11

Learning JavaScript

There are many programming languages in existence today, and in this chapter, you will begin learning the basics of a programming language called JavaScript, which is by far the most common programming language used in web pages.

Although it is not possible to teach you everything there is to learn about JavaScript in one or two chapters, there are thousands of free scripts available on the Web that you can use. Therefore, the aim of this chapter is to teach you enough to start using these scripts in your web pages and to understand how they work. You should even be able to customize these scripts and write some basic scripts of your own based upon what you will learn in this and the following chapter. In addition, it will serve as a good introduction to general programming concepts.

Once you have covered the basics of JavaScript in this chapter, Chapter 12 will show you lots of examples that should both act as a library of helpful scripts you can use in your own pages and also clarify how the basic concepts you learned in this chapter work in practice.

As you will see, JavaScript gives web developers a programming language to use in web pages that allows them to perform tasks such as the following:

- Read elements from documents and write new elements and text into documents
- Manipulate or move text
- Perform mathematical calculations on data
- React to events, such as a user clicking a button
- Retrieve the current date and time from a user's computer or the last time a document was modified
- Determine the user's screen size, browser version, or screen resolution
- Perform actions based on conditions such as alerting users if they enter the wrong information into a form

You might need to read through this chapter more than once to get a good grasp of what you can do with JavaScript. Then, once you have seen the examples in the next chapter, you should have a better idea of its power. There is a lot to learn, but these two chapters should get you well on your way.

> **JavaScript is not the same as Java, which is a different programming language (although there are some similarities).**

What Is Programming About?

As you will see in this chapter, programming is largely about performing different types of *calculations* upon various types of data (including numbers, text and graphics). In all programming languages. you can perform tasks such as:

- Performing mathematical calculations on numbers such as addition, subtraction, multiplication, and division.
- Working with text to find out how long a sentence is, or where the first occurrence of a specified letter is within a section of text.
- Checking if one value (numbers or letters) matches another.
- Checking if one value is shorter or longer, lower or higher than another.
- Performing different actions based on whether a condition (or one of several conditions) is met. For example, if a user enters a number less than 10, a script or program can perform one action; otherwise it will perform a different action.
- Repeating an action a certain number of times or until a condition is met (such as a user pressing a button).

These actions might sound rather simple, but they can be combined so that they become complicated and powerful. As you will see, different sets of actions can be performed in different situations different numbers of times, to create a huge variety of results.

But before you can learn how to perform these kinds of calculations, first you need some data for the programming language to work with, and to know how the language can work with this data. For our purposes, the data we will be working will be the web page loaded in the browser at the time. When the browser loads a page, it stores it in an electronic form that programmers can then access through something known as an *interface*. The interface is a little like a predefined set of questions and commands. For example, you can ask questions like:

- What is the title of the page?
- What is the third item in the bulleted list whose `id` attribute has a value of `ToDoList`?
- What is the URL of the page in the first link on the page?

You can also use commands to tell the browser to change some of these values, or even add new elements into the page. The interface that works with web pages is called the *Document Object Model*. So, let's have a closer look at what an interface is and what an object model is?

In so-called *object-oriented* programming languages, real-life objects are represented (or modeled) using a set of *objects*, which form an *object model*. For example, a `car` object might represent a car, a `basket` object might represent a shopping basket, and a `document` object could represent a document such as a web page.

Each object can have a set of *properties* that describes aspects of the object. A `car` object might have properties that describe its color or engine size. A `basket` object might have properties that describe the number of items it contains or the value of those items. A `document` object has properties that describe the background color of the web page or the title of the page.

Then there are *methods*; each method describes an action that can be done to (or with) the object. For example, a method of an object representing a car might be to accelerate, or to change gear. A method of a shopping basket might be to add an item, or to recalculate the value of items in the basket. A method on a document could be to write a new line of text into the web page.

Finally, there are *events*; in a programming language, an event is the object putting up its hand and saying "*x* just happened," usually because a program might want to do something as a result of the event. For example, a `car` object might raise an event to say the ignition started, or that it is out of fuel. A `basket` might raise an event when an item is added to it, or when it is full. A document might raise an event when the user presses Submit on a form, or clicks a link. Furthermore, an event can also trigger actions; for example, if a car is out of fuel then the car will stop.

An object model is therefore a description of how a program can represent real-life entities using a set of objects, and it also specifies a set of methods, properties, and events an object may have.

All of the main browsers implement an object model called the *Document Object Model* that was devised to represent web pages. In this Document Object Model, the page as a whole is represented using a `document` object; then the links in that page are represented using a `links` object, the forms are represented in a `forms` object, images are represented using an `image` object, and so on.

So, the Document Object Model describes how you can:

- Get and set *properties* of a web page such as the background color.
- Call *methods* that perform actions such as writing a new line into a page.
- React to *events* such as a user pressing a Submit button on a form.

Web browsers implement the Document Object Model as a way for programming languages to access (and work with) the content of web pages. Once you have learned how the Document Object Model allows you to access and change web pages, you can then see how the programming language can ask the Document Object Model about the web page, perform calculations with the data it receives back, and then tell the browser to change something based upon these calculations.

If you think of the Document Object Model as an *interface* between the browser and the programming language, you can compare it to a remote control that acts as the interface between your TV and you. You know that pressing 1 on your remote control will turn your TV to channel 1, or that the volume-up button will increase the volume. If you call up the TV schedule, you can see what is on next, and you might choose to change the channel (or turn the TV off) as a result. Similarly, the Document Object Model

is like the remote control that allows a programming language (such as JavaScript) to work with the browser (and the web page in that browser).

To take the remote control analogy a little further, it doesn't matter what language you speak; as long as someone presses the right button on the remote, the TV behaves in the same way. If you get a new TV, the core functions of the remote will be very similar. When working with the Document Object Model, it does not matter what language you program with. As long as you use the right properties and methods, the effect will be the same.

Also, under the covers, it does not matter how your remote tells the TV to turn up the volume, as long as the effect is the same (the volume goes up). Likewise, it doesn't matter how the browser retrieves the title of the page when you ask for it, or the background color of the page when you ask for that, as long as it returns it to you in the way you expect (which is specified in the Document Object Model).

The point of this analogy is that your script will achieve the same goal in any browser that implements the Document Object Model. It does not matter how a browser implements the Document Object Model, as long as you can ask it for properties, or to perform methods in the standard way set out in the Document Object Model, and you then get the results in an expected format. How the browser actually does the work under the hood does not matter.

In this chapter, you start by looking at how the Document Object Model allows you to work with a web page; then you look at JavaScript to see what you can do with this data and how you can use it to get and set values of the document, call methods, and respond to events.

How to Add a Script to Your Pages

JavaScript can either be embedded in a page or placed in an external script file (rather like CSS). But in order to work in the browser, the browser must have JavaScript enabled. (The major browsers allow users to disable JavaScript, although very few people do.)

You add scripts to your page inside the `<script>` element. The `type` attribute on the opening `<script>` tag indicates what scripting language will be found inside the element, so for JavaScript you use the value `text/JavaScript`.

> *There are several other scripting languages that do a very similar job to JavaScript (such as VBScript or Perl), but JavaScript is the main programming language used in web browsers.*

Here you can see a very simple script that will write the words "My first JavaScript" into the page (`ch11_eg01.html`):

```
<html>
<body>
 <p>
  <script type="text/javascript">
    document.write("My first JavaScript")
  </script>
 </p>
</body>
</html>
```

In this case, we are using the `write()` method to add a new line of text into the web page (and the web page is represented using the `document` object). The text is added into the page where the script is written in the page. Figure 11-1 shows what this simple page would look like.

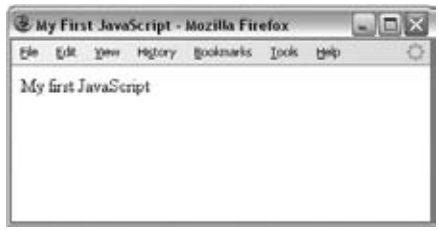

Figure 11-1

Where you put your JavaScript within a page is very important. If you put it in the body of a page — as in this example — then it will run (or *execute*) as the page loads. Sometimes, however, you will want a script ready to use as soon as the page has loaded, or you might want to use the same script in several parts of the page, in which case it tends to live inside the `<head>` element on the page (because scripts in the head of the page load before the page is displayed). Scripts that live in the head of a page are triggered (or *called*) by an event such as when the page finishes loading or when a visitor presses the submit button on a form.

You can also write JavaScript in external documents that have the file extension `.js` (just in the same way that you can write external style sheets). This is a particularly good option because:

- If your script is used by more than one page you do not need to repeat the script in each page that uses it.
- If you want to update your script you need only change it in one place.
- It makes the XHTML page cleaner and easier to read.

When you place your JavaScript in an external file, you need to use the `src` attribute on the `<script>` element; the value of the `src` attribute should be an absolute or relative URL pointing to the file containing the JavaScript. For example:

```
<script type="JavaScript" src="scripts/validation.js"></script>
```

So there are three places where you can put your JavaScripts, and a single XHTML document can use all three because there is no limit on the number of scripts one document can contain:

- **In the <head> of a page:** These scripts will be called when an event triggers them.
- **In the <body> of a page:** These scripts will run as the page loads.
- **In an external file:** If the link is placed inside the `<head>` element, the script is treated the same as when the script lives inside the head of the document waiting for an event to trigger it, whereas if it is placed in the `<body>` element it will act like a script in the body section and execute as the page loads.

Sometimes you will see examples of JavaScript written inside an XHTML comment.

```
<script type="text/javascript">
  <!--
    document.write("My first JavaScript")
  //-->
</script>
```

This was done because some old browsers do not support JavaScript, and therefore the comment hides the script from them. All browsers that support JavaScript just ignore these comments in the `<script>` element. Note how two forward slash characters (`//`) precede the closing characters of the XHTML comment. This is actually a JavaScript comment that prevents the JavaScript from trying to process the `-->` characters.

In XHTML, you should not use characters such as the angle brackets for anything other than tags. If you do, they are supposed to go in something known as a *CDATA section* (which tells the browser that the contents are not markup, and should not be processed as markup). Because JavaScript uses the angle brackets, even though it will not cause a problem in major browsers, you could put your scripts inside a CDATA section like this for technical accuracy (and to ensure that your page validates):

```
<script type="text/javascript">
//<![CDATA[
...
]]>
</script>
```

If you were to write a site that had to work on very old browsers, a good alternative to worrying about browsers that cannot support scripts is to use external scripts because if the browser cannot process the `<script>` element, it will not even try to load the document containing the script.

Comments in JavaScript

You can add comments to your JavaScript code in two ways. The first way, which you have already seen, allows you to comment out anything on that line after the comment marks. Here, anything on the same line after the two forward slash characters is treated as a comment:

```
<script type="text/javascript">
  document.write("My first JavaScript") // comment goes here
</script>
```

You can also comment out multiple lines using the following syntax, holding the comment between the opening characters `/*` and closing characters `*/` like so:

```
/* This whole section is commented
out so it is not treated as a part of
the script. */
```

This is similar to comments in CSS.

As with all code, it's good practice to comment your code clearly, even if you are the only person likely to be using it, because what may have seemed clear when you wrote a script may not be so obvious when you come back to it later.

The <noscript> Element

The `<noscript>` element offers alternative content for users who have disabled JavaScript. It can contain any XHTML content that the author wants to be seen in the browser if the user does not have JavaScript enabled.

Strictly speaking, the W3C's recommendations say that the content of this element should be displayed only when the browser does not support the scripting language required; however, the browser manufacturers have decided that it should also work when scripting is disabled.

Try It Out — Creating an External JavaScript

You have already seen a basic example of a JavaScript that writes to a page. In this example, you will move that code to an external file. The external file is going to be used to write some text to the page.

1. Open your editor and type the following code:

```
document.write("Here is some text from an external file.");
```

2. Save this file as `external.js`.

3. Open a new page in your editor and add the following. Note that the `<script>` element is empty this time, but carries the `src` attribute whose value is the JavaScript file (just as it does on the `<img>` element). This may be a relative or full URL.

```
<html>
<body>
  <script src="external.js" type="text/JavaScript">
  </script>
  <noscript>This only shows if the browser has JavaScript turned off.
  </noscript>
</body>
</html>
```

4. Save this example as `ch11_eg02.html` and open it in your browser. You should see something like Figure 11-2.

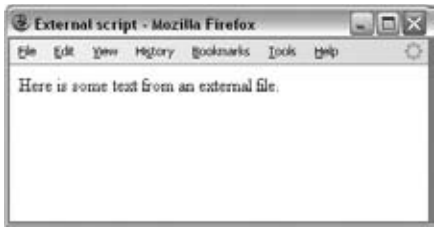

Figure 11-2

You can use this approach to include external JavaScripts in either the `<head>` or the `<body>` of a document. If you place them in the body of the document, as in this example, they are executed as the page loads — just as if the script were actually in the page there. If you place them in the head, then they will load when the page loads, and be triggered by an event.

I often use external JavaScript files for functions and place the `<script>` element in the head of the document because it allows me to re-use scripts on different sites I develop and ensures that the XHTML documents focus on content rather than being littered with scripts. You will see more of this approach later in the chapter.

The Document Object Model

As I mentioned at the start of the chapter, JavaScript by itself doesn't do much more than allow you to perform calculations or work with basic strings. In order to make a document more interactive, the script needs to be able to access the contents of the document and know when the user is interacting with it. The script does this by interacting with the browser by using the properties, methods and events set out in the interface called the Document Object Model.

The Document Object Model, or DOM, represents the web page that is loaded into the browser using a series of objects. The main object is the `document` object, which in turn contains several other child objects.

The DOM explains what *properties* of a document a script can retrieve and which ones it can alter; it also defines some *methods* that can be called to perform an action on the document.

Figure 11-3 shows you an illustration of the Level 0 HTML Document Object Model (as you will see shortly, there are different levels of the DOM).

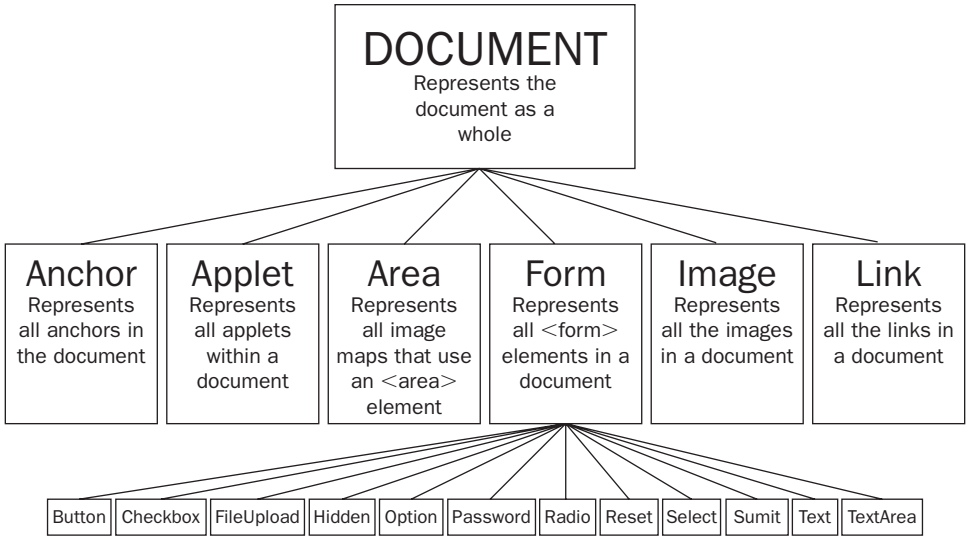

Figure 11-3

As you can see from Figure 11-3 the document object represents the whole document, and then each of the child objects represents a *collection* of similar tags within that document:

- The *anchor* collection represents all the anchors in a document that you can link to (`<a>` elements with a `name` attribute).
- The `applet` collection represents all the applets within a document.
- The `area` collection represents all the image maps that use an `<area>` element in the document.
- The `forms` collection contains all the `<form>` tags in the document.
- The `image` collection represents all the images in a document.
- The `link` collection represents all the hyperlinks within a page.

The `forms` collection also has child objects to represent each of the different types of form controls that can appear on a form: `Button`, `CheckBox`, `FileUpload`, `Hidden`, `Option`, `Password`, `Radio`, `Reset`, `Select`, `Submit`, `Text`, and `TextArea`.

To better understand how to access the document using the DOM, take a look at the following simple document, which contains one form and two links:

```
<h1>User Registration</h1>
<form name="frmLogin" action="login.aspx" method="post">
  Username <input type="text" name="txtUsername" size="12" /> <br />
  Password <input type="password" name="pwdPassword" size="12" /> <br />
  <input type="submit" value="Log In" />
</form>
<p>New user? <a href="register.aspx">Register here</a> |
 <a href="lostPassword.aspx">Retrieve password</a>.</p>
```

You can see this page in Figure 11-4.

Figure 11-4

The DOM would allow a script to access:

- The content of the form as part of the `forms` collection
- The two links as part of the `links` collection

Accessing Values Using Dot Notation

In order to access the different objects in the DOM, you list the objects in the tree shown in Figure 11-3, starting with the `document` object, working down to the object that holds the data you are after. Each object is separated by a period or full-stop character; hence, this is known as a *dot notation*.

For example, in order to access the first link in the document, you would want the `links` object, which is a child of the `document` object, so you could use something like this:

```
document.links[0].href
```

There are four parts of this statement, three of which are separated by periods, to get to the first link:

- The word `document` indicates I am accessing the `document` object.
- The word `links` corresponds to the `links` collection (after all, this example is to retrieve the value of the first link in the document).
- The `[0]` indicates that I want the first link in the document. Rather confusingly, the items of a collection are numbered from 0 rather than 1, which means the second link in the links collection is represented using `[1]`, the third using `[2]`, and so on.
- I have indicated that I want to retrieve the `href` property for this link.

Each object has different properties that correspond to that type of element; for example, links have properties such as the `href` property that accesses the value of the `href` attribute on this `<a>` element. Similarly, a `<textarea>` object has properties such as `cols`, `disabled`, `readOnly`, and `rows`, which correspond to the attributes on that element.

Rather than using the names of the type of object (such as forms and links), you can use the value of `name` attributes on the elements to navigate through the document. For example, the following line requests the value of the password box:

```
document.frmLogin.pwdPassword.value
```

Again, there are four parts to this statement:

- The `document` object comes first again as it represents the whole page (it is the top-level object).
- The name of the form, `frmLogin`.
- This is followed by the name of the form control, `pwdPassword`.
- Finally, the property I am interested in is the value of the password box, and this property is called `value`.

Both of these approaches enable you to navigate through a document, choosing the elements and properties of those elements you are interested in. Then you can retrieve those values, perform calculations upon them, and provide alternative values.

For the purpose of learning JavaScript, we are dealing with what is often called DOM Level 0 in this chapter because it works in most browsers. Its syntax was created before the W3C created its DOM Level 1, 2, and 3 recommendations (which get more complicated and have varying levels of support in different browsers). Once you are familiar with the basics, you can move on to look at these in more detail if you wish.

There is also a second type of object model, the Browser Object Model, which makes features of the browser available to the programmer, such as the `window` *object, which can be used to create new pop-up windows. The Browser Object Model can vary from browser to browser, although most browsers have common support for core functionality. You learn about the* `window` *object later in the chapter.*

The Document Object

In this section, we are going to take a closer look at the document object — this is the main object in the DOM and represents the document as a whole (and therefore allows you to access other child elements).

As you already know, an object can have properties that can tell you about the object, and methods to perform an action upon that object.

Once you understand how to work with one object, it's much easier to work with all kinds of objects — and you will come across many different types of objects when you start programming.

Properties of the Document Object

In the following table, you can see the properties of the document object. Several of these properties correspond to attributes that would be carried by the `<body>` element, which contains the document.

Many properties can be set as well as read. If you can set a property, it is known as a read/write property (because you can read it or write to it), whereas the ones you can only read are known as read-only. You can see which properties can be read and which can be written to in the last column of the table that follows.

Property Name	Purpose	Read/Write
`alinkColor`	Specifies link colors (like the deprecated `alink` attribute on the `<body>` element).	Read/write
`bgcolor`	Specifies background color (like the deprecated `bgcolor` attribute on the `<body>` element).	Read/write
`fgcolor`	Foreground/text color (like the deprecated `text` attribute of the `<body>` element).	Read/write
`lastModified`	The date the document was last modified. (This is usually sent by the web server in things known as HTTP headers that you do not see).	Read only
`linkColor`	Specifies link colors (like the deprecated `link` attribute of the `<body>` element).	Read/write
`referrer`	The URL of the page that users came from if they clicked a link. It is empty if there is no referrer.	Read only
`title`	The `title` of the page in the `<title>` element.	Read only (until IE 5 and Netscape 6 and later versions)
`vlinkColor`	The color of links that have been clicked on (like the deprecated `vlink` attribute of the `<body>` element).	Read/write

The properties that correspond to deprecated attributes of the `<body>` element should generally be avoided because CSS should be used to style text, links, and backgrounds.

To access any of the properties, again you use dot notation, so you can access the title of a document like so:

```
document.title
```

Or you could find out the date a document was last modified like so:

```
document.lastModified
```

Note that if the server does not support the `lastModified` property, IE will display the current date, while other browsers will often display `1 January 1970` (which is the date from which most computers calculate all dates).

Methods of the Document Object

Methods perform actions and are always written followed by a pair of brackets. Inside the brackets of some methods, you can see things known as *parameters* or *arguments*, which can affect what action the method takes.

For example, in the table that follows, you can see two methods that write new content into the web page. Both of these methods need to know what should be written into the page, so they take a string as an argument (a *string* is a sequence of characters that may include letters, numbers, spaces, and punctuation), and the string is what gets written into the page.

Method Name	Purpose
`write(string)`	Allows you to add text or elements into a document
`writeln(string)`	The same as `write()`, but adds a new line at the end of the output (as if you had pressed the Enter key after you had finished what you were writing)

You have already seen the `write()` method of the document object in `ch11_eg01.html`, which showed how it can be used to write content into a document:

```
document.write('This is a document');
```

You can also put something called an *expression* as a parameter of the `write()` method. For example, the following will write the text string `Page last modified on` followed by the last modified date of the document.

```
document.write('Page last modified on ' + document.lastModified);
```

You will see more about expressions later in the chapter, but in this case, the expression *evaluates* into (or results in) a string. For example, you might see something like `Page last modified on 12th December 2009`.

Now that you've seen the properties and methods of the `document` object, let's look at the properties and methods of some of the other objects, too.

The Forms Collection

The `forms` collection holds references corresponding to each of the `<form>` elements in the page. This might sound a little complicated, but you can probably imagine a web page that has more than one form — a login form, a registration form for new users, and a search box on the same page — the DOM deals with this by having a separate `form` object to represent each of the individual forms.

Imagine you want to use a script that can access the address of the page that the login form sends its data to. In the XHTML document you would be looking at the `action` attribute on the opening `<form>` tag. There is a corresponding `action` property on the `form` object, which holds the value of this attribute. So, you would need to access the `form` object that represented the login form, then retrieve the value of the property called `action`.

If the login form is the first form in the document, you might use the following index number to select the appropriate form and access the value of its `action` attribute (remember that index numbers start at 0 for the first form, 1 for the second form, 2 for the third, and so on):

```
document.forms[0].action
```

Alternatively, you can directly access that `form` object using its name (this is generally considered the preferred option because if another form were added to the page it could break the script):

```
document.frmLogin.action
```

The form that you select has its own object with properties and methods (each property generally corresponds to the attributes of the `<form>` element). Once you have seen the properties and methods of the forms, you will then see the objects, properties, and methods that correspond to the different types of form control.

Properties of the Form Objects

The following table lists the properties of the `form` objects; as you will see, most of them correspond to the attributes of the `<form>` element.

Property Name	Purpose	Read/Write
`action`	Specifies the value of the `action` attribute on the `<form>` element	Read/write
`length`	Gives the total number of form controls that are in this form	Read only
`method`	The value of the `method` attribute of the `<form>` element (either `get` or `post`)	Read/write
`name`	The value of the `name` attribute of the `<form>` element	Read only
`target`	The value of the `target` attribute of the `<form>` element	Read/write

Methods of the Form Objects

The following table lists the methods of the `form` objects.

Method Name	Purpose
`reset()`	Resets all `form` elements to their default values
`submit()`	Submits the form

A <form> element can have an attribute called onsubmit whose value is a script that should run when the user manually presses the Submit button. If JavaScript is used to submit a form, this attribute is ignored.

Form Elements

When you access a form, you usually want to access one or more of its elements. Each <form> element has an elements collection object as a property, which represents all of the elements in that form. This works in a similar way to the forms collection; it allows you to access the elements you want by index (an index being a number corresponding to their order in the document that starts with 0). Alternatively, you can use their names.

Here are some of the things you might want to do with the elements in a form:

- **Text fields:** Read data a user has entered or write new text to these elements.
- **Checkboxes and radio buttons:** Test if they are checked and check or uncheck them.
- **Buttons:** Disable them until a user has selected an option.
- **Select boxes:** Select an option or see which option the user has selected.

Properties of Form Elements

The following table lists the properties that correspond to the elements that may appear on a form.

Property	Applies to	Purpose	Read/Write
checked	Checkboxes and radio buttons	Returns true when checked or false when not	Read/write
disabled	All except hidden elements	Returns true when disabled and user cannot interact with it	Read/write
form	All elements	Returns a reference to the form it is part of	Read only
length	Select boxes	Number of options in the <select> element	Read only
name	All elements	Accesses the value of the name attribute of the element	Read only
selectedIndex	Select boxes	Returns the index number of the currently selected item	Read/write
type	All	Returns type of form control	Read only
value	All	Accesses the value attribute of the element or content of a text input	Read/write

If you want one of the form controls to be disabled until someone has performed an action — for example, if you want to disable the Submit button until the user has agreed to the terms and conditions — you should disable the form control in the script as the page loads, rather than disabling it in the form control itself using XHTML in case the user has turned off JavaScript (if this were the case, they would not be able to enable the Submit button). You will see more about this topic in Chapter 12.

Methods of Form Elements

The following table lists the methods of form elements.

Property Name	Applies to	Read/Write
`blur()`	All except hidden	Takes focus away from currently active element to next in tabbing order
`click()`	All except text	Simulates clicking the mouse over the element
`focus()`	All except hidden	Gives focus to the element
`select()`	Text elements except hidden	Selects the text in the element

Try It Out Collecting Form Data

In this example, you are going to retrieve the value of a textbox and write it into something known as a JavaScript alert box. The main purpose of the example is to show you how the value of the form can be retrieved, although it will also introduce you to an event and the JavaScript alert box.

The alert box is created using a method called `alert()` that is part of the JavaScript language (not the DOM). Alert boxes used to be very common on web sites, but you do not see them as much any more. However, in examples like this they are helpful to demonstrate how JavaScript can access documents.

The simple form will contain just one text input and a submit button. When you enter something into the textbox and click the submit button, the value you have entered in the textbox will appear in the alert box. You can see the page once the user has clicked the submit button in Figure 11-5.

Figure 11-5

When you click OK, the alert box disappears.

1. Create a skeleton document for a Transitional XHTML page, and add a heading that explains what the example demonstrates:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>
  <title>Accessing form data</title>
</head>
<body>
  <h1>Accessing Form Data</h1>
</body>
</html>
```

2. Add a `<form>` element to the body of the document. The form should contain a text input for a username and a submit button, like so:

```
<body>
<h1>Accessing Form Data</h1>
<form name="frmLogin">
  Username <input type="text" name="txtUsername" size="12" /><br /><br />
  <input type="submit" value="Click here" />
</form>
</body>
```

3. Add the `onsubmit` attribute to the `<form>` element, and give it the following value:

```
<form name="frmLogin" onsubmit="alert(document.frmLogin.txtUsername.value)">
  Username <input type="text" name="txtUsername" size="12" /> <br />
  <input type="submit" value="Click here" />
</form>
```

Save the file as `ch11_eg03.html`, and open it in your browser. When you enter something into the text input and click Submit, you should see an alert box like that in Figure 11-5, which displays the value you entered into the textbox.

When the `onsubmit` event fires (which happens when the user clicks the Submit button), this simple line of script is run. In this case, the `alert()` method is called:

```
alert(document.frmLogin.txtUsername.value)
```

The `alert(string)` method allows you to write a string into the pop-up alert box. Like the `write()` method of the document object, which you saw earlier, the string does not need to be the actual text you want to display. In this example, rather than writing the same string to the alert box every time the script is run, whatever the user has entered into the text box will be written to the alert box.

You can see that inside the `alert()`, the text input has been selected using `document.frmLogin.txtUsername` along with the `value` property of this form control. So the value of the text input is written to the alert box.

As you have seen the `alert()` box takes a string as a parameter (just like the `write()` method you saw earlier), but we are not specifying the exact words to write out (rather we are telling the method to find out what the user entered). When you tell a method (such as the `alert()` method or the `write()` method) exactly what to write you put the value in double quote marks, but when you want the script to collect the information it is to display, you do not use the double quotes.

When the user clicks the Submit button, the `onsubmit` event fires, which creates the alert box that contains the value of the text input.

Images Collection

The `images` collection provides references to image objects, one representing each image in a document. These can again be referenced by name or by their index number in the collection. So the `src` attribute of the first image can be found using the index number like so:

```
document.images[0].src
```

or using its name; for example, if the image had a `name` attribute whose value was `imgHome`, you could access it using the following:

```
document.imgHome.src
```

There are no methods for the image objects in the Level 0 DOM, although there are several properties.

Properties of the Image Object

The following table lists the properties of the image object.

Property	Purpose	Read/Write
`border`	The `border` attribute of the `<img>` element, specifying the width of the border in pixels	Read/write
`complete`	Indicates whether an image has loaded successfully	Read only
`height`	The `height` attribute of the `<img>` element, specifying the height of the image in pixels	Read/write
`hspace`	The `hspace` attribute of the `<img>` element, specifying the gap above and below an image to separate it from its surrounding elements	Read/write
`lowsrc`	The `lowsrc` attribute of the `<img>` element (indicating a lower resolution version of the image)	Read/write
`name`	The `name` attribute of the `<img>` element	Read/write
`src`	The `src` attribute of the `<img>` element, indicating where the file is located	Read/write

Property	Purpose	Read/Write
`vspace`	The `vspace` attribute of the `<img>` element, specifying the gap to the left and right of an image to separate it from its surrounding elements	Read/write
`width`	The `width` attribute of the `<img>` element, specifying the width of the image in pixels	Read/write

Try It Out A Simple Image Rollover

In this example, you are going to see how to replace one image with another one when the user rolls over the image with the mouse. Most developers now use CSS to create image rollovers in their code, but this is a good way to demonstrate how to access image properties.

In this example, you are going to see two simple images, both saying "click here." When the page loads, the image will be in green with white writing, but as soon as the user hovers over the image, the script will access and change the `src` property of the image and change it to the image that is red with white writing.

1. Create the skeleton of a Transitional XHTML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>
  <title>Image Rollover</title>
</head>
<body>
</body>
</html>
```

2. Add the following link and image to the body of your document:

```
<p>Hover over the image with your mouse to see the simple rollover effect.
<br/>
<a href=""
  <img src="images/click_green.gif" width="100" height="50" border="0"
       alt="Example button" name="button" />
</a>
</p>
```

3. Now add the following `onmouseover` and `onmouseout` event handler attributes to the `<a>` element with the specified values:

```
<a href=""
   onmouseover="document.images.button.src='images/click_red.gif';"
   onmouseout="document.images.button.src='images/click_green.gif'">
```

When the user rolls over the image, the `onmouseover` event fires, and when the user moves off it again, the `onmouseout` event fires. This is why there are separate attributes that correspond to each of these events, and when one of these two events fires, the script held as a value for the corresponding attribute is executed.

The script in the `onmouseover` and `onmouseout` event handler attributes tells the browser to change the `src` attribute of the image, and therefore a different image is displayed to the user.

The first (`onmouseover`) indicates what should happen when the mouse is placed over the image; the second (`onmouseout`) indicates what should be done when the mouse is moved off the image.

When the user puts the mouse over an image, the `src` property of the image inside the link — named using the notation `document.images.button` — is changed.

4. Save this example as `ch11_eg4.html` and open it in your browser. Then roll your mouse over the image (without clicking it). You should see something like Figure 11-6 with the mouse over the image.

Figure 11-6

The `<img>` element must have a `name` attribute so that the image can be referenced in this way in the link (otherwise you would have to use its index in the images collection). It is generally best to use the name in situations like this, rather than the index of that image in the images collection, because if you were to add another image into the document before this one the whole script would need changing.

Note that if no event indicated what should happen when the user takes the mouse off the image, it would remain red rather than turning back to green. An image rollover script is a good example of changing or *setting* that property rather than just reading it.

Different Types of Objects

You will come across several types of objects in JavaScript, each of which is responsible for a related set of functionalities. For example, the `document` object has methods and properties that relate to the document; the `forms` collection, which is part of the `document` object, deals with information regarding forms; and so on. As you are about to see, there can be lots of different objects, each of which deals with a different set of functionalities and properties.

So, here are some of the types of objects you are likely to come across:

- **W3C DOM objects:** These are like those already covered in this chapter, although in more recent browsers several more objects are made available to allow you more control over a document.
- **Built-in objects:** Several objects are part of the JavaScript language itself. These include the `date` object, which deals with dates and times, and the `math` object, which provides mathematical functions. You will be learning more about these built-in objects later in the chapter.
- **Custom objects:** If you start to write advanced JavaScript, you might even start creating your own JavaScript objects that contain related functionality; for example, you might have a `validation` object that you have written just to use to validate your forms.

While it is not possible to cover the creation of custom objects in this chapter, you learn about the built-in objects later in this chapter.

Starting to Program with JavaScript

Now that you have seen how JavaScript is able to access a document in the web browser using the DOM, it is time to look at how you use these properties and methods in scripts.

As I mentioned earlier, a programming language mainly performs calculations. So here are the key concepts you need to learn in order to perform different types of calculations:

- A *variable* is used to store some information; it's like a little bit of the computer's memory where you can store numbers, strings (which are a series of characters), or references to objects. You can then perform calculations to alter the data held in variables within your code.
- *Operators* perform functions on variables. There are different types of operators — for example:
  - Arithmetic operators enable you to do things such as add (+) numbers together, or subtract (-) one from another (providing they are numbers).
  - Comparison operators enable you to compare two strings and see if one is the same as the other, or different (for example, whether x is equal to y or whether a is greater than b).
- *Functions* are parts of a script that are grouped together to perform a specific task. For example, you could have a function that calculates loan repayments, and when you tell the loan calculator function the information it needs (the amount of money to be borrowed, the number of years the loan will last, and the interest rate) the function will be able to return the monthly payment. Functions are objects in their own right and are very similar to things called methods; one of the key differences is that methods often belong to an object already, whereas functions are customized.
- *Conditional statements* allow you to perform different actions based upon a condition. For example, a condition might be whether a variable holding the current time is greater than 12. If the condition is true, code to write "Good Afternoon" might be run. Whereas, if it is less than 12, a different block of code saying "Good Morning" could be shown.

- *Loops* can be set up so that a block of code runs a specified number of times or until a condition is met. For example, you can use a loop to get a document to write your name 100 times.
- There are also several built-in JavaScript objects that have methods that are of practical use. For example, in the same way that the `document` object of the DOM has methods that allowed you to write to the document, the built-in JavaScript `date` object can tell you the date, time, or day of the week.

The following section looks at these key concepts in more detail.

Variables

Variables are used to store data. To store information in a variable, you can give the variable a name and put an equal sign between it and the value you want it to have. Here is an example:

```
userName = "Bob Stewart"
```

The variable is called `userName` and the value is `Bob Stewart`. If no value is given, then its value is *undefined*.

The script can access the value of the variable using its name (in this case `userName`). It can also change the value.

You can create a variable, but not store anything with it by using the `var` keyword; this is known as *declaring* a variable (unlike some other languages, you do not have to declare a variable before you can use it, although it is commonly considered good practice to do so).

```
var userName
```

There are a few rules you must remember about variables in JavaScript:

- They must begin with a letter or the underscore character.
- Variable names are case-sensitive.
- Avoid giving two variables the same name within the same document as one might override the value of the other, creating an error.
- Do not call two variables the same name, but use different cases to distinguish them (e.g., `username` and `UserName`) as this is a common source of confusion later.
- Try to use descriptive names for your variables. This makes your code easier to understand (and will help you debug your code if there is a problem with it).

Assigning a Value to a Variable

When you want to give a value to a variable, you put the variable name first, then an equal sign, and then on the right the value you want to assign to the variable. You have already seen values being assigned to these variables when they were declared a moment ago. So, here is an example of a variable being assigned a value and then the value being changed:

```
userName = "Bob Stewart"
userName = "Robert Stewart"
```

`userName` is now the equivalent of `Robert Stewart`.

Lifetime of a Variable

When you declare a variable in a function, it can be accessed only in that function. (You will learn about functions shortly.) After the function has run, you cannot call the variable again. Variables in functions are called *local variables*.

Because a local variable works only within a function, you can have different functions that contain variables of the same name (because each is recognized by that function only).

> *If you declare a variable using the* `var` *keyword inside a function, it will use memory up only when the function is run, and once the function has finished it will not take up any memory.*

If you declare a variable outside a function, all the functions on your page can access it. The lifetime of these variables starts when they are declared and ends when the page is closed.

Local variables take up less memory and resources than page-level variables because they require only the memory during the time that the function runs, rather than having to be created and remembered for the life of the whole page.

Operators

The operator itself is a keyword or symbol that does something to a value when used in an *expression*. For example, the arithmetic operator + adds two values together.

The symbol is used in an expression with either one or two values and performs a calculation on the values to generate a result. For example, here is an expression that uses the multiplication operator:

```
area = (width * height)
```

An expression is just like a mathematical expression. The values are known as *operands*. Operators that require only one operand (or value) are sometimes referred to as *unary operators*, while those that require two values are sometimes called *binary operators*.

The different types of operators you will see in this section are:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- String operators

You will see lots of examples of the operators in action both later in this chapter and in the next chapter. First, however, it's time to learn about each type of operator.

Arithmetic Operators

Arithmetic operators perform arithmetic operations upon operands. (Note that in the examples in the following table, x = 10.)

Symbol	Description	Example (x = 10)	Result
+	Addition	x+5	15
-	Subtraction	x-2	8
*	Multiplication	x*3	30
/	Division	x/2	5
%	Modulus (division remainder)	x%3	1
++	Increment (increments the variable by 1 — this technique is often used in counters)	x++	11
--	Decrement (decreases the variable by 1)	x--	9

Assignment Operators

The basic assignment operator is the equal sign, but do not take this to mean that it checks whether two values are equal. Rather, it's used to assign a value to the variable on the left of the equal sign, as you have seen in the previous section, which introduced variables.

The basic assignment operator can be combined with several other operators to allow you to assign a value to a variable *and* perform an operation in one step. For example, take a look at the following statement where there is an assignment operator and an arithmetic operator:

```
total = total - profit
```

This can be reduced to the following statement:

```
total -= profit
```

While it might not look like much, this kind of shorthand can save a lot of code if you have a lot of calculations like this (see table that follows) to perform.

Symbol	Example Using Shorthand	Equivalent Without Shorthand
+=	x+=y	x=x+y
-=	x-=y	x=x-y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
%=	x%=y	x=x%y

Comparison Operators

As you can see in the table that follows, comparison operators compare two operands and then return either `true` or `false` based on whether the comparison is true or not.

Note that the comparison for checking whether two operands are equal is two equal signs (a single equal sign would be an assignment operator).

Operator	Description	Example
==	Equal to	1==2 returns `false` 3==3 returns `true`
!=	Not equal to	1!=2 returns `true` 3!=3 returns `false`
>	Greater than	1>2 returns `false` 3>3 returns `false` 3>2 returns `true`
<	Less than	1<2 returns `true` 3<3 returns `false` 3<1 returns `false`
>=	Greater than or equal to	1>=2 returns `false` 3>=2 returns `true` 3>=3 returns `true`
<=	Less than or equal to	1<=2 returns `true` 3<=3 returns `true` 3<=2 returns `false`

Logical or Boolean Operators

Logical or Boolean operators return one of two values: `true` or `false`. They are particularly helpful when you want to evaluate more than one expression at a time.

Operator	Name	Description	Example (where x=1 and y=2)
`&&`	And	Allows you to check if both of two conditions are met	`(x < 2 && y > 1)` Returns `true` (because both conditions are met)
`??`	Or	Allows you to check if one of two conditions are met	`(x < 2 ?? y < 2)` Returns `true` (because the first condition is met)
`!`	Not	Allows you to check if something is not the case	`! (x > y)` Returns `true` (because x is not more than y)

The two operands in a logical or Boolean operator evaluate to either `true` or `false`. For example, if `x=1` and `y=2`, then `x<2` is `true` and `y>1` is `true`. So the following expression:

```
(x<2 && y>1)
```

returns `true` because both of the operands evaluate to true (you can see more examples in the right-hand column of this table).

String Operator (Using + with Strings)

You can also add text to strings using the + operator. For example, here the + operator is being used to add two variables that are strings together:

```
firstName = "Bob"
lastName = "Stewart"
name = firstName + lastName
```

The value of the `name` variable would now be `Bob Stewart`. The process of adding two strings together is known as *concatenation*.

You can also compare strings using the comparison operators you just met. For example, you could check whether a user has entered a specific value into a text box. (You will see more about this topic when you look at the "Conditional Statements" section shortly.)

Functions

A function is made up of related code that performs a particular task. For example, a function could be written to calculate area given width and height. The function can then be called elsewhere in the script, or when an event fires.

Functions are either written in the `<head>` element or in an external file that is linked from inside the `<head>` element, which means that they can be reused in several places within the page.

How to Define a Function

There are three parts to creating or defining a function:

- Define a name for it.
- Indicate any values that might be required; these are known as arguments.
- Add statements to the body of the function.

For example, if you want to create a function to calculate the area of a rectangle, you might name the function `calculateArea()` (note that a function name should be followed by parentheses). In order to calculate the area, you need to know the rectangle's width and height, so these would be passed in as *arguments* (arguments are the information the function needs to do its job). Inside the body of the function (the part between the curly braces) are the *statements*, which indicate that area is equal to the width multiplied by the height (both of which have been passed into the function). The area is then returned.

```
function calculateArea(width, height) {
  area = width * height
  return area
}
```

If a function has no arguments it should still have parentheses after its name — for example, you might have a function that will run without any extra information passed as an argument such as, `logOut()`.

How To Call a Function

The `calculateArea()` function does nothing sitting on its own in the head of a document; it has to be *called*. In this example, you can call the function from a simple form using the `onclick` event, so that when the user clicks the Submit button the area will be calculated and shown in an alert box.

Here you can see that the form contains two text inputs for the width and height, and these are passed as arguments to the function like so (`ch11_eg05.html`):

```
<form name="frmArea" action="">
Enter the width and height of your rectangle to calculate the size:<br />
Width: <input type="text" name="txtWidth" size="5" /><br />
Height: <input type="text" name="txtHeight" size="5" /><br />
<input type="button" value="Calculate area"
  onclick="alert(calculateArea(document.frmArea.txtWidth.value,
  document.frmArea.txtHeight.value))" />
</form>
```

Take a closer look at what is happening when the `onclick` event fires. First, a JavaScript alert is being called, and then the `calculateArea()` function is being called inside the alert, so that the area is the value that is written to the alert box. Inside the parentheses where the `calculateArea()` function is being called, the two parameters being passed are the values of the width text box (`document.frmArea.txtWidth.value`) and the height text box (`document.frmArea.txtWidth.value`) using the dot notation you learned earlier in the section on the DOM.

The Return Statement

Functions that return a result must use the `return` statement. This statement specifies the value that will be returned to where the function was called. The `calculateArea()` function, for example, returned the area of the rectangle:

```
function calculateArea(width, height) {
  area = width * height
  return area
}
```

What is returned depends on the code inside the function; for example, our area function will return the area of the rectangle. By contrast, if you had a form where people could enter an e-mail address to sign up for a newsletter, you might use a function to check whether that person had entered a valid e-mail address before submitting the form. In that case, the function might just return `true` or `false` values.

What happens when the value is returned depends on how the function was called. With our function to calculate area, we could display the area to the user with some more JavaScript code. If we were checking whether an e-mail address was in a valid format before subscribing that e-mail address to a newsletter, the return value would determine whether the form was submitted or not.

Conditional Statements

Conditional statements allow you to take different actions depending upon different statements. There are three types of conditional statements you will learn about here:

- `if` statements, which are used when you want the script to execute if a condition is true
- `if...else` statements, which are used when you want to execute one set of code if a condition is true and another if it is false
- `switch` statements, which are used when you want to select one block of code from many depending on a situation

if Statements

`if` statements allow code to be executed when the condition specified is met; if the condition is true then the code in the curly braces is executed. Here is the syntax for an `if` statement:

```
if (condition)
{
  code to be executed if condition is true
}
```

For example, you might want to start your homepage with the text "Good Morning" if the time is in the morning. You could achieve this using the following script (`ch11_eg06.html`):

```
<script type="text/JavaScript">
  date = new Date();
  time = date.getHours();
  if (time < 12) {
    document.write('Good Morning');
  }
</script>
```

If you are executing only one statement (as we are here), the curly braces are not strictly required, so the following would do exactly the same job (although it is good practice to include them anyway as we did previously).

```
<script type="text/JavaScript">
  date = new Date();
  time = date.getHours();
  if (time < 12)
    document.write('Good Morning');
</script>
```

This example first creates a `date` object (which you learn about later in the chapter) and then calls the `getHours()` method of the `date` object to find the time in hours (using the 24-hour clock). If the time in hours is less than 12, then the script writes `Good Morning` to the page (if it is after 12, you will see a blank page because nothing is written to it).

if . . . else Statements

When you have two possible situations and you want to react differently for each, you can use an `if...else` statement. This means: "If the conditions specified are met, run the first block of code; otherwise run the second block." The syntax is as follows:

```
if (condition)
{
  code to be executed if condition is true
}
else
{
  code to be executed if condition is false
}
```

Returning to the previous example, you can write `Good Morning` if the time is before noon, and `Good Afternoon` if it is after noon (`ch11_eg07.html`).

```
<script type="text/JavaScript">
  date = new Date();
  time = date.getHours();
  if (time < 12) {
    document.write('Good Morning');
  }
  else {
    document.write('Good Afternoon');
  }
</script>
```

As you can imagine there are a lot of possibilities for using conditional statements.

switch Statements

A `switch` statement allows you to deal with several possible results of a condition. You have a single expression, which is usually a variable. This is evaluated immediately. The value of the expression is then compared with the values for each case in the structure. If there is a match, the block of code will execute.

Here is the syntax for a `switch` statement:

```
switch (expression)
{
case option1:
  code to be executed if expression is what is written in option1
  break;
case option2:
  code to be executed if expression is what is written in option2
  break;
case option3:
  code to be executed if expression is what is written in option3
  break;
default:
  code to be executed if expression is different from option1, option2,
  and option3
}
```

You use the `break` to prevent code from running into the next case automatically. For example, you might be checking what type of animal a user has entered into a textbox, and you want to write out different things to the screen depending upon what kind of animal is in the text input. Here is a form that appears on the page. When the user has entered an animal and clicks the button, the `checkAnimal()` function contained in the head of the document is called (`ch11_eg08.html`):

```
<p>Enter the name of your favorite type of animal that stars in a
cartoon:</p>
<form name="frmAnimal">
  <input type="text" name="txtAnimal" /><br />
  <input type="button" value="Check animal" onclick="checkAnimal()" />
</form>
```

Here is the function that contains the `switch` statement:

```
function checkAnimal() {
  switch (document.frmAnimal.txtAnimal.value){
    case "rabbit":
      alert("Watch out, it's Elmer Fudd!")
      break;
    case "coyote":
      alert("No match for the road runner - meep meep!")
      break;
    case "mouse":
      alert("Watch out Jerry, here comes Tom!")
      break;
    default : alert("Are you sure you picked an animal from a cartoon?");
  }
}
```

The final option — the default — is shown if none of the cases are met. You can see what this would look like when the user has entered **rabbit** into the textbox in Figure 11-7.

Note that, should the user enter text in a different case, it will not match the options in the `switch` statement. Because JavaScript is case-sensitive, if the letter's case does not match the value of the case in the `switch` statement, it will not be a match. You can solve this by making the text all lowercase in the first place before checking it using the `toLowerCase()` method of the built-in JavaScript string object, which you meet later in the chapter.

Figure 11-7

Looping

Looping statements are used to execute the same block of code a specified number of times (which is very handy because repetitive tasks are something that computers are particularly well suited to):

- A `while` loop runs the same block of code while or until a condition is true.
- A `do while` loop runs once before the condition is checked. If the condition is true, it will continue to run until the condition is false. (The difference between a `do` and a `do while` loop is that `do while` runs once whether or not the condition is met.)
- A `for` loop runs the same block of code a specified number of times (for example, five times).

while

In a `while` loop, a code block is executed if a condition is true and for as long as that condition remains true. The syntax is as follows:

```
while (condition)
{
  code to be executed
}
```

In the following example, you can see a `while` loop that shows the multiplication table for the number 3. This works based on a counter called `i`; every time the `while` script loops, the counter increments by one (this uses the `++` arithmetic operator, as you can see from the line that says `i++`). So, the first time the script runs the counter is 1, and the loop writes out the line 1 × 3 = 3; the next time it loops around the counter is 2, so the loop writes out 2 × 3 = 6. This continues until the condition — that *i* is no longer less than 11 — is true (`ch11_eg09.html`):

```
<script type="text/JavaScript">
i = 1
while (i < 11) {
 document.write(i + " x 3 = " + (i * 3) + "<br />" );
  i ++
}
</script>
```

You can see the result of this example in Figure 11-8.

Figure 11-8

Before we go on to look at the next type of loop (a `do . . . while` loop), it is worth noting that a `while` loop may never run at all (because the condition may not be true when it is called).

do . . . while

A `do . . . while` loop executes a block of code once and then checks a condition. For as long as the condition is true, it continues to loop. So, whatever the condition, the loop runs at least once (as you can see the condition is after the instructions). Here is the syntax:

```
do
{
 code to be executed
}
while (condition)
```

For example, here is the example with the 3 times table again. The counter is set with an initial value of 12, which is higher than required in the condition, so you will see the sum 12 × 3 = 36 once, but nothing after that because when it comes to the condition, it has been met (`ch11_eg10.html`):

```
<script type="text/JavaScript">
i = 12
do {
 document.write(i + " x 3 = " + (i * 3) + "<br />" );
  i ++
}
while (i < 11)
</script>
```

Now, if you changed the value of the initial counter to 1, you would see that the script loops through the multiplication table as it did in the last example until it gets to 11.

for

The `for` statement executes a block of code a specified number of times. You use it when you want to specify how many times you want the code to be executed (rather than running while a particular condition is true/false). It is worth noting here that the number of times that the `for` loop runs could be specified by some other part of the code. First, here is the syntax (which always takes three arguments):

```
for (a; b; c)
{
 code to be executed
}
```

Now you need to look at what `a`, `b`, and `c` represent:

- `a` is evaluated before the loop is run, and is only evaluated once. It is ideal for assigning a value to a variable; for example, you might use it to set a counter to 0 using `i=0`.
- `b` should be a condition that indicates whether the loop should be run again; if it returns `true` the loop runs again. For example, you might use this to check whether the counter is less than 11.
- `c` is evaluated after the loop has run and can contain multiple expressions separated by a comma (for example, `i++, j++;`). For example, you might use it to increment the counter.

So if you come back to the 3 times table example again, it would be written something like this (`ch11_eg11.html`):

```
for (i=0; i<11; i++) {
 document.write(i + " x 3 = " + (i * 3) + "<br />" );
}
```

Let's look at the `for` statement in small chunks:

- `i=0` The counter is assigned to have a value of 0.
- `i<11` The loop should run if the value of the counter is less than 11.
- `i++` The counter is incremented by 1 every time the loop runs.

The assignment of the counter variable, the condition, and the incrementing of the counter all appear in the parentheses after the keyword `for`.

You can also assign several variables at once in the part corresponding to the letter `a` if you separate them with a comma, for example, `i = 0, j = 5;`. It is also worth noting that you can count downward with loops as well as up.

Infinite Loops and the break Statement

Note that, if you have an expression that always evaluates to `true` in any loop, you end up with something known as an *infinite loop*. These can tie up system resources and can even crash the computer, although some browsers try to detect infinite loops and then stop the loop.

You can, however, add a `break` statement to stop an infinite loop; here it is set to 100 (`ch11_eg12.html`):

```
for (i=0; /* no condition here */ ; i++) {
 document.write(i + " x 3 = " + (i * 3) + "<br />" );
  if (i == 100) {
   break;
  }
}
```

When the script gets to a `break` statement it simply stops running. This effectively prevents a loop from running too many times.

Events

All browsers are expected to support a set of events known as *intrinsic events* such as the `onload` event, which happens when a page has finished loading, `onclick` for when a user clicks on an element, and `onsubmit` for when a form is submitted. These events can be used to trigger a script.

You have already seen event handlers used as attributes on XHTML elements — such as the `onclick` attribute on an `<a>` element and the `onsubmit` attribute on the `<form>` element. The value of the attribute is the script that should be executed when the event occurs on that element (sometimes this will be a function in the `<head>` of the document).

There are two types of events that can be used to trigger scripts:

- Window events, which occur when something happens to a window. For example, a page loads or unloads (is replaced by another page or closed) or focus is being moved to or away from a window or frame.
- User events, which occur when the user interacts with elements in the page using a mouse (or other pointing device) or a keyboard, such as placing the mouse over an element, clicking on an element, or moving the mouse off an element.

For example, the `onmouseover` and `onmouseout` events can be used to change an image's `src` attribute and create a simple image rollover, as you saw earlier in the chapter:

```
<a href=""
  onmouseover="document.images.link.src='images/click_red.gif';"
  onmouseout="document.images.link.src='images/click_green.gif'">
  <img src="images/click_green.gif" width="100" height="50" border="0"
  name="link">
</a>
```

The table that follows provides a recap of the most common events you are likely to come across.

Event	Purpose	Applies To
`onload`	Document has finished loading (if used in a frameset, all frames have finished loading).	`<body>` `<frameset>`
`onunload`	Document is unloaded, or removed, from a window or frameset.	`<body>` `<frameset>`
`onclick`	Button on mouse (or other pointing device) has been clicked over the element.	Most elements
`ondblclick`	Button on mouse (or other pointing device) has been double-clicked over the element.	Most elements
`onmousedown`	Button on mouse (or other pointing device) has been depressed (but not released) over the element.	Most elements
`onmouseup`	Button on mouse (or other pointing device) has been released over the element.	Most elements
`onmouseover`	Cursor on mouse (or other pointing device) has been moved onto the element.	Most elements
`onmousemove`	Cursor on mouse (or other pointing device) has been moved while over the element.	Most elements
`onmouseout`	Cursor on mouse (or other pointing device) has been moved off the element.	Most elements

Continued

Event	Purpose	Applies To
onkeypress	A key is pressed and released.	Most elements
onkeydown	A key is held down.	Most elements
onkeyup	A key is released.	Most elements
onfocus	Element receives focus either by mouse (or other pointing device) clicking it, tabbing order giving focus to that element, or code giving focus to the element.	<a> <area> <button> <input> <label> <select> <textarea>
onblur	Element loses focus.	<a> <area> <button> <input> <label> <select> <textarea>
onsubmit	A form is submitted.	<form>
onreset	A form is reset.	<form>
onselect	User selects some text in a text field.	<input> <textarea>
onchange	A control loses input focus and its value has been changed since gaining focus.	<input> <select> <textarea>

You will see examples of these events used throughout this and the next chapter. You can also check which elements support which methods in Chapters 1 through 6 as those elements are discussed; almost every element can be associated with at least one event.

Built-in Objects

You learned about the `document` object at the beginning of the chapter and now it is time to see some of the objects that are built into the JavaScript language. You will see the methods that allow you to perform actions upon data, and properties that tell you something about the data.

String

The `string` object allows you to deal with strings of text. Before you can use a built-in object, you need to create an instance of that object. You create an instance of the `string` object by assigning it to a variable like so:

```
myString = new String('Here is some bold text')
```

The `string` object now contains the words "Here is some bold text" and this is stored in a variable called `myString`. Once you have this object in a variable, you can write the string to the document or perform actions upon it. For example, the following method writes the string as if it were in a `<b>` element:

```
document.write(myString.bold())
```

Note that if you viewed the source of this element, it would not actually have the `<b>` element in it; rather, you would see the JavaScript, so that a user who did not have JavaScript enabled would not see these words at all.

You can check the length of this string like so; the result will be the number of characters including spaces and punctuation (in this case 41):

```
MyString = new String("How many characters are in this sentence?")
alert(myString.length)
```

Before you can use the `string` object, remember you first have to create it and then give it a value.

Properties

The following table shows the main property for the `string` object and its purpose.

Property	Purpose
`length`	Returns the number of characters in a string.

Methods

The following table lists the methods for the `string` object and their purposes.

Method	Purpose
`anchor(name)`	Creates an anchor element (an `<a>` element with a `name` or `id` attribute rather than an `href` attribute).
`big()`	Displays text as if in a `<big>` element.
`bold()`	Displays text as if in a `<bold>` element.
`charAt(index)`	Returns the character at a specified position (for example, if you have a string that says "banana" and your method reads `charAt(2)`, then you will end up with the letter `n` — remember that indexes start at 0).
`fixed()`	Displays text as if in a `<tt>` element.
`fontcolor(color)`	Displays text as if in a `<font>` element with a `color` attribute.
`fontsize(fontsize)`	Displays text as if in a `<font>` element with a `size` attribute.

Continued

Method	Purpose
`indexOf(searchValue, [fromindex])`	Returns the position of the first occurrence of a specified character (or set of characters) inside another string. For example, if you have the word "banana" as your string, and you want to find the first occurrence of the letter `n` within "banana" you use `indexOf(n)`. If you supply a value for the `fromIndex` argument, the search will begin at that position. For example, you might want to start after the fourth character, in which case you could use `indexOf(n,3)`. The method returns `-1` if the string being searched for never occurs.
`italics()`	Displays text as if in an `<i>` element.
`lastIndexOf(searchValue, [fromIndex])`	Same as `indexOf()` method, but runs from right to left.
`link(targetURL)`	Creates a link in the document.
`small()`	Displays text as if in a `<small>` element.
`strike()`	Displays text as if in a `<strike>` element.
`sub()`	Displays text as if in a `<sub>` element.
`substr(start, [length])`	Returns the specified characters. 14,7 returns 7 characters, from the 14th character (starts at 0).
`substring(startPosition, endPosition)`	Returns the specified characters between the start and end index points. 7,14 returns all characters from the 7th up to but not including the 14th (starts at 0).
`sup()`	Displays text as if in a `<sup>` element.
`toLowerCase()`	Converts a string to lowercase.
`toUpperCase()`	Converts a string to uppercase.

Try It Out — Using the String Object

In this example, you see a subsection of a string collected and turned into all uppercase letters. The full string (at the beginning of the example) will hold the words "Learning about Built-in Objects is easy"; then the code just extracts the words "Built-in Objects" from the string, and finally it turns the selected part of the string into uppercase characters.

1. Create a skeleton XHTML document, like so:

```
<?xml version="1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:land="en">
<head>
  <title>String Object</title>
</head>
<body>
</body>
</html>
```

2. Because the code in this example is going to be run in only one place, the script can be added inside the body of the document, so add the `<script>` element and inside it write the following code:

```
<script type="text/JavaScript">
  myString = new String('Learning about Built-in Objects is easy')
  myString = myString.substring(15, 31)
  myString = myString.toUpperCase()
  document.write(myString)
</script>
```

   Let's look at this a little closer. First you have to create an instance of the `string` object, which is assigned to the variable `myString`:

```
myString = new String('Learning about Built-in Objects is easy')
```

   As it has been created, the string object has been made to hold the words `Learning about Built-in Objects is easy`. But, the idea of this exercise is just to select the words "Built-in Objects" so you use the `substring()` method to extract those words. The syntax is as follows:

```
substring(startPosition, endPosition)
```

   So you select the `string` object (which is in the variable `myString`) and make its value the new substring you want (this is reassigning the value of the variable with the substring you want):

```
myString = myString.substring(15, 32)
```

   This selects the string from the 16th character to the 33rd character — because it starts at position 0.

   Next, you must convert the string to uppercase using the `toUpperCase()` method:

```
myString = myString.toUpperCase()
```

   And finally you can write it to the document like so:

```
document.write(myString)
```

3. Save this file as `ch11_eg14.html` and when you open it in the browser, you should see the text shown in Figure 11-9.

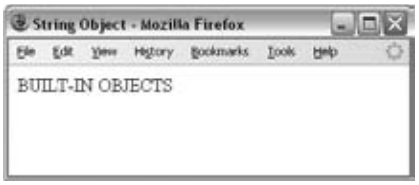

Figure 11-9

The result looks quite simple, but when you consider that the original string was `Learning about Built-in Objects is easy`, it now looks substantially different.

Date

The `date` object helps you work with dates and times. You create a new `date` object using the date constructor like so:

```
new Date()
```

You can create a `date` object set to a specific date or time, in which case you need to pass it one of four parameters:

- `milliseconds`: This value should be the number of milliseconds since 01/01/1970.
- `dateString`: Can be any date in a format recognized by the `parse()` method.
- `yr_num`, `mo_num`, `day_num`: Represents year, month, and day.
- `yr_num`, `mo_num`, `day_num`, `hr_num`, `min_num`, `seconds_num`, `ms_num`: Represents the years, days, hours, minutes, seconds, and milliseconds.

Here are some examples; the first uses milliseconds and will read `Thu Nov 27 05:33:20 UTC 1975`:

```
var birthDate = new Date(8298400000)
document.write(birthDate)
```

The second uses a `dateString`, and will read `Wed Apr 16 00:00:00 UTC+0100 1975`:

```
var birthDate = new Date("April 16, 1975")
document.write(birthDate)
```

The third uses `yr_num`, `mo_num`, and `day_num`, and will read `Mon May 12 00:00:00 UTC+0100 1975`:

```
var birthDate = new Date(1975, 4, 28)
document.write(birthDate)
```

There are a few things to watch out for:

- The first confusing thing you might notice here is that the number 4 corresponds to the month of May! That makes January 0. Similarly, when working with days, Sunday is treated as 0.
- You might find that you get different time zones than I do. I am based in London, so I run on Greenwich Mean Time (GMT) or Coordinated Universal Time (UTC). All the date object's workings are performed using UTC, even though your computer may display a time that is consistent with your time zone.
- While you can add or subtract dates, your result will end up in milliseconds. For example, if I wanted to find out the number of days until the end of the year, I might use something like this:

```
var today = new Date()
var newYear = new Date(2010,11,31)
var daysRemaining = (newYear - today)
document.write(daysRemaining)
```

The problem with this is that you end up with a result that is very long (plus if you read this during 2010 or minus if you read it after 2010). With 86,400,000 milliseconds in each day, you are likely to see a very large figure.

So, you need to divide the `daysRemaining` by the number of milliseconds in the day (`86400000`) to find the number of days (`ch11_eg15.html`):

```
var today = new Date()
var newYear = new Date(2010,11,31)
var daysRemaining = (newYear - today)
daysRemaining = daysRemaining/86400000
document.write(daysRemaining)
```

When you use the `date` object, you need to bear in mind that a user's computer click may well be inaccurate and the fact that different users could be in various time zones.

The following table shows some commonly used methods of the `date` object.

Method	Purpose
`date()`	Returns a `Date` object
`getDate()`	Returns the date of a `Date` object (from 1 to 31)
`getDay()`	Returns the day of a `Date` object (from 0 to 6; 0=Sunday, 1=Monday, and so on)
`getMonth()`	Returns the month of a `Date` object (from 0 to 11; 0=January, 1=February, and so on)
`getFullYear()`	Returns the year of a `Date` object (four digits)
`getYear()`	Returns the year of a `Date` object using only two digits (from 0 to 99)

Continued

Method	Purpose
`getHours()`	Returns the hours of a `Date` object (from 0 to 23)
`getMinutes()`	Returns the minutes of a `Date` object (from 0 to 59)
`getSeconds()`	Returns the seconds of a `Date` object (from 0 to 59)
`getTime()`	Returns the number of milliseconds since midnight 1/1/1970
`getTimezoneOffset()`	Returns the time difference between the user's computer and GMT
`parse()`	Returns a string date value that holds the number of milliseconds since January 01 1970 00:00:00
`setDate()`	Sets the date of the month in the `Date` object (from 1 to 31)
`setFullYear()`	Sets the year in the `Date` object (four digits)
`setHours()`	Sets the hours in the `Date` object (from 0 to 23)
`setMinutes()`	Sets the minutes in the `Date` object (from 0 to 59)
`setMonth()`	Sets the months in the `Date` object (from 0 to 11; 0=January, 1=February)
`setSeconds()`	Sets the seconds in the `Date` object (from 0 to 59)
`setTime()`	Sets the milliseconds after 1/1/1970
`setYear()`	Sets the year in the `Date` object (00 to 99)
`toGMTString()`	Converts the `Date` object to a string, set to GMT time zone
`toLocaleString()`	Converts the `Date` object to a string, set to the current time zone of the user
`toString()`	Converts the `Date` object to a string

Many of the methods in the table that follows were then added offering support for the universal (UTC) time, which takes the format `Day Month Date, hh,mm,ss UTC Year`.

Method	Purpose
`getUTCDate()`	Returns the date of a `Date` object in universal (UTC) time
`getUTCDay()`	Returns the day of a `Date` object in universal time
`getUTCMonth()`	Returns the month of a `Date` object in universal time
`getUTCFullYear()`	Returns the four-digit year of a `Date` object in universal time
`getUTCHours()`	Returns the hour of a `Date` object in universal time

Method	Purpose
`getUTCMinutes()`	Returns the minutes of a `Date` object in universal time
`getUTCSeconds()`	Returns the seconds of a `Date` object in universal time
`getUTCMilliseconds()`	Returns the milliseconds of a `Date` object in universal time
`setUTCDate()`	Sets the date in the `Date` object in universal time (from 1 to 31)
`setUTCDay()`	Sets the day in the `Date` object in universal time (from 0 to 6; Sunday=0, Monday=1, and so on)
`setUTCMonth()`	Sets the month in the `Date` object in universal time (from 0 to 11; 0=January, 1=February)
`setUTCFullYear()`	Sets the year in the `Date` object in universal time (four digits)
`setUTCHour()`	Sets the hour in the `Date` object in universal time (from 0 to 23)
`setUTCMinutes()`	Sets the minutes in the `Date` object in universal time (from 0 to 59)
`setUTCSeconds()`	Sets the seconds in the `Date` object in universal time (from 0 to 59)
`setUTCMilliseconds()`	Sets the milliseconds in the `Date` object in universal time (from 0 to 999)

Math

The `math` object helps in working with numbers. It has properties for mathematical constants and methods representing mathematical functions such as the Tangent and Sine functions.

For example, the following sets a variable called `numberPI` to hold the constant of pi and then write it to the screen (`ch11_eg16.html`):

```
numberPI = Math.PI
document.write (numberPI)
```

The following example rounds pi to the nearest whole number (integer) and writes it to the screen (also shown in `ch11_eg16.html`):

```
numberPI = Math.PI
numberPI = Math.round(numberPI)
document.write (numberPI)
```

Properties

The following table lists the properties of the `math` object.

Property	Purpose
`E`	Returns the base of a natural logarithm
`LN2`	Returns the natural logarithm of 2
`LN10`	Returns the natural logarithm of 10
`LOG2E`	Returns the base-2 logarithm of E
`LOG10E`	Returns the base-10 logarithm of E
`PI`	Returns pi
`SQRT1_2`	Returns 1 divided by the square root of 2
`SQRT2`	Returns the square root of 2

Methods

The following table lists the methods for the `math` object.

Method	Purpose
`abs(x)`	Returns the absolute value of *x*
`acos(x)`	Returns the arccosine of *x*
`asin(x)`	Returns the arcsine of *x*
`atan(x)`	Returns the arctangent of *x*
`atan2(y,x)`	Returns the angle from the x-axis to a point
`ceil(x)`	Returns the nearest integer greater than or equal to *x*
`cos(x)`	Returns the cosine of *x*
`exp(x)`	Returns the value of E raised to the power of *x*
`floor(x)`	Returns the nearest integer less than or equal to *x*
`log(x)`	Returns the natural log of *x*
`max(x,y)`	Returns the number with the highest value of *x* and *y*
`min(x,y)`	Returns the number with the lowest value of *x* and *y*
`pow(x,y)`	Returns the value of the number *x* raised to the power of *y*
`random()`	Returns a random number between 0 and 1

Method	Purpose
`round(x)`	Rounds *x* to the nearest integer
`sin(x)`	Returns the sine of *x*
`sqrt(x)`	Returns the square root of *x*
`tan(x)`	Returns the tangent of *x*

Array

An *array* is like a special variable. It's special because it can hold more than one value, and these values can be accessed individually. Arrays are particularly helpful when you want to store a group of values in the same variable rather than having separate variables for each value. You may want to do this because all the values correspond to one particular item, or just for the convenience of having several values in the same variable rather than in differently named variables; or it might be because you do not know how many items of information are going to be stored (for example, you might store the items that would appear in a shopping basket in an array). You often see arrays used in conjunction with loops, where the loop is used to add information into an array or read it from the array.

You need to use a *constructor* with an `array` object, so you can create an array by specifying either the name of the array and how many values it will hold or by adding all the data straight into the array. For example, here is an array that is created with three items; it holds the names of musical instruments:

```
instruments = new Array("guitar", "drums", "piano")
```

The items in the array can be referred to by a number that reflects the order in which they are stored in the array. The number is an index, so it begins at 0. For example, you can refer to the guitar as `instruments[0]`, the drums as `instruments[1]`, and so on.

An array does need to know how many items you want to store in it, but you do not need to provide values for each item in the array when it is created; you can just indicate how many items you want to be able to store (to confuse matters, this value does not start at 0 so it will create three elements not four):

```
instruments = new Array(3)
```

This number is stored in the `length` property of the `array` object and the contents are not actually assigned yet. If you want to increase the size of an array, you can just assign a new value to the `length` property that is higher than the current length.

Here is an example that creates an array with five items and then checks how many items are in the array using the `length` property:

```
fruit = new Array("apple", "banana", "orange", "mango", "lemon")
document.write(fruit.length)
```

Here is an example of the `toString()` method, which converts the array to a string.

```
document.write('These are ' + fruit.toString())
```

Keeping the related information in the one variable tends to be easier than having five variables, such as `fruit1`, `fruit2`, `fruit3`, `fruit4`, and `fruit5`. Using one array like this also takes up less memory than storing five separate variables, and in situations when you might have varying numbers of fruit it allows the variable to grow and shrink in accordance with your requirements (rather than creating ten variables, half of which might be empty).

Methods

The table that follows lists the methods of an array:

Method	Purpose
`concat()`	Joins (or concatenates) two or more arrays to create one new one
`join(separator)`	Joins all of the elements of an array together separated by the character specified as a separator (the default is a comma)
`reverse()`	Returns the array with items in reverse order
`slice()`	Returns a selected part of the array (if you do not need it all)
`sort()`	Returns a sorted array, sorted by alphabetical or numerical order

Window

Every browser window and frame has a corresponding `window` object that is created with every instance of a `<body>` or `<frameset>` element.

For example, you can change the text that appears in the browser's status bar using the `status` property of the `window` object. To do this, first you need to add a function in the head that is going to be triggered when the page loads, and then you use this function to indicate what should appear in the status bar:

```
<script type="text/javascript">
  function statusBarText()
  {
    window.status = "Did you see me down here?"
  }
</script>
```

You then call this function from the `<body>` element's `onload` event, like so:

```
<body onload="statusBarText()">
```

Properties

The table that follows lists the properties of the `window` object.

Property	Purpose
`closed`	A Boolean determining if a window has been closed. If it has, the value returned is `true`.
`defaultStatus`	Defines the default message displayed in a browser window's status bar (usually at the bottom of the page on the left).
`document`	The document object contained in that window.
`frames`	An array containing references to all named child frames in the current window.
`history`	A history object that contains details and URLs visited from that window (mainly for use in creating back and forward buttons like those in the browser).
`location`	The location object; the URL of the current window.
`name`	The window's name.
`status`	Can be set at any time to define a temporary message displayed in the status bar; for example, you could change the message in the status bar when a user hovers over a link by using it with an `onmouseover` event on that link.
`statusbar`	The status bar has a property that indicates whether the status bar is visible. The value is a Boolean `true` or `false` — for example, `window.statusbar[.visible=false]`.
`toolbar`	The toolbar has a property that indicates whether the scrollbar is visible or not. Its value is a Boolean `true` or `false` — for example, `window.toolbar[.visible=false]`. This can be set only when you create the new window.
`top`	A reference for the topmost browser window if several windows are open on the desktop.
`window`	The current window or frame.

Methods

The table that follows lists the methods of the `window` object.

Method	Purpose
`alert()`	Displays an alert box containing a message and an OK button.
`back()`	Same effect as the browser's Back button.
`blur()`	Removes focus from the current window.

Continued

Method	Purpose
`close()`	Closes the current window or another window if a reference to another window is supplied.
`confirm()`	Brings up a dialog box asking users to confirm that they want to perform an action with either OK or Cancel as the options. They return `true` and `false`, respectively.
`focus()`	Gives focus to the specified window and brings it to the top of others.
`forward()`	Equivalent to clicking the browser's Forward button.
`home()`	Takes users to their browser's designated homepage.
`moveBy(horizontalPixels, verticalPixels)`	Moves the window by the specified number of pixels in relation to current coordinates.
`moveTo(Xpostion, Yposition)`	Moves the top left of the window to the specified x-y coordinates.
`open(URL, name [,features])`	Opens a new browser window (this method is covered in more detail in the next chapter).
`print()`	Prints the content of the current window (or brings up the browser's print dialog).
`prompt()`	Creates a dialog box for the user to enter an input.
`stop()`	Same effect as clicking the Stop button in the browser.

Writing JavaScript

You need to be aware of a few points when you start writing JavaScript:

- JavaScript is case-sensitive, so a variable called `myVariable` is different than a variable called `MYVARIABLE`, and both are different than a variable called `myvariable`.
- When you come across symbols such as `(`, `{`, `[`, `˝`, and `` ` `` they must have a closing symbol to match: `´`, `˝`, `]`, `}`, and `)`. (Note how the first bracket opened is the last one to be closed, which is why the closing symbols are in reverse order here.)
- Like XHTML, JavaScript ignores extra spaces, so you can add white space to your script to make it more readable. The following two lines are equivalent, even though there are more spaces in the second line:

```
myVariable="some value"
myVariable = "some value"
```

- If you have a large string, you can break it up with a backslash, as you can see here:

```
document.write("My first \
  JavaScript example")
```

- But you must not break anything other than strings, so this would be wrong:

```
document.write \
  ("My first JavaScript example")
```

- You can insert special characters such as `"`, `'`, `;`, and `&`, which are otherwise reserved (because they have a special meaning in JavaScript), by using a backslash before them like so:

```
document.write("I want to use a \"quote\" mark \& an ampersand.")
```

  This writes out the following line to the browser:

```
I want to use a "quote" mark & an ampersand.
```

- If you have ever used a full programming language such as C++ or Java, you know they require a semicolon at the end of each line. This is optional in JavaScript unless you want to put more than one statement on a line.

A Word About Data Types

By now you should be getting the idea that you can do different things with different types of data. For example, you can add numbers together but you cannot mathematically add the letter *A* to the letter *B*. Some forms of data require that you are able to deal with numbers that have decimal places (floating point numbers); currency is a common example. Other types of data have inherent limitations; for example, if I am dealing with dates and time, I want to be able to add hours to certain types of data without getting 25:30 as a time (even though I often wish I could add more hours to a day).

Different types of data (letters, whole numbers, decimal numbers, dates) are known to have different *data types*; these allow programs to manage the different types of data in different ways. For example, if you use the + operator with a string, it concatenates two strings, whereas if it is used with numbers, it adds the two numbers together. Some programming languages require that you specifically indicate what type of data a variable is going to hold and require you to be able to convert between types. While JavaScript supports different data types, as you are about to see, it handles conversion between types itself, so you never need to worry about telling JavaScript that a certain type of data is a date or a *string* (a string being a set of characters that may include letters and numbers).

There are three simple data types in JavaScript:

- **Number:** Used to perform arithmetic operations (addition, subtraction, multiplication, and division). Any whole number or decimal number that does not appear between quotation marks is considered a number.
- **String:** Used to handle text. It is a set of characters (including numbers, spaces, and punctuation) enclosed by quotation marks.
- **Boolean:** A Boolean value has only two possible values: `true` and `false`. This data allows you to perform logical operations and check whether something is true or false.

You may also come across two other data types:

- **Null:** Indicates that a value does not exist. This is written using the keyword `null`. This is an important value because it explicitly states that no value has been given. This can mean a very different thing from a string that just contains a space or a zero.
- **Undefined:** Indicates a situation where the value has not been defined previously in code and uses the JavaScript keyword `undefined`. You might remember that if you declare a variable but do not give it a value, the variable is said to be undefined (you are particularly likely to see this when something is not right in your code).

Keywords

You may have noticed that there are several keywords in JavaScript that perform functions, such as `break`, `for`, `if`, and `while`, all of which have special meaning; therefore, these words should not be used in variable, function, method, or object names. The following is a list of the keywords that you should avoid using (some of these are not actually used yet, but are reserved for future use):

```
abstract boolean break byte case catch char class const
continue default do double else extends false final
finally float for function goto if implements import
in instanceof int interface long native new null
package private protected public return short static
super switch synchronized this throw throws transient
true try var void while with
```

Summary

This chapter has introduced you to a lot of new concepts: objects, methods, properties, events, arrays, functions, interfaces, object models, data types, and keywords. While it's a lot to take in all at once, by the time you have looked at some of the examples in the next chapter it should be a lot clearer. After reading that chapter, you can read through this chapter again and you should be able to understand more examples of what can be achieved with JavaScript.

You started off by looking at how you can access information from a document using the Document Object Model (this chapter focused on the Level 0 DOM).

Once you have figured out how to get information from a document, you can use JavaScript to perform calculations upon the data in the document. JavaScript mainly performs calculations using features such as the following:

- Variables (which store information in memory)
- Operators (such as arithmetic and comparison operators)
- Functions (which live in the `<head>` of a document and contain code that is called by an event)
- Conditional statements (to handle choices of actions based on different circumstances)
- Loops (to repeat statements until a condition has been met)

As you will see in Chapter 12, these simple concepts can be brought together to create quite powerful results. In particular, when you see some of the validation scripts that will check the form data users enter, you will see some quite advanced JavaScript, and you will have a good idea of how basic building blocks can create complex structures.

Finally, you looked at a number of other objects made available through JavaScript; you met the `string`, `date`, `math`, `array`, and `window` objects. Each object contains related functionality; each has properties that tell you about the object (such as the date, the time, the size of window, or length of string), and methods that allow you to do things with this data stored in the object.

I hope you are starting to get a grasp of how JavaScript can help you add interactivity to your pages, but you will really get to see how it does this in the next chapter when you delve into examples of JavaScript libraries and look at examples that will really help you make use of JavaScript.

Exercises

1. Create a script to write out the multiplication table for the number 5 from 1 to 20 using a `while` loop.
2. Modify `ch11_eg06.html` so that it can say one of three things:
   - "Good Morning" to visitors coming to the page before 12 P.M. (using an `if` statement).
   - "Good Afternoon" to visitors coming to the page between 12 and 6 P.M. — again using an `if` statement. (Hint: You might need to use a logical operator.)
   - "Good Evening" to visitors coming to the page after 6 P.M. up until midnight (again using an `if` statement).

12

Working with JavaScript

You learned the key concepts behind the JavaScript language in Chapter 11; in this chapter, you see how these concepts come together in working scripts. By looking at several examples, you learn different ways in which JavaScript can interact with your web pages, some helpful coding practices for writing your own JavaScripts, and some shortcuts to creating interactive pages. The chapter is roughly split into two sections:

- **Creating your own basic scripts:** The first section focuses on how to write your own basic scripts. Most of these examples work with form elements.
- **Using pre-written JavaScript libraries:** The second section focuses on a number of scripts that have already been written and shows you how you can add powerful and complex features to your site with just a few lines of code.

By the end of the chapter, not only will you have learned a lot about using JavaScript in your pages, but you will also have seen many helpful tools and techniques you can use in your own pages.

Practical Tips for Writing Scripts

Before you start looking at the examples, I'd like to share a few practical hints on developing JavaScripts that should save you time.

Has Someone Already Written This Script?

There are thousands of free JavaScripts already on the Web, so before you start writing a script from scratch, it is worth searching to see if someone has already done all the hard work for you. Here are a couple of sites that will help you get going (and don't forget you can search using a search engine such as Google, too):

- `www.HotScripts.com`
- `www.JavaScriptKit.com`

- `www.webreference.com/programming/javascript/`
- `http://JavaScript.Internet.com`

Of course, for some tasks you will have to write your own script, but you may still find that someone has written a script that does something similar (and could learn something just by looking at how they approached the problem).

You will see more about this topic near the end of the chapter when you look at using existing JavaScript libraries.

Reusable Functions

Along with reusing other people's scripts and folders, you can also write code that you can reuse yourself. For example, you might build several sites that use a similar form that allows people to contact the site owners. On each contact form there might be several fields that are required, and you might decide to write a script to ensure that people fill in the required fields. Rather than writing a new script for each site, you can create a script that you can use on any contact form you write.

So, you should aim to make your code as reusable as possible, rather than writing a script that will only work with one page. You will see examples of this shortly.

Using External JavaScript Files

Whenever you are going to use a script in more than one page it's a good idea to place it in an external JavaScript file (a technique you learned about at the beginning of Chapter 11). For example, if you wanted to create a newsletter signup form on each page of your site, then you might use a script to check that the text entered into the e-mail address box is in a valid e-mail format. Rather than including this script on every page, if the script lives in an external JavaScript file:

- You do not have to copy and paste the same code into several files.
- The file size of the pages is smaller because the JavaScript is in one file that is included on each page rather than repeated in multiple pages.
- If you need to change something about the script, you need to change only the one script, not every page that uses it.

Place Scripts in a Scripts Folder

When you use external scripts you should create a special `scripts` folder — just as you would an `images` folder. This helps improve the organization of your site and your directory structure. Whenever you need to look at or change a script, you know exactly where it will be.

You should also use intuitive names for your script files so that you can find them quickly and easily.

Form Validation

Form validation is one of the most common tasks performed using JavaScript. You have likely come across forms on the Web that have shown you a prompt when you have not entered a value into a field that requires one, or when you have entered the wrong kind of value; this is because the form has been *validated*. That is, a script has checked to see whether the text you have entered or choices you have made match some rules that the programmer has written into the page. For example, if you are expected to enter an e-mail address, these validation rules may check what you entered to ensure that it contains an `@` symbol and at least one period or full stop. These kinds of rules help ensure that the data provided by users meets the requirements of the application before being submitted.

When to Validate

Validation can happen in two places: in the browser using JavaScript, and on the server using one of several languages such as ASP.NET or PHP. In fact, applications that collect important information using a form (such as e-commerce orders) are usually validated both in the browser *and* on the server. You may wonder why forms are validated in the browser if they will only get checked again when they reach the server; the reason is that it helps the user enter the correct data required for the job without the form being sent to the server, being processed, and then being sent back again if there are any errors. This has two key advantages:

- It's quicker for the user because the form does not need to be sent to the server, processed, and returned to the user with any relevant error messages.
- It saves the load on the server because some errors will get caught before the form is submitted.

It is very important to validate on the server because you cannot guarantee that the user has JavaScript enabled in his or her browser, and if a user entered a wrong value into a database or other program it could prevent the entire application from running properly. (It is also possible for hackers to bypass JavaScript if they are intending to send some incorrect information.)

What You Can Check For

When it comes to validating a form you cannot always check whether users have given you the correct information, but you can check whether they have given you some information in the correct format. For example, you cannot ensure that the user has entered his or her correct phone number; the user could be entering anyone's phone number, but you can check that it's a number rather than letters or other characters, and you can check that the number contains a minimum number of digits. As another example, you can't ensure someone has entered a real e-mail address rather than a false address, but you can check that whatever was entered followed the general structure of an e-mail address (including an `@` sign and a period, and that it is at least seven characters long). So JavaScript form validation is a case of minimizing the possibility of user errors by validating form controls.

When it comes to form controls that allow users to indicate their choice from a selection of options (such as checkboxes, drop-down select boxes, and radio buttons), you can use JavaScript to check that a user has selected one of the options (for example, to check that a user has checked the terms and conditions).

How to Check a Form

There are several ways in which you can check a form. Usually when the user presses the submit button on a form, it triggers the `onsubmit` event handler on the `<form>` element, which in turn calls a validation function stored either in a separate script or in the head of the document. The function must then return `true` in order for the form to be sent, or, if an error is encountered, the function returns `false` and the user's form will not be sent — at which point the form should indicate to the user where there is a problem with the information the user entered.

> *If you use a validation function that is called by the* `onsubmit` *event handler, but the user's browser does not support JavaScript, then the form will still be submitted without the validation checks taking place.*

In the validation functions you meet in this chapter, the first task will be to set a variable that can be returned to say whether the script found errors or not. At first, this is set to `true` (indicating that the form can be sent because problems were found); then as the script checks the values the user has entered, if the function finds an error this value can be turned to `false` to prevent the form from being submitted.

Some forms also check values as the user moves between form fields — in which case the values the user entered are passed to a function that checks that specific form control using the `onblur` event (which fires when that form control loses focus).

Checking Text Fields

You have probably seen forms on web sites that ask you to provide a username and password, and then to re-enter the password to make sure you did not mistype something. It might resemble Figure 12-1.

Figure 12-1

Let's take a look at the code for this form, and how it calls the JavaScript that will validate what the user has entered (`ch12_eg01.html`).

```
<form name="frmRegister" method="post" action="register.aspx"
  onsubmit="return validate(this);">
  <div>
    <label for="txtUserName">Username:</label>
```

```
      <input type="text" name="txtUserName" id="txtUserName" size="12" />
    </div>
    <div>
      <label for="txtPassword">Password: </label>
      <input type="password" name="txtPassword" id="txtPassword" size="12" />
    </div>
    <div>
      <label for="txtPassword2">Confirm your password:</label>
      <input type="password" name="txtPassword2" id="txtPassword2" size="12"/>
    </div>
    <div>
      <input type="submit" value="Log in" />
    </div>
  </form>
```

The opening `<form>` tag has an `onsubmit` attribute; when the user presses the submit button on the form, the script specified in this attribute will be run.

```
<form name="frmRegister" method="post" action="register.aspx"
  onsubmit="return validate(this);">
```

In this case, when the user presses the submit button, a function called `validate()` will run. Before the name of the function is the keyword `return`; this indicates that the `validate()` function will return a value of `true` or `false` (in order for the form to be submitted, the function must return `true`; if it returns `false` the form is not submitted).

The `validate()` function we are writing will take one parameter; it tells the function the form that you want to process. So, inside the parentheses of the `validate()` function, you can see the word `this`, which indicates that *this* is the form you wish to validate (as opposed to any other form that might appear on the page).

With a login form like this one, you might want to check a few things:

- That the username is of a minimum length
- That the password is of a minimum length
- That the two passwords match

In this section, you are going to look at two different ways to approach the `validate()` function:

- Creating a single function to check the form
- Creating re-usable functions that are called by this function

In both cases, the `validate()` function will live in the `<head>` element, and will start by setting a variable called `returnValue` to `true`; if no errors are found this will be the value that the function returns, which will in turn allow the form to be sent. If an error is met, the variable will be set to `false`, and the form will not send.

Single Function Approach

The first approach we will look at to validate this form is to write a validation script especially for this one form, where all of the rules live inside the one function (`ch12_eg01.html`).

The function will be called `validate()` and it will expect to be told which form it is working with as a parameter of the function:

```
function validate(form) {
```

This is why the `onsubmit` attribute of the `<form>` element used the keyword `this` when calling the function, indicating that *this* is the form it wanted to process.

In the script, you want to collect the values that the user entered into the text controls and store these in variables. You identify the form controls within the page one by one, using the dot notation you met in the last chapter. You then collect the value the user typed into that control using the `value` property:

```
function validate(form) {
  var returnValue = true;
  var username = form.txtUserName.value;
  var password1 = form.txtPassword.value;
  var password2 = form.txtPassword2.value;
```

Because we have told the function which form we are working with, the dot notation can start with `form` rather than `document.frmRegister`; the following two lines would do exactly the same thing:

```
var username = document.frmRegister.txtUserName.value;
var username = form.txtUserName.value;
```

Now you can start to check whether the data that was entered by the user meets the criteria you require for the form to be submitted. First you can check whether the username is at least six characters long. The value the user entered into the username form control is already stored in a variable called `username`, and because it is a string (a set of letters or numbers), you can use the `length` property to tell how long it is.

If the username is not long enough, you will need to take action, so you place your validation test in an `if` statement. The following states that if the `length` of the variable is less than six characters long, the code in curly braces will be run:

```
if (username.length < 6) {
  username is less than 6 characters so do something
}
```

In this case, if it is less than six characters long, you will do three things:

- Set the variable `returnValue` to `false`, so that the form will not be submitted.
- Tell the user what has happened so that he or she can correct the error. For this example, we will use the JavaScript `alert()` function to create an alert box (like the ones you met in the last chapter) containing a message for the user.
- Pass focus back to this item (using the JavaScript `focus()` method) on the form so that the user can change what he or she had put in this form.

Here you can see the `if` statement with all of these actions:

```
if(username.length < 6) {
  returnValue = false;
  alert("Your username must be at least\n6 characters long.\n
  Please try again.");
  frmRegister.txtUserName.focus();
}
```

The alert box used to be a very popular way to show users errors on their forms (as it is a very simple technique of providing feedback). These days, it is more popular to write an error message into the page itself; however, this is more complicated, so you will see how to do that later in the chapter.

You may have noticed `\n` *in the middle of the error message; this creates a line break in JavaScript.*

Next you want to check the length of the first password. To do this, you can use the same approach. But if the password is not long enough you will empty both password controls, and give focus to the first password box:

```
if (password1.length < 6) {
  returnValue = false;
  alert("Your password must be at least\n6 characters long.\n
    Please try again.");
  frmRegister.txtPassword.value = "";
  frmRegister.txtPassword2.value = "";
  frmRegister.txtPassword.focus();
}
```

If the code has gotten this far, the username and first password are both long enough. Now, you just have to check whether the value of the first password box is the same as the second one, as shown here. Remember that the `!=` operator used in this condition means "not equal," so the `if` statement will take action if the value of `password1` does not equal that of `password2`.

```
if (password1.value != password2.value) {
  returnValue = false;
  alter("Your password entries did not match.\nPlease try again.");
  frmRegister.txtPassword.value = "";
  frmRegister.txtPassword2.value = "";
  frmRegister.txtPassword.focus();
}
```

You can see here that when the user has entered passwords that do not match, the user is shown an alert box with an error message reporting that the password entries did not match. Also, the contents of both password inputs are cleared and the focus is passed back to the first password box.

Because a password input will show dots or asterisks rather than the characters, when a user makes a mistake with a password input, he or she will not be able to see where the mistake is. This is why I clear password boxes when a user has made a mistake in them.

The only thing left to do in this function is return the value of the `returnValue` variable — which will be `true` if all the conditions are met or `false` if not.

```
return returnValue;
}
```

Here is the function in its entirety (`ch12_eg01.html`):

```
function validate(form) {

  var returnValue = true;

  var username = form.txtUserName.value;
  var password1 = form.txtPassword.value;
  var password2 = form.txtPassword2.value;

if(username.length < 6) {
  returnValue = false;
  alert("Your username must be at least\n6 characters long.\n
  Please try again.");
  document.frmRegister.txtUserName.focus();
}

if (password1.length < 6) {
  returnValue = false;
  alert("Your password must be at least\n6 characters long.\n
  Please try again.");
  document.frmRegister.txtPassword.value = "";
  document.frmRegister.txtPassword2.value = "";
  document.frmRegister.txtPassword.focus();
}

if (password1 != password2) {
  returnValue = false;
  alert("Your password entries did not match.\nPlease try again.");
  document.frmRegister.txtPassword.value = "";
  document.frmRegister.txtPassword2.value = "";
  document.frmRegister.txtPassword.focus();
}
  return returnValue;
}
```

In Figure 12-2 you can see the result if the user's password is not long enough.

Figure 12-2

This example should have given you a good idea of how to check what a user has entered into a form against a set of rules.

Re-Usable Functions Approach

While the example you just saw works fine, you can save time and effort by writing code that you can re-use. For example, many programmers have a single JavaScript file that will contain a set of form validation functions they can use in any form that they write. These functions can check for things like the following:

- Whether the user has entered text that is longer than the minimum required length as demonstrated with the username and password in the last example
- Whether the text entered by the user consists only of numbers (no other characters) — which could be handy for telephone numbers or for asking customers the quantity of goods they want in an e-commerce store
- Whether, when you are requesting an e-mail address, the user has entered that address in the correct form

JavaScript files that contain functions that can be re-used are often known as JavaScript libraries. Let's look at the same form again, but develop a validation approach that utilizes this approach.

Using the second approach, we still have to write a `validate()` function for each form, but the function is much shorter than the last one you saw. Its job is to pass values from the form to functions in the

JavaScript validation library that does the real validation work. So, this time our `validate()` function will call two other functions in the JavaScript library:

- `validateConfirmPassword()` will check that the two password fields match.
- `validateMinimumLength()` will ensure that the user has entered a minimum number of characters for the form field.

You will look at these functions in a moment, but first let's look at the `validate()` function that calls them (which lives in the same page as the form); its job is to pass values from the form to the library functions. If there is an error, it will then prevent the submission of the form, and will tell the user the error message. Before you look at each line of the function individually, here is the entire function:

```
function validate(form) {
  var returnValue = "";
  returnValue += validateConfirmPassword(form.txtPassword,
      form.txtPassword2,
      'Your passwords did not match');
  returnValue += validateMinimumLength(form.txtPassword, 6,
      'Your password must be at least 6 characters long');
  returnValue += validateMinimumLength(form.txtUserName, 6,
      'Your username must be at least 6 characters long');
  if (returnValue != "") {
    return false;
  }
  return true;
}
```

Let's take a closer look at this; it starts with the function name. The function takes one parameter, the form it has to validate, which is given in parentheses.

```
function validate(form) {
```

Then a variable called `returnValue` is declared; this time, rather than being set to `true`, it is set to an empty string using empty quotes.

```
var returnValue = "";
```

You then make three calls to functions, each of which checks a different aspect of the form. We'll look at each of these in turn in a moment. If these functions find a problem (because the user has not entered what you wanted), the functions will return an error. You may remember from the last chapter that `+=` adds a value onto an existing variable, so if the function returns an error the error message is appended to the variable `returnValue`; if there is not an error, nothing will be added to the variable.

First you can see we're calling the `validateConfirmPassword()` function. We tell it the two password controls that we want it to check using the dot notation, along with the error message we want to display if the fields do not match.

```
returnValue += validateConfirmPassword(form.txtPassword, form.txtPassword2,
      'Your passwords did not match');
```

Because the `validateConfirmPassword()` function will return the error message if the passwords do not match, we will know if there is a problem because the `returnValue` attribute will contain this error message; it will no longer be empty.

We then call the `validateMinimumLength()` function to check the length of the first password control. Here we are passing in the form control to check, the minimum number of characters the user can enter for that form control, and the error message to show if the form control is not that length.

```
returnValue += validateMinimumLength(form.txtPassword, 6,
    'Your password must be at least 6 characters long');
```

We then check that the username entered is more than six characters. To do this we call the `validateMinimumLength()` function a second time. Here you can already start to see the benefits of code re-use; rather than repeating the entire function again we are using the same function we just used a second time, but this time we are telling it to check a different form control and display a different error message if there is a problem.

```
returnValue += validateMinimumLength(form.txtUserName, 6,
    'Your username must be at least 6 characters long');
```

Having checked these three aspects of the form, if there is a problem `returnValue` will no longer be empty. The following `if` statement says if `returnValue` is not empty, the `validate()` function should return `false` to say that the form should not be submitted. Otherwise, if it is blank, we can return `true`, which will allow the form to be submitted.

```
if (returnValue != "") {
  return false;
}
return true;
```

You might notice that we start with the rules that apply to the last items in the form, and work backwards to the rules that apply to the first form controls. We do this because, when there is an error, we want to return focus to the form field with the error (just as we did in the first example). If there is more than one error, we want the user to start at the top of the form, and work through it correcting all errors, so we want the last error that the script *processes* to be the first one that the user will see.

As you will see in a moment, when these functions find an error, not only do they give focus to that form element, they also write the error message into the page, which means we need to change the form in the XHTML page slightly. Let's look at the form again; this time there are empty `<span>` elements after the form controls. These will hold any error messages that are returned. When designing your form, it is important to leave enough space in the page for these error messages. It is also *very* important that there be no spaces between the closing of the `<input />` tag and the start of the `<span>` tag; otherwise, the form will not work.

```
<form name="frmRegister" method="post" action="register.aspx"
  onsubmit="return validate(this);">
  <div>
    <label for="txtUserName">Username:</label>
    <input type="text" id="txtUserName" size="12" /><span
       class="message"></span>
  </div>
  <div>
    <label for="txtPassword">Password: </label>
    <input type="password" id="txtPassword" size="12" /><span
       class="message"></span>
  </div>
  <div>
    <label for="txtPassword2">Confirm your password:</label>
    <input type="password" id="txtPassword2" size="12" /><span
       class="message"></span>
  </div>
  <div>
    <input type="submit" value="Log in" />
  </div>
</form>
```

So, now let's take a look at the function that will check that a user has entered a minimum number of characters into a specified form field.

```
function validateMinimumLength (control, length, errormessage) {
  var error="";
  document.getElementById(control.id).nextSibling.innerHTML="";
  if (control.value.length < length) {
    error = errormessage;
    document.getElementById(control.id).nextSibling.innerHTML=errormessage;
    document.getElementById(control.id).focus();
    }
  return error;
}
```

We are passing three things into the function so that it can perform its calculation:

- The form control we want to check
- The length of the text we expect the user to have entered
- An error message we will show to the user if the control is not long enough

```
function validateMinimumLength (control, length, errormessage) {
```

You can see in the brackets the words `control`, `length`, and `errormessage`. Inside the function, we can use these words to refer to the values passed into the function when it was called. For example, look at this line:

```
if (control.value.length < length) {
```

We have an `if` statement checking whether the length of the control passed in (`control`) is less than the minimum number of characters we will allow (`length`).

If the user has not entered enough characters, we set the value of a variable called `error` (which was declared in the second line of the function) to contain the error message.

```
error = errormessage;
```

We then write the error message into the extra `<span>` element we added after each form control to display the problem.

```
document.getElementById(control.id).nextSibling.innerHTML=errormessage;
```

There is a lot going on in this one line, so let's break it down:

- `control.id` gives you the value of the `id` attribute of the form control that you are working with.
- `document.getElementById(control.id)` gives you the form control you are working with (the `getElementById()` method returns an element given an `id` attribute, and you have just seen how to get the value of the `id` attribute for the form control you are working with using `control.id`).
- `nextSibling` returns the next element after this form control, which is the `<span>` element following it that will hold the error message.
- `innerHTML=errormessage` adds the error message inside that `<span>` element.

If there is a problem, we also give the form control with the problem focus:

```
document.getElementById(control.id).focus();
```

We then return the error back to the `validate()` function in the page with the form (so that it can prevent the form from being submitted). Before looking at how we pass the error back, you may have noticed this line of code, which appears before the `if` statement that checks the length of the value entered into the form control:

```
document.getElementById(control.id).nextSibling.innerHTML="";
```

Its purpose is to ensure the `<span>` element next to the form control is blank. We need to do this because, once a form has been submitted and errors have been spotted, the user may resubmit the form again — in which case we need to clear any error messages, and then check the form control again, and only if there is still a problem display the error message again.

As you might imagine, if you are creating a lot of sites, with many forms, the ability to re-use validation functions saves you from having to code each form again and again.

Figure 12-3 shows the error message generated when the user has not entered a value for the username.

Figure 12-3

The replace() Method

Now that you have seen examples of how to validate text fields using a single function and using re-usable functions, it's time to look at some other ways that you can work with text inputs.

The JavaScript `replace()` method often comes in handy with text inputs because it allows you to replace certain characters with other characters. The simplest way to use the `replace()` method is to use the following syntax:

```
string.replace(oldSubString, newSubString);
```

For example, imagine you had a variable called `message`. The following line would look at this variable and replace instances of the letters "bad" with the letters "good":

```
message.replace('bad', 'good');
```

Let's add this into a form so you can see it working; we have a `<textarea>` that contains the sentence "I think it would be a bad idea to make a badge." When the user clicks on the button, we will replace the letters `bad` with the word `good`. Rather than creating a function to demonstrate this, we can simply put the script we want to run in the `onclick` event of the button (`ch12_eg03.html`).

```
<form name="myForm">
  Message: <textarea name="myTextArea" id="myTextArea" cols="40" rows="10">
  I think it would be a bad idea to make a badge.</textarea>
  <input type="button" value="Replace characters bad"
         onclick="document.myForm.myTextArea.value =
         document.myForm.myTextArea.value.replace('bad', 'good');" />
</form>
```

If you look at the value of the `onclick` attribute, it takes whatever was in the text area and replaces any occurrence of the letters `bad` with the letters `good`. Figure 12-4 shows you what this might look like after the user has pressed the button once.

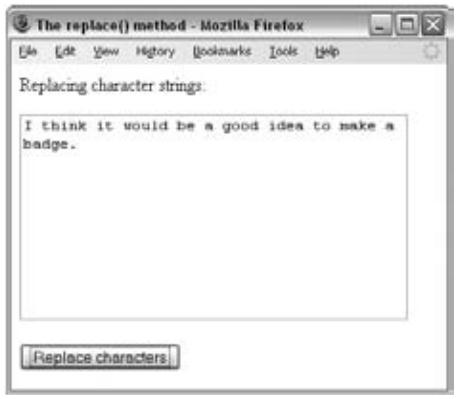

Figure 12-4

There is something interesting to note here. The `replace()` function replaces the *first* instance of the string `bad`. If the button were pressed again, you would see another issue: the word "badge" would turn into "goodge". We can address both of these issues using something known as a *Regular Expression* inside the `replace()` function.

Regular Expressions

Regular Expressions provide a very powerful way to find a particular string, although the downside of this power is that they can become quite complicated. For example, look at this modified example of the `replace()` method we were just looking at (`ch12_eg04.html`):

```
replace(/\bbad\b/gi, 'good');
```

There are four things going on in this Regular Expression; let's build up to this expression, starting with the letters `bad` that we want to replace:

- `/bad/` The forward slashes around the string `bad` indicate that it is looking for a match for that string.
- `/bad/g` The `g` after the second slash (known as a *flag*) indicates that the document is looking for a global match across the whole of the string (without the `g` flag, only the first match in the string is replaced).
- `/bad/gi` The `i` flag indicates that it should be a case-insensitive match (so the string `bad` should be replaced in any mix of characters in upper- and lowercase).
- `/\bbad\b/gi` The `\b` on either side of the string `bad` indicates a word boundary — each specifies that you just want to look for whole words — so the string will be replaced only if the string `bad` is a word on its own. The letters `bad` in the word `badge` would not be replaced because the regular expression says there should be a word boundary on both sides of the letters `bad`. (You cannot just check for the presence of a space on either side of the letters `bad`, because there might be punctuation next to one of the letters.)

Using Regular Expressions, you could also match more than one string using the pipestem character; the following example looks for a match with `bad`, `terrible`, or `awful`:

```
/bad|terrible|awful/
```

Note that if you want to search for any of the following characters, they must be escaped because they have special meanings in Regular Expressions:

```
\ | ( ) [ { ^ $ * + ? .
```

If you want to escape these characters, they must be preceded by a backslash (for example `/\ \ /` matches a backslash and `/\$/` matches a dollar sign).

The table that follows lists some other interesting characters used in regular expressions.

Expression	Meaning
`\n`	Linefeed
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\f`	Form-feed
`\d`	A digit (same as `[0-9]`, which means any digit 0 through 9)
`\D`	A non-digit (same as `[^0-9]` where ^ means not)
`\w`	A word (alphanumeric) character (same as `[a-zA-Z_0-9]`)
`\W`	A non-word character (same as `[^a-zA-Z_0-9]`)
`\s`	A white-space character (same as `[\t\v\n\r\f]`)
`\S`	A non–white-space character (same as `[^\t\v\n\r\f]`)

For a slightly more complex example, if you wanted to replace all carriage returns or linefeeds with the `<br />` tag, you could use the following (`ch12_eg05.html`):

```
replace(/\r|\n|\r\n|\n\r|/g),'<br />');"
```

In this case, the `replace()` method is looking for either linefeeds using `\n` or carriage returns using `\r`. Then these are being replaced with `<br />`. Figure 12-5 shows you what this example could look like with the carriage returns and line feeds replaced with `<br />` tags. If you needed to replace carriage returns or line feeds with the `<br />` element, it is more likely that task would be done behind the scenes when the form is submitted, rather that giving the user a button to replace these characters. But it does illustrate how to use the `replace()` function with a Regular Expression.

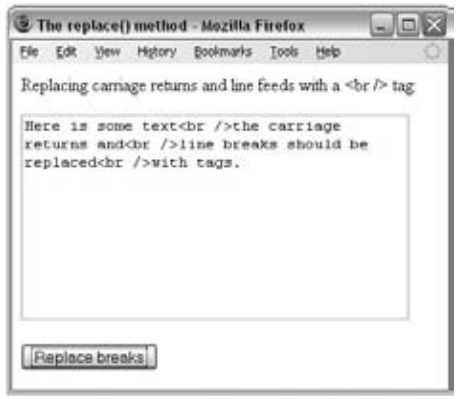

Figure 12-5

Testing Characters Using test() and Regular Expressions

Regular Expressions really come into their own when you need to test whether strings entered by users conform to a pattern. For example, Regular Expressions can be used to test whether the string follows a pattern for e-mail addresses, for an amount of currency, or for a phone number.

To check whether a value a user has entered matches a regular expression, you can use the `test()` method, which takes two parameters: the Regular Expression and the value the user entered. The `test()` method returns `true` if the value entered by the user matches the regular expression, and `false` if it does not.

Here is an example of a function that checks whether a user entered a currency (`ch12_eg06.html`). We will look at it line by line in a moment:

```
function validate(form) {
  var returnValue = true;
  var amountEntered = form.txtAmount.value;

  if (!/^\d+(\.\d{1,2})?$/.test(amountEntered))
  {
    alert("You did not enter an amount of money");
    document.frmCurrency.txtAmount.focus();
    returnValue = false;
  }

return returnValue;
}
```

To start, a variable called `returnValue` is set to `true`; this is what will be returned from the function unless we find an error. Then a variable is set to hold the value the user entered into the form:

```
  var returnValue = true;
  var amountEntered = form.txtAmount.value;
```

Next, we have to test whether the value the user entered is a currency. We will use the `test()` method with the following regular expression:

```
/^\d+(\.\d{1,2})?$/
```

The `test()` method is used inside an `if` statement; note that the first exclamation mark is there to say "If the test fails, perform an action":

```
if (!/^\d+(\.\d{1,2})?$/.test(amountEntered)){
```

If the test fails, we have to tell the user, give the focus back to that form control, and set the variable `returnValue` to have a value of `false`.

```
    alert("You did not enter an amount of money");
    document.frmCurrency.txtAmount.focus();
    returnValue = false;
```

Here is the simple form to test this example:

```
<form name="myForm" onsubmit="return validate(this);"
      action="money.aspx" method="get">
  Enter an amount of money here $
  <input type="text" name="txtAmount" id="txtAmount" size="7" />
  <input type="submit" value="Check format" />
</form>
```

Figure 12-6 shows this form in action.

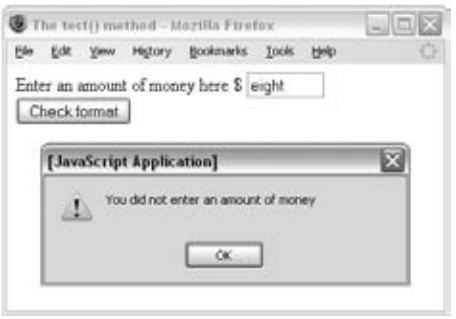

Figure 12-6

Regular Expressions are not the easiest thing to learn to write, and there are entire books devoted to writing complex expressions. However, the table that follows lists some helpful ones that you can use to get you started.

Test for	Description	Regular Expression
White space	No white-space characters.	`\S/;`
Alphabetic characters	No characters of the alphabet or the hyphen, period, or comma may appear in the string.	`/[^a-z \-\.']/gi;`
Alphanumeric characters	No letters or number may appear in the string.	`/[^a-z0-9]/gi;`
Credit card details	A 16-digit credit card number following the pattern XXXX XXXX XXXX XXXX.	`/^\d{4}([-]?\d{4}){3}$/;`
Decimal number	A number with a decimal place.	`/^\d+(\.\d+)?$/;`
Currency	A group of one or more digits followed by an optional group consisting of a decimal point plus one or two digits.	`/^\d+(\.\d\{1,2})?$/;`
E-mail address	An e-mail address.	`/^\w(\.?[\w-])*@\w(\.?[\w-])*\.[a-z]{2,6}(\.[a-z]{2})?$/i;`

Select Box Options

When you want to work with a drop-down select box, the `select` object (which represents the select box) has a very helpful property called `selectedIndex`, which tells you which option the user has selected.

Because this is an index, it will start at 0, so if the user has selected the first option, the `selectedIndex` property will have a value of `0`. If the user selects the second option, the `selectedIndex` property will be given a value of `1`, the third will be given a value of `2`, and so on.

By default, if the user does not change the value that the control has when the page loads, the value will be `0` for a standard select box (because the first option is automatically selected when the form loads). In a multiple select box (which allows users to select more than one option from the list), the default value will be `1` if none of the options are selected (which indicates that the user has not selected any option).

Look at the following simple select box; the first option in this select box asks the user to select a suit of cards (`ch12_eg07.html`):

```
<form name="frmCards" action="cards.aspx" method="get"
      onsubmit="return validate(this)">
  <select name="selCards" id="selCards">
    <option>Select a suit of cards</option>
    <option value="hearts">Hearts</option>
    <option value="diamonds">Diamonds</option>
    <option value="spades">Spades</option>
    <option value="clubs">Clubs</option>
  </select>
  <input type="submit" value="Send selection" />
</form>
```

Now, to check that one of the suits of cards has been selected, you have the `validate()` function, which will have been passed the `form` object as a parameter. In the case of this example, if the `selectedIndex` property of the object representing the select box has a value of `0`, then you have to alert the users that an option has not been selected and ask them to do so.

```
function validate(form) {
  var returnValue = true;
  var selectedOption = form.selCards.selectedIndex;
  if (selectedOption==0)
  {
    returnValue = false
    alert("Please select a suit of cards.");
  }
  return returnValue;
}
```

In Figure 12-7, you can see the warning if the user has not selected a suit of cards.

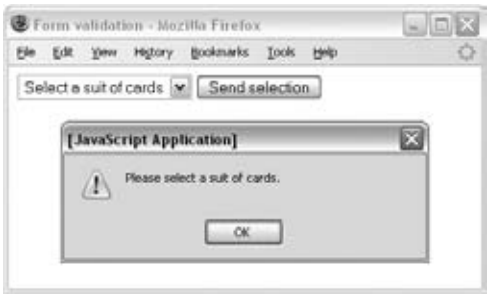

Figure 12-7

If you wanted to access the `value` attribute on the selected option (rather than its index number) you would use the following syntax:

```
form.selCards.options[selected].value
```

This is because you need to look at which of the [option] elements was selected to get its value rather than just the index number of the selected element.

Radio Buttons

A group of radio buttons is different from other form controls in that only one option from a group can be selected at a time, and all members of the group share a value for the name attribute. Scripts that interact with radio buttons usually want to either check that one of the options has been selected, or find out which of the options has been selected.

A set of radio buttons is represented as an array in JavaScript, and in order to find out which one was selected, you need to loop through the array, looking at the checked property of each radio button. If it is selected, the value will be true and false if not. For example, the following is a form with four radio buttons (ch12_eg08.html):

```
<form name="frmCards" action="cards.aspx" method="post"
      onsubmit="return validateForm(this)" >
  <p>Please select a suit of cards.</p>
  <p><input type="radio" name="radSuit" value="hearts" /> Hearts</p>
  <p><input type="radio" name="radSuit" value="diamonds" /> Diamonds</p>
  <p><input type="radio" name="radSuit" value="spades" /> Spades</p>
  <p><input type="radio" name="radSuit" value="clubs" /> Clubs</p>
  <p><input type="submit" value="Submit choice" /></p>
</form>
```

In order to loop through each of the radio buttons in the collection and see which one has a checked property, you will use a for loop.

The following function uses a variable I will call radioChosen to indicate whether one of the radio buttons has been chosen. Its value starts off as false, and if a button has been chosen the value is set to true. Once the loop has gone through each of the radio buttons, an if statement tests whether one of the options has been selected by looking at the value of this variable:

```
function validate(form) {
   var radioButtons = form.radSuit;
   var radioChosen = false;
   for (var i=0; i<radioButtons.length; i++) {
    if (radioButtons[i].checked)
     {
       radioChosen=true;
       returnValue=true;
     }
   }
   if (radioChosen == false) {
     returnValue = false;
     alert("You did not select a suit of cards");
   }
return returnValue;
}
```

While the order of attributes on an element should not matter in XHTML, there was a bug in Netscape 6 and some versions of Mozilla, which means it will show a `checked` property of the radio button only if the `type` attribute is the first attribute given on the `<input />` element.

You can see the result in Figure 12-8.

Figure 12-8

Another way to ensure that one of the options is selected is to preselect an option when the page loads.

Checkboxes

Checkboxes allow a user to select zero, one, or more items from a set of choices (they are not mutually exclusive as radio buttons are). As with radio buttons, when a group of checkboxes share the same name they are made available in JavaScript as an array.

The following is a slight change to the last example using checkboxes instead of radio buttons, and the user can select more than one suit of cards (`ch12_eg09.html`):

```
<form name="frmCards" action="cards.aspx" method="post">
  <p>Please select one or more suits of cards.</p>
  <p><input type="checkbox" name="chkSuit" value="hearts" /> Hearts</p>
  <p><input type="checkbox" name="chkSuit" value="diamonds" /> Diamonds</p>
  <p><input type="checkbox" name="chkSuit" value="spades" /> Spades</p>
  <p><input type="checkbox" name="chkSuit" value="clubs" /> Clubs</p>
  <p><input type="button" value="Count checkboxes"
      onclick="countCheckboxes(frmCards.chkSuit)" /></p>
</form>
```

The following is the function that counts how many checkboxes have been selected and displays that number to the user. As with the last example, if no checkboxes have been selected, you can alert the user that she must select an option value.

```
function countCheckboxes(field) {
  var intCount = 0
  for (var i = 0; i < field.length; i++) {
     if (field[i].checked)
         intCount++; }
  alert("You selected " + intCount + " checkbox(es)");
}
```

You can see the form in Figure 12-9 where the user has selected two checkboxes.

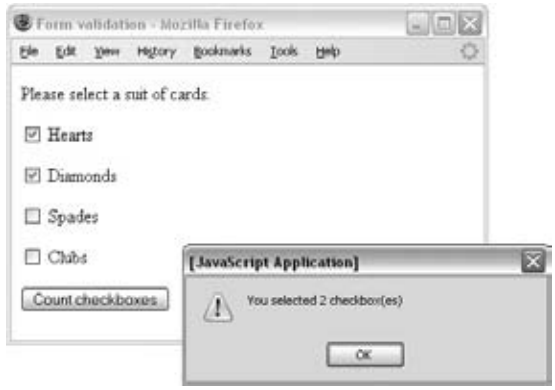

Figure 12-9

Preventing a Form Submission Until a Checkbox Has Been Selected

If you want to ensure that a single checkbox has been selected — for example, if you want a user to agree to certain terms and conditions — you can do so by adding a function to the `onsubmit` event handler similar to those you have seen already. The function checks whether the checkbox has been checked, and if the function returns `true` the form will be submitted. If the function returns `false`, the user would be prompted to check the box. The function might look like this (`ch12_eg10.html`):

```
function checkCheckBox(myForm){
  if (myForm.agree.checked == false )
  {
    alert('You must agree to terms and conditions to continue');
    return false;
  } else
    return true;
}
```

Another technique you may sometimes see is to use script to simply disable the Submit button until users have clicked the box to say that they agree with the terms and conditions.

If you use a script to disable a form control until a user has clicked on an option, you should disable the control in the script when the page loads rather than using the `disabled` *attribute on the element itself. This is important for those who do not have JavaScript enabled in their browsers. If you use the* `disabled` *attribute on a* `<form>` *element and users do not have JavaScript enabled, they will never be able to use that form control. However, if you have used a script to disable it when the page loads, then*

you know that the script will be able to re-enable the form control when the user clicks the appropriate box. This is a great reminder that JavaScript should be used to enhance usability of pages and should not be required in order to use a page.

The following is a very simple page with a form. When the page loads, the Submit button is disabled in the `onload` event. If the user clicks the `chkAgree` checkbox, then the Submit button will be re-enabled (`ch12_eg10.html`):

```
<body onload="document.frmAgree.btnSubmit.disabled=true">
<form name="frmAgree" action="test.aspx" method="post">
I understand that this software has no liability:
<input type="checkbox" value="0" name="chkAgree" id="chkAgree"
       onclick="document.frmAgree.btnSubmit.disabled=false" />
<input type="submit" name="btnSubmit" value="Go to download" /><br />
<p>You will not be able to submit this form unless you agree to the
   <a href="terms.html">terms and conditions</a> and check the terms and
   conditions box.</p>
</form>
</body>
```

You can see this example in Figure 12-10. Note how there is an explanation of why the Submit button might be disabled. This helps users understand why they might not be able to click the Submit button.

This technique can also be used with other form controls — you will see an example that enables a text input later in the chapter.

Figure 12-10

Form Enhancements

The examples you are going to meet in this section do not actually help you validate a form; rather, they simply enhance the usability of a form.

Focus on First Form Item

If a lot of users are likely to interact with your page using a form, you can give focus to that text box when the page loads so that users do not have to move their mouse, click the text input, and then move their hands back to the keyboard before they enter any text. You might choose to do this on a page where there is a prominent login box or an option to search your site.

To give focus to the first text input on a form, simply add an `onload` event handler to the `<body>` element of the document. This handler selects the form control that you want to highlight and uses the `focus()` method of that control to give it focus, as follows (`ch12_eg11.html`):

```
<body onload="document.myForm.myTextBox.focus();">
```

When the page loads, the cursor should be flashing in the form control that you have selected, ready for the user to enter some text as shown in Figure 12-11 (some browsers may also have other ways of indicating an active text box, such as a highlighted border or shaded background).

Note that the `onload` event fires when the complete page has loaded (not as soon as the browser loads that individual element). This is worth bearing in mind because, if you have a very complicated page that takes a long time to load, this might not be a great option for the user; the focus might not be passed to the form until after the user has already had a chance to start typing in that field.

Figure 12-11

Auto-Tabbing Between Fields

The `focus()` method can also be used to pass the focus of one control to another control. For example, if one of the controls on a form is to provide a date of birth in MM/DD/YYYY format, then you can move focus between the three boxes as soon as the user enters a month, and then again once the user has entered a day (`ch12_eg12.html`):

```
<form name="frmDOB">
  Enter your date of birth:<br />
  <input name="txtMonth" id="txtMonth" size="3" maxlength="2"
       onkeyup="if(this.value.length>=2)
       this.form.txtDay.focus();"/>
  <input name="txtDay" id="txtDay" size="3" maxlength="2"
       onkeyup="if(this.value.length>=2)
       this.form.txtYear.focus();" />
  <input name="txtYear" id="txtYear" size="5" maxlength="4"
       onkeyup="if(this.value.length>=4)
       this.form.submit.focus();" />
  <input type="submit" name="submit" value="Send" />
</form>
```

This example uses the `onkeyup` event handler to check that the length of the text the user has entered is equal to or greater than the required number of characters for that field. If the user has entered the required number of characters, the focus is moved to the next box.

Note how the length of the text input is discovered using `this.value.length`. The `this` keyword indicates the current form control, whereas the `value` property indicates the value entered for the control. Then the `length` property returns the length of the value entered for the control. This is a quicker way of determining the length of the value in the current form control than the full path, which would be as follows:

```
document.fromDOB.txtMonth.value.length
```

The other advantage of using the `this` *keyword rather than the full path is that the code would work if you copied and pasted these controls into a different form, as you have not hard-coded the name of the form.*

You can see this example in Figure 12-12; the user has entered an appropriate number of digits in one field so the focus is moved on to the next.

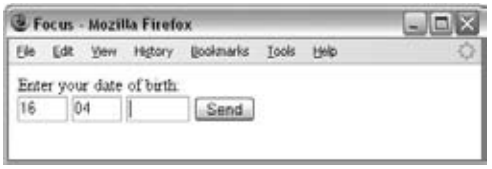

Figure 12-12

You might have noticed that the value of the `size` attribute is also one digit larger than the maximum length of the field to ensure that there is enough space for all of the characters (otherwise the width of the control can sometimes be slightly too small to see all of the characters at once).

If you use this technique, it is always worth testing the form on a few users to check if it is acting as an enhancement. This is important because some web users have become used to pressing the tab key very quickly after entering short numbers (such as individual components of dates of birth) to take them to the next form control.

I have also seen this technique used to allow users to enter their credit card details using four blocks of four codes. While 16 digits is the most common length for a credit card number, and they are often printed in blocks of 4 digits, some Visa cards, for example, contain 13 digits and some American Express cards use 15 digits. So this is not a good idea.

Disabling a Text Input

Sometimes you will want to disable a text input until a certain condition has been met — just as the Submit button was disabled until the user clicked the checkbox to agree to terms and conditions in Figure 12-10.

This example features a form that asks users how they heard about the site; radio buttons are used for several options such as Friend, TV ad, magazine ad, and Other. If the user selects the Other option, the text input next to that option allows the user to indicate how they heard about the site. You can see the form in Figure 12-13.

In this example, it's not just a case of enabling the text box when the user selects the other radio button; you really need to check the value of each radio button as it is selected — after all, if the user selects Other as his or her first choice, but then changes her mind and selects TV or one of the other options, you will want to disable the text input and change its value again. Therefore, each time the user selects a radio button, a function in the head of the document is called that is responsible for enabling and disabling the control and setting values.

Figure 12-13

First, here is the form that gives users the options (`ch12_eg13.html`). Note how the text input is disabled using the `onload` event handler of the `<body>` element and that the text input does not use the `disabled` attribute (this is the same as the earlier example with the Submit button).

```
<body onload="document.frmReferrer.txtOther.disabled=true;
              document.frmReferrer.txtOther.value='not applicable' ">
<h2>How did you hear about us?</h2>
<form name="frmReferrer">
  <input type="radio" name="radHear" value="1"
         onclick="handleOther(this.value);" />From a friend<br />
  <input type="radio" name="radHear" value="2"
         onclick="handleOther(this.value);" />TV Ad<br />
  <input type="radio" name="radHear" value="3"
         onclick="handleOther(this.value);" />Magazine Ad<br />
  <input type="radio" name="radHear" value="4"
         onclick="handleOther(this.value);" />Newspaper Ad<br />
  <input type="radio" name="radHear" value="5"
         onclick="handleOther(this.value);" />Internet<br />
  <input type="radio" name="radHear" value="other"
         onclick="handleOther(this.value);" />Other... Please specify:
  <input type="text" name="txtOther" />
</form>
```

As you can see from this form, every time the user selects one of the options on this form, the `onclick` event calls a function called `handleOther()`. This function is passed the value of the form control as a parameter.

Looking at the function, you can see a simple `if...else` statement that checks whether the value of the selected form control is equal to the text `other` (remember that checking whether one value is equal to another value uses two equal signs because the single equal sign is used to set a variable). If the value of

the selected radio button is "other," the textbox's `disabled` property is set to `false`, and the value cleared. For all other options the textbox is disabled and its value set to `not applicable`.

```
function handleOther(strRadio) {
  if (strRadio == "other") {
    document.frmReferrer.txtOther.disabled = false;
    document.frmReferrer.txtOther.value = '';
  }
  else {
    document.frmReferrer.txtOther.disabled = true;
    document.frmReferrer.txtOther.value = 'not applicable';
  }
}
```

Case Conversion

There are times when it is helpful to change the case of text a user has entered to make it all uppercase or all lowercase — in particular because JavaScript is case-sensitive. To change the case of text, there are two built-in methods of JavaScript's `string` object:

- `toLowerCase()`
- `toUpperCase()`

To demonstrate, here is an example of a text input that changes case as focus moves away from the text input (`ch12_eg14.html`):

```
<form>
  <input type="text" name="case" size="20"
         onblur="this.value=this.value.toLowerCase();" />
</form>
```

If your form data is being sent to a server, it is generally considered better practice to make these changes on the server because they are less distracting for users — a form that changes letter case as you use it can appear a little odd to users.

Trimming Spaces from Beginning and End of Fields

You might want to remove spaces (white space) from the beginning or end of a form field for many reasons, even simply because the user did not intend to enter it there. The technique I will demonstrate here uses the `substring()` method of the `String` object, whose syntax is:

```
substring(startPosition, endPosition)
```

This method returns the part of the string specified by the start and end points — if no end position is given, then the default is the end of the string. The start and end positions are zero-based, so the first character is 0. For example, if you have a string that says `Welcome`, then the method `substring(0, 1)` returns the letter `W`.

First we will look at removing white space from the start of a string. To do this the `substring()` method will be called upon twice. First you use the `substring()` method to retrieve the first letter that the user has entered into a text control. If this character is a space, you can then call the `substring()` method a second time to remove the space.

So the first call to the `substring()` goes in a `while` loop like so:

```
while (this.value.substring(0,1) == ' ')
```

The second time the `substring()` method is called, it selects the value of the control from the second character to the end of the string (ignoring the first character). This is set to be the new value for the form control; so you have removed the first character, which was a space:

```
this.value = this.value.substring(1, this.value.length);
```

This whole process of checking whether the first character is a blank, and then removing it if it is, will be called using the `onblur` event handler; so when focus moves away from the form control, the process starts. Here you can see the entire process using the `while` loop to indicate that, for as long as the first character is a blank, it should be removed using the second call to the `substring()` method (`ch12_eg15.html`).

```
<form>
  <input type="text" name="txtName" size="100"
    value=" Enter text leaving whitespace at start. Then change focus."
    onblur="while (this.value.substring(0,1) == ' ')
    this.value = this.value.substring(1, this.value.length);" /><br />
</form>
```

To trim any trailing spaces, the process is similar but reversed. The first `substring()` method collects the last character of the string, and if it is blank removes it, as follows:

```
<form>
<input type="text" name="txtName" size="100"
       value="Enter text leaving whitespace at end. Then change focus. "
       onblur="while (this.value.substring
          (this.value.length-1,this.value.length) == ' ')
          this.value = this.value.substring(0, this.value.length-1);" /><br/>
</form>
```

As long as you are not targeting browsers as old as Netscape 4 and IE4, you can alternatively use a Regular Expression to trim the spaces, as follows:

```
<form>
  <input type="text" name="removeLeadingAndTrailingSpace" size="100"
    value=" Enter text with white space, then change focus. "
    onblur="this.value=this.value.replace(/^\ \s+/, ").replace(/\s+$/,");"
  /><br />
</form>
```

This removes both trailing and leading spaces.

Regular Expressions are quite a large topic in themselves and were introduced earlier in this chapter. If you want to learn more about them, refer to *Beginning JavaScript, 2nd Edition* by Paul Wilton (Wrox, 2000).

Selecting All the Content of a Text Area

If you want to allow users to select the entire contents of a text area (so they don't have to manually select all the text with the mouse), you can use the `focus()` and `select()` methods.

In this example, the `selectAll()` function takes one parameter — the form control that you want to select the content of (`ch12_eg16.html`):

```
<html>
<head><title>Select whole text area</title>
<script language="JavaScript">
  function selectAll(strControl) {
    strControl.focus();
    strControl.select();
  }
</script>
</head>
<body>
  <form name="myForm">
    <textarea name="myTextArea" rows="5" cols="20">This is some
text</textarea>
    <input type="button" name="btnSelectAll" value="Select all"
           onclick="selectAll(document.myForm.myTextArea);" />
  </form>
</body>
</head>
</html>
```

The button that allows the user to select all has an `onclick` event handler to call the `selectAll()` function and tell it which control it is whose contents should be selected.

The `selectAll()` function first gives that form control focus using the `focus()` method and then selects its content using the `select()` method, because the form control must gain focus before it can have its content selected. The same method would also work on a single-line text input and a password field.

Check and Uncheck All Checkboxes

If there are several checkboxes in a group of checkboxes, it can be helpful to allow users to select or clear a whole group of checkboxes at once. The following are two functions that allow precisely this:

```
function check(field) {
  for (var i = 0; i < field.length; i++) {
    field[i].checked = true;}
}
function uncheck(field) {
  for (var i = 0; i < field.length; i++) {
    field[i].checked = false; }
}
```

In order for these functions to work, more than one checkbox must be in the group. You then add two buttons that call the check or uncheck functions, passing in the array of checkbox elements that share the same, name such as the following (`ch12_eg17.html`):

```
<form name="frmSnacks" action="">
   Your basket order<br />
   <input type="checkbox" name="basketItem" value="1" />Chocolate
   cookies<br />
   <input type="checkbox" name="basketItem" value="2" />Potato chips<br />
   <input type="checkbox" name="basketItem" value="3" />Cola<br />
   <input type="checkbox" name="basketItem" value="4" />Cheese<br />
   <input type="checkbox" name="basketItem" value="5" />Candy bar<br /><br
/>
   <input type="button" value="Select All"
          onclick="check(document.frmSnacks.basketItem);" />

   <input type="button" value="Deselect All"
          onclick="uncheck(document.frmSnacks.basketItem);" />
</form>
```

You can see how this form appears in Figure 12-14.

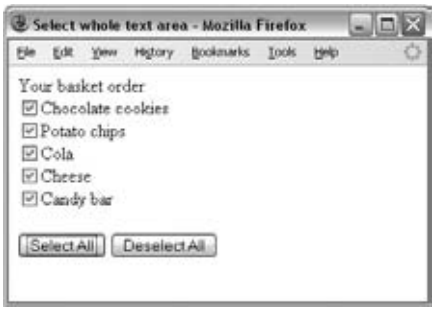

Figure 12-14

This could also be combined into a single function, which could be called from the same button, such as the following:

```
function checkUncheckAll(field) {
 var theForm = field.form, i = 0;
 for(i=0; i<theForm.length;i++){
   if(theForm[i].type == 'checkbox' && theForm[i].name != 'checkall'){
   theForm[i].checked = field.checked;
   }
  }
}
```

Try It Out An E-mail Form

Having seen lots of examples of how to work with forms, it is time to try to put some of these techniques together. In this exercise you are going to create an e-mail form that has a few interesting features. It checks that all fields have an entry of some kind, and uses a Regular Expression to check the structure of an e-mail address. The form also includes a quick address book that contains addresses of potential recipients of the e-mail. Figure 12-15 shows you what the form is going to look like.

Figure 12-15

1. Create a skeleton XHTML document with `<head>`, `<title>`, and `<body>` elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>
  <title>E-mail form</title>
</head>
<body>
  </body>
</html>
```

2. In the body of the document, add the `<form>` element and two `<div>` elements. The first `<div>` holds the To, CC, and Subject fields, while the second holds the quick address. Note

how there is an empty `<span>` element next to the form controls; this will hold any error messages.

```
<form name="frmEmail" onsubmit="return validate(this)" action="success.html"
      method ="post">
  <div id="toCCsubject">

    <label>Send to:</label>
    <input type="text" size="50" name="txtTo" id="txtTo" /><span
      class="error"></span>

    <label>CC:</label>
    <input type="text" size="50" name="txtCC" id="txtCC" /><span
       class="error"></span>

    <label>Subject:</label>
    <input type="text" size="50" name="txtSubject" id="txtSubject" /><span
      class="error"></span>

  </div>
  <div id="addressBook">
    <! —  quick address book will go here  — ></td>
  </div>
```

3. Next you need to add the quick address book into the second `<div>` element. The address book uses a multiple-line select box. Underneath it are two buttons: one to add addresses to the `txtTo` field and one to add addresses to the `txtCC` field. Both of these buttons call the `add()` function when clicked:

```
Quick address book:<br />
<select size="4" name="selectList1" style="width:150px">
  <option value="sales@example.org">Sales</option>
  <option value="marketing@example.org">Marketing</option>
  <option value="research@example.org">Research</option>
  <option value="support@example.org">Customer Support</option>
  <option value="it@example.org">IT</option>
</select><br />
<input type="button" onclick="add(txtTo, document.frmEmail.selectList1);"
       value="Send to" />
<input type="button" onclick="add(txtCC, document.frmEmail.selectList1);"
       value="CC" />
```

4. Add the message `<textarea>` element and a `Send E-mail` button (and close the `</form>` element):

```
<div class="message">
  Message:<br />
  <textarea name="txtMessage" rows="20" cols="115"></textarea><br />
  <input type="submit" value="Send E-mail" />
</div>
</form>
```

5. Now that the form itself is complete, we can start to look at the script. The `add()` function adds e-mail addresses from the address book into the To or CC fields (if there is already an address in there, the semicolon is added to separate out multiple addresses).

   The `add()` function takes two parameters:

   - `objInput`: The field that the selected address is being sent to
   - `objList`: The select box list that contains the e-mail addresses

   This function starts by collecting the value of the selected item, using the `selectedIndex` property of the select box, and placing it in a variable called `strGroup`. Next it checks whether the form field the address is being added to is empty; if it is, the e-mail address stored in the `strGroup` attribute is added to the field. If the `To` or `CC` field is not empty, a semicolon and a space will be added before the e-mail address because this is the usual delimiter for multiple e-mail addresses:

```
function add(objInput, objList){
var strGroup = objList.options[objList.selectedIndex].value;
  if (objInput.value == "")
  {
     objInput.value = strGroup
  }
  else
  {
     objInput.value += ('; ' + strGroup)
  }
}
```

6. Now, let's add the second function — the `validate()` function, which you can see is quite long:

```
function validate(form) {

  var returnValue = true;
  var sendTo = form.txtTo.value;
  var cc = form.txtCC.value;
  var subject = form.txtSubject.value;
  var message = form.txtMessage.value;
  var rxEmail=/^\w(\.?[\w-])*@\w(\.?[\w-])*\.[a-z]{2,6}(\.[a-z]{2})?$/i;

  document.getElementById("txtTo").nextSibling.innerHTML="";

  if (sendTo == "")  {
    returnValue = false;
      document.getElementById("txtTo").nextSibling.innerHTML=
        "There are no email addresses in the To field";
    document.getElementById("txtTo").focus();
  }
  else {
    var arrTo = sendTo.split("; ");
```

```
    for (var i=0; i<(arrTo.length); i++) {
      if (!rxEmail.test(arrTo[i]))
      {
        returnValue = false;
        document.getElementById("txtTo").nextSibling.innerHTML=
          "The e-mail address(es) provided does not appear to be valid";
        document.getElementById("txtTo").focus();
      }
    }
  }


  document.getElementById("txtCC").nextSibling.innerHTML="";
  if (sendTo != "")
  {
    var arrCC = cc.split("; ");
    document.getElementById("txtCC").nextSibling.innerHTML="";

    for (var i=0; i<(arrCC.length); i++) {
      if (!rxEmail.test(arrCC[i]))
        {
          returnValue = false;
          document.getElementById("txtCC").nextSibling.innerHTML=
            "The e-mail address(es) provided does not appear to be valid";
          document.getElementById("txtCC").focus();
        }
      }
    }

  document.getElementById("txtSubject").nextSibling.innerHTML="";
  if (subject == "")
  {
    returnValue = false;
      document.getElementById("txtSubject").nextSibling.innerHTML=
        "There is no subject line for this e-mail";
      document.getElementById("txtSubject").focus();
  }

  document.getElementById("txtMessage").nextSibling.innerHTML="";
  if (message=="")
  {
    returnValue = false;
      document.getElementById("txtMessage").nextSibling.innerHTML=
        "There is no message for this e-mail";
      document.getElementById("txtMessage").focus();
  }


return returnValue;
}
```

The `validate()` function is quite a bit more complex, so let's break it down and look at it step by step. It starts off by setting a `returnValue` variable to `true` and collecting the form's values into variables:

```
function validate(form) {
  var returnValue = true;
  var sendTo = form.txtTo.value;
  var cc = form.txtCC.value;
  var subject = form.txtSubject.value;
  var message = form.txtMessage.value;
  var rxEmail=/^\w(\.?[\w-])*@\w(\.?[\w-])*\.[a-z]{2,6}(\.[a-z]{2})?$/i;
```

The first task for the validate form is to check whether the user has entered an e-mail into the To field, and if so check that the e-mail address is valid. Since we will be writing any errors into the `<span>` elements that follow the form fields, we need to empty any content from the `<span>` element that follows the To field (because we are about to recheck the values the user has entered, and if the user has corrected an earlier mistake we do not want to leave an error message there).

The real checks appear in an `if ... else` statement; it then checks whether the user has entered a value into the To field. If there is no value, it sets the `returnValue` attribute to `false`, adds an error into the `<span>` element following the field, and passes focus to the To field.

```
if (sendTo == "")  {
    returnValue = false;
    document.getElementById("txtTo").nextSibling.innerHTML="There are no email
    addresses in the To field";
    document.getElementById("txtTo").focus();
  }
```

The `validate` function gets more interesting when it comes to checking that valid e-mail addresses have been entered into the form. The Regular Expression that's used to check the e-mail addresses was stored in a variable at the top of the page — this time called `rxEmail`:

```
var rxEmail=/^\w(\.?[\w-])*@\w(\.?[\w-])*\.[a-z]{2,6}(\.[a-z]{2})?$/i;
```

In order to check that the e-mail addresses entered are valid, the `To` field gets split into an array using the `split()` method of the `String` object. This function will take a string and split it into separate values whenever it comes across a specified character or set of characters. In this case, the method looks for any instances of a semicolon followed by a space, and wherever it finds these it creates a new item in the array.

```
 var arrTo = sendTo.split("; ");
```

Imagine having the following e-mail addresses (note that this is just to illustrate the `split()` method; it is not part of the code):

```
sales@example.com; accounts@example.com; marketing@example.com
```

These would be split into the following array (again, this is not part of the code from the example):

```
arrTo[0] = "sales@example.com"
arrTo[1] = "accounts@example.com"
arrTo[2] = "marketing@example.com"
```

So now there has to be a `for` loop in the code that will go through each e-mail address in the array and check that it follows the pattern described in the Regular Expression. The `for` loop has three parameters; the first sets a counter called `i` to be `0`, checks that the counter is less than the number of items in the array, and finally increments the counter. Inside the loop is an `if` statement that checks whether the e-mail address matches the Regular Expression using the `test()` method; if it does not, it will set the `returnValue` to `false` and add a message to the page to indicate that the value does not seem to be a valid e-mail address:

```
for (var i=0; i<(arrTo.length); i++) {
      if (!rxEmail.test(arrTo[i]))
      {
        returnValue = false;
        document.getElementById("txtTo").nextSibling.innerHTML="The e-mail
        address(es) provided does not appear to be valid";
        document.getElementById("txtTo").focus();
      }
    }
```

After this, you can see a similar setup for the CC field:

```
  document.getElementById("txtCC").nextSibling.innerHTML="";
  if (sendTo != "")
  {
    var arrCC = cc.split("; ");
    document.getElementById("txtCC").nextSibling.innerHTML="";

    for (var i=0; i<(arrCC.length); i++) {
      if (!rxEmail.test(arrCC[i]))
        {
          returnValue = false;
          document.getElementById("txtCC").nextSibling.innerHTML=
            "The e-mail address(es) provided does not appear to be valid";
          document.getElementById("txtCC").focus();
        }
      }
    }
```

The last checks ensure that something was entered into the subject and message inputs:

```
document.getElementById("txtSubject").nextSibling.innerHTML="";
  if (subject == "")
  {
    returnValue = false;
      document.getElementById("txtSubject").nextSibling.innerHTML=
        "There is no subject line for this e-mail";
    document.getElementById("txtSubject").focus();
```

```
    }

    document.getElementById("txtMessage").nextSibling.innerHTML="";
    if (message=="")
    {
      returnValue = false;
        document.getElementById("txtMessage").nextSibling.innerHTML=
          "There is no message for this e-mail";
      document.getElementById("txtMessage").focus();
    }
```

Finally we return the `returnValue` attribute to indicate whether the form can be submitted.

```
  return returnValue;
  }
```

7. Save the file as `emailform.html`, and when you open it in the browser window it should resemble the example you saw in Figure 12-15.

Now you have an example of a form that has more than one function. It uses JavaScript to create a quick address book and validates the entries to stop the user from trying to send an e-mail address that is not valid.

JavaScript Libraries

The examples you have seen so far in this chapter have been designed to give you a better understanding of how JavaScript can be integrated into your XHTML documents — how events (such as a user hovering over, or clicking on, an element) can be used to trigger JavaScript functions.

In the case of the form validation script, you have seen an example of how to include an external JavaScript into a page with one line of code, and how the functionality of that script can be used in multiple web pages. This demonstrated that if you write a script carefully, once the work has been done, you can use it again and again in other pages with minimum effort.

JavaScript files that contain functions that you want to use in several pages are often referred to as *JavaScript libraries* (because, once you have included the file in your page, you can borrow its functionality).

In this section, you are going to meet several scripts that offer functionality you might want to use in several of your web pages. For example, you will see how to create:

- Animated effects and drag-and-drop lists
- Lightboxes and modal windows
- Sortable tables
- Calendar controls
- Auto-completing text inputs

Each of these example scripts is built on top of a one of the major JavaScript libraries that you will see used again and again if you explore other examples on the Web. So, the examples you meet in this chapter are built upon:

- Scriptaculous (which is actually built on top of another JavaScript library called Prototype)
- JQuery
- MochiKit
- Yahoo User Interface (YUI)

While you will meet several helpful scripts before the end of this chapter, the real purpose of this section is to illustrate how you can easily integrate this type of JavaScript code (that someone else has written) into your pages to offer complex functionality with minimum effort.

> *I have included versions of each of these libraries with the code download for this chapter. If you look in the code folder for Chapter 12, you will see inside the scripts folder that there are folders called* `scriptaculous`, `mochikit`, *and* `yui` *(each folder corresponding to the three libraries you will be using). It is worth, however, checking the web sites for each of these libraries as they may have been updated since this book was written.*

Animated Effects Using Scriptaculous

To start with, I am going to show you some basic animation effects that can be created using a JavaScript library called Scriptaculous, which is built on top of another JavaScript library called Prototype. I have included a copy of Scriptaculous 1.8.2 and Prototype 1.6.0 with the code download for this chapter; however, you can check for more recent versions and download your own copy of these files from `http://script.aculo.us/`.

Scriptaculous can help you with many kinds of tasks: animation, adding drag-and-drop functionality, building editing tools, and creating autocompleting text inputs. It also includes utilities to help create DOM fragments. But in this section, you will just focus on some of the animation effects. (You will see examples of drag-and-drop and autocompleting text inputs later in the chapter.)

Scriptaculous contains many functions that help you create different types of animations. The first example is going to feature four of these animated effects, and it will demonstrate how easily you can add powerful features into your page with very little code. You can see what this page will look like in Figure 12-16, although you really need to try the example to see the actual animation by opening `ch12_eg18.html` in your browser.

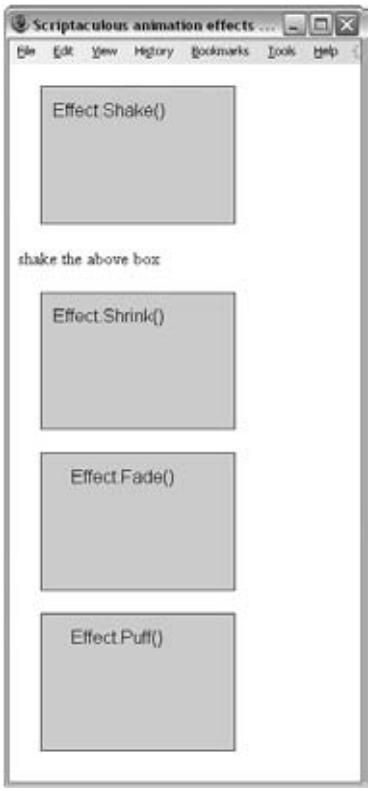

Figure 12-16

As you may remember from the last chapter, JavaScript programmers use objects to represent real life objects or concepts. In the case of the Scriptaculous library, the animated effects are handled by an object called `Effect`. There are different types of effects, which are each called using different methods. For example, to create a shaking effect you need to call the `Shake()` method of the `Effect` object. To create a fading effect you use the `Fade()` method of the `Effect` object. Some of these methods can take different parameters that allow you to vary how the effect works.

In order to use the animated effects and other features of the Scriptaculous library, you need to add both the `prototype.js` library and the `scriptaculous.js` library, which I have placed in a folder called `scriptaculous`:

```
<script src="scripts/scriptaculous/prototype.js"
  type="text/javascript"></script>
<script src="scripts/scriptaculous/scriptaculous.js"
   type="text/javascript"></script>
```

In the body of the page, there is a box to demonstrate each effect; when you click on the box, the effect is triggered. Each box is created using a `<div>` element, inside which I have written the name of the `Effect` object and the method that box will demonstrate. Let's start by looking at the first box, which demonstrates the shake effect:

```
<div id="effect-shake" onclick="Effect.Shake(this)">
  Effect.Shake()
</div>
```

As you already know, the event attributes allow events to trigger JavaScripts. In this example, the `onclick` event causes the `Effect` object's `Shake()` method to be called. As with the forms you looked at earlier, the keyword `this` (which you can see inside the parentheses of the `Shake()` method) indicates that we want to create a shake animation for this element.

If you wanted the effect to be triggered by the user interacting with a different element, you could pass the `Shake()` method the `id` of the box you want to shake. For example, here is a simple link; when the user clicks on the words "shake the above box," the first box will shake.

```
<a onclick="Effect.Shake('effect-shake')">
  shake the above box
</a>
```

Let's look at the second example, which shrinks the box when you click on it until it vanishes. It is very similar but this time we are calling the `Shrink()` method of the `Effect` object.

```
<div id="demo-effect-shrink" onclick="Effect.Shrink(this);">
    Effect.Shrink
</div>
```

Because the box disappears once it has shrunk, you can call the `Appear()` method of the `Effect` object to make it re-appear. You cannot call this method right away; otherwise, you will not be able to see the effect working, so you can call this method after a set period using the `setTimeout()` method of the `window` object (this is a JavaScript object, not one created by one of the JavaScript libraries). In this case, the `Appear()` method is called after 2500 milliseconds.

```
<div id="demo-effect-shrink" onclick="Effect.Shrink(this);
window.setTimeout('Effect.Appear(\'effect-shrink\')',2500);">
```

Here are the other two effects:

```
<div id="effect-fade" onclick="Effect.Fade(this);
    window.setTimeout('Effect.Appear(\'effect-fade\')',2500);">
      Effect.Fade()
</div>

<div id="effect-puff" onclick="Effect.Puff(this);
    window.setTimeout('Effect.Appear(\'effect-puff\')',2500);">
      Effect.Puff()
</div>
```

You do not need to know *how* the Script creates these effects; all you need to know is the syntax of the method that calls each effect.

As you can see, this is a very simple way of creating animated effects using JavaScript; one that lots of other scripts make use of.

Drag-and-Drop Sortable Lists Using Scriptaculous

Having seen how simple it is to create some animations using Scriptaculous, let's have a look at an example of something you might want to do in a user interface: create drag-and-drop lists. You may have seen some sites where you can re-order lists (such as "to do" lists or top 10 lists) just by dragging and dropping the elements.

You can see the example you are going to build in Figure 12-17; when the page loaded, the boxes were in numerical order. However, they have now been dragged and dropped to a different order.

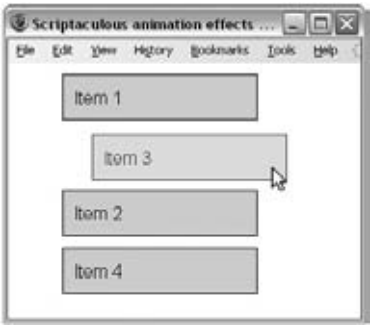

Figure 12-17

In this example (`ch12_eg19.html`), you need to include the Scriptaculous and Prototype libraries again. Then you have a simple unordered list (there are some CSS rules in the head of the document that control the presentation of the list to make each list item appear in its own box).

```
<script src="scripts/prototype.js" type="text/javascript"></script>
<script src="scripts/scriptaculous.js" type="text/javascript"></script>
<style type="text/css">
  li {border;1px solid #000000; padding:10px; margin-top:10px;
  font-family:arial, verdana, sans-serif;background-color:#d6d6d6;
  list-style-type:none; width:150px;}
</style>
</head>
<body>
<ul id="items_list">
   <li id="item_1">Item 1</li>
   <li id="item_2">Item 2</li>
   <li id="item_3">Item 3</li>
   <li id="item_4">Item 4</li>
</ul>
```

In order to make this list sortable, you just need to add one `<script>` element after the list:

```
<script type="text/javascript" language="javascript">
   Sortable.create("items_list",{dropOnEmpty:true,constraint:false});
</script>
```

Here you are using the `Sortable` object that is part of the Scriptaculous library, and calling its `create()` method to turn the list into a sortable one. The `create()` method takes two parameters:

- The first is the value for the `id` attribute of the unordered list element, in this case `items_list`.
- The second features options that describe how the sortable list should work. The first of the options specified here is `dropOnEmpty` with a value of `true` to indicate that element should only be dropped between elements, not on top of another one, and the `constraint` property, which is set to `false`. (If this were left off or `true` it would allow items to be moved along a vertical axis only, whereas set to `false` you can move the list items to the left or right while dragging them).

This kind of drag-and-drop list is often linked to some kind of functionality that will update a database (such as allowing users to re-order a "to do" or preference list), which would involve a script on the server written in a language such as ASP.NET or PHP code. However, this example does demonstrate something that is achieved very easily with just a few lines of code, thanks to the Scriptaculous library.

Creating a Lightbox

A lightbox is the term given to an image (or set of images) that opens up in the foreground of the browser without reloading the page. The image appears in the middle of the browser window, and the rest of the page is often dulled out. Figure 12-18 shows an example of a lightbox activated on a page (`ch12_eg20.html`).

Figure 12-18

The script we will be looking at in this section is called Lightbox2, and was written by Lokesh Dhakar. It is based on the Prototype and Scriptaculous libraries so we will include them as well as the lightbox script.

So, to create a lightbox, first you add in the following three scripts (`ch12_eg20.html`):

```
<script type="text/javascript" src="scripts/scriptaculous/prototype.js">
</script>
<script type="text/javascript"
    src="scripts/scriptaculous/scriptaculous.js?load=effects,builder"></
script>
<script type="text/javascript" src="scripts/lightbox.js"></script>
```

There are also some CSS styles that are used in the lightbox, so you can either link to this style sheet or include the rules from it in your own style sheet:

```
<link rel="stylesheet" href="css/lightbox.css" type="text/css"
    media="screen" />
```

Finally, you create a link to each of the images in the lightbox; here you can see a lightbox that contains three images:

```
<a href="images/image-1.jpg" rel="lightbox[Japan]">Picture 1</a>
<a href="images/image-2.jpg" rel="lightbox[Japan]">Picture 2</a>
<a href="images/image-3.jpg" rel="lightbox[Japan]">Picture 3</a>
```

Note the use of the `rel` attribute on the links; this uses the keyword `lightbox`, followed by a name for the `lightbox` in square brackets. This lightbox is called `Japan`.

Inside the link, you can have anything. There is just some simple text in this example, although you could include a thumbnail image to represent each of the larger images.

You do not have to indicate the size of the image, as the script will automatically determine this and resize the lightbox to fit the image.

You can also have multiple lightboxes on the same page as long as you give each lightbox a different name; here you can see a second lightbox added with photographs from Paris:

```
<h1>Images from Japan</h1>
<a href="images/Japan1.jpg" rel="lightbox[Japan]">Picture 1</a>
<a href="images/Japan2.jpg" rel="lightbox[Japan]">Picture 2</a>
<a href="images/Japan3.jpg" rel="lightbox[Japan]">Picture 3</a>
<h1>Images from Paris</h1>
<a href="images/Paris1.jpg" rel="lightbox[Paris]">Picture 1</a>
<a href="images/Paris2.jpg" rel="lightbox[Paris]">Picture 2</a>
<a href="images/Paris3.jpg" rel="lightbox[Paris]">Picture 3</a>
```

This is just one of many examples of lightbox scripts you would find if you searched for a lightbox script (other popular examples include Thickbox, Fancybox, and JQuery Lightbox).

This is a similar technique you may have seen used to create modal dialog boxes.

Creating a Modal Window

A modal window is a "child" window that you have to interact with before you can go back to the main window. You have probably seen modal windows on web sites you have visited; they are often used for login or contact forms, and their appearance is similar to the lightbox that you just saw (with a grayed-out page). You can see the example we are going to create in Figure 12-19.

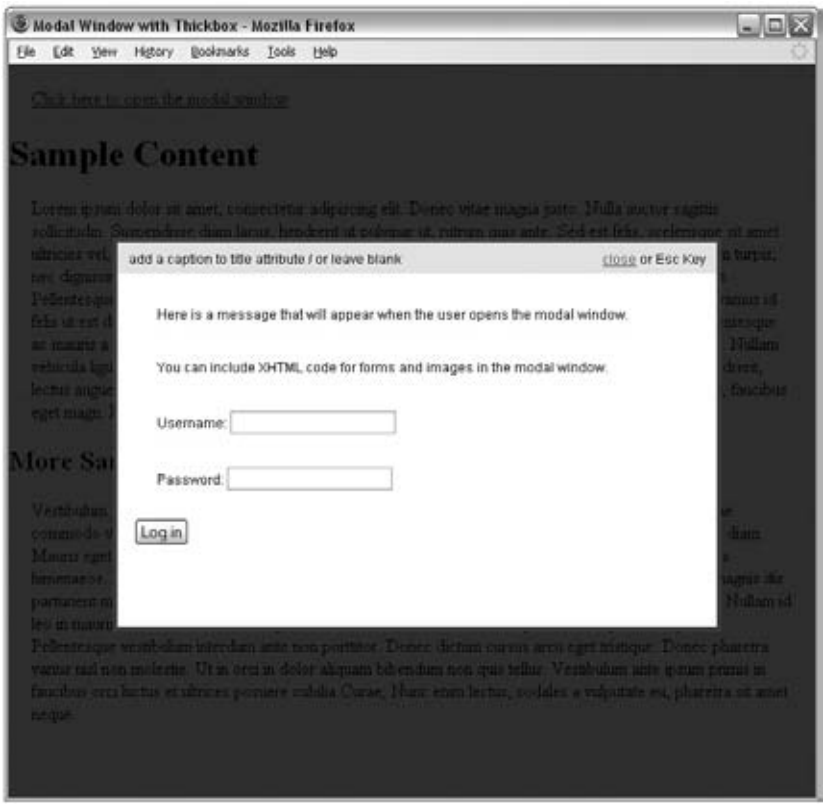

Figure 12-19

The example of a modal window we will look at uses a script called Thickbox written by Cody Lindley; this example is built on another of the main JavaScript libraries called JQuery.

To re-create this example, first you need to include the JQuery library along with the Thickbox script and style sheet in the head of your document like so (`ch12_eg21.html`):

```
<script type="text/javascript"
    src="scripts/jquery/jquery-1.3.2.js"></script>
<script type="text/javascript" src="scripts/thickbox/thickbox.js"></script>
<link rel="stylesheet" href="scripts/thickbox/thickbox.css"
    type="text/css" media="screen" />
```

Next, you can write the content that you want to appear in the modal window:

```
<div id="myOnPageContent"><p>
  Here is a message that will appear when the user opens the modal window.
</p></div>
```

You do not want this to show when the page loads, so add the following style sheet rule into the header of the page (or your style sheet):

```
<style type="text/css">#myOnPageContent {display:none;}</style>
```

Finally, you can add a link that will open up the modal window. To do this, we will use an `<a>` element, although you could trigger it using the `onload` event of the document (to bring the window up when the page first loads), or you could use a form element.

There are a few things to note about this `<a>` element, starting with the `class` attribute whose value is `thickbox`:

```
<a class="thickbox"
  href="#TB_inline?height=300&width=500&inlineId=myOnPageContent"
  title="add a caption to title attribute / or leave blank">link</a>
```

As you can see, the value of the `href` attribute is quite long. Let's break that down:

- `#TB_inline?` is used to trigger the lightbox.
- `height` and `width` specify the height and width of the box in pixels. You can adjust these depending on the size of your message.
- `inlineID` is used to specify the value of the `id` attribute of the element that contains the message to appear in the modal window. In our case, the `<div>` element containing the message for the modal window was `myOnPageContent`.

You may also have noticed that the link used a `title` attribute whose value appeared as a title at the top of the modal window.

Again this demonstrates how you can add a powerful technique to your page with very little code.

Sortable Tables with MochiKit

In this example, you will create a *sortable table*, which means you can re-order the contents of the table by clicking on the heading for each column. This kind of feature is particularly helpful when dealing with a long table where different users might want to sort the data in different ways (according to different column headings).

Figure 12-20 illustrates a table of employees; the up arrow next to the "Date started" table heading indicates that the table's contents are being ordered by the date the employee started (in ascending order). If you clicked on the heading "Name" you would be able to sort the table by alphabetical order of employee names.

Figure 12-20

This example uses another JavaScript library — MochiKit. By looking at examples that use different libraries, you can see how easy it is to work with the various libraries (which offer different functionality). You can download the latest version of MochiKit from `www.mochikit.com/`, although I have included version 1.3.1 with the download code for this chapter.

In order to create a sortable table, you again need to include two scripts; the first is for the `MochiKit.js` JavaScript library, and the second is for the `sortable_tables.js` file (`ch12_eg22.html`).

```
<script type="text/javascript"
   src="scripts/MochiKit/MochiKit.js"></script>
<script type="text/javascript"
  src="scripts/MochiKit/examples/sortable_tables/sortable_tables.js">
</script>
```

Next, I have added a couple of CSS styles to distinguish the headers from the columns and to set the font used:

```
<style type="text/css">
  th, td {font-family:arial, verdana, sans-serif;}
  th {background-color:#000000;width:200px;color:#ffffff;}
</style>
```

Now let's look at the actual table; there are just three things you need to add to your table so that you can sort the contents by clicking on the headings:

- The `<table>` element needs an `id` attribute whose value is `sortable_table`.
- For each column, the `<th>` (table heading) elements need to have an attribute called `mochi:sortcolumn` (we will look at the values this attribute takes after you have seen the code).
- The first row of `<td>` elements needs to have `mochi:content` (again we will look at the values for this attribute after you have seen the code).

So here is the table with these additions:

```
<table id="sortable_table" class="datagrid">
  <thead>
    <tr>
      <th mochi:sortcolumn="name str">Name</th>
      <th mochi:sortcolumn="department str">Department</th>
      <th mochi:sortcolumn="datestarted isoDate">Date started</th>
      <th mochi:sortcolumn="extension str">Employee ID</th>
    </tr>
  </thead>
  <tbody>
    <tr mochi:repeat="item domains">
      <td mochi:content="item.name">Tim Smith</td>
      <td mochi:content="item.department">IT</td>
      <td mochi:content="item.datestarted">2007-02-10</td>
      <td mochi:content="item.extension">12</td>
    </tr>
    <tr>
      <td>Claire Waters</td>
      <td>Finance</td>
      <td>2006-09-24</td>
      <td>24</td>
    </tr>
    <tr>
      <td>Hetal Patel</td>
      <td>HR</td>
      <td>2006-01-10</td>
      <td>05</td>
    </tr>
    <tr>
      <td>Mark Whitehouse</td>
      <td>Sales</td>
      <td>2007-03-28</td>
      <td>09</td>
    </tr>
  </tbody>
</table>
```

The `<th>` elements carry the `mochi:sortcolumn` attribute, which contains two items of information separated by a space:

- A unique ID for that column.
- The format of the data in that column. This can be `str` if the data is a string or `isoDate` for a date in the format shown.

Now take a look at the first row of data because the `<td>` elements in this row carry `mochi:content` attributes. Again these are made up of two items, this time separated by a period or full stop:

- The keyword `item`
- The unique ID for the column that was specified in the `mochi:sortcolumn` attribute in the corresponding header

As with the other examples in this section, it is best to try it using the download code so you can see how it works and understand how easy it can be to add quite complex functionality to a table — creating an effect similar to the Sort Data options in Excel, which are useful when dealing with large amounts of data.

Creating Calendars with YUI

The fourth and final JavaScript library you will be looking at is the Yahoo User Interface library; it is the largest of the three libraries, with all kinds of functionality split into many separate scripts. I have only included a subset of version 2.7.0 of the YUI library with the code download for this chapter (the full version is over 11MB in size); however, you can download the full and latest version from `http://developer.yahoo.com/yui/`. Broadly speaking, the YUI is split into four sections:

- At the heart of the YUI there are three scripts: the Yahoo Global Object, the DOM Collection, and the Event Utility. When using the library, sometimes you will only need to include one or two of these scripts; other times you will need all three.
- Then there are library utilities, which provide functionality that you might use in many tasks, such as libraries that help you create animated effects, drag-and-drop functionality, and image loading. When one of these scripts requires functionality from one of the core scripts, you are told in the accompanying documentation.
- At the other end of the spectrum, the library contains scripts to create individual kinds of user interface controls, such as calendars, color pickers, image carousels, and a text editor. There are helpful "cheat sheets" that show you how to quickly create each of these user interface components.
- Finally, there is a set of CSS style sheets, which you might need to use with some of the UI controls in order to make them appear as you want.

In this section, we are going to see how to use the YUI to add a Calendar to your web page with just a few lines of code. Figure 12-21 shows what the calendar will look like.

Figure 12-21

To start, you have to include three core JavaScript files from the YUI library:

```
<script type="text/javascript" src="scripts/yui/yahoo/yahoo.js"></script>
<script type="text/javascript" src="scripts/yui/event/event.js" ></script>
<script type="text/javascript" src="scripts/yui/dom/dom.js" ></script>
```

Next, you need to add the `calendar.js` script, which is used to create the calendar (`ch12_eg23.html`).

```
<script type="text/javascript"
  src="scripts/yui/build/calendar/calendar.js"></script>
```

For this example, you will also include one of the css files that is included with the YUI download:

```
<link type="text/css" rel="stylesheet"
href="scripts/YUI/skins/sam/calendar.css">
```

In the body of the page, you need to add a `<div>` element, which will be populated by the calendar.

```
<div id="cal1Container"></div>
```

Finally, you add in the script, which calls the YUI library, and fills the `<div>` element with the calendar.

```
<script>
    YAHOO.namespace("example.calendar");
      YAHOO.example.calendar.init = function() {
      YAHOO.example.calendar.cal1 =
        new YAHOO.widget.Calendar("cal1","cal1Container");
      YAHOO.example.calendar.cal1.render();
    }
  YAHOO.util.Event.onDOMReady(YAHOO.example.calendar.init);
</script>
```

Rather like some of the other examples in this section, this is likely to be tied into some other kind of functionality, such as a holiday booking form where you are specifying dates you want to travel or an events list where you are looking at what is happening on a particular date. But this does demonstrate how libraries can be used to add significant functionality to your pages with ease.

Auto-Completing Text Inputs with YUI

The final example you will look at in this section allows you to create a text input where users are offered suggestions of options they might be trying to type. The example allows you to enter the name of a U.S. state, and as you start typing suggestions will appear as to which state you are trying to enter.

You can see what the input will look like in Figure 12-22.

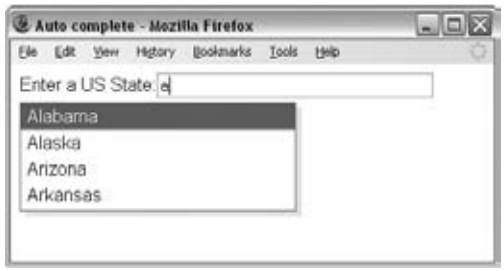

Figure 12-22

To start with in this example (`ch12_eg24.html`), you include the three core JavaScript files:

```
<script type="text/javascript" src="scripts/yui/yahoo/yahoo.js"></script>
<script type="text/javascript" src="scripts/yui/event/event.js"></script>
<script type="text/javascript" src="scripts/yui/dom/dom.js"></script>
```

Then you add the animation and data source library utilities.

```
<script type="text/javascript"
    src="scripts/yui/animation/animation.js"></script>
<script type="text/javascript"
    src="scripts/yui/datasource/datasource.js"></script>
```

Finally, you add in the `autocomplete.js` JavaScript file:

```
<script type="text/javascript"
    src="scripts/yui/automcomplete/automcomplete.js"></script>
```

Then, in the body of the page, you add the text input and a `<div>` that will contain suggestions of what you are trying to type in.

```
Select a US state:
<input id="myInput" type="text">
<div id="myContainer"></div>
```

Next, a JavaScript array is created with all of the possibilities that someone might be trying to enter.

```
<script type="text/javascript">
YAHOO.example.arrayStates = [
    "Alabama",
    "Alaska",
    "Arizona",
    "Arkansas",
    "California",
    "Colorado",
    // other states go here
];
</script>
```

Finally, the JavaScript is added to the page that ties the text input form control to the array, and calls the Auto-Complete function so that the suggestions are made as users enter their cursors into the text input.

```
<script type="text/javascript">
YAHOO.example.ACJSArray = new function() {
    // Instantiate first JS Array DataSource
    this.oACDS = new YAHOO.widget.DS_JSArray(YAHOO.example.statesArray);
    // Instantiate first AutoComplete
    this.oAutoComp =
      new YAHOO.widget.AutoComplete('statesinput','myContainer',
        this.oACDS);
    this.oAutoComp.prehighlightClassName = "yui-ac-prehighlight";
    this.oAutoComp.typeAhead = true;
    this.oAutoComp.useShadow = true;
    this.oAutoComp.minQueryLength = 0;
    this.oAutoComp.textboxFocusEvent.subscribe(function(){
        var sInputValue = YAHOO.util.Dom.get('statesinput').value;
        if(sInputValue.length === 0) {
            var oSelf = this;
            setTimeout(function(){oSelf.sendQuery(sInputValue);},0);
        }
    });

};
</script>
```

Again, you can see that by following a simple example made available with a JavaScript toolkit, you can significantly enhance the usability or functionality of your page (without the need to write all of the code to do the job from scratch).

There are many more JavaScript libraries on the Web, each of which has different functionality, and each of which is continually being developed and refined, so it is worthwhile taking some time to look at the different libraries that are available, and checking in with your favorites every so often to see how they have been updated.

Summary

In this chapter, you have seen many uses for JavaScript, and you should now have a better understanding of how to apply it. With the help of these scripts you should now be able to use these and other scripts in your page. You should also have an idea of how you can tailor or even write your own scripts.

You have seen how you can help a user fill in a form correctly by providing validation. For example, you might check to make sure required fields have something in them or that an e-mail address follows the expected pattern. This saves users time by telling them what they have to do before a page gets sent to a server, processed, and then returned with errors. The validation examples highlight the access the DOM gives to document content, so that you can perform operations on the values users provide.

You also saw how the DOM can help make a form generally more usable by putting the focus on appropriate parts of the form and manipulating the text users have entered, by removing or replacing certain characters.

Finally, you took a look at some popular JavaScript libraries: Scriptaculous, JQuery, MochiKit, and the Yahoo User Interface Library. JavaScript libraries offer sophisticated functionality that you can easily drop into your pages with just a few lines of code, and are the basis for many of the scripts available on the web that allow you to add complex functionality to your site with minimum effort.

Exercises

There is only one exercise for this chapter because it is quite a long one. The answers to all the exercises are in Appendix A.

1. Your task is to create a validation function for the competition form in Figure 12-23.

   The function should check that the user has done the following things:

   - Entered his or her name
   - Provided a valid e-mail address
   - Selected one of the radio buttons as an answer to the question
   - Given an answer for the tiebreaker question, which is no more than 20 words

These should be in the order that the controls appear on the form.

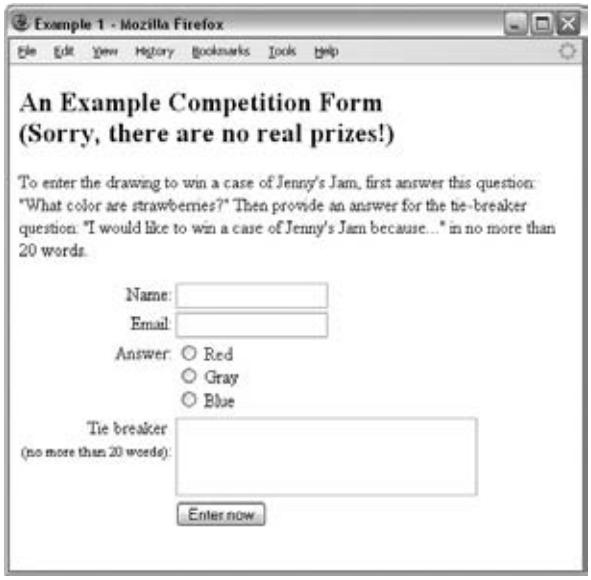

Figure 12-23

Here is the code for the form:

```
<form name="frmCompetition" action="competition.aspx" method="post"
onsubmit="return validate(this);">
<h2>An Example Competition Form <br />(Sorry, there are no real
prizes!)</h2>
<p> To enter the drawing to win a case of Jenny's Jam, first answer
this question: "What color are strawberries?" Then provide an answer for
the tie-breaker question: "I would like to win a case of Jenny's Jam
because..." in no more than 20 words.</p>
<table>
  <tr>
    <td class="formTitle">Name: </td>
    <td><input type="text" name="txtName" size="18" /></td>
  </tr>
  <tr>
    <td class="formTitle">Email: </td>
    <td><input type="text" name="txtEmail" size="18" /></td>
  </tr>
  <tr>
    <td class="formTitle">Answer: </td>
    <td><input type="radio" name="radAnswer" value="Red" /> Red<br />
          <input type="radio" name="radAnswer" value="Gray" /> Gray<br />
          <input type="radio" name="radAnswer" value="Blue" /> Blue
    </td>
    </tr>
   <tr>
     <td class="formTitle">Tie breaker <br/ ><small>(no more than 20 words)
     </small>: </td>
     <td><textarea name="txtTieBreaker" cols="30" rows="3"/></textarea>
     </td>
   </tr>
   <tr>
     <td class="formTitle"></td>
     <td><input type="submit" value="Enter now" /></td>
   </tr>
</table>
</form>
```