJSON();
229
230 // update zip code validity tracker and show city and state
231 if (data.Validity == 'Valid')
232 {
233 zipValid = true; // update validity tracker
234
235 // display city and state
236 document.getElementById('validateZip').innerHTML = '';
237 document.getElementById('city').innerHTML = data.City;
238 document.getElementById('state').
239 innerHTML = data.State;
240 } // end if
241 else
242 {
243 zipValid = false; // update validity tracker
244 document.getElementById('validateZip').
245 innerHTML = data.ErrorText; // display the error
246
247 // clear city and state values if they exist
248 document.getElementById('city').innerHTML = '';
249 document.getElementById('state').innerHTML = '';
250 } // end else
251 } // end if
252 else if (asyncRequest.status == 500)
253 {
254 document.getElementById('validateZip').
255 innerHTML = 'Zip validation service not available';
256 } // end else if
257 } // end if
258 } // end function showCityState
259
260 // send the telephone number to the server to validate format
261 function validatePhone(phone)
262 {
263 var params = '[{ "param": "tel", "value": "' + phone + '"}]';
264 callWebService("validateTel", params, showPhoneError);
265 } // end function validatePhone
266
267 // show whether the telephone number has correct format
268 function showPhoneError(asyncRequest)
269 {
270 // if request has completed successfully, process the response
271 if (asyncRequest.readyState == 4 && asyncRequest.status == 200)
272 {
273 // convert the JSON string to an object
274 var data = asyncRequest.responseText.parseJSON();
275
276 if (data.ErrorText != "Valid Telephone Format")
277 {

```

Fig. 15.9 | Ajax-enabled address-book application. (Part 6 of 10.)

```
278 phoneValid = false; // update validity tracker
279 } // end if
280 else
281 {
282 phoneValid = true; // update validity tracker
283 } // end else
284
285 document.getElementById('validatePhone').
286 innerHTML = data.ErrorText; // display the error
287 } // end if
288 } // end function showPhoneError
289
290 // enter the user's data into the database
291 function saveForm()
292 {
293 // retrieve the data from the form
294 var first = document.getElementById('first').value;
295 var last = document.getElementById('last').value;
296 var street = document.getElementById('street').value;
297 var city = document.getElementById('city').innerHTML;
298 var state = document.getElementById('state').innerHTML;
299 var zip = document.getElementById('zip').value;
300 var phone = document.getElementById('phone').value;
301
302 // check if data is valid
303 if (!zipValid || !phoneValid)
304 {
305 // display error message
306 document.getElementById('success').innerHTML =
307 'Invalid data entered. Check form for more information';
308 } // end if
309 else if ((first == "") || (last == ""))
310 {
311 // display error message
312 document.getElementById('success').innerHTML =
313 'First Name and Last Name must have a value.';
314 } // end if
315 else
316 {
317 // hide the form and show the addressbook
318 document.getElementById('addEntry')
319 .style.display = 'none';
320 document.getElementById('addressBook').
321 style.display = 'block';
322
323 // build the parameter to include in the web service URL
324 params = '[{"param": "first", "value": "' + first +
325 '"}, {"param": "last", "value": "' + last +
326 '"}, {"param": "street", "value": "' + street +
327 '"}, {"param": "city", "value": "' + city +
328 '"}, {"param": "state", "value": "' + state +
329 '"}, {"param": "zip", "value": "' + zip +
330 '"}, {"param": "tel", "value": "' + phone + '"}]';
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 7 of 10.)

```
331
332 // call the web service to insert data into the database
333 callWebService("addEntry", params, parseData);
334 } // end else
335 } // end function saveForm
336 //-->
337 </script>
338 </head>
339 <body onload = "showAddressBook()">
340 <div>
341 <input type = "button" value = "Address Book"
342 onclick = "showAddressBook()"/>
343 <input type = "button" value = "Add an Entry"
344 onclick = "addEntry()"/>
345 </div>
346 <div id = "addressBook" style = "display : block;">
347 Search By Last Name:
348 <input onkeyup = "search(this.value)"/>
349

350 <div id = "Names">
351 </div>
352 </div>
353 <div id = "addEntry" style = "display : none">
354 First Name: <input id = 'first' />
355

356 Last Name: <input id = 'last' />
357

358 Address:
359

360 Street: <input id = 'street' />
361

362 City:
363

364 State:
365

366 Zip: <input id = 'zip' onblur = 'validateZip(this.value)' />
367
368
369

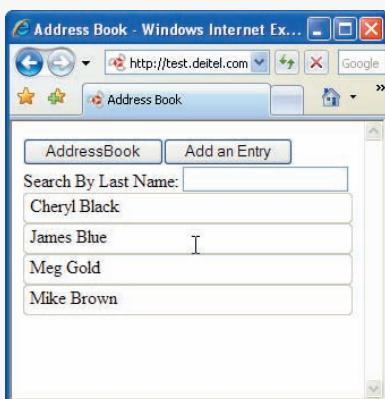
370 Telephone:<input id = 'phone'
371 onblur = 'validatePhone(this.value)' />
372
373
374

375 <input type = "button" value = "Submit"
376 onclick = "saveForm() " />
377

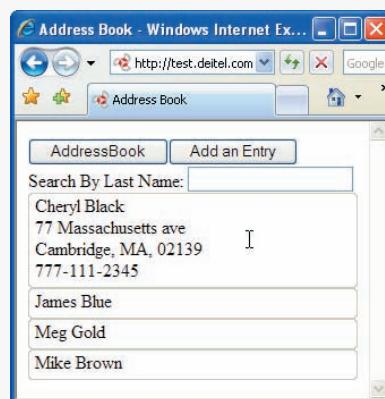
378 <div id = "success" class = "validator">
379 </div>
380 </div>
381 </body>
382 </html>
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 8 of 10.)

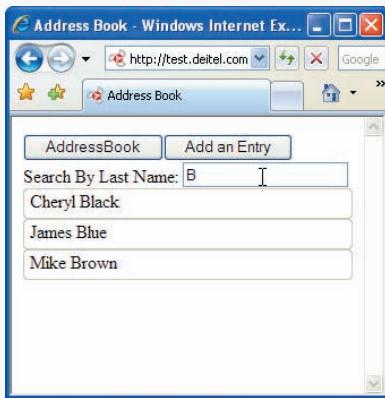
a) Page is loaded. All the entries are displayed.



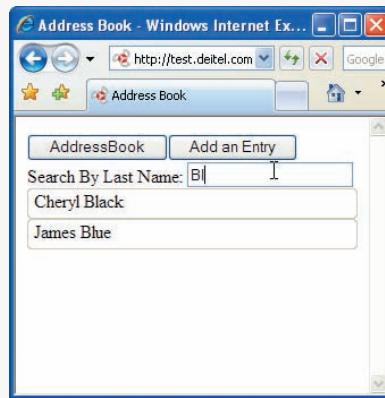
b) User clicks on an entry. The entry expands, showing the address and the telephone.



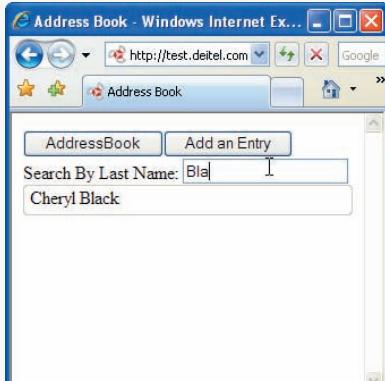
c) User types "B" in the search field. Application loads the entries whose last names start with "B".



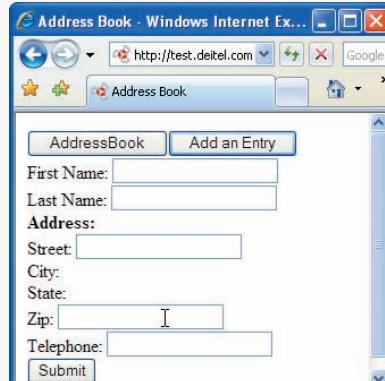
d) User types "Bl" in the search field. Application loads the entries whose last names start with "Bl".



e) User types "Bla" in the search field. Application loads the entries whose last names start with "Bla".



f) User clicks **Add an Entry** button. The form allowing user to add an entry is displayed.



**Fig. 15.9** | Ajax-enabled address-book application. (Part 9 of 10.)

g) User types in a nonexistent zip code. An error is displayed.

This screenshot shows the address book application interface. The user has entered "1910" in the Zip field. A tooltip message "Zip code does not exist" is displayed next to the Zip input field. The rest of the fields (First Name, Last Name, Address, City, State, Telephone) are empty.

i) The server finds the city and state associated with the zip code entered and displays them on the page.

This screenshot shows the address book application after the user has entered "19106" in the Zip field. The application has populated the City ("Philadelphia") and State ("PA") fields. The rest of the fields (First Name, Last Name, Address, Telephone) are empty.

k) The user enters the last name and the first name and clicks the Submit button.

This screenshot shows the address book application after the user has entered "John" in the First Name field and "Gray" in the Last Name field, then clicked the Submit button. The application has displayed a success message "First Name and Last Name must have a value." The rest of the fields (Address, Zip, Telephone) are empty.

h) User enters a valid zip code. While the server processes it, Checking Zip... is displayed on the page.

This screenshot shows the address book application while the server is processing a zip code. The Zip field contains "19106" and the message "Checking Zip..." is displayed next to it. The rest of the fields (First Name, Last Name, Address, City, State, Telephone) are empty.

j) The user enters a telephone number and tries to submit the data. The application does not allow this, because the First Name and Last Name are empty.

This screenshot shows the address book application after the user has entered "555-111-2222" in the Telephone field and clicked the Submit button. The application has displayed an error message "First Name and Last Name must have a value." The rest of the fields (First Name, Last Name, Address, Zip, Telephone) are empty.

l) The address book is redisplayed with the new name added in.

This screenshot shows the address book application after the user has successfully submitted the new contact. The contact "John Gray" has been added to the list, which also includes "Cheryl Black", "James Blue", "Meg Gold", and "Mike Brown". The rest of the fields (First Name, Last Name, Address, Zip, Telephone) are empty.

**Fig. 15.9** | Ajax-enabled address-book application. (Part 10 of 10.)

The application also enables the user to add another entry to the address book by clicking the **addEntry** button (Fig. 15.9(f)). The application displays a form that enables live field validation. As the user fills out the form, the zip-code value is validated and used to generate the city and state (Fig. 15.9(g), Fig. 15.9(h) and Fig. 15.9(i)). The telephone number is validated for correct format (Fig. 15.9(j)). When the **Submit** button is clicked, the application checks for invalid data and stores the values in a database on the server (Fig. 15.9(k) and Fig. 15.9(l)). You can test-drive this application at [test.deitel.com/examples/iw3http4/ajax/fig15\\_09\\_10/AddressBook.html](http://test.deitel.com/examples/iw3http4/ajax/fig15_09_10/AddressBook.html).

### *Interacting with a Web Service on the Server*

When the page loads, the `onLoad` event (line 339) calls the `showAddressBook` function to load the address book onto the page. Function `showAddressBook` (lines 21–29) shows the `addressBook` element and hides the `addEntry` element using the HTML DOM (lines 24–25). Then it calls function `callWebService` to make an asynchronous request to the server (line 28). Function `callWebService` requires an array of parameter objects to be sent to the server. In this case, the function we are invoking on the server requires no arguments, so line 27 creates an empty array to be passed to `callWebService`. Our program uses an ASP.NET web service that we created for this example to do the server-side processing. The web service contains a collection of methods that can be called from a web application.

Function `callWebService` (lines 32–72) contains the code to call our web service, given a method name, an array of parameter bindings (i.e., the method's parameter names and argument values) and the name of a callback function. The web-service application and the method that is being called are specified in the request URL (line 35). When sending the request using the `GET` method, the parameters are concatenated URL starting with a `?` symbol and followed by a list of *parameter=value* bindings, each separated by an `&`. Lines 39–49 iterate over the array of parameter bindings that was passed as an argument, and add them to the request URL. In this first call, we do not pass any parameters because the web method that returns all the entries requires none. However, future web method calls will send multiple parameter bindings to the web service. Lines 52–71 prepare and send the request, using similar functionality to the previous two examples. There are many types of user interaction in this application, each requiring a separate asynchronous request. For this reason, we pass the appropriate `asyncRequest` object as an argument to the function specified by the `callback` parameter. However, event handlers cannot receive arguments, so lines 57–60 assign an anonymous function to `asyncRequest`'s `onreadystatechange` property. When this anonymous function gets called, it calls function `callBack` and passes the `asyncRequest` object as an argument. Lines 64–65 set an `Accept` request header to receive JSON formatted data.

### *Parsing JSON Data*

Each of our web service's methods in this example returns a JSON representation of an object or array of objects. For example, when the web application requests the list of names in the address book, the list is returned as a JSON array, as shown in Fig. 15.10. Each object in Fig. 15.10 has the attributes `first` and `last`.

Line 11 links the `json.js` script to the XHTML file so we can parse JSON data. When the `XMLHttpRequest` object receives the response, it calls function `parseData` (lines 75–84). Line 81 calls the string's `parseJSON` function, which converts the JSON string into a JavaScript object. Then line 82 calls function `displayNames` (lines 87–106), which

```

1 [{ "first": "Cheryl", "last": "Black" },
2 { "first": "James", "last": "Blue" },
3 { "first": "Mike", "last": "Brown" },
4 { "first": "Meg", "last": "Gold" }]

```

**Fig. 15.10** | Address-book data formatted in JSON.

displays the first and last name of each address-book entry passed to it. Lines 90–91 use the DOM to store the placeholder `div` element `Names` in the variable `listbox`, and clear its content. Once parsed, the JSON string of address-book entries becomes an array, which this function traverses (lines 94–105).

### *Creating XHTML Elements and Setting Event Handlers on the Fly*

Line 99 uses an XHTML `fieldset` element to create a box in which the entry will be placed. Line 100 registers function `handleOnClick` as the `onclick` event handler for the `div` created in line 98. This enables the user to expand each address-book entry by clicking it. Function `handleOnClick` (lines 109–113) calls the `getAddress` function whenever the user clicks an entry. The parameters are generated dynamically and not evaluated until the `getAddress` function is called. This enables each function to receive arguments that are specific to the entry the user clicked. Line 102 displays the names on the page by accessing the `first` (first name) and `last` (last name) fields of each element of the data array.

Function `getAddress` (lines 136–166) is called when the user clicks an entry. This request must keep track of the entry where the address is to be displayed on the page. Lines 151–154 set the `displayAddress` function (lines 168–187) as the callback function, and pass it the entry element as a parameter. Once the request completes successfully, lines 174–178 parse the response and display the addresses. Lines 181–184 update the `div`'s `onclick` event handler to hide the address data when that `div` is clicked again by the user. When the user clicks an expanded entry, function `clearField` (lines 190–197) is called. Lines 192–196 reset the entry's content and its `onclick` event handler to the values they had before the entry was expanded.

### *Implementing Type-Ahead*

The `input` element declared in line 348 enables the user to search the address book by last name. As soon as the user starts typing in the input box, the `onkeyup` event handler calls the `search` function (lines 117–133), passing the `input` element's value as an argument. The `search` function performs an asynchronous request to locate entries with last names that start with its argument value. When the response is received, the application displays the matching list of names. Each time the user changes the text in the input box, function `search` is called again to make another asynchronous request.

The `search` function (lines 117–133) first clears the address-book entries from the page (lines 120–121). If the `input` argument is the empty string, line 126 displays the entire address book by calling function `showAddressBook`. Otherwise lines 130–131 send a request to the server to search the data. Line 130 creates a JSON string to represent the parameter object to be sent as an argument to the `callWebServices` function. Line 131 converts the string to an object and calls the `callWebServices` function. When the server responds, callback function `parseData` is invoked, which calls function `displayNames` to display the results on the page.

### *Implementing a Form with Asynchronous Validation*

When the **Add an Entry** button (lines 343–344) is clicked, the `addEntry` function (lines 200–204) is called, which hides the `addressBook` element and shows the `addEntry` element that allows the user to add a person to the address book. The `addEntry` element (lines 353–380) contains a set of entry fields, some of which have event handlers that enable validation that occurs asynchronously as the user continues to interact with the page. When a user enters a zip code, the `validateZip` function (lines 207–212) is called. This function calls an external web service to validate the zip code. If it is valid, that external web service returns the corresponding city and state. Line 210 builds a parameter object containing `validateZip`'s parameter name and argument value in JSON format. Line 211 calls the `callWebService` function with the appropriate method, the parameter object created in line 210 and `showCityState` (lines 216–258) as the callback function.

Zip-code validation can take a long time due to network delays. The `showCityState` function is called every time the request object's `readyState` property changes. Until the request completes, lines 219–220 display "Checking zip code..." on the page. After the request completes, line 228 converts the JSON response text to an object. The response object has four properties—`Validity`, `ErrorText`, `City` and `State`. If the request is valid, line 233 updates the `zipValid` variable that keeps track of zip-code validity (declared at line 18), and lines 237–239 show the city and state that the server generated using the zip code. Otherwise lines 243–245 update the `zipValid` variable and show the error code. Lines 248–249 clear the city and state elements. If our web service fails to connect to the zip-code validator web service, lines 252–256 display an appropriate error message.

Similarly, when the user enters the telephone number, the function `validatePhone` (lines 261–265) sends the phone number to the server. Once the server responds, the `showPhoneError` function (lines 268–288) updates the `validatePhone` variable (declared at line 17) and shows the message that the web service returned.

When the **Submit** button is clicked, the `saveForm` function is called (lines 291–335). Lines 294–300 retrieve the data from the form. Lines 303–308 check if the zip code and telephone number are valid, and display the appropriate error message in the `Success` element on the bottom of the page. Before the data can be entered into a database on the server, both the first-name and last-name fields must have a value. Lines 309–314 check that these fields are not empty and, if they are empty, display the appropriate error message. Once all the data entered is valid, lines 318–321 hide the entry form and show the address book. Lines 324–333 build the parameter object using JSON and send the data to the server using the `callWebService` function. Once the server saves the data, it queries the database for an updated list of entries and returns them; then function `parseData` displays the entries on the page.

## 15.8 Dojo Toolkit

Developing web applications in general, and Ajax applications in particular, involves a certain amount of painstaking and tedious work. Cross-browser compatibility, DOM manipulation and event handling can get cumbersome, particularly as an application's size increases. Dojo is a free, open source JavaScript library that takes care of these issues. Dojo reduces asynchronous request handling to a single function call. Dojo also provides cross-browser DOM functions that simplify partial page updates. It covers many more areas of web development, from simple event handling to fully functional rich GUI controls.

To install Dojo, download the Dojo version 0.4.3 from [www.Dojotoolkit.org/downloads](http://www.Dojotoolkit.org/downloads) to your hard drive. Extract the files from the archive file you downloaded to your web development directory or web server. Including the `dojo.js` script file in your web application will give you access to all the Dojo functions. To do this, place the following script in the head element of your XHTML document:

```
<script type = "text/javascript" src = "path/Dojo.js">
```

where `path` is the relative or complete path to the Dojo toolkit's files. Quick installation instructions for Dojo are provided at [Dojotoolkit.org/book/Dojo-book-0-9/part-1-life-Dojo/quick-installation](http://Dojotoolkit.org/book/Dojo-book-0-9/part-1-life-Dojo/quick-installation).

Figure 15.11 is a calendar application that uses Dojo to create the user interface, communicate with the server asynchronously, handle events and manipulate the DOM. The application contains a calendar control that shows the user six weeks of dates (see the screen captures in Fig. 15.11). Various arrow buttons allow the user to traverse the calendar. When the user selects a date, an asynchronous request obtains from the server a list of the scheduled events for that date. There is an **Edit** button next to each scheduled event. When the **Edit** button is clicked, the item is replaced by a text box with the item's content, a **Save** button and a **Cancel** button. When the user presses **Save**, an asynchronous request saves the new value to the server and displays it on the page. This feature, often referred to as **edit-in-place**, is common in Ajax applications. You can test-drive this application at [test.deitel.com/examples/iw3htp4/ajax/fig15\\_11/calendar.html](http://test.deitel.com/examples/iw3htp4/ajax/fig15_11/calendar.html).

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <!-- Fig. 15.11 Calendar.html -->
6 <!-- Calendar application built with dojo. -->
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8 <head>
9 <script type = "text/javascript" src = "/dojo043/dojo.js"></script>
10 <script type = "text/javascript" src = "json.js"></script>
11 <script type = "text/javascript">
12 <!--
13 // specify all the required dojo scripts
14 dojo.require("dojo.event.*"); // use scripts from event package
15 dojo.require("dojo.widget.*"); // use scripts from widget package
16 dojo.require("dojo.dom.*"); // use scripts from dom package
17 dojo.require("dojo.io.*"); // use scripts from the io package
18
19 // configure calendar event handler
20 function connectEventHandler()
21 {
22 var calendar = dojo.widget.byId("calendar"); // get calendar
23 calendar.setDate("2007-07-04");
24 dojo.event.connect(
25 calendar, "onValueChanged", "retrieveItems");
26 } // end function connectEventHandler
27

```

**Fig. 15.11** | Calendar application built with Dojo. (Part 1 of 7.)

```

28 // location of CalendarService web service
29 var webServiceUrl = "/CalendarService/CalendarService.asmx";
30
31 // obtain scheduled events for the specified date
32 function retrieveItems(eventDate)
33 {
34 // convert date object to string in yyyy-mm-dd format
35 var date = dojo.date.toRFC3339(eventDate).substring(0, 10);
36
37 // build parameters and call web service
38 var params = '[{"param":"eventDate", "value":"' +
39 date +
40 '"}]';
41 callWebService('getItemsByDate', params, displayItems);
42 } // end function retrieveItems
43
44 // call a specific web service asynchronously to get server data
45 function callWebService(method, params, callback)
46 {
47 // url for the asynchronous request
48 var requestUrl = webServiceUrl + "/" + method;
49 var params = paramString.parseJSON();
50
51 // build the parameter string to append to the url
52 for (var i = 0; i < params.length; i++)
53 {
54 // check if it is the first parameter and build
55 // the parameter string accordingly
56 if (i == 0)
57 requestUrl = requestUrl + "?" + params[i].param +
58 "=" + params[i].value; // add first parameter to url
59 else
60 requestUrl = requestUrl + "&" + params[i].param +
61 "=" + params[i].value; // add other parameters to url
62 } // end for
63
64 // call asynchronous request using dojo.io.bind
65 dojo.io.bind({ url: requestUrl, handler: callback,
66 accept: "application/json; charset=utf-8" });
67 } // end function callWebService
68
69 // display the list of scheduled events on the page
70 function displayItems(type, data, event)
71 {
72 if (type == 'error') // if the request has failed
73 {
74 alert('Could not retrieve the event'); // display error
75 } // end if
76 else
77 {
78 var placeholder = dojo.byId("itemList"); // get placeholder
79 placeholder.innerHTML = ''; // clear placeholder
80 var items = data.parseJSON(); // parse server data

```

Fig. 15.11 | Calendar application built with Dojo. (Part 2 of 7.)

```

81 // check whether there are events;
82 // if none then display message
83 if (items == "")
84 {
85 placeholder.innerHTML = 'No events for this date.';
86 }
87
88 for (var i = 0; i < items.length; i++)
89 {
90 // initialize item's container
91 var item = document.createElement("div");
92 item.id = items[i].id; // set DOM id to database id
93
94 // obtain and paste the item's description
95 var text = document.createElement("div");
96 text.innerHTML = items[i].description;
97 text.id = 'description' + item.id;
98 dojo.dom.insertAtIndex(text, item, 0);
99
100 // create and insert the placeholder for the edit button
101 var buttonPlaceHolder = document.createElement("div");
102 dojo.dom.insertAtIndex(buttonPlaceHolder, item, 1);
103
104 // create the edit button and paste it into the container
105 var editButton = dojo.widget.
106 createWidget("Button", {}, buttonPlaceHolder);
107 editButton.setCaption("Edit");
108 dojo.event.connect(
109 editButton, 'buttonClick', handleEdit);
110
111 // insert item container in the list of items container
112 dojo.dom.insertAtIndex(item, placeholder, i);
113 } // end for
114 } // end else
115 } // end function displayItems
116
117 // send the asynchronous request to get content for editing and
118 // run the edit-in-place UI
119 function handleEdit(event)
120 {
121 var id = event.currentTarget.parentNode.id; // retrieve id
122 var params = '[{ "param": "id", "value": "' + id + '"}]';
123 callWebService('getItemById', params, displayForEdit);
124 } // end function handleEdit
125
126 // set up the interface for editing an item
127 function displayForEdit(type, data, event)
128 {
129 if (type == 'error') // if the request has failed
130 {
131 alert('Could not retrieve the event'); // display error
132 }

```

Fig. 15.11 | Calendar application built with Dojo. (Part 3 of 7.)

```

133 else
134 {
135 var item = data.parseJSON(); // parse the item
136 var id = item.id; // set the id
137
138 // create div elements to insert content
139 var editElement = document.createElement('div');
140 var buttonElement = document.createElement('div');
141
142 // hide the unedited content
143 var oldItem = dojo.byId(id); // get the original element
144 oldItem.id = 'old' + oldItem.id; // change element's id
145 oldItem.style.display = 'none'; // hide old element
146 editElement.id = id; // change the "edit" container's id
147
148 // create a textbox and insert it on the page
149 var editArea = document.createElement('textarea');
150 editArea.id = 'edit' + id; // set textbox id
151 editArea.innerHTML = item.description; // insert description
152 dojo.dom.insertAtIndex(editArea, editElement, 0);
153
154 // create button placeholders and insert on the page
155 // these will be transformed into dojo widgets
156 var saveElement = document.createElement('div');
157 var cancelElement = document.createElement('div');
158 dojo.dom.insertAtIndex(saveElement, buttonElement, 0);
159 dojo.dom.insertAtIndex(cancelElement, buttonElement, 1);
160 dojo.dom.insertAtIndex(buttonElement, editElement, 1);
161
162 // create "save" and "cancel" buttons
163 var saveButton =
164 dojo.widget.createWidget("Button", {}, saveElement);
165 var cancelButton =
166 dojo.widget.createWidget("Button", {}, cancelElement);
167 saveButton.setCaption("Save"); // set saveButton label
168 cancelButton.setCaption("Cancel"); // set cancelButton text
169
170 // set up the event handlers for cancel and save buttons
171 dojo.event.connect(saveButton, 'buttonClick', handleSave);
172 dojo.event.connect(
173 cancelButton, 'buttonClick', handleCancel);
174
175 // paste the edit UI on the page
176 dojo.dom.insertAfter(editElement, oldItem);
177 } // end else
178 } // end function displayForEdit
179
180 // sends the changed content to the server to be saved
181 function handleSave(event)
182 {
183 // grab user entered data
184 var id = event.currentTarget.parentNode.parentNode.id;
185 var descr = dojo.byId('edit' + id).value;

```

**Fig. 15.11** | Calendar application built with Dojo. (Part 4 of 7.)

```

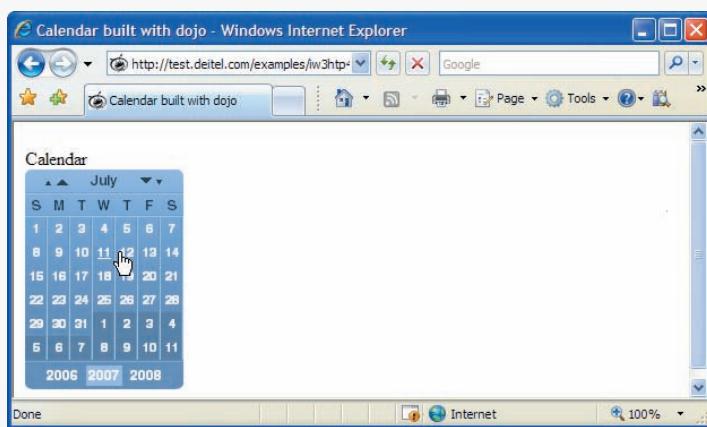
186 // build parameter string and call the web service
187 var params = '[{ "param": "id", "value": "' + id +
188 '"}, {"param": "descr", "value": "' + descr + '"}]';
189 callWebService('Save', params, displayEdited);
190 } // end function handleSave
191
192
193 // restores the original content of the item
194 function handleCancel(event)
195 {
196 var voidEdit = event.currentTarget.parentNode.parentNode;
197 var id = voidEdit.id; // retrieve the id of the item
198 dojo.dom.removeNode(voidEdit, true); // remove the edit UI
199 var old = dojo.byId('old' + id); // retrieve pre-edit version
200 old.style.display = 'block'; // show pre-edit version
201 old.id = id; // reset the id
202 } // end function handleCancel
203
204 // displays the updated event information after an edit is saved
205 function displayEdited(type, data, event)
206 {
207 if (type == 'error')
208 {
209 alert('Could not retrieve the event');
210 }
211 else
212 {
213 editedItem = data.parseJSON(); // obtain updated description
214 var id = editedItem.id; // obtain the id
215 var editElement = dojo.byId(id); // get the edit UI
216 dojo.dom.removeNode(editElement, true); // delete edit UI
217 var old = dojo.byId('old' + id); // get item container
218
219 // get pre-edit element and update its description
220 var oldText = dojo.byId('description' + id);
221 oldText.innerHTML = editedItem.description;
222
223 old.id = id; // reset id
224 old.style.display = 'block'; // show the updated item
225 } // end else
226 } // end function displayEdited
227
228 // when the page is loaded, set up the calendar event handler
229 dojo.addOnLoad(connectEventHandler);
230 // -->
231 </script>
232 <title> Calendar built with dojo </title>
233 </head>
234 <body>
235 Calendar
236 <div dojoType = "datePicker" style = "float: left"
237 widgetID = "calendar"></div>
238 <div id = "itemList" style = "float: left"></div>

```

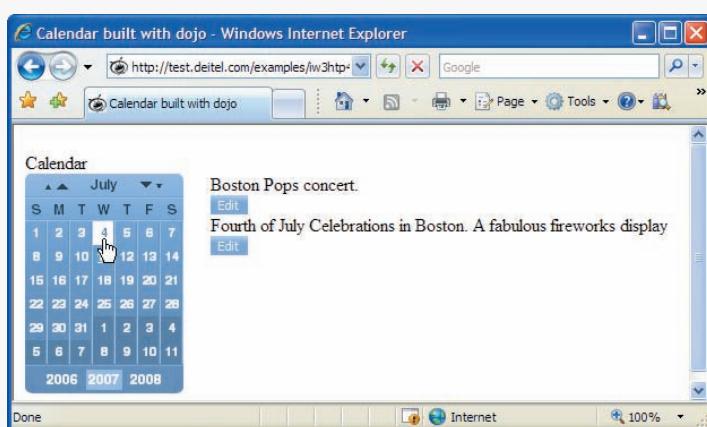
Fig. 15.11 | Calendar application built with Dojo. (Part 5 of 7.)

**239** </body>  
**240** </html>

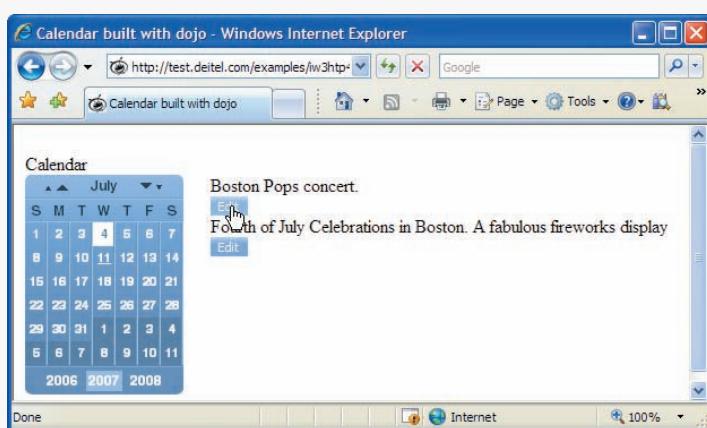
- a) DatePicker Dojo widget after the web page loads.



- b) User selects a date and the application asynchronously requests a list of events for that date and displays the results with a partial page update.

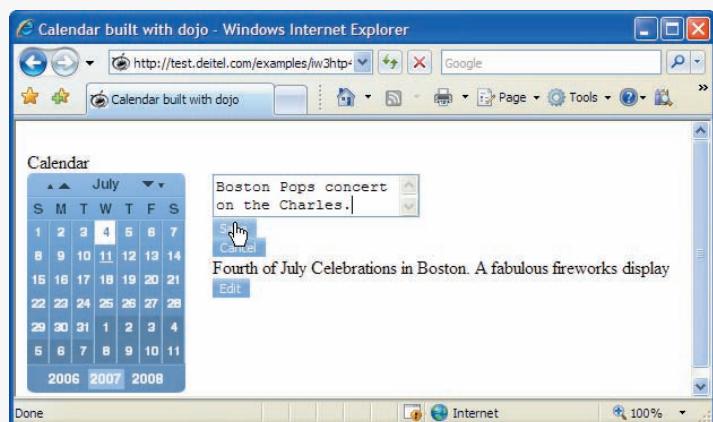


- c) User clicks the **Edit** button to modify an event's description.

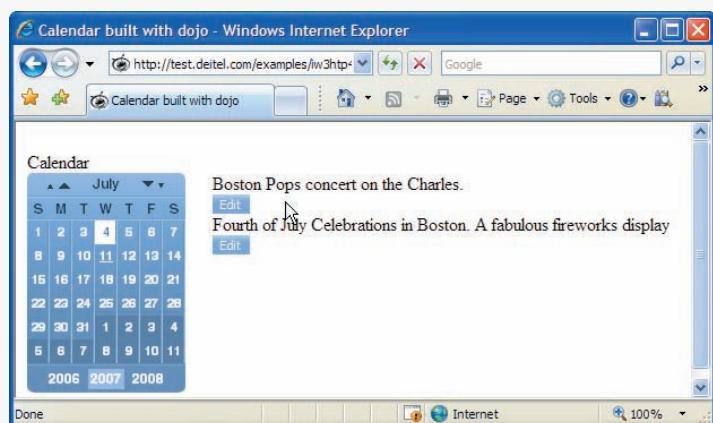


**Fig. 15.11** | Calendar application built with Dojo. (Part 6 of 7.)

d) Application performs a partial page update, replacing the original description and the **Edit** button with a text box, a **Save** button and a **Cancel** button. User modifies the event description and clicks the **Save** button.



d) The **Save** button's event handler uses an asynchronous request to update the server and uses the server's response to perform a partial page update, replacing the editing GUI components with the updated description and an **Edit** button.



**Fig. 15.11** | Calendar application built with Dojo. (Part 7 of 7.)

### Loading Dojo Packages

Lines 9–17 load the Dojo framework. Line 9 links the `dojo.js` script file to the page, giving the script access to all the functions in the Dojo toolkit. Dojo is organized in packages of related functionality. Lines 14–17 use the `dojo.require` call, provided by the `dojo.js` script to include the packages we need. The `dojo.io` package functions communicate with the server, the `dojo.event` package simplifies event handling, the `dojo.widget` package provides rich GUI controls, and the `dojo.dom` package contains additional DOM functions that are portable across many different browsers.

The application cannot use any of this functionality until all the packages have been loaded. Line 229 uses the `dojo.addOnLoad` method to set up the event handling after the page loads. Once all the packages have been loaded, the `connectEventHandler` function (lines 20–26) is called.

### Using an Existing Dojo Widget

A **Dojo widget** is any predefined user interface element that is part of the Dojo toolkit. The calendar control on the page is the `DatePicker` widget. To incorporate an existing

Dojo widget onto a page, you must set the `DojoType` attribute of any HTML element to the type of widget that you want it to be (line 236). Dojo widgets also have their own `widgetID` property (line 237). Line 22 uses the `dojo.widget.byId` method, rather than the DOM's `document.getElementById` method, to obtain the calendar widget element. The `dojo.events.connect` method links functions together. Lines 24–25 use it to connect the calendar's `onValueChanged` event handler to the `retrieveItems` function. When the user picks a date, a special `onValueChanged` event that is part of the `DatePicker` widget calls `retrieveItems`, passing the selected date as an argument. The `retrieveItems` function (lines 32–41) builds the parameters for the request to the server, and calls the `callWebService` function. Line 35 uses the `dojo.date.toRFC3339` method to convert the date passed by the calendar control to `yyyy-mm-dd` format.

### *Asynchronous Requests in Dojo*

The `callWebService` function (lines 44–66) sends the asynchronous request to the specified web-service method. Lines 47–61 build the request URL using the same code as Fig. 15.9. Dojo reduces the asynchronous request to a single call to the `dojo.io.bind` method (lines 64–65), which works on all the popular browsers such as Firefox, Internet Explorer, Opera, Mozilla and Safari. The method takes an array of parameters, formatted as a JavaScript object. The `url` parameter specifies the destination of the request, the `handler` parameter specifies the callback function, and the `mimetype` parameter specifies the format of the response. The `handler` parameter can be replaced by the `load` and `error` parameters. The function passed as `load` handles successful requests and the function passed as `error` handles unsuccessful requests.

Response handling is done differently in Dojo. Rather than calling the callback function every time the request's `readyState` property changes, Dojo calls the function passed as the “`handler`” parameter when the request completes. In addition, in Dojo the script does not have access to the request object. All the response data is sent directly to the callback function. The function sent as the `handler` argument must have three parameters—`type`, `data` and `event`.

In the first request, the function `displayItems` (lines 69–115) is set as the callback function. Lines 71–74 check if the request is successful, and display an error message if it isn't. Lines 77–78 obtain the place-holder element (`itemList`), where the items will be displayed, and clear its content. Line 79 converts the JSON response text to a JavaScript object, using the same code as the example in Fig. 15.9.

***Partial Page Updates Using Dojo's Cross-Browser DOM Manipulation Capabilities***  
 The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser portable manner. Lines 83–86 check if the server-side returned any items, and display an appropriate message if it didn't. For each item object returned from the server, lines 91–92 create a `div` element and set its `id` to the item's `id` in the database. Lines 95–97 create a container element for the item's description. Line 98 uses Dojo's `dojo.dom.insertAtIndex` method to insert the description element as the first element in the item's element.

For each entry, the application creates an `Edit` button that enables the user to edit the event's content on the page. Lines 101–109 create a Dojo `Button` widget programmatically. Lines 101–102 create a `buttonPlaceholder` `div` element for the button and paste it on the page. Lines 105–106 convert the `buttonPlaceholder` element to a Dojo `Button`

widget by calling the `dojo.widget.createWidget` function. This function takes three parameters—the type of widget to be created, a list of additional widget parameters and the element which is to be converted to a Dojo widget. Line 107 uses the button’s `setCaption` method to set the text that appears on the button. Line 112 uses the `insertAtIndex` method to insert the items into the `itemList` placeholder, in the order in which they were returned from the server.

### *Adding Edit-In-Place Functionality*

Dojo Button widgets use their own `buttonClick` event instead of the DOM `onclick` event to store the event handler. Lines 108–109 use the `dojo.event.connect` method to connect the `buttonClick` event of the Dojo Button widget and the `handleEdit` event handler (lines 119–124). When the user clicks the `Edit` button, the `Event` object gets passed to the event handler as an argument. The `Event` object’s `currentTarget` property contains the element that initiated the event. Line 121 uses the `currentTarget` property to obtain the `id` of the item. This `id` is the same as the item’s `id` in the server database. Line 123 calls the web service’s `getItemById` method, using the `callWebService` function to obtain the item that needs to be edited.

Once the server responds, the function `displayForEdit` (lines 127–178) replaces the item on the screen with the user interface used for editing the item’s content. The code for this is similar to the code in the `displayItems` function. Lines 129–132 make sure the request was successful and parse the data from the server. Lines 139–140 create the container elements into which we insert the new user-interface elements. Lines 143–146 hide the element that displays the item and change its `id`. Now the `id` of the user-interface element is the same as the `id` of the item that it’s editing stored in the database. Lines 149–152 create the text-box element that will be used to edit the item’s description, paste it into the text box, and paste the resulting text box on the page. Lines 156–173 use the same syntax that was used to create the `Edit` button widget to create `Save` and `Cancel` button widgets. Line 176 pastes the resulting element, containing the text box and two buttons, on the page.

When the user edits the content and clicks the `Cancel` button, the `handleCancel` function (lines 194–202) restores the item element to what it looked like before the button was clicked. Line 198 deletes the edit UI that was created earlier, using Dojo’s `removeNode` function. Lines 200–201 show the item with the original element that was used to display the item, and change its `id` back to the item’s `id` on the server database.

When the user clicks the `Save` button, the `handleSave` function (lines 181–191) sends the text entered by the user to the server. Line 185 obtains the text that the user entered in the text box. Lines 188–190 send to the server the `id` of the item that needs to be updated and the new description.

Once the server responds, `displayEdited` (lines 205–226) displays the new item on the page. Lines 214–217 contain the same code that was used in `handleCancel` to remove the user interface used to edit the item and redisplay the element that contains the item. Line 221 changes the item’s description to its new value.

## 15.9 Wrap-Up

In this chapter, we introduced Ajax and showed how to use it to create Rich Internet Applications (RIAs) that approximate the look, feel and usability of desktop applications.

You learned that RIAs have two key attributes—performance and a rich GUI. We discussed that RIA performance comes from Ajax (Asynchronous JavaScript and XML), which uses client-side scripting to make web applications more responsive by separating client-side user interaction and server communication, and running them in parallel.

You learned various ways to develop Ajax applications. We showed how to use “raw” Ajax with its component technologies (XHTML, CSS, JavaScript, dynamic HTML, the DOM, XML and the XMLHttpRequest object) to manage asynchronous requests to the server, then process the server responses (via JavaScript event handling) to perform partial page updates with the DOM on the client. You learned how to implement client/server communication using XML, and how to parse server responses using the DOM.

We discussed the impracticality of “raw” Ajax for developing large-scale applications and the hiding of such portability issues by Ajax toolkits, such as Dojo, Prototype, Script.aculo.us and ASP.NET Ajax, and by RIA environments such as Adobe’s Flex (Chapter 18), Microsoft’s Silverlight (Chapter 19) and JavaServer Faces (Chapters 26–27). You also learned that these Ajax libraries and RIA environments provide powerful ready-to-use GUI controls and functions that enrich web applications. You used two data formats—XML and JSON (JavaScript Object Notation)—for communicating between the server and client. We then built an Ajax application with a rich calendar GUI using the Dojo Ajax toolkit. In the applications we presented, you learned techniques including callback functions for handling asynchronous responses, partial page updates to display response data, JavaScript exception handling, type-ahead capabilities for making suggestions to users as they type in a text field and edit-in-place capabilities so users can edit entries directly in a web page.

In subsequent chapters, we use tools such as Adobe Flex, Microsoft Silverlight, Ruby on Rails and JavaServer Faces to build RIAs using Ajax. In Chapter 24, we’ll demonstrate features of the Prototype and Script.aculo.us Ajax libraries, which come with the Ruby on Rails framework. Prototype provides capabilities similar to Dojo. Script.aculo.us provides many “eye candy” effects that enable you to beautify your Ajax applications and create rich interfaces. In Chapter 27, we present Ajax-enabled JavaServer Faces (JSF) components. JSF uses Dojo to implement many of its client-side Ajax capabilities.

## 15.10 Web Resources

[www.deitel.com/ajax](http://www.deitel.com/ajax)

Our *Ajax Resource Center* contains links to some of the best Ajax resources on the web from which you can learn more about Ajax and its component technologies. Find categorized links to Ajax tools, code, forums, books, libraries, frameworks, conferences, podcasts and more. Check out the tutorials for all skill levels, from introductory to advanced. See our comprehensive list of developer toolkits and libraries. Visit the most popular Ajax community websites and blogs. Explore many popular commercial and free open-source Ajax applications. Download code snippets and complete scripts that you can use on your own website. Also, be sure to visit our Resource Centers with information on Ajax’s component technologies, including XHTML ([www.deitel.com/xhtml/](http://www.deitel.com/xhtml/)), CSS 2.1 ([www.deitel.com/css21/](http://www.deitel.com/css21/)), XML ([www.deitel.com/XML/](http://www.deitel.com/XML/)), and JavaScript ([www.deitel.com/javascript/](http://www.deitel.com/javascript/)). For a complete list of Resource Centers, visit [www.deitel.com/ResourceCenters.html](http://www.deitel.com/ResourceCenters.html).

## Summary

### Section 15.1 Introduction

- Despite the tremendous technological growth of the Internet over the past decade, the usability of web applications has lagged behind compared to desktop applications.
- Rich Internet Applications (RIAs) are web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and rich GUI.
- RIA performance comes from Ajax (Asynchronous JavaScript and XML), which uses client-side scripting to make web applications more responsive.
- Ajax applications separate client-side user interaction and server communication, and run them in parallel, making the delays of server-side processing more transparent to the user.
- “Raw” Ajax uses JavaScript to send asynchronous requests to the server, then updates the page using the DOM.
- When writing “raw” Ajax you need to deal directly with cross-browser portability issues, making it impractical for developing large-scale applications.
- Portability issues are hidden by Ajax toolkits, such as Dojo, Prototype and Script.aculo.us, which provide powerful ready-to-use controls and functions that enrich web applications and simplify JavaScript coding by making it cross-browser compatible.
- We achieve rich GUI in RIAs with Ajax toolkits and with RIA environments such as Adobe’s Flex, Microsoft’s Silverlight and JavaServer Faces. Such toolkits and environments provide powerful ready-to-use controls and functions that enrich web applications.
- The client-side of Ajax applications is written in XHTML and CSS, and uses JavaScript to add functionality to the user interface.
- XML and JSON are used to structure the data passed between the server and the client.
- The Ajax component that manages interaction with the server is usually implemented with JavaScript’s XMLHttpRequest object—commonly abbreviated as XHR.

### Section 15.2 Traditional Web Applications vs. Ajax Applications

- In traditional web applications, the user fills in the form’s fields, then submits the form. The browser generates a request to the server, which receives the request and processes it. The server generates and sends a response containing the exact page that the browser will render, which causes the browser to load the new page and temporarily makes the browser window blank. The client *waits* for the server to respond and  *reloads the entire page* with the data from the response.
- While a synchronous request is being processed on the server, the user cannot interact with the client web browser.
- The synchronous model was originally designed for a web of hypertext documents—what some people call the “brochure web.” This model yielded “choppy” application performance.
- In an Ajax application, when the user interacts with a page, the client creates an XMLHttpRequest object to manage a request. The XMLHttpRequest object sends the request to and awaits the response from the server. The requests are asynchronous, allowing the user to continue interacting with the application while the server processes the request concurrently. When the server responds, the XMLHttpRequest object that issued the request invokes a callback function, which typically uses partial page updates to display the returned data in the existing web page *without reloading the entire page*.
- The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications.

### **Section 15.3 Rich Internet Applications (RIAs) with Ajax**

- A classic XHTML registration form sends all of the data to be validated to the server when the user clicks the **Register** button. While the server is validating the data, the user cannot interact with the page. The server finds invalid data, generates a new page identifying the errors in the form and sends it back to the client—which renders the page in the browser. Once the user fixes the errors and clicks the **Register** button, the cycle repeats until no errors are found, then the data is stored on the server. The entire page reloads every time the user submits invalid data.
- Ajax-enabled forms are more interactive. Entries are validated dynamically as the user enters data into the fields. If a problem is found, the server sends an error message that is asynchronously displayed to inform the user of the problem. Sending each entry asynchronously allows the user to address invalid entries quickly, rather than making edits and resubmitting the entire form repeatedly until all entries are valid. Asynchronous requests could also be used to fill some fields based on previous fields' values.

### **Section 15.4 History of Ajax**

- The term Ajax was coined by Jesse James Garrett of Adaptive Path in February 2005, when he was presenting the previously unnamed technology to a client.
- All of the technologies involved in Ajax (XHTML, JavaScript, CSS, dynamic HTML, the DOM and XML) have existed for many years.
- In 1998, Microsoft introduced the `XMLHttpRequest` object to create and manage asynchronous requests and responses.
- Popular applications like Flickr, Google's Gmail and Google Maps use the `XMLHttpRequest` object to update pages dynamically.
- The name Ajax immediately caught on and brought attention to its component technologies. Ajax has quickly become one of the hottest technologies in web development, as it enables web-top applications to challenge the dominance of established desktop applications.

### **Section 15.5 “Raw” Ajax Example using the `XMLHttpRequest` Object**

- The `XMLHttpRequest` object (which resides on the client) is the layer between the client and the server that manages asynchronous requests in Ajax applications. This object is supported on most browsers, though they may implement it differently.
- To initiate an asynchronous request, you create an instance of the `XMLHttpRequest` object, then use its `open` method to set up the request, and its `send` method to initiate the request.
- When an Ajax application requests a file from a server, the browser typically caches that file. Subsequent requests for the same file can load it from the browser's cache.
- For security purposes, the `XMLHttpRequest` object does not allow a web application to request resources from servers other than the one that served the web application.
- Making a request to a different server is known as cross-site scripting (also known as XSS). You can implement a server-side proxy—an application on the web application's web server—that can make requests to other servers on the web application's behalf.
- When the third argument to `XMLHttpRequest` method `open` is `true`, the request is asynchronous.
- An exception is an indication of a problem that occurs during a program's execution.
- Exception handling enables you to create applications that can resolve (or handle) exceptions—in some cases allowing a program to continue executing as if no problem had been encountered.
- A `try` block encloses code that might cause an exception and code that should not execute if an exception occurs. A `try` block consists of the keyword `try` followed by a block of code enclosed in curly braces (`{}`).

- When an exception occurs, a `try` block terminates immediately and a `catch` block (also called a `catch` clause or exception handler) catches (i.e., receives) and handles an exception.
- The `catch` block begins with the keyword `catch` and is followed by an exception parameter in parentheses and a block of code enclosed in curly braces.
- The exception parameter's name enables the `catch` block to interact with a caught exception object, which contains `name` and `message` properties.
- A callback function is registered as the event handler for the `XMLHttpRequest` object's `onreadystatechange` event. Whenever the request makes progress, the `XMLHttpRequest` calls the `onreadystatechange` event handler.
- Progress is monitored by the `readyState` property, which has a value from 0 to 4. The value 0 indicates that the request is not initialized and the value 4 indicates that the request is complete.

### Section 15.6 Using XML and the DOM

- When passing structured data between the server and the client, Ajax applications often use XML because it consumes little bandwidth and is easy to parse.
- When the `XMLHttpRequest` object receives XML data, the `XMLHttpRequest` object parses and stores the data as a DOM object in the `responseXML` property.
- The `XMLHttpRequest` object's `responseXML` property contains the XML returned by the server.
- DOM method `createElement` creates an XHTML element of the specified type.
- DOM method `setAttribute` adds or changes an attribute of an XHTML element.
- DOM method `appendChild` inserts one XHTML element into another.
- The `innerHTML` property of a DOM element can be used to obtain or change the XHTML that is displayed in a particular element.

### Section 15.7 Creating a Full-Scale Ajax-Enabled Application

- JSON (JavaScript Object Notation)—a simple way to represent JavaScript objects as strings—is an alternative way (to XML) for passing data between the client and the server.
- Each JSON object is represented as a list of property names and values contained in curly braces.
- An array is represented in JSON with square brackets containing a comma-separated list of values.
- Each value in a JSON array can be a string, a number, a JSON representation of an object, `true`, `false` or `null`.
- JavaScript's `eval` function can convert JSON strings into JavaScript objects. To evaluate a JSON string properly, a left parenthesis should be placed at the beginning of the string and a right parenthesis at the end of the string before the string is passed to the `eval` function.
- The `eval` function creates a potential security risk—it executes any embedded JavaScript code in its string argument, possibly allowing a harmful script to be injected into JSON. A more secure way to process JSON is to use a JSON parser
- JSON strings are easier to create and parse than XML and require fewer bytes. For these reasons, JSON is commonly used to communicate in client/server interaction.
- When a request is sent using the `GET` method, the parameters are concatenated to the URL. URL parameter strings start with a `?` symbol and have a list of *parameter-value* bindings, each separated by an `&`.
- To implement type-ahead, you can use an element's `onkeyup` event handler to make asynchronous requests.

### Section 15.8 Dojo Toolkit

- Developing web applications in general, and Ajax applications in particular, involves a certain amount of painstaking and tedious work. Cross-browser compatibility, DOM manipulation and event handling can get cumbersome, particularly as an application's size increases. Dojo is a free, open source JavaScript library that takes care of these issues.
- Dojo reduces asynchronous request handling to a single function call.
- Dojo provides cross-browser DOM functions that simplify partial page updates. It also provides event handling and rich GUI controls.
- To install Dojo, download the latest release from [www.Dojotoolkit.org/downloads](http://www.Dojotoolkit.org/downloads) to your hard drive. Extract the files from the archive file you downloaded to your web development directory or web server. To include the `Dojo.js` script file in your web application, place the following script in the head element of your XHTML document:

```
<script type = "text/javascript" src = "path/Dojo.js">
```

where `path` is the relative or complete path to the Dojo toolkit's files.

- Edit-in-place enables a user to modify data directly in the web page, a common feature in Ajax applications.
- Dojo is organized in packages of related functionality.
- The `dojo.require` method is used to include specific Dojo packages.
- The `dojo.io` package functions communicate with the server, the `dojo.event` package simplifies event handling, the `dojo.widget` package provides rich GUI controls, and the `dojo.dom` package contains additional DOM functions that are portable across many different browsers.
- A Dojo widget is any predefined user interface element that is part of the Dojo toolkit.
- To incorporate an existing Dojo widget onto a page, you must set the `dojoType` attribute of any HTML element to the type of widget that you want it to be.
- The `dojo.widget.byId` method can be used to obtain a Dojo widget.
- The `dojo.events.connect` method links functions together.
- The `dojo.date.toRFC3339` method converts a date to `yyyy-mm-dd` format.
- The `dojo.io.bind` method configures and sends asynchronous requests. The method takes an array of parameters, formatted as a JavaScript object. The `url` parameter specifies the destination of the request, the `handler` parameter specifies the callback function, and the `mimetype` parameter specifies the format of the response. The `handler` parameter can be replaced by the `load` and `error` parameters. The function passed as the `load` handler processes successful requests and the function passed as the `error` handler processes unsuccessful requests.
- Dojo calls the function passed as the `handler` parameter only when the request completes.
- In Dojo, the script does not have access to the request object. All the response data is sent directly to the callback function.
- The function sent as the `handler` argument must have three parameters—`type`, `data` and `event`.
- The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser manner.
- Dojo's `dojo.dom.insertAtIndex` method inserts an element at the specified index in the DOM.
- Dojo's `removeNode` function removes an element from the DOM.
- Dojo Button widgets use their own `buttonClick` event instead of the DOM `onclick` event to store the event handler.
- The Event object's `currentTarget` property contains the element that initiated the event.

## Terminology

|                                                          |                                                |
|----------------------------------------------------------|------------------------------------------------|
| Ajax                                                     | partial page update                            |
| Ajax toolkit                                             | Prototype Ajax library                         |
| asynchronous request                                     | “raw” Ajax                                     |
| callback function                                        | readyState property of XMLHttpRequest object   |
| catch block                                              | responseText property of XMLHttpRequest object |
| catch clause                                             | ject                                           |
| catch keyword                                            | responseXML property of XMLHttpRequest object  |
| cross-browser compatibility                              | same origin policy (SOP)                       |
| cross-site scripting (XSS)                               | Script.aculo.us Ajax library                   |
| Dojo Ajax library                                        | send method of XMLHttpRequest                  |
| edit-in-place                                            | setRequestHeader method of XMLHttpRequest      |
| exception                                                | object                                         |
| exception handler                                        | status property of XMLHttpRequest object       |
| exception handling                                       | statusText property of XMLHttpRequest object   |
| GET method of XMLHttpRequest object                      | synchronous request                            |
| getResponseHeader method of XMLHttpRequest object        | try block                                      |
| JavaScript Object Notation (JSON)                        | try keyword                                    |
| onReadyStateChange property of the XMLHttpRequest object | type ahead                                     |
| open method of XMLHttpRequest                            | XHR (abbreviation for XMLHttpRequest)          |
|                                                          | XMLHttpRequest object                          |

## Self-Review Exercises

**15.1** Fill in the blanks in each of the following statements:

- Ajax applications use \_\_\_\_\_ requests to create Rich Internet Applications.
- In Ajax applications, the \_\_\_\_\_ object manages asynchronous interaction with the server.
- The event handler called when the server responds is known as a(n) \_\_\_\_\_ function.
- The \_\_\_\_\_ attribute can be accessed through the DOM to update an XHTML element’s content without reloading the page.
- JavaScript’s XMLHttpRequest object is commonly abbreviated as \_\_\_\_\_.
- \_\_\_\_\_ is a simple way to represent JavaScript objects as strings.
- Making a request to a different server is known as \_\_\_\_\_.
- JavaScript’s \_\_\_\_\_ function can convert JSON strings into JavaScript objects.
- A(n) \_\_\_\_\_ encloses code that might cause an exception and code that should not execute if an exception occurs.
- The XMLHttpRequest object’s \_\_\_\_\_ contains the XML returned by the server.

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

- Ajax applications must use XML for server responses.
- The technologies that are used to develop Ajax applications have existed since the 1990s.
- The event handler that processes the response is stored in the readyState property of XMLHttpRequest.
- An Ajax application can be implemented so that it never needs to reload the page on which it runs.
- The responseXML property of the XMLHttpRequest object stores the server’s response as a raw XML string.

- f) The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser manner.
- g) An exception indicates successful completion of a program's execution.
- h) When the third argument to XMLHttpRequest method open is `false`, the request is asynchronous.
- i) For security purposes, the XMLHttpRequest object does not allow a web application to request resources from servers other than the one that served the web application.
- j) The `innerHTML` property of a DOM element can be used to obtain or change the XML that is displayed in a particular element.

## Answers to Self-Review Exercises

**15.1** a) asynchronous. b) XMLHttpRequest. c) callback. d) `innerHTML`. e) XHR. f) JSON.  
g) cross-site scripting (or XSS). h) `eval`. i) try block. j) `responseXML` property.

**15.2** a) False. Ajax applications can use any type of textual data as a response. For example, we used JSON in this chapter.  
b) True.  
c) False. `readyState` is the property that keeps track of the request's progress. The event handler is stored in the `onreadystatechange` property.  
d) True.  
e) False. If the response data has XML format, the XMLHttpRequest object parses it and stores it in a document object.  
f) True.  
g) False. An exception is an indication of a problem that occurs during a program's execution.  
h) False. The third argument to XMLHttpRequest method open must be `true` to make an asynchronous request.  
i) True.  
j) True.

## Exercises

**15.3** Describe the differences between client/server interactions in traditional web applications and client/server interactions in Ajax web applications.

**15.4** Consider the AddressBook application in Fig. 15.9. Describe how you could reimplement the type-ahead capability so that it could perform the search using data previously downloaded rather than making an asynchronous request to the server after every keystroke.

**15.5** Describe each of the following terms in the context of Ajax:

- a) type-ahead
- b) edit-in-place
- c) partial page update
- d) asynchronous request
- e) XMLHttpRequest
- f) “raw” Ajax
- g) callback function
- h) same origin policy
- i) Ajax libraries
- j) RIA

[*Note to Instructors and Students:* Due to security restrictions on using XMLHttpRequest, Ajax applications must be placed on a web server (even one on your local computer) to enable the appli-

cations to work correctly, and when they need to access other resources, those must reside on the same web server. **Students:** You'll need to work closely with your instructors to understand your lab setup so you can run your solutions to the exercises (the examples are already posted on our web server) and to run many of the other server-side applications that you'll learn later in the book.]

**15.6** The XML files used in the book-cover catalog example (Fig. 15.8) also store the titles of the books in a `title` attribute of each `cover` node. Modify the example so that every time the mouse hovers over an image, the book's title is displayed below the image.

**15.7** Create an Ajax-enabled version of the feedback form from Fig. 4.13. As the user moves between form fields, ensure that each field is non-empty. For the e-mail field, ensure that the e-mail address has valid format. In addition, create an XML file that contains a list of email addresses that are not allowed to post feedback. Each time the user enters an e-mail address check whether it is on that list; if so, display an appropriate message.

**15.8** Create an Ajax-based product catalog that obtains its data from JSON files located on the server. The data should be separated into four JSON files. The first file should be a summary file, containing a list of products. Each product should have a title, an image filename for a thumbnail image and a price. The second file should contain a list of descriptions for each product. The third file should contain a list of filenames for the full-size product images. The last file should contain a list of the thumbnail image file names. Each item in a catalogue should have a unique ID that should be included with the entries for that product in every file. Next, create an Ajax-enabled web page that displays the product information in a table. The catalog should initially display a list of product names with their associated thumbnail images and prices. When the mouse hovers over a thumbnail image, the larger product image should be displayed. When the user moves the mouse away from that image, the original thumbnail should be redisplayed. You should provide a button that the user can click to display the product description.

**15.9** Create a version of Exercise 15.8 that uses Dojo's capabilities and widgets to display the product catalog. Modify the asynchronous request's to use `dojo.io.bind` functions rather than raw Ajax. Use Dojo's DOM functionality to place elements on the page. Improve the look of the page by using Dojo's button widgets rather than XHTML button elements..

# 3

PART

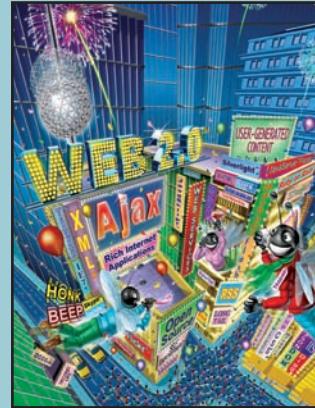
# *Rich Internet Application Client Technologies*

*The user should feel in control of the computer; not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency.*

—Inside Macintosh, Volume 1,  
Apple Computer, Inc., 1985

# 16

## Adobe® Flash® CS3



*Science and technology and the various forms of art, all unite humanity in a single and interconnected system.*

—Zhores Aleksandrovich Medvedev

*All the world's a stage, and all the men and women merely players; they have their exits and their entrances; and one man in his time plays many parts. . .*

—William Shakespeare

*Music has charms to soothe a savage breast,  
To soften rocks, or bend a knotted oak.*

—William Congreve

*A flash and where previously the brain held a dead fact,  
the soul grasps a living truth!  
At moments we are all artists.*

—Arnold Bennett

### OBJECTIVES

In this chapter you will learn:

- Flash CS3 multimedia development.
- To develop Flash movies.
- Flash animation techniques.
- ActionScript 3.0, Flash's object-oriented programming language.
- To create a preloading animation for a Flash movie.
- To add sound to Flash movies.
- To publish a Flash movie.
- To create special effects with Flash.
- To create a Splash Screen.

**Outline**

- 16.1** Introduction
- 16.2** Flash Movie Development
- 16.3** Learning Flash with Hands-On Examples
  - 16.3.1** Creating a Shape with the Oval Tool
  - 16.3.2** Adding Text to a Button
  - 16.3.3** Converting a Shape into a Symbol
  - 16.3.4** Editing Button Symbols
  - 16.3.5** Adding Keyframes
  - 16.3.6** Adding Sound to a Button
  - 16.3.7** Verifying Changes with **Test Movie**
  - 16.3.8** Adding Layers to a Movie
  - 16.3.9** Animating Text with Tweening
  - 16.3.10** Adding a Text Field
  - 16.3.11** Adding ActionScript
- 16.4** Publishing Your Flash Movie
- 16.5** Creating Special Effects with Flash
  - 16.5.1** Importing and Manipulating Bitmaps
  - 16.5.2** Creating an Advertisement Banner with Masking
  - 16.5.3** Adding Online Help to Forms
- 16.6** Creating a Website Splash Screen
- 16.7** ActionScript
- 16.8** Wrap-Up
- 16.9** Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 16.1 Introduction

Adobe **Flash CS3** (Creative Suite 3) is a commercial application that you can use to produce interactive, animated **movies**. Flash can be used to create web-based banner advertisements, interactive websites, games and web-based applications with stunning graphics and multimedia effects. It provides tools for drawing graphics, generating animations, and adding sound and video. Flash movies can be embedded in web pages, distributed on CDs and DVDs as independent applications, or converted into stand-alone, executable programs. Flash includes tools for coding in its scripting language—**ActionScript 3.0**—which is similar to JavaScript and enables interactive applications. A fully functional, 30-day trial version of Flash CS3 is available for download from:

[www.adobe.com/products/flash/](http://www.adobe.com/products/flash/)

To follow along with the examples in this chapter, please install this software before continuing. Follow the on-screen instructions to install the trial version of the Flash software.

To play Flash movies, the **Flash Player plug-in** must be installed in your web browser. The most recent version of the plug-in (at the time of this writing) is version 9. You can download the latest version from:

[www.adobe.com/go/getflashplayer](http://www.adobe.com/go/getflashplayer)

According to Adobe's statistics, approximately 98.7 percent of web users have Flash Player version 6 or greater installed, and 83.4 percent of web users have Flash Player version 9 installed.<sup>1</sup> There are ways to detect whether a user has the appropriate plug-in to view Flash content. Adobe provides a tool called the Flash Player Detection Kit which contains files that work together to detect whether a suitable version of Adobe Flash Player is installed in a user's web browser. This kit can be downloaded from:

[www.adobe.com/products/flashplayer/download/detection\\_kit/](http://www.adobe.com/products/flashplayer/download/detection_kit/)

This chapter introduces building Flash movies. You'll create interactive buttons, add sound to movies, create special graphic effects and integrate ActionScript in movies.

## 16.2 Flash Movie Development

Once Flash CS3 is installed, open the program. Flash's **Welcome Screen** appears by default. The **Welcome Screen** contains options such as **Open a Recent Item**, **Create New** and **Create from Template**. The bottom of the page contains links to useful help topics and tutorials. [Note: For additional help, refer to Flash's **Help** menu.]

To create a blank Flash document, click **Flash File (ActionScript 3.0)** under the **Create New** heading. Flash opens a new file called **Untitled-1** in the Flash development environment (Fig. 16.1).

At the center of the development environment is the movie **stage**—the white area in which you place graphic elements during movie development. Above the stage is the **timeline**, which represents the time period over which a movie runs. The timeline is divided into increments called **frames**, represented by gray and white rectangles. Each frame depicts a moment in time during the movie, into which you can insert movie elements. The **playhead** indicates the current frame.



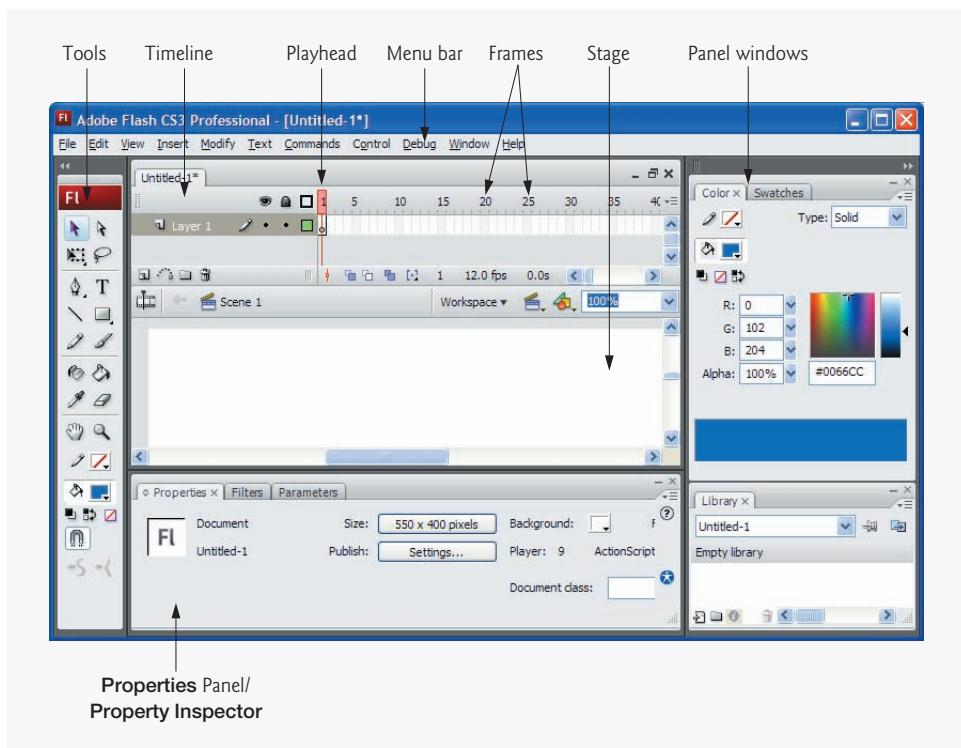
### Common Programming Error 16.1

*Elements placed off stage can still appear if the user changes the aspect ratio of the movie. If an element should not be visible, use an alpha of 0% to hide the element.*

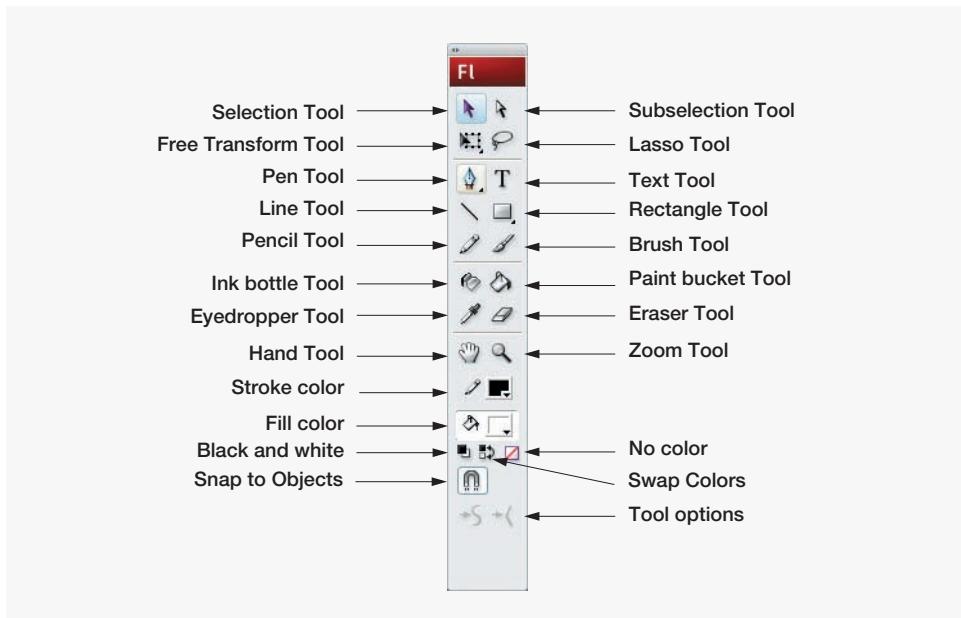
The development environment contains several windows that provide options and tools for creating Flash movies. Many of these tools are located in the **Tools bar**, the vertical window located at the left side of the development environment. The **Tools** bar (Fig. 16.2) is divided into multiple sections, each containing tools and functions that help you create Flash movies. The tools near the top of the **Tools** bar select, add and remove graphics from Flash movies. The **Hand** and **Zoom** tools allow you to pan and zoom in the stage. Another section of tools provides colors for shapes, lines and filled areas. The last section contains settings for the **active tool** (i.e., the tool that is highlighted and in use). You can make a tool behave differently by selecting a new mode from the options section of the **Tools** bar.

Application windows called **panels** organize frequently used movie options. Panel options modify the size, shape, color, alignment and effects associated with a movie's graphic elements. By default, panels line the right and bottom edges of the window. Panels

1. Flash Player statistics from Adobe's Flash Player Penetration Survey website at [www.adobe.com/products/player\\_census/flashplayer/version\\_penetration.html](http://www.adobe.com/products/player_census/flashplayer/version_penetration.html).



**Fig. 16.1** | Flash CS3 development environment.



**Fig. 16.2** | CS3 Tools bar.

may be placed anywhere in the development environment by dragging the tab at the left edge of their bars.

The context-sensitive **Properties** panel (frequently referred to as the **Properties** window) is located at the bottom of the screen by default. This panel displays various information about the currently selected object. It is Flash's most useful tool for viewing and altering an object's properties.

The **Color**, **Swatches**, **Properties**, **Filters** and **Parameters** panels also appear in the development environment by default. You can access different panels by selecting them from the **Window** menu. To save and manage customized panel layouts, select **Window > Workspace**, then use the **Save Current...** and **Manage...** options to save a layout or load an existing layout, respectively.

## 16.3 Learning Flash with Hands-On Examples

Now you'll create several complete Flash movies. The first example demonstrates how to create an interactive, animated button. ActionScript code will produce a random text string each time the button is clicked. To begin, create a new Flash movie. First, select **File > New**. In the **New Document** dialog (Fig. 16.3), select **Flash File (ActionScript 3.0)** under the **General** tab and click **OK**. Next, choose **File > Save As...** and save the movie as **Ceo-Assistant fla**. The **.fla** file extension is a Flash-specific extension for editable movies.



### Good Programming Practice 16.1

*Save each project with a meaningful name in its own folder. Creating a new folder for each movie helps keep projects organized.*

Right click the stage to open a menu containing different movie options. Select **Document Properties...** to display the **Document Properties** dialog (Fig. 16.4). This dialog can also be accessed by selecting **Document...** from the **Modify** menu. Settings such as the **Frame rate**, **Dimensions** and **Background color** are configured in this dialog.

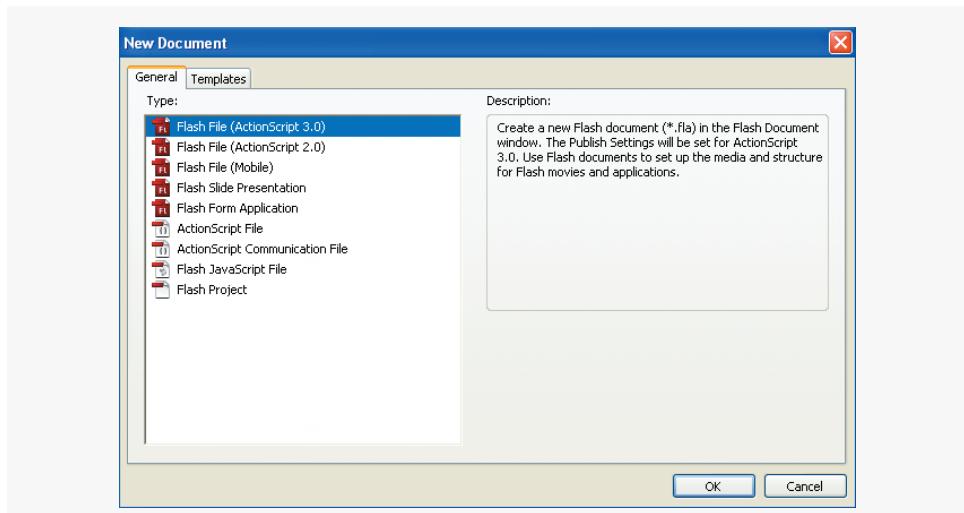
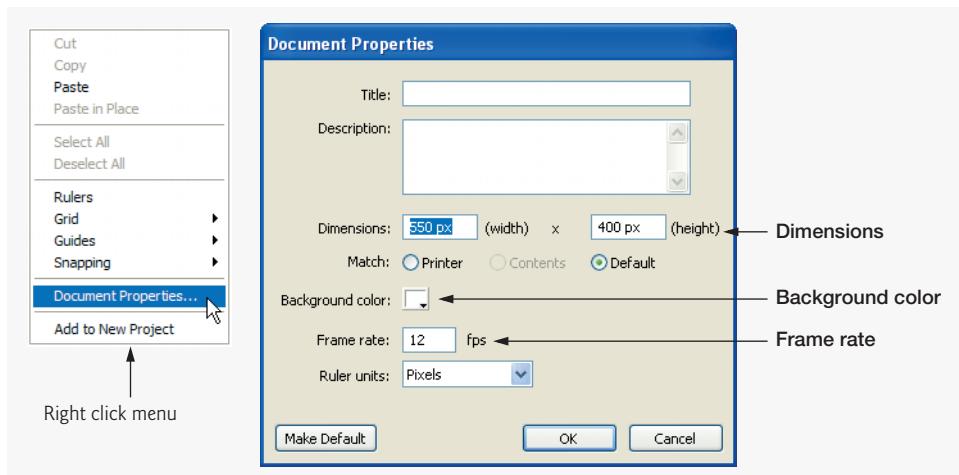


Fig. 16.3 | New Document dialog.



**Fig. 16.4 |** Document Properties dialog.

The **Frame rate** sets the speed at which movie frames display. A higher frame rate causes more frames to be displayed in a given unit of time (the standard measurement is seconds), thus creating a faster movie. The frame rate for Flash movies on the web is generally between 12 and 60 frames per second (**fps**). Flash's default frame rate is 12 fps. For this example, set the **Frame Rate** to 10 frames per second.

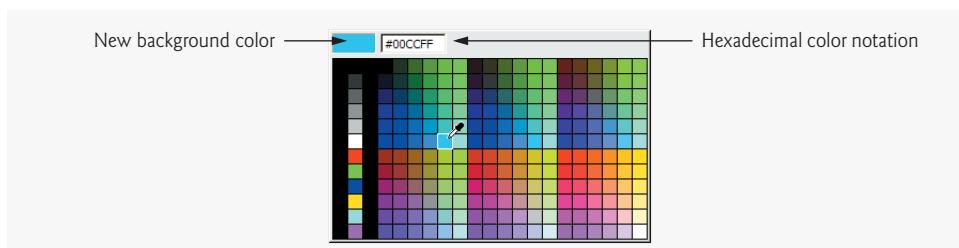


### Performance Tip 16.1

*Higher frame rates increase the amount of information to process, and thus increase the movie's processor usage and file size. Be especially aware of file sizes when catering to low bandwidth web users.*

The background color determines the color of the stage. Click the background-color box (called a **swatch**) to select the background color. A new panel opens, presenting a web-safe palette. Web-safe palettes and color selection are discussed in detail in Chapter 3. Note that the mouse pointer changes into an eyedropper, which indicates that you may select a color. Choose a light blue color (Fig. 16.5).

The box in the upper-left corner of the dialog displays the new background color. The **hexadecimal notation** for the selected color appears to the right of this box. The hexadecimal notation is the color code that a web browser uses to render color. Hexadecimal notation is discussed in detail in Appendix E, Number Systems.



**Fig. 16.5 |** Selecting a background color.

Dimensions define the size of the movie as it displays on the screen. For this example, set the movie width to 200 pixels and the movie height to 180 pixels. Click OK to apply the changes in the movie settings.



### Software Engineering Observation 16.1

*A movie's contents are not resized when you change the size of the movie stage.*

With the new dimensions, the stage appears smaller. Select the **Zoom Tool** from the toolbox (Fig. 16.2) and click the stage once to enlarge it to 200 percent of its size (i.e., zoom in). The current zoom percentage appears in the upper-right above the stage editing area. Editing a movie with small dimensions is easier when the stage is enlarged. Press the **Alt** key while clicking the zoom tool to reduce the size of the work area (i.e., zoom out). Select the **Hand Tool** from the toolbox, and drag the stage to the center of the editing area. The hand tool may be accessed at any time by holding down the *spacebar* key.

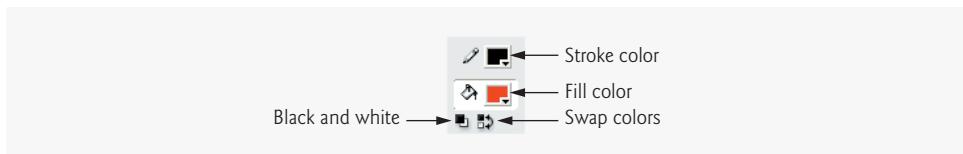
#### 16.3.1 Creating a Shape with the Oval Tool

Flash provides several editing tools and options for creating graphics. Flash creates shapes using **vectors**—mathematical equations that Flash uses to define size, shape and color. Some other graphics applications create raster graphics or bitmapped graphics. When vector graphics are saved, they are stored using equations. **Raster graphics** are defined by areas of colored **pixels**—the unit of measurement for most computer monitors. Raster graphics typically have larger file sizes because the computer saves the information for every pixel. Vector and raster graphics also differ in their ability to be resized. Vector graphics can be resized without losing clarity, whereas raster graphics lose clarity as they are enlarged or reduced.

We will now create an interactive button out of a circular shape. You can create shapes by dragging with the shape tools. Select the Oval tool from the toolbox. If the Oval tool is not already displayed, click and hold the Rectangle/Oval tool to display the list of rectangle and oval tools. We use this tool to specify the button area. Every shape has a **Stroke color** and a **Fill color**. The stroke color is the color of a shape's outline, and the fill color is the color that fills the shape. Click the swatches in the **Colors** section of the toolbox (Fig. 16.6) to set the fill color to red and the stroke color to black. Select the colors from the web-safe palette or enter their hexadecimal values.

Clicking the **Black and white** button resets the stroke color to black and the fill color to white. Selecting the **Swap colors** option switches the stroke and fill colors. A shape can be created without a fill or stroke color by selecting the **No color** option () when you select either the stroke or fill swatch.

Create the oval anywhere on the stage by dragging with the Oval tool while pressing the **Shift** key. The **Shift** key constrains the oval's proportions to have equal height and



**Fig. 16.6** | Setting the fill and stroke colors.

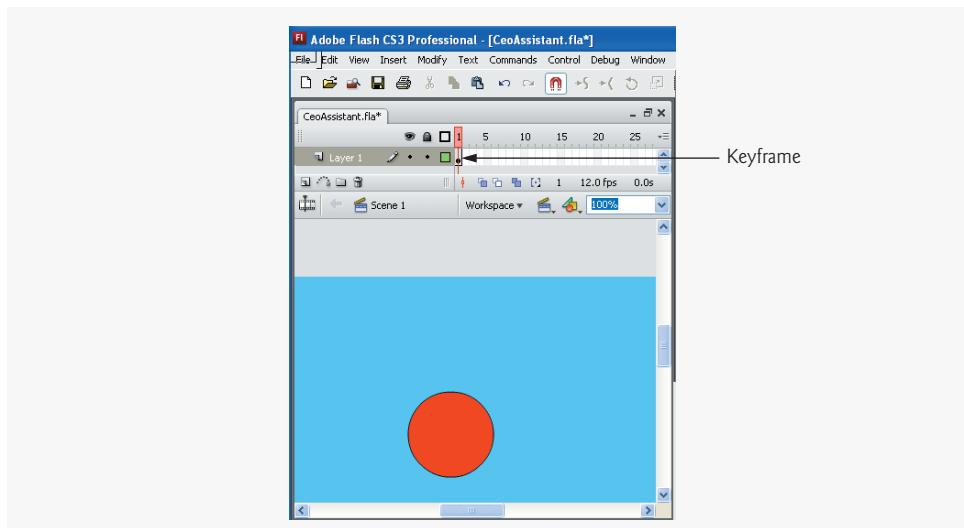
width (i.e., a circle). The same technique creates a square with the Rectangle tool or draws a straight line with the Pencil tool. Drag the mouse until the circle is approximately the size of a dime, then release the mouse button.

After you draw the oval, a dot appears in frame 1, the first frame of the timeline for **Layer 1**. This dot signifies a **keyframe** (Fig. 16.7), which indicates a point of change in a timeline. Whenever you draw a shape in an empty frame, Flash creates a keyframe.

The shape's fill and stroke may be edited individually. Click the red area with the **Selection** tool (black arrow) to select the circle fill. A grid of white dots appears over an object when it is selected (Fig. 16.8). Click the black stroke around the circle while pressing the *Shift* key to add to this selection. You can also make multiple selections by dragging with the selection tool to draw a selection box around specific items.

A shape's size can be modified with the **Properties** panel when the shape is selected (Fig. 16.9). If the panel is not open, open it by selecting **Properties** from the **Window** menu or pressing *<Ctrl>-F3*.

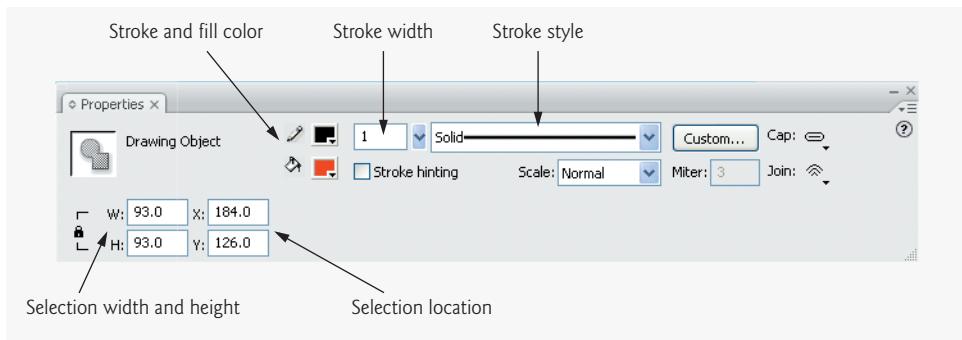
Set the width and height of the circle by typing **30** into the **W:** text field and **30** into the **H:** text field. Entering an equal width and height maintains a **constrained aspect ratio** while changing the circle's size. A constrained aspect ratio maintains an object's proportions as it is resized. Press *Enter* to apply these values.



**Fig. 16.7** | Keyframe added to the timeline.



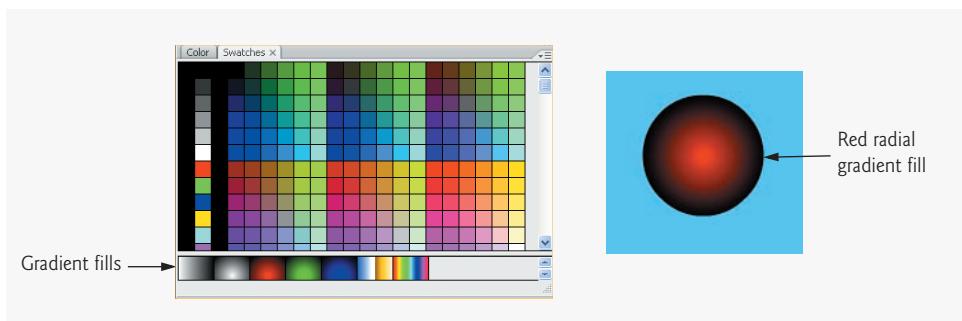
**Fig. 16.8** | Making multiple selections with the Selection tool.



**Fig. 16.9** | Modifying the size of a shape with the **Properties** window.

The next step is to modify the shape's color. We will apply a **gradient fill**—a gradual progression of color that fills the shape. Open the **Swatches** panel (Fig. 16.10), either by selecting **Swatches** from the **Window** menu or by pressing **<Ctrl>-F9**. The **Swatches** panel provides four **radial gradients** and three **linear gradients**, although you also can create and edit gradients with the **Color** panel.

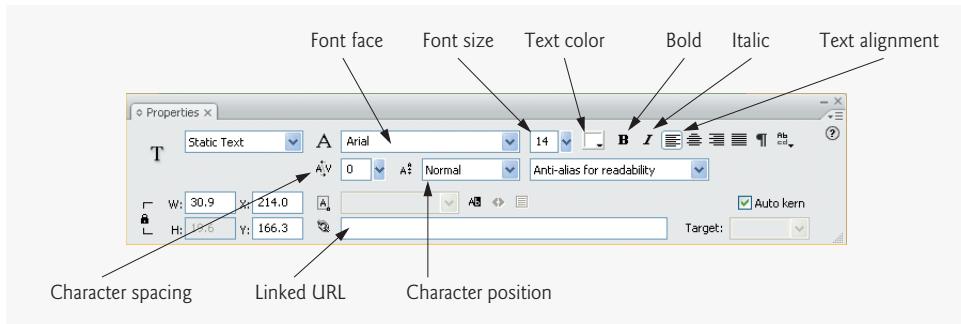
Click outside the circle with the Selection tool to deselect the circle. Now, select only the red fill with the Selection tool. Change the fill color by clicking the red radial gradient fill in the **Swatches** panel. The gradient fills are located at the bottom of the **Swatches** panel (Fig. 16.10). The circle should now have a red radial gradient fill with a black stroke surrounding it.



**Fig. 16.10** | Choosing a gradient fill.

### 16.3.2 Adding Text to a Button

Button titles communicate a button's function to the user. The easiest way to create a title is with the **Text** tool. Create a button title by selecting the **Text** tool and clicking the center of the button. Next, type **G0** in capital letters. Highlight the text with the **Text** tool. Once text is selected, you can change the font, text size and font color with the **Properties** window (Fig. 16.11). Select a sans-serif font, such as **Arial** or **Verdana**, from the font dropdown list. Set the font size to **14 pt** either by typing the size into the font size field or by pressing the arrow button next to it, revealing the **size selection slider**—a vertical slider that, when moved, changes the font size. Set the font weight to bold by clicking the bold



**Fig. 16.11** | Setting the font face, size, weight and color with the **Properties** window.

button (B). Finally, change the font color by clicking the text color swatch and selecting white from the palette.



### Look-and-Feel Observation 16.1

Sans-serif fonts, such as Arial, Helvetica and Verdana, are easier to read on a computer monitor, and therefore ensure better usability.

If the text does not appear in the correct location, drag it to the center of the button with the Selection tool. The button is almost complete and should look similar to Fig. 16.12.



**Fig. 16.12** | Adding text to the button.

### 16.3.3 Converting a Shape into a Symbol

A Flash movie consists of **scenes** and **symbols**. Each scene contains all graphics and symbols. The parent movie may contain several symbols that are reusable movie elements, such as **graphics**, **buttons** and **movie clips**. A scene timeline can contain numerous symbols, each with its own timeline and properties. A scene may have several **instances** of any given symbol (i.e., the same symbol can appear multiple times in one scene). You can edit symbols independently of the scene by using the symbol's **editing stage**. The editing stage is separate from the scene stage and contains only one symbol.



### Good Programming Practice 16.2

Reusing symbols can drastically reduce file size, thereby allowing faster downloads.

To make our button interactive, we must first convert the button into a button symbol. The button consists of distinct text, color fill and stroke elements on the parent

stage. These items are combined and treated as one object when the button is converted into a symbol. Use the Selection tool to drag a **selection box** around the button, selecting the button fill, the button stroke and the text all at one time (Fig. 16.13).

Now, select **Convert to Symbol...** from the **Modify** menu or use the shortcut *F8* on the keyboard. This opens the **Convert to Symbol** dialog, in which you can set the properties of a new symbol (Fig. 16.14).

Every symbol in a Flash movie must have a unique name. It is a good idea to name symbols by their contents or function, because this makes them easier to identify and reuse. Enter the name **go button** into the **Name** field of the **Convert to Symbol** dialog. The **Behavior** option determines the symbol's function in the movie.

You can create three different types of symbols—movie clips, buttons and graphics. A **movie clip symbol**'s behavior is similar to that of a scene and thus it is ideal for recurring animations. **Graphic symbols** are ideal for static images and basic animations. **Button symbols** are objects that perform button actions, such as **rollovers** and hyperlinking. A rollover is an action that changes the appearance of a button when the mouse passes over it. For this example, select **Button** as the type of symbol and click **OK**. The button should now be surrounded by a blue box with crosshairs in the upper-left corner, indicating that the button is a symbol. Also, in the **Properties** window panel, name this instance of the **go button** symbol **goButton** in the field containing **<Instance Name>**. Use the selection tool to drag the button to the lower-right corner of the stage.

The **Library** panel (Fig. 16.15) stores every symbol present in a movie and is accessed through the **Window** menu or by the shortcuts *<Ctrl>-L* or *F11*. Multiple instances of a symbol can be placed in a movie by dragging and dropping the symbol from the **Library** panel onto the stage.

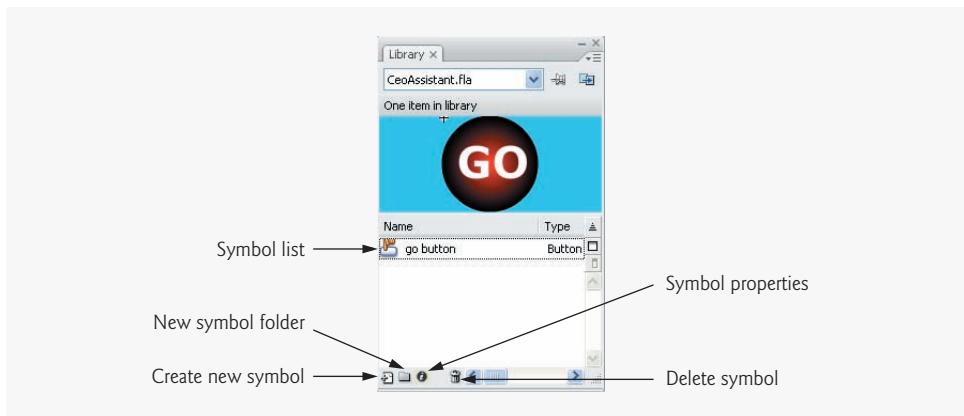
The **Movie Explorer** displays the movie structure and is accessed by selecting **Movie Explorer** from the **Window** menu or by pressing *<Alt>-F3* (Fig. 16.16). The **Movie Explorer** panel illustrates the relationship between the current scene (**Scene 1**) and its symbols.



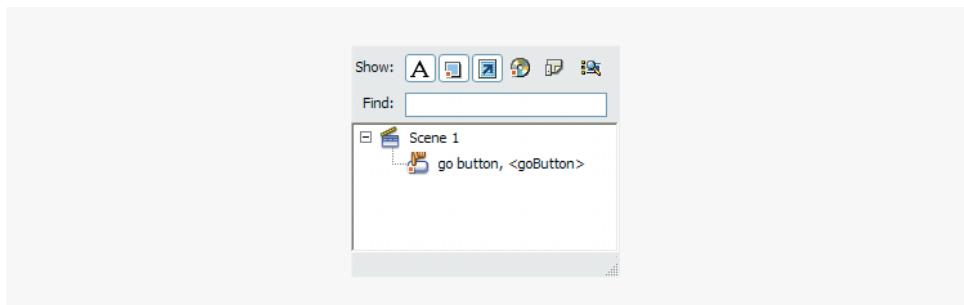
**Fig. 16.13** | Selecting an object with the selection tool.



**Fig. 16.14** | Creating a new symbol with the **Convert to Symbol** dialog.



**Fig. 16.15** | Library panel.



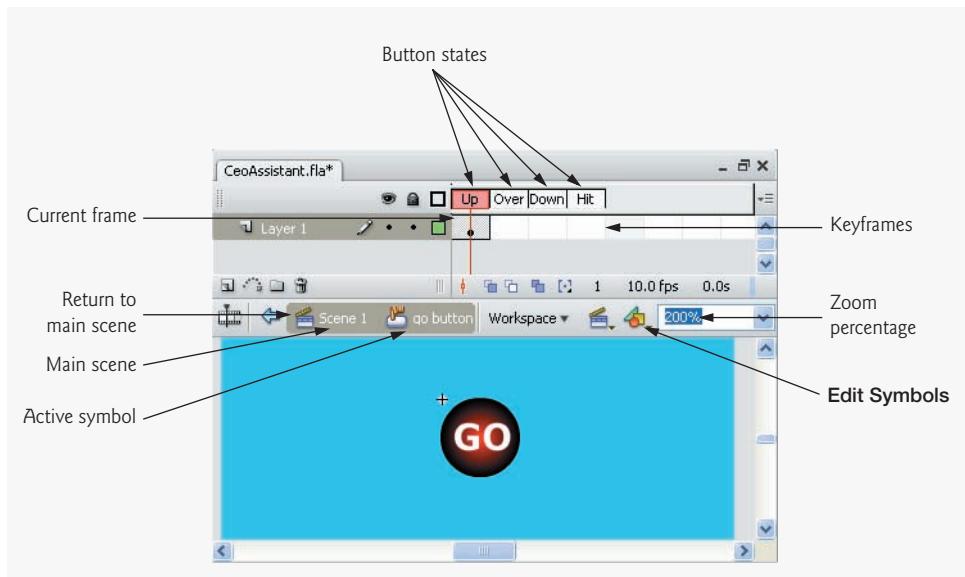
**Fig. 16.16** | Movie Explorer for CeoAssistant.fla.

#### 16.3.4 Editing Button Symbols

The next step in this example is to make the button symbol interactive. The different components of a button symbol, such as its text, color fill and stroke, may be edited in the symbol's editing stage, which you can access by double clicking the icon next to the symbol in the **Library**. A button symbol's timeline contains four frames, one for each of the button states (up, over and down) and one for the hit area.

The **up state** (indicated by the **Up** frame on screen) is the default state before the user presses the button or rolls over it with the mouse. Control shifts to the **over state** (i.e., the **Over** frame) when the user rolls over the button with the mouse cursor. The button's **down state** (i.e., the **Down** frame) plays when a user presses a button. You can create interactive, user-responsive buttons by customizing the appearance of a button in each of these states. Graphic elements in the **hit state** (i.e., the **Hit** frame) are not visible to a viewer of the movie; they exist simply to define the active area of the button (i.e., the area that can be clicked). The hit state will be discussed further in Section 16.6.

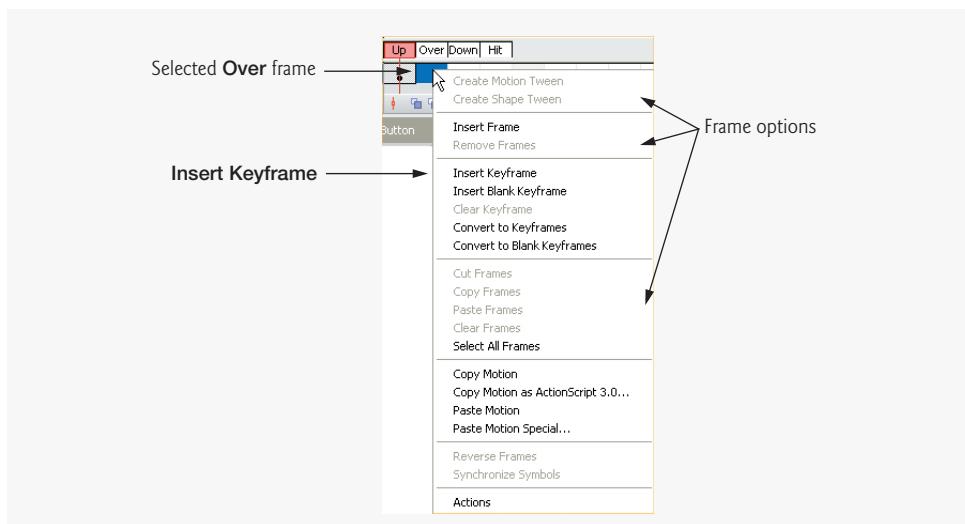
By default, buttons have only the up state activated when they are created. You may activate other states by adding keyframes to the other three frames. Keyframes for a button, discussed in the next section, determine how a button reacts when it is rolled over or clicked with the mouse.



**Fig. 16.17** | Modifying button states with a button's editing stage.

### 16.3.5 Adding Keyframes

Keyframes are points of change in a Flash movie and appear in the timeline with a dot. By adding keyframes to a button symbol's timeline, you can control how the button reacts to user interactions. The following step shows how to create a button rollover effect, which is accomplished by inserting a keyframe in the button's **Over** frame, then changing the button's appearance in that frame. Right click the **Over** frame and select **Insert Keyframe** from the resulting menu or press **F6** (Fig. 16.18).



**Fig. 16.18** | Inserting a keyframe.

Select the **Over** frame and click outside the button area with the selection tool to deselect the button's components. Change the color of the button in the **Over** state from red gradient fill to green gradient fill by selecting only the fill portion of the button with the Selection tool. Select the green gradient fill in the **Swatches** panel to change the color of the button in the **Over** state. Changing the color of the button in the over state does not affect the color of the button in the up state. Now, when the user moves the cursor over the button (in the up state) the button animation is replaced by the animation in the **Over** state. Here, we change only the button's color, but we could have created an entirely new animation in the **Over** state. The button will now change from red to green when the user rolls over the button with the mouse. The button will return to red when the mouse is no longer positioned over the button.

### 16.3.6 Adding Sound to a Button

The next step is to add a sound effect that plays when a user clicks the button. Flash imports sounds in the WAV (Windows), AIFF (Macintosh) or MP3 formats. Several button sounds are available free for download from sites such as Flashkit ([www.flashkit.com](http://www.flashkit.com)) and Muinar ([www.sounds.muinar.com](http://www.sounds.muinar.com)). For this example, download the **cash register** sound in WAV format from

[www.flashkit.com/soundfx/Industrial\\_Commercial/Cash](http://www.flashkit.com/soundfx/Industrial_Commercial/Cash)

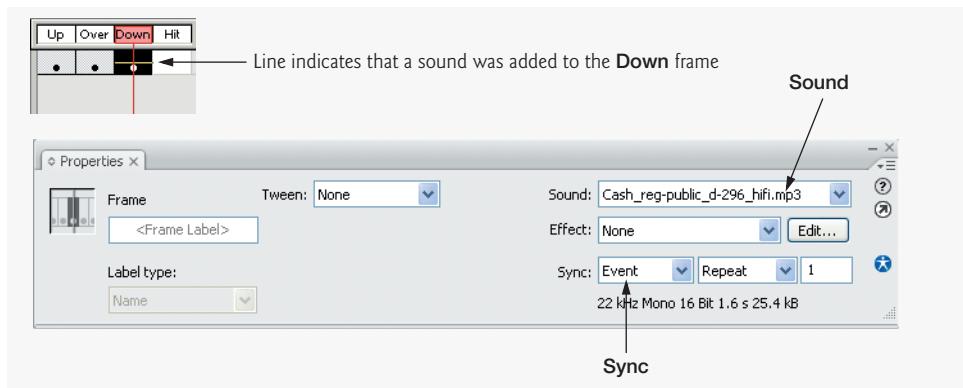
Click the **Download** link to download the sound from this site. This link opens a new web page from which the user chooses the sound format. Choose MP3 as the file format by clicking the **mp3** link. Save the file to the same folder as **CeoAssistant.fla**. Extract the sound file and save it in the same folder as **CeoAssistant.fla**.

Once the sound file is extracted, it can be imported into Flash. Import the sound into the **Library** by choosing **Import to Library...** from the **Import** submenu of the **File** menu. Select **All Formats** in the **Files of type** field of the **Import** dialog so that all available files are displayed. Select the sound file and press **Open**. This imports the sound file and places it in the movie's **Library**, making it available to use in the movie.

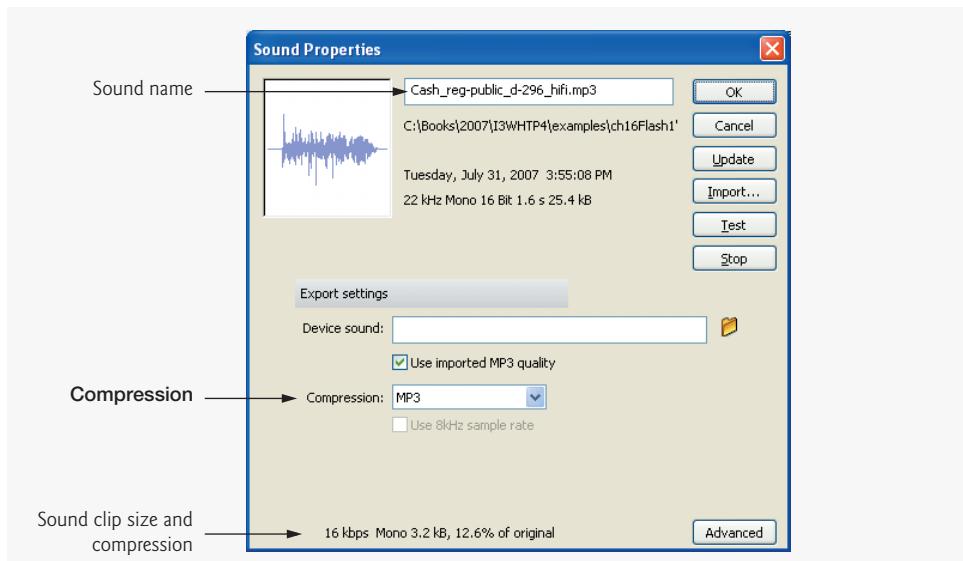
You can add sound to a movie by placing the sound clip in a keyframe or over a series of frames. For this example, we add the sound to the button's down state so that the sound plays when the user presses the button. Select the button's **Down** frame and press **F6** to add a keyframe.

Add the sound to the **Down** keyframe by dragging it from the **Library** to the stage. Open the **Properties** window (Fig. 16.19) and select the **Down** frame in the timeline to define the sound's properties in the movie. To ensure the desired sound has been added to the keyframe, choose the sound filename from the **Sound** drop-down list. This list contains all the sounds that have been added to the movie. Make sure the **Sync** field is set to **Event** so that the sound plays when the user clicks the button. If the **Down** frame has a blue wave or line through it, the sound effect has been added to the button.

Next, optimize the sound for the web. Double click the sound icon in the **Library** panel to open the **Sound Properties** dialog (Fig. 16.20). The settings in this dialog change the way that the sound is saved in the final movie. Different settings are optimal for different sounds and different audiences. For this example, set the **Compression** type to **MP3**, which reduces file size. Ensure that **Use imported MP3 quality** is selected. If the sound clip



**Fig. 16.19** | Adding sound to a button.



**Fig. 16.20** | Optimizing sound with the **Sound Properties** dialog.

is long, or if the source MP3 was encoded with a high bitrate, you may want to deselect this and specify your own bitrate to save space.

The sound clip is now optimized for use on the web. Return to the scene by pressing the **Edit Scene** button (  ) and selecting **Scene 1** or by clicking **Scene 1** at the top of the movie window.

### 16.3.7 Verifying Changes with Test Movie

It is a good idea to ensure that movie components function correctly before proceeding further with development. Movies can be viewed in their **published state** with the Flash Player. The published state of a movie is how it would appear if viewed over the web or with the Flash Player. Published Flash movies have the Shockwave Flash extension **.swf** (pronounced “swiff”). SWF files can be viewed but not edited.

Select **Test Movie** from the **Control** menu (or press **<Ctrl>-Enter**) to **export** the movie into the Flash Player. A window opens with the movie in its published state. Move the cursor over the **GO** button to view the color change (Fig. 16.21), then click the button to play the sound. Close the test window to return to the stage. If the button's color does not change, return to the button's editing stage and check that you followed the steps correctly.



**Fig. 16.21** | GO button in its up and over states.

### 16.3.8 Adding Layers to a Movie

The next step in this example is to create the movie's title animation. It's a good idea for you to create a new **layer** for new movie items. A movie can be composed of many layers, each having its own attributes and effects. Layers organize movie elements so that they can be animated and edited separately, making the composition of complex movies easier. Graphics in higher layers appear over the graphics in lower layers.

Before creating a new title layer, double click the text **Layer 1** in the timeline. Rename the layer by entering the text **Button** into the name field (Fig. 16.22).

Create a new layer for the title animation by clicking the **Insert a new layer** button or by selecting **Layer** from the **Timeline** submenu of the **Insert** menu. The **Insert a new layer** button places a layer named **Layer 2** above the selected layer. Change the name of **Layer 2** to **Title**. Activate the new layer by clicking its name.



#### Good Programming Practice 16.3

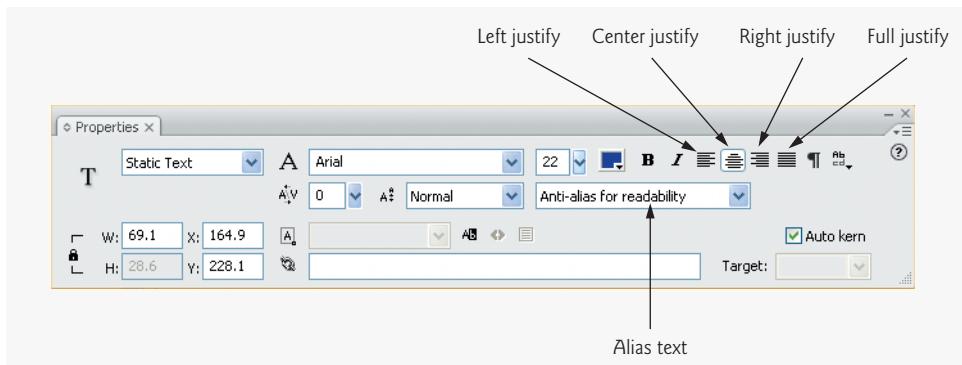
*Always give movie layers descriptive names. Descriptive names are especially helpful when working with many layers.*

Select the **Text** tool to create the title text. Click with the **Text** tool in the center of the stage toward the top. Use the **Property** window to set the font to **Arial**, the text color to navy blue (hexadecimal value **#000099**) and the font size to **20 pt** (Fig. 16.23). Set the text alignment to center by clicking the center justify button.

Type the title **CEO Assistant 1.0** (Fig. 16.24), then click the selection tool. A blue box appears around the text, indicating that it is a **grouped object**. This text is a grouped object because each letter is a part of a text string and cannot be edited independently. Text can be broken apart for color editing, shape modification or animation (shown in a later example). Once text has been broken apart, it may not be edited with the **Text** tool.



**Fig. 16.22** | Renaming a layer.



**Fig. 16.23** | Setting text alignment with the **Properties** window.



**Fig. 16.24** | Creating a title with the **Text** tool.

### 16.3.9 Animating Text with Tweening

Animations in Flash are created by inserting keyframes into the timeline. Each keyframe represents a significant change in the position or appearance of the animated object.

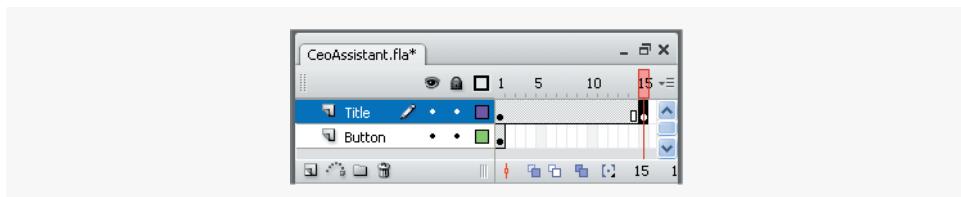
You may use several methods to animate objects in Flash. One is to create a series of successive keyframes in the timeline. Modifying the animated object in each keyframe creates an animation as the movie plays. Another method is to insert a keyframe later in the timeline representing the final appearance and position of the object, then create a tween between the two keyframes. **Tweening** is an automated process in which Flash creates the intermediate steps of the animation between two keyframes.

Flash provides two tweening methods. **Shape tweening** morphs an object from one shape to another. For instance, the word "star" could morph into the shape of a star. Shape tweening can be applied only to ungrouped objects, not symbols or grouped objects. Be sure to break apart text before attempting to create a shape tween. **Motion tweening** moves objects around the stage. Motion tweening can be applied to symbols or grouped objects.

You can only have one symbol per layer if you intend to tween the symbol. At this point in the development of the example movie, only frame 1 is occupied in each layer. Keyframes must be designated in the timeline before adding the motion tween. Click frame 15 in the **Title** layer and press **F6** to add a new keyframe. All the intermediate frames in the timeline should turn gray, indicating that they are active (Fig. 16.25). Until the motion tween is added, each active frame contains the same image as the first frame.

The button disappears from the movie after the first frame because only the first frame is active in the button layer. Before the movie is completed, we'll move the button to frame 15 of its layer so that the button appears once the animation stops.

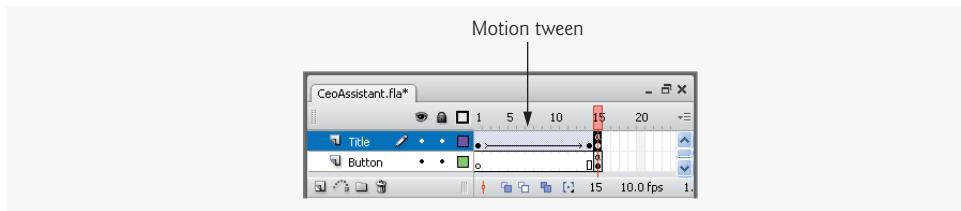
We now create a motion tween by modifying the position of the title text. Select frame 1 of the **Title** layer and select the title text with the Selection tool. Drag the title text directly above the stage. When the motion tween is added, the title will move onto the stage. Add



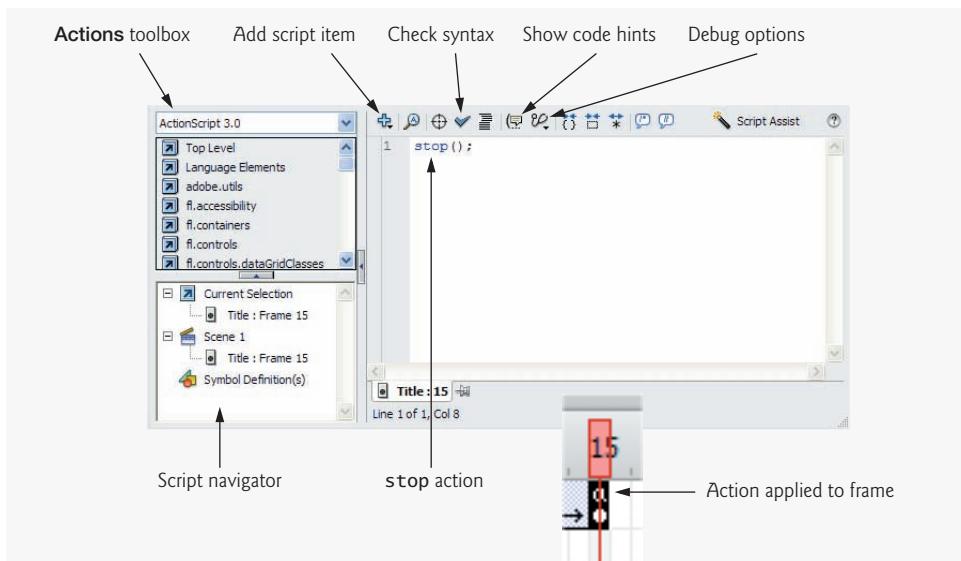
**Fig. 16.25** | Adding a keyframe to create an animation.

the motion tween by right clicking frame 1 in the **Title** layer. Then select **Create Motion Tween** from the **Insert > Timeline** menu. Tweens also can be added using the **Tween type** drop down menu in the **Properties** window. Frames 2–14 should turn light blue, with an arrow pointing from the keyframe in frame 1 to the keyframe in frame 15 (Fig. 16.26).

Test the movie again with the Flash Player by pressing *<Ctrl>-Enter* to view the new animation. Note that the animation continues to loop—all Flash movies loop by default. Adding the ActionScript function **stop** to the last frame in the movie stops the movie from looping. For this example, click frame 15 of the **Title** layer, and open the **Actions** panel by selecting **Window > Actions** or by pressing *F9* (Fig. 16.27). The **Actions** panel is used to add



**Fig. 16.26** | Creating a motion tween.



**Fig. 16.27** | Adding ActionScript to a frame with the **Actions** panel.

actions (i.e., scripted behaviors) to symbols and frames. Here, add `stop()`; so that the movie does not loop back to the first frame.

Minimize the **Actions** panel by clicking the down arrow in its title bar. The small letter **a** in frame 15 of the **Title** layer indicates the new action. Test the movie again in Flash Player. Now, the animation should play only once.

The next step is to move the button to frame 15 so that it appears only at the end of the movie. Add a keyframe to frame 15 of the **Button** layer. A copy of the button should appear in the new keyframe. Select the button in the first frame and delete it by pressing the *Delete* key. The button will now appear only in the keyframe at the end of the movie.

### 16.3.10 Adding a Text Field

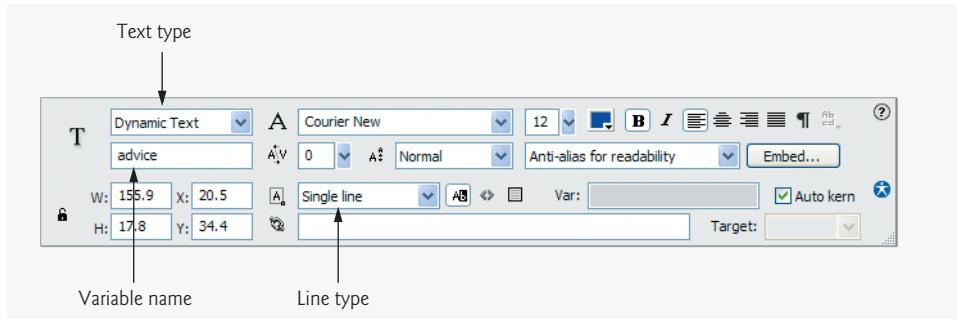
The final component of our movie is a **text field**, which contains a string of text that changes every time the user presses the button. An instance name is given to the text field so that ActionScript added to the button can control its contents.

Create a layer named **Advice** for the new text field, and add a keyframe to frame 15 of the **Advice** layer. Select the Text tool and create the text field by dragging with the mouse in the stage (Fig. 16.28). Place the text field directly below the title. Set the text font to **Courier New, 12 pt** and the style to bold in the **Properties** window. You can alter the size of the text field by dragging the **anchor** that appears in its upper-right corner.

You'll now assign an instance name to the text field. Select the text field and open the **Properties** window (Fig. 16.29). The **Properties** window contains several options for modifying text fields. The top-left field contains the different types of text fields. **Static Text**, the default setting for this panel, creates text that does not change. The second option,



**Fig. 16.28** | Creating a text field.



**Fig. 16.29** | Creating a dynamic text field with the **Properties** window.

**Dynamic Text**, creates text that can be changed or determined by outside variables through ActionScript. When you select this text type, new options appear below this field. The **Line type** drop-down list specifies the text field size as either a single line or multiple lines of text. The **Instance Name** field allows you to give the text field an instance name by which it can be referenced in script. For example, if the text field instance name is `newText`, you could write a script setting `newText.text` equal to a string or a function output. The third text type, **Input Text**, creates a text field into which the viewers of the movie can input their own text. For this example, select **Dynamic Text** as the text type. Set the line type to **Single Line** and enter `advice` as the instance name. This instance name will be used in ActionScript later in this example.

### 16.3.11 Adding ActionScript

All the movie objects are now in place, so CEO Assistant 1.0 is almost complete. The final step is to add ActionScript to the button, enabling the script to change the contents of the text field every time a user clicks the button. Our script calls a built-in Flash function to generate a random number. This random number corresponds to a message in a list of possible messages to display. [Note: The ActionScript in this chapter has been formatted to conform with the code-layout conventions of this book. The Flash application may produce code that is formatted differently.]

Select frame 15 of the **Button** layer and open the **Actions** panel. We want the action to occur when the user clicks the button. To achieve this, insert the statement:

```
goButton.addEventListener(MouseEvent.MOUSE_DOWN, goFunction);
```

This statement uses the button object's instance name (`goButton`) to call the `addEventListener` function, which registers an event handler (`goFunction` in this example) that will be called when the event takes place (i.e., when you click the button). The first argument, `MouseEvent.MOUSE_DOWN`, specifies that an action is performed when the user presses the button with the mouse.

The next step is to add the function that handles this event. Create a new function named `goFunction` by using the code

```
function goFunction(event : MouseEvent) : void
{
} // end function goFunction
```

The function's one parameter is a `MouseEvent`, implying that the function has to be provided with a mouse action to be accessed. The function does not return anything, hence the `void` return value. Inside this function, add the following statement:

```
var randomNumber : int = Math.floor((Math.random() * 5));
```

which creates an integer variable called `randomNumber` and assigns it a random value. For this example, we use the `Math.random` function to choose a random number from 0 to 1. `Math.random` returns a random floating-point number from 0.0 up to, but not including, 1.0. Then, it is scaled accordingly, depending on what the range should be. Since we want all the numbers between 0 and 4, inclusive, the value returned by the `Math.random` should be multiplied by 5 to produce a number in the range 0.0 up to, but not including, 5.0.

Finally, this new number should be rounded down to the largest integer smaller than itself, using the `Math.floor` function.



### Error-Prevention Tip 16.1

*ActionScript is case sensitive. Be aware of the case when entering arguments or variable names.*

The value of `randomNumber` determines the text string that appears in the text field. A `switch` statement sets the text field's value based on the value of `randomNumber`. [Note: For more on `switch` statements, refer to Chapter 8.] On a new line in the `goFunction` function, insert the following `switch` statement:

```
switch (randomNumber)
{
 case 0:
 advice.text = "Hire Someone!";
 break;
 case 1:
 advice.text = "Buy a Yacht!";
 break;
 case 2:
 advice.text = "Buy stock!";
 break;
 case 3:
 advice.text = "Go Golfing!";
 break;
 case 4:
 advice.text = "Hold a meeting!";
 break;
} // end switch
```

This statement displays different text in the `advice` text field based on the value of the variable `randomNumber`. The text field's `text` property specifies the text to display. If you feel ambitious, increase the number of `advice` statements by producing a larger range of random values and adding more cases to the `switch` statement. Minimize the **Actions** panel to continue.

Congratulations! You have now completed building CEO Assistant 1.0. Test the movie by pressing `<Ctrl>-Enter` and clicking the **GO** button. After testing the movie with the Flash Player, return to the main window and save the file.

## 16.4 Publishing Your Flash Movie

Flash movies must be **published** for users to view them outside the Flash CS3 environment and Flash Player. This section discusses the more common methods of publishing Flash movies. For this example, we want to publish in two formats, Flash and **Windows Projector**, which creates a standard Windows-executable file that works even if the user hasn't installed Flash. Select **Publish Settings...** from the **File** menu to open the **Publish Settings** dialog.

Select the **Flash**, **HTML** and **Windows Projector** checkboxes and uncheck all the others. Then click the **Flash** tab at the top of the dialog. This section of the dialog allows you to choose the Flash settings. Flash movies may be published in an older Flash version if you

wish to support older Flash Players. Note that ActionScript 3.0 is not supported by older players, so choose a version with care. Publish the movie by clicking **Publish** in the **Publish Settings** dialog or by selecting **Publish** from the **File** menu. After you've published the movie, the directory in which you saved the movie will have several new files (Fig. 16.30). If you wish to place your movie on a website, be sure to copy the HTML, JavaScript and SWF files to your server.



### Good Programming Practice 16.4

*It is not necessary to transfer the .fla version of your Flash movie to a web server unless you want other users to be able to download the editable version of the movie.*

As we can see in the Ceo Assistant 1.0 example, Flash is a feature-rich program. We have only begun to use Flash to its full potential. ActionScript can create sophisticated programs and interactive movies. It also enables Flash to interact with ASP.NET (Chapter 25), PHP (Chapter 23), and JavaScript (Chapters 6–11), making it a program that integrates smoothly into a web environment.

|                                                       |                           |          |                     |
|-------------------------------------------------------|---------------------------|----------|---------------------|
| JavaScript file for browser and Flash detection (.js) | AC_RunActiveContent.js    | 9 KB     | JScript Script File |
| Windows Executable (.exe)                             | Cash_reg-public_d-296.wav | 66 KB    | Wave Sound          |
| Flash (.fla)                                          | CeoAssistant.exe          | 2,420 KB | Application         |
| HTML document to view movie in browser                | CeoAssistant.fla          | 128 KB   | Flash Document      |
| Flash Player Movie (.swf)                             | CeoAssistant.html         | 3 KB     | HTML Document       |
|                                                       | CeoAssistant.swf          | 30 KB    | Flash Movie         |

**Fig. 16.30** | Published Flash files.

## 16.5 Creating Special Effects with Flash

The following sections introduce several Flash special effects. The preceding example familiarized you with basic movie development. The next sections cover many additional topics, from importing bitmaps to creating splash screens that display before a web page loads.

### 16.5.1 Importing and Manipulating Bitmaps

Some of the examples in this chapter require importing bitmapped images and other media into a Flash movie. The importing process is similar for all types of media, including images, sound and video. The following example shows how to import an image into a Flash movie.

Begin by creating a new Flash document. The image we are going to import is located in the Chapter 16 examples folder. Select **File > Import > Import to Stage...** (or press **<Ctrl>-R**) to display the **Import** dialog. Browse to the folder on your system containing this chapter's examples and open the folder labeled **images**. Select **bug.bmp** and click **OK** to continue. A bug image should appear on the stage. The **Library** panel stores imported images. You can convert imported images into editable shapes by selecting the image and pressing **<Ctrl>-B** or by choosing **Break Apart** from the **Modify** menu. Once an imported image is broken apart, it may be shape tweened or edited with **editing tools**, such as the

Lasso, Paint bucket, Eraser and Paintbrush. The editing tools are found in the toolbox and apply changes to a shape.

Dragging with the **Lasso tool** selects areas of shapes. The color of a selected area may be changed or the selected area may be moved. Click and drag with the Lasso tool to draw the boundaries of the selection. As with the button in the last example, when you select a shape area, a mesh of white dots covers the selection. Once an area is selected, you may change its color by selecting a new fill color with the fill swatch or by clicking the selection with the Paint bucket tool. The Lasso tool has different options (located in the **Options** section of the toolbox) including **Magic wand** and **Polygon mode**. The Magic wand option changes the Lasso tool into the Magic wand tool, which selects areas of similar colors. The polygonal lasso selects straight-edged areas.

The **Eraser tool** removes shape areas when you click and drag the tool across an area. You can change the eraser size using the tool options. Other options include settings that make the tool erase only fills or strokes.

The **Brush tool** applies color in the same way that the eraser removes color. The paintbrush color is selected with the fill swatch. The paintbrush tool options include a **Brush mode** option. These modes are **Paint behind**, which sets the tool to paint only in areas with no color information; **Paint selection**, which paints only areas that have been selected; and **Paint inside**, which paints inside a line boundary.

Each of these tools can create original graphics. Experiment with the different tools to change the shape and color of the imported bug graphic.



### Portability Tip 16.1

---

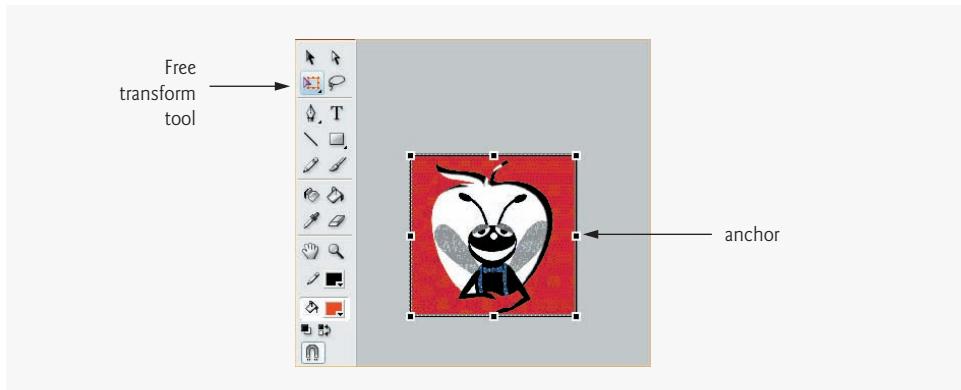
*When building Flash movies, use the smallest possible file size and web-safe colors to ensure that most people can view the movie regardless of bandwidth, processor speed or monitor resolution.*

## 16.5.2 Creating an Advertisement Banner with Masking

Masking hides portions of layers. A **masking layer** hides objects in the layers beneath it, revealing only the areas that can be seen through the shape of the mask. Items drawn on a masking layer define the mask's shape and cannot be seen in the final movie. The next example, which builds a website banner, shows how to use masking frames to add animation and color effects to text.

Create a new Flash document and set the size of the stage to **470** pixels wide by **60** pixels high. Create three layers named **top**, **middle** and **bottom** according to their positions in the layer hierarchy. These names help track the masked layer and the visible layers. The **top** layer contains the mask, the **middle** layer becomes the masked animation and the **bottom** layer contains an imported bitmapped logo. Import the graphic *bug\_apple.bmp* (from the *images* folder in this chapter's examples folder) into the first frame of the **top** layer, using the method described in the preceding section. This image will appear too large to fit in the stage area. Select the image with the selection tool and align it with the upper-left corner of the stage. Then select the **Free transform** tool in the toolbox (Fig. 16.31).

The Free transform tool allows us to resize an image. When an object is selected with this tool, anchors appear around its corners and sides. Click and drag an anchor to resize the image in any direction. Holding the *Shift* key while dragging a corner anchor ensures



**Fig. 16.31** | Resizing an image with the Free transform tool.

that the image maintains the original height and width ratio. Hold down the **Shift** key while dragging the lower-right anchor upward until the image fits on the stage.

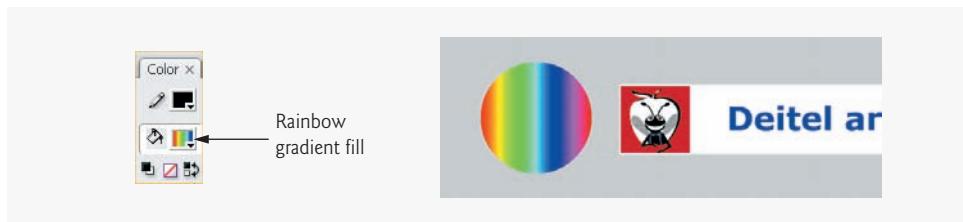
Use the text tool to add text to frame 1 of the **top** layer. Use **Verdana**, 28 pt bold, as the font. Select a blue text color, and make sure that **Static Text** is selected in the **Properties** window. Type the banner text “Deitel and Associates”, making sure that the text fits inside the stage area, and use the Selection tool to position the text next to the image. This text becomes the object that masks an animation.

We must convert the text into a shape before using it as a mask. Click the text field with the Selection tool to ensure that it is active and select **Break Apart** twice from the **Modify** menu. Breaking the text apart once converts each letter into its own text field. Breaking it apart again converts the letters into shapes that cannot be edited with the text tool, but can be manipulated as regular graphics.

Copy the contents of the **top** layer to the **bottom** layer before creating the mask, so that the text remains visible when the mask is added. Right click frame 1 of the **top** layer, and select **Copy Frames** from the resulting menu. Paste the contents of the **top** layer into frame 1 of the **bottom** layer by right clicking frame 1 of the **bottom** layer and selecting **Paste Frames** from the menu. This shortcut pastes the frame’s contents in the same positions as the original frame. Delete the extra copy of the bug image by selecting the bug image in the **top** layer with the selection tool and pressing the **Delete** key.

Next, you’ll create the animated graphic that the banner text in the **top** layer masks. Click in the first frame of the **middle** layer and use the Oval tool to draw a circle to the left of the image that is taller than the text. The oval does not need to fit inside the banner area. Set the oval stroke to **no color** by clicking the stroke swatch and selecting the **No color** option. Set the fill color to the rainbow gradient (Fig. 16.32), found at the bottom of the **Swatches** panel.

Select the oval by clicking it with the Selection tool, and convert the oval to a symbol by pressing **F8**. Name the symbol **oval** and set the behavior to **Graphic**. When the banner is complete, the oval will move across the stage; however, it will be visible only through the text mask in the **top** layer. Move the oval just outside the left edge of the stage, indicating the point at which the oval begins its animation. Create a keyframe in frame 20 of the **middle** layer and another in frame 40. These keyframes indicate the different locations of the **oval** symbol during the animation. Click frame 20 and move the oval just outside

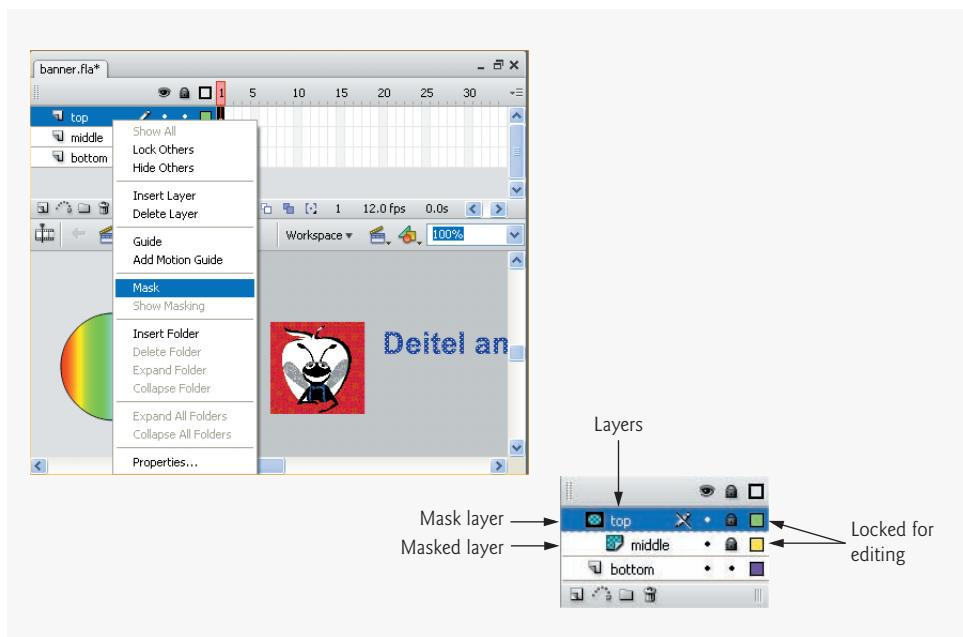


**Fig. 16.32** | Creating the oval graphic.

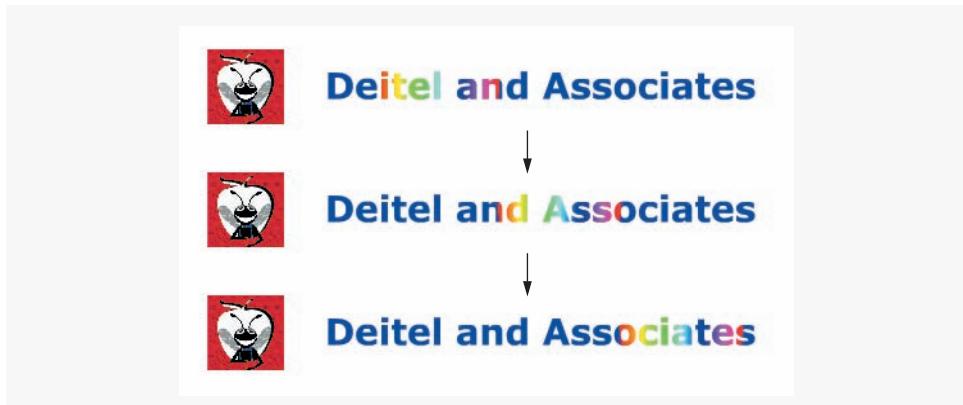
the right side of the stage to indicate the animation's next key position. Do not move the position of the **oval** graphic in frame 40, so that the oval will return to its original position at the end of the animation. Create the first part of the animation by right clicking frame 1 of the **middle** layer and choosing **Create Motion Tween** from the menu. Repeat this step for frame 20 of the **middle** layer, making the **oval** symbol move from left to right and back. Add keyframes to frame 40 of both the **top** and **bottom** layers so that the other movie elements appear throughout the movie.

Now that all the supporting movie elements are in place, the next step is to apply the masking effect. To do so, right click the **top** layer and select **Mask** (Fig. 16.33). Adding a mask to the **top** layer masks only the items in the layer directly below it (the **middle** layer), so the bug logo in the **bottom** layer remains visible at all times. Adding a mask also locks the **top** and **middle** layers to prevent further editing.

Now that the movie is complete, save it as **banner.fla** and test it with the Flash Player. The rainbow oval is visible through the text as it animates from left to right. The text in the bottom layer is visible in the portions not containing the rainbow (Fig. 16.34).



**Fig. 16.33** | Creating a mask layer.



**Fig. 16.34** | Completed banner.

### 16.5.3 Adding Online Help to Forms

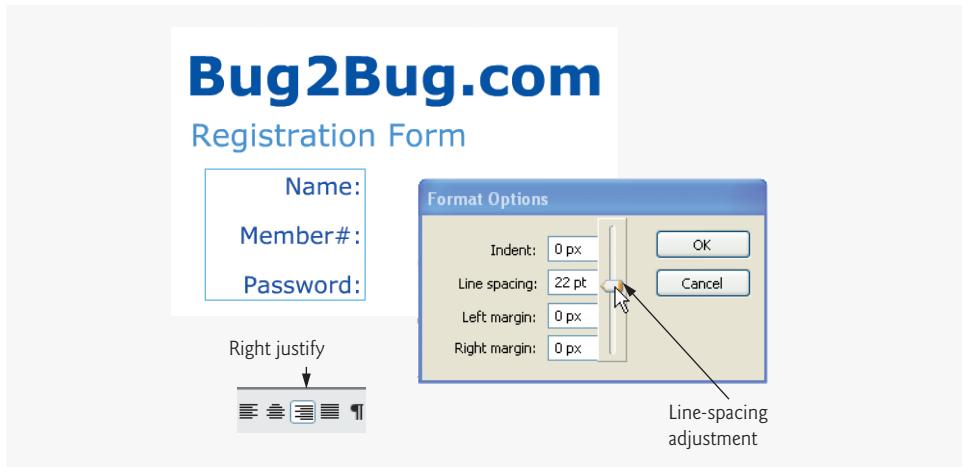
In this section, we build on Flash techniques introduced earlier in this chapter, including tweening, masking, importing sound and bitmapped images, and writing ActionScript. In the following example, we apply these various techniques to create an online form that offers interactive help. The interactive help consists of animations that appear when a user presses buttons located next to the form fields. Each button contains a script that triggers an animation, and each animation provides the user with information regarding the form field that corresponds to the pressed button.

Each animation is a movie-clip symbol that is placed in a separate frame and layer of the scene. Adding a `stop` action to frame 1 pauses the movie until the user presses a button.

Begin by creating a new movie, using default movie size settings. Set the frame rate to 24 fps. The first layer will contain the site name, form title and form captions. Change the name of **Layer 1** to **text**. Add a `stop` action to frame 1 of the text layer. Create the site name `Bug2Bug.com` as static text in the **text** layer using a large, bold font, and place the title at the top of the page. Next, place the form name `Registration Form` as static text beneath the site name, using the same font, but in a smaller size and different color. The final text element added to this layer is the text box containing the form labels. Create a text box using the **Text Tool**, and enter the text: `Name:`, `Member #:` and `Password:`, pressing `Enter` after entering each label to put it on a different line. Next, adjust the value of the **Line Spacing** field (the amount of space between lines of text) found by clicking the **Edit Format Options** button ( ) in the **Properties** window. Change the form field caption line spacing to 22 in the **Format Options** dialog (Fig. 16.35) and set the text alignment (found in the **Properties** window) to right justify.

Now we'll create the form fields for our help form. The first step in the production of these form fields is to create a new layer named **form**. In the **form** layer, draw a rectangle that is roughly the same height as the caption text. This rectangle will serve as a background for the form text fields (Fig. 16.36). We set a **Rectangle corner radius** of 6 px in the **Properties** panel. Feel free to experiment with other shapes and colors.

The next step is to convert the rectangle into a symbol so that it may be reused in the movie. Select the rectangle fill and stroke with the selection tool and press `F8` to convert the selection to a symbol. Set the symbol behavior to **Graphic** and name the symbol **form**.



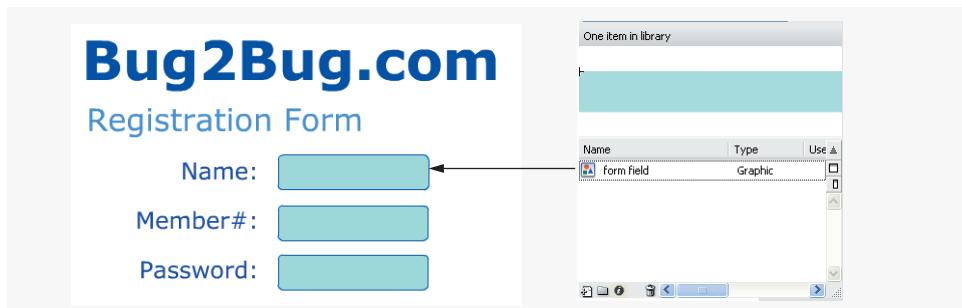
**Fig. 16.35** | Adjusting the line spacing with the **Format Options** dialog.



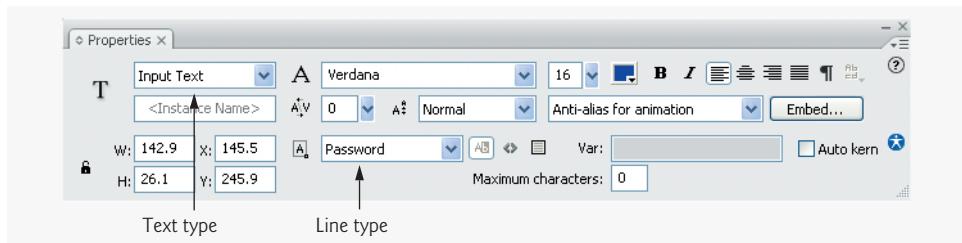
**Fig. 16.36** | Creating a rectangle with rounded corners.

**field.** This symbol should be positioned next to the **Name:** caption. When the symbol is in place, open the **Library** panel by pressing **<Ctrl>-L**, select the **form** layer and drag two copies of the **form field** symbol from the **Library** onto the stage. This will create two new instances of this symbol. Use the Selection tool to align the fields with their corresponding captions. For more precise alignment, select the desired object with the Selection tool and press the arrow key on the keyboard in the direction you want to move the object. After alignment of the **form field** symbols, the movie should resemble Fig. 16.37.

We now add **input text fields** to our movie. An input text field is a text field into which the user can enter text. Select the Text tool and, using the **Properties** window, set the font to **Verdana, 16 pt**, with dark blue as the color. In the **Type** pull-down menu in the **Properties** window, select **Input Text** (Fig. 16.38). Then, click and drag in the stage to create a text field slightly smaller than the **form field** symbol we just created. With the Selection tool, position the text field over the instance of the **form field** symbol associated with the name. Create a similar text field for member number and password. Select the Password text field, and select **Password** in the **Type** pull-down menu in the **Properties** window. Selecting **Password** causes any text entered into the field by the user to appear as an asterisk (\*). We have now created all the input text fields for our help form. In this



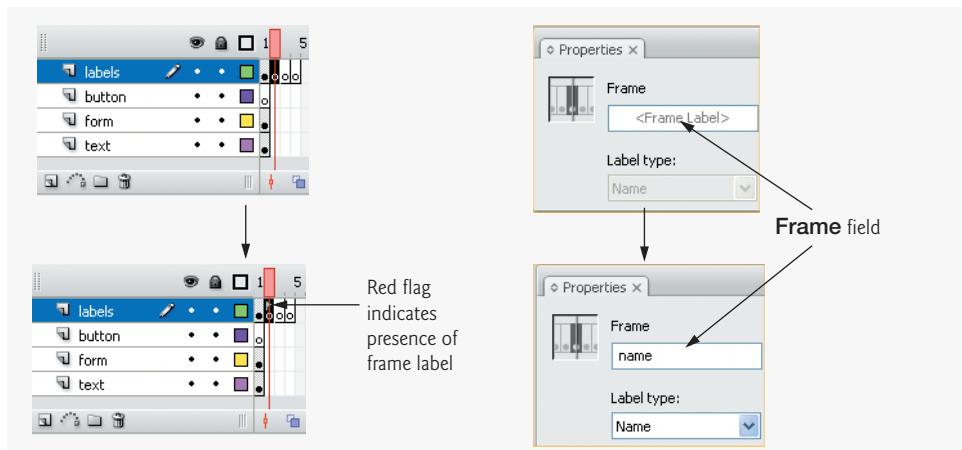
**Fig. 16.37** | Creating multiple instances of a symbol with the **Library** panel.



**Fig. 16.38** | Input and password text-field creation.

example, we won't actually process the text entered into these fields. Using ActionScript, we could give each input text field a variable name, and send the values of these variables to a server-side script for processing.

Now that the form fields are in place, we can create the help associated with each field. Add two new layers. Name one layer **button** and the other **labels**. The **labels** layer will hold the **frame label** for each keyframe. A frame label is a text string that corresponds to a specific frame or series of frames. In the **labels** layer, create keyframes in frames 2, 3 and 4. Select frame 2 and enter name into the **Frame** field in the **Properties** window (Fig. 16.39).



**Fig. 16.39** | Adding **Frame Labels** using the **Properties** window.

Name frame 3 and frame 4 `memberNumber` and `password`, respectively. These frames can now be accessed either by number or by name. We use the labels again later in this example.

In frame 1 of the **button** layer, create a small circular button containing a question mark. [Note: the **Text** type property of the **Text Tool** will still be **Input Text**, so you must change it back to **Static Text**.] Position it next to the `name` field. When the button is complete, select all of its pieces with the selection tool, and press *F8* to convert the shape into a button symbol named `helpButton`. Drag two more instances of the `helpButton` symbol from the **Library** panel onto the stage next to each of the form fields.

These buttons will trigger animations that provide information about their corresponding form fields. A script will be added to each button so that the movie jumps to a particular frame when a user presses the button. Click the `helpButton` symbol associated with the `name` field and give it the instance name `nameHelp`. As in Section 16.3.11, we'll now add event-handling code. Open the **Actions** for the first frame of the **buttons** layer and invoke the `nameHelp` button's `addEventListener` function to register the function `nameFunction` as the handler of the mouse-click event. In `nameFunction`, add a `gotoAndStop` action, causing the movie play to skip to a particular frame and stop playing. Enter "name" between the function's parentheses. The script should now read as follows:

```
nameHelp.addEventListener(MouseEvent.MOUSE_DOWN, nameFunction);

function nameFunction(event : MouseEvent) : void
{
 gotoAndStop("name");
}
```

When the user presses the `nameHelp` button, this script advances to the frame labeled `name` and stops. [Note: We could also have entered `gotoAndStop( 2 )`, referenced by frame number, in place of `gotoAndStop( "name" )`.] Add similar code for the `memberHelp` and `passwordHelp` buttons, changing the frame labels to `memberNumber` and `password`, the button instance names to `memberHelp` and `passwordHelp` and the function names to `memberFunction` and `passwordFunction`, respectively. Each button now has an action that points to a distinct frame in the timeline. We next add the interactive help animations to these frames.

The animation associated with each button is created as a movie-clip symbol that is inserted into the scene at the correct frame. For instance, the animation associated with the **Password** field is placed in frame 4, so that when the button is pressed, the `gotoAndStop` action skips to the frame containing the correct animation. When the user clicks the button, a help rectangle will open and display the purpose of the associated field.

Each movie clip should be created as a **new symbol** so that it can be edited without affecting the scene. Select **New Symbol...** from the **Insert** menu (or use the shortcut *<Ctrl>-F8*), name the symbol `nameWindow` and set the behavior to **Movie Clip**. Press **OK** to open the symbol's stage and timeline.

Next, you'll create the interactive help animation. This animation contains text that describes the form field. Before adding the text, we are going to create a small background animation that we will position behind the text. Begin by changing the name of **Layer 1** to **background**. Draw a dark blue rectangle with no border. This rectangle can be of any size, because we will customize its proportions with the **Properties** window. Select the rectangle

with the Selection tool, then open the **Properties** window. Set the **W:** field to **200** and the **H:** field to **120**, to define the rectangle's size. Next, center the rectangle by entering **-100** and **-60** into the **X:** and **Y:** fields, respectively (Fig. 16.40).

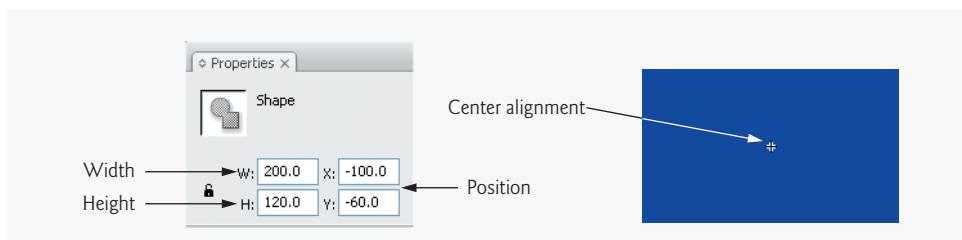
Now that the rectangle is correctly positioned, we can begin to create its animation. Add keyframes to frames 5 and 10 of the **background** layer. Use the **Properties** window to change the size of the rectangle in frame 5, setting its height to **5.0**. Next, right click frame 5 and select **Copy Frames**. Then right click frame 1 and select **Paste Frames**. While in frame 1, change the width of the rectangle to **5.0**.

The animation is created by applying shape tweening to frames 1 and 5. Recall that shape tweening morphs one shape into another. The shape tween causes the dot in frame 1 to grow into a line by frame 5, then into a rectangle in frame 10. Select frame 1 and apply the shape tween by right clicking frame 1 and selecting **Create Shape Tween**. Do the same for frame 5. Shape tweens appear green in the timeline (Fig. 16.41). Follow the same procedure for frame 5.

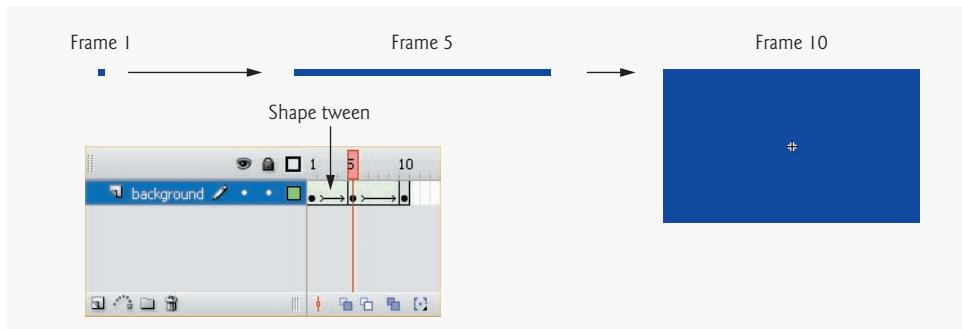
Now that this portion of the animation is complete, it may be tested on the stage by pressing *Enter*. The animation should appear as the dot from frame 1 growing into a line by frame 5 and into a rectangle by frame 10.

The next step is to add a mock form field to this animation which demonstrates what the user would type in the actual field. Add two new layers above the **background** layer, named **field** and **text**. The **field** layer contains a mock form field, and the **text** layer contains the help information.

First, create an animation similar to the growing rectangle we just created for the mock form field. Add a keyframe to frame 10 in both the **field** and **text** layers. Fortunately, we have a form field already created as a symbol. Select frame 10 of the **field** layer, and drag



**Fig. 16.40** | Centering an image on the stage with the **Properties** window.



**Fig. 16.41** | Creating a shape tween.

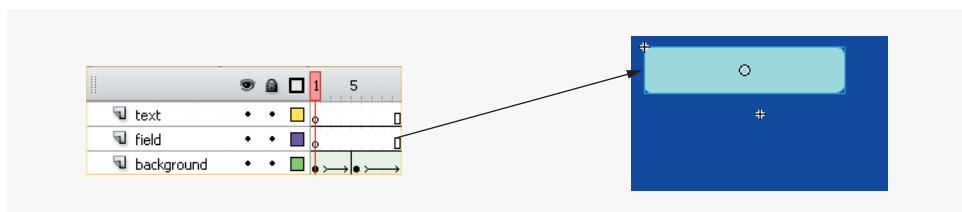
the **form field** symbol from the **Library** panel onto the stage, placing it within the current movie clip. Symbols may be embedded in one another; however, they cannot be placed within themselves (i.e., an instance of the **form field** symbol cannot be dragged onto the **form field** symbol editing stage). Align the **form field** symbol with the upper-left corner of the background rectangle, as shown in Fig. 16.42.

Next, set the end of this movie clip by adding keyframes to the **background** and **field** layers in frame 40. Also add keyframes to frames 20 and 25 of the **field** layer. These keyframes define intermediate points in the animation. Refer to Fig. 16.43 for correct keyframe positioning.

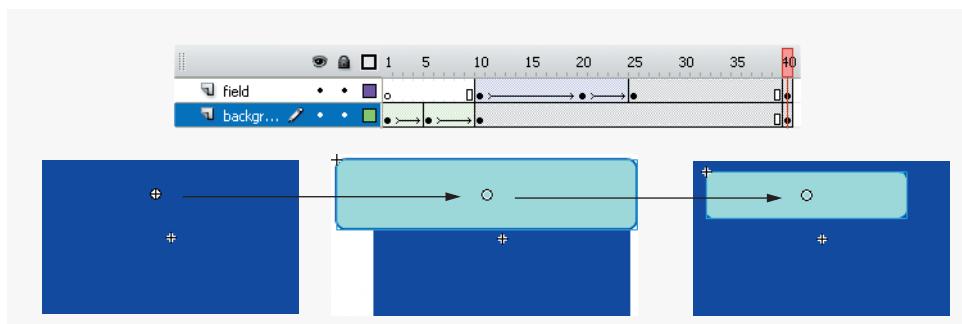
The next step in creating the animation is to make the **form field** symbol grow in size. Select frame 20 of the **field** layer, which contains only the **form field** symbol. Next open the **Transform** panel from the **Window** menu. The **Transform** panel can be used to change an object's size. Check the **Constrain** checkbox to constrain the object's proportions as it is resized. Selecting this option causes the **scale factor** to be equal in the width and height fields. The scale factor measures the change in proportion. Set the scale factor for the width and height to **150%**, and press *Enter* to apply the changes. Repeat the previous step for frame 10 of the **field** layer, but scale the **form field** symbol down to **0%**.

The symbol's animation is created by adding a motion tween. Adding the motion tween to **field** layer frames 10 and 20 will cause the **form field** symbol to grow from 0% of the original size to 150%, then to 100%. Figure 16.43 illustrates this portion of the animation.

Next, you'll add text to the movie clip to help the user understand the purpose of the corresponding text field. You'll set text to appear over the **form field** symbol as an example to the user. The text that appears below the **form field** symbol tells the user what should be typed in the text field.



**Fig. 16.42** | Adding the **field** symbol to the **nameWindow** movie clip.



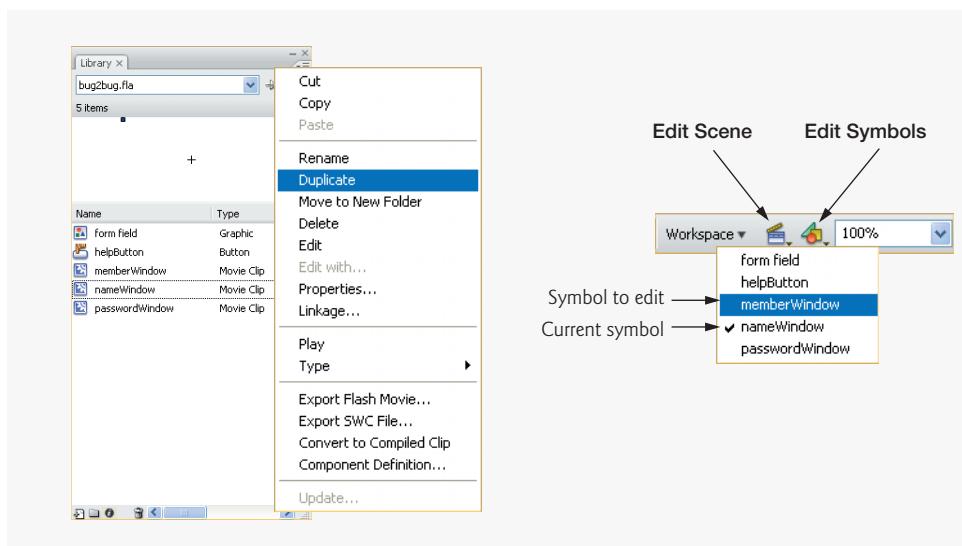
**Fig. 16.43** | Creating an animation with the **form field** symbol.

To add the descriptive text, first insert a keyframe in frame 25 of the **text** layer. Use the Text tool (white, Arial, 16 pt text and 3 pt line spacing) to type the help information for the **Name** field. Make sure the text type is **Static Text**. This text will appear in the help window. For instance, our example gives the following directions for the **Name** field: **Enter your name in this field. First name, Last name.** Align this text with the left side of the rectangle. Next, add a keyframe to frame 40 of this layer, causing the text to appear throughout the animation.

Next, duplicate this movie clip so that it may be customized and reused for the other two help button animations. Open the **Library** panel and right click the **nameWindow** movie clip. Select **Duplicate** from the resulting menu, and name the new clip **passwordWindow**. Repeat this step once more, and name the third clip **memberWindow** (Fig. 16.44).

You must customize the duplicated movie clips so their text reflects the corresponding form fields. To begin, open the **memberWindow** editing stage by pressing the **Edit Symbols** button, which is found in the upper-right corner of the editing environment, and selecting **memberWindow** from the list of available symbols (Fig. 16.44). Select frame 25 of the **text** layer and change the form field description with the Text tool so that the box contains the directions **Enter your member number here in the form: 556677**. Copy the text in frame 25 by selecting it with the Text tool and using the shortcut **<Ctrl>-C**. Click frame 40 of the **text** layer, which contains the old text. Highlight the old text with the Text tool, and use the shortcut **<Ctrl>-V** to paste the copied text into this frame. Repeat these steps for the **passwordWindow** movie clip using the directions **Enter your secret password in this field**. [Note: Changing a symbol's function or appearance in its editing stage updates the symbol in the scene.]

The following steps further customize the help boxes for each form field. Open the **nameWindow** symbol's editing stage by clicking the **Edit Symbols** button (Fig. 16.44) and selecting **nameWindow**. Add a new layer to this symbol called **typedText** above the **text** layer. This layer contains an animation that simulates the typing of text into the form field.



**Fig. 16.44** | Duplicating movie-clip symbols with the **Library** panel.

Insert a keyframe in frame 25. Select this frame and use the Text tool to create a text box on top of the **form** field symbol. Type the name **John Doe** in the text box, then change the text color to black.

The following frame-by-frame animation creates the appearance of the name being typed into the field. Add a keyframe to frame 40 to indicate the end of the animation. Then add new keyframes to frames 26–31. Each keyframe contains a new letter being typed in the sequence, so when the playhead advances, new letters appear. Select the **John Doe** text in frame 25 and delete everything except the first **J** with the Text tool. Next, select frame 26 and delete all of the characters except the **J** and the **o**. This step must be repeated for all subsequent keyframes up to frame 31, each keyframe containing one more letter than the last (Fig. 16.45). Frame 31 should show the entire name. When this process is complete, press *Enter* to preview the frame-by-frame typing animation.

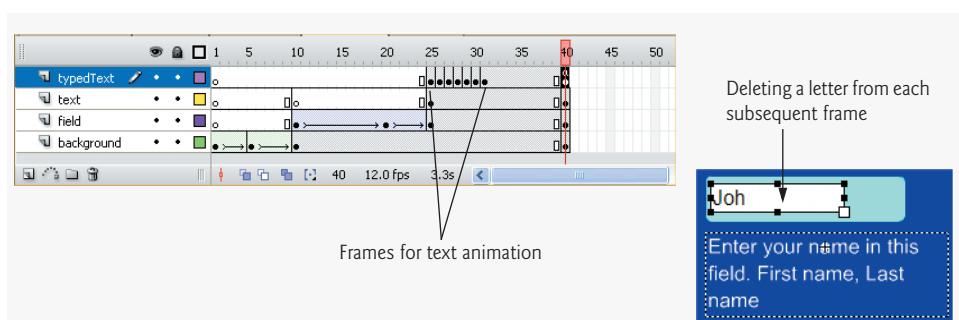
Create the same type of animation for both the **passwordWindow** and the **memberWindow** movie clips, using suitable words. For example, we use six asterisks for the **passwordWindow** movie clip and six numbers for the **memberWindow** movie clip. Add a stop action to frame 40 of all three movie clips so that the animations play only once.

The movie clips are now ready to be added to the scene. Click the **Edit Scene** button next to the **Edit Symbols** button, and select **Scene 1** to return to the scene. Before inserting the movie clips, add the following layers to the timeline: **nameMovie**, **memberMovie** and **passwordMovie**, one layer for each of the movie clips. Add a keyframe in frame 2 of the **nameMovie** layer. Also, add keyframes to frame 4 of the **form**, **text** and **button** layers, ensuring that the form and text appear throughout the movie.

Now you'll place the movie clips in the correct position in the scene. Recall that the ActionScript for each help button contains the script

```
function functionName(event : MouseEvent) : void
{
 gotoAndStop(frameLabel);
}
```

in which *functionName* and *frameLabel* depend on the button. This script causes the movie to skip to the specified frame and stop. Placing the movie clips in the correct frames causes the playhead to skip to the desired frame, play the animation and stop. This effect is created by selecting frame 2 of the **nameMovie** layer and dragging the **nameWindow** movie clip onto the stage. Align the movie clip with the button next to the **Name** field, placing it halfway between the button and the right edge of the stage.

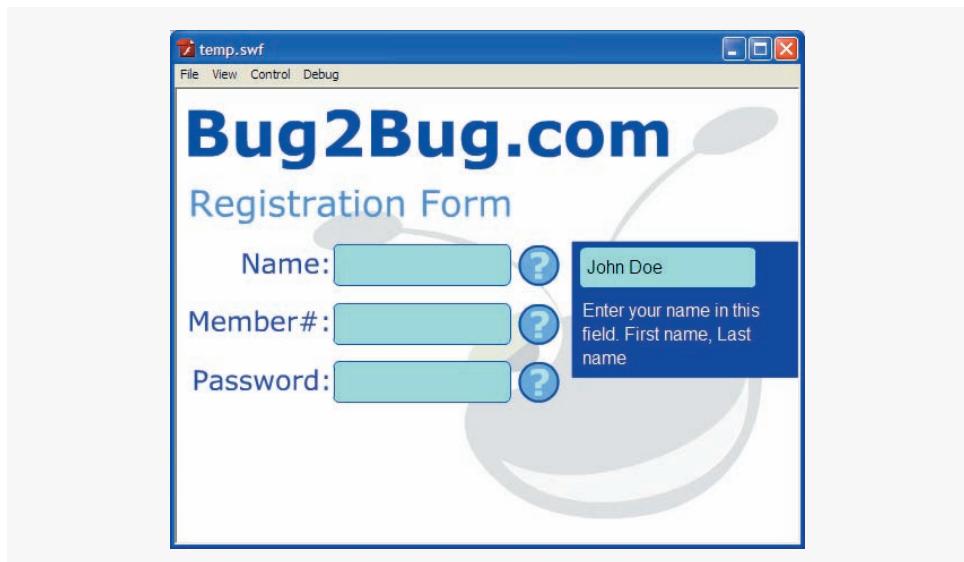


**Fig. 16.45** | Creating a frame-by-frame animation.

The preceding step is repeated twice for the other two movie clips so that they appear in the correct frames. Add a keyframe to frame 3 of the **memberMovie** layer and drag the **memberWindow** movie clip onto the stage. Position this clip in the same manner as the previous clip. Repeat this step for the **passwordWindow** movie clip, dragging it into frame 4 of the **passwordMovie** layer.

The movie is now complete. Press **<Ctrl>-Enter** to preview it with the Flash Player. If the triggered animations do not appear in the correct locations, return to the scene and adjust their position. The final movie is displayed in Fig. 16.46.

In our example, we have added a picture beneath the text layer. Movies can be enhanced in many ways, such as by changing colors and fonts or by adding pictures. Our movie (**bug2bug.fla**) can be found in the this chapter's examples directory. If you want to use our symbols to recreate the movie, select **Open External Library...** from the **Import** submenu of the **File** menu and open **bug2bug.fla**. The **Open External Library...** option allows you to reuse symbols from another movie.



**Fig. 16.46** | Bug2Bug.com help form.

## 16.6 Creating a Website Splash Screen

Flash is becoming an important tool for e-businesses. Many organizations use Flash to create website splash screens (i.e., introductions), product demos and web applications. Others use Flash to build games and interactive entertainment in an effort to attract new visitors. However, these types of applications can take a long time to load, causing visitors—especially those with slow connections—to leave the site. One way to alleviate this problem is to provide visitors with an animated Flash introduction that draws and keeps their attention. Flash animations are ideal for amusing visitors while conveying information as the rest of a page downloads “behind the scenes.”

A **preloader** or **splash screen** is a simple animation that plays while the rest of the web page is loading. Several techniques are used to create animation preloaders. The following

example creates an animation preloader that uses ActionScript to pause the movie at a particular frame until all the movie elements have loaded.

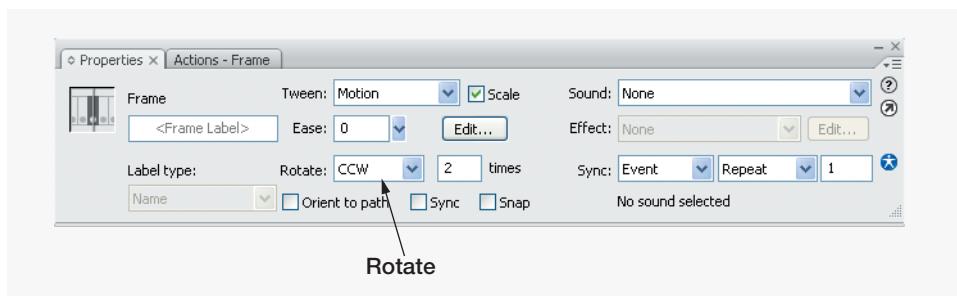
To start building the animation preloader, create a new Flash document. Use the default size, and set the background color to a light blue. First, you'll create the movie pieces that will be loaded later in the process. Create five new layers, and rename **Layer 2** to **C++**, **Layer 3** to **Java** and **Layer 4** to **IW3**. **Layer 5** will contain the movie's ActionScript, so rename it **actions**. Because **Layer 1** contains the introductory animation, rename this layer **animation**.

The preloaded objects we use in this example are animated movie clip symbols. Create the first symbol by clicking frame 2 of the **C++** layer, inserting a keyframe, and creating a new movie-clip symbol named **cppbook**. When the symbol's editing stage opens, import the image **cpphttp.gif** (found in the **images** folder with this chapter's examples). Place a keyframe in frame 20 of **Layer 1** and add a **stop** action to this frame. The animation in this example is produced with the motion tween **Rotate** option, which causes an object to spin on its axis. Create a motion tween in frame 1 with the **Properties** window, setting the **Rotate** option to **CCW** (counterclockwise) and the **times** field to **2** (Fig. 16.47). This causes the image **cpphttp.gif** to spin two times counterclockwise over a period of 20 frames.

After returning to the scene, drag and drop a copy of the **cppbook** symbol onto the stage in frame 2 of the **C++** layer. Move this symbol to the left side of the stage. Insert a frame in frame 25 of the **C++** layer.

Build a similar movie clip for the **Java** and **IW3** layers, using the files **java.gif** and **iw3.gif** to create the symbols. Name the symbol for the **Java** layer **jbook** and the **IW3** symbol **ibook** to identify the symbols with their contents. In the main scene, create a keyframe in frame 8 of the **Java** layer, and place the **jbook** symbol in the center of the stage. Insert a frame in frame 25 of the **Java** layer. Insert the **ibook** symbol in a keyframe in frame 14 of the **IW3** layer, and position it to the right of the **jbook** symbol. Insert a frame in frame 25 of the **IW3** layer. Make sure to leave some space between these symbols so that they will not overlap when they spin (Fig. 16.48). Add a keyframe to the 25th frame of the actions layer, then add a **stop** to the **Actions** panel of that frame.

Now that the loading objects have been placed, it is time to create the preloading animation. By placing the preloading animation in the frame preceding the frame that contains the objects, we can use ActionScript to pause the movie until the objects have loaded. Begin by adding a **stop** action to frame 1 of the **actions** layer. Select frame 1 of the **animation** layer and create another new movie-clip symbol named **loader**. Use the text tool with a medium-sized sans-serif font, and place the word **Loading** in the center of the symbol's



**Fig. 16.47** | Creating a rotating object with the motion tween **Rotate** option.



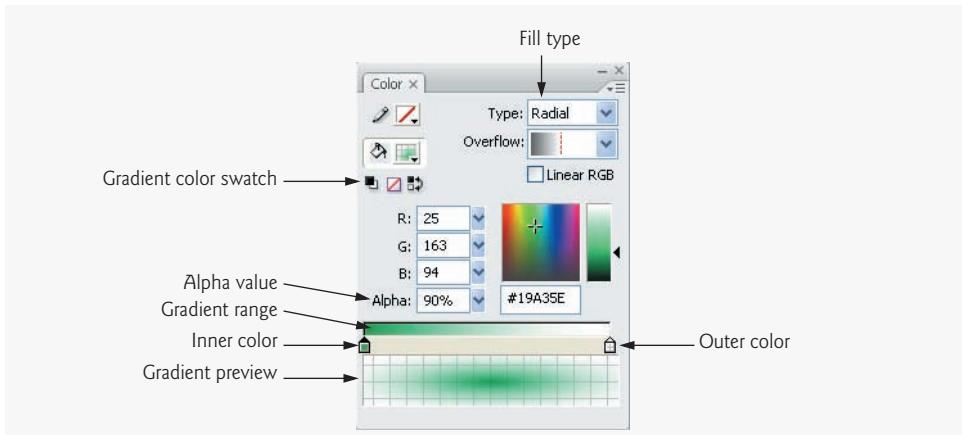
**Fig. 16.48** | Inserted movie clips.

editing stage. This title indicates to the user that objects are loading. Insert a keyframe into frame 14 and rename this layer **load**.

Create a new layer called **orb** to contain the animation. Draw a circle with no stroke about the size of a quarter above the word **Loading**. Give the circle a green-to-white radial gradient fill color. The colors of this gradient can be edited in the **Color** panel (Fig. 16.49).

The block farthest to the left on the **gradient range** indicates the innermost color of the radial gradient, whereas the block farthest to the right indicates the outermost color of the radial gradient. Click the left block to reveal the **gradient color swatch**. Click the swatch and select a medium green as the inner color of the gradient. Select the right, outer color box and change its color to white. Deselect the circle by clicking on a blank portion of the stage. Note that a white ring appears around the circle due to the colored background. To make the circle fade into the background, we adjust its **alpha** value. Alpha is a value between 0 and 100% that corresponds to a color's transparency or opacity. An alpha value of 0% appears transparent, whereas a value of 100% appears completely opaque. Select the circle again and click the right gradient box (white). Adjust the value of the **Alpha** field in the **Color Mixer** panel to 0%. Deselect the circle. It should now appear to fade into the background.

The rate of progression in a gradient can also be changed by sliding the color boxes. Select the circle again. Slide the left color box to the middle so that the gradient contains more green than transparent white, then return the slider to the far left. Intermediate colors may be added to the gradient range by clicking beneath the bar, next to one of the existing



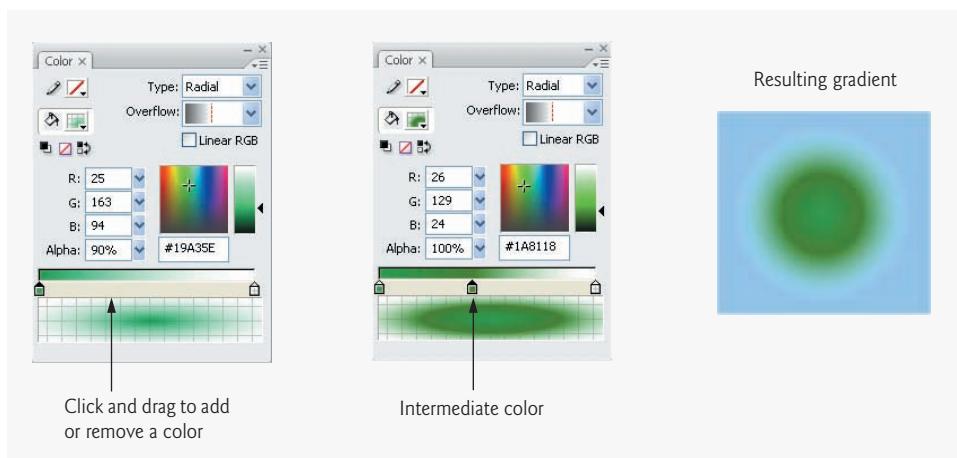
**Fig. 16.49** | Changing gradient colors with the **Color** panel.

color boxes. Click to the right of the inner color box to add a new color box (Fig. 16.50). Slide the new color box to the right and change its color to a darker green. Any color box may be removed from a gradient by dragging it downward off the gradient range.

Insert keyframes into frame 7 and 14 of the **orb** layer. Select the circle in frame 7 with the selection tool. In the **Color** panel change the alpha of every color box to 0%. Select frame 1 in the **Timeline** and add shape tween. Change the value of the **Ease** field in the **Properties** window to **-100**. **Ease** controls the rate of change during tween animation. Negative values cause the animated change to be gradual at the beginning and become increasingly drastic. Positive values cause the animation to change quickly in the first frames, becoming less drastic as the animation progresses. Add shape tween to frame 7 and set the **Ease** value to 100. In frame 14, add the action `gotoAndPlay(1);` to repeat the animation. You can preview the animation by pressing *Enter*. The circle should now appear to pulse.

Before inserting the movie clip into the scene, we are going to create a **hypertext linked button** that will enable the user to skip over the animations to the final destination. Add a new layer called **link** to the **loader** symbol with keyframes in frames 1 and 14. Using the text tool, place the words **skip directly to Deitel website** below **Loading** in a smaller font size. Select the words with the selection tool and convert them into a button symbol named **skip**. Converting the text into a button simulates a text hyperlink created with XHTML. Double click the words to open the **skip** button's editing stage. For this example, we are going to edit only the hit state. When a button is created from a shape, the button's hit area is, by default, the area of the shape. It is important to change the hit state of a button created from text so that it includes the spaces between the letters; otherwise, the link will work only when the user hovers over a letter's area. Insert a keyframe in the hit state. Use the rectangle tool to draw the hit area of the button, covering the entire length and height of the text. This rectangle is not visible in the final movie, because it defines only the hit area (Fig. 16.51).

The button is activated by giving it an action that links it to another web page. After returning to the **loader** movie-clip editing stage, give the **skip** button the instance name **skipButton** and open the **Actions** panel for the first frame of the **link** layer. Invoke the **addEventListerner** function using the **skipButton** instance to call function **onClick** whenever



**Fig. 16.50** | Adding an intermediate color to a gradient.



**Fig. 16.51** | Defining the hit area of a button.

the button is clicked. Then, create an object of type `URLRequest` and give the constructor a parameter value of "`http://www.deitel.com`". The function `onClick` employs Flash's `navigateToURL` function to access the website given to it. Thus, the code now reads

```
skipButton.addEventListener(MouseEvent.CLICK, onClick);
var url : URLRequest = new URLRequest("http://www.deitel.com");

function onClick(e : MouseEvent) : void
{
 navigateToURL(url, "_blank");
} // end function onClick
```

The "`_blank`" parameter signifies that a new browser window displaying the Deitel website should open when the user presses the button.

Return to the scene by clicking **Scene 1** directly below the timeline, next to the name of the current symbol. Drag and drop a copy of the **loader** movie clip from the **Library** panel into frame 1 of the **animation** layer, center it on the stage, and set its **Instance name** to **loadingClip**.

The process is nearly complete. Open the **Actions** panel for the **actions** layer. The following actions direct the movie clip to play until all the scene's objects are loaded. First, add a `stop` to the frame so that it doesn't go to the second frame until we tell it to. Using the **loadingClip** movie instance, use the `addEventListener` function to invoke the function `onBegin` whenever the event `Event.ENTER_FRAME` is triggered. The `ENTER_FRAME` event occurs every time the playhead enters a new frame. Since this movie's frame rate is 12 fps (frames per second), the `ENTER_FRAME` event will occur 12 times each second.

```
loadingClip.addEventListener(Event.ENTER_FRAME, onBegin);
```

The next action added to this sequence is the function `onBegin`. The condition of the `if` statement will be used to determine how many frames of the movie are loaded. Flash movies load frame by frame. Frames that contain complex images take longer to load. Flash will continue playing the current frame until the next frame has loaded. For our movie, if the number of frames loaded (`frameLoaded`) is equal to the total number of frames (`totalFrames`), then the movie is finished loading, so it will play frame 2. It also invokes the `removeEventListener` function to ensure that `onBegin` is not called for the remainder of the movie. If the number of frames loaded is less than the total number of frames, then the current movie clip continues to play. The code now reads:

```
stop();
loadingClip.addEventListener(Event.ENTER_FRAME, onBegin);

// check if all frames have been loaded
function onBegin(event : Event) : void
{
 if (framesLoaded == totalFrames)
 {
```

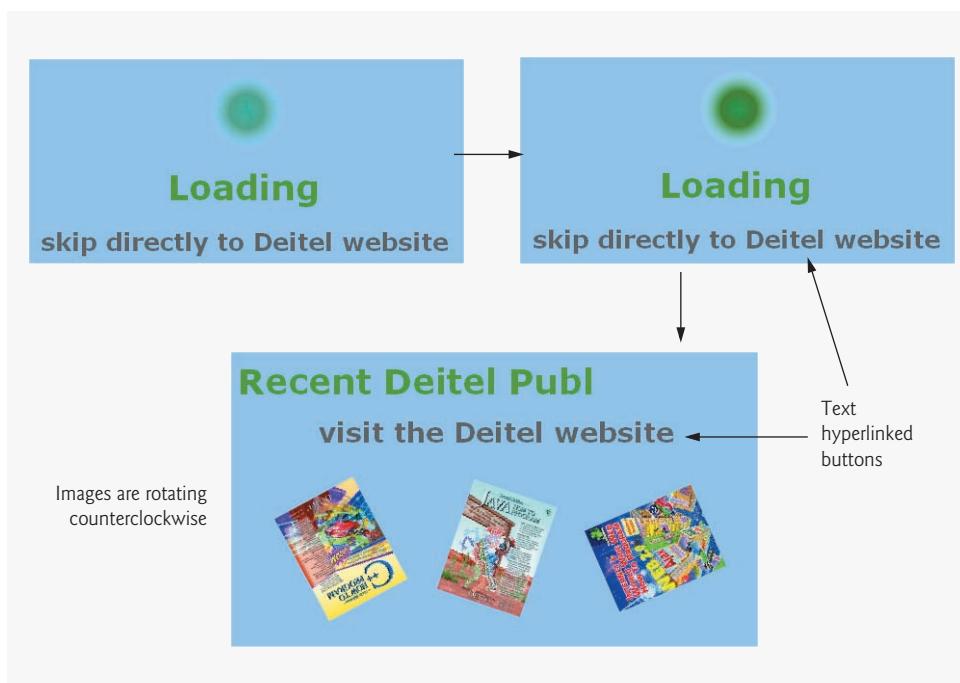
```

loadingClip.removeEventListener(Event.ENTER_FRAME, onBegin);
gotoAndPlay(2);
} // end if
} // end function onBegin

```

Create one more layer in the scene, and name the layer **title**. Add a keyframe to frame 2 of this layer, and use the Text tool to create a title that reads **Recent Deitel Publications**. Below the title, create another text hyperlink button to the Deitel website. The simplest way to do this is to duplicate the existing **skip** button and modify the text. Right click the **skip** symbol in the **Library** panel, and select **Duplicate**. Name the new button **visit**, and place it in frame 2 of the **title** layer. Label the instance **visitButton**, then create a keyframe in the second frame of the **actions** layer. Duplicate the code from the **Actions** panel of the first frame of the **link** layer in the **loader** symbol, and replace **skipButton** with **visitButton**. Double click the **visit** button and edit the text to say **visit the Deitel website**. Add keyframes to each frame of the **title** layer and manipulate the text to create a typing effect similar to the one we created in the **bug2bug** example.

The movie is now complete. Test the movie with the Flash Player (Fig. 16.52). When viewed in the testing window, the loading sequence will play for only one frame because your processor loads all the frames almost instantly. Flash can simulate how a movie would appear to an online user, though. While still in the testing window, select **56K** from the **Download Settings** submenu of the **View** menu. Also, select **Bandwidth Profiler** from the **View** menu. Then select **Simulate Download** from the **View** menu or press **<Ctrl>-Enter**. The graph at the top of the window displays the amount of bandwidth required to load each frame.



**Fig. 16.52** | Creating an animation to preload images.

## 16.7 ActionScript

Figure 16.53 lists common Flash ActionScript 3.0 functions. By attaching these functions to frames and symbols, you can build some fairly complex Flash movies.

| Function           | Description                                                                                                                                                                                                                                                                                  |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| gotoAndPlay        | Jump to a frame or scene in another part of the movie and start playing the movie.                                                                                                                                                                                                           |
| gotoAndStop        | Jump to a frame or scene in another part of the movie and stop the movie.                                                                                                                                                                                                                    |
| play               | Start playing a movie from the beginning, or from wherever it has been stopped.                                                                                                                                                                                                              |
| stop               | Stop a movie.                                                                                                                                                                                                                                                                                |
| SoundMixer.stopAll | Stop the sound track without affecting the movie.                                                                                                                                                                                                                                            |
| navigateToUrl      | Load a URL into a new or existing browser window.                                                                                                                                                                                                                                            |
| fscommand          | Insert JavaScript or other scripting languages into a Flash movie.                                                                                                                                                                                                                           |
| Loader class       | Load a SWF or JPEG file into the Flash Player from the current movie. Can also load another SWF into a particular movie.                                                                                                                                                                     |
| framesLoaded       | Check whether certain frames have been loaded.                                                                                                                                                                                                                                               |
| addEventListener   | Assign functions to a movie clip based on specific events. The events include <code>load</code> , <code>unload</code> , <code>enterFrame</code> , <code>mouseUp</code> , <code>mouseDown</code> , <code>mouseMove</code> , <code>keyUp</code> , <code>keyDown</code> and <code>data</code> . |
| if                 | Set up condition statements that run only when the condition is true.                                                                                                                                                                                                                        |
| while/do while     | Run a collection of statements while a condition statement is true.                                                                                                                                                                                                                          |
| trace              | Display programming notes or variable values while testing a movie.                                                                                                                                                                                                                          |
| Math.random        | Returns a random number less than or equal to 0 and less than 1.                                                                                                                                                                                                                             |

**Fig. 16.53** | Common ActionScript functions.

## 16.8 Wrap-Up

In this chapter, we introduced Adobe Flash CS3 (Creative Suite 3) and demonstrated how to produce interactive, animated movies. You learned that Flash CS3 can be used to create web-based banner advertisements, interactive websites, games and web-based applications, and that it provides tools for drawing graphics, generating animations, and adding sound and video. We discussed how to embed Flash movies in web pages and how to execute Flash movies as stand-alone programs. You also learned some ActionScript 3.0 programming and created interactive buttons, added sound to movies, created special graphic effects. In the next chapter, you'll build an interactive game using Flash.

## 16.9 Web Resources

[www.deitel.com/flash9/](http://www.deitel.com/flash9/)

The Deitel Flash 9 Resource Center contains links to some of the best Flash 9 and Flash CS3 resources on the web. There you'll find categorized links to forums, conferences, blogs, books, open source projects, videos, podcasts, webcasts and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on Microsoft Silverlight ([www.deitel.com/silverlight/](http://www.deitel.com/silverlight/)) and Adobe Flex ([www.deitel.com/flex/](http://www.deitel.com/flex/)).

---

## Summary

### Section 16.1 Introduction

- Adobe Flash CS3 (Creative Suite 3) is a commercial application that you can use to produce interactive, animated movies.
- Flash can be used to create web-based banner advertisements, interactive websites, games and web-based applications with stunning graphics and multimedia effects.
- Flash movies can be embedded in web pages, placed on CDs or DVDs as independent applications or converted into stand alone, executable programs.
- Flash includes tools for coding in its scripting language, ActionScript 3.0. ActionScript, which is similar to JavaScript, enables interactive applications.
- To play Flash movies, the Flash Player plug-in must be installed in your web browser. This plug-in has several versions, the most recent of which is version 9.

### Section 16.2 Flash Movie Development

- The stage is the white area in which you place graphic elements during movie development. Only objects in this area will appear in the final movie.
- The timeline represents the time period over which a movie runs.
- Each frame depicts a moment in the movie's timeline, into which you can insert movie elements.
- The playhead indicates the current frame.
- The **Tools** bar is divided into multiple sections, each containing tools and functions that help you create Flash movies.
- Windows called panels organize frequently used movie options. Panel options modify the size, shape, color, alignment and effects associated with a movie's graphic elements.
- The context-sensitive **Properties** panel displays information about the currently selected object. It is a useful tool for viewing and altering an object's properties.
- You can access different panels by selecting them from the **Window** menu.

### Section 16.3 Learning Flash with Hands-On Examples

- The **.fla** file extension is a Flash-specific extension for editable movies.
- **Frame rate** sets the speed at which movie frames display.
- The background color determines the color of the stage.
- **Dimensions** define the size of a movie as it displays on the screen.

### Section 16.3.1 Creating a Shape with the Oval Tool

- Flash creates shapes using vectors—mathematical equations that define the shape's size, shape and color. When vector graphics are saved, they are stored using equations.

- Vector graphics can be resized without losing clarity.
- You can create shapes by dragging with the shape tools.
- Every shape has a stroke color and a fill color. The stroke color is the color of a shape's outline, and the fill color is the color that fills the shape.
- Clicking the Black and white button resets the stroke color to black and the fill color to white.
- Selecting the Swap colors option switches the stroke and fill colors.
- The *Shift* key constrains a shape's proportions to have equal width and height.
- A dot in a frame signifies a keyframe, which indicates a point of change in a timeline.
- A shape's size can be modified with the **Properties** panel when the shape is selected.
- Gradient fills are gradual progressions of color.
- The **Swatches** panel provides four radial gradients and three linear gradients.

#### **Section 16.3.2 Adding Text to a Button**

- Button titles communicate a button's function to the user. You can create a title with the Text tool.
- With selected text, you can change the font, text size and font color with the **Properties** window.
- To change the font color, click the text color swatch and select a color from the palette.

#### **Section 16.3.3 Converting a Shape into a Symbol**

- The scene contains graphics and symbols. The parent movie may contain several symbols that are reusable movie elements, such as graphics, buttons and movie clips.
- A scene timeline can contain numerous symbols with their own timelines and properties.
- A scene may have several instances of any given symbol.
- Symbols can be edited independently of the scene by using the symbol's editing stage. The editing stage is separate from the scene stage and contains only one symbol.
- Selecting **Convert to Symbol...** from the **Modify** menu or using the shortcut *F8* on the keyboard opens the **Convert to Symbol** dialog, in which you can set the properties of a new symbol.
- Every symbol in a Flash movie must have a unique name.
- You can create three different types of symbols—movie clips, buttons and graphics.
- A movie-clip symbol is ideal for recurring animations.
- Graphic symbols are ideal for static images and basic animations.
- Button symbols are objects that perform button actions, such as rollovers and hyperlinking. A rollover is an action that changes the appearance of a button when the mouse passes over it.
- The **Library** panel stores every symbol in a movie and is accessed through the **Window** menu or by the shortcuts *<Ctrl>-L* or *F11*. Multiple instances of a symbol can be placed in a movie by dragging and dropping the symbol from the **Library** panel onto the stage.

#### **Section 16.3.4 Editing Button Symbols**

- The different components of a button symbol, such as its fill and type, may be edited in the symbol's editing stage. You may access a symbol's editing stage by double clicking the symbol in the **Library** or by pressing the **Edit Symbols** button and selecting the symbol name.
- The pieces that make up a button can all be changed in the editing stage.
- A button symbol's timeline contains four frames, one for each of the button states (up, over and down) and one for the hit area.

- The up state (indicated by the **Up** frame on screen) is the default state before the user presses the button or rolls over it with the mouse.
- Control shifts to the over state (i.e., the **Over** frame) when the mouse moves over the button.
- The button's down state (i.e., the **Down** frame) plays when a user presses a button. You can create interactive, user-responsive buttons by customizing the appearance of a button in each state.
- Graphic elements in the hit state (i.e., the **Hit** frame) are not visible when viewing the movie; they exist simply to define the active area of the button (i.e., the area that can be clicked).
- By default, buttons only have the up state activated when they are created. You may activate other states by adding keyframes to the other three frames.

#### *Section 16.3.5 Adding Keyframes*

- Keyframes are points of change in a Flash movie and appear in the timeline as gray with a black dot. By adding keyframes to a button symbol's timeline, you can control how the button reacts to user input.
- A rollover is added by inserting a keyframe in the button's **Over** frame, then changing the button's appearance in that frame.
- Changing the button color in the over state does not affect the button color in the up state.

#### *Section 16.3.6 Adding Sound to a Button*

- Flash imports sounds in the **WAV** (Windows), **AIFF** (Macintosh) or **MP3** formats.
- Sounds can be imported into the **Library** by choosing **Import to Library** from the **Import** submenu of the **File** menu.
- You can add sound to a movie by placing the sound clip in a keyframe or over a series of frames.
- If a frame has a blue wave or line through it, a sound effect has been added to it.

#### *Section 16.3.7 Verifying Changes with Test Movie*

- Movies can be viewed in their published state with the Flash Player. The published state of a movie is how it would appear if viewed over the web or with the Flash Player.
- Published Flash movies have the Shockwave Flash extension (**.swf**). SWF files can be viewed but not edited.

#### *Section 16.3.8 Adding Layers to a Movie*

- A movie can be composed of many layers, each having its own attributes and effects.
- Layers organize different movie elements so that they can be animated and edited separately, making the composition of complex movies easier. Graphics in higher layers appear over the graphics in lower layers.
- Text can be broken apart or regrouped for color editing, shape modification or animation. However, once text has been broken apart, it may not be edited with the Text tool.

#### *Section 16.3.9 Animating Text with Tweening*

- Animations in Flash are created by inserting keyframes into the timeline.
- Tweening, also known as morphing, is an automated process in which Flash creates the intermediate steps of the animation between two keyframes.
- Shape tweening morphs an ungrouped object from one shape to another.
- Motion tweening moves symbols or grouped objects around the stage.
- Keyframes must be designated in the timeline before adding the motion tween.

- Adding the `stop` function to the last frame in a movie stops the movie from looping.
- The small letter `a` in a frame indicates that it contains an action.

#### ***Section 16.3.10 Adding a Text Field***

- **Static Text** creates text that does not change.
- **Dynamic Text** creates can be changed or determined by outside variables through ActionScript.
- **Input Text** creates a text field into which the viewers of the movie can input their own text.

#### ***Section 16.3.11 Adding ActionScript***

- The `addEventListener` function helps make an object respond to an event by calling a function when the event takes place.
- `MouseEvent.MOUSE_DOWN` specifies that an action is performed when the user clicks the button.
- `Math.random` returns a random floating-point number from 0.0 up to, but not including, 1.0.

#### ***Section 16.4 Publishing Your Flash Movie***

- Flash movies must be published for users to view them outside Flash CS3 and the Flash Player.
- Flash movies may be published in a different Flash version to support older Flash Players.
- Flash can automatically generate an XHMTL document that embeds your Flash movie.

#### ***Section 16.5.1 Importing and Manipulating Bitmaps***

- Once an imported image is broken apart, it may be shape tweened or edited with editing tools such as the Lasso, Paint bucket, Eraser and Paintbrush. The editing tools are found in the toolbox and apply changes to a shape.
- Dragging with the Lasso tool selects areas of shapes. The color of a selected area may be changed, or the selected area may be moved.
- Once an area is selected, its color may be changed by selecting a new fill color with the fill swatch or by clicking the selection with the Paint bucket tool.
- The Eraser tool removes shape areas when you click and drag the tool across an area. You can change the eraser size using the tool options.

#### ***Section 16.5.2 Creating an Advertisement Banner with Masking***

- Masking hides portions of layers. A masking layer hides objects in the layers beneath it, revealing only the areas that can be seen through the shape of the mask.
- Items drawn on a masking layer define the mask's shape and cannot be seen in the final movie.
- The Free transform tool allows us to resize an image. When an object is selected with this tool, anchors appear around its corners and sides.
- Breaking text apart once converts each letter into its own text field. Breaking it apart again converts the letters into shapes that cannot be edited with the Text tool, but can be manipulated as regular graphics.
- Adding a mask to a layer masks only the items in the layer directly below it.

#### ***Section 16.5.3 Adding Online Help to Forms***

- Use the Selection tool to align objects with their corresponding captions. For more precise alignment, select the desired object with the Selection tool and press the arrow key on the keyboard in the direction you want to move the object.
- An input text field is a text field into which the user can type text.

- Each movie clip should be created as a new symbol so that it can be edited without affecting the scene.
- Symbols may be embedded in one another; however, they cannot be placed within themselves.
- The **Transform** panel can be used to change an object's size.
- The **Constrain** checkbox causes the scale factor to be equal in the height and width fields. The scale factor measures the change in proportion.
- Changing a symbol's function or appearance in its editing stage updates the symbol in the scene.

### **Section 16.6 Creating a Website Splash Screen**

- Many organizations use Flash to create website splash screens (i.e., introductions), product demos and web applications.
- Flash animations are ideal for amusing visitors while conveying information as the rest of a page downloads “behind the scenes.”
- A preloader is a simple animation that plays while the rest of the web page is loading.
- Alpha is a value between 0 and 100% that corresponds to a color’s transparency or opacity. An alpha value of 0% appears transparent, whereas a value of 100% appears completely opaque.
- The rate of progression in a gradient can also be changed by sliding the color boxes.
- Any color box may be removed from a gradient by dragging it downward off the gradient range.
- **Ease** controls the rate of change during tween animation. Negative values cause the animated change to be gradual at the beginning and become increasingly drastic. Positive values cause the animation to change quickly in the first frames and less drastically as the animation progresses.
- When a button is created from a shape, the button’s hit area is, by default, the area of the shape.
- It is important to change the hit state of a button created from text so that it includes the spaces between the letters; otherwise, the link will work only when the user hovers over a letter’s area.
- The “`_blank`” signifies that a new browser window should open when the user presses the button.
- Flash movies load frame by frame, and frames containing complex images take longer to load. Flash will continue playing the current frame until the next frame has loaded.

## **Terminology**

|                                         |                                    |
|-----------------------------------------|------------------------------------|
| ActionScript 3.0                        | frame                              |
| active tool                             | frame label                        |
| <code>addEventListener</code> function  | <b>Frame Rate</b>                  |
| Adobe Flash CS3                         | frames per second                  |
| alpha value                             | <code>framesLoaded</code> property |
| anchor                                  | free transform tool                |
| <b>Bandwidth Profiler</b>               | <code>fscommand</code> function    |
| bitmapped graphics                      | <code>gotoAndPlay</code> function  |
| break apart                             | <code>gotoAndStop</code> function  |
| <b>Brush Mode</b>                       | gradients                          |
| <b>Brush Tool</b>                       | <b>Hand</b> tool                   |
| constrained aspect ratio                | hexadecimal notation               |
| <code>do while</code> control structure | hit state                          |
| down state                              | hypertext link                     |
| duplicate symbol                        | <code>if</code> control structure  |
| <b>Eraser tool</b>                      | input text field                   |
| .fla file format                        | instance                           |

|                              |                             |
|------------------------------|-----------------------------|
| instance name                | raster graphic              |
| interactive animated movies  | raw compression             |
| JavaScript                   | Rectangle tool              |
| keyframe                     | Sample Rate                 |
| <b>Lasso tool</b>            | scenes                      |
| layer                        | Selection tool              |
| Library panel                | shape tween                 |
| Loader class                 | SoundMixer.stopAll function |
| <b>Magic wand</b>            | splash screen               |
| masking layer                | stage                       |
| math.random function         | stop function               |
| motion tween                 | .swf file format            |
| movie clip                   | symbol                      |
| movie clip symbol            | Text tool                   |
| MP3 audio compression format | timeline                    |
| navigateToUrl function       | trace function              |
| Oval tool                    | tween                       |
| over state                   | up state                    |
| play function                | vector graphic              |
| playhead                     | web-safe palette            |
| preload                      | while control structure     |
| radial gradient              | Zoom tool                   |

## Self-Review Exercises

**16.1** Fill in the blanks in each of the following statements:

- a) Adobe Flash's \_\_\_\_\_ feature draws the in-between frames of an animation.
- b) Graphics, buttons and movie clips are all types of \_\_\_\_\_.
- c) The two types of tweening in Adobe Flash are \_\_\_\_\_ tweening and \_\_\_\_\_ tweening.
- d) Morphing one shape into another over a period of time can be accomplished with \_\_\_\_\_ tweening.
- e) Adobe Flash's scripting language is called \_\_\_\_\_.
- f) The area in which the movie is created is called the \_\_\_\_\_.
- g) Holding down the *Shift* key while drawing with the Oval tool draws a perfect \_\_\_\_\_.
- h) By default, shapes in Flash are created with a fill and a(n) \_\_\_\_\_.
- i) \_\_\_\_\_ tell Flash how a shape or symbol should look at the beginning and end of an animation.
- j) A graphic's transparency can be altered by adjusting its \_\_\_\_\_.

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

- a) A button's hit state is entered when the button is clicked.
- b) To draw a straight line in Flash, hold down the *Shift* key while drawing with the Pencil tool.
- c) Motion tweening moves objects within the stage.
- d) The more frames you give to an animation, the slower it is.
- e) Flash's `math.random` function returns a number between 1 and 100.
- f) The maximum number of layers allowed in a movie is ten.
- g) Flash can shape tween only one shape per layer.
- h) When a new layer is created, it is placed above the selected layer.

- i) The **Lasso Tool** selects objects by drawing freehand or straight-edge selection areas.
- ii) The **Ease** value controls an object's transparency during motion tween.

## Answers to Self-Review Exercises

**16.1** a) tweening. b) symbols. c) shape, motion. d) shape. e) ActionScript. f) stage. g) circle. h) stroke. i) keyframes. j) alpha value.

**16.2** a) False. The down state is entered when the button is clicked. b) True. c) True. d) True. e) False. Flash's `math.random` function returns a number greater than or equal to 0 and less than 1. f) False. Flash allows an unlimited number of layers for each movie. g) False. Flash can tween as many shapes as there are on a layer. The effect is usually better when the shapes are placed on their own layers. h) True. i) True. j) False. The **Ease** value controls the acceleration of a tween animation.

## Exercises

**16.3** Using the combination of one movie-clip symbol and one button symbol to create a navigation bar that contains four buttons, make the buttons trigger an animation (contained in the movie clip) when the user rolls over the buttons with the mouse. Link the four buttons to [www.nasa.gov](http://www.nasa.gov), [www.w3c.org](http://www.w3c.org), [www.flashkit.com](http://www.flashkit.com) and [www.cnn.com](http://www.cnn.com).

**16.4** Download and import five `WAV` files from [www.coolarchive.com](http://www.coolarchive.com). Create five buttons, each activating a different sound when it is pressed.

**16.5** Create an animated mask that acts as a spotlight on an image. First, import the file `arches.jpg` from the `images` folder in the Chapter 16 examples directory. Then, change the background color of the movie to black. Animate the mask in the layer above to create a spotlight effect.

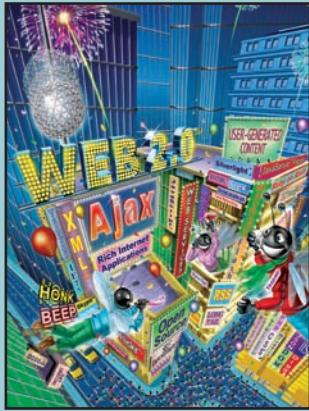
**16.6** Create a text “morph” animation using a shape tween. Make the text that appears in the first frame of the animation change into a shape in the last frame. Make the text and the shape different colors.

**16.7** Give a brief description of the following terms:

- a) symbol
- b) tweening
- c) ActionScript
- d) **Frame rate**
- e) **Library panel**
- f) masking
- g) context-sensitive **Properties** window
- h) **Bandwidth Profiler**
- i) **Frame Label**

**16.8** Describe what the following file extensions are used for in Flash movie development.

- a) `.fla`
- b) `.swf`
- c) `.exe`
- d) `.html`



*Knowledge must come through action.*

—Sophocles

*It is circumstance and proper timing that give an action its character and make it either good or bad.*

—Agesilaus

*Life's but a walking shadow, a poor player that struts and frets his hour upon the stage and then is heard no more: it is a tale told by an idiot, full of sound and fury, signifying nothing.*

—William Shakespeare

*Come, Watson, come! The game is afoot.*

—Sir Arthur Conan Doyle

*Cannon to right of them,  
Cannon to left of them,  
Cannon in front of them  
Volley'd and thunder'd.*

—Alfred, Lord Tennyson

# Adobe® Flash® CS3: Building an Interactive Game

## OBJECTIVES

In this chapter you'll learn:

- Advanced ActionScript 3 in Flash CS3.
- How to build on Flash CS3 skills learned in Chapter 16.
- The basics of object-oriented programming in Flash CS3.
- How to create a functional, interactive Flash game.
- How to make objects move in Flash.
- How to embed sound and text objects into a Flash movie.
- How to detect collisions between objects in Flash.

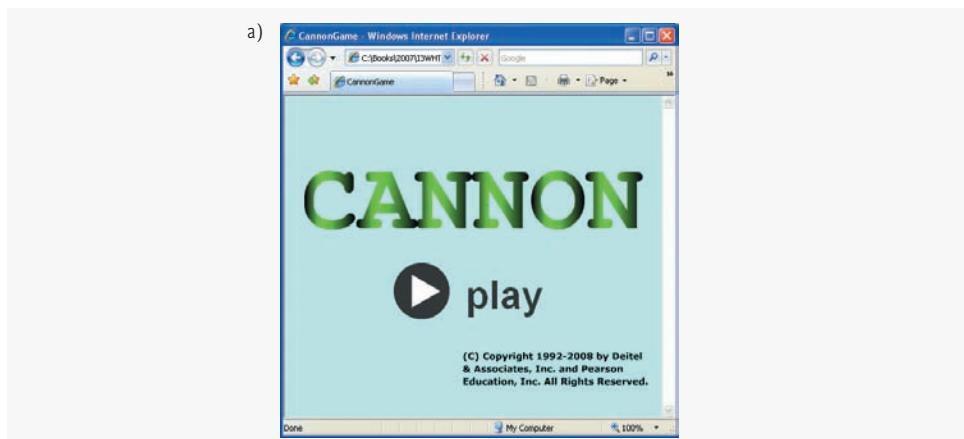
**Outline**

- 17.1 Introduction
- 17.2 Object-Oriented Programming
- 17.3 Objects in Flash
- 17.4 Cannon Game: Preliminary Instructions and Notes
- 17.5 Adding a Start Button
- 17.6 Creating Moving Objects
- 17.7 Adding the Rotating Cannon
- 17.8 Adding the Cannonball
- 17.9 Adding Sound and Text Objects to the Movie
- 17.10 Adding the Time Counter
- 17.11 Detecting a Miss
- 17.12 Adding Collision Detection
- 17.13 Finishing the Game
- 17.14 ActionScript 3.0 Elements Introduced in This Chapter

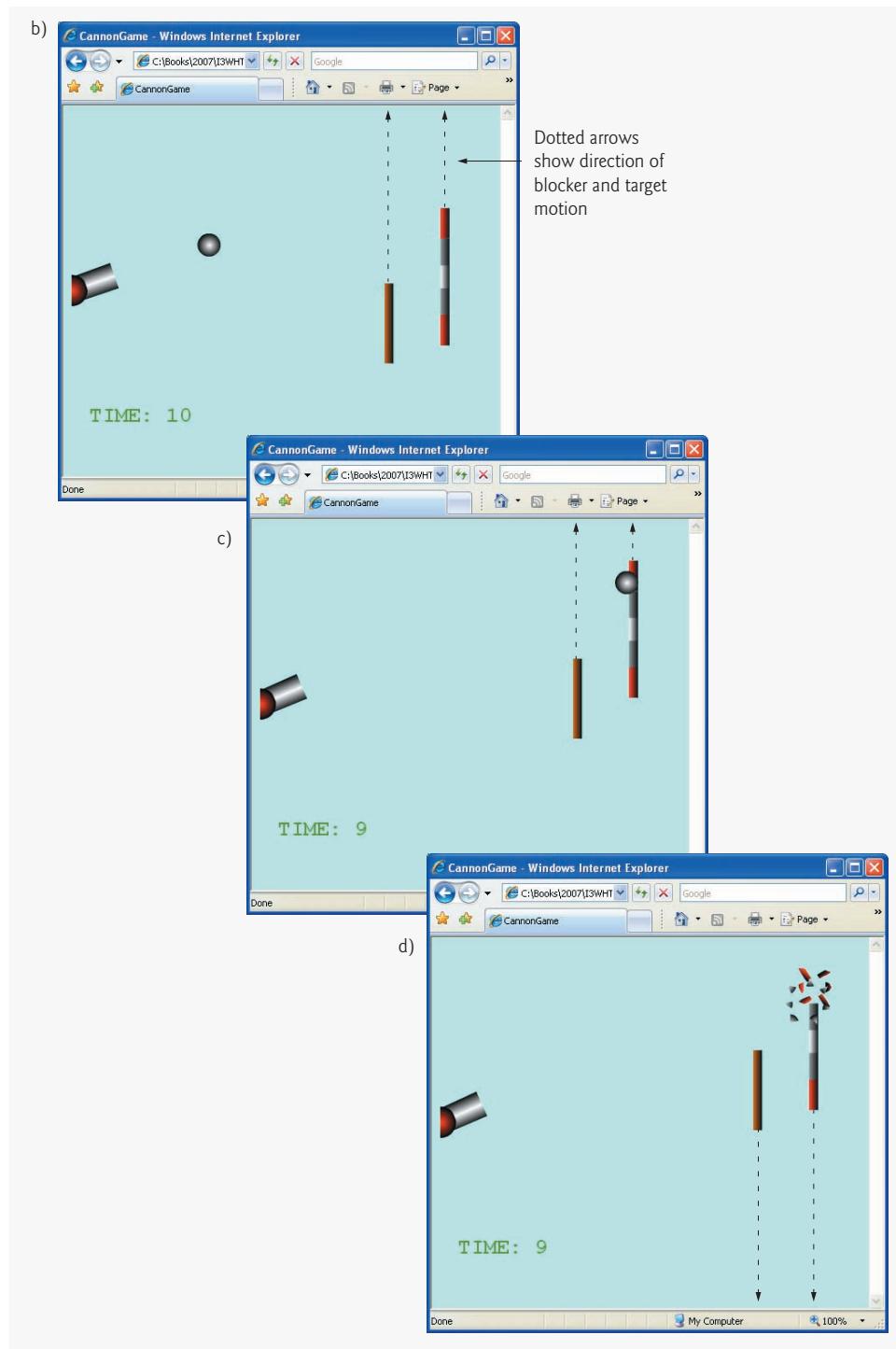
[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 17.1 Introduction

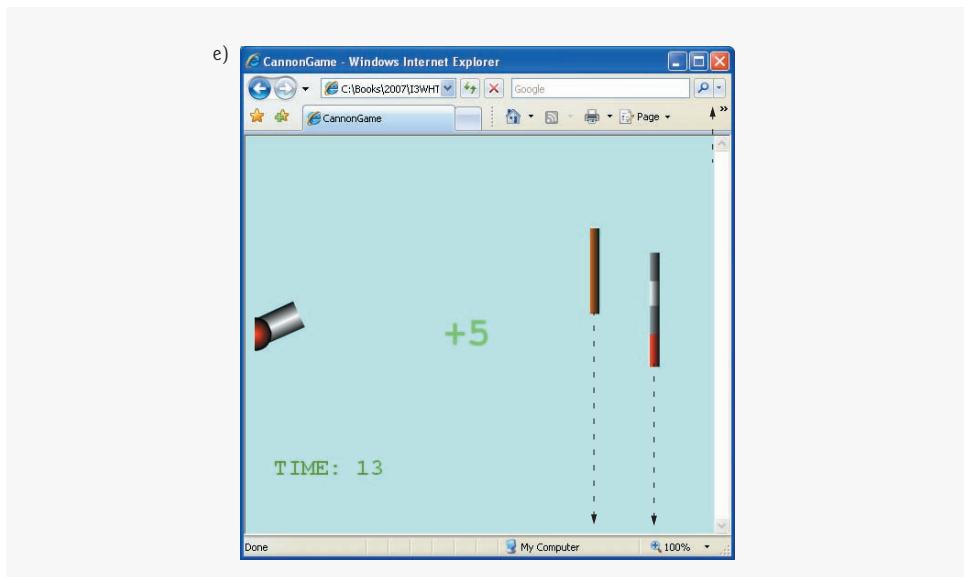
While Adobe Flash CS3 is useful for creating short animations, it is also capable of building large, interactive applications. In this chapter, we build a fully functional interactive video game. First, download the Chapter 17 examples from [www.deitel.com/books/iw3htp4](http://www.deitel.com/books/iw3htp4). Then, open `FullCannon.swf` and run the completed game. In the cannon game, the player has a limited amount of time to hit every part of a moving target. Hitting the target increases the remaining time, and missing the target or hitting the blocker decreases it. Some elements of the `FullCannon.swf` game are not discussed in the body of the chapter, but are presented as supplementary exercises. This case study will sharpen the Flash skills you acquired in Chapter 16 and introduce you to more advanced ActionScript. For this case study, we assume that you are comfortable with the material on Flash in Chapter 16. The completed game should run similar to what is shown in Fig. 17.1. Notice how in



**Fig. 17.1** | Ball fired from the cannon and hitting the target. (Part I of 3.)



**Fig. 17.1** | Ball fired from the cannon and hitting the target. (Part 2 of 3.)



**Fig. 17.1** | Ball fired from the cannon and hitting the target. (Part 3 of 3.)

Fig. 17.1(c), before the ball collides with the target, the timer displays 9 seconds on the clock. Once the topmost part of the target is hit, 5 seconds are added onto the clock but the clock displays only 13 seconds in Fig. 17.1(d). This is because one second has already passed in this duration, causing the timer to decrease from 14 to 13.

## 17.2 Object-Oriented Programming

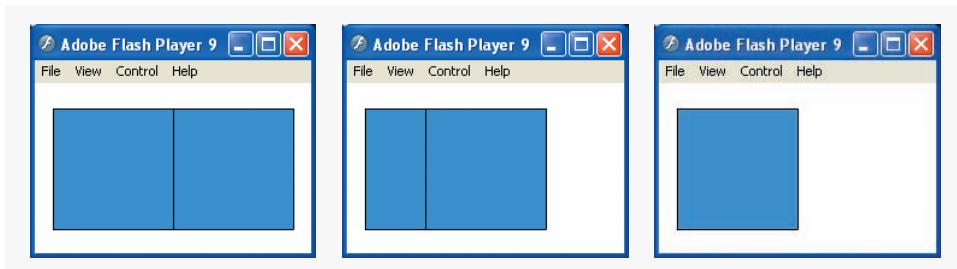
ActionScript 3.0 is an object-oriented scripting language that closely resembles JavaScript. The knowledge you gained from the JavaScript treatment in Chapters 6–11 will help you understand the ActionScript used in this case study.

An ActionScript class is a collection of characteristics known as **properties** and behaviors known as **functions**. You can create your own classes or use any of Flash's predefined classes. A symbol stored in the **Library** is a class. A class can be used to create many objects. For example, when you created the rotating-book movie clips in the preloader exercise in Chapter 16, you created a class. Dragging a symbol from the **Library** onto the stage created an instance (object) of the class. Multiple instances of one class can exist on the stage at the same time. Any changes made to an individual instance (resizing, rotating, etc.) affect only that one instance. Changes made to a class (accessed through the **Library**), however, affect every instance of the class.

## 17.3 Objects in Flash

In this section, we introduce object-oriented programming in Flash. We also demonstrate dynamic positioning (i.e., moving an object). We create two boxes. Each time you click on the left box, the right box will move to the left.

Start by creating a new Flash document named `Box.fla`. Set the movie's dimensions to 230 px wide and 140 px high. On the stage, draw a 100-px-wide blue box with a black



**Fig. 17.2** | Dynamic positioning.

outline and convert it into a movie-clip symbol. You can name this symbol **box**, but that is not necessary. Next, select the box on the stage and delete it, leaving the stage empty. This step ensures that the box will be added using the ActionScript code and that the box is not already on the stage. Now, create a new **ActionScript File** from the **File > New** menu, save it as **BoxCode.as** in the same directory as **Box.fla**, and add the code in Fig. 17.3.

The properties **x** and **y** refer to the respective **x**- and **y**-coordinates of the boxes. The two **imports** (lines 5–6) at the top of the code allow for the code to utilize those two classes, which in this case are **MouseEvent** and **Sprite**, both of which are built-in Flash classes. Inside the class, the two **Box** instances are declared. By declaration of the two **Box** objects at the beginning of the class (lines 11–12), they become instance variables and have scope through the entire class. Once the boxes have been allocated positions (lines 18–21), they must be placed on the stage using the **addChild** function (lines 23–24). The function **handleClick** is called every time the user clicks **box1**. The **addEventListener** function, which is invoked by **box1**, specifies that **handleClick** will be called whenever **box1** is clicked (line 27).

```

1 // Fig. 17.2: BoxCode.as
2 // Object animation in ActionScript.
3 package
4 {
5 import flash.events.MouseEvent; // import MouseEvent class
6 import flash.display.Sprite; // import Sprite class
7
8 public class BoxCode extends Sprite
9 {
10 // create two new box objects
11 public var box1 = new Box();
12 public var box2 = new Box();
13
14 // initialize Box coordinates, add Boxes
15 // to the stage and register MOUSE_DOWN event handler
16 public function BoxCode() : void
17 {
18 box1.x = 15; // set box1's x-coordinate
19 box1.y = 20; // set box1's y-coordinate
20 box2.x = 115; // set box2's x-coordinate
21 box2.y = 20; // set box2's y-coordinate

```

**Fig. 17.3** | Object animation in ActionScript. (Part I of 2.)

```
22 addChild(box1); // add box1 to the stage
23 addChild(box2); // add box2 to the stage
24
25 // handleClick is called when box1 is clicked
26 box1.addEventListener(MouseEvent.MOUSE_DOWN, handleClick);
27 } // end BoxCode constructor
28
29
30 // move box2 5 pixels to the left whenever box1 is clicked
31 private function handleClick(args : MouseEvent)
32 {
33 box2.x -= 5;
34 } // end function handleClick
35 } // end class BoxCode
36 } // end package
```

Fig. 17.3 | Object animation in ActionScript. (Part 2 of 2.)

To test the code, return to `Box.fla`. In the **Library** panel, right click the `Box` symbol and select **Linkage**. In the pop-up box, check off the box next to **Export for ActionScript** and type `Box` in the space provided next to **Class**. Ignore Flash's warning that a definition for this class doesn't exist in the classpath. Once you return to the stage, go to the **Property Inspector** panel and in the space next to **Document Class**, type `BoxCode` and press *Enter*. Now, the `BoxCode` ActionScript file has been linked to this specific Flash document. Type `<Ctrl>-Enter` to test the movie.

## 17.4 Cannon Game: Preliminary Instructions and Notes

Open the template file named `CannonTemplate.fla` from Chapter 17's examples folder. We'll build our game from this template. For this case study, the graphics have already been created so that we can focus on the ActionScript. We created all the images using Flash. Chapter 16 provides a detailed coverage of Flash's graphical capabilities. Take a minute to familiarize yourself with the symbols in the **Library**. Note that the **target** movie clip has movie clips within it. Also, the **ball**, **sound**, **text** and **scoreText** movie clips have **stop** actions and labels already in place. Throughout the game, we play different sections of these movie clips by referencing their frame labels. The **stop** action at the end of each section ensures that only the desired animation will be played.

### *Labeling Frames*

Before writing any ActionScript to build the game, we must label each frame in the main timeline to represent its purpose in the game. First, add a keyframe to frames 2 and 3 of the **Labels** layer. Select the first frame of the **Labels** layer and enter `intro` into the **Frame Label** field in the **Property Inspector**. A flag should appear in the corresponding box in the **timeline**. Label the second frame `game` and the third frame `end`. These labels will provide useful references as we create the game.

### *Using the Actions Layer*

In our game, we use an **Actions** layer to hold any ActionScript attached to a specific frame. ActionScript programmers often create an **Actions** layer to better organize Flash movies.

Add keyframes in the second and third frame of the **Actions** layer, and place a `stop` function in all three frames.

## 17.5 Adding a Start Button

Most games start with an introductory animation. In this section, we create a simple starting frame for our game (Fig. 17.1(a)).

Select the first frame of the **Intro/End** layer. From the **Library**, drag the **introText** movie clip and the **Play** button onto the stage. Resize and position both objects any way you like. Set the **Play** button's instance name to **playButton**. Don't worry that **introText** is invisible when deselected; it will fade in when the movie is viewed.

Test the movie. The text effects were created by manipulating alpha and gradient values with shape tweening. Explore the different symbols in the **Library** to see how they were created. Now, in the first frame of the **Actions** layer, add the code shown in Fig. 17.4 in the **Actions** panel. When the **Play** button is clicked, the movie will now play the second frame, labeled **game**.

```

1 // Fig. 17.4: Handle playButton click event.
2
3 // call function playFunction when playButton is clicked
4 playButton.addEventListener(MouseEvent.MOUSE_DOWN, playFunction);
5
6 // go to game frame
7 function playFunction(event : MouseEvent) : void
8 {
9 gotoAndPlay("game");
10 } // end function playFunction

```

**Fig. 17.4** | Handle **playButton** click event.

## 17.6 Creating Moving Objects

### *Adding the Target*

In our game, the player's goal is to hit a moving target, which we create in this section. Create a keyframe in the second frame of the **Target** layer, then drag an instance of the **target** movie clip from the **Library** onto the stage. Using the **Property Inspector**, position the target at the *x*- and *y*-coordinates 490 and 302, respectively. The position (0, 0) is located in the upper-left corner of the screen, so the target should appear near the lower-right corner of the stage. Give the **target** symbol the instance name **target**. Right click the **target** symbol in the **Library** and select **Linkage**. In the box that pops up, select **Export for ActionScript** and enter **Target** in the **Class** field.

The **target** symbol is now linked with a class named **Target**. Create a new **ActionScript File** from the **File > New** menu. Save this file immediately and give it the name **Target.as**. This will serve as the **Target** class definition. In this file, add the code in Fig. 17.5.

The **Target** class has four instance variables—the speed of the **Target** (**speed**), the direction of the **Target** (**upDown**), the number of times the **Target** has been hit by the ball (**hitCounter**), and the **Timer** variable (**moveTargetTimer**). We specify that **moveTargetTimer** is a **Timer** using the colon syntax in line 18. The first parameter of the **Timer** constructor is the delay between timer events in milliseconds. The second parameter is the

```
1 // Fig. 17.5: Target.as
2 // Move target, set direction and speed,
3 // and keep track of number of blocks hit.
4 package
5 {
6 // import relevant classes
7 import flash.display.MovieClip;
8 import flash.events.TimerEvent;
9 import flash.utils.Timer;
10
11 public class Target extends MovieClip
12 {
13 var speed; // speed of Target
14 var upDown; // direction of Target
15 var hitCounter; // number of times Target has been hit
16
17 // timer runs indefinitely every 33 ms
18 var moveTargetTimer : Timer = new Timer (33, 0);
19
20 // register function moveTarget as moveTargetTimer's
21 // event handler, start timer
22 public function Target() : void
23 {
24 moveTargetTimer.addEventListener (
25 TimerEvent.TIMER, moveTarget);
26 moveTargetTimer.start(); // start timer
27 } // end Target constructor
28
29 // move the Target
30 private function moveTarget(t : TimerEvent)
31 {
32 // if Target is at the top or bottom of the stage,
33 // change its direction
34 if (y > 310)
35 {
36 upDown = -1; // change direction to up
37 } // end if
38
39 else if (y < 90)
40 {
41 upDown = 1; // change direction to down
42 } // end else
43
44 y += (speed * upDown); // move target
45 } // end function moveTarget
46
47 // set direction of the Target
48 public function setUpDown(newUpDown : int)
49 {
50 upDown = newUpDown;
51 } // end function setUpDown
52 }
```

Fig. 17.5 | Move target, set direction and speed, and track number of blocks hit. (Part I of 2.)

```

53 // get direction of the Target
54 public function getUpDown() : int
55 {
56 return upDown;
57 } // end function getUpDown
58
59 // set speed of the Target
60 public function setSpeed (newSpeed : int)
61 {
62 speed = newSpeed;
63 } // end function setSpeed
64
65 // get speed of the Target
66 public function getSpeed() : int
67 {
68 return speed;
69 } // end function getSpeed
70
71 // set the number of times the Target has been hit
72 public function setHitCounter(newCount : int)
73 {
74 hitCounter = newCount;
75 } // end setHitCounter function
76
77 // return the number of times the Target has been hit
78 public function getHitCounter () : int
79 {
80 return hitCounter;
81 } // end function getHitCounter
82
83 // stop moveTargetTimer
84 public function stopTimers() : void
85 {
86 moveTargetTimer.stop();
87 }
88 } // end class Target
89 } // end package

```

**Fig. 17.5** | Move target, set direction and speed, and track number of blocks hit. (Part 2 of 2.)

number of times the `Timer` should repeat. A value of 0 means that the `Timer` will run indefinitely. The constructor function (lines 22–27) activates `moveTargetTimer`, which in turn calls the `moveTarget` function (lines 30–45) to move the `Target` every 33 milliseconds. The `moveTarget` function contains a nested `if...else` statement (lines 34–42) that sets `upDown` to -1 (up) when the target reaches the bottom of the screen and sets `upDown` to 1 (down) when it reaches the top of the screen. It does this by testing if the target's *y*-coordinate is greater than 310 or less than 90. [Note: The property *y* refers specifically to the *y*-coordinate of the small white circle that appears on the main stage.] Since the stage is 400 pixels high and the target is 180 pixels high (half of which is below its *y*-coordinate), when the target's *y*-coordinate is equal to 310, the bottom end of the target is even with bottom of the stage. Similar logic applies when the target is at the top of the stage.

Line 44 moves the target by incrementing its *y*-coordinate by the result of `getSpeed()`\* `setUpDown`. The remaining functions in this class are the public *get* and *set* functions for the `upDown`, `speed` and `hitCounter` variables. These allow us to retrieve and set the values outside of the class. The `stopTimers` function allows us to stop the `moveTargetTimer` from outside of the class.

Now, we can enable the target on stage, `target`, to move vertically simply by adding the calling methods `setSpeed`, `setUpDown` and `setHitCounter` in the second frame of the **Actions** layer:

```
target.setSpeed(8);
targetsetUpDown(-1);
target.setHitCounter(0);
```

Now, test the movie to see the target oscillate between the top and bottom of the stage.

### *Adding the Blocker*

An additional moving object is used to block the ball, increasing the game's difficulty. Insert a keyframe in the second frame of the **Blocker** layer and drag an instance of the **blocker** object from the **Library** onto the stage. Give this **blocker** instance the name **blocker**. Set the blocker instance's *x*- and *y*-coordinates to 415 and 348, respectively. Create a **Blocker.as** file and class and link it to the **blocker** symbol. In this file, add the code in Fig. 17.6.

```
1 // Fig. 17.6: Blocker.as
2 // Set position and speed of Blocker.
3 package
4 {
5 // import relevant classes
6 import flash.display.MovieClip;
7 import flash.events.TimerEvent;
8 import flash.utils.Timer;
9
10 public class Blocker extends MovieClip
11 {
12 var speed : int; // speed of Blocker
13 var upDown : int; // direction of Blocker
14 var moveBlockerTimer : Timer = new Timer (33, 0);
15
16 // call function moveBlocker as moveBlockerTimer event handler
17 public function Blocker() : void
18 {
19 moveBlockerTimer.addEventListener (
20 TimerEvent.TIMER, moveBlocker);
21 moveBlockerTimer.start();
22 } // end Blocker constructor
23
24 // move the Blocker
25 private function moveBlocker(t : TimerEvent)
26 {
27 // if Blocker is at the top or bottom of the stage,
28 // change its direction
```

**Fig. 17.6** | Set position and speed of Blocker. (Part I of 2.)

```

29 if (y > 347.5)
30 {
31 upDown = -1;
32 } // end if
33
34 else if (y < 52.5)
35 {
36 upDown = 1;
37 } // end else
38
39 y += getSpeed() * upDown;
40 } // end function moveBlocker
41
42 // set speed for the Blocker
43 public function setSpeed (v : int)
44 {
45 speed = v;
46 } // end function setSpeed
47
48 // get speed of the Blocker
49 public function getSpeed() : int
50 {
51 return speed;
52 } // end function getSpeed
53
54 // set direction for the Blocker
55 public function setUpDown(newUpDown : int)
56 {
57 upDown = newUpDown;
58 } // end function setUpDown
59
60 // get direction of the Blocker
61 public function getUpDown() : int
62 {
63 return upDown;
64 } // end function getUpDown
65
66 // stop moveBlockerTimer
67 public function stopTimers() : void
68 {
69 moveBlockerTimer.stop();
70 }
71 } // end class Blocker
72 } // end package

```

**Fig. 17.6** | Set position and speed of Blocker. (Part 2 of 2.)

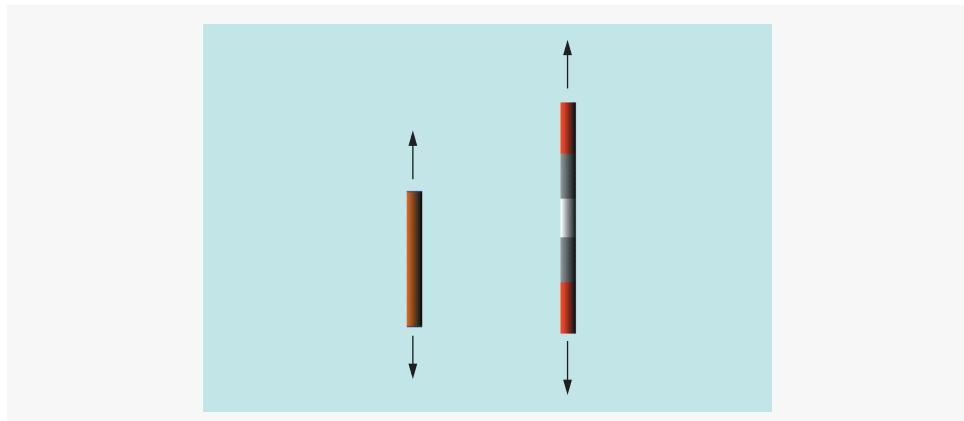
This code is very similar to that of the Target.as. Add the following code in the second frame of the **Actions** layer to set the speed and direction of the blocker:

```

blocker.setSpeed(5);
blockersetUpDown(1);

```

Test the movie. The blocker and target should both oscillate at different speeds (Fig. 17.7).

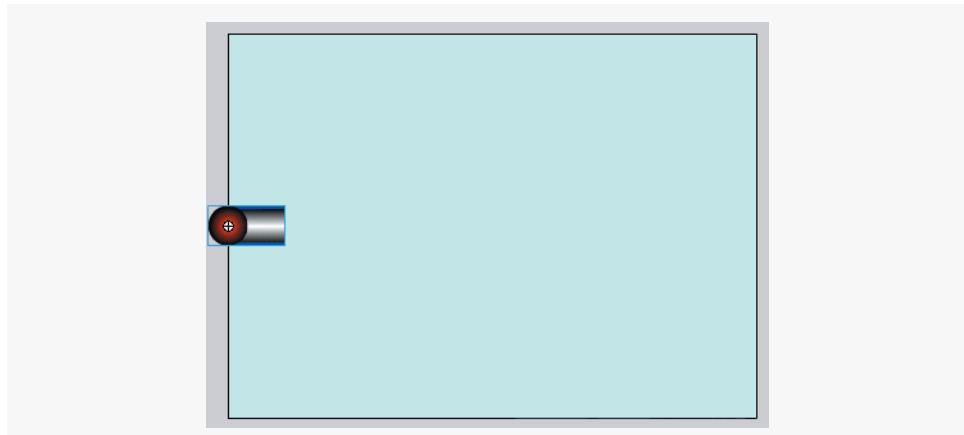


**Fig. 17.7** | Oscillating blocker and target.

## 17.7 Adding the Rotating Cannon

Many Flash applications include animation that responds to mouse-cursor motions. In this section, we discuss how to make the cannon's barrel follow the cursor, allowing the player to aim at the moving target. The skills you learn here can be used to create many effects that respond to cursor movements.

Add a keyframe to the second frame of the **Cannon** layer and drag the **cannon** object from the **Library** onto the stage. Set its *x*- and *y*-coordinates to 0 and 200. Give this cannon instance the name **cannon**. The cannon should appear in the middle of the stage's left edge (Fig. 17.8).



**Fig. 17.8** | Cannon position.

### *Coding the Cannon's Rotation*

Now, add the code from Fig. 17.9 to the second frame of the **Actions** layer. This code rotates the cannon barrel to point toward the cursor. The *x*- and *y*-coordinates of the cursor are directly accessed using the `stage.mouseX` and `stage.mouseY` properties. The code ex-

```

1 // Fig. 17.9: Register mouseInHandler as MOUSE_MOVE event handler.
2 stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseInHandler);
3
4 // rotate cannon when mouse is moved
5 function mouseInHandler(args : MouseEvent) : void
6 {
7 // rotates cannon if cursor is within stage
8 if ((stage.mouseX > 0) && (stage.mouseY > 0) &&
9 (stage.mouseX < 550) && (stage.mouseY < 400))
10 {
11 // adjust cannon rotation based on cursor position
12 var angle = Math.atan2((stage.mouseY - 200), stage.mouseX);
13 cannon.rotation = angle * (180 / Math.PI);
14 } // end if
15 } // end function mouseInHandler

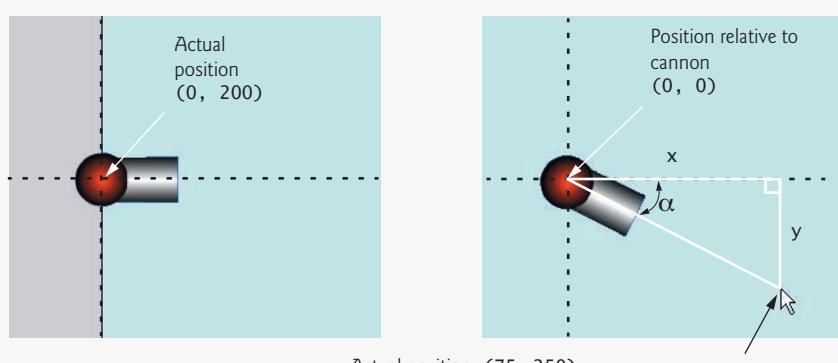
```

**Fig. 17.9** | Register mouseInHandler as MOUSE\_MOVE event handler.

executes any time the player moves the cursor, so the cannon always points toward the mouse cursor. The addEventListener function of the stage object registers the mouseInHandler function as the event handler of the MOUSE\_MOVE event (line 2). Inside the mouseInHandler function, an if statement (lines 8–9) checks whether the cursor is within the stage. If it is, the code adjusts the cannon's rotation so that it points toward the cursor (line 13). The **rotation property** (line 13) controls an object's rotation, assuming its natural orientation to be 0 degrees.

ActionScript's **Math class** contains various mathematical functions and values that are useful when performing complex operations. For a full list of the Math class's functions and values, refer to the **Flash Help** in the **Help** menu. We use the Math class to help us compute the rotation angle required to point the cannon toward the cursor.

First, we need to find the cursor's coordinates relative to the cannon. Subtracting 200 from the cursor's y-coordinate gives us the cursor's vertical position, assuming (0, 0) lies at the cannon's center (Fig. 17.10). We then determine the desired angle of rotation. Note the



**Fig. 17.10** | Trigonometry of the **cannon** object.

right triangle created by the cannon and the cursor in Fig. 17.10. From trigonometry, we know that the tangent of angle  $\alpha$  equals the length of side  $y$  divided by side  $x$ :  $\tan(\alpha) = y/x$ . We want the value of  $\alpha$ , though, not the value of  $\tan(\alpha)$ . Since the arc tangent is the inverse of the tangent, we can rewrite this equation as  $\alpha = \arctan(y/x)$ . The `Math` object provides us with an arc tangent function: `Math.atan2(y, x)`. This function returns a value, in radians, equal to the angle opposite side  $y$  and adjacent to side  $x$  (line 12). Radians are a type of angle measurement similar to degrees that range from 0 to  $2\pi$  instead of 0 to 360. To convert from radians to degrees, we multiply by  $180/\pi$  (line 13). The constant `Math.PI` provides the value of  $\pi$ . Since this rotation adjustment is performed every time the mouse moves within the stage, the cannon barrel appears to constantly point at the cursor. Test the movie to observe this effect.



### Error-Prevention Tip 17.1

*If your code is not working and no error message displays, ensure that every variable points to the correct object. One incorrect reference can prevent an entire function from operating correctly.*

### Hiding the Cannon Layer

We won't make any other changes to the **Cannon** layer. Hide the **Cannon** layer by selecting the **show/hide** selector (dot) in the portion of the **Timeline** to the right of the layer name (Fig. 17.11). A red **x** should appear in place of the dot to indicate that the layer is hidden while editing the movie. The layer will still be visible when the movie is viewed. Clicking the **show/hide** **x** again makes the **Cannon** layer visible.

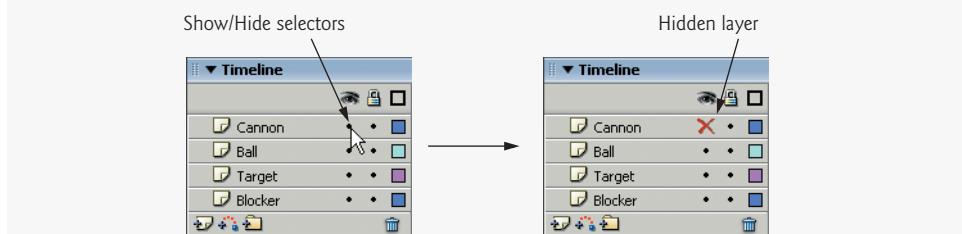


Fig. 17.11 | Using selectors to show/hide layers.

## 17.8 Adding the Cannonball

In this section, we add the cannonball to our game. Create a keyframe in frame 2 of the **Ball** layer, then drag the **ball** symbol from the **Library** onto the stage. Give the **ball** object the instance name **ball**. Notice that the ball instance appears as a small white circle on the stage. This circle is Flash's default appearance for a movie clip that has no graphic in its first frame. The ball will appear hidden beneath the cannon when the movie is viewed, because it is on a lower layer. Provide the **Ball** object with the  $x$ - and  $y$ -coordinates 0 and 200, respectively. This places the ball right under the cannon, so that when the ball is fired, it appears to have been fired from inside the cannon.

### Initializing the Ball's Motion Variables

Link the **ball** symbol to a **Ball** class, as we did previously with the **Target** and **Blocker** classes. Next, create a new ActionScript 3.0 file named **Ball.as** and add the code shown in Fig. 17.12 to the file.

This code defines the `Ball` class. It has three properties—the speed in the *x*-direction, `speedX` (line 12), the speed in the *y*-direction, `speedY` (line 13), and a timer that moves the ball, `moveBallTimer` (line 16). Since the speed in the *x*-direction will be only integer values, it is of type `int`. However, the speed in the *y*-direction is also dependent on `fireRatio`, which can be a decimal value, and thus, `speedY` is of type `Number`, which is ActionScript 3's floating-point variable type. The class definition also creates the *get* and *set* functions for these properties. When the `Ball` object is created, the `Ball` constructor function starts the `moveBallTimer`, which calls the `moveBall` function every 33 ms. Function `moveBall` (lines 28–32) increments the *x*- and *y*-coordinates by `speedX` and `speedY`. The `stopTimers` function allows us to stop the `moveBallTimer` from outside of the class.

```

1 // Fig. 17.12: Ball.as
2 // Move ball and set speed.
3 package
4 {
5 // import relevant classes
6 import flash.display.MovieClip;
7 import flash.events.TimerEvent;
8 import flash.utils.Timer;
9
10 public class Ball extends MovieClip
11 {
12 var speedX : int; // speed in x-direction
13 var speedY : Number; // speed in y-direction
14
15 // Create Timer object to move ball
16 var moveBallTimer : Timer = new Timer(33, 0);
17
18 // Ball constructor starts moveBallTimer
19 public function Ball() : void
20 {
21 // call function moveBall as moveBallTimer event handler
22 moveBallTimer.addEventListener(
23 TimerEvent.TIMER, moveBall);
24 moveBallTimer.start();
25 } // end Ball constructor
26
27 // update the x and y coordinates using the specific speeds
28 private function moveBall(t : TimerEvent) :
29 {
30 x += speedX;
31 y += speedY;
32 } // end function moveBall
33
34 // set speed in x direction
35 public function setSpeedX(v : int) :
36 {
37 speedX = v;
38 } // end function setSpeedX
39

```

**Fig. 17.12** | Move ball and set speed. (Part I of 2.)

```

40 // get speed in x direction
41 public function getSpeedX() : int
42 {
43 return speedX;
44 } // end function getSpeedX
45
46 // set speed in y direction
47 public function setSpeedY(v : int, fireRatio : Number)
48 {
49 speedY = v * fireRatio;
50 } // end function setSpeedY
51
52 // get speed in y direction
53 public function getSpeedY() : Number
54 {
55 return speedY;
56 } // end function getSpeedY
57
58 public function stopTimers() : void
59 {
60 moveBallTimer.stop();
61 } // end function stopTimer
62 } // end class Ball
63 } // end package

```

**Fig. 17.12** | Move ball and set speed. (Part 2 of 2.)

### Scripting the Ball's Motion

In the second frame of the **Actions** layer, add the code in Fig. 17.13. The new code moves the ball along a straight line in the direction the cannon was pointing when the mouse was clicked.

```

1 // Fig. 17.13: Fire ball on click event.
2 var firing : Boolean = false; // is ball firing?
3 var exploding : Boolean = false; // is ball exploding?
4 var fireRatio : Number = 0; // firing direction of ball
5 var speed : int = 30; // speed of ball
6 ball.setSpeedX(0);
7 ball.setSpeedY(0, 0);
8
9 // register function mouseDownHandler as MOUSE_DOWN event handler
10 stage.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownHandler);
11
12 function mouseDownHandler(args : MouseEvent) : void
13 {
14 // if the mouse is within the stage and the ball has not been fired or
15 // exploded yet, fire ball toward mouse cursor
16 if ((!firing) && (!exploding) && (stage.mouseX > 0) &&
17 (stage.mouseY > 0) && (stage.mouseX < 550) &&
18 (stage.mouseY < 400))
19 {

```

**Fig. 17.13** | Fire ball on click event. (Part 1 of 2.)

```

20 firing = true;
21 fireRatio = (stage.mouseY - 200) / stage.mouseX;
22 ball.setSpeedX(speed);
23 ball.setSpeedY(speed, fireRatio);
24 } // end if
25 } // end function mouseDownHandler

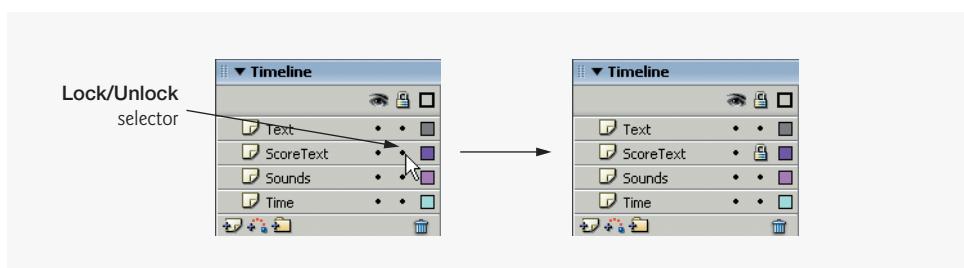
```

**Fig. 17.13** | Fire ball on click event. (Part 2 of 2.)

This code initializes four variables—`firing`, `exploding`, `fireRatio` and `speed` (lines 2–5). Variables `firing` and `exploding` are set to `false` to signify that the ball is not moving or exploding. Later, we will set `exploding` to `true` and play a brief explosion animation upon collision with the target or the blocker. Variables `fireRatio` and `speed` specify the ball’s firing direction and speed, respectively. The function `addEventListener` (line 10) registers the function `mouseDownHandler` (lines 12–25) as the event handler for the stage’s `MOUSE_DOWN` event. The `if` statement in the `mouseDownHandler` function (lines 16–24) checks that the ball is not currently in flight (`!firing`) or exploding (`!exploding`), and the mouse is within the stage (lines 16–18). If the condition evaluates to true, `firing` is set to `true` (line 20), and `fireRatio` is set to the mouse’s *y*-coordinate (relative to the cannon) divided by its *x*-coordinate (line 21). This `fireRatio` will move the ball toward the cursor’s position when it is fired. Lines 22–23 set the ball’s speed in the *x*-direction to `speed`, and the ball’s speed in the *y*-direction to `speed * fireRatio`. The expression `speed * fireRatio` returns the appropriate change in *y* based on the given change in *x* (`speed`).

## 17.9 Adding Sound and Text Objects to the Movie

Next, we add sound and text to our movie. Add a keyframe to frame 2 of the **Text** layer and drag the **text** symbol from the **Library** onto the stage. Note that the **text** object, like the ball, is represented by a small white dot. This dot will not appear when the movie is viewed. Position the **text** symbol in the center of the stage at coordinate (275, 200), and name the instance **text**. Then, add a keyframe to the second frame of the **Sounds** and **ScoreText** layers, and add an instance of the **sound** and **scoreText** objects, respectively. Center both objects on the stage and give them instance names matching their symbol names—**sound** and **scoreText**. Lock these three layers by clicking the **lock/unlock** selector (dot) to the right of the layer name in the timeline (Fig. 17.14). When a layer is locked,



**Fig. 17.14** | Lock/Unlock layers.

its elements are visible, but they cannot be edited. This allows you to use one layer's elements as visual references for designing another layer, while ensuring that no elements are moved unintentionally. To unlock a layer, click the lock symbol to the right of the layer name in the **Timeline**.

All the necessary sound and text capabilities have now been added to the game. In the next section, we add the time counter to the game.

## 17.10 Adding the Time Counter

What makes the cannon game challenging is the limited amount of time the player has to completely destroy the target. Time, whether increasing or decreasing, is an important aspect of many games and applications. In this section, we discuss adding a time counter that decreases as the game progresses.

### *Adding the Time Box*

In the cannon game, the player has a limited amount of time to destroy every section of the target. The amount of time remaining appears in a dynamic text box in the bottom-left corner of the screen. To create this dynamic text box, first add a keyframe to frame 2 of the **Time** layer and add an instance of the **time** symbol to the stage. Position it at coordinate (100, 365), which should be in the lower-left corner of the stage. Name the instance **time**. Open the **time** symbol from the **Library** and select the text field. In the **Property Inspector**, change the text type to **Dynamic Text** and name the instance **timeText**. Return to the main scene, and add the code in Fig. 17.15 to the **Actions** panel in the second frame.

The variable **timer** is initialized to 10 (line 2). This variable will hold the amount of time remaining. The **countTimer** (lines 3–7) calls the **countDown** function (lines 9–18) each second. The **countDown** function decrements the **timer** by 1, and also sets the **text** of the **time** symbol's **timeText** element to "TIME: ", followed by the current value of **timer**.

```

1 // Fig. 17.15: Timer countdown.
2 var timer = 10; // set initial timer value to 10 seconds
3 var countTimer : Timer = new Timer(1000, 0);
4
5 // call function countDown as countTimer event handler
6 countTimer.addEventListener(TimerEvent.TIMER, countDown);
7 countTimer.start(); // start Timer
8
9 function countDown(t : TimerEvent)
10 {
11 --timer; // decrement timer by 1 every second
12 time.timeText.text = "TIME: " + timer;
13 // player loses game if he/she runs out of time
14 if(!firing) && timer <= 0)
15 {
16 gotoAndPlay("end"); // call end frame sequence
17 } // end if
18 } // end function countDown

```

Fig. 17.15 | Timer countdown.

Lines 14–17 test whether time has run out and the ball is not firing. If the condition is true, the movie will skip to the (empty) end frame.

Test the movie. The time should decrease from 10 to 0. When the time reaches 0, the **end** frame should play. Because none of the functions or objects in the second frame currently exist in the third frame, the timers that are still active will try to call those functions and fail, returning an error message. To fix this, we must first include all of the objects on the stage in the second frame in the third frame. You can do this by inserting a new frame (not keyframe) in the third frame of the **Text**, **ScoreText**, **Sounds**, **Time**, **Cannon**, **Ball**, **Target**, and **Blocker** layers. Next, we must stop all timers and hide most of the elements at the beginning of the third frame. Add the code in Fig. 17.16 to the third frame of the **Actions** layer.

Lines 2–3 remove the event listeners for the stage's **MOUSE\_MOVE** and **MOUSE\_DOWN** events, which are no longer needed. Lines 4–7 stop all of the timers that we have used, either by accessing the timer's **stop** method directly (line 4), or by accessing the class's **stopTimers** method (lines 5–7). Lines 8–13 hide every element on the stage by playing the **hidden** frame of each element, which is an empty frame.

```
1 // Fig. 17.16: Stops all timers and sends objects into hidden frame.
2 stage.removeEventListener(MouseEvent.MOUSE_MOVE, mouseInHandler);
3 stage.removeEventListener(MouseEvent.MOUSE_DOWN, mouseDownHandler);
4 countTimer.stop();
5 blocker.stopTimers();
6 ball.stopTimers();
7 target.stopTimers();
8 blocker.gotoAndPlay("hidden");
9 cannon.gotoAndPlay("hidden");
10 ball.gotoAndPlay("hidden");
11 target.gotoAndPlay("hidden");
12 time.gotoAndPlay("hidden");
13 scoreText.gotoAndPlay("hidden");
```

**Fig. 17.16** | Stops all timers and sends objects into hidden frame.

### *Creating a Final Animation Sequence*

Games generally have a final animation sequence that informs the player of the outcome of the game. In this section, we create a final animation sequence for the game.

First, we must create a winner boolean to keep track of whether the player has won or lost the game. To do this, add the following code to the second frame of the **Actions** layer.

```
var winner : Boolean = false; // Keep track of who won
```

Next, add the code in Fig. 17.17 to the third frame of the **Actions** layer. This **if...else** statement checks the **winner** variable. If **winner** is true, the **text** movie clip goes to the **win** frame. Otherwise **text** goes to the **lose** frame. Test the movie again. When the time runs out, the **lose** frame, containing the text **Game Over**, should appear on an otherwise blank stage.

```

1 // Fig. 17.17: Check winner and show "Winner" or "Game Over".
2 if (winner == true)
3 {
4 text.gotoAndPlay("win");
5 }
6
7 else
8 {
9 text.gotoAndPlay("lose");
10}

```

**Fig. 17.17** | Check winner and show **Winner** or **Game Over**.

## 17.11 Detecting a Miss

We now add code to the **ball** instance that detects when the ball has moved outside of the stage. First, add the **checkBall** function in Fig. 17.18 to the second frame of the **Actions** panel.

Lines 5–6 test whether the ball is outside the bounds of the stage. If it is, the **checkBallTimer** is stopped (line 8), because the ball is no longer in motion and does not need to be checked until it has been fired again. Boolean **firing** is set to **false** (line 9). Then, lines 10–12 play the **text** movie clip's **miss** frame, the **scoreText** movie clip's **minusTwo** frame and the **sound** movie clip's **miss** frame. This will display **MISS** and **-2** on the stage, and play the miss sound. Also, the **timer** variable is decreased by 2 (line 13). Finally, the **Ball** object is reset to its starting position and speed (lines 14–17).

In order to check the ball at regular intervals, we create a timer that calls **checkBall** every 33 ms. First, add the following code to the second frame of the **Actions** layer:

```

// Check ball at 33-ms intervals
var checkBallTimer : Timer = new Timer(33, 0);

```

```

1 // Fig. 17.18: Detecting a miss.
2 function checkBall (e : TimerEvent)
3 {
4 // if ball is not inside stage, go through miss sequence
5 if ((ball.x < 0) || (ball.y < 0) || (ball.x > 550) ||
6 (ball.y > 400))
7 {
8 checkBallTimer.stop(); // stop checkBallTimer
9 firing = false; // ball is no longer being fired
10 text.gotoAndPlay("miss"); // display miss on the stage
11 scoreText.gotoAndPlay ("minusTwo"); // display "-2"
12 sound.gotoAndPlay ("miss"); // miss sound is played
13 timer -= 2; // deduct 2 seconds from timer
14 ball.x = 0; // set ball back to initial x-coordinate
15 ball.y = 200; // set ball back to initial y-coordinate
16 ball.setSpeedX(0); // set ball speed in x-direction to 0
17 ball.setSpeedY(0, 0); // set ball speed in y-direction to 0
18 } // end if
19 } // end function checkBall

```

**Fig. 17.18** | Detecting a miss.

Next, we must start the timer. Since this timer needs to run only after the ball has been fired, we will start the timer in the `mouseDownHandler`. Insert the following code between lines 13 and 14 of Fig. 17.13.

```
// call function checkBall as checkBallTimer event handler
checkBallTimer.addEventListener(TimerEvent.TIMER, checkBall);
checkBallTimer.start(); // start Timer
```

We must also stop this timer at the end of the game by adding the following code to the third frame of the Actions layer.

```
checkBallTimer.stop();
```

Test the movie with your computer's sound turned on. At this point, every fired ball should travel off the stage and count as a miss. In the next few sections, we discuss how to add features that allow the player to gain time and win the game.

## 17.12 Adding Collision Detection

Before we add collision detection to the target and blocker, we add a function that handles the actions common to all of our collisions. Add the `onBallContact` function (Fig. 17.19) to the second frame of the Actions layer.

```
1 // Fig. 17.19: Common actions after collision.
2 function onBallContact(timeChange : int)
3 {
4 // adjust variables to play exploding sequence
5 exploding = true;
6 firing = false;
7 timer += timeChange; // add the amount of time passed as parameter
8 ball.gotoAndPlay("explode"); // explode the Ball object
9 ball.setSpeedX(0); // set ball speed in x-direction to 0
10 ball.setSpeedY(0, 0); // set ball speed in y-direction to 0
11
12 // give explode animation time to finish, then call resetBall
13 explodeTimer.addEventListener(TimerEvent.TIMER, resetBall);
14 explodeTimer.start();
15
16 // play appropriate sound and text based on timeChange
17 if (timeChange < 0)
18 {
19 sound.gotoAndPlay("blocked");
20 text.gotoAndPlay("blocked");
21 if (timeChange == -5)
22 {
23 scoreText.gotoAndPlay("minusFive");
24 } // end if
25 } // end if
26
27 else if (timeChange >= 0)
28 {
```

**Fig. 17.19** | Common actions after collision. (Part I of 2.)

```

29 sound.gotoAndPlay("hit");
30 text.gotoAndPlay("hit");
31
32 // increment the hitCounter by 1
33 target.setHitCounter(target.getHitCounter() + 1);
34
35 if (target.getHitCounter() >= 5)
36 {
37 // if target has been hit 5 times, then declare player winner
38 winner = true;
39 gotoAndPlay("end"); // go to third frame
40 } // end if
41
42 if (timeChange == 5)
43 {
44 scoreText.gotoAndPlay("plusFive");
45 } // end if
46
47 else if (timeChange == 10)
48 {
49 scoreText.gotoAndPlay("plusTen");
50 } // end else
51
52 else if (timeChange == 20)
53 {
54 scoreText.gotoAndPlay("plus20");
55 } // end else
56 } // end else
57 } // end function onBallContact

```

**Fig. 17.19** | Common actions after collision. (Part 2 of 2.)

The `onBallContact` function takes a `timeChange` parameter that specifies how many seconds to add or remove from the time remaining. Line 7 adds `timeChange` to the timer. Lines 8–10 tell the ball to `explode` and stop. Lines 13–14 start a timer that calls the `resetBall` function after completion. We must create this timer by adding the following code to the second frame of the **Actions** layer:

```
// Delay for ball explosion
var explodeTimer : Timer = new Timer(266, 1);
```

This timer gives the ball's `explode` animation time to complete before it calls `resetBall`. We must stop this timer at the end of the game, by adding the following code to the third frame of the **Actions** layer.

```
explodeTimer.stop();
```

The `resetBall` function (Fig. 17.20) sets `exploding` to `false` (line 4), then resets the ball to the starting frame and position (lines 5–7). Add the `resetBall` function to the second frame of the **Actions** layer.

The `onBallContact` function (Fig. 17.19) also plays a frame from the `sound`, `text` and `scoreText` movie clips, depending on the `timeChange`, to notify the player whether

```

1 // Fig. 17.20: Reset the ball to its original position.
2 function resetBall(t : TimerEvent)
3 {
4 exploding = false; // set the ball explosion status to false
5 ball.gotoAndPlay("normal");
6 ball.x = 0; // set x-coordinate to original position
7 ball.y = 200; // set y-coordinate to original position
8 } // end function resetBall

```

**Fig. 17.20** | Reset the ball to its original position.

they hit a target or a blocker, and to show the player how many points they gained or lost (lines 17–56). Lines 35–40 test whether the target has been hit 5 times. If it has, `winner` is set to `true` and the end frame is played.

#### *Adding Collision Detection to the Target and Blocker*

Flash has a built-in `collision detection` function that determines whether two objects are touching. The function `object1.hitTestObject(object2)` returns `true` if any part of `object1` touches `object2`. Many games must detect collisions between moving objects to control game play or add realistic effects. In this game, we rely on collision detection to determine if the ball hits either the blocker or the target.

In this section, we add code to `target` and `blocker` that increases or decreases your remaining time, depending on what you hit. Note that the `target` object comprises five instances of three different symbols: one `targetCenter` (white), two `targetMiddles` (gray) and two `targetOuts` (red). The closer to the center the `target` is hit, the more seconds get added to the total time.

Add the collision detection function shown in Fig. 17.21 to the second frame of the `Actions` layer.

```

1 // Fig. 17.21: Detect collision using hitTestObject.
2 function collisionDetection()
3 {
4 if (target.out1.hitTestObject(ball) && (!exploding))
5 {
6 onBallContact (5); // hit upper outer part of target
7 target.out1.gotoAndPlay("hit");
8 } // end if
9
10 else if (target.mid1.hitTestObject(ball) && (!exploding))
11 {
12 onBallContact (10); // hit upper middle part of target
13 target.mid1.gotoAndPlay("hit");
14 } // end else
15
16 else if (target.center.hitTestObject(ball) && (!exploding))
17 {
18 onBallContact (20); // hit center of target
19 target.center.gotoAndPlay("hit");
20 } // end else

```

**Fig. 17.21** | Detect collision using `hitTestObject`. (Part 1 of 2.)

```

21
22 else if (target.mid2.hitTestObject(ball) && (!exploding))
23 {
24 onBallContact (10); // hit lower middle part of target
25 target.mid2.gotoAndPlay("hit");
26 } // end else
27
28 else if (target.out2.hitTestObject(ball) && (!exploding))
29 {
30 onBallContact (5); // hit lower outer part of target
31 target.out2.gotoAndPlay("hit");
32 } // end else
33
34 else if (blocker.hitTestObject(ball) && (!exploding))
35 {
36 onBallContact (-5);
37
38 // if timer runs out, player loses
39 if (timer < 0)
40 {
41 winner = false;
42 gotoAndPlay("end");
43 } // end if
44 } // end else
45 } // end function collisionDetection

```

**Fig. 17.21** | Detect collision using `hitTestObject`. (Part 2 of 2.)

Function `collisionDetection` consists of an `if...else` statement that tests whether the ball has hit the blocker or one of the parts of the target. It ensures that the ball is not currently exploding, to prevent the ball from hitting more than one part of the target at a time. If these conditions return true, the `onBallContact` function is called with the appropriate number of seconds to add to or subtract from the timer as the parameter. For each of the parts of the target, that part's hit animation is played upon being hit. For the blocker, an `if` statement (lines 39–43) checks whether time has run out (line 39). If it has, the `winner` is set to `false` and the movie moves to the `end` frame.

To run the `collisionDetection` function at a regular interval, we will call it from the `checkBall` function. Add the following at the beginning of the `checkBall` function.

```
collisionDetection();
```

Now test the movie. The target pieces should disappear when hit by the ball, as shown in Fig. 17.1. The player can now gain time by hitting the target and lose time by hitting the blocker. The ball should explode in the position where it hit the blocker, then reset to under the cannon, allowing the player to fire again.

## 17.13 Finishing the Game

Open the `text` symbol's editing stage from the **Library**. An action that plays the frame labeled `intro` in the main scene has already been attached to the last frame of the sections labeled `win` and `lose`. These actions, which were included in the original `CannonTemplate.fla` file, cause the game to restart after the final text animation is played.

To change the game's difficulty, adjust speed in the `blocker` and/or the `target`. Adjusting the time change in the `timeText` instance also changes the difficulty (a smaller decrement gives more time).

Congratulations! You have created an interactive Flash game. Now you can play the completed version. In the chapter exercises, you can improve the game and add additional levels of difficulty.

## 17.14 ActionScript 3.0 Elements Introduced in This Chapter

Figure 17.22 lists the Flash ActionScript 3.0 elements introduced in this chapter, which are useful in building complex Flash movies.

| Element                                                      | Description                                                                                                               |
|--------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>object.x</code>                                        | Property that refers to <i>object</i> 's <i>x</i> -coordinate.                                                            |
| <code>object.y</code>                                        | Property that refers to <i>object</i> 's <i>y</i> -coordinate.                                                            |
| <code>addChild(<i>object</i>)</code>                         | Function that adds the object to the stage.                                                                               |
| <code>addEventListener(<i>event</i>, <i>function</i>)</code> | Function that invokes another function in response to an event.                                                           |
| <code>mouseX</code>                                          | Mouse's <i>x</i> -coordinate property.                                                                                    |
| <code>mouseY</code>                                          | Mouse's <i>y</i> -coordinate property.                                                                                    |
| <code>object.rotation</code>                                 | Property that rotates the <i>object</i> .                                                                                 |
| <code>stage</code>                                           | Manipulates objects on the stage.                                                                                         |
| <code>object1.hitTestObject(<i>object2</i>)</code>           | Built-in function that determines when two objects collide                                                                |
| <code>Math</code>                                            | Built-in object that contains useful functions and properties (refer to Flash's ActionScript Dictionary for a full list). |

**Fig. 17.22** | ActionScript 3.0 elements.

## Summary

### Section 17.1 Introduction

- Adobe Flash CS3 is capable of building large, interactive applications.

### Section 17.2 Object-Oriented Programming

- ActionScript 3.0 is an object-oriented scripting language that closely resembles JavaScript. The knowledge you gained from the JavaScript treatment in Chapters 6–11 will help you understand the ActionScript used in this case study.
- An ActionScript class is a collection of characteristics known as properties and of behaviors known as functions.

- You can create your own classes or use any of Flash's predefined classes.
- A symbol stored in the **Library** is a class.
- A class can be used to create many instances, or objects, of the class.
- Dragging a symbol from the **Library** onto the stage creates an instance (object) of the class. Multiple instances of one class can exist on the stage at the same time.
- Any changes made to an individual instance (resizing, rotating, etc.) affect only that one instance.
- Changes made to a class (accessed through the **Library**), affect every instance of the class.

### ***Section 17.3 Objects in Flash***

- The properties `x` and `y` refer to the respective `x`- and `y`-coordinates of an object.
- `import` allows you to utilize built-in classes of ActionScript 3.0, such as `MouseEvent` and `Sprite`.
- Instance variables have scope through the entire class.
- Movie clips in the **Library** can be placed on the stage using the `addChild` function.
- Function `addEventListener` registers an event handler to be called when an event is triggered.

### ***Section 17.4 Cannon Game: Preliminary Instructions and Notes***

- The `stop` action at the end of a section ensures that only the desired animation will be played.
- ActionScript programmers often create an **Actions** layer to better organize Flash movies.

### ***Section 17.5 Adding a Start Button***

- Most games start with an introductory animation.

### ***Section 17.6 Creating Moving Objects***

- The first parameter of a `Timer` constructor is the delay between timer events in milliseconds. The second parameter is the number of times the `Timer` should repeat. A value of 0 means that the `Timer` will run indefinitely.

### ***Section 17.7 Adding the Rotating Cannon***

- Many Flash applications include animation that responds to mouse cursor motions.
- ActionScript's `Math` class contains various mathematical functions and values that are useful when performing complex operations. For a full list of the `Math` class's functions and values, refer to the **Flash Help** dictionary from the **Help** menu.
- The `Math` object provides us with an arc tangent function: `Math.atan2(y, x)`. This function returns a value, in radians, equal to the angle opposite side `y` and adjacent to side `x`.
- The constant `Math.PI` provides the value of  $\pi$ .
- If your code is not working and no error message displays, ensure that every variable points to the correct object. One incorrect `stage` can prevent an entire function from operating correctly.
- Hide a layer by selecting the show/hide selector (dot) in the portion of the **Timeline** to the right of the layer name. A red `x` should appear in place of the dot to indicate that the layer is hidden while editing the movie. The layer will still be visible when the movie is viewed. Clicking the show/hide `x` again makes the layer visible.

### ***Section 17.8 Adding the Cannonball***

- A small white circle is Flash's default appearance for a movie clip that has no graphic in its first frame.
- The `Number` type is ActionScript 3's floating-point variable type.

**Section 17.9 Adding Sound and Text Objects to the Movie**

- Lock a layer by clicking the lock/unlock selector (dot) to the right of the layer name in the timeline. When a layer is locked, its elements are visible, but they cannot be edited. This allows you to use one layer's elements as visual references for designing another layer, while ensuring that no elements are moved unintentionally. To unlock a layer, click the lock symbol to the right of the layer name in the Timeline.

**Section 17.10 Adding the Time Counter**

- Time, whether increasing or decreasing, is an important aspect of many games and applications.
- Games generally have a final animation sequence that informs the player of the outcome of the game.

**Section 17.12 Adding Collision Detection**

- Flash has a built-in collision detection function that determines whether two objects are touching. The function `object1.hitTestObject(object2)` returns `true` if any part of `object1` touches `object2`. Many games must detect collisions between moving objects to control game play or add realistic effects. In this game, we rely on collision detection to determine if the ball hits either the blocker or the target.

**Terminology**

|                                            |                           |
|--------------------------------------------|---------------------------|
| ActionScript 3.0                           | Math class                |
| <code>addChild</code>                      | MouseEvent                |
| <code>addEventListener</code>              | Number                    |
| Adobe Flash CS 3                           | property                  |
| <code>function</code>                      | Property Inspector        |
| <code>hitTestObject</code>                 | rotation property         |
| ActionScript collision detection<br>method | show/hide layers          |
| <code>import</code>                        | Sprite                    |
| <code>instance</code>                      | stage                     |
| instance name                              | <code>stage.mouseX</code> |
| instance variable                          | <code>stage.mouseY</code> |
| <b>Library</b>                             | stop                      |
| lock/unlock layer                          |                           |

**Self-Review Exercises**

- 17.1** State whether each of the following is *true* or *false*. If *false*, explain why.
- ActionScript 3.0 is an object-oriented scripting language that contains functions and classes.
  - There can be multiple instances of one symbol.
  - Locking a layer is the same as hiding it, except that a hidden layer can still be edited.
  - New functions can never be created in Flash. We must rely on Flash's predefined functions.
- 17.2** Fill in the blanks for each of the following statements.
- Property \_\_\_\_\_ accesses the main timeline object.
  - A movie clip with no animation in its first frame appears as a(n) \_\_\_\_\_.
  - Flash has a built-in \_\_\_\_\_ function that returns `true` when two objects touch.

## Answers to Self-Review Exercises

**17.1** a) True. b) True. d) False. Neither locked nor hidden layers can be edited. Locked layers are visible, though, whereas hidden layers are not. e) False. New functions can be created inside a package, or inside a frame's **Actions** using the keyword **function**.

**17.2** a) stage. b) small white circle. c) collision detection.

## Exercises

**17.3** Add an **instructions** button to the **intro** frame of the main scene. Make it play a brief movie clip explaining the rules of the game. The instructions should not interfere with the actual game play.

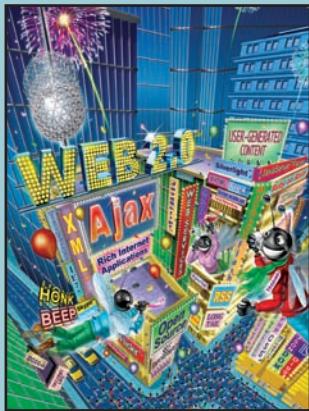
**17.4** Use Flash's **random( n )** function (discussed in Chapter 16) to assign a random speed between 1 and 4 to the blocker, and a speed between 5 and 7 to the target. Remember that **random( n )** returns a random integer between 0 and **n**.

**17.5** Add a text field to the **end** frame that displays the player's final score (i.e., the time remaining) if the player wins. Output different phrases depending on the player's final score (e.g., 1–10: Nice job, 11–15: Great!, 16–20: Amazing!). [Hint: Create a new global variable **finalTime** if the player wins. Create an **if...else** statement to determine which text phrase to use based on **finalTime**.]

**17.6** Add a second level to the game with two blockers instead of one. Try to do this without adding a fourth frame to the timeline. Instead, create a duplicate **blocker** symbol and modify it to appear invisible at first. Think about reversing the process we used to make the sections of the **target** invisible. The final score should be a combination of first- and second-round scores. [Hint: Create an instance variable **level** that stores the current level (i.e., 1 or 2). Make the second blocker visible only if **level == 2**.]

**17.7** Give a brief description of each of the following terms:

- a) Lock/unlock
- b) Instance
- c) Collision detection
- d) x and y
- e) Event handler
- f) Function



*Becoming more flexible, open-minded, having a capacity to deal with change is a good thing.*

—Eda Le Shan

*We wove a web in childhood, A web of sunny air.*

—Charlotte Brontë

*Genius is the ability to put into effect what is on your mind.*

—F. Scott Fitzgerald

*A fair request should be followed by the deed in silence.*

—Dante

*The transition from cause to effect, from event to event, is often carried on by secret steps, which our foresight cannot divine, and our sagacity is unable to trace.*

—Joseph Addison

# Adobe® Flex™ 2 and Rich Internet Applications

## OBJECTIVES

In this chapter you will learn:

- What Flex is and what it's used for.
- How to design user interfaces in Flex's user interface markup language, MXML.
- How to embed multimedia in a Flex application.
- How to use data binding to create responsive user interfaces.
- How to access XML data from a Flex application.
- Client-side scripting in ActionScript 3.0, Flex's object-oriented scripting language.
- How to interact with a web service.
- How to create an advanced user interface.
- How the Adobe Integrated Runtime allows Flex applications to run on the desktop without an Internet connection.

## Outline

- 18.1 Introduction
- 18.2 Flex Platform Overview
- 18.3 Creating a Simple User Interface
- 18.4 Accessing XML Data from Your Application
- 18.5 Interacting with Server-Side Applications
- 18.6 Customizing Your User Interface
- 18.7 Creating Charts and Graphs
- 18.8 Connection-Independent RIAs on the Desktop: Adobe Integrated Runtime (AIR)
- 18.9 Flex 3 Beta
- 18.10 Wrap-Up
- 18.11 Web Resources

[Summary](#) | [Terminology](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 18.1 Introduction

In Chapter 15, we introduced Ajax, which uses a combination of XHTML, JavaScript and XML to produce a web application with a desktop-like feel through client-side processing. In this chapter, we introduce **Adobe Flex**, another means of achieving that same goal. Flex uses Adobe's ubiquitous Flash platform to deliver a rich graphical user interface backed by **ActionScript 3**, Adobe's implementation of **ECMAScript 4** (better known as JavaScript 2). The relationship between Flex and ActionScript is similar to that between Ajax libraries and JavaScript. The powerful graphical capabilities and cross-platform nature of Flash allow web developers to deliver Rich Internet Applications (RIAs) to a large user base. The term RIA was coined in 2001 by Macromedia, the creator of Flash and Flex; Adobe acquired Macromedia in 2005.

Flex provides user interface library elements that can easily be accessed and customized. You can see these user interface elements in action using Adobe's Flex 2 Component Explorer at [examples.adobe.com/flex2/inproduct/sdk/explorer/explorer.html](http://examples.adobe.com/flex2/inproduct/sdk/explorer/explorer.html). The user interface library helps you present a consistent user experience in all applications, a quality that various Ajax and Flash applications lack. Additionally, Flash has the advantage of a large installed base—98.6% penetration for Flash 6 and up, and 84.0% penetration for Flash 9 in the United States as of March 2007.<sup>1</sup> This allows applications developed in Flex to be used on most Windows, Mac and Linux computers. Since the Flash engine is virtually equivalent across browsers and platforms, Flex developers can avoid the cross-platform conflicts of Ajax and even Java. This significantly reduces development time.

The Flex framework enables a wide variety of web applications, from simple image viewers to RSS feed readers to complex data analysis tools. This flexibility is partly derived from Flex's separation of the user interface from the data. Visually appealing and consistent user interfaces are easily described using the **MXML markup language**, which is converted to Flash's executable **SWF (Shockwave Flash)** format when the application is compiled.

1. Adobe Flash Player Version Penetration, March 2007, [www.adobe.com/products/player\\_census/flashplayer/version\\_penetration.html](http://www.adobe.com/products/player_census/flashplayer/version_penetration.html).

Flex is appropriate for online stores, where Flex's versatile user interface library allows for drag-and-drop, dynamic content, multimedia, visual feedback and more. Applications that require real-time streaming data benefit from Flex's ability to accept data "pushed" from the server and instantly update content, without constantly polling the server as some Ajax applications do. Applications that require data visualization benefit from Flex's Charting library which can create interactive and customized charts and graphs. ActionScript adds to the power of the Flex user interface library by allowing you to code powerful logic into your Flex applications.

In this chapter, you'll learn how to implement these elements in real-world applications. You'll run the examples from your local computer as well as from [deitel.com](http://deitel.com). A comprehensive list of Flex resources is available in our Flex Resource Center at [www.deitel.com/flex](http://www.deitel.com/flex). Another helpful resource is Adobe's Flex 2 Language Reference at [www.adobe.com/go/flex2\\_apiref](http://www.adobe.com/go/flex2_apiref).

## 18.2 Flex Platform Overview

The Flex platform requires the Flash Player 9 runtime environment. Flash Player 9 provides the [ActionScript Virtual Machine](#) and graphical capabilities that execute Flex applications. Flash Player 9, as described in Chapters 16–17, is a multimedia-rich application environment that runs on most platforms. Flash Player installation is detailed in those chapters, but for end users, only the [Flash Player 9](#) browser plug-in is required. The plug-in, including a debug version, is included as part of the [Flex SDK \(Software Development Kit\)](#) installation. Flex applications are essentially Flash programs that use the Flex framework of user interface elements, web services, animations and more. The Flex development environment is programming-centric in contrast to the animation-centric Flash authoring tool.

In addition to describing user interfaces, MXML can describe web services, data objects, visual effects and more. Flex's user interface elements are much richer and more consistent than those of HTML and AJAX because they're rendered the same way on all platforms by the Flash player. The root element of every MXML application is the [Application element](#) (`<mx:Application>`), inside which all Flex elements reside.

The Flex SDK is a free download, which you can get from [www.adobe.com/products/flex/downloads](http://www.adobe.com/products/flex/downloads). It includes an MXML compiler and debugger, the Flex framework, the user interface components, and some templates and examples. You can extract the zip file anywhere on your computer. The compiler and debugger included with the Flex SDK are command-line utilities. They're written in Java, so you must have Java Runtime Edition 1.4.2\_06 (or later) installed. To check your current version, run `java -version` in your command line.

ActionScript 3 is Adobe's object-oriented scripting language. Flash Player 9 uses version 2 of the ActionScript Virtual Machine (AVM2), which adds support for ActionScript 3 and provides many performance improvements over the previous version. This virtual machine is being submitted as open source to the Mozilla Firefox project to provide support for ActionScript 3 and JavaScript 2. This engine, called Tamarin, is slated to be included in Firefox 4.

ActionScript 3 supports such object-oriented capabilities as inheritance, encapsulation and polymorphism. Also, it uses an [asynchronous programming model](#). This means that the program will continue to execute while another operation is being completed, such as a call to a web service. This ensures that the user interface is responsive even while the appli-

cation is busy processing data, an important feature of RIAs. In many cases, you'll need to take advantage of event handling and data binding to handle asynchronous operations.

**Flex Builder** is Adobe's graphical IDE for Flex applications. A 30-day free trial is available at [www.adobe.com/go/tryflex](http://www.adobe.com/go/tryflex). It is based on Eclipse, a popular open source IDE. Because Flex Builder costs money, and because you can develop Flex applications without it, we won't use Flex Builder in this book.

**Adobe LiveCycle Data Services ES** extends Flex's built-in data connectivity, allowing for such features as data push and synchronization. It also enables Flex applications to handle disconnection from the server, synchronizing data upon reconnection. The Express edition of Adobe LiveCycle Data Services ES is available for free at [www.adobe.com/go/trylivecycle\\_dataservices/](http://www.adobe.com/go/trylivecycle_dataservices/). This version is limited to use on a single server with a single CPU (the license agreement is included with the download).

**Flex Charting** provides an extensible library of plotting and graphing elements, including pie charts, line graphs, bar graphs, bubble charts and plots. Flex Charting also provides appealing animations for dynamic data representations. Flex Charting is available for purchase from Adobe, and a 30-day free trial is available at [www.adobe.com/go/tryflex](http://www.adobe.com/go/tryflex). An excellent demonstration of Flex Charting is the **Flex Charting Sampler** available at [demo.quietlyscheming.com/ChartSampler/app.html](http://demo.quietlyscheming.com/ChartSampler/app.html).

## 18.3 Creating a Simple User Interface

Our first example application is a simple image viewer (Fig. 18.1) that displays thumbnail (i.e., small) images of several Deitel book covers. In this example, we specify the images with a static array within the MXML, but you could load this type of data dynamically from a web service. You can select a thumbnail to view a larger cover image, or use the horizontal slider to select an image. These two elements are bound to each other, meaning that when the user changes one, the other is updated. The image viewer also allows you to zoom the image. You can try this application at [test.deitel.com/examples/iw3htp4/flex/coverViewer/](http://test.deitel.com/examples/iw3htp4/flex/coverViewer/) (Fig. 18.2).

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!-- Fig. 18.1: coverViewer.mxml -->
3 <!-- Creating a simple book cover viewer in Flex 2 -->
4 <mx:Application xmlns:mx = "http://www.adobe.com/2006/mxml">
5 <!-- an array of images -->
6 <mx:ArrayCollection id = "bookCovers">
7 <!-- each image has a name and source attribute -->
8 <mx:Object name = "C How to Program" source = "chtp5.jpg" />
9 <mx:Object name = "C++ How to Program" source = "cpphtp6.jpg" />
10 <mx:Object name = "Internet How to Program"
11 source = "iw3htp4.jpg" />
12 <mx:Object name = "Java How to Program" source = "jhtp7.jpg" />
13 <mx:Object name = "VB How to Program" source = "vbhtp3.jpg" />
14 <mx:Object name = "Visual C# How to Program"
15 source = "vcsharphtp2.jpg" />
16 <mx:Object name = "Simply C++" source = "simplycpp.jpg" />
17 <mx:Object name = "Simply VB 2005" source = "simplyvb2005.jpg" />
```

Fig. 18.1 | Creating a simple book cover viewer in Flex 2. (Part 1 of 3.)

```
18 <mx:Object name = "Simply Java" source = "simplyjava.jpg" />
19 <mx:Object name = "Small C++ How to Program"
20 source = "smallcpphtp5.jpg" />
21 <mx:Object name = "Small Java" source = "smalljavahtp6.jpg" />
22 </mx:ArrayCollection>
23
24 <!-- bind largeImage's source to the slider and selected thumbnail -->
25 <mx:Binding
26 source = "'fullsize/' +
27 bookCovers.getItemAt(selectCoverSlider.value).source"
28 destination = "largeImage.source" />
29 <mx:Binding source = "'fullsize/' + thumbnailList.selectedItem.source"
30 destination = "largeImage.source" />
31
32 <!-- user interface begins here -->
33 <mx:Panel id = "viewPanel" title = "Deitel Book Cover Viewer"
34 width = "100%" height = "100%" horizontalAlign = "center">
35
36 <mx:HBox height = "100%" width = "100%">
37 <mx:VSlider id = "zoomSlider" value = "100" minimum = "0"
38 maximum = "100" liveDragging = "true"
39 change = "largeImage.percentWidth = zoomSlider.value;
40 largeImage.percentHeight = zoomSlider.value;"
41 height = "100%" width = "0%"
42 labels = "['0%', 'Zoom', '100%']" />
43 <mx:VBox width = "100%" height = "100%"
44 horizontalAlign = "center">
45
46 <!-- We bind the source of this image to the source of -->
47 <!-- the selected thumbnail, and center it in the VBox. -->
48 <!-- completeEffect tells Flex to fade the image in -->
49 <mx:Image id = "largeImage"
50 source = ""
51 horizontalAlign = "center"
52 verticalAlign = "middle"
53 width = "100%" height = "100%"
54 completeEffect = "Fade" />
55
56 <!-- bind this Label to the name of the selected thumbnail -->
57 <mx:Label text = "{ thumbnailList.selectedItem.name }" />
58 </mx:VBox>
59 </mx:HBox>
60
61 <!-- slider can switch between images -->
62 <mx:HSlider id = "selectCoverSlider" height = "0%"
63 minimum = "0" maximum = "{ bookCovers.length - 1 }"
64 showDataTip = "false" snapInterval = "1" tickInterval = "1"
65 liveDragging = "true"
66 change = "thumbnailList.selectedIndex =
67 selectCoverSlider.value;
68 thumbnailList.scrollToIndex(selectCoverSlider.value)" />
69
```

**Fig. 18.1** | Creating a simple book cover viewer in Flex 2. (Part 2 of 3.)

```

70 <!-- display thumbnails of the images in bookCovers horizontally -->
71 <mx:HorizontalList id = "thumbnailList"
72 dataProvider = "{ bookCovers }" width = "100%" height = "160"
73 selectedIndex = "0"
74 change = "selectCoverSlider.value = thumbnailList.selectedIndex">
75
76 <!-- define how each item is displayed -->
77 <mx:itemRenderer>
78 <mx:Component>
79 <mx:VBox width = "140" height = "160"
80 horizontalAlign = "center" verticalAlign = "middle"
81 verticalScrollPolicy = "off"
82 horizontalScrollPolicy = "off" paddingBottom = "20">
83
84 <!-- display a thumbnail of each image -->
85 <mx:Image source = "{ 'thumbs/' + data.source }"
86 verticalAlign = "middle" />
87
88 <!-- display the name of each image -->
89 <mx:Label text = "{ data.name }" />
90 </mx:VBox>
91 </mx:Component>
92 </mx:itemRenderer>
93 </mx:HorizontalList>
94 </mx:Panel>
95 </mx:Application>

```

**Fig. 18.1** | Creating a simple book cover viewer in Flex 2. (Part 3 of 3.)

Line 1 of Fig. 18.1 declares the document to be an XML document, because MXML is a type of XML. The **mx prefix**, defined in line 4, is commonly associated with the **"<http://www.adobe.com/2006/mxml>" namespace**, which is used for the Flex elements in an MXML document. The **Panel** element (lines 33–94) is a container, and is generally the outermost container of a Flex application. This element has many attributes, including **title**, **width**, **height**, **horizontalAlign** and **verticalAlign**. The **id = "viewPanel"** attribute allows us to reference this item in ActionScript using the identifier **viewPanel**. Flex elements can contain an **id attribute**, so that their properties can be accessed programmatically. The value of the **title** attribute is displayed at the top of the **Panel**, in the border. Inside the **Panel** element, there is an **HBox element** (lines 36–59), which is a container that organizes its enclosed elements horizontally. There is also a **VBox element** available for organizing elements vertically, which we will use later.

In the **HBox**, we have a **VS1ider** (lines 37–42) and a **VBox** (lines 43–58) containing an **Image element** (lines 49–54) and a **Label element** (line 57). The **VS1ider element** provides a vertically oriented slider user interface element. The **VS1ider** controls the zoom level of the image. The **value** attribute sets the slider's initial value (100 in this example). The **minimum** and **maximum** attributes set the range of values you can select with the slider. The **change** attribute (lines 39–40) allows ActionScript to execute whenever the user changes the slider's **value**. Lines 39–40 scale the image by setting its **percentWidth** and **percentHeight** properties to the slider's **value**. The **liveDragging** attribute with the value **"true"** indicates that the ActionScript in the **change** attribute executes immediately