Part IV

Client-Side Scripting

CLIENT-SIDE SCRIPTING

You have seen JavaScript used in most of the web projects presented in this book. That's because without JavaScript or some other equivalent type of programming language, it is impossible to create truly interactive web pages and applications. Of all browser-based programming languages currently available, JavaScript is by far the most popular and universally supported. JavaScript helps bind together (X)HTML, CSS, and the DOM. A good understanding of JavaScript is essential to any serious web developer.

Specifically, you will learn how to:

- Create and embed JavaScript in web pages
- Collect, store, and modify data using variables and to apply conditional and iterative programming logic
- Organize JavaScript statements into functions that process and return data
- Use arrays to store and process collections of data
- Trigger function execution when browser events occur

PROJECT PREVIEW: THE WORD DECODER CHALLENGE

In this chapter's web project, you will learn how to create a new game called the Word Decoder Challenge. This game will present the player with a series of

scrambled words and challenge him to attempt to unscramble them. When first started, the screen shown in Figure 9.1 is displayed. Game play occurs within a white rectangle area that is centered horizontally on the browser window.

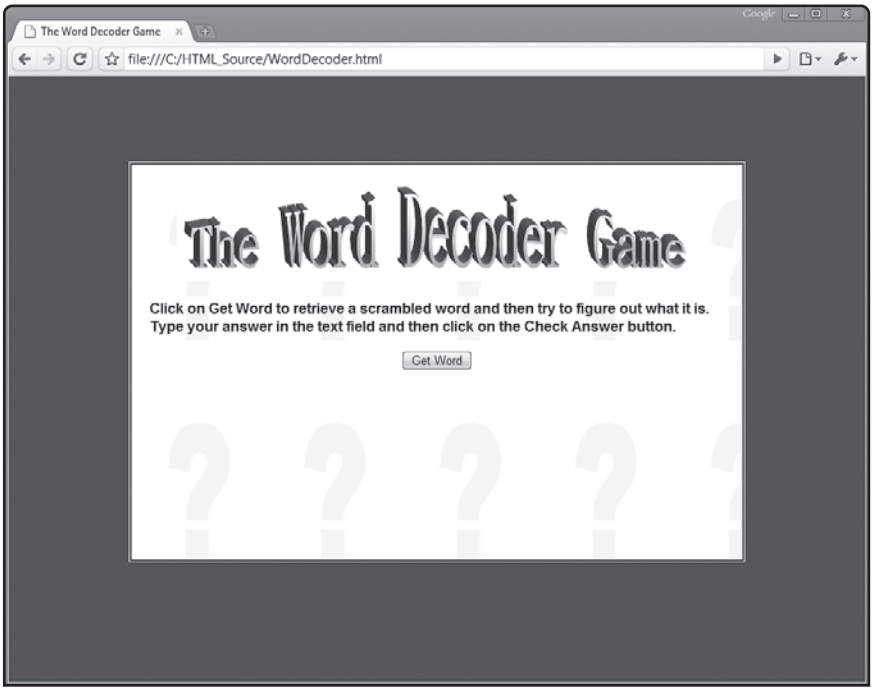


FIGURE 9.1
To begin game play the player clicks on the Get Word button.

Each time a new word is displayed, a browser window dynamically refreshes its content and displays a scrambled word in red, as demonstrated in Figure 9.2.

Once the player thinks the word has been properly decoded, he clicks on the Check Answer button to see the results of his effort. If the player's answer is correct, the player is notified via a message displayed in a popup dialog window. Similarly, if the answer supplied is not correct or if the player failed to key in anything, other messages are displayed.

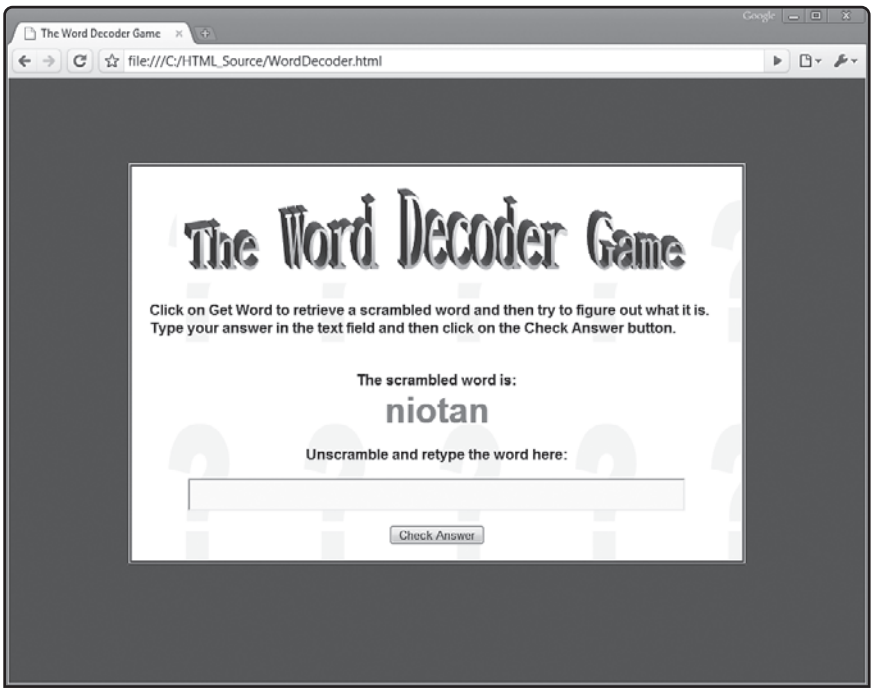


FIGURE 9.2
The player's job is to correctly retype the word in unscrambled form in the text field.

INTRODUCING JAVASCRIPT

JavaScript is a computer programming language used in the development of scripts. Scripts are small programs embedded inside HTML pages. Scripts allow you to add interactive content to web pages. JavaScript is an interpreted programming language. This means that scripts are not converted to an executable form until the HTML page they reside in is processed. The drawback to interpreted scripts is that they execute slower than programs written in compiled programming languages, which are converted into executable code at development time.

JavaScript is an object-based programming language, seeing everything within web documents and the browser as *objects*. To JavaScript, image files, text controls, and button controls are all just different types of objects. All objects have properties. *Properties* describe attributes about an object. For example, buttons display text specified using the `value` property. `img` elements also have properties. For example, there are properties that you can set to control a graphic's height and width and that you can use to specify its URL.

Objects also have *methods*, which are collections of statements that can be called upon to perform specified actions and tasks. For example, you can define methods within your JavaScripts that when called will validate form content. Object properties and methods enable JavaScripts to dynamically alter both content and its presentation on web pages.

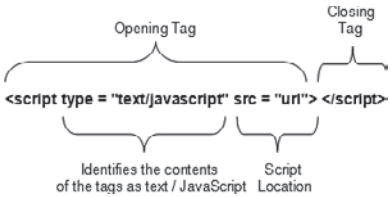
JavaScript is also capable of interacting with the user and responding to changes that occur within the browser. This is facilitated by JavaScript's ability to initiate code execution based on the occurrence of different events. An *event* is an action initiated as a result of user interaction with your web page in the browser. Events occur when the user clicks on or interacts with form elements. Events occur when web pages are opened and closed. Events also occur when the user moves the mouse pointer or enters keystrokes.

Working with JavaScript

As you have already seen, JavaScripts are inserted into HTML pages using the script element. Figure 9.3 outlines the syntax that you must follow when working with this element.

Figure 9.3

An examination of the syntax required to integrate (X)HTML and JavaScript in your documents.



The syntax outlined in Figure 9.3 includes the use of two key attributes, both of which are located in the opening `<script>` tag. The keyword `type` attribute is always set equal to `text/javascript` and the optional `scr` tells the browser where to locate an external JavaScript file (used in situations where you want to keep your markup and JavaScript code separate).

JavaScript is a case-sensitive programming language. In order to prevent errors from occurring, you must use correct capitalization when writing JavaScript code. For example, JavaScript requires that whenever you use the `document` object on any of its methods and properties that you use all lowercase.

Case-sensitivity aside, JavaScript is regarded as a very flexible programming language. It imposes a minimal set of rules regarding statement syntax. Statements begin and end on the same line. However, you can continue a statement onto another line using concatenation (explained later in this chapter). If desired, you can place multiple statements on a single line if you separate them with semicolons (;). In JavaScript, semicolons are used to identify the end of statements. Technically, the use of semicolons to mark the end of statements is optional; however, it is considered good form to use them anyway.

Like (X)HTML and CSS, JavaScript allows you to make liberal use of white space. You may insert blank lines in between statements or indent statements to make your scripts more readable.

What about Browsers That Do Not Support JavaScript?

Today, there are still many people surfing the Internet with web browsers that do not support JavaScript or that have been configured not to support it. This makes things challenging for web developers. To address this challenge, most web developers use (X)HTML comments to hide JavaScript statements from non-supporting browsers. As you know, (X)HTML comments are created using the `<!--` and `-->` characters. Anything placed in a comment is ignored by the browser.

All browsers, even those that do not support JavaScript, know not to display the `<script>` tags. However, browsers without support for JavaScript do not know how to process statements embedded within the `<script>` tags. As a result, these browsers will display the script statements as part of the web page. The result is not pretty. To keep this from occurring, enclose any JavaScript statements located inside the `script` elements within (X)HTML comments. Doing this prevents browsers that do not support JavaScript from displaying any statements on your web pages (intermixed with real content). The following example demonstrates how to do this.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Hiding JavaScript from non-supporting browsers</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      document.write("If you see this your browser supports JavaScript!");
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

Creating a Simple JavaScript

Now that you know the syntax required to work with the `script` element, let's put this knowledge to use by creating a simple JavaScript and adding it to an XHTML document. The markup and JavaScript statements that make up this example are shown here:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Chapter 9 - A simple JavaScript</title>
  </head>

  <body>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      document.write("Here I am!");
    // End hiding JavaScript statements -->
    </script>
  </body>

</html>
```

In this example, a simple JavaScript has been embedded directly into the `body` section of the web document. The first and last JavaScript statements are the script's opening and closing tags. The statement in the middle instructs the browser to write a text string of `"Here I am "` to the current document (e.g., on the web page).

Running Your JavaScripts

To test the execution of your new JavaScript, simply open the web document that contains it using your web browser. In response, the browser will render the document's markup and then execute the scripts. Figure 9.4 shows the result that you will see in your web browser.

Assuming you did not mistype the document's JavaScript when keying it in, you should see the sentence `Here I Am!` on the browser window.

FIGURE 9.4

Testing the execution of a JavaScript embedded within an XHTML document.

DIFFERENT WAYS OF INTEGRATING JAVASCRIPT INTO YOUR DOCUMENTS

You can integrate JavaScript into your web documents in a variety of ways, including embedding scripts into either the `head` or `body` sections. You can also execute external JavaScript files, separating JavaScript code from your markup. You can even embed JavaScript statements directly into (X)HTML markup.

Embedding JavaScripts in the head Section

Most Ajax developers embed their JavaScripts in the `head` sections of their (X)HTML pages. JavaScripts placed in the `head` section can be automatically or conditionally executed when your (X)HTML pages load. By placing your JavaScript functions and variable declarations in the `head` section, you ensure they are defined when the web page loads, making them ready and available when called upon for execution.

A *variable* is a pointer or reference to a location in memory where data is stored. A *function* is a named collection of code statements that can be called on to execute and perform a specific task.

The following XHTML document provides an example of how to embed a JavaScript in a document's `head` section. In this example, the script is automatically executed when the web browser loads the document.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
```

```
  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Chapter 9 - Embedding a JavaScript in the head section</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      document.write("Up, up, and away!");
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

In the previous example the `window` object's `alert` method (`window.alert`) is used to display a text string in a popup dialog window. This method is often used to display a simple message that does not require user interaction. The `alert` method has the following syntax.

```
window.alert("message");
```

message represents a text string that is to be displayed. The popup dialog window that is used to display the message includes an OK button that when clicked closes the window. More about how to work with objects and methods is provided later in this chapter.

Figure 9.5 shows the output that is generated when this example is loaded into the browser.

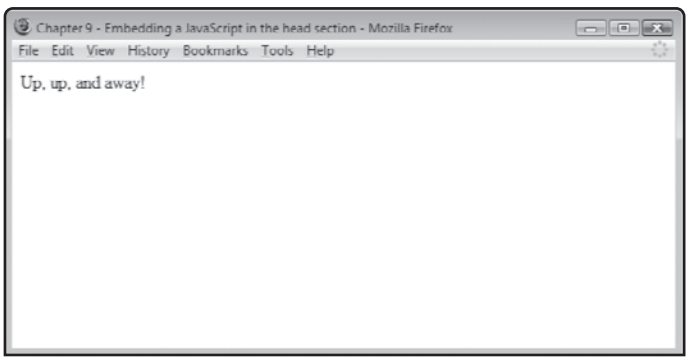


FIGURE 9.5
By default, any JavaScript embedded in an (X)HTML document's `head` section is automatically executed when loaded by the browser.

By default, any JavaScript embedded inside a document's `head` section is automatically executed when the page loads. However, any JavaScript statements organized into functions are only executed when explicitly called to run. The following document shows just such an example. Here, an XHTML page contains an embedded JavaScript that is made up of a function named `Fly()`. When the document that contains it is loaded, the function is not automatically executed.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Chapter 9 - Creating a JavaScript function</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      function Fly() {
        window.alert("Up, up, and away!");
      }
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

You will learn all about functions and how to control their execution later in this chapter.

Embedding JavaScripts in the body Section

As you have seen, JavaScripts can also be placed in the `body` section of your (X)HTML document. These scripts are automatically executed when the document loads.

```
<body>
  <script type = "text/javascript">
  <!-- Start hiding JavaScript statements
      window.alert("Up, up, and away!");
```

```
  // End hiding JavaScript statements -->
  </script>
</body>
```

Storing Your JavaScripts Externally

Large (X)HTML documents can be made of many different scripts with various levels of complexity. As with CSS, it is often a good idea to keep your markup and JavaScript separate. This is easily accomplished using external JavaScript files. To create an external JavaScript file, all you have to do is create a plain text file, add your JavaScript code to it, and save it with a .js file extension. Once created, you can refer to it using the `script` element's `scr` attribute as demonstrated here:

```
<script src = "Test.js" type = text/javascript"> </script>
```

The external JavaScript file can be used to store any number of JavaScript statements. It cannot, however, contain any (X)HTML markup. If any markup is found, an error will occur.

There are many benefits to using external JavaScripts. External JavaScripts mean smaller (X)HTML documents, which makes your (X)HTML documents easier to manage. External JavaScripts can be used by more than one document, allowing for code reuse. If you ever need to modify an external JavaScript, you can do so without having to edit every (X)HTML document that references it.

Embedding JavaScript Statements inside HTML Tags

One last option for integrating JavaScript into your (X)HTML document is to embed individual JavaScript statements within individual (X)HTML tags, as demonstrated here:

```
<body onLoad = document.write("Hello!")> <body>
```

In this example, a JavaScript statement `(onLoad = document.write("Hello!")` has been embedded within the document's opening `<body>` tag. This statement executes when the document is loaded into the browser. It instructs the browser to display a text string directly in the browser window. Embedding JavaScript statements within (X)HTML elements in this manner provides an easy way to execute small JavaScript statements.

Documenting Your Scripts

In order to make your JavaScripts self-documenting and easier to maintain, you should include embedded comments in your code that document what is going on in your web documents. Comments do not affect script performance, so use them liberally. JavaScript supports two types of comments. You can add a single line comment to a script by typing `//` followed by the comment, as demonstrated here:

```
//The following statement displays a greeting
document.write("Hello!");
```

If you want, you can append comments to the end of statements as shown here.

```
document.write("Hello!");  //The following statement displays a greeting
```

You can also create multi-line comments by enclosing text inside the `/*` and `*/` characters, as shown here:

```
/* The following statement displays a text message in the browser windows
that greet the user */
document.write("Hello!");
```

Dealing with Different Types of Values

Most JavaScripts store and process some type of data when they execute. JavaScript automatically does its best to make a determination about the type of data it is presented with. This determination has a direct impact on how the data is handled. Table 9.1 outlines different types of values supported by JavaScript.

Table 9.1 Values Supported by JavaScript

Value	Description
Boolean	A value indicating a condition of either `true` or `false`
Null	An empty value
Numbers	A numeric value
Strings	A string of text enclosed in matching quotation marks

Once specified or collected, you can store data in your scripts using variables. A *variable* is a pointer or reference to a location in memory where a piece of data is stored.

Storing and Retrieving Data

JavaScript allows you to store and retrieve individual pieces of data as well as collections of data. Individual pieces of data are managed using variables. Collections of data are managed using arrays. You will learn how to work with both variables and arrays in the sections that follow.

Defining JavaScript Variables

In order to use a variable in a JavaScript, it must be declared (or defined). You can declare a variable explicitly or implicitly. An explicitly declared variable is defined before it is referenced using the `var` keyword. The following example demonstrates how to explicitly declare a variable.

```
var playerScore = 1000;
```

Note that in the preceding example a numeric value of `1000` was assigned to the variable. If you assign a text string to a variable, you must enclose the string inside quotation marks, as demonstrated here:

```
var playerCharacter = "Wizard";
```

Here, a variable named `playerScore` has been declared. It has also been assigned an initial value of `1000`. To implicitly declare a variable, you simply reference it for the first time without first declaring it, as demonstrated here:

```
playerScore = 1000;
```

Explicit variable declaration is considered to be good form and is highly recommended.

It is not necessary to assign a variable a value at declaration time. However, doing so can make your script code easier to understand and also helps JavaScript determine the type of data the variable will be used to store.

Naming Your Variable

When assigning names to your variables, you should take care to assign names that are descriptive of the variable's purpose. For example, `totalScore` is a much more descriptive variable name than `x`. In addition, you must follow the rules outlined below when assigning variable names.

- Variable names must start with a letter or the underscore character.
- Variable names cannot contain blank spaces.
- Variable names can only be made of letters, numbers, and the underscore character.
- You cannot use reserved words as variable names.

JavaScript variables are case-sensitive. If you assign a variable a name of `totalScore`, you must use the exact same case when referring to it in other parts of your JavaScript. If you accidently make a typo of `totalscore` or use a different case like `TOTALSCORE` or `totalscore`, JavaScript will regard each of these as being different variables.

Understanding Variable Scope

Variable scope refers to the location within a script where a variable is accessible. JavaScript supports both global and location scope. Global variables can be accessed by any script embedded within an (X)HTML document. Local variables exist within functions and can only be accessed by statements located in the functions where the variables are defined.

Global variables are accessible by any script located within a web document. There are two ways of defining global variables. These include:

- Making an initial reference to a new variable from inside a function (without using the `var` keyword).
- Declaring a variable outside of a function (with or without the `var` keyword).

A *local variable* is declared inside a function by preceding its initial reference with the `var` keyword. A *function* is a named collection of code statements. Functions can be called from different locations within a document. The following example demonstrates how to create a local variable.

```
function DisplayMsg() {
  var message = "Well, hello there!";
  document.write(message);
}
```

In this example, a function named `DisplayMsg()` has been defined. When executed statements in the function declare a local variable named `message`, assign it a text string, and then display the string (in the browser window). Because it is a local variable, the value assigned to `message` is not accessible from outside the function.

Working with Collections of Data

Depending on how much data you need to collect and process on your web pages, there may be times when working with individual variables becomes impractical. In these situations, you can use arrays to store large collections of data. An *array* is an indexed list of values. Arrays can store any type of value. For example, the following statements declare an array and then use it to store three text strings.

```
powers = new Array(2);
powers[0] = "Flight";
powers[1] = "Super Strength";
powers[2] = "Heat Vision";
```

The first statement declares an array named `powers` using the required `new` keyword. The array has been set up to store five values. The remaining statements populate the array with data. In JavaScript, array's indices begin with an index position of 0. So, in the previous example, the `powers` array can store three values, in index positions 0 through 2.

Accessing Array Elements

To access a value stored in an array, you must specify the array's name followed by the value's index position enclosed in square brackets. As an example, look at the following document.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Working with arrays</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      powers = new Array(2);
      powers[0] = "Flight";
      powers[1] = "Super Strength";
      powers[2] = "Heat Vision";
      document.write("The first super power is " + powers[0]);
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

Remember, JavaScript arrays begin at index position 0. The first item in the array is located at index position 0 (e.g., `Flight`).

Processing Arrays with Loops

Processing the array elements an element at a time is impractical for large arrays containing dozens, hundreds, or thousands of elements. Instead, web developers use loops to process array contents as demonstrated here:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Working with functions</title>

    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      powers = new Array(2);
      powers[0] = "Flight";
      powers[1] = "Super Strength";
      powers[2] = "Heat Vision";
      document.write("List of Super Powers <br />");
      for (var i = 0; i < powers.length; i++) {
        document.write("* " + powers[i] + "<br />");
      }
    // End hiding JavaScript statements -->
    </script>

  </head>

  <body>
  </body>

</html>
```

In this example, an array named `powers` has been defined and populated with three values. A `for` loop is used to process the array, starting at `powers[0]`. Each time the loop iterates, the value of `i` is incremented by 1, causing the loop to process the next element stored in the array. Take note of the use of the array object's `length` property. This property stores a numeric value representing the array's length. It is used to control loop execution, halting the loop when the end of the array is reached. Figure 9.6 shows the output produced when this example is executed.

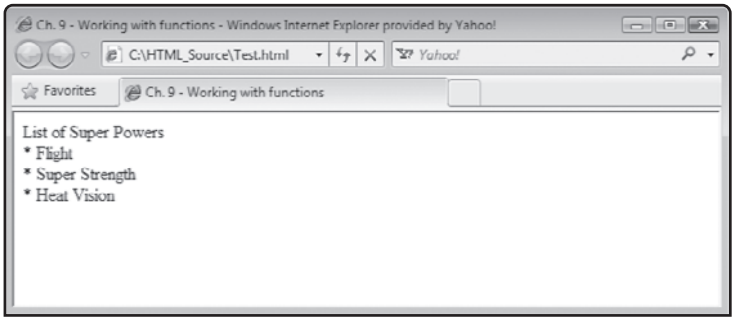


FIGURE 9.6
Processing the contents of an array using a loop.

Manipulating and Comparing Data

There is a lot more to working with data than collecting, storing, retrieving, and displaying it. Most scripts analyze data in some manner. To do this, you need to learn how to work with a number of operators that facilitate mathematic operations, data assignment, and value comparison.

Performing Mathematic Calculations

JavaScript supports a range of arithmetic operations that facilitate the development of arithmetic calculations when working with numeric data. Table 9.2 outlines and demonstrates the use of these operations. With this collection of operators at your disposal, you can develop programming logic that performs virtually any type of calculation you might want to process.

While use of the first four operators shown in Table 9.2 is self-explanatory, the remaining operators require explanation. The `x++` and `++x` operators are used to increment a value of a numeric variable by 1. The difference between these two operators lies in when the update occurs. Suppose you had two variables, `totalScore` and `points`. If `points` was equal to `100` when the following statement executed, the value of `points` would first be incremented by 1 and then its value (101) would be assigned to `totalScore`.

```
totalScore = ++points;
```

As demonstrated next, use the `x++` operator in place of the `++x` operators to generate a different result.

```
totalScore = points++;
```

TABLE 9.2 JAVASCRIPT OPERATORS

Operator	Description	Example
+	Adds two values together	`totalScore = 5 + 10`
-	Subtracts one value from another	`totalScore = 10 - 5`
*	Multiplies two values together	`totalScore = 5 * 10`
/	Divides one value by another	`totalScore = 10 / 5`
-x	Reverses a variable's sign	`count = -count`
x++	Post-increment (returns x, then increments x by one)	`x = y++`
++x	Pre-increment (increments x by one, then returns x)	`x = ++y`
x--	Post-decrement (returns x, then decrements x by one)	`x = y--`
--x	Pre-decrement (decrements x by one, then returns x)	`x = --y`

Here, the value of `points` (e.g., 100) is first assigned to `totalScore` (setting it equal to 100). Then the value of `points` is incremented by 1 to 101. The `--x` and `x--` operators work just like the `++x` and `x++` operators—the difference being that they decrement a variable's value instead of incrementing it.

Assigning Values to Variables

To assign a value to a variable, you use the = (equals) operator. To change a variable's value, all you have to do is assign it a different value, as demonstrated here.

```
totalScore = 0;
.
.
.
total = 100;
```

In addition to the = operator, JavaScript supports all of the operators listed in Table 9.3.

TABLE 9.3 JAVASCRIPT ASSIGNMENT OPERATORS

Operator	Description	Examples
=	Sets a variable value equal to some value	x I y + I
+=	Shorthand for x I x + y (addition)	x +I y
-=	Shorthand for x I x – y (subtraction)	x -I y
*=	Shorthand for x I x * y (multiplication)	x *I y
/=	Shorthand for x I x / y (division)	x /I y
%=	Shorthand for x I x % y (remainder)	x %I y

Let's look at an example of how to work with each of the operators shown in Table 9.3.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Working with JavaScript assignment operators</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var a = 3;
      var b = 10;
      var c = 0;
      c = a + b;
      document.write("a = " + a + ", b = " + b + ", c = " + c + "<br />");
      document.write("<br />a + b = " + c + "<br />");
      c += 2
      document.write("<br />c += 2 = " + c + "<br />");
      c -= 5
      document.write("<br />c -= 5 = " + c + "<br />");
      c *= 3
      document.write("<br />c *= 3 = " + c + "<br />");
      c /= 2
      document.write("<br />c /= 2 = " + c + "<br />");
      c %= 7
```

```
      document.write("<br />c %= 7 = " + c + "<br />");
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

Notice that the `<br />` tag has been embedded within various statements as a means of controlling line breaks. Also note the use of the + operator. When used with strings, the + operator joins or concatenates two strings together to create a new larger string.

Figure 9.7 shows the output generated when this example is loaded into the browser.

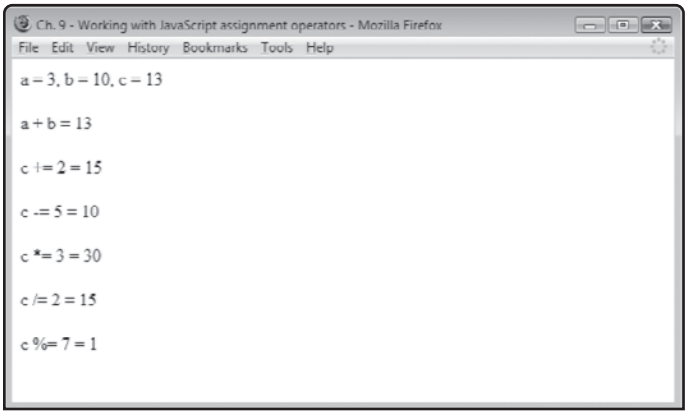


FIGURE 9.7
A demonstration of how to work with JavaScript's operators.

Comparing Values

To be useful, data generally needs to be analyzed. With numeric data this typically means performing different types of comparison operations. Based on the results of this analysis, a JavaScript can alter its execution. A JavaScript might, for example, take one action if the value of a variable is greater than `1000` and another action if it is not. Table 9.4 provides a listing of the different types of comparison operators supported by JavaScript.

TABLE 9.4 JAVASCRIPT COMPARISON OPERATORS

Operator	Description	Example
==	Equal to	x == y
!==	Not equal to	x !== y
>	Greater than	x > y
>=	Greater than or equal to	x >= y
<	Less than	x < y
<=	Less than or equal to	x <= y

To determine if two values are equal, you must use the == comparison operator (not the = assignment operator). If you get these two operators mixed up an error will occur. As an example of how to work with these operators, look at the following example.

```
if (x == y) {
document.write("x equals y");
}
```

Here, a comparison is made between two values stored in the `x` and `y` variables. If the values are equal a text string is displayed. If you modify this example, as demonstrated here, you can determine if the value of `x` is greater than or equal to the value of `y`.

```
if (x >= y) {
document.write("x is greater than or equal to y");
}
```

MAKING DECISIONS

Conditional programming logic gives you the ability to make decisions and to alter the logical execution flow of code statements in your JavaScripts based on the result of comparison operations. Conditional logic allows you to execute a set of statements based on whether a tested condition evaluates as true.

Working with the if Statement

The `if` statement gives you the ability to compare two values and to conditionally execute one or more statements based on the result of that analysis. In its most basic form, the `if` statement uses the following syntax.

```
if (condition) statement
```

Here, ***condition*** is an expression that when evaluated generates a value of `true` or `false`. Note that the expression that is analyzed must be enclosed in parentheses. Take a look at the following example which shows you how to apply this version of the `if` statement.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - An example of how to work with the if statement</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var totalScore = 1000;
      if (totalScore > 1001) document.write("You win!");
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

In this example, a variable named `totalScore` is declared and assigned a value of `1000`. Next, an `if` statement is used to determine if the value of `totalScore` is greater than 1001 (which it is not). Had it been greater than 1001, a message would have been displayed. However, since `totalScore` is equal to `1000`, nothing happens.

Generating Multiline if Statements

If you include `{` and `}` characters, as demonstrated in the following example, you can use the `if` statement to create a code block, which can include any number of statements. Every statement in the code block will execute if the tested condition evaluates as `true`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
```

```
  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - An example of how to work with the if statement</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var totalScore = 20000;
      if (totalScore > 10000) {
        document.write("Game over. You win!");
      }
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

In this example, two statements are executed if the value assigned to `totalScore` is greater than or equal to `10000`.

Handling Alternative Conditions

The `if` statement supports an optional `else` keyword that when used lets you modify an `if` statement code block so that it can execute an alternative set of statements if the tested condition proves `false`. An example of how this works is provided here:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - An example of how to work with the if statement</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var totalScore = 9999;
      if (totalScore <= 10000) {
        document.write("Game over. You lose.");
```

```
      }
      else {
        document.write("Game over. You win.");
      }
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

In this example, the message `Game over. You lose.` is displayed if `totalScore` is less than or equal to `10000` and a string of `Game over. You win!` is displayed if `totalScore` is not less than or equal to `10000`. Note that in this example two separate code blocks have been defined, each of which has its own set of opening `{` and closing `}` characters.

Nesting if Statements

If you need to perform complex conditional logic that involves comparing multiple values, where one decision is based on the outcome of another decision, you can nest `if` statements to outline the required logic. The following example demonstrates how to nest `if` statements.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - An example of how to work with the if statement</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var gameOver = false;
      var totalScore = 9999;
      if (gameOver == true) {
       if (totalScore <= 10000) {
          document.write("Game over. You lose.");
        }
```

```
        else {
          document.write("Game over. You win.");
        }
      }
      else {
        document.write("Please try again.");
      }
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

Here, the value assigned to `gameOver` is analyzed to determine if it is equal to `true`. If it does not equal `true`, the statement embedded in the `else` code block is executed.

Evaluating Conditions with the switch Statement

If you need to evaluate a series of possible values to determine a match, you can do so using a series of `if` statements. However, an easier and more efficient option is to use the `switch` statement. The `switch` statement specifies an expression which is then compared to a series of `case` statements to see if a match can be found. If a match is found, statements belonging to the matching `case` statement are executed. The `switch` statement's syntax is outlined here:

```
switch (expression) {
  case label:
    statements;
  break;
    .
    .
    .
  case label:
    statements;
  break;
  default:
   statements;
}
```

The value of ***expression*** is compared to the expression outlined in each `case` statement's ***label***. The statements of the first `case` statement that result in a match are executed. If no match is found, the statements that belong to the optional `default` statement, if specified, are executed.

The `break` statement at the end of each set of statements belonging to the `case` statement is optional. When specified, the `break` statement instructs JavaScript to exit the `switch` statement. If you do not specify a `break` statement at the end of each `case` statement code block, the script will execute the statements of any `case` statement that matches (instead of just the statements belonging to the first `case` statement that matches).

To better understand how to work with the `switch` statement, let's look at an example.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - An example of how to work with the switch statement</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var color = window.prompt("Pick a color: red, green, or blue.");
      switch (color) {
        case "red":
          document.write("Rose petals are red.");
          break;
        case "green":
          document.write("Stems are green.");
          break;
        case "blue":
          document.write("Violets are blue.");
          break;
        default:
          document.write("Error: Invalid input.");
      }
    // End hiding JavaScript statements -->
    </script>
  </head>
```

```
  <body>
  </body>

</html>
```

In this example, the user is prompted to enter a color. Once the user's input is collected, it is assigned to a variable named `color`. The value of `color` is then analyzed. The message displayed depends on the result of that analysis.

Note the manner in which the `color` variable's value is assigned in the previous example. The `window` object's `prompt` method is used to display the message in a popup dialog window that prompts the user to input. The `prompt` method's syntax is outlined here:

```
window.prompt("message" [, "default"]);
```

message represents the string displayed in the popup dialog window. *default* is an optional parameter. If specified, it displays default input in the popup dialog window's text field. The `window` object's `prompt` method is great for situations where you need to collect a small piece of data from the user.

Using Loops to Work Efficiently

A *loop* is a collection of statements that are executed repeatedly. Programmers use loops to process large amounts of data and to execute repetitive tasks. The power of loops lie in the fact that with just a few lines of code you can process an unlimited amount of data or perform a given task an infinite number of times.

Creating a Loop Using the for Statement

The `for` statement allows you to set up a loop that executes for as long as a specified condition remains `true`. A variable is used to manage loop execution. The `for` loop consists of three parts: a starting expression, a tested condition, and an increment statement. The loop's syntax is outlined here:

```
for (expression; condition; increment) {
  statements;
}
```

You may place as many JavaScript statements as you want in between the loop's opening and closing bracket. These statements are executed every time the loop iterates (repeats). As an example of how to work with the `for` statement, look at the following example.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Executing a for loop</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      for (i = 1; i <= 10; i++) {
        document.write(i,"<br />");
      }
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

In this example, a loop has been set up to repeat 10 times. In its first iteration, the value of `i` is set to 1. The loop repeats 10 times, terminating when the value exceeds 10. Figure 9.8 shows the output generated when this example executes.

FIGURE 9.8
Using a `for` loop to count from 1 to 10.

Creating a Loop Using the while Statement

JavaScript also supports the `while` statement, which you can use to set up a loop that executes for as long as a specified condition is `true`. The `while` loop's syntax is outlined here:

```
while (condition) {
  statements;
}
```

The following example demonstrates how to work with the `while` statements.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Executing a while loop</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var counter = 10;
      document.write("<p>Begin countdown.</p>");
      while (counter > 0) {
        document.write(counter + "<br />");
        counter = counter - 1;
      }
      document.write("<br />Blastoff!");
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

In this example, a `while` loop is configured to iterate for as long as the value of `counter` is greater than 0. Every time the loop repeats, the value of `counter` is displayed and decremented by 1. Figure 9.9 shows the output that is displayed when this example is executed.

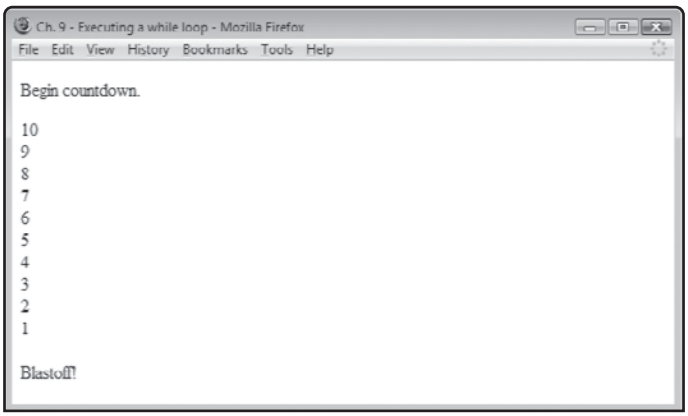


FIGURE 9.9
An example of how to use a `while` loop to iterate 10 times.

Creating a Loop Using the do. . .while Statement

Another type of loop supported by JavaScript is the `do. . .while` loop. This loop repeats until a tested condition becomes `false`. The `do...while` loop's syntax is outlined here:

```
do {
  statements;
} while (condition)
```

The `do. . .while` loop distinguishes itself from the `while` loop in that it always executes at least once. This is because the loop's *condition* is not evaluated until the end of the loop. The following example demonstrates how use the `do…while` loop.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Executing a do...while loop</title>

    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      var counter = 11;
      document.write("<p>Begin countdown.</p>");
      do {
        counter = counter - 1;
```

```
        document.write(counter, "<br />");
      } while (counter > 1)
      document.write("<br />Blastoff!");
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
  </body>

</html>
```

The output generated by this example is the same as that produced by the `while` loop example.

Breaking out of Loops

Loops automatically execute over and over again from beginning to end. While this is usually what you want, there may be times you will want to halt loop execution or skip an iteration of the loop. The following example shows you can use the `break` statement to halt a loop's execution.

```
for (i = 1; i <= 5; i++) {
  document.write(i,"<br />");
  if (i == 3) break;
}
```

When executed, this example displays the following output on the browser window.

```
1
2
3
```

In this example, the `break` statement has been used to halt the loop when the value of i becomes 5. In similar fashion, you can use the `continue` statement to prematurely terminate the current execution of a `while` loop. This allows the loop to continue running. The following example demonstrates how this works.

```
for (i = 1; i <= 5; i++) {
  if (i == 3) continue;
  document.write(i,"<br />");
}
```

When executed, this example displays the following output on the browser window.

```
1
2
4
5
```

Note that the number 3 is missing from this output as a result of the premature termination of the fifth iteration of the loop.

Organizing Your JavaScripts into Functions

A *function* is a collection of code statements that can be called by name to execute and perform a specific task. Web developers that use JavaScript usually store functions in the `head` section of (X)HTML documents ensuring they are available as soon as the web document is loaded by the browser. Placing all your functions in one place also makes them easier to find and maintain.

Defining Functions

A function must be defined before it can be executed. If an attempt is made to create a function that has not been defined, an error will occur. The syntax that you must follow when defining functions is outlined here:

```
function FunctionName(p1, p2,....pn) {
  statements;
return
}
```

`FunctionName` represents the function's name. The function's name must be immediately followed by parentheses. The parentheses are used to define an optional list of comma-separated arguments for the function to process. You must provide the parentheses even if a function does not define any arguments. A function can hold as many statements as you want. These statements must be embedded within the function's opening and closing curly braces. Functions can include an optional `return` statement that, when present, halts the function's execution. The result statement can also return an optional value back to the statement that executed the function.

In the following example a function named `Verify()` is defined. It accepts one argument as input. The function automatically assigns the incoming argument to a local variable named `age`. It then analyzes the value assigned to `age` to determine if the user is old enough to play and then displays either of two messages:

```
function Verify(age) {
  if (age < 21) {
    window.alert("Sorry, you are not old enough to play this game.");
    return "true"
   } else {
    window.alert("Let's get ready to rumble...");
    return "false"
  }
}
```

JavaScript functions are not automatically executed. They must be explicitly called upon in order to execute. Functions facilitate code reuse by allowing you to call upon a function from any location within a document. This eliminates the need to duplicate script statements and results in smaller scripts.

Executing Functions

You can execute a JavaScript function in two different ways. First, you can call on a function by typing its name, as shown here:

```
DisplayMsg();
```

Here, a function named `DisplayMsg()` is executed. No arguments are passed to the function but the parentheses are required anyway. When called, the function executes. When finished, processing flow is returned to the next statement in the script.

Functions designed to process arguments can be passed data when called, as demonstrated here:

```
UpdateScore(1000);
```

In this example, a function named `UpdateScore()` is executed and passed a value of `1000`. You can pass any number of arguments to a function as demonstrated below (provided the function has been set up to handle them).

```
UpdateAllScores(1000, 850, 75);
```

A second way of executing a function is to call on the function as part of an expression. This option requires that the function be set up to return a value. As an example, the following statement executes the `UpdateScore()` function and stores the result that is returned in a variable named `totalScore`.

```
totalScore = UpdateScore();
```

To make sure that you have a good understanding of how to execute functions and return a result from them, look at the following example.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Working with functions</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      function Verify() {
        var result;
        result = window.prompt("How old are you?","");
        return result;
      }
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      playerAge = Verify();
      if (playerAge > 17) {
        window.alert("Let's get ready to rumble...");
      } else {
        window.alert("Sorry. Come back and play when you are 18.");
      }
    // End hiding JavaScript statements -->
    </script>
  </body>

</html>
```

In this example, a function named `Verify()` is defined. When executed, the function displays a popup dialog window that asks the user to enter his age. The user's input is then assigned to a variable named `result`, which is then returned to the calling statement. The function is executed by a statement located in the `body` section. The script in the `body` section takes the value that is returned by the function and assigns it to a local variable named `playerAge`, which it then analyzes.

Creating Interactive Web Pages Using Event-Driven Scripts

Events are things that occur within the browser. Events execute when the mouse clicks on something. Events occur when the mouse moves, when keys are pressed, and the browser window is moved, opened, or closed. Web browsers know how to recognize events and respond to them. For example, if you click on a link, the `onClick` event occurs and the browser's default response is to load the link's URL.

Using JavaScript you can create functions and associate them with specific events. To do this you need to know how to work with event handlers. An *event handler* detects events and reacts to them. Event handlers can be used to execute individual JavaScript statements or to call on JavaScript functions. Events are associated with objects. The following example demonstrates how to set up an event handler.

```
<body onload = "window.alert('Ta Da!')">
```

This event handler executes when the `load` event occurs, triggering the display of an alert message. As this example demonstrates, you can execute any JavaScript statement using an event handler. However, the real benefit of event handlers is their ability to execute functions.

Different Types of Javascript Events

JavaScript can react to a host of events. Table 9.5 provides a list of commonly used events along with their associated event handlers.

Each event has an accompanying event handler. Examples of how to work with these event handlers are provided in the sections that follow.

TABLE 9.5 JAVASCRIPT EVENTS AND EVENT HANDLERS

Event	Handler	This event occurs when:
abort	onabort	An action is aborted
blur	onblur	An item loses focus
change	onchange	A control's data is changed
click	onclick	When an element is clicked
dblclick	ondblclick	When an element is double clicked
dragdrop	ondragdrop	An element is dragged and dropped
error	onerror	A JavaScript error occurs
focus	onfocus	An element receives focus
keydown	onkeydown	A keyboard key is held down
keypress	onkeypress	A keyboard key is pressed
keyup	onkeyup	A keyboard key is released
load	onload	A web page is loaded
mousedown	onmousedown	One of the mouse buttons is pressed
mousemove	onmousemove	The mouse is moved
mouseout	onmouseout	The mouse is moved off an element
mouseover	onmouseover	The mouse is moved over an element
mouseup	onmouseup	The mouse's button is released
reset	onreset	A form's Reset button is clicked
resize	onresize	An element is resized
submit	onsubmit	A form's Submit button is clicked
unload	onunload	The browser unloads a web page

Managing Window Events

JavaScript can react to a number of different events that involve the browser window. These events include the `load`, `unload`, and `resize` events. To create web pages that can respond to these events all you need do is place the appropriate event handlers in your (X)HTML document's opening `<bodytag`, as demonstrated here:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Working with window events</title>
```

```
  </head>

  <body onload = "window.alert('Page loaded!')"
    onresize = "window.alert('Ouch, that hurts!')"
    onunload = "window.alert('Goodbye cruel world...')">
  </body>

</html>
```

Figures 9.10 through 9.12 show the output displayed when this document is loaded by the browser, when its web page is resized, and when it is unloaded.

FIGURE 9.10
The `onLoad` event occurs when a document is first loaded into the browser.

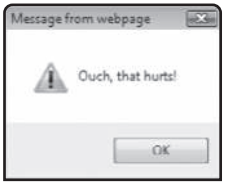


FIGURE 9.11
The `onResize` event occurs when the user changes the size of the browser window.

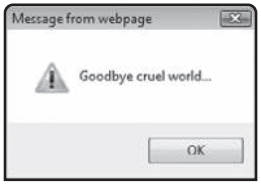


FIGURE 9.12
The `onUnload` event occurs when the user loads a different page or closes the web page or the current page.

Events are also triggered when the user moves or clicks the mouse within the confines of the browser window. Examples of mouse-related events include the `onMouseOver` and `onMouseOut` events. As an example of how to develop a script that reacts to these two events, look at the following document.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
    <title>Ch. 9 - Working with mouse events</title>
    <script type = "text/javascript">
    <!-- Start hiding JavaScript statements
      function ShowMsg(input) {
        document.getElementById('placeholder').innerHTML = input;
      }
    // End hiding JavaScript statements -->
    </script>
  </head>

  <body>
    <p>
      <a href="http://www.courseptr.com"
        onmouseover = 'ShowMsg("Visit www.courseptr.com")'
        onmouseout = 'ShowMsg(" ")'>
        Go to www.courseptr.com</a>
   </p>
   <div id = "placeholder"> </div>
  </body>

</html>
```

In this example, a link is displayed on the browser window. `onMouseOver` and `onMouseOut` event handlers are displayed and clear the display of text on the browser window whenever the mouse pointer is moved over and away from the link. Each time the mouse pointer moves over the link, the `DisplayMessage()` function is executed. The function establishes a reference to the document's `div` element and modifies its content. It does this using the `getElementByID()` method in conjunction with the object's `innerHTML` property. This allows it to display text string that was passed to it as an argument.

Back to the Word Decoder Challenge Project

It is time to turn your attention back to this chapter's project, the Word Decoder Challenge game. This game presents the player with a scrambled word and challenges the player to decode it. To complete this game you will have to develop a JavaScript that makes use of variables, arrays, conditional logic, and events. The JavaScript will have to make use of methods and properties that interact with the DOM. To help give the game a little extra pizzazz, CSS will be used to enhance presentation.

Designing the Application

As with every game project covered in this book, this game will be developed in a series of steps, as outlined here:

1. Create a new XHTML document.
2. Develop the document's markup.
3. Add `meta` and `title` elements.
4. Specify document content.
5. Create the document's script.
6. Create an external style sheet.
7. Load and Test the Word Decoder Challenge Project.

Step 1: Creating a New XHTML Document

The first step in the development of the Word Decoder Challenge game is to create a new web document. Do so using your preferred code or text editor. Save the document as a plain text file named WordDecoder.html. This web document will make use of CSS style rules. Therefore, you will need to create a second file named wd.css.

Step 2: Developing the Document's Markup

The next step in the development of this project is to assemble the document's markup. To do so, add the following elements to the WordDecoder.html file.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

  <head>

  </head>
```

```
  <body>

  </body>

</html>
```

Step 3: Adding meta and title Elements

Now it is time to complete the document `head` section by adding the elements responsible for defining its `meta` and `title` elements. To do so, add the following elements to the document's `head` section.

```
<meta http-equiv="Content-type" content="text/xhtml; charset=UTF-8" />
<title>The Word Decoder Game</title>
```

This web page will use an external style sheet named wd.css to enhance its presentation. Therefore, you will need to add the following tag to the end of the `head` section to make the connection to the external style sheet.

```
<link href = "wd.css" type = "text/css" rel = "stylesheet" />
```

Step 4: Specifying Document Content

The markup for this document, shown below, is very straightforward. It uses `div` and `p` elements to group content and insert blank spaces. It displays a graphic image at the top of the web page using an `img` element. It also makes use of a form that contains two button controls and a text field. Note that this game also makes use of inline styles that set the initial visibility state of these controls (to either `visible` or `hidden`). Later in the document's JavaScript you will add programming logic that takes control over the visible state of form control during game play, making the control dynamically appear and disappear during game play.

```
<div id = "mainDiv">

 <img src="logo.png" alt = "A graphic logo that displays the game's name" />

 <p id = "instructions">
   Click on Get Word to retrieve a scrambled word and then try to
   figure out what it is. Type your answer in the text field and then click
   on the Check Answer button.
 </p>

 <form action = "WordDecorder.html">
```

```
    <div>

      <input type = "button" id = "GetWordBtn" value = "Get Word"
        onclick = "StartGame()"/>

      <div id = "ScrambledHeading" style="visibility: hidden">The scrambled
        word is:
      </div>

      <div id = "ScrambledDiv" > </div>

      <div id = "UnscrambledHeading" style = "visibility:hidden">
        <p>Unscramble and retype the word here:</p>
      </div>

      <input type = "text" size = "45" style = "visibility: hidden"
        id = "inputField"/>

      <p>
        <input type = "button" id = "checkAnswerBtn" value = "Check Answer"
          style="visibility:hidden" onclick = "CheckAnswer()"/>
      </p>

    </div>

  </form>

</div>
```

Note the including of the `ID` attribute in various elements throughout the markup. This provides hooks into each element, allowing the document's JavaScript to interact with and control these elements.

Step 5: Creating the Document's Script

The document's markup is now complete. The next step in the development of this game is to lay out its JavaScript code. Begin by adding the following statements to the document's `head` section.

```
<script type = "text/javascript">
<!-- Start hiding JavaScript statements

// End hiding JavaScript statements -->
</script>
```

Now that the script's opening and closing tags are in place, add the following statements to the script.

```
//Define global variables and an array
var Request = false;
var wordArray = new Array(10);
var scrambledWord = "";
var unscrambledWord = "";

//Populate the array in a string made up of copies of the same word (one
//unscrambled and the other scrambled
wordArray[0] = "dog gdo";
wordArray[1] = "cat tac";
wordArray[2] = "lion niol";
wordArray[3] = "elephant tnaphele";
wordArray[4] = "car rac";
wordArray[5] = "desk ksed";
wordArray[6] = "pen epn";
wordArray[7] = "envelope evneelop";
wordArray[8] = "nation niotan";
wordArray[9] = "imperial liaimper";
```

The first three statements above define three global variables and an array named `wordArray`. The next ten statements populate the array with ten strings containing two versions of the same word, separated by a blank space. The first version of the word is the word in its unscrambled format. The second version of the word is the word in a scrambled format.

The rest of the script is made up of three functions. The `StartGame()` function, shown here, is called for execution when the player clicks on the `Get Word` button (e.g., `GetWordBtn`). It passes the `getElementById()` method `ID` of five form elements in order to establish a reference to those elements. The statements then use the element's `style` property's `visibility` property to control whether each element is visible. The next two statements clear out any text displayed in the form's text control and in the `div` element named `ScrambledDiv`. The last thing the function does is call on the `GetWord()` function to execute.

```
//This function is executed when the game's Get Word button is clicked.
function StartGame() {

  //These statements control form element visibility
  document.getElementById("checkAnswerBtn").style.visibility = "visible";
  document.getElementById("GetWordBtn").style.visibility = "hidden";
  document.getElementById("inputField").style.visibility = "visible";
  document.getElementById("ScrambledHeading").style.visibility = "visible";
  document.getElementById("UnscrambledHeading").style.visibility =
    "visible";

  //These statements clear out any text displayed by these elements
  document.getElementById("inputField").value="";
  document.getElementById('ScrambledDiv').innerHTML = "";

  //This statement executes the getWord() function
  getWord();

}
```

The code statements that make up the `getWord()` function are shown below and should be added to the end of the document's JavaScript. The function begins by declaring a number of variables used within the function. It then generates a random number between 0 and 9 and uses that number to select an element from the `wordArray` array, which is then assigned to a variable named `selectedWord`. Next, the `indexOf()` method is used to locate the character position of the blank space within the string stored in `selectedWord`. The `substr()` method is used to assign the unscrambled portion of the string, starting at character position 0 out to the character position of the blank space, to the `unscrambled` variable. Similarly, the `substr()` function is used a second time to assign the scrambled portion of the string to the `scrambledWord` variable, starting at one character position past the blank space out to the end of the string. Lastly, the scrambled word is displayed on the browser window for the player to see.

```
//This function retrieves a random word string for the player to guess
function getWord() {

  //Declare variables used within the function
  var randomNo = 0;
  var selectedWord = "";
```

```
  var loc = 0;

  //Generate a random number from 0 to 9 and use it to select an element
  //from the array
  randomNo = Math.random() * 9;
  randomNo = Math.round(randomNo);
  selectedWord = wordArray[randomNo]

  loc = selectedWord.indexOf(" ");                //Locate the blank space
  unscrambledWord = selectedWord.substr(0, loc); //Assign the unscrambled
                                                  //word
  scrambledWord = selectedWord.substr(loc + 1); //Assign the scrambled word

  //Display the scrambled version of the word in the browser window
  document.getElementById('ScrambledDiv').innerHTML = scrambledWord;

}
```

The `indexOf()` method is a built-in JavaScript method that is used to locate the starting character position of one string within another string. The `substr()` method is used to extract a portion of a string. To learn more about these and other built-in methods that come with JavaScript, visit http://www.w3schools.com/jsref/jsref_obj_string.asp.

The last function in the document's JavaScript is the `CheckAnswer()` function, which is executed when the player clicks on the button control labeled Check Answer (e.g., `checkAnswerBtn`). It retrieves the player's input and determines whether the player successfully decoded the scrambled word. It ends by resetting the form's elements back to their starting state, readying the game for another round of play.

```
//This function analyzes the player's input and resets the form to its
//starting state
function CheckAnswer() {

  //Analyze the player's input
  switch (document.getElementById("inputField").value) {
    case unscrambledWord:
      window.alert("Correct. You successfully decoded the secret word!");
      break;
```

```
    case "":
      window.alert("You did not type anything!");
      break;
    default:
      window.alert("Incorrect. The secret word was " +
        unscrambledWord + ".");
  }

  //Reset the form back to its starting state
  document.getElementById("checkAnswerBtn").style.visibility = "hidden";
  document.getElementById("GetWordBtn").style.visibility = "visible";
  document.getElementById("inputField").style.visibility = "hidden";
  document.getElementById("ScrambledHeading").style.visibility = "hidden";
  document.getElementById("UnscrambledHeading").style.visibility = "hidden"
  document.getElementById('ScrambledDiv').innerHTML = "";
  document.getElementById("inputField").value = "";

}
```

Step 6: Creating an External Style Sheet

The WordDecoder.html document is styled using an external style sheet named wd.css. The rules stored in this style sheet are shown here:

```
body {
  background-color: blue;
  font-family: Arial, sans-serif;
  font-weight: bold;
  text-align: center;
  }

#mainDiv {
  border-width: thick;
  border-style: double;
  border-color: blue;
  background-image: url("questionmark.png");
  width: 640px;
  height: 400px;
  padding: 20px;
  margin-top: 10%;
```

```
  margin-left: auto;
  margin-right: auto;
  }

#ScrambledDiv {
  color: red;
  font-size: 2.5pc;
  }

#inputField {
  color: midnightblue;
  background-color: honeydew;
  font-size: 1.5pc;
  font-weight: bold;
  }

#instructions {
  text-align: left;
  }
```

The first style rule affects the document's `body` section, setting the background color of the browser window to blue, setting Arial as the font type, and assigning a bold font weight. In addition, the alignment of text is set to center.

The second rule applies to the `div` element named `mainDiv` (e.g., a `div` element wrapped around all of the elements located in the `body` section). The `div` element's border is set to thick and displayed using double blue lines. A background image is assigned to the `div` element that will repeatedly display a light blue character as the element's background. Next, the height and width of the `div` element is set and a padding of 20 pixels is applied. Lastly, a margin is applied to the top, left, and right side. Note the assignment of `auto` as the value of the `margin-left` and `margin-right` properties. Together these properties instruct the browser to horizontally center the `div` element and all its contents in the browser window.

The third rule styles a `div` element named `ScrambledDiv`, setting its color to red and its font size to 2.5 times its default size. This rule controls the presentation of the scrambled word on the browser window.

The fourth rule modifies the presentation of text in the form's text control, setting its color to midnight blue, its background to honeydew, its font size to 1.5 times its default size, and its weight to bold. The last rule overrides the centered text alignment setting, inherited from the first rule, left aligning the text for the `p` element whose `ID` is `instructions`.

Step 7: Loading and Testing the Word Decoder Challenge Game

As long as you have followed along carefully with all of the instructions that have been provided, your copy of the WordDecoder.html document should now be ready for testing. If you have not already saved your new game, do so now and then load it into your web browser to see how it works. Once you are confident that everything is working just right, post a copy of it online at your website so that the world can play.

A complete copy of the source code for this project, including its style sheet and the graphics needed to create its graphic controls, is available on the book's companion web page, located at www.courseptr.com/downloads.

Summary

In this chapter you learned how to program using JavaScript. You learned different ways of embedding JavaScripts in your web pages. You learned how to collect, store, and process data using variables and to apply conditional and iterative programming logic. You also learned how to use arrays to store and process collections of data. This chapter explained the importance and benefits of using functions to improve script organization. You learned how to create functions that process arguments and return data. You also learned how to trigger function execution back on the occurrence of events. On top of all this, you learned how to create the Word Decoder Challenge Game.

Before you move on to Chapter 10, consider setting aside a few minutes to enhance the Word Decoder Challenge game by implementing the following challenges.

Challenges

1. **As currently written, the Word Decoder Challenge game only has 10 words from which to choose. These words are stored in an array named `wordArray`. Consider doubling or tripling the size of this array and its contents.**
2. **Consider modifying the game by replacing its words with more complex words and also adding the display of a sentence that provides the player with an extra hint about what the word means. One way of accomplishing this is to add a blank space to the end of each array element followed by a string describing the word. Another option is to display a third button on the browser window that the player can click on if he wants a hint.**