8

Windows and Frames

Until now, the pages you have been looking at have just been single pages. However, many web applications use frames to split up the browser's window, much as panes of glass split up a real window. It's quite possible that you'll want to build web sites that make use of such frames. The good news is that JavaScript enables the manipulation of frames and allows functions and variables you create in one frame to be used from another frame. One advantage of this is that you can keep common variables and functions in one place but use them from many places. This chapter starts by looking at how you can script across such frames.

A number of other good reasons exist for wanting to access variables and functions in another frame. Two important reasons are to make your code *modular* and to gain the ability to maintain information between pages.

What does *modular* mean? In other programming languages, like C, C++, or Visual Basic, you can create a module — an area to hold general functions and variables — and reuse it from different places in your program. When using frames, you can put all of your general functions and variables into one area, such as the top frame, which you can think of as your code module. Then you can call the functions repeatedly from different pages and different frames.

If you put the general functions and variables in a page that defines the frames that it contains (that is, a frameset-defining page), then if you need to make changes to the pages inside the frames, any variables defined in the frameset page will retain their value. This provides a very useful means of holding information even when the user is navigating your web site. A further advantage is that any functions defined in the frameset-defining page can be called by subsequent pages and have to be loaded into the browser only once, making your page's loading faster.

The second subject of this chapter is how you can open up and manipulate new browser windows. There are plenty of good uses for new windows. For example, you may wish to open up an *external* web site in a new window from your web site, but still leave your web site open for the user. *External* here means a web site created and maintained by another person or company. Let's say you have a web site about cars — well, you may wish to have a link to external sites, such as manufacturing web sites (for example, that of Ford or General Motors). Perhaps even more useful is using small windows as dialog boxes, which you can use to obtain information from the user. Just as you can script between frames, you can do similar things between certain windows. You find out how later in the chapter, but let's start by looking at scripting between frames.

Frames and the window Object

Frames are a means of splitting up the browser window into various panes, into which you can then load different HTML documents. The frames are defined in a frameset-defining page by the `<frameset/>` and `<frame/>` elements. The `<frameset/>` element contains `<frame/>` elements and specifies how the frames should look on the page. The `<frame/>` elements are then used to specify each frame and to include the required documents in the page.

You saw in Chapter 6 that the `window` object represents the browser's frame on your page or document. If you have a page with no frames, there will be just one `window` object. However, if you have more than one frame, there will be one `window` object for each frame. Except for the very top-level window of a frameset, each `window` object is contained inside another.

The easiest way to demonstrate this is through an example in which you create three frames, a top frame with two frames inside it.

Try It Out Multiple Frames

For this multi-frame example, you'll need to create three HTML files. The first is the frameset-defining page.

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Chapter 8: Example 1</title>
</head>
<frameset row="50%, *" id="topWindow">
    <frame name="upperWindow" src="ch08_examp1_upper.htm" />
    <frame name="lowerWindow" src="ch08_examp1_lower.htm" />
</frameset>
</html>
```

Save this as `ch08_examp1.htm`. Note that the `src` attributes for the two `<frame />` elements in this page are `ch08_examp1_upper.htm` and `ch08_examp1_lower.htm`. You will create these next.

```
<!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">
<head>
    <title>Chapter 8: Example 1 Upper Frame</title>
    <script type="text/javascript">
    function window_onload()
    {
        alert("The name of the upper frame's window object is " +
            window.name);

        alert("The location of upperWindow's parent is " +
            window.parent.location.href);
    }
    </script>
</head>
<body onload="window_onload()">
    <p>
        Upper Frame
    </p>
</body>
</html>
```

The preceding code block is the source page for the top frame with the name `upperWindow` and needs to be saved as `ch08_examp1_upper.htm`. The final page is very similar to it:

```
<!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">
<head>
    <title>Chapter 8: Example 1 Lower Frame</title>
    <script type="text/javascript">
    function window_onload()
    {
        alert("The name of the lower frame's window object is " +
            window.name);

        alert("The location of lowerWindow's parent is " +
            window.parent.location.href);
    }
    </script>
</head>
<body onload="window_onload()">
    <p>
        Lower Frame
    </p>
</body>
</html>
```

This is the source page for the lower frame; save it as `ch08_examp1_lower.htm`.

These three pages fit together so that `ch08_examp1_upper.htm` and `ch08_examp1_lower.htm` are contained within the `ch08_examp1.htm` page.

When you load them into the browser, you have three `window` objects. One is the *parent* `window` object and contains the file `ch08_examp1.htm`, and two are *child* `window` objects, containing the files `ch08_examp1_upper.htm` and `ch08_examp1_lower.htm`. The two child `window` objects are contained within the parent `window`, as shown in Figure 8-1.

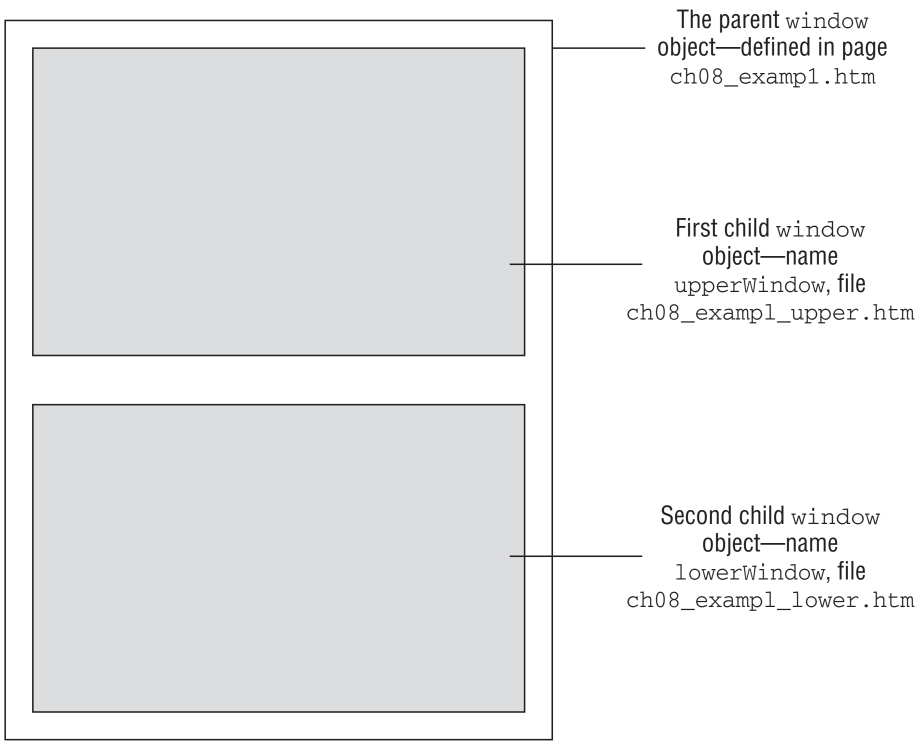
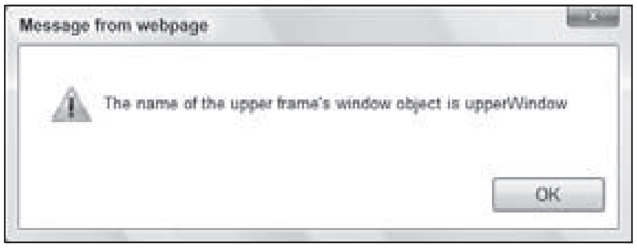

The parent window object—defined in page ch08_examp1.htm
First child window object—name upperWindow, file ch08_exampl_upper.htm
Second child window object—name lowerWindow, file ch08_exampl_lower.htm

Figure 8-1

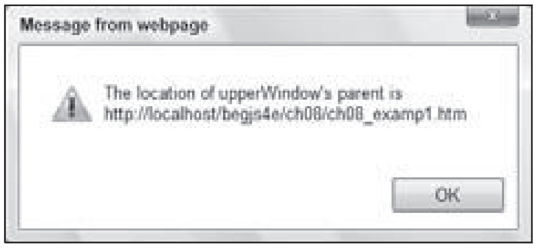
If any of the frames had frames contained inside them, these would have `window` objects that were children of the `window` object of that frame.

When you load `ch08_examp1.htm` into your browser, you'll see a series of four message boxes, as shown in Figures 8-2 through 8-5. These are making use of the `window` object's properties to gain information and demonstrate the `window` object's place in the hierarchy.



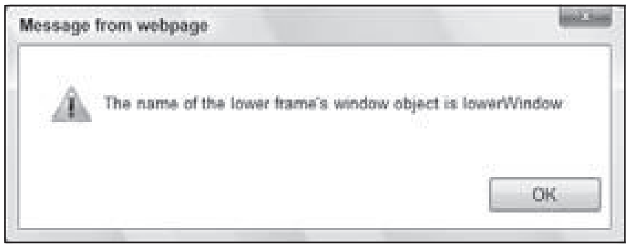
The name of the upper frame's window object is upperWindow

Figure 8-2



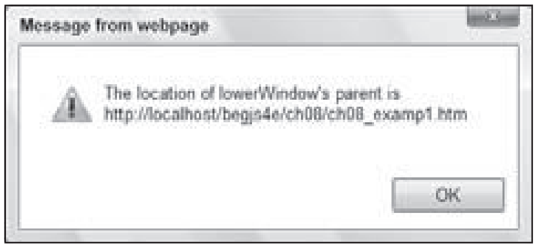
The location of upperWindow's parent is
http://localhost/begjs4e/ch08/ch08_examp1.htm

Figure 8-3


Message from webpage

The name of the lower frame's window object is lowerWindow

Figure 8-4


Message from webpage

The location of lowerWindow's parent is
http://localhost/begjs4e/ch08/ch08_examp1.htm

Figure 8-5

The paths in Figures 8-3 and 8-5 will vary depending upon where the files are stored on your computer.

Look at the frameset-defining page, starting with ch08_examp1.htm, as shown in the following snippet:

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Chapter 8: Example 1</title>
</head>
<frameset rows="50%, *" id="topWindow">
    <frame name="upperWindow" src="ch08_examp1_upper.htm" />
    <frame name="lowerWindow" src="ch08_examp1_lower.htm" />
</frameset>
</html>
```

The frameset is defined with the `<frameset />` element. You use two attributes: `rows` and `id`. The `rows` attribute takes the value `"50%,*"` meaning that the first frame should take up half of the height of the window, and the second frame should take up the rest of the room. The `id` attribute is used to give a name that you can use to reference the page.

The two child windows are created using `<frame />` elements; each of which contains a `name` attribute by which the `window` objects will be known and a `src` attribute of the page that will be loaded into the newly created windows.

Let's take a look at the `ch08_examp1_upper.htm` file next. In the `<body />` element, you attach the `window_onload()` function to the `window` object's `onload` event handler. This event handler is called when the browser has finished loading the window, the document inside the window, and all the objects within the document. It's a very useful place to put initialization code or code that needs to change things after the page has loaded but before control passes back to the user.

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

This function is defined in a script block in the head of the page as follows:

```
function window_onload()
{

    alert("The name of the upper frame's window object is " + window.name);
    alert("The location of UpperWindow's parent is " +

        window.parent.location.href);
}
```

The `window_onload()` function makes use of two properties of the `window` object for the frame that the page is loaded in: its `name` and `parent` properties. The `name` property is self-explanatory — it's the name you defined in the frameset page. In this case, the name is `upperWindow`.

The second property, the `parent` property, is very useful. It gives you access to the `window` object of the frame's parent. This means you can access all of the parent `window` object's properties and methods. Through these, you can access the `document` within the parent `window` as well as any other frames defined by the parent. Here, you display a message box giving details of the parent frame's file name or URL by using the `href` property of the `location` object (which itself is a property of the `window` object).

The code for `ch08_examp1_lower.htm` is identical to the code for `ch08_examp1_upper.htm`, but with different results because you are accessing a different `window` object. The `name` of the `window` object this time is `lowerWindow`. However, it shares the same parent `window` as `upperWindow`, and so when you access the `parent` property of the `window` object, you get a reference to the same `window` object as in `upperWindow`. The message box demonstrates this by displaying the file name/URL or `href` property, and this matches the file name of the page displayed in the `upperWindow` frame.

The order of display of messages may vary among different types of browsers and even different operating systems. This may not be important here, but there will be times when the order in which events fire is important and affects how your code works. It's an incompatibility that's worth noting and watching out for in your own programs.

Coding Between Frames

You've seen that each frame exists as a different window and gets its own `window` object. In addition, you saw that you can access the `window` object of a frameset-defining page from any of the frame pages it specifies, by using the `window` object's `parent` property. When you have a reference to the parent window's `window` object, you can access its properties and methods in the same way that you access the `window` object of the current page. In addition, you have access to all the JavaScript variables and functions defined in that page.

Try It Out Using the Frameset Page as a Module

Let’s look at a more complex example, wherein you use the top frame to keep track of pages as the user navigates the web site. You’re creating five pages in this example, but don’t panic; four of them are almost identical. The first page that needs to be created is the frameset-defining page.

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Chapter 8: Example 2</title>
    <script type="text/javascript">
    var pagesVisited = new Array();
    function returnPagesVisited()
    {
        var returnValue = "So far you have visited the following pages\n";
        var pageVisitedIndex;
        var numberOfPagesVisited = pagesVisited.length;
        for (pageVisitedIndex = 0; pageVisitedIndex < numberOfPagesVisited;
            pageVisitedIndex++)
        {
            returnValue = returnValue + pagesVisited[pageVisitedIndex] + "\n";
        }
        return returnValue;
    }

    function addPage(fileName)
    {
        var fileNameStart = fileName.lastIndexOf("/") + 1;
        fileName = fileName.substr(fileNameStart);
        pagesVisited[pagesVisited.length] = fileName;

        return true;
    }
    </script>
</head>
<frameset cols="50%,*">
    <frame name="fraLeft" src="ch08_examp2_a.htm">
    <frame name="fraRight" src="ch08_examp2_b.htm">
</frameset>
</html>
```

Save this page as `ch08_examp2.htm`.

Notice that the two frames have the `src` attributes initialized as `ch08_examp2_a.htm` and `ch08_examp2_b.htm`. However, you also need to create `ch08_examp2_c.htm` and `ch08_examp2_d.htm` because you will be allowing the user to choose the page loaded into each frame from these four pages. You’ll create the `page_a.htm` page first, as shown in the following:

```
<!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">
```

```
<head>
    <title>Chapter 8: Example 2 Page A</title>
    <script type="text/javascript">
    function btnShowVisited_onclick()
    {
        document.form1.txtaPagesVisited.value =
            window.parent.returnPagesVisited();
    }
    </script>
</head>
<body onload="window.parent.addPage(window.location.href)">
    <h2>This is Page A</h2>
    <p>
        <a href="ch08_examp2_a.htm">Page A</a>
        <a href="ch08_examp2_b.htm">Page B</a>
        <a href="ch08_examp2_c.htm">Page C</a>
        <a href="ch08_examp2_d.htm">Page D</a>
    </p>
<form name="form1" action="">
        <textarea rows="10" cols="35" name="txtaPagesVisited"></textarea>
        <br />
        <input type="button" value="List Pages Visited" name="btnShowVisited"
            onclick="btnShowVisited_onclick()" />
    </form>
</body>
</html>
```

Save this page as `ch08_examp2_a.htm`.

The other three pages are identical to `ch08_examp2_a.htm`, except for the page's title and the `<h2/>` element, so you can just cut and paste the text from `ch08_examp2_a.htm`. Change the HTML that displays the name of the page loaded to the following:

```
<h2>This is Page B</h2>
```

Then save this as `ch08_examp2_b.htm`.

Do the same again, to create the third page (page C):

```
<h2>This is Page C</h2>
```

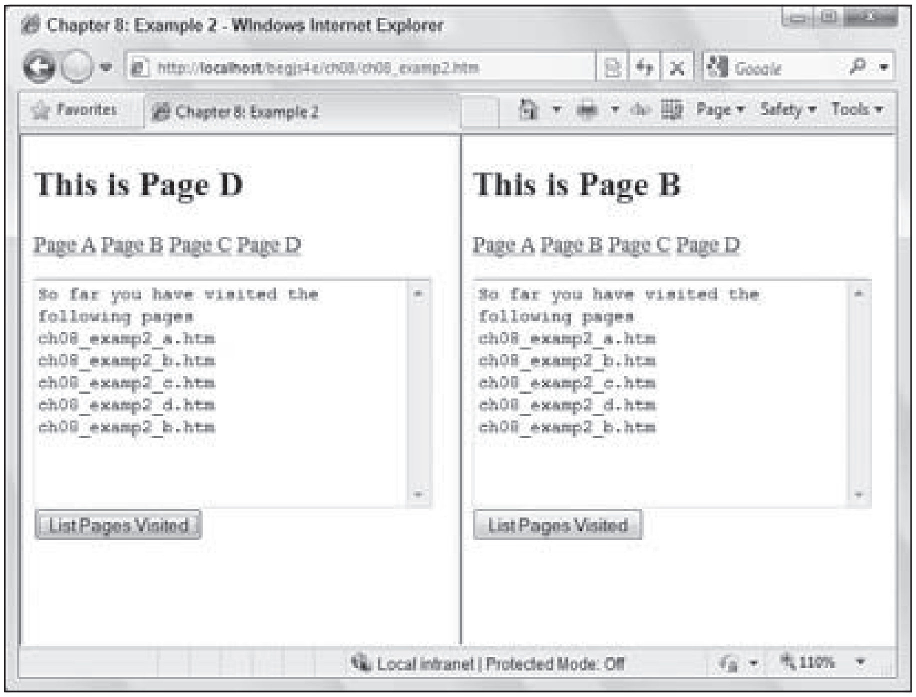
Save this as `ch08_examp2_c.htm`.

The final page is again a copy of `ch08_examp2_a.htm` except for the following lines:

```
<h2>This is Page D</h2>
```

Save this as `ch08_examp2_d.htm`.

Load `ch08_examp2.htm` into your browser and navigate to various pages by clicking the links. Then click the List Pages Visited button in the left-hand frame, and you should see a screen similar to the one shown in Figure 8-6.


Chapter 8: Example 2 - Windows Internet Explorer
http://localhost/begjs4e/ch08/ch08_examp2.htm
Favorites
Chapter 8: Example 2
Page
Safety
Tools
List Pages Visited
List Pages Visited
Local intranet | Protected Mode: Off
110%

This is Page D

Page A Page B Page C Page D

```
So far you have visited the
following pages
ch08_examp2_a.htm
ch08_examp2_b.htm
ch08_examp2_c.htm
ch08_examp2_d.htm
ch08_examp2_b.htm
```

This is Page B

Page A Page B Page C Page D

```
So far you have visited the
following pages
ch08_examp2_a.htm
ch08_examp2_b.htm
ch08_examp2_c.htm
ch08_examp2_d.htm
ch08_examp2_b.htm
```

Figure 8-6

Click the links in either frame to navigate to a new location. For example, click the Page C link in the right frame, then the Page D link in the left frame. Click the left frame's List Pages Visited button and you'll see that `ch08_examp2_c.htm` and `ch08_examp2_d.htm` have been added to the list.

Normally when a new page is loaded, any variables and their values in the previous page are lost, but when using frameset pages as modules, it does not matter which page is loaded into each frame — the top frame remains loaded and its variables keep their values. What you are seeing in this example is that, regardless of which page is loaded in each frame, some global variable in the top frame is keeping track of the pages that have been viewed and the top frame's variables and functions can be accessed by any page loaded into either frame.

There are restrictions when the pages you load into the frames are from external sources — more on this later in the chapter.

Let's first look at the JavaScript in `ch08_examp2.htm`, which is the frameset-defining page. The head of the page contains a script block. The first thing in this script block is the declaration of the `pagesVisited` variable, and set it to reference a new `Array` object. In the array, you'll be storing the file name of each page visited as the user navigates the site.

```
var pagesVisited = new Array();
```

You then have two functions. The first of the two functions, `returnPagesVisited()`, does what its name suggests — it returns a string containing a message and a list of each of the pages visited. It does this by looping through the `pagesVisited` array, building up the message string inside the `returnValue` variable, which is then returned to the calling function.

```
function returnPagesVisited()
{
    var returnValue = "So far you have visited the following pages\n";
    var pageVisitedIndex;
    var numberOfPagesVisited = pagesVisited.length;
    for (pageVisitedIndex = 0; pageVisitedIndex < numberOfPagesVisited;
        pageVisitedIndex++)
    {
        returnValue = returnValue + pagesVisited[pageVisitedIndex]
            + "\n";
    }
    return returnValue;
}
```

The second function, `addPage()`, adds the name of a page to the `pagesVisited` array.

```
function addPage(fileName)
{
    var fileNameStart = fileName.lastIndexOf("/") + 1;
    fileName = fileName.substr(fileNameStart);
    pagesVisited[pagesVisited.length] = fileName;

    return true;
}
```

The `fileName` parameter passed to this function is the full file name and path of the visited page, so you need to strip out the path to get just the file name. The format of the string will be something like `file:///D:/myDirectory/ch08_examp2_b.htm`, and you need just the bit after the last `/` character. So in the first line of code, you find the position of that character and add one to it because you want to start at the next character.

Then, using the `String`'s `substr()` method in the following line, you extract everything from character position `fileNameStart` right up to the end of the string. Remember that the `substr()` method accepts two parameters, namely the starting character you want and the length of the string you want to extract, but if the second parameter is missing, all characters from the start position to the end are extracted.

You then add the file name into the array, the `length` property of the array providing the next free index position.

You'll now turn to look collectively at the frame pages, namely `ch08_examp2_a.htm`, `ch08_examp2_b.htm`, `ch08_examp2_c.htm`, and `ch08_examp2_d.htm`. In each of these pages, you create a form called `form1`.

```
<form name="form1" action="">
    <textarea rows="10" cols="35" name="txtaPagesVisited"></textarea>
    <br />
    <input type="button" value="List Pages Visited"
        name="btnShowVisited" onclick="btnShowVisited_onclick()" />
</form>
```

This contains the `textarea` control that displays the list of visited pages, and a button the user can click to populate the `<textarea />` element.

When one of these pages is loaded, its name is put into the `pagesVisited` array defined in `ch08_examp2.htm` by the `window` object's `onload` event handler's being connected to the `addPage()` function that you also created in `ch08_examp2.htm`. You connect the code to the event handler in the `<body />` element of the page as follows:

```
<body onload="window.parent.addPage(window.location.href)">
```

Recall that all the functions you declare in a page are contained, like everything else in a page, inside the `window` object for that page; because the `window` object is the global object, you don't need to prefix the name of your variables or functions with `window`.

However, this time the function is not in the current page, but in the `ch08_examp2.htm` page. The `window` containing this page is the parent window to the window containing the current page. You need, therefore, to refer to the parent frame's `window` object using the `window` object's `parent` property. The code `window.parent` gives you a reference to the `window` object of `ch08_examp2.htm`. With this reference, you can now access the variables and functions contained in `ch08_examp2.htm`. Having stated which `window` object you are referencing, you just add the name of the function you are calling, in this instance the `addPage()` function. You pass this function the `location.href` string, which contains the full path and file name of the page, as the value for its one parameter.

As you saw earlier, the button on the page has its `onclick` event handler connected to a function called `btnShowVisited_onclick()`. This is defined in the head of the page.

```
function btnShowVisited_onclick()
{
    document.form1.txtaPagesVisited.value =
        window.parent.returnPagesVisited();
}
```

In this function, you call the parent `window` object's `returnPagesVisited()` function, which, as you saw earlier, returns a string containing a list of pages visited. The `value` property of the `textarea` object is set to this text.

That completes your look at the code in the frame pages, and as you can see, there's not much of it because you have placed all the general functions in the frameset page. Not only does this code reuse make for less typing, but it also means that all your functions are in one place. If there is a bug in a function, fixing the bug for one page also fixes it for all other pages that use the function. Of course, it only makes sense to put general functions in one place; functions that are specific to a page and are never used again outside it are best kept in that page.

Code Access Between Frames

You've just seen how a child window can access its parent window's variables and functions, but how can frames inside a frameset access each other?

You saw a simple example earlier in this chapter, so this time let's look at a much more complex example. When created, your page will look like the one shown in Figure 8-7.

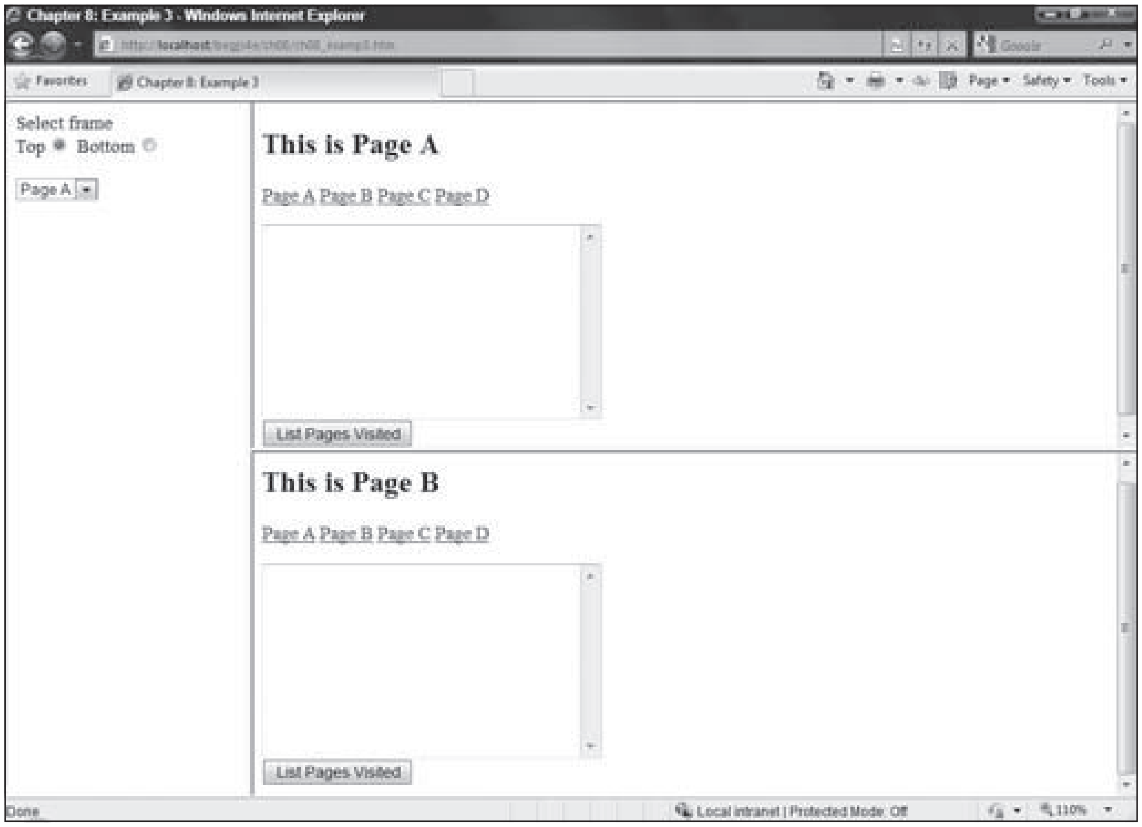

Chapter 8: Example 3 - Windows Internet Explorer
Select frame
Top
Bottom
Page A
This is Page A
Page A Page B Page C Page D
List Pages Visited
This is Page B
Page A Page B Page C Page D
List Pages Visited
Local intranet | Protected Mode: Off

Figure 8-7

A diagram of the frame layout is shown in Figure 8-8. The text labels indicate the names that each frame has been given in the `<frameset/>` and `<frame/>` elements, with the exception of the top frame, which is simply the window at the top of the frameset hierarchy.

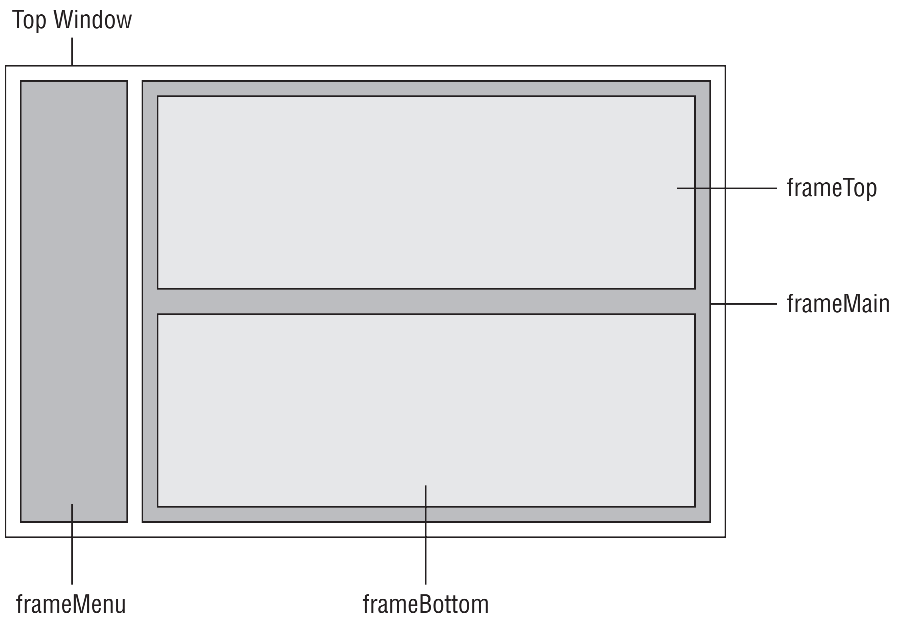


Figure 8-8

The easiest way to think of the hierarchy of such a frames-based web page is in terms of familial relationships, which can be shown in a family tree. If you represent your frameset like that, it looks something like the diagram in Figure 8-9.

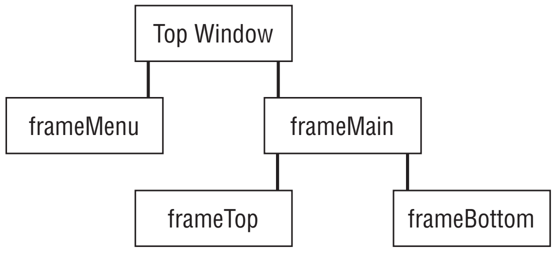


Figure 8-9

From the diagram you can see that `frameBottom`, the right-hand frame's bottom frame, has a parent frame called `frameMain`, which itself has a parent, the top window. Therefore, if you wanted to access a function in the top window from the `frameBottom` window, you would need to access `frameBottom`'s parent's parent's `window` object. You know that the `window` object has the `parent` property, which is a reference to the parent window of that `window` object. So let's use that and create the code to access a function, for example, called `myFunction()`, in the top window.

```
window.parent.parent.myFunction();
```

Let's break this down. The following code gets you a reference to the parent `window` object of the window in which the code is running.

```
window.parent
```

The code is in `frameBottom`, so `window.parent` will be `frameMain`. However, you want the top window, which is `frameMain`'s parent, so you add to the preceding code to make this:

```
window.parent.parent
```

Now you have a reference to the top window. Finally, you call `myFunction()` by adding that to the end of the expression.

```
window.parent.parent.myFunction();
```

What if you want to access the `window` object of `frameMenu` from code in `frameBottom`? Well, you have most of the code you need already. You saw that `window.parent.parent` gives you the top window, so now you want that window's child `window` object called `frameMenu`. You can get it in three ways, all with identical results.

You can use its index in the `frames` collection property of the `window` object as follows:

```
window.parent.parent.frames[0]
```

Alternatively, you can use its name in the `frames` collection like this:

```
window.parent.parent.frames["frameMenu"]
```

Finally, you can reference it directly by using its name as you can with any `window` object:

```
window.parent.parent.frameMenu
```

The third method is the easiest unless you don't know the name of a frame and need to access it by its index value in the `frames` collection, or are looping through each child frame in turn.

Since `window.parent.parent.frameMenu` gets you a reference to the `window` object associated with `frameMenu`, to access a function `myFunction()` or variable `myVariable`, you would just type one of these lines:

```
window.parent.parent.frameMenu.myFunction
```

or

```
window.parent.parent.frameMenu.myVariable
```

What if you want to access not a function or variable in a page within a frame, but a control on a form or even the links on that page? Well, let's imagine you want to access, from the `frameBottom` page, a control named `myControl`, on a form called `myForm` in the `frameMenu` page.

You found that `window.parent.parent.frameMenu` gives you the reference to `frameMenu`'s `window` object from `frameBottom`, but how do you reference a form there?

Basically, it's the same as how you access a form from the inside of the same page as the script, except that you need to reference not the `window` object of that page but the `window` object of `frameMenu`, the page you're interested in.

Normally you write `document.myForm.myControl.value`, with `window` being assumed since it is the global object. Strictly speaking, it's `window.document.myForm.myControl.value`.

Now that you're accessing another window, you just reference the window you want and then use the same code. So you need this code if you want to access the `value` property of `myControl` from `frameBottom`:

```
window.parent.parent.frameMenu.document.myForm.myControl.value
```

As you can see, references to other frames can get pretty long, and in this situation it's a very good idea to store the reference in a variable. For example, if you are accessing `myForm` a number of times, you could write this:

```
var myFormRef = window.parent.parent.frameMenu.document.myForm;
```

Having done that, you can now write

```
myFormRef.myControl.value;
```

rather than

```
window.parent.parent.frameMenu.document.myForm.myControl.value;
```

The top Property

Using the `parent` property can get a little tedious when you want to access the very top window from a frame quite low down in the hierarchy of frames and `window` objects. An alternative is the `window` object's `top` property. This returns a reference to the `window` object of the very top window in a frame hierarchy. In the current example, this is top window.

For instance, in the example you just saw, this code:

```
window.parent.parent.frameMenu.document.myForm.myControl.value;
```

could be written like this:

```
window.top.frameMenu.document.myForm.myControl.value;
```

Although, because the `window` is a global object, you could shorten that to just this:

```
top.frameMenu.document.myForm.myControl.value;
```

So when should you use `top` rather than `parent`, or vice versa?

Both properties have advantages and disadvantages. The `parent` property enables you to specify `window` objects relative to the current window. The window above this window is `window.parent`, its parent is `window.parent.parent`, and so on. The `top` property is much more generic; `top` is always the very top window regardless of the frameset layout being used. There will always be a `top,` but there's not necessarily going to always be a `parent.parent`. If you put all your global functions and variables that you want accessible from any page in the frameset in the very top window, `window.top` will always be valid regardless of changes to framesets beneath it, whereas the `parent` property is dependent on the frameset structure above it. However, if someone else loads your web site inside a frameset page of his own, then suddenly the `top` window is not yours but his, and `window.top` is no longer valid. You can't win, or can you?

One trick is to check to see whether the `top` window contains your page; if it doesn't, reload the `top` page again and specify that your `top` page is the one to be loaded. For example, check to see that the file name of the `top` page actually matches the name you expect. The `window.top.location.href` will give you the name and path — if they don't match what you want, use `window.top.location .replace("myPagename.htm")` to load the correct `top` page. However, as you'll see later, this will cause problems if someone else is loading your page into a frameset they have created — this is where something called the *same-origin policy* applies. More on this later in the chapter.

Try It Out Scripting Frames

Let's put all you've learned about frames and scripting into an example based on the frameset you last looked at in `ch08_examp2.htm`. You're going to be reusing a lot of the pages and code from the previous example in this chapter.

The first page you're creating is the top window page. The highlighted lines of code show changes made to `ch08_examp2.htm`.

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

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

```
<head>
    <title>Chapter 8: Example 3</title>
    <script type="text/javascript">
    var pagesVisited = new Array();
    function returnPagesVisited()
    {
        var returnValue = "So far you have visited the following pages\n";
        var pageVisitedIndex;
        var numberOfPagesVisited = pagesVisited.length;
        for (pageVisitedIndex = 0; pageVisitedIndex < numberOfPagesVisited;
            pageVisitedIndex++)
        {
            returnValue = returnValue + pagesVisited[pageVisitedIndex] + "\n";
        }
        return returnValue;
    }

    function addPage(fileName)
    {
        var fileNameStart = fileName.lastIndexOf("/") + 1;
        fileName = fileName.substr(fileNameStart);
        pagesVisited[pagesVisited.length] = fileName;

        return true;
    }
    </script>
</head>
<frameset cols="200,*">
    <frame name="frameMenu" src="ch08_examp3_menu.htm">
    <frame name="frameMain" src="ch08_examp3_main.htm">
</frameset>
</html>
```

As you can see, you've reused a lot of the code from `ch08_examp2.htm`, so you can cut and paste the script block from there. Only the different code lines are highlighted. Save this page as `ch08_examp3.htm`.

Next, create the page that will be loaded into `frameMenu`, namely `ch08_examp3_menu.htm`.

```
<!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">
<head>
    <title>Chapter 8: Example 3 Menu</title>
    <script type="text/javascript">
    function choosePage_onchange()
    {
        var choosePage = document.form1.choosePage;
        var windowobject;
        if (document.form1.radFrame[0].checked == true)
        {
            windowobject = window.parent.frameMain.frameTop;
        }
        else
        {
            windowobject = window.parent.frameMain.frameBottom;
        }
        windowobject.location.href =
            choosePage.options[choosePage.selectedIndex].value;
        return true;
    }
    </script>
</head>
<body>
    <form name="form1" action="">
        Select frame
        <br />
        Top
        <input name="radFrame" checked="checked" type="radio" />
        Bottom
        <input name="radFrame" type="radio" />
        <br />
        <br />
        <select name="choosePage" onchange="choosePage_onchange()">
            <option value="ch08_examp3_a.htm">Page A</option>
            <option value="ch08_examp3_b.htm">Page B</option>
            <option value="ch08_examp3_c.htm">Page C</option>
            <option value="ch08_examp3_d.htm">Page D</option>
        </select>
    </form>
</body>
</html>
```

Save this as `ch08_examp3_menu.htm`.

The `frameMain` frame contains a page that is simply a frameset for the `frameTop` and `frameBottom` pages.

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Chapter 8: Example 3 Main</title>
</head>
<frameset rows="50%,*">
   <frame name="frameTop" src="ch08_examp3_a.htm">
   <frame name="frameBottom" src="ch08_examp3_b.htm">
</frameset>
</html>
```

Save this as `ch08_examp3_main.htm`.

The next four pages are mainly copies of the four pages — `ch08_examp2_a.htm`, `ch08_examp2_b.htm`, `ch08_examp2_c.htm`, and `ch08_examp2_d.htm` — from example two. You'll need to make a few changes, as highlighted in the following code. (Again, all the pages are identical except for the text shown in the page, so only modifications to `ch08_examp2_a.htm` are shown. Amend the rest in a similar way.)

```
<!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">
```

```
<head>
    <title>Chapter 8: Example 3 Page A</title>
    <script type="text/javascript">
    function btnShowVisited_onclick()
    {
        document.form1.txtaPagesVisited.value =
            window.top.returnPagesVisited();
    }

    function setFrameAndPageControls(linkIndex)
    {
        var formobject = window.parent.parent.frameMenu.document.form1;
        formobject.choosePage.selectedIndex = linkIndex;

        if (window.parent.frameTop == window.self)
        {
            formobject.radFrame[0].checked = true;
        }
        else
        {
            formobject.radFrame[1].checked = true;
        }

        return true;
    }
    </script>
</head>
<body onload="window.top.addPage(window.location.href)">
    <h2>This is Page A</h2>
    <p>
        <a href="ch08_examp3_a.htm" name="pageALink"
            onclick="return setFrameAndPageControls(0)">Page A</a>
        <a href="ch08_examp3_b.htm" name="pageBLink"
            onclick="return setFrameAndPageControls(1)">Page B</a>
        <a href="ch08_examp3_c.htm" name="pageCLink"
            onclick="return setFrameAndPageControls(2)">Page C</a>
        <a href="ch08_examp3_d.htm" name="pageDLink"
            onclick="return setFrameAndPageControls(3)">Page D</a>
    </p>
    <form name="form1" action="">
        <textarea rows="10" cols="35" name="txtaPagesVisited"></textarea>
        <br />
        <input type="button" value="List Pages Visited" name="btnShowVisited"
            onclick="btnShowVisited_onclick()" />
    </form>
</body>

</html>
```

Save the pages as `ch08_examp3_a.htm`, `ch08_examp3_b.htm`, `ch08_examp3_c.htm`, and `ch08_examp3_d.htm`.

Load `ch08_examp3.htm` into your browser, and you'll see a screen similar to the one shown in Figure 8-7.

The radio buttons allow the user to determine which frame he wants to navigate to a new page. When he changes the currently selected page in the drop-down list, that page is loaded into the frame selected by the radio buttons.

If you navigate using the links in the pages inside the `frameTop` and `frameBottom` frames, you'll notice that the selected frame radio buttons and the drop-down list in `frameMenu` on the left will be automatically updated to the page and frame just navigated to. Note that as the example stands, if the user loads `ch08_examp3_a.htm` into a frame the select list doesn't allow it to load the same page in the other frame. You could improve on this example by adding a button that loads the currently selected page into the chosen frame.

The List Pages Visited buttons display a list of visited pages, as they did in the previous example.

You've already seen how the code defining the top window in `ch08_examp3.htm` works, as it is very similar to the previous example. However, look quickly at the `<frameset/>` element, where, as you can see, the names of the windows are defined in the names of the `<frame/>` elements.

```
<frameset cols="200,*">
    <frame name="frameMenu" src="ch08_examp3_menu.htm">
    <frame name="frameMain" src="ch08_examp3_main.htm">
</frameset>
```

Notice also that the `cols` attribute of the `<frameset/>` element is set to `"200,*"`. This means that the first frame will occupy a column 200 pixels wide, and the other frame will occupy a column taking up the remaining space.

Let's look in more detail at the `frameMenu` frame containing `ch08_examp3_menu.htm`. At the top of the page, you have your main script block. This contains the function `choosePage_onchange()`, which is connected to the `onchange` event handler of the select box lower down on the page. The select box has `options` containing the various page URLs.

The function starts by defining two variables. One of these, `choosePage`, is a shortcut reference to the `choosePage Select` object further down the page.

```
function choosePage_onchange()
{
    var choosePage = document.form1.choosePage;
    var windowobject;
```

The `if...else` statement then sets your variable `windowobject` to reference the `window` object of whichever frame the user has chosen in the `radFrame` radio button group.

```
    if (document.form1.radFrame[0].checked == true)
    {
        windowobject = window.parent.frameMain.fraTop;
    }
    else
    {
        windowobject = window.parent.frameMain.fraBottom;
    }
```

As you saw earlier, it's just a matter of following through the references, so `window.parent` gets you a reference to the parent `window` object. In this case, `window.top` would have done the same thing. Then `window.parent.frameMain` gets you a reference to the `window` object of the `frameMain` frame. Finally, depending on which frame you want to navigate in, you reference the `frameTop` or `frameBottom` `window` objects contained within `frameMain`, using `window.parent.frameMain.frameTop` or `window.parent.frameMain.frameBottom`.

Now that you have a reference to the `window` object of the frame in which you want to navigate, you can go ahead and change its `location.href` property to the value of the selected drop-down list item, causing the frame to load that page.

```
        windowobject.location.href = 

            choosePage.options[choosePage.selectedIndex].value;
        return true;
    }
```

As you saw before, `ch08_examp3_main.htm` is simply a frameset-defining page for `frameTop` and `frameBottom`. Let's now look at the pages you're actually loading into `frameTop` and `frameBottom`. Because they are all the same, you'll look only at `ch08_examp3_a.htm`.

Let's start by looking at the top script block. This contains two functions, `btnShowVisited_onclick()` and `setFrameAndPageControls()`. You saw the function `btnShowVisited_onclick()` in the previous example.

```
    function btnShowVisited_onclick()
    {
        document.form1.txtaPagesVisited.value = window.top.returnPagesVisited();
    }
```

However, because the frameset layout has changed, you do need to change the code. Whereas previously the `returnPagesVisited()` function was in the parent window, it's now moved to the top window. As you can see, all you need to do is change the reference from `window.parent.returnPagesVisited();` to `window.top.returnPagesVisited();`.

As it happens, in the previous example the `parent` window was also the `top` window, so if you had written your code in this way in the first place, there would have been no need for changes here. It's often quite a good idea to keep all your general functions in the top frameset page. That way all your references can be `window.top`, even if the frameset layout is later changed.

The new function in this page is `setFrameAndPageControls()`, which is connected to the `onclick` event handler of the links defined lower down on the page. This function's purpose is to make sure that if the user navigates to a different page using the links rather than the controls in the `frameMenu` window, those controls will be updated to reflect what the user has done.

The first thing you do is set the `formobject` variable to reference the `form1` in the `frameMenu` page, as follows:

```
    function setFrameAndPageControls(linkIndex)
    {
        var formobject = window.parent.parent.frameMenu.document.form1;
```

Let's break this down.

```
    window.parent
```

gets you a reference to the `frameMain` `window` object. Moving up the hierarchy, you use the following code to get a reference to the `window` object of the top window:

```
    window.parent.parent
```

Yes, you're right. You could have used `window.top` instead, and this would have been a better way to do it. We're doing it the long way here just to demonstrate how the hierarchy works.

Now you move down the hierarchy, but on the other side of your tree diagram, to reference the `frameMenu`'s `window` object.

```
window.parent.parent.frameMenu
```

Finally, you are interested only in the form and its controls, so you reference that object like this:

```
window.parent.parent.frameMenu.document.form1
```

Now that you have a reference to the form, you can use it just as you would if this were code in `frameMenu` itself.

The function's parameter `linkIndex` tells you which of the four links was clicked, and you use this value in the next line of the function's code to set which of the options is selected in the drop-down list box on `frameMenu`'s form.

```
formobject.choosePage.selectedIndex = linkIndex;
```

The `if...else` statement is where you set the `frameMenu`'s radio button group `radFrame` to the frame the user just clicked on, but how can you tell which frame this is?

```
    if (window.parent.frameTop == window.self)

    {
        formobject.radFrame[0].checked = true
    }
    else
    {
        formobject.radFrame[1].checked = true
    }
```

You check to see whether the current `window` object is the same as the `window` object for `frameTop`. You do this using the `self` property of the `window` object, which returns a reference to the current `window` object, and `window.parent.frameTop`, which returns a reference to `frameTop`'s `window` object. If one is equal to the other, you know that they are the same thing and that the current window is `frameTop`. If that's the case, the `radFrame` radio group in the `frameMenu` frame has its first radio button checked. Otherwise, you check the other radio button for `frameBottom`.

The last thing you do in the function is return `true`. Remember that this function is connected to an `A` object, so returning `false` cancels the link's action, and `true` allows it to continue, which is what you want.

```
    return true;
}
```

Scripting IFrames

Inline frames (iframes), introduced by Microsoft in Internet Explorer (IE) 3, became a part of the HTML standard in HTML 4. They're a unique element in that you can add a frame to a web page without using a frameset, and they're much simpler to add to the page because of it. For example:

```
<iframe name="myIFrame" src="child_frame.htm" />
```

This HTML adds a frame with the name `myIFrame` to the page, which loads the `child_frame.htm` file. As you may guess, this simplicity carries over to your JavaScript. Accessing the iframe's `document` object of the page loaded in it is straightforward. For example:

```
window.myIFrame.document.bgColor = "red";
```

As you can see, it's very similar to conventional frames within a frameset (you can also use the `frames` collection like `window.frames["myIFrame"]`). Accessing the parent `window` from within the iframe is also familiar; use the `parent` property. For example:

```
window.parent.document.bgColor = "yellow";
```

Opening New Windows

So far in this chapter, you have been looking at frames and scripting between them. In this section, you'll change direction slightly and look at how you can open up additional browser windows.

Why would you want to bother opening up new windows? Well, they can be useful in all sorts of different situations, such as the following:

- You might want a page of links to web sites, in which clicking a link opens up a new window with that web site in it.
- Additional windows can be useful for displaying information. For example, if you had a page with products on it, the user could click a product image to bring up a new small window listing the details of that product. This can be less intrusive than navigating the existing window to a new page with product details, and then requiring the user to click Back to return to the list of products. You'll be creating an example demonstrating this later in this chapter.
- Dialog windows can be very useful for obtaining information from users, although overuse may annoy them.

The latest versions of all modern browsers include a pop-up blocking feature. By default, new windows created automatically when a page loads are usually blocked. However, windows that open only when the user must perform an action, for example clicking a link or button, are not normally blocked by default, but the user may change the browser settings to block them.

Opening a New Browser Window

The `window` object has an `open()` method, which opens up a new window. It accepts three parameters, although the third is optional, and it returns a reference to the `window` object of the new browser window.

The first parameter of the `open()` method is the URL of the page that you want to open in the new window. However, you can pass an empty string for this parameter and get a blank page and then use the `document.write()` method to insert HTML into the new window dynamically. You'll see an example of this later in the chapter.

The second parameter is the name you want to allocate to the new window. This is not the name you use for scripting, but instead is used for the `target` attribute of things such as hyperlinks and forms. For example, if you set this parameter to `myWindow` and set the `target` attribute of a hyperlink on the original page to the same value (like in the following code example), clicking that hyperlink will cause the hyperlink to act on the new window opened.

```
<a href="test3.htm" target="myWindow">Test3.htm</a>
```

This means that `test3.htm` loads into the new window and not the current window when the user clicks the link. The same applies to the `<form />` element's `target` attribute. In this case, if a form is submitted from the original window, the response from the server can be made to appear in the new window.

When a new window is opened, it is opened (by default) with a certain set of properties, such as `width` and `height`, and with the normal browser-window features. Browser-window features include things such as a location entry field and a menu bar with navigation buttons.

The third parameter of the `open()` method can be used to specify values for the `height` and `width` properties. Also, because by default most of the browser window's features are switched off, you can switch them back on using the third parameter of the `open()` method. You'll look at browser features in more detail shortly.

Let's first look at an example of the code you need to open a basic window. You'll name this window `myWindow` and give it a `width` and `height` of 250 pixels. You want the new window to open with the `test2.htm` page inside.

```
var newWindow = window.open("test2.htm","myWindow","width=250,height=250");
```

You can see that `test2.htm` has been passed as the first parameter; that is the URL of the page you want to open. The window is named `myWindow` in the second parameter. In the third parameter, you've set the `width` and `height` properties to `250`.

Also notice that you've set the variable `newWindow` to the return value returned by the `open()` method, which is a reference to the `window` object of the newly opened window. You can now use `newWindow` to manipulate the new window and gain access to the `document` contained inside it using the `newWindow .document` property. You can do everything with this reference that you did when dealing with frames and their `window` objects. For example, if you wanted to change the background color of the `document` contained inside the new window, you would type this:

```
newWindow.document.bgColor = "red";
```

How would you close the window you just opened? Easy, just use the `window` object's `close()` method like this:

```
newWindow.close();
```

Try It Out Opening New Windows

Let's look at the example mentioned earlier of a products page in which clicking a product brings up a window listing the details of that product. In a shameless plug, you'll be using a couple of Wrox books as examples — though with just two products on your page, it's not exactly the world's most extensive online catalog.

```
<!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">
<head>
    <title>Chapter 8: Example 4</title>
    <script type="text/javascript">
    var detailsWindow;
    function showDetails(bookURL)
    {
        detailsWindow = window.open(bookURL, "bookDetails",
            "width=400,height=350");
        detailsWindow.focus();
    }
    </script>
</head>
<body>
    <h2>Online Book Buyer</h2>
    <p>
        Click any of the images below for more details
    </p>
    <h4>Professional Ajax</h4>
    <p>
        <img src="pro_ajax.jpg" alt="Professional Ajax, 2nd Edition" border="0"
            onclick="showDetails('pro_ajax_details.htm')" />
    </p>
    <h4>Professional JavaScript for Web Developers</h4>
    <p>
        <img src="pro_js.jpg" alt="Professional JavaScript, 2nd Edition"
            border="0" onclick="showDetails('pro_js_details.htm')" />
    </p>
</body>
</html>
```

Save this page as `ch08_examp4.htm`. You'll also need to create two images and name them `pro_ajax.jpg` and `pro_js.jpg`. Alternatively, you can find these files in the code download.

Note that the window will not open if the user disabled JavaScript — effectively breaking your web page. You can, however, get around this by surrounding the `<img/>` element with an `<a/>` element, assigning the `href` attribute to the book details page, and using the `<a/>` element's `onclick` event handler to return `false` after launching the new window as follows:

```
<a href="pro_ajax_details.htm"
 onclick="showDetails(this.href); return false;">
    <img src="pro_ajax.jpg" alt="Professional Ajax" />
</a>
```

In a JavaScript-enabled browser, clicking the link results in a new window containing the `pro_ajax_details.htm` page, and because the `onclick` handler returns `false`, the browser does not navigate the main window to the page defined in the link's `href` attribute. However, in browsers that have JavaScript disabled, the browser ignores and does not execute the code within the link's `onclick` event handler, thus navigating the user's browser to the book details page because it is defined in the `href` attribute.

You now need to create the two details pages, both plain HTML.

```
<!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">
<head>
    <title>Professional ASP.NET 2.0</title>
</head>
<body>
    <h3>Professional Ajax, 2nd Edition</h3>
    <strong>Subjects</strong><br />
    Ajax<br />
    Internet<br />
    JavaScript<br />
    ASP.NET<br />
    PHP<br />
    XML<br />
    <hr color="#cc3333" />
    <h3>Book overview</h3>
    <p>
        A comprehensive look at the technologies and techniques used in Ajax,
        complete with real world examples and case studies. A must have for
        any Web professional looking to build interactive Web sites.
    </p>
</body>
</html>
```

Save this as `pro_ajax_details.htm`.

```
<!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">
<head>
    <title>Professional JavaScript</title>
</head>
<body>
    <h3>Professional JavaScript, 2nd Edition</h3>
    <strong>Subjects</strong><br />
    ECMAScript<br />
    Internet<br />
    JavaScript
    <br />
    XML and Scripting<br />
    <hr color="#cc3333" />
    <p>
        This book takes a comprehensive look at the JavaScript language
        and prepares the reader in-depth knowledge of the languages.
    </p>
```

```
        <p>
            It includes a guide to the language - when where and how to get
            the most out of JavaScript - together with practical case studies
            demonstrating JavaScript in action. Coverage is bang up-to-date,
            with discussion of compatability issues and version differences,
            and the book concludes with a comprehensive reference section.
        </p>
    </body>
    </html>
```

Save the final page as `pro_js_details.htm`.

Load `ch08_examp4.htm` into your browser and click either of the two images. A new window containing the book's details should appear above the existing browser window. Click the other book image, and the window will be replaced by one containing the details of that book.

The files `pro_ajax_details.htm` and `pro_js_details.htm` are both plain HTML files, so you won't look at them. However, in `ch08_examp4.htm` you find some scripting action, which you *will* look at here.

In the script block at the top of the page, you first define the variable `detailsWindow`.

```
var detailsWindow;
```

You then have the function that actually opens the new windows.

```
function showDetails(bookURL)
{
    detailsWindow = window.open(bookURL,"bookDetails","width=400,height=350");
    detailsWindow.focus();

}
```

This function is connected to the `onclick` event handlers of book images that appear later in the page. The parameter `bookURL` is passed by the code in the `onclick` event handler and will be either `pro_ajax_details.htm` or `pro_js_details.htm`.

You create the new window with the `window.open()` method. You pass the `bookURL` parameter as the URL to be opened. You pass `bookDetails` as the name you want applied to the new window. If the window already exists, another new window won't be opened, and the existing one will be navigated to the URL that you pass. This only occurs because you are using the same name (`bookDetails`) when opening the window for each book. If you had used a different name, a new window would be opened.

By storing the reference to the `window` object just created in the variable `detailsWindow`, you can access its methods and properties. On the next line, you'll see that you use the `window` object, referenced by `detailsWindow`, to set the focus to the new window — otherwise it will appear behind the existing window if you click the same image in the main window more than once.

Although you are using the same function for each of the image's `onclick` event handlers, you pass a different parameter for each, namely the URL of the details page for the book in question.

```
<h4>Professional Ajax</h4>
<p>
    <img src="pro_ajax.jpg" alt="Professional Ajax, 2nd Edition" border="0"
        onclick="showDetails('pro_ajax_details.htm')" />
```

```
</p>
<h4>Professional JavaScript for Web Developers</h4>
<p>
    <img src="pro_js.jpg" alt="Professional JavaScript, 2nd Edition" border="0"
        onclick="showDetails('pro_js_details.htm')" />
</p>
```

Adding HTML to a New Window

You learned earlier that you can pass an empty string as the first parameter of the `window` object's `open()` method and then write to the page using HTML. Let's see how you would do that.

First, you need to open a blank window by passing an empty value to the first parameter that specifies the file name to load.

```
var newWindow = window.open("","myNewWindow","width=150,height=150");
```

Now you can open the window's `document` to receive your HTML.

```
newWindow.document.open();
```

This is not essential when a new window is opened, because the page is blank; but with a document that already contains HTML, it has the effect of clearing out all existing HTML and blanking the page, making it ready for writing.

Now you can write out any valid HTML using the `document.write()` method.

```
newWindow.document.write("<h4>Hello</h4>");
newWindow.document.write("<p>Welcome to my new little window</p>");
```

Each time you use the `write()` method, the text is added to what's already there until you use the `document.close()` method.

```
newWindow.document.close();
```

If you then use the `document.write()` method again, the text passed will replace existing HTML rather than adding to it.

Adding Features to Your Windows

As you have seen, the `window.open()` method takes three parameters, and it's the third of these parameters that you'll be looking at in this section. Using this third parameter, you can control things such as the size of the new window created, its start position on the screen, whether the user can resize it, whether it has a toolbar, and so on.

Features such as menu bar, status bar, and toolbar can be switched on or off with `yes` or `1` for on and `no` or `0` for off. You can also switch these features on by including their names without specifying a value.

The list of possible options shown in the following table is not complete, and not all of them work with both IE and Firefox browsers.

Window Feature	Possible Values	Description
`copyHistory`	`yes,no`	Copy the history of the window doing the opening to the new window
`directories`	`yes,no`	Show directory buttons
`height`	`integer`	Height of new window in pixels
`left`	`integer`	Window's offset from left of screen.
`location`	`yes,no`	Show location text field
`menubar`	`yes,no`	Show menu bar
`resizable`	`yes,no`	Enable the user to resize the window after it has been opened
`scrollbars`	`yes,no`	Show scrollbars if the page is too large to fit in the window
`status`	`yes,no`	Show status bar
`toolbar`	`yes,no`	Show toolbar
`top`	`integer`	Window's offset from top of screen.
`width`	`integer`	Width of new window in pixels

As mentioned earlier, this third parameter is optional. If you don't include it, then all of the window features default to `yes`, except the window's size and position properties, which default to preset values. For example, if you try the following code, you'll see a window something like the one shown in Figure 8-10:

```
<!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">
<head>
    <script type="text/javascript">
    var newWindow;
    newWindow = window.open("","myWindow");
    </script>
</head>
<body>
</body>
</html>
```

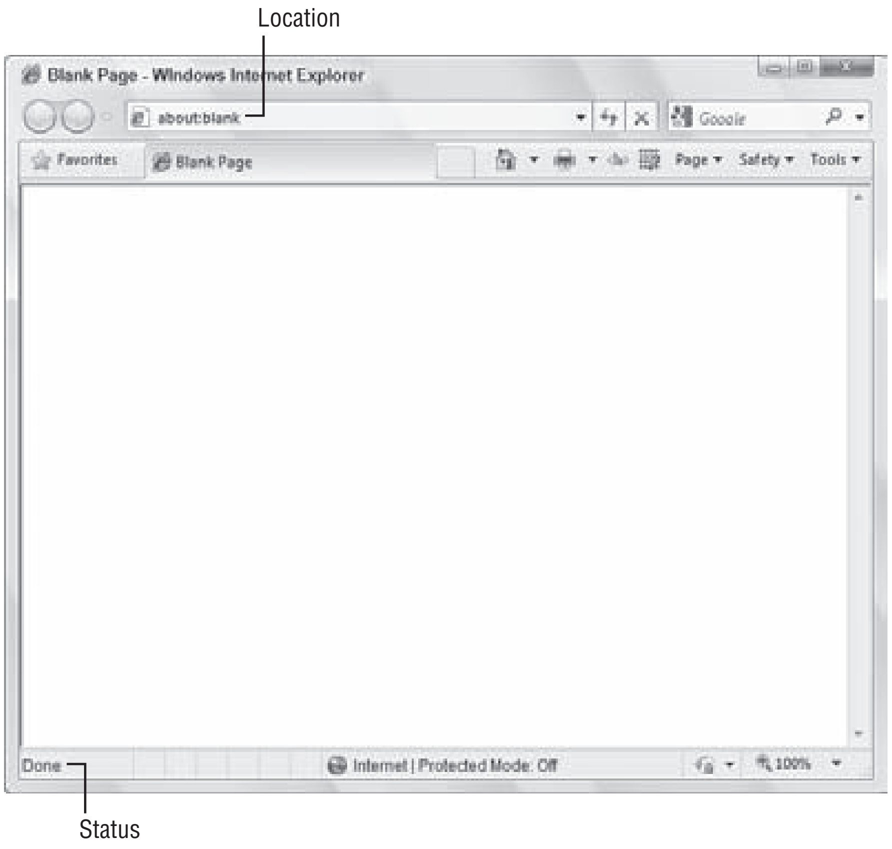


Figure 8-10

Figure 8-10 is of IE8. The default UI hides the menu and toolbars; so as long as the default settings are in effect, opened windows will not show the menu and toolbars.

However, if you specify even one of the features, all the others (except size and position properties) are set to `no` by default. For example, although you have defined its size, the following code produces a window with no features, as shown in Figure 8-11:

```
var newWindow = window.open("","myWindow","width=200,height=120")
```

The larger window is the original page, and the smaller one on top (shown in Figure 8-11) is the pop-up window.

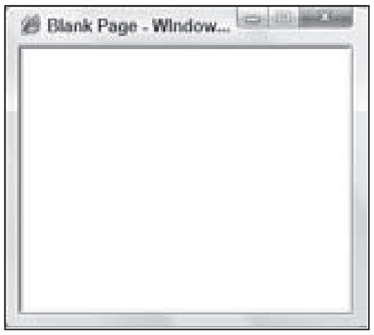


Figure 8-11

Let's see another example. The following creates a resizable 250-by-250-pixel window, with a location field and menu bar:

```
var newWindow = window.open("","myWindow",
                    "width=250,height=250,location,menubar,resizable");
```

A word of warning, however: Never include spaces inside the features string; otherwise some browsers will consider the string invalid and ignore your settings.

Scripting Between Windows

You've taken a brief look at how you can manipulate the new window's properties and methods, and access its `document` object using the return value from the `window.open()` method. Now you're going to look at how the newly opened window can access the window that opened it and, just as with frames, how it can use functions there.

The key to accessing the `window` object of the window that opened the new window is the `window` object's `opener` property. This returns a reference to the `window` object of the window that opened the new window. So the following code will change the background color of the `opener` window to red:

```
window.opener.document.bgColor = "red";
```

You can use the reference pretty much as you used the `window.parent` and `window.top` properties when using frames.

Try It Out   Inter-Window Scripting

Let's look at an example wherein you open a new window and access a form on the opener window from the new window.

```
<!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">
<head>
    <title>Chapter 8: Example 5</title>
    <script type="text/javascript">
    var newWindow;
    function btnOpenWin_onclick()
    {
        var winTop = (screen.height / 2) - 125;
        var winLeft = (screen.width / 2) - 125;
        var windowFeatures = "width=250,height=250,";
        windowFeatures = windowFeatures + "left=" + winLeft + ",";
        windowFeatures = windowFeatures + "top=" + winTop;
        newWindow = window.open("ch08_examp5_popup.htm", "myWindow",
            windowFeatures);
    }
    function btnGetText_onclick()
    {
        if (typeof (newWindow) == "undefined" || newWindow.closed == true)
        {
            alert("No window is open");
        }
        else
        {
            document.form1.text1.value = newWindow.document.form1.text1.value;
        }
    }

    function window_onunload()
    {
        if (typeof (newWindow) != "undefined" && newWindow.closed == false)
        {
            newWindow.close();
        }
    }
    </script>
</head>
<body onunload="window_onunload()">
    <form name="form1" action="">
        <input type="button" value="Open newWindow" name="btnOpenWin"
            onclick="btnOpenWin_onclick()" />
        <p>
            newWindow's Text:<br />
            <input type="text" name="text1" />
            <input type="button" value="Get Text" name="btnGetText"
                onclick="btnGetText_onclick()" />
        </p>
    </form>
</body>
</html>
```

This is the code for your original window. Save it as `ch08_examp5.htm`. Now you'll look at the page that will be loaded by the opener window.

```
<!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">
<head>
    <title>Chapter 8: Example 5 Popup</title>
    <script type="text/javascript">
    function btnGetText_onclick()
    {
        document.form1.text1.value = window.opener.document.form1.text1.value;
    }
    </script>
</head>
<body>
    <form name="form1" action="">
        Opener window's text<br />
        <input type="text" name="text1" />
```

```
            <input type="button" value="Get Text" name="btnGetText"
                onclick="btnGetText_onclick()" />
        </form>
    </body>
    </html>
```

Save this as `ch08_examp5_popup.htm`.

Open `ch08_examp5.htm` in your browser, and you'll see a page with the simple form shown in Figure 8-12.

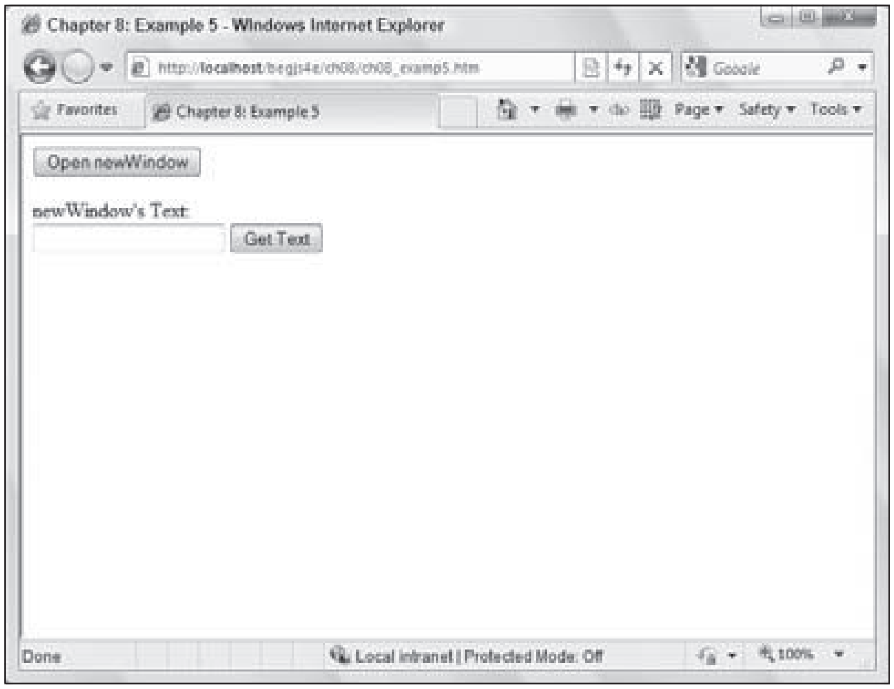

Chapter 8: Example 5 - Windows Internet Explorer
http://localhost/begjs4e/ch08/ch08_examp5.htm
Google
Favorites
Chapter 8: Example 5
Page
Safety
Tools
Open newWindow
newWindow's Text
Get Text
Done
Local intranet | Protected Mode: Off
100%

Figure 8-12

Click the `Open newWindow` button, and you'll see the window shown in Figure 8-13 open above the original page.

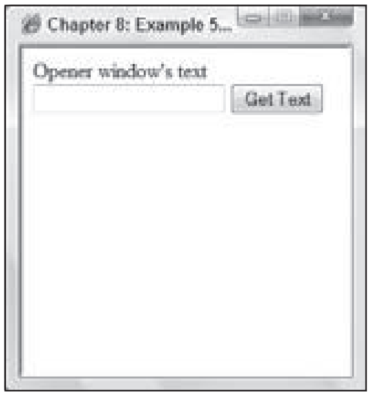


Figure 8-13

Type something into the text box of the new window. Then return to the original opener window, click the Get Text button, and you'll see what you just typed into newWindow appear in the text box on the opener window's form.

Change the text in the opener window's text box and then return to the newWindow and click the Get Text button. The text you typed into the opener window's text box will appear in newWindow's text box.

Let's look at the opener window first. In the head of the page is a script block in which a variable and three functions are defined. At the top you have declared a new variable, newWindow, which will hold the window object reference returned by the window.open() method you'll use later. Being outside any function gives this variable a global scope, so you can access it from any function on the page.

```
var newWindow;
```

Then you have the first of the three functions in this page, btnOpenWin_onclick(), which is connected further down the page to the Open newWindow button's onclick event handler. Its purpose is simply to open the new window.

Rather than have the new window open up anywhere on the page, you use the built-in screen object, which is a property of the window object, to find out the resolution of the user's display and place the window in the middle of the screen. The screen object has a number of read-only properties, but you're interested here in the width and height properties. You initialize the winTop variable to the vertical position onscreen at which you want the top edge of the popup window to appear. The winLeft variable is set to the horizontal position onscreen at which you want the left edge of the pop-up window to appear. In this case, you want the position to be in the middle of the screen both horizontally and vertically.

```
function btnOpenWin_onclick()
{
    var winTop = (screen.height / 2) - 125;
    var winLeft = (screen.width / 2) - 125;
```

You build up a string for the window features and store it in the windowFeatures variable. You set the width and height to 250 and then use the winLeft and winTop variables you just populated to create the initial start positions of the window.

```
    var windowFeatures = "width=250,height=250,";
    windowFeatures = windowFeatures + "left=" + winLeft + ",";
    windowFeatures = windowFeatures + "top=" + winTop;
```

Finally, you open the new window, making sure you put the return value from window.open() into global variable newWindow so you can manipulate it later.

```
    newWindow = window.open("newWindow.htm","myWindow",windowFeatures);
}
```

The next function is used to obtain the text from the text box on the form in newWindow.

In this function you use an if statement to check two things. First, you check that newWindow is defined and second, that the window is actually open. You check because you don't want to try to access a nonexistent window, for example if no window has been opened or a window has been closed by the user. The typeof operator returns the type of information held in a variable, for example number, string, Boolean, object, and undefined. It returns undefined if the variable has never been given a value, as newWindow won't have been if no new window has been opened.

Having confirmed that a window has been opened at some point, you now need to check whether it's still open, and the `window` object's `closed` property does just that. If it returns `true`, the window is closed, and if it returns `false`, it's still open. (Do not confuse this `closed` property with the `close()` method you saw previously.)

In the `if` statement, you'll see that checking if `newWindow` is defined comes first, and this is no accident. If `newWindow` really were undefined, `newWindow.closed` would cause an error, because there are no data inside `newWindow`. However, you are taking advantage of the fact that if an `if` statement's condition will be `true` or `false` at a certain point regardless of the remainder of the condition, the remainder of the condition is not checked.

```
function butGetText_onclick()
{
    if (typeof(newWindow) == "undefined" || newWindow.closed == true)
    {
        alert("No window is open");
    }
```

If `newWindow` exists and is open, the `else` statement's code will execute. Remember that `newWindow` will contain a reference to the `window` object of the window opened. This means you can access the form in `newWindow`, just as you'd access a form on the page the script's running in, by using the `docu-ment` object inside the `newWindow` `window` object.

```
    else
    {
        document.form1.text1.value = newWindow.document.form1.text1.value;
    }
}
```

The last of the three functions is `window_onunload()`, which is connected to the `onunload` event of this page and fires when either the browser window is closed or the user navigates to another page. In the `window_onunload()` function, you check to see if `newWindow` is valid and open in much the same way that you just did. You must check to see if the `newWindow` variable is defined first. With the `&&` operator, JavaScript checks the second part of the operation only if the first part evaluates to `true`. If `newWindow` is defined, and does therefore hold a `window` object (even though it's possibly a closed window), you can check the `closed` property of the window. However, if `newWindow` is undefined, the check for its `closed` property won't happen, and no errors will occur. If you check the `closed` property first and `newWindow` is undefined, an error will occur, because an undefined variable has no `closed` property.

```
function window_onunload()
{
    if (typeof(newWindow) != "undefined" && newWindow.closed == false)
    {
        newWindow.close();
    }
}
```

If `newWindow` is defined and open, you close it. This prevents the `newWindow`'s Get Text button from being clicked when there is no opener window in existence to get text from (since this function fires when the opener window is closed).

Let's now look at the code for the page that will be loaded in the `newWindow`: `ch08_examp5_popup.htm`. This page contains one function, `btnGetText_onclick()`, which is connected to the `onclick` event handler of the Get Text button in the page. It retrieves the text from the opener window's text box.

```
function btnGetText_onclick()
{
    document.form1.text1.value = window.opener.document.form1.text1.value;
}
```

In this function, you use the `window.opener` property to get a reference to the `window` object of the window that opened this one, and then use that reference to get the value out of the text box in the form in that window. This value is placed inside the text box in the current page.

Moving and Resizing Windows

In addition to opening and closing windows, it's also possible to move and resize windows.

After opening a window, you can change its onscreen position and its size using the `window` object's `resizeTo()` and `moveTo()` methods, both of which take two arguments in pixels.

Consider the following code that opens a new window:

```
var newWindow = window.open(myURL,"myWindow","width=125,height=150,resizable");
```

You want to make it 350 pixels wide by 200 pixels high and move it to a position 100 pixels from the left of the screen and 400 pixels from the top. What code would you need?

```
newWindow.resizeTo(350,200);
newWindow.moveTo(100,400);
```

You can see that you can resize your window to 350 pixels wide by 200 pixels high using `resizeTo()`. Then you move it so it's 100 pixels from the left of the screen and 400 pixels from the top of the screen using `moveTo()`.

The `window` object also has `resizeBy()` and `moveBy()` methods. Both of these methods accept two parameters, in pixels. For example:

```
newWindow.resizeBy(100,200);
```

This code will increase the size of `newWindow` by 100 pixels horizontally and 200 pixels vertically. Similarly, the following code moves the `newWindow` by 20 pixels horizontally and 50 pixels vertically:

```
newWindow.moveBy(20,50);
```

When using these methods, you must bear in mind that users can manually resize these windows if they so wish. In addition, the size of the client's screen in pixels will vary between users.

Security

Browsers put certain restrictions on what information scripts can access between frames and windows.

If all the pages in these frames and windows are served from the same server, or on the same computer when you're loading them into the browser locally, as you are in these examples, you have a reasonably free rein over what your scripts can access and do. However, some restrictions do exist. For example, if you try to use the `window.close()` method in a script page loaded into a browser window that the user opened, as opposed to a window opened by your script, a message box will appear giving the user the option of canceling your `close()` method and keeping the window open.

When a page in one window or frame hosted on one server tries to access the properties of a window or frame that contains a page from a different server, the same-origin policy comes into play, and you'll find yourself very restricted as to what your scripts can do.

Imagine you have a page hosted on a web server whose URL is `http://www.myserver.com`. Inside the page is the following script:

```
var myWindow =
    window.open("http://www.anotherserver.com/anotherpage.htm","myWindow");
```

Now you have two windows, one that is hosted at `www.myserver.com` and another that is hosted on a different server, `www.anotherserver.com`. Although this code does work, the same-origin policy prevents any access to the `document` object of one page from another. For example, the following code in the opener page will cause a security problem and will be prevented by the browser:

```
var myVariable = myWindow.document.form1.text1.value;
```

Although you do have access to the `window` object of the page on the other server, you have access to a limited subset of its properties and methods.

The same-origin restriction applies to frames (conventional and iframes) and windows equally. The idea behind it is very sound: It is there to prevent hackers from putting your pages inside their own and extracting information by using code inside their pages. However, the restrictions are fairly severe, perhaps too severe, and mean that you should avoid scripting across frames or windows if the pages are hosted on different servers.

Summary

For various reasons, having a frame-based web site can prove very useful. Therefore, you need to be able to create JavaScript that can interact with frames and with the documents and code within those frames.

- You saw that an advantage of frames is that, by putting all of your general functions in a single frame, you can create a JavaScript code module that all of your web site can use.
- You saw that the key to coding with frames is getting a reference to the `window` objects of other frames. You saw two ways of accessing frames higher in the hierarchy, using the `window` object's `parent` property and its `top` property.

- The `parent` property returns the `window` object that contains the current `window` object, which will be the page containing the frameset that created the window. The `top` property returns the `window` object of the window containing all the other frames.
- Each frame in a frameset can be accessed through three methods. One is to use the name of the frame. The second is to use the `frames` collection and specify the index of the frame. The third way is to access the frame by its name in the frames collection — for example, `parent.frames .frameName`. This the safest way, because it avoids any collision with global variables.
- If the frame you want to access is defined in another window, you need the `parent` or `top` property to get a reference to the `window` object defining that frame, and then you must specify the name or position in the `frames` collection.

You then looked at how you can open new, additional browser windows using script.

- Using the `window` object's `open()` method, you can open new windows. The URL of the page you want to open is passed as the first parameter; the name of the new window is passed as the second parameter; the optional third parameter enables you to define what features the new window will have.
- The `window.open()` method returns a value, which is a reference to the `window` object of the new window. Using this reference, you can access the document, script, and methods of that window, much as you do with frames. You need to make sure that the reference is stored inside a variable if you want to do this.
- To close a window, you simply use the `window.close()` method. To check if a window is closed, you use the `closed` property of the `window` object, which returns `true` if it's closed and `false` if it's still open.
- For a newly opened `window` object to access the window that opened it, you need to use the `window.opener` property. Like `window.parent` for frames, this gives a reference to the `window` object that opened the new one and enables you to access the `window` object and its properties for that window.
- After a window is opened, you can resize it using `resizeTo(x,y)` and `resizeBy(x,y)`, and move it using `moveTo(x,y)` and `moveBy(x,y)`.

You also looked briefly at security restrictions for windows and frames that are not of the same origin. By "not of the same origin," you're referring to a situation in which the document in one frame is hosted on one server and the document in the other is hosted on a different server. In this situation, very severe restrictions apply, which limit the extent of scripting between frames or windows.

In the next chapter, you look at advanced string manipulation.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

1. In the previous chapter's exercise questions, you created a form that allowed the user to pick a computer system. They could view the details of their system and its total cost by clicking a button that wrote the details to a `textarea`. Change the example so it's a frames-based web

page; instead of writing to a text area, the application should write the details to another frame. Hint: use `about:blank` as the src of the frame you write to. Hint: use the `document` object's `close()` and `open()` methods to clear the details frame from previously written data.

2. The fourth example (`ch08.examp4.htm`) was a page with images of books, in which clicking on a book's image brought up information about that book in a pop-up window. Amend this so that the pop-up window also has a button or link that, when clicked, adds the item to a shopping basket. Also, on the main page, give the user some way of opening up a shopping basket window with details of all the items they have purchased so far, and give them a way of deleting items from this basket.

9

String Manipulation

In Chapter 4 you looked at the `String` object, which is one of the native objects that JavaScript makes available to you. You saw a number of its properties and methods, including the following:

- ❑ `length` — The length of the string in characters
- ❑ `charAt()` and `charCodeAt()` — The methods for returning the character or character code at a certain position in the string
- ❑ `indexOf()` and `lastIndexOf()` — The methods that allow you to search a string for the existence of another string and that return the character position of the string if found
- ❑ `substr()` and `substring()` — The methods that return just a portion of a string
- ❑ `toUpperCase()` and `toLowerCase()` — The methods that return a string converted to upper- or lowercase

In this chapter you'll look at four new methods of the `String` object, namely `split()`, `match()`, `replace()`, and `search()`. The last three, in particular, give you some very powerful text-manipulation functionality. However, to make full use of this functionality, you need to learn about a slightly more complex subject.

The methods `split()`, `match()`, `replace()`, and `search()` can all make use of *regular expressions*, something JavaScript wraps up in an object called the `RegExp` object. Regular expressions enable you to define a pattern of characters, which can be used for text searching or replacement. Say, for example, that you have a string in which you want to replace all single quotes enclosing text with double quotes. This may seem easy — just search the string for `'` and replace it with `"` — but what if the string is `Bob O'Hara said "Hello"`? You would not want to replace the `single-quote character` in `O'Hara`. You can perform this text replacement without regular expressions, but it would take more than the two lines of code needed if you do use regular expressions.

Although `split()`, `match()`, `replace()`, and `search()` are at their most powerful with regular expressions, they can also be used with just plain text. You'll take a look at how they work in this simpler context first, to become familiar with the methods.

Additional String Methods

In this section you will take a look at the `split()`, `replace()`, `search()`, and `match()` methods, and see how they work without regular expressions.

The split() Method

The `String` object's `split()` method splits a single string into an array of substrings. Where the string is split is determined by the separation parameter that you pass to the method. This parameter is simply a character or text string.

For example, to split the string `"A,B,C"` so that you have an array populated with the letters between the commas, the code would be as follows:

```
var myString = "A,B,C";
var myTextArray = myString.split(',');
```

JavaScript creates an array with three elements. In the first element it puts everything from the start of the string `myString` up to the first comma. In the second element it puts everything from after the first comma to before the second comma. Finally, in the third element it puts everything from after the second comma to the end of the string. So, your array `myTextArray` will look like this:

A	B	C

If, however, your string were `"A,B,C,"` JavaScript would split it into four elements, the last element containing everything from the last comma to the end of the string; in other words, the last string would be an empty string.

A	B	C	

This is something that can catch you off guard if you're not aware of it.

Try It Out Reversing the Order of Text

Let's create a short example using the `split()` method, in which you reverse the lines written in a `<textarea>` element.

```
<!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">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Example 1</title>
<script language="JavaScript" type="text/JavaScript">
function splitAndReverseText(textAreaControl)
{
   var textToSplit = textAreaControl.value;
   var textArray = textToSplit.split('\n');
   var numberOfParts = 0;
   numberOfParts = textArray.length;
   var reversedString = "";
   var indexCount;
   for (indexCount = numberOfParts - 1; indexCount >= 0; indexCount--)
   {
      reversedString = reversedString + textArray[indexCount];
      if (indexCount > 0)
      {
         reversedString = reversedString + "\n";
      }
   }

   textAreaControl.value = reversedString;
}
</script>
</head>
<body>
<form name="form1">
<textarea rows="20" cols="40" name="textarea1" wrap="soft">Line 1
Line 2
Line 3
Line 4</textarea>
<br />
<input type="button" value="Reverse Line Order" name="buttonSplit"
   onclick="splitAndReverseText(document.form1.textarea1)">
</form>
</body>
</html>
```

Save this as `ch9_examp1.htm` and load it into your browser. You should see the screen shown in Figure 9-1.

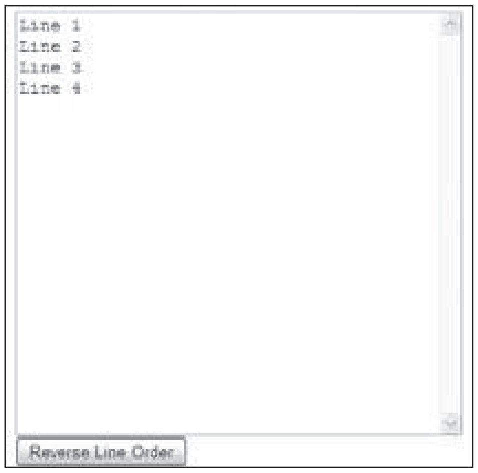
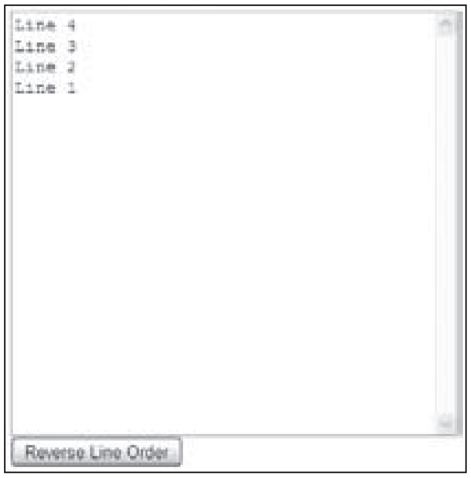


Figure 9-1

Clicking the Reverse Line Order button reverses the order of the lines, as shown in Figure 9-2.



```
Line 4
Line 3
Line 2
Line 1
```

Figure 9-2

Try changing the lines within the text area to test it further.

Although this example works on Internet Explorer (IE) as it is, an extra line gets inserted. If this troubles you, you can fix it by replacing each instance of `\n` *with* `\r\n` *for IE.*

The key to how this code works is the function `splitAndReverseText()`. This function is defined in the script block in the head of the page and is connected to the `onclick` event handler of the button further down the page.

```
<input type="button" value="Reverse Line Order" name=buttonSplit
   onclick="splitAndReverseText(document.form1.textarea1)">
```

As you can see, you pass a reference of the text area that you want to reverse as a parameter to the function. By doing it this way, rather than just using a reference to the element itself inside the function, you make the function more generic, so you can use it with any `textarea` element.

Now, on with the function. You start by assigning the value of the text inside the `textarea` element to the `textToSplit` variable. You then split that string into an array of lines of text using the `split()` method of the `String` object and put the resulting array inside the `textArray` variable.

```
function splitAndReverseText(textAreaControl)
{
   var textToSplit = textAreaControl.value;
   var textArray = textToSplit.split('\n');
```

So what do you use as the separator to pass as a parameter for the `split()` method? Recall from Chapter 2 that the escape character `\n` is used for a new line. Another point to add to the confusion is that IE seems to need `\r\n` rather than `\n`.

You next define and initialize three more variables.

```
   var numberOfParts = 0;
   numberOfParts = textArray.length;
   var reversedString = "";
   var indexCount;
```

Now that you have your array of strings, you next want to reverse them. You do this by building up a new string, adding each string from the array, starting with the last and working toward the first. You do this in the `for` loop, where instead of starting at `0` and working up as you usually do, you start at a number greater than `0` and decrement until you reach `0`, at which point you stop looping.

```
        for (indexCount = numberOfParts - 1; indexCount >= 0; indexCount--)
        {
           reversedString = reversedString + textArray[indexCount];
           if (indexCount > 0)
           {
              reversedString = reversedString + "\n";
           }
        }
```

Finally, you assign the text in the `textarea` element to the new string you've built.

```
        textAreaControl.value = reversedString;
     }
```

After you've looked at regular expressions, you'll revisit the `split()` method.

The replace() Method

The `replace()` method searches a string for occurrences of a substring. Where it finds a match for this substring, it replaces the substring with a third string that you specify.

Let's look at an example. Say you have a string with the word `May` in it, as shown in the following:

```
var myString = "The event will be in May, the 21st of June";
```

Now, say you want to replace `May` with `June`. You can use the `replace()` method like so:

```
myCleanedUpString = myString.replace("May","June");
```

The value of `myString` will not be changed. Instead, the `replace()` method returns the value of `myString` but with `May` replaced with `June`. You assign this returned string to the variable `myCleanedUpString`, which will contain the corrected text.

```
"The event will be in June, the 21st of June"
```

The search() Method

The `search()` method enables you to search a string for a particular piece of text. If the text is found, the character position at which it was found is returned; otherwise `-1` is returned. The method takes only one parameter, namely the text you want to search for.

When used with plain text, the `search()` method provides no real benefit over methods like `indexOf()`, which you've already seen. However, you'll see later that it's when you use regular expressions that the power of this method becomes apparent.

In the following example, you want to find out if the word Java is contained within the string called `myString`.

```
var myString = "Beginning JavaScript, Beginning Java, Professional JavaScript";
alert(myString.search("Java"));
```

The alert box that occurs will show the value `10`, which is the character position of the `J` in the first occurrence of `Java`, as part of the word `JavaScript`.

The match() Method

The `match()` method is very similar to the `search()` method, except that instead of returning the position at which a match was found, it returns an array. Each element of the array contains the text of each match that is found.

Although you can use plain text with the `match()` method, it would be completely pointless to do so. For example, take a look at the following:

```
var myString = "1997, 1998, 1999, 2000, 2000, 2001, 2002";
myMatchArray = myString.match("2000");
alert(myMatchArray.length);
```

This code results in `myMatchArray` holding an element containing the value `2000`. Given that you already know your search string is `2000`, you can see it's been a pretty pointless exercise.

However, the `match()` method makes a lot more sense when you use it with regular expressions. Then you might search for all years in the twenty-first century — that is, those beginning with `2`. In this case, your array would contain the values `2000`, `2000`, `2001`, and `2002`, which is much more useful information!

Regular Expressions

Before you look at the `split()`, `match()`, `search()`, and `replace()` methods of the `String` object again, you need to look at regular expressions and the `RegExp` object. Regular expressions provide a means of defining a pattern of characters, which you can then use to split, search for, or replace characters in a string when they fit the defined pattern.

JavaScript's regular expression syntax borrows heavily from the regular expression syntax of Perl, another scripting language. The latest versions of languages, such as VBScript, have also incorporated regular expressions, as do lots of applications, such as Microsoft Word, in which the Find facility allows regular expressions to be used. The same is true for Dreamweaver. You'll find that your regular expression knowledge will prove useful even outside JavaScript.

Regular expressions in JavaScript are used through the `RegExp` object, which is a native JavaScript object, as are `String`, `Array`, and so on. There are two ways of creating a new `RegExp` object. The easier is with a regular expression literal, such as the following:

```
var myRegExp = /\b'|'\b/;
```

The forward slashes (`/`) mark the start and end of the regular expression. This is a special syntax that tells JavaScript that the code is a regular expression, much as quote marks define a string's start and end. Don't worry about the actual expression's syntax yet (the `\b'|'\b`) — that will be explained in detail shortly.

Alternatively, you could use the `RegExp` object's constructor function `RegExp()` and type the following:

```
var myRegExp = new RegExp("\\b'|'\\b");
```

Either way of specifying a regular expression is fine, though the former method is a shorter, more efficient one for JavaScript to use and therefore is generally preferred. For much of the remainder of the chapter, you'll use the first method. The main reason for using the second method is that it allows the regular expression to be determined at runtime (as the code is executing and not when you are writing the code). This is useful if, for example, you want to base the regular expression on user input.

Once you get familiar with regular expressions, you will come back to the second way of defining them, using the `RegExp()` constructor. As you can see, the syntax of regular expressions is slightly different with the second method, so we'll return to this subject later.

Although you'll be concentrating on the use of the `RegExp` object as a parameter for the `String` object's `split()`, `replace()`, `match()`, and `search()` methods, the `RegExp` object does have its own methods and properties. For example, the `test()` method enables you to test to see if the string passed to it as a parameter contains a pattern matching the one defined in the `RegExp` object. You'll see the `test()` method in use in an example shortly.

Simple Regular Expressions

Defining patterns of characters using regular expression syntax can get fairly complex. In this section you'll explore just the basics of regular expression patterns. The best way to do this is through examples.

Let's start by looking at an example in which you want to do a simple text replacement using the `replace()` method and a regular expression. Imagine you have the following string:

```
var myString = "Paul, Paula, Pauline, paul, Paul";
```

and you want to replace any occurrence of the name "Paul" with "Ringo."

Well, the pattern of text you need to look for is simply `Paul`. Representing this as a regular expression, you just have this:

```
var myRegExp = /Paul/;
```

As you saw earlier, the forward-slash characters mark the start and end of the regular expression. Now let's use this expression with the `replace()` method.

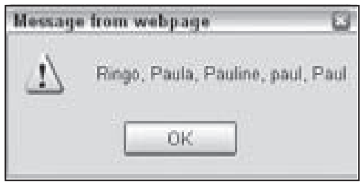
```
myString = myString.replace(myRegExp, "Ringo");
```

You can see that the `replace()` method takes two parameters: the `RegExp` object that defines the pattern to be searched and replaced, and the replacement text.

If you put this all together in an example, you have the following:

```
<!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">
<body>
<script language="JavaScript" type="text/JavaScript">
  var myString = "Paul, Paula, Pauline, paul, Paul";
  var myRegExp = /Paul/;
  myString = myString.replace(myRegExp, "Ringo");
  alert(myString);
</script>
</body>
</html>
```

If you load this code into a browser, you will see the screen shown in Figure 9-3.



Ringo, Paula, Pauline, paul, Paul

Figure 9-3

You can see that this has replaced the first occurrence of `Paul` in your string. But what if you wanted all the occurrences of `Paul` in the string to be replaced? The two at the far end of the string are still there, so what happened?

By default, the `RegExp` object looks only for the first matching pattern, in this case the first `Paul`, and then stops. This is a common and important behavior for `RegExp` objects. Regular expressions tend to start at one end of a string and look through the characters until the first complete match is found, then stop.

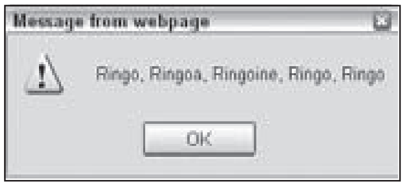
What you want is a global match, which is a search for all possible matches to be made and replaced. To help you out, the `RegExp` object has three attributes you can define. You can see these listed in the following table.

Attribute Character	Description
`G`	Global match. This looks for all matches of the pattern rather than stopping after the first match is found.
`I`	Pattern is case-insensitive. For example, `Paul` and `paul` are considered the same pattern of characters.
`M`	Multi-line flag. Only available in IE 5.5+ and NN 6+, this specifies that the special characters ^ and $ can match the beginning and the end of lines as well as the beginning and end of the string. You'll learn about these characters later in the chapter.

If you change the RegExp object in the code to the following, a global case-insensitive match will be made.

```
var myRegExp = /Paul/gi;
```

Running the code now produces the result shown in Figure 9-4.


Message from webpage

Ringo, Ringoa, Ringoine, Ringo, Ringo

Figure 9-4

This looks as if it has all gone horribly wrong. The regular expression has matched the Paul substrings at the start and the end of the string, and the penultimate paul, just as you wanted. However, the Paul substrings inside Pauline and Paula have also been replaced.

The RegExp object has done its job correctly. You asked for all patterns of the characters Paul to be replaced and that's what you got. What you actually meant was for all occurrences of Paul, when it's a single word and not part of another word, such as Paula, to be replaced. The key to making regular expressions work is to define exactly the pattern of characters you mean, so that only that pattern can match and no other. So let's do that.

1. You want paul or Paul to be replaced.
2. You don't want it replaced when it's actually part of another word, as in Pauline.

How do you specify this second condition? How do you know when the word is joined to other characters, rather than just joined to spaces or punctuation or the start or end of the string?

To see how you can achieve the desired result with regular expressions, you need to enlist the help of regular expression special characters. You'll look at these in the next section, by the end of which you should be able to solve the problem.

Try It Out Regular Expression Tester

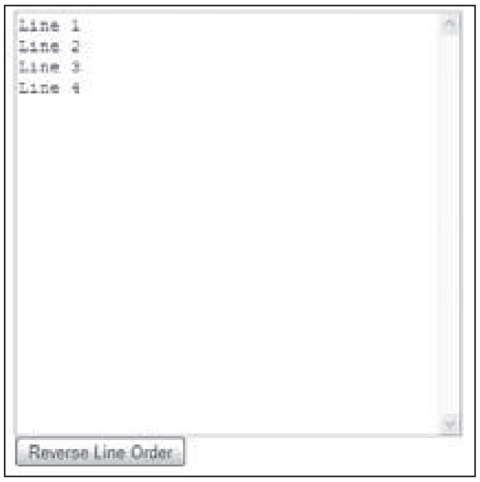
Getting your regular expression syntax correct can take some thought and time, so in this exercise you'll create a simple regular expression tester to make life easier.

Type the following code in to your text editor and save it as ch9_examp2.htm:

```
<!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">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Regular Expression Tester</title>
<style type="text/css">

body,td,th {
       font-family: Arial, Helvetica, sans-serif;
}

</style>

<script type="text/javascript">

function getRegExpFlags()
{
        var regExpFlags = '';
        if ( document.form1.chkGlobal.checked )
        {
                regExpFlags = 'g';
        }

        if ( document.form1.chkCaseInsensitive.checked )
        {
                regExpFlags += 'i';
        }

        if ( document.form1.chkMultiLine.checked )
        {
                regExpFlags += 'm';
        }

        return regExpFlags;

}

function doTest()
{
        var testRegExp = new RegExp(document.form1.txtRegularExpression.value,
 getRegExpFlags());
        if ( testRegExp.test(document.form1.txtTestString.value) )
        {
                document.form1.txtTestResult.value = "Match Found!";
        }
        else
        {
                document.form1.txtTestResult.value = "Match NOT Found!";
        }
}

function findMatches()
{
        var testRegExp = new RegExp(document.form1.txtRegularExpression.value,
getRegExpFlags());
        var myTestString = new String(document.form1.txtTestString.value)
        var matchArray = myTestString.match(testRegExp);

        document.form1.txtTestResult.value = matchArray.join('\n');

}

</script>

</head>
```

```
<body>

<form id="form1" name="form1" method="post" action="">
  <p>
    Regular Expression:<br />
<label>
      <input name="txtRegularExpression" type="text" id="txtRegularExpression"
size="100" value=""/>
      <br />
      Global
      <input name="chkGlobal" type="checkbox" id="chkGlobal" value="true" />
</label>

  Case Insensitive
  <label>
    <input name="chkCaseInsensitive" type="checkbox" id="chkCaseInsensitive"
value="true" />
  </label>

  Multi Line
  <label>
    <input name="chkMultiLine" type="checkbox" id="chkMultiLine" value="true" />
  </label>
  </p>
  <p>
    <label>
      Test Text:<br />
      <textarea name="txtTestString" id="txtTestString" cols="100"
rows="8"></textarea>
    </label>
  </p>
  <p>Result:<br />
    <textarea name="txtTestResult" id="txtTestResult" cols="100"
rows="8"></textarea>
  </p>
  <p>
    <label>
      <input type="button" name="cmdTest" id="cmdTest" value="TEST"
onclick="doTest();"/>
    </label>
    <label>
      <input type="button" name="cmdMatch" id="cmdMatch" value="MATCH"
onclick="findMatches();" />
    </label>
    <label>
      <input type="reset" name="cmdClearForm" id="cmdClearForm" value="Reset Form"
 />
    </label>
  </p>
  <p>&nbsp;</p>
</form>
</body>
</html>
```

Load the page into your browser, and you'll see the screen shown in Figure 9-5.



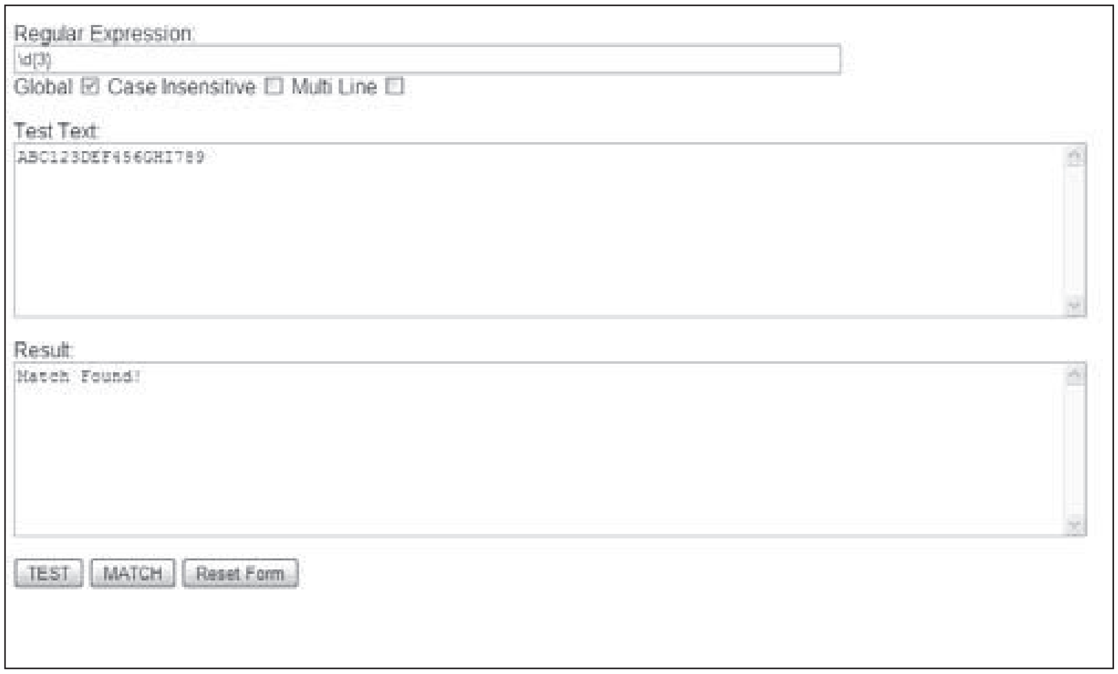
```
Line 1
Line 2
Line 3
Line 4
```

Figure 9-5

In the top box, you enter your regular expression. You can set the attributes such as global and case sensitivity by ticking the tick boxes. The text to test the regular expression against goes in the Test Text box, and the result is displayed in the Result box.

As a test, enter the regular expression `\d{3}`, which as you'll discover shortly, will match three digits. Also tick the Global box so all the matches will be found. Finally, your test text is `ABC123DEF456GHI789`.

If you click the Test button, the code will test to see if there are any matches (that is, if the test text contains three numbers). The result, as you can see in Figure 9-6, is that a match is found.



Regular Expression: \d{3}

Global [✓] Case Insensitive ☐ Multi Line ☐

Test Text:

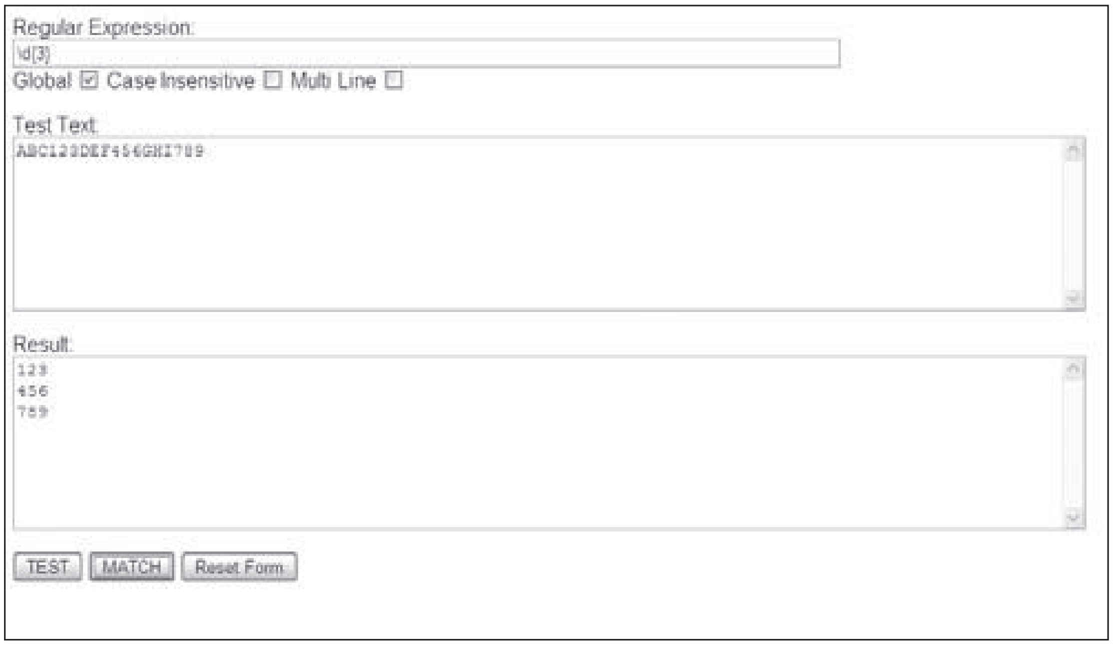
```
ABC123DEF456GHI789
```

Result:

```
Match Found!
```

Figure 9-6

Now to find all the matches, click the Match button, and this results in the screen shown in Figure 9-7.



```
\d{3}
```

Global ☑ Case Insensitive ☐ Multi Line ☐

```
ABC123DEF456GHI789
```

```
123
456
789
```

Figure 9-7

Each match of your regular expressions found in Test Text box is put on a separate line in the Results box.

The buttons cmdTest and cmdMatch have their click events linked to the `doTest()` and `findMatches()` functions. Let's start by looking at what happens in the `doTest()` function.

First, the regular expression object is created.

```
var testRegExp = new RegExp(document.form1.txtRegularExpression.value,
                            getRegExpFlags());
```

The first parameter of the object constructor is your regular expression as contained in the `txtRegularExpression` text box. This is easy enough to access, but the second parameter contains the regular expression flags, and these are generated via the tick boxes in the form. To convert the tick boxes to the correct flags, the function `getRegExpFlags()` has been created, and the return value from this function provides the flags value for the regular expressions constructor. The function `getRegExpFlags()` is used by both the `doTest()` and `getMatches()` functions. The `getRegExpFlags()` function is fairly simple. It starts by declaring `regExpFlags` and setting it to an empty string.

```
var regExpFlags = '';
```

Then for each of the tick boxes, it checks to see if the tick box is ticked. If it is, the appropriate flag is added to `regExpFlags` as shown here for the global flag:

```
        if ( document.form1.chkGlobal.checked )
        {
               regExpFlags = 'g';
        }
```

The same principle is used for the case-insensitive and multi-line flags.

Okay, back to the `doTest()` function. The regular expression object has been created and its flags have been set, so now you test to see if the regular expression matches anything in the Test Text box.

```
if ( testRegExp.test(document.form1.txtTestString.value) )
{
       document.form1.txtTestResult.value = "Match Found!";
}
else
{
       document.form1.txtTestResult.value = "Match NOT Found!";
}
```

If a match is found, `"Match Found!"` is written to the Results box; otherwise `"Match NOT Found!"` is written.

The regular expression object's `test()` method is used to do the actual testing for a match of the regular expression with the test string supplied as the method's only parameter. It returns `true` when a match is found or `false` when it's not. The global flag is irrelevant for the `test()` method, because it simply looks for the first match and returns `true` if found.

Now let's look at the `findMatches()` function, which runs when the `cmdMatches` button is clicked. As with the `doTest()` function, the first line creates a new regular expression object with the regular expression entered in the Regular Expression text box in the form and the flags being set via the `getRegExpFlags()` function.

```
var testRegExp = new RegExp(document.form1.txtRegularExpression.value,
                            getRegExpFlags());
```

Next, a new `String` object is created, and you then use the String object's `match()` method to find the matches.

```
var myTestString = new String(document.form1.txtTestString.value)
var matchArray = myTestString.match(testRegExp);
```

The `match()` method returns an array with all the matches found in each element of the array. The variable `matchArray` is used to store the array.

Finally, the match results are displayed in the Results box on the form:

```
document.form1.txtTestResult.value = matchArray.join('\n');
```

The `String` object's `join()` method joins all the elements in an array and returns them as a single string. Each element is separated by the value you pass as the `join()` method's only parameter. Here `\n` or the newline character has been passed, which means when the string is displayed in the Results box, each match is on its own individual line.

Regular Expressions: Special Characters

You will be looking at three types of special characters in this section.

Text, Numbers, and Punctuation

The first group of special characters you'll look at contains the character class's special characters. *Character class* means digits, letters, and whitespace characters. The special characters are displayed in the following table.

Character Class	Characters It Matches	Example
`\d`	Any digit from 0 to 9	`\d\d` matches 72, but not aa or 7a
`\D`	Any character that is not a digit	`\D\D\D` matches abc, but not 123 or 8ef
`\w`	Any word character; that is, A–Z, a–z, 0–9, and the underscore character (_)	`\w\w\w\w` matches Ab_2, but not £$%* or Ab_@
`\W`	Any non-word character	`\W` matches @, but not a
`\s`	Any whitespace character	`\s` matches tab, return, formfeed, and vertical tab
`\S`	Any non-whitespace character	`\S` matches `A`, but not the tab character
`.`	Any single character other than the new-line character (`\n`)	`.` matches a or 4 or @
`[...]`	Any one of the characters between the brackets`[a-z]` will match any character in the range a to z	`[abc]` will match a or b or c, but nothing else
`[^...]`	Any one character, but not one of those inside the brackets	`[^abc]` will match any character except a or b or c `[^a-z]` will match any character that is not in the range a to z

Note that uppercase and lowercase characters mean very different things, so you need to be extra careful with case when using regular expressions.

Let's look at an example. To match a telephone number in the format 1-800-888-5474, the regular expression would be as follows:

```
\d-\d\d\d-\d\d\d-\d\d\d\d
```

You can see that there's a lot of repetition of characters here, which makes the expression quite unwieldy. To make this simpler, regular expressions have a way of defining repetition. You'll see this a little later in the chapter, but first let's look at another example.

Try It Out Checking a Passphrase for Alphanumeric Characters

You'll use what you've learned so far about regular expressions in a full example in which you check that a passphrase contains only letters and numbers — that is, alphanumeric characters, not punctuation or symbols like `@`, `%`, and so on.

```
<!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">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Example 3</title>

<script type="text/JavaScript">
function regExpIs_valid(text)
{
   var myRegExp = /[^a-z\d ]/i;
   return !(myRegExp.test(text));
}
function butCheckValid_onclick()
{
   if (regExpIs_valid(document.form1.txtPhrase.value) == true)
   {
      alert("Your passphrase contains only valid characters");
   }
   else
   {
      alert("Your passphrase contains one or more invalid characters");
   }
}
</script>

</head>
<body>

<form name="form1">
Enter your passphrase:
<br />
<input type="text" name="txtPhrase">
<br />
<input type="button" value="Check Character Validity" name="butCheckValid"
   onclick="butCheckValid_onclick()">
</form>

</body>
</html>
```

Save the page as `ch9_examp3.htm`, and then load it into your browser. Type just letters, numbers, and spaces into the text box; click the Check Character Validity button, and you'll be told that the phrase contains valid characters. Try putting punctuation or special characters like `@`, `^`, `$`, and so on into the text box, and you'll be informed that your passphrase is invalid.

Let's start by looking at the `regExpIs_valid()` function defined at the top of the script block in the head of the page. That does the validity checking of the passphrase using regular expressions.

```
function regExpIs_valid(text)
{
   var myRegExp = /[^a-z\d ]/i;
   return !(myRegExp.test(text));
}
```

The function takes just one parameter: the text you want to check for validity. You then declare a variable, `myRegExp`, and set it to a new regular expression, which implicitly creates a new `RegExp` object.

The regular expression itself is fairly simple, but first think about what pattern you are looking for. What you want to find out is whether your passphrase string contains any characters that are not letters between A and Z or between a and z, numbers between 0 and 9, or spaces. Let's see how this translates into a regular expression:

1. You use square brackets with the `^` symbol.

   ```
   [^]
   ```

   This means you want to match any character that is not one of the characters specified inside the square brackets.

2. You add `a-z`, which specifies any character in the range a through z.

   ```
   [^a-z]
   ```

   So far, your regular expression matches any character that is not between a and z. Note that, because you added the `i` to the end of the expression definition, you've made the pattern case-insensitive. So your regular expression actually matches any character not between A and Z or a and z.

3. Add `\d` to indicate any digit character, or any character between 0 and 9.

   ```
   [^a-z\d]
   ```

4. Your expression matches any character that is not between a and z, A and Z, or 0 and 9. You decide that a space is valid, so you add that inside the square brackets.

   ```
   [^a-z\d ]
   ```

   Putting this all together, you have a regular expression that will match any character that is not a letter, a digit, or a space.

5. On the second and final line of the function, you use the `RegExp` object's `test()` method to return a value.

   ```
   return !(myRegExp.test(text));
   ```

The `test()` method of the `RegExp` object checks the string passed as its parameter to see if the characters specified by the regular expression syntax match anything inside the string. If they do, `true` is returned; if not, `false` is returned. Your regular expression will match the first invalid character found, so if you get a result of `true`, you have an invalid passphrase. However, it's a bit illogical for an `is_valid` function to return `true` when it's invalid, so you reverse the result returned by adding the NOT operator (`!`).

Previously you saw the two-line validity checker function using regular expressions. Just to show how much more coding is required to do the same thing without regular expressions, here is a second function that does the same thing as `regExpIs_valid()` but without regular expressions.

```
function is_valid(text)
{
   var isValid = true;
   var validChars = "abcdefghijklmnopqrstuvwxyz1234567890 ";
   var charIndex;
   for (charIndex = 0; charIndex < text.length;charIndex++)
   {
      if ( validChars.indexOf(text.charAt(charIndex).toLowerCase()) < 0)
      {
         isValid = false;
         break;
      }
   }
   return isValid;
}
```

This is probably as small as the non-regular expression version can be, and yet it's still 15 lines long. That's six times the amount of code for the regular expression version.

The principle of this function is similar to that of the regular expression version. You have a variable, `validChars`, which contains all the characters you consider to be valid. You then use the `charAt()` method in a `for` loop to get each character in the passphrase string and check whether it exists in your `validChars` string. If it doesn't, you know you have an invalid character.

In this example, the non-regular expression version of the function is 15 lines, but with a more complex problem you could find it takes 20 or 30 lines to do the same thing a regular expression can do in just a few.

Back to your actual code: The other function defined in the head of the page is `butCheckValid_onclick()`. As the name suggests, this is called when the `butCheckValid` button defined in the body of the page is clicked.

This function calls your `regExpis_valid()` function in an `if` statement to check whether the passphrase entered by the user in the `txtPhrase` text box is valid. If it is, an alert box is used to inform the user.

```
function butCheckValid_onclick()
{
   if (regExpIs_valid(document.form1.txtPhrase.value) == true)
   {
      alert("Your passphrase contains valid characters");
   }
```

If it isn't, another alert box is used to let users know that their text was invalid.

```
   else
   {
      alert("Your passphrase contains one or more invalid characters");
   }
}
```

Repetition Characters

Regular expressions include something called repetition characters, which are a means of specifying how many of the last item or character you want to match. This proves very useful, for example, if you want to specify a phone number that repeats a character a specific number of times. The following table lists some of the most common repetition characters and what they do.

Special Character	Meaning	Example
`{n}`	Match `n` of the previous item	`x{2}` matches xx
`{n,}`	Match `n` or more of the previous item	`x{2,}` matches xx, xxx, xxxx, xxxxx, and so on
`{n,m}`	Match at least `n` and at most `m` of the previous item	`x{2,4}` matches xx, xxx, and xxxx
`?`	Match the previous item zero or one time	`x?` matches nothing or x
`+`	Match the previous item one or more times	`x+` matches x, xx, xxx, xxxx, xxxxx, and so on
`*`	Match the previous item zero or more times	`x*` matches nothing, or x, xx, xxx, xxxx, and so on

You saw earlier that to match a telephone number in the format 1-800-888-5474, the regular expression would be `\d-\d\d\d-\d\d\d-\d\d\d\d`. Let's see how this would be simplified with the use of the repetition characters.

The pattern you're looking for starts with one digit followed by a dash, so you need the following:

```
\d-
```

Next are three digits followed by a dash. This time you can use the repetition special characters — `\d{3}` will match exactly three `\d,` which is the any-digit character.

```
\d-\d{3}-
```

Next, there are three digits followed by a dash again, so now your regular expression looks like this:

```
\d-\d{3}-\d{3}-
```

Finally, the last part of the expression is four digits, which is `\d{4}`.

```
\d-\d{3}-\d{3}-\d{4}
```

You'd declare this regular expression like this:

```
var myRegExp = /\d-\d{3}-\d{3}-\d{4}/
```

Remember that the first / and last / tell JavaScript that what is in between those characters is a regular expression. JavaScript creates a `RegExp` object based on this regular expression.

As another example, what if you have the string `Paul Paula Pauline`, and you want to replace `Paul` and `Paula` with `George`? To do this, you would need a regular expression that matches both `Paul` and `Paula`.

Let's break this down. You know you want the characters `Paul`, so your regular expression starts as

```
Paul
```

Now you also want to match `Paula`, but if you make your expression `Paula`, this will exclude a match on `Paul`. This is where the special character `?` comes in. It enables you to specify that the previous character is optional — it must appear zero (not at all) or one time. So, the solution is

```
Paula?
```

which you'd declare as

```
var myRegExp = /Paula?/
```

Position Characters

The third group of special characters you'll look at are those that enable you to specify either where the match should start or end or what will be on either side of the character pattern. For example, you might want your pattern to exist at the start or end of a string or line, or you might want it to be between two words. The following table lists some of the most common position characters and what they do.

Position Character	Description
`^`	The pattern must be at the start of the string, or if it's a multi-line string, then at the beginning of a line. For multi-line text (a string that contains carriage returns), you need to set the multi-line flag when defining the regular expression using `/myreg ex/m`. Note that this is only applicable to IE 5.5 and later and NN 6 and later.
`$`	The pattern must be at the end of the string, or if it's a multi-line string, then at the end of a line. For multi-line text (a string that contains carriage returns), you need to set the multi-line flag when defining the regular expression using `/myreg ex/m`. Note that this is only applicable to IE 5.5 and later and NN 6 and later.
`\b`	This matches a word boundary, which is essentially the point between a word character and a non-word character.
`\B`	This matches a position that's not a word boundary.

For example, if you wanted to make sure your pattern was at the start of a line, you would type the following:

```
^myPattern
```

This would match an occurrence of `myPattern` if it was at the beginning of a line.

To match the same pattern, but at the end of a line, you would type the following:

```
myPattern$
```

The word-boundary special characters `\b` and `\B` can cause confusion, because they do not match characters but the positions between characters.

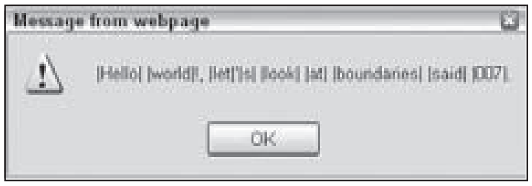
Imagine you had the string `"Hello world!, let's look at boundaries said 007."` defined in the code as follows:

```
var myString = "Hello world!, let's look at boundaries said 007.";
```

To make the word boundaries (that is, the boundaries between the words) of this string stand out, let's convert them to the | character.

```
var myRegExp = /\b/g;
myString = myString.replace(myRegExp, "|");
alert(myString);
```

You've replaced all the word boundaries, `\b`, with a `|`, and your message box looks like the one in Figure 9-8.



```
|Hello| |world|!, |let|'|s| |look| |at| |boundaries| |said| |007|.
```

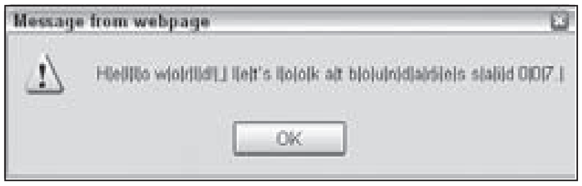
Figure 9-8

You can see that the position between any word character (letters, numbers, or the underscore character) and any non-word character is a word boundary. You'll also notice that the boundary between the start or end of the string and a word character is considered to be a word boundary. The end of this string is a full stop. So the boundary between the full stop and the end of the string is a non-word boundary, and therefore no | has been inserted.

If you change the regular expression in the example, so that it replaces non-word boundaries as follows:

```
var myRegExp = /\B/g;
```

you get the result shown in Figure 9-9.



H|e|l|l|o w|o|r|l|d!|,| l|e|t's l|o|o|k a|t b|o|u|n|d|a|r|i|e|s s|a|i|d 0|0|7.|

Figure 9-9

Now the position between a letter, number, or underscore and another letter, number, or underscore is considered a non-word boundary and is replaced by an | in the example. However, what is slightly confusing is that the boundary between two non-word characters, such as an exclamation mark and a comma, is also considered a non-word boundary. If you think about it, it actually does make sense, but it's easy to forget when creating regular expressions.

You'll remember this example from when you started looking at regular expressions:

```
<!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">
<body>
<script language="JavaScript" type="text/JavaScript">

  var myString = "Paul, Paula, Pauline, paul, Paul";
  var myRegExp = /Paul/gi;
  myString = myString.replace(myRegExp, "Ringo");
  alert(myString);



</script>
</body>
</html>
```

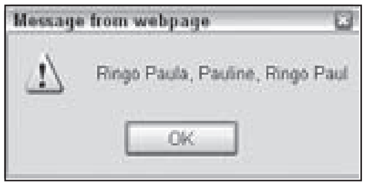
You used this code to convert all instances of `Paul` or `paul` to `Ringo`.

However, you found that this code actually converts all instances of `Paul` to `Ringo`, even when the word `Paul` is inside another word.

One way to solve this problem would be to replace the string `Paul` only where it is followed by a non-word character. The special character for non-word characters is `\W`, so you need to alter the regular expression to the following:

```
var myRegExp = /Paul\W/gi;
```

This gives the result shown in Figure 9-10.



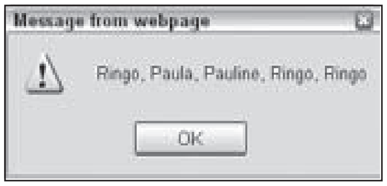
Ringo Paula, Pauline, Ringo Paul

Figure 9-10

It's getting better, but it's still not what you want. Notice that the commas after the second and third `Paul` substrings have also been replaced because they matched the `\W` character. Also, you're still not replacing `Paul` at the very end of the string. That's because there is no character after the letter `l` in the last `Paul`. What is after the `l` in the last `Paul`? Nothing, just the boundary between a word character and a non-word character, and therein lies the answer. What you want as your regular expression is `Paul` followed by a word boundary. Let's alter the regular expression to cope with that by entering the following:

```
var myRegExp = /Paul\b/gi;
```

Now you get the result you want, as shown in Figure 9-11.


Message from webpage

Ringo, Paula, Pauline, Ringo, Ringo

Figure 9-11

At last you've got it right, and this example is finished.

Covering All Eventualities

Perhaps the trickiest thing about a regular expression is making sure it covers all eventualities. In the previous example your regular expression works with the string as defined, but does it work with the following?

```
var myString = "Paul, Paula, Pauline, paul, Paul, JeanPaul";
```

Here the `Paul` substring in `JeanPaul` will be changed to `Ringo`. You really only want to convert the substring `Paul` where it is on its own, with a word boundary on either side. If you change your regular expression code to

```
var myRegExp = /\bPaul\b/gi;
```

you have your final answer and can be sure only `Paul` or `paul` will ever be matched.

Grouping Regular Expressions

The final topic under regular expressions, before you look at examples using the `match()`, `replace()`, and `search()` methods, is how you can group expressions. In fact, it's quite easy. If you want a number of expressions to be treated as a single group, you just enclose them in parentheses, for example, `/(\d\d)/`. Parentheses in regular expressions are special characters that group together character patterns and are not themselves part of the characters to be matched.

Why would you want to do this? Well, by grouping characters into patterns, you can use the special repetition characters to apply to the whole group of characters, rather than just one.

Let's take the following string defined in `myString` as an example:

```
var myString = "JavaScript, VBScript and Perl";
```

How could you match both `JavaScript` and `VBScript` using the same regular expression? The only thing they have in common is that they are whole words and they both end in `Script`. Well, an easy way would be to use parentheses to group the patterns `Java` and `VB`. Then you can use the `?` special character to apply to each of these groups of characters to make the pattern match any word having zero or one instances of the characters `Java` or `VB`, and ending in `Script`.

```
var myRegExp = /\b(VB)?(Java)?Script\b/gi;
```

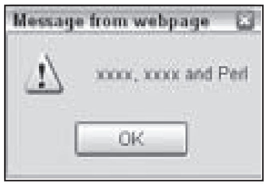
Breaking this expression down, you can see the pattern it requires is as follows:

1. A word boundary: `\b`
2. Zero or one instance of VB: `(VB)?`
3. Zero or one instance of Java: `(Java)?`
4. The characters `Script`: `Script`
5. A word boundary: `\b`

Putting these together, you get this:

```
var myString = "JavaScript, VBScript and Perl";
var myRegExp = /\b(VB)?(Java)?Script\b/gi;
myString = myString.replace(myRegExp, "xxxx");
alert(myString);
```

The output of this code is shown in Figure 9-12.



xxxx, xxxx and Perl

Figure 9-12

If you look back at the special repetition characters table, you'll see that they apply to the item preceding them. This can be a character, or, where they have been grouped by means of parentheses, the previous group of characters.

However, there is a potential problem with the regular expression you just defined. As well as matching VBScript and JavaScript, it also matches VBJavaScript. This is clearly not exactly what you meant.

To get around this you need to make use of both grouping and the special character `|`, which is the alternation character. It has an or-like meaning, similar to `||` in `if` statements, and will match the characters on either side of itself.

Let's think about the problem again. You want the pattern to match `VBScript` or `JavaScript`. Clearly they have the `Script` part in common. So what you want is a new word starting with `Java` or starting with `VB`; either way, it must end in `Script`.

First, you know that the word must start with a word boundary.

```
\b
```

Next you know that you want either `VB` or `Java` to be at the start of the word. You've just seen that in regular expressions | provides the "or" you need, so in regular expression syntax you want the following:

```
\b(VB|Java)
```

This matches the pattern `VB` or `Java`. Now you can just add the `Script` part.

```
\b(VB|Java)Script\b
```

Your final code looks like this:

```
var myString = "JavaScript, VBScript and Perl";
var myRegExp = /\b(VB|Java)Script\b/gi;
myString = myString.replace(myRegExp, "xxxx");
alert(myString);
```

Reusing Groups of Characters

You can reuse the pattern specified by a group of characters later on in the regular expression. To refer to a previous group of characters, you just type `\` and a number indicating the order of the group. For example, the first group can be referred to as `\1`, the second as `\2`, and so on.

Let's look at an example. Say you have a list of numbers in a string, with each number separated by a comma. For whatever reason, you are not allowed to have two instances of the same number in a row, so although

```
009,007,001,002,004,003
```

would be okay, the following:

```
007,007,001,002,002,003
```

would not be valid, because you have `007` and `002` repeated after themselves.

How can you find instances of repeated digits and replace them with the word `ERROR`? You need to use the ability to refer to groups in regular expressions.

First, let's define the string as follows:

```
var myString  = "007,007,001,002,002,003,002,004";
```

Now you know you need to search for a series of one or more number characters. In regular expressions the `\d` specifies any digit character, and `+` means one or more of the previous character. So far, that gives you this regular expression:

```
\d+
```

You want to match a series of digits followed by a comma, so you just add the comma.

```
\d+,
```

This will match any series of digits followed by a comma, but how do you search for any series of digits followed by a comma, then followed again by the same series of digits? As the digits could be any digits, you can't add them directly into the expression like so:

```
\d+,007
```

This would not work with the `002` repeat. What you need to do is put the first series of digits in a group; then you can specify that you want to match that group of digits again. This can be done with `\1`, which says, "Match the characters found in the first group defined using parentheses." Put all this together, and you have the following:

```
(\d+),\1
```

This defines a group whose pattern of characters is one or more digit characters. This group must be followed by a comma and then by the same pattern of characters as in the first group. Put this into some JavaScript, and you have the following:

```
var myString  = "007,007,001,002,002,003,002,004";
var myRegExp = /(\d+),\1/g;
myString = myString.replace(myRegExp,"ERROR");
alert(myString);
```

The alert box will show this message:

```
ERROR,1,ERROR,003,002,004
```

That completes your brief look at regular expression syntax. Because regular expressions can get a little complex, it's often a good idea to start simple and build them up slowly, as was done in the previous example. In fact, most regular expressions are just too hard to get right in one step — at least for us mere mortals without a brain the size of a planet.

If it's still looking a bit strange and confusing, don't panic. In the next sections, you'll be looking at the `String` object's `split()`, `replace()`, `search()`, and `match()` methods with plenty more examples of regular expression syntax.

The String Object — split(), replace(), search(), and match() Methods

The main functions making use of regular expressions are the `String` object's `split()`, `replace()`, `search()`, and `match()` methods. You've already seen their syntax, so you'll concentrate on their use with regular expressions and at the same time learn more about regular expression syntax and usage.

The split() Method

You've seen that the `split()` method enables us to split a string into various pieces, with the split being made at the character or characters specified as a parameter. The result of this method is an array with each element containing one of the split pieces. For example, the following string:

```
var myListString = "apple, banana, peach, orange"
```

could be split into an array in which each element contains a different fruit, like this:

```
var myFruitArray = myListString.split(", ");
```

How about if your string is this instead?

```
var myListString = "apple, 0.99, banana, 0.50, peach, 0.25, orange, 0.75";
```

The string could, for example, contain both the names and prices of the fruit. How could you split the string, but retrieve only the names of the fruit and not the prices? You could do it without regular expressions, but it would take many lines of code. With regular expressions you can use the same code and just amend the `split()` method's parameter.

Try It Out Splitting the Fruit String

Let's create an example that solves the problem just described — it must split your string, but include only the fruit names, not the prices.

```
<!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">
<body>

<script type="text/JavaScript">
var myListString = "apple, 0.99, banana, 0.50, peach, 0.25, orange, 0.75";
var theRegExp = /[^a-z]+/i;
var myFruitArray = myListString.split(theRegExp);
document.write(myFruitArray.join("<br />"));

</script>
</body>
</html>
```

Save the file as `ch9_examp4.htm` and load it in your browser. You should see the four fruits from your string written out to the page, with each fruit on a separate line.

Within the script block, first you have your string with fruit names and prices.

```
var myListString = "apple, 0.99, banana, 0.50, peach, 0.25, orange, 0.75";
```

How do you split it in such a way that only the fruit names are included? Your first thought might be to use the comma as the `split()` method's parameter, but of course that means you end up with the prices. What you have to ask is, "What is it that's between the items I want?" Or in other words, what is between the fruit names that you can use to define your split? The answer is that various characters are between the names of the fruit, such as a comma, a space, numbers, a full stop, more numbers, and finally another comma. What is it that these things have in common and makes them different from the fruit names that you want? What they have in common is that none of them are letters from a through z. If you say "Split the string at the point where there is a group of characters that are not between a and z," then you get the result you want. Now you know what you need to create your regular expression.

You know that what you want is not the letters a through z, so you start with this:

```
[^a-z]
```

The `^` says "Match any character that does not match those specified inside the square brackets." In this case you've specified a range of characters not to be matched — all the characters between a and z. As specified, this expression will match only one character, whereas you want to split wherever there is a single group of one or more characters that are not between a and z. To do this you need to add the `+` special repetition character, which says "Match one or more of the preceding character or group specified."

```
[^a-z]+
```

The final result is this:

```
var theRegExp = /[^a-z]+/i
```

The `/` and `/` characters mark the start and end of the regular expression whose `RegExp` object is stored as a reference in the variable `theRegExp`. You add the `i` on the end to make the match case-insensitive.

Don't panic if creating regular expressions seems like a frustrating and less-than-obvious process. At first, it takes a lot of trial and error to get it right, but as you get more experienced, you'll find creating them becomes much easier and will enable you to do things that without regular expressions would be either very awkward or virtually impossible.

In the next line of script you pass the `RegExp` object to the `split()` method, which uses it to decide where to split the string.

```
var myFruitArray = myListString.split(theRegExp);
```

After the split, the variable `myFruitArray` will contain an `Array` with each element containing the fruit name, as shown here:

Array Element Index	0	1	2	3
Element value	apple	banana	peach	orange

You then join the string together again using the `Array` object's `join()` methods, which you saw in Chapter 4.

```
document.write(myFruitArray.join("<BR>"))
```

The replace() Method

You've already looked at the syntax and usage of the `replace()` method. However, something unique to the `replace()` method is its ability to replace text based on the groups matched in the regular expression. You do this using the `$` sign and the group's number. Each group in a regular expression is given a number from `1` to `99`; any groups greater than `99` are not accessible. Note that in earlier browsers, groups could only go from `1` to `9` (for example, in IE 5 or earlier or Netscape 4 and earlier). To refer to a group, you write `$` followed by the group's position. For example, if you had the following:

```
var myRegExp = /(\d)(\W)/g;
```

then `$1` refers to the group`(\d)`, and `$2` refers to the group `(\W)`. You've also set the global flag `g` to ensure that all matching patterns are replaced — not just the first one.

You can see this more clearly in the next example. Say you have the following string:

```
var myString = "1999, 2000, 2001";
```

If you wanted to change this to `"the year 1999, the year 2000, the year 2001"`, how could you do it with regular expressions?

First, you need to work out the pattern as a regular expression, in this case four digits.

```
var myRegExp = /\d{4}/g;
```

But given that the year is different every time, how can you substitute the year value into the replaced string?

Well, you change your regular expression so that it's inside a group, as follows:

```
var myRegExp = /(\d{4})/g;
```

Now you can use the group, which has group number `1`, inside the replacement string like this:

```
myString = myString.replace(myRegExp, "the year $1");
```

The variable `myString` now contains the required string `"the year 1999, the year 2000, the year 2001"`.

Let's look at another example in which you want to convert single quotes in text to double quotes. Your test string is this:

```
'Hello World' said Mr. O'Connerly.
He then said 'My Name is O'Connerly, yes that's right, O'Connerly'.
```

One problem that the test string makes clear is that you want to replace the single-quote mark with a double only where it is used in pairs around speech, not when it is acting as an apostrophe, such as in the word `that's`, or when it's part of someone's name, such as in `O'Connerly`.

Let's start by defining the regular expression. First you know that it must include a single quote, as shown in the following code:

```
var myRegExp = /'/;
```

However, as it is this would replace every single quote, which is not what you want.

Looking at the text, you should also notice that quotes are always at the start or end of a word — that is, at a boundary. On first glance it might be easy to assume that it would be a word boundary. However, don't forget that the `'` is a non-word character, so the boundary will be between it and another non-word character, such as a space. So the boundary will be a non-word boundary or, in other words, `\B`.

Therefore, the character pattern you are looking for is either a non-word boundary followed by a single quote or a single quote followed by a non-word boundary. The key is the "or," for which you use | in regular expressions. This leaves your regular expression as the following:

```
var myRegExp = /\B'|'\B/g;
```

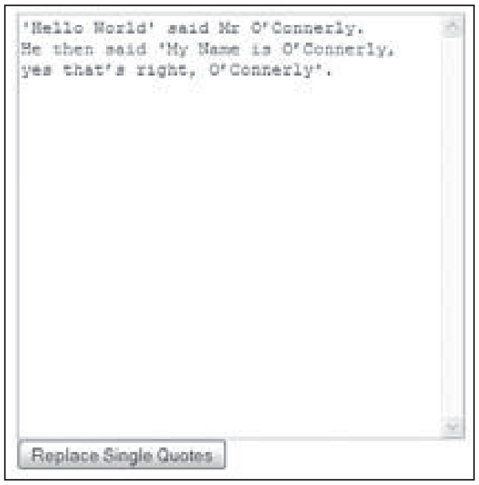
This will match the pattern on the left of the | or the character pattern on the right. You want to replace all the single quotes with double quotes, so the `g` has been added at the end, indicating that a global match should take place.

Try It Out Replacing Single Quotes with Double Quotes

Let's look at an example using the regular expression just defined.

```
<!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">
<head>
<title>example</title>
<script type="text/JavaScript">
function replaceQuote(textAreaControl)
{
   var myText = textAreaControl.value;
   var myRegExp = /\B'|'\B/g;
   myText = myText.replace(myRegExp,'"');
   textAreaControl.value = myText;
}
</script>
</head>
<body>
<form name="form1">
<textarea rows="20" cols="40" name="textarea1">
'Hello World' said Mr O'Connerly.
He then said 'My Name is O'Connerly, yes that's right, O'Connerly'.
</textarea>
<br>
<input type="button" VALUE="Replace Single Quotes" name="buttonSplit"
   onclick="replaceQuote(document.form1.textarea1)">
</form>

</body>
</html>
```

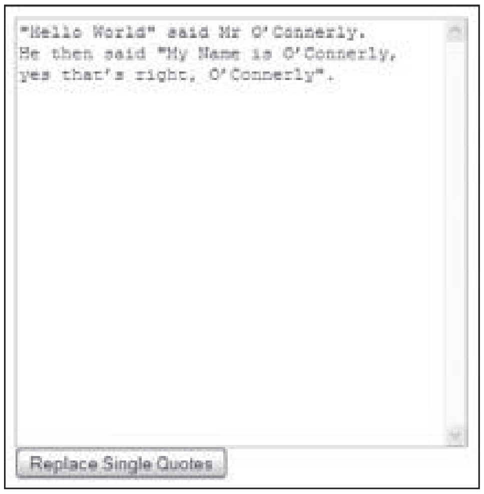
Save the page as `ch9_examp5.htm`. Load the page into your browser and you should see what is shown in Figure 9-13.



```
'Hello World' said Mr O'Connerly.
He then said 'My Name is O'Connerly,
yes that's right, O'Connerly'.
```

Figure 9-13

Click the Replace Single Quotes button to see the single quotes in the text area replaced as in Figure 9-14.



```
"Hello World" said Mr O'Connerly.
He then said "My Name is O'Connerly,
yes that's right, O'Connerly".
```

Figure 9-14

Try entering your own text with single quotes into the text area and check the results.

You can see that by using regular expressions, you have completed a task in a couple of lines of simple code. Without regular expressions, it would probably take four or five times that amount.

Let's look first at the `replaceQuote()` function in the head of the page where all the action is.

```
function replaceQuote(textAreaControl)
{
   var myText = textAreaControl.value;
   var myRegExp = /\B'|'\B/g;
```

```
        myText = myText.replace(myRegExp,'"');
        textAreaControl.value = myText;
    }
```

The function's parameter is the `textarea` object defined further down the page — this is the text area in which you want to replace the single quotes. You can see how the `textarea` object was passed in the button's tag definition.

```
    <input type="button" value="Replace Single Quotes" name="buttonSplit"
      onclick="replaceQuote(document.form1.textarea1)">
```

In the `onclick` event handler, you call `replaceQuote()` and pass `document.form1.textarea1` as the parameter — that is the `textarea` object.

Returning to the function, you get the value of the `textarea` on the first line and place it in the variable `myText`. Then you define your regular expression (as discussed previously), which matches any non-word boundary followed by a single quote or any single quote followed by a non-word boundary. For example, `'H` will match, as will `H'`, but `O'R` won't, because the quote is between two word boundaries. Don't forget that a word boundary is the position between the start or end of a word and a non-word character, such as a space or punctuation mark.

In the function's final two lines, you first use the `replace()` method to do the character pattern search and replace, and finally you set the `textarea` object's value to the changed string.

The search() Method

The `search()` method enables you to search a string for a pattern of characters. If the pattern is found, the character position at which it was found is returned, otherwise `-1` is returned. The method takes only one parameter, the `RegExp` object you have created.

Although for basic searches the `indexOf()` method is fine, if you want more complex searches, such as a search for a pattern of any digits or one in which a word must be in between a certain boundary, then `search()` provides a much more powerful and flexible, but sometimes more complex, approach.

In the following example, you want to find out if the word `Java` is contained within the string. However, you want to look just for `Java` as a whole word, not part of another word such as `JavaScript`.

```
    var myString = "Beginning JavaScript, Beginning Java 2, Professional JavaScript";
    var myRegExp = /\bJava\b/i;
    alert(myString.search(myRegExp));
```

First, you have defined your string, and then you've created your regular expression. You want to find the character pattern `Java` when it's on its own between two word boundaries. You've made your search case-insensitive by adding the `i` after the regular expression. Note that with the `search()` method, the `g` for global is not relevant, and its use has no effect.

On the final line, you output the position at which the search has located the pattern, in this case `32`.

The match() Method

The `match()` method is very similar to the `search()` method, except that instead of returning the position at which a match was found, it returns an array. Each element of the array contains the text of a match made.

For example, if you had the string

```
var myString = "The years were 1999, 2000 and 2001";
```

and wanted to extract the years from this string, you could do so using the `match()` method. To match each year, you are looking for four digits in between word boundaries. This requirement translates to the following regular expression:

```
var myRegExp = /\b\d{4}\b/g;
```

You want to match all the years so the `g` has been added to the end for a global search.

To do the match and store the results, you use the `match()` method and store the `Array` object it returns in a variable.

```
var resultsArray = myString.match(myRegExp);
```

To prove it has worked, let's use some code to output each item in the array. You've added an `if` statement to double-check that the results array actually contains an array. If no matches were made, the results array will contain `null` — doing `if (resultsArray)` will return `true` if the variable has a value and not `null`.

```
if (resultsArray)
{
  var indexCounter;
  for (indexCounter = 0; indexCounter < resultsArray.length; indexCounter++)
  {
     alert(resultsArray[indexCounter]);
  }
}
```

This would result in three alert boxes containing the numbers `1999`, `2000`, and `2001`.

Try It Out Splitting HTML

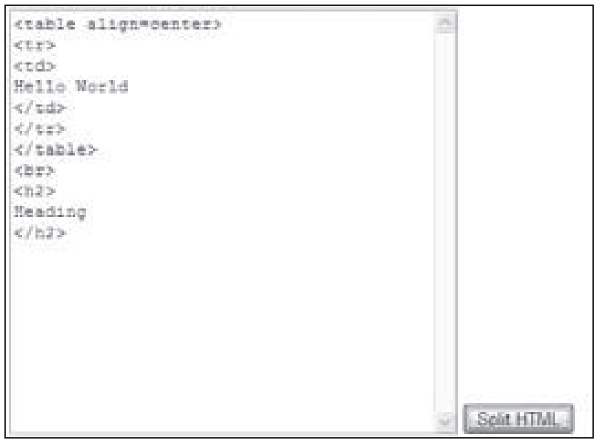
In the next example, you want to take a string of HTML and split it into its component parts. For example, you want the HTML `<P>Hello</P>` to become an array, with the elements having the following contents:

<P>	Hello	</P>

```
<!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">
```

```
<head>
<title>example 6</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/JavaScript">
function button1_onclick()
{
   var myString = "<table align=center><tr><td>";
   myString = myString + "Hello World</td></tr></table>";
   myString = myString +"<br><h2>Heading</h2>";
   var myRegExp = /<[^>\r\n]+>|[^<>\r\n]+/g;
   var resultsArray = myString.match(myRegExp);
   document.form1.textarea1.value = "";
   document.form1.textarea1.value = resultsArray.join ("\r\n");
}
</script>
</head>
<body>
<form name="form1">
   <textarea rows="20" cols="40" name="textarea1"></textarea>
   <input type="button" value="Split HTML" name="button1"
      onclick="return button1_onclick();">
</form>

</body>
</html>
```

Save this file as `ch9_examp6.htm`. When you load the page into your browser and click the Split HTML button, a string of HTML is split, and each tag is placed on a separate line in the text area, as shown in Figure 9-15.



```
<table align=center>
<tr>
<td>
Hello World
</td>
</tr>
</table>
<br>
<h2>
Heading
</h2>
```

Figure 9-15

The function `button1_onclick()` defined at the top of the page fires when the Split HTML button is clicked. At the top, the following lines define the string of HTML that you want to split:

```
function button1_onclick()
{
   var myString = "<table align=center><tr><td>";
```

```
        myString = myString + "Hello World</td></tr></table>";
        myString = myString +"<br><h2>Heading</h2>";
```

Next you create your `RegExp` object and initialize it to your regular expression.

```
    var myRegExp = /<[^>\r\n]+>|[^<>\r\n]+/g;
```

Let's break it down to see what pattern you're trying to match. First, note that the pattern is broken up by an alternation symbol: `|`. This means that you want the pattern on the left or the right of this symbol. You'll look at these patterns separately. On the left, you have the following:

- ❑ The pattern must start with a `<`.
- ❑ In `[^>\r\n]+`, you specify that you want one or more of any character except the `>` or a `\r` (carriage return) or a `\n` (linefeed).
- ❑ `>` specifies that the pattern must end with a `>`.

On the right, you have only the following:

- ❑ `[^<>\r\n]+` specifies that the pattern is one or more of any character, so long as that character is not a `<`, `>`, `\r`, or `\n`. This will match plain text.

After the regular expression definition you have a `g`, which specifies that this is a global match.

So the `<[^>\r\n]+>` regular expression will match any start or close tags, such as `<p>` or `</p>`. The alternative pattern is `[^<>\r\n]+`, which will match any character pattern that is not an opening or closing tag.

In the following line, you assign the `resultsArray` variable to the `Array` object returned by the `match()` method:

```
    var resultsArray = myString.match(myRegExp);
```

The remainder of the code deals with populating the text area with the split HTML. You use the `Array` object's `join()` method to join all the array's elements into one string with each element separated by a `\r\n` character, so that each tag or piece of text goes on a separate line, as shown in the following:

```
        document.form1.textarea1.value = "";
        document.form1.textarea1.value = resultsArray.join("\r\n");
    }
```

Using the RegExp Object's Constructor

So far you've been creating `RegExp` objects using the `/` and `/` characters to define the start and end of the regular expression, as shown in the following example:

```
    var myRegExp = /[a-z]/;
```

Although this is the generally preferred method, it was briefly mentioned that a `RegExp` object can also be created by means of the `RegExp()` constructor. You might use the first way most of the time. However, there are occasions, as you'll see in the trivia quiz shortly, when the second way of creating a `RegExp` object is necessary (for example, when a regular expression is to be constructed from user input).

As an example, the preceding regular expression could equally well be defined as

```
var myRegExp = new RegExp("[a-z]");
```

Here you pass the regular expression as a string parameter to the `RegExp()` constructor function.

A very important difference when you are using this method is in how you use special regular expression characters, such as `\b`, that have a backward slash in front of them. The problem is that the backward slash indicates an escape character in JavaScript strings — for example, you may use `\b`, which means a backspace. To differentiate between `\b` meaning a backspace in a string and the `\b` special character in a regular expression, you have to put another backward slash in front of the regular expression special character. So `\b` becomes `\\b` when you mean the regular expression `\b` that matches a word boundary, rather than a backspace character.

For example, say you have defined your `RegExp` object using the following:

```
var myRegExp = /\b/;
```

To declare it using the `RegExp()` constructor, you would need to write this:

```
var myRegExp = new RegExp("\\b");
```

and not this:

```
var myRegExp = new RegExp("\b");
```

All special regular expression characters, such as `\w`, `\b`, `\d`, and so on, must have an extra `\` in front when you create them using `RegExp()`.

When you defined regular expressions with the `/` and `/` method, you could add after the final `/` the special flags `m`, `g`, and `i` to indicate that the pattern matching should be multi-line, global, or case-insensitive, respectively. When using the `RegExp()` constructor, how can you do the same thing?

Easy. The optional second parameter of the `RegExp()` constructor takes the flags that specify a global or case-insensitive match. For example, this will do a global case-insensitive pattern match:

```
var myRegExp = new RegExp("hello\\b","gi");
```

You can specify just one of the flags if you wish — such as the following:

```
var myRegExp = new RegExp("hello\\b","i");
```

or

```
var myRegExp = new RegExp("hello\\b","g");
```

Try It Out Form Validation Module

In this Try It Out, you'll create a set of useful JavaScript functions that use regular expressions to validate the following:

- ❑ Telephone numbers
- ❑ Postal codes
- ❑ E-mail addresses

The validation only checks the format. So, for example, it can't check that the telephone number actually exists, only that it would be valid if it did.

First is the `.js` code file with the input validation code. Please note that the lines of code in the following block are too wide for the book — make sure each regular expression is contained on one line.

```
function isValidTelephoneNumber( telephoneNumber )
{
               var telRegExp = /^(\+\d{1,3} ?)?(\(\d{1,5}\)|\d{1,5}) ?\d{3}
 ?\d{0,7}( (x|xtn|ext|extn|pax|pbx|extension)?\.? ?\d{2-5})?$/i
               return telRegExp.test( telephoneNumber );
}

function isValidPostalCode( postalCode )
{
        var pcodeRegExp = /^(\d{5}(-\d{4})?|([a-z][a-z]?\d\d?|[a-z{2}\d[a-z])
    ?\d[a-z][a-z])$/i
        return pcodeRegExp.test( postalCode );
}

function isValidEmail( emailAddress )
{
        var emailRegExp = /^(([^<>()\[\]\\.,;:@"\x00-\x20\x7F]|\\.)+|("""
([^\x0A\x0D"\\]|\\\\)+"""))@(([a-z]|#\d+?)([a-z0-9-]|#\d+?)*
([a-z0-9]|#\d+?)\.)+([a-z]{2,4})$/i
       return emailRegExp.test( emailAddress );
}
```

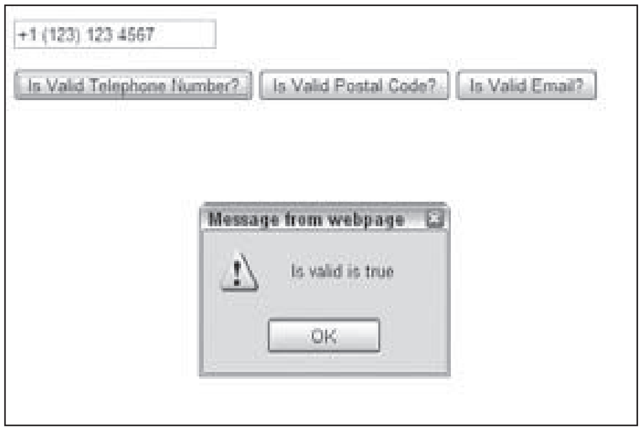
Save this as `ch9_examp7_module.js`.

To test the code, you need a simple page with a text box and three buttons that validate the telephone number, postal code, or e-mail address.

```
<!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">
<head>
<title>example 7</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript" src="ch9_examp7_module.js"></script>
</head>
<body>
<form name="form1">
  <p>
    <label>
```

```
      <input type="text" name="txtString" id="txtString" />
    </label>
  </p>
  <p>
    <label>
      <input type="button" name="cmdIsValidTelephoneNumber"
    id="cmdIsValidTelephoneNumber"
    value="Is Valid Telephone Number?"
onclick="alert('Is valid is ' +
                    isValidTelephoneNumber( document.form1.txtString.value ))"
     />

      <input type="button" name="cmdIsValidPostalCode"
             id="cmdIsValidPostalCode"
             value="Is Valid Postal Code?"
             onclick="alert('Is valid is '
+ isValidPostalCode( document.form1.txtString.value ))" />
      <input type="button" name="cmdIsEmailValid" id="cmdIsEmailValid"
value="Is Valid Email?"
onclick="alert('Is valid is '
+ isValidEmail( document.form1.txtString.value ))" />
    </label>
  </p>
</form>

</body>
</html>
```

Save this as `ch9_examp7.htm` and load it into your browser, and you'll see a page with a text box and three buttons. Enter a valid telephone number (the example uses +1 (123) 123 4567), click the Is Valid Telephone Number button, and the screen shown in Figure 9-16 is displayed.



Is valid is true

Figure 9-16

If you enter an invalid phone number, the result would be `Is Valid is false`. This is pretty basic but it's sufficient for testing your code.

The actual code is very simple, but the regular expressions are tricky to create, so let's look at those in depth starting with telephone number validation.

Telephone Number Validation

Telephone numbers are more of a challenge to validate. The problems are:

❑ Phone numbers differ from country to country.

❑ There are different ways of entering a valid number (for example, adding the national or international code or not).

For this regular expression, you need to specify more than just the valid characters; you also need to specify the format of the data. For example, all of the following are valid:

+1 (123) 123 4567

+1123123 456

+44 (123) 123 4567

+44 (123) 123 4567 ext 123

+44 20 7893 4567

The variations that our regular expression needs to deal with (optionally separated by spaces) are shown in the following table:

The international number	"+" followed by one to three digits (optional)
The local area code	Two to five digits, sometimes in parentheses (compulsory)
The actual subscriber number	Three to 10 digits, sometimes with spaces (compulsory)
An extension number	Two to five digits, preceded by `x`, `xtn`, `extn`, `pax`, `pbx`, or `extension`, and sometimes in parentheses

Obviously, there will be countries where this won't work, which is something you'd need to deal with based on where your customers and partners would be. The following regular expression is rather complex, its length meant it had to be split across two lines; make sure you type it in on one line.

```
^(\+\d{1,3} ?)?(\(\d{1,5}\)|\d{1,5}) ?\d{3} ?\d{0,7}
( (x|xtn|ext|extn|pax|pbx|extension)?\.? ?\d{2-5})?$
```

You will need to set the case-insensitive flag with this, as well as the explicit capture option. Although this seems complex, if broken down, it's quite straightforward.

Let's start with the pattern that matches an international dialing code:

```
(\+\d{1,3} ?)?
```

So far, you've matching a plus sign `(\+)` followed by one to three digits `(\d{1,3})` and an optional space (`?`). Remember that since the + character is a special character, you add a \ character in front of it to specify that you mean an actual + character. The characters are wrapped inside parentheses to specify a group of characters. You allow an optional space and match this entire group of characters zero or one times, as indicated by the ? character after the closing parenthesis of the group.

Next is the pattern to match an area code:

```
(\(\d{1,5}\)|\d{1,5})
```

This pattern is contained in parentheses, which designate it as a group of characters, and matches either one to five digits in parentheses (`(\d{1,5})`) or just one to five digits (`\d{1,5}`). Again, since the parenthesis characters are special characters in regular expression syntax and you want to match actual parentheses, you need the \ character in front of them. Also note the use of the pipe symbol (|), which means "OR" or "match either of these two patterns."

Next, let's match the subscriber number:

```
 ?\d{3,4} ?\d{0,7}
```

Note that there is a space before the first ? symbol: this space and question mark mean "match zero or one space." This is followed by three or four digits (`\d{3,4}`) — although there are always three digits in the U.S., there are often four in the UK. Then there's another "zero or one space," and finally between zero and seven digits `(\d{0,7})`.

Finally, add the part to cope with an optional extension number:

```
( (x|xtn|ext|extn|extension)?\.? ?\d{2-5})?
```

This group is optional, since its parentheses are followed by a question mark. The group itself checks for a space, optionally followed by `x`, `ext`, `xtn`, `extn`, or `extension`, followed by zero or one periods (note the \ character, since `.` is a special character in regular expression syntax), followed by zero or one space, followed by between two and five digits. Putting these four patterns together, you can construct the entire regular expression, apart from the surrounding syntax. The regular expression starts with `^` and ends with `$`. The `^` character specifies that the pattern must be matched at the beginning of the string, and the `$` character specifies that the pattern must be matched at the end of the string. This means that the string must match the pattern completely; it cannot contain any other characters before or after the pattern that is matched.

Therefore, with the regular expression explained, you can now add it to your JavaScript module `ch9_examp7_module.js` as follows:

```
function isValidTelephoneNumber( telephoneNumber )
{
                var telRegExp = /^(\+\d{1,3} ?)?
                (\(\d{1,5}\)|\d{1,5}) ?\d{3} ?\d{0,7}
               ( (x|xtn|ext|extn|pax|pbx|extension)?
                \.? ?\d{2-5})?$/i
                return telRegExp.test( telephoneNumber );
}
```

Note in this case that it is important to set the case-insensitive flag by adding an `i` on the end of the expression definition; otherwise, the regular expression could fail to match the `ext` parts. Please also note that the regular expression itself must be on one line in your code — it's shown in four lines here due to the page-width restrictions of this book.

Validating a Postal Code

We just about managed to check worldwide telephone numbers, but doing the same for postal codes would be something of a major challenge. Instead, you'll create a function that only checks for U.S. zip codes and UK postcodes. If you needed to check for other countries, the code would need modifying. You may find that checking more than one or two postal codes in one regular expression begins to get unmanageable, and it may well be easier to have an individual regular expression for each country's postal code you need to check. For this purpose though, let's combine the regular expression for the UK and the U.S.:

```
^(\d{5}(-\d{4})?|[a-z][a-z]?\d\d? ?\d[a-z][a-z])$
```

This is actually in two parts: The first part checks for zip codes, and the second part checks UK postcodes. Start by looking at the zip code part.

Zip codes can be represented in one of two formats: as five digits (12345), or five digits followed by a dash and four digits (12345-1234). The zip code regular expression to match these is as follows:

```
\d{5}(-\d{4})?
```

This matches five digits, followed by an optional non-capturing group that matches a dash, followed by four digits.

For a regular expression that covers UK postcodes, let's consider their various formats. UK postcode formats are one or two letters followed by either one or two digits, followed by an optional space, followed by a digit, and then two letters. Additionally, some central London postcodes look like this: SE2V 3ER, with a letter at the end of the first part. Currently, it is only some of those postcodes starting with SE, WC, and W, but that may change. Valid examples of UK postcode include: CH3 9DR, PR29 1XX, M27 1AE, WC1V 2ER, and C27 3AH.

Based on this, the required pattern is as follows:

```
([a-z][a-z]?\d\d?|[a-z]{2}\d[a-z]) ?\d[a-z][a-z]
```

These two patterns are combined using the | character to "match one or the other" and grouped using parentheses. You then add the ^ character at the start and the $ character at the end of the pattern to be sure that the only information in the string is the postal code. Although postal codes should be uppercase, it is still valid for them to be lowercase, so you also set the case-insensitive option as follows when you use the regular expression:

```
^(\d{5}(-\d{4})?|([a-z][a-z]?\d\d?|[a-z{2}\d[a-z]) ?\d[a-z][a-z])$
```

The following function needed for your validation module is much the same as it was with the previous example:

```
function isValidPostalCode( postalCode )
{
        var pcodeRegExp = /^(\d{5}(-\d{4})?|
([a-z][a-z]?\d\d?|[a-z{2}\d[a-z]) ?\d[a-z][a-z])$/i
        return pcodeRegExp.test( postalCode );
}
```

Again please remember that the regular expression must be on one line in your code.

Validating an E-mail Address

Before working on a regular expression to match e-mail addresses, you need to look at the types of valid e-mail addresses you can have. For example:

- someone@mailserver.com
- someone@mailserver.info
- someone.something@mailserver.com
- someone.something@subdomain.mailserver.com
- someone@mailserver.co.uk
- someone@subdomain.mailserver.co.uk
- someone.something@mailserver.co.uk
- someone@mailserver.org.uk
- some.one@subdomain.mailserver.org.uk

Also, if you examine the SMTP RFC (`http://www.ietf.org/rfc/rfc0821.txt`), you can have the following:

- someone@123.113.209.32
- """Paul Wilton"""@somedomain.com

That's quite a list and contains many variations to cope with. It's best to start by breaking it down. First, there are a couple of things to note about the two immediately above. The latter two versions are exceptionally rate and not provided for in the regular expression you'll create.

You need to break up the e-mail address into separate parts, and you will look at the part after the @ symbol, first.

Validating a Domain Name

Everything has become more complicated since Unicode domain names have been allowed. However, the e-mail RFC still doesn't allow these, so let's stick with the traditional definition of how a domain can be described using ASCII. A domain name consists of a dot-separated list of words, with the last word being between two and four characters long. It was often the case that if a two-letter country word was used, there would be at least two parts to the domain name before it: a grouping domain (`.co`, `.ac`, and so on) and a specific domain name. However, with the advent of the `.tv` names, this is no longer the case. You could make this very specific and provide for the allowed top-level domains (TLDs), but that would make the regular expression very large, and it would be more productive to perform a DNS lookup instead.

Each part of a domain name has certain rules it must follow. It can contain any letter or number or a hyphen, but it must start with a letter. The exception is that, at any point in the domain name, you can use a #, followed by a number, which represents the ASCII code for that letter, or in Unicode, the 16-bit Unicode value. Knowing this, let's begin to build up the regular expression, first with the name part, assuming that the case-insensitive flag will be set later in the code.

```
([a-z]|#\d+)([a-z0-9-]|#\d+)*([a-z0-9]|#\d+)
```

This breaks the domain into three parts. The RFC doesn't specify how many digits can be contained here, so neither will we. The first part must only contain an ASCII letter; the second must contain zero or more of a letter, number, or hyphen; and the third must contain either a letter or number. The top-level domain has more restrictions, as shown here:

```
[a-z]{2,4}
```

This restricts you to a two, three, or four letter top-level domain. So, putting it all together, with the periods you end up with this:

```
^(([a-z]|#\d+?)([a-z0-9-]|#\d+?)*([a-z0-9]|#\d+?)\.)+([a-z]{2,4})$
```

Again, the domain name is anchored at the beginning and end of the string. The first thing is to add an extra group to allow one or more `name.` portions and then anchor a two-to-four-letter domain name at the end in its own group. We have also made most of the wildcards lazy. Because much of the pattern is similar, it makes sense to do this; otherwise, it would require too much backtracking. However, you have left the second group with a "greedy" wildcard: It will match as much as it can, up until it reaches a character that does not match. Then it will only backtrack one position to attempt the third group match. This is more resource-efficient than a lazy match is in this case, because it could be constantly going forward to attempt the match. One backtrack per name is an acceptable amount of extra processing.

Validating a Person's Address

You can now attempt to validate the part before the `@` sign. The RFC specifies that it can contain any ASCII character with a code in the range from 33 to 126. You are assuming that you are matching against ASCII only, so you can assume that there are only 128 characters that the engine will match against. This being the case, it is simpler to just exclude the required values as follows:

```
[^<>()\[\],;:@"\x00-\x20\x7F]+
```

Using this, you're saying that you allow any number of characters, as long as none of them are those contained within the square brackets. The `[`, `]`, and `\` characters have to be escaped. However, the RFC allows for other kinds of matches.

Validating the Complete Address

Now that you have seen all the previous sections, you can build up a regular expression for the entire e-mail address. First, here's everything up to and including the `@` sign:

```
^([^<>()\[\],;:@"\x00-\x20\x7F]|\\.)+@
```

That was straightforward. Now for the domain name part.

```
^([^<>()\[\],;:@"\x00-\x20\x7F]|\\.)+@(([a-z]|#\d+?)([a-z0-9-]
|#\d+?)*([a-z0-9]|#\d+?)\.)+([a-z]{2,4})$
```

We've had to put it on two lines to fit this book's page width, but in your code this must all be on one line.

Finally, let's create the function for the JavaScript module.

```
function isValidEmail( emailAddress )
{
        var emailRegExp = /^([^<>()\[\],;:@"\x00-\x20\x7F]|\\.)+
@(([a-z]|#\d+?)([a-z0-9-]|
#\d+?)*([a-z0-9]|#\d+?)\.)+([a-z]{2,4})$/i
        return emailRegExp.test( emailAddress );
}
```

Please note the regular expression must all be on one line in your code.

With the module completed, let's take a look at the code to test the module.

First, the module is linked to the test page like this:

```
<script type="text/javascript" src="ch9_examp7_module.js"></script>
```

Then each of the three test buttons has its click events linked to the validation functions in the module as follows:

```
<input type="button" name="cmdIsValidTelephoneNumber"
         id="cmdIsValidTelephoneNumber"
 value="Is Valid Telephone Number?"
onclick="alert('Is valid is ' +
isValidTelephoneNumber( document.form1.txtString.value ))" />
      <input type="button" name="cmdIsValidPostalCode" id="cmdIsValidPostalCode"
 value="Is Valid Postal Code?"
onclick="alert('Is valid is ' +
isValidPostalCode( document.form1.txtString.value ))" />
      <input type="button" name="cmdIsEmailValid" id="cmdIsEmailValid"
value="Is Valid Email?"
onclick="alert('Is valid is ' + isValidEmail( document.form1.txtString.value ))" />
```

So taking telephone validation test button, an `onclick` event handler is added.

```
onclick="alert('Is valid is ' +
isValidTelephoneNumber( document.form1.txtString.value ))"
```

This shows an alert box returning the true or false value from the `isValidTelephoneNumber()` function in your validation module. In a non-test situation, you'd want a more user-friendly message. The other two test buttons work in the same way but just call different validation functions.

Summary

In this chapter you've looked at some more advanced methods of the `String` object and how you can optimize their use with regular expressions.

To recap, the chapter covered the following points:

- The `split()` method splits a single string into an array of strings. You pass a string or a regular expression to the method that determines where the split occurs.
- The `replace()` method enables you to replace a pattern of characters with another pattern that you specify as a second parameter.
- The `search()` method returns the character position of the first pattern matching the one given as a parameter.
- The `match()` method matches patterns, returning the text of the matches in an array.
- Regular expressions enable you to define a pattern of characters that you want to match. Using this pattern, you can perform splits, searches, text replacement, and matches on strings.
- In JavaScript the regular expressions are in the form of a `RegExp` object. You can create a `RegExp` object using either `myRegExp = /myRegularExpression/` or `myRegExp = new RegExp("myRegularExpression")`. The second form requires that certain special characters that normally have a single `\` in front now have two.
- The `g` and `i` characters at the end of a regular expression (as in, for example, `myRegExp = /Pattern/gi;`) ensure that a global and case-insensitive match is made.
- As well as specifying actual characters, regular expressions have certain groups of special characters, which allow any of certain groups of characters, such as digits, words, or non-word characters, to be matched.
- Special characters can also be used to specify pattern or character repetition. Additionally, you can specify what the pattern boundaries must be, for example at the beginning or end of the string, or next to a word or non-word boundary.
- Finally, you can define groups of characters that can be used later in the regular expression or in the results of using the expression with the `replace()` method.

In the next chapter, you'll take a look at using and manipulating dates and times using JavaScript, and time conversion between different world time zones. Also covered is how to create a timer that executes code at regular intervals after the page is loaded.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

1. What problem does the following code solve?

```
var myString = "This sentence has has a fault and and we need to fix it."
var myRegExp = /(\b\w+\b) \1/g;
myString = myString.replace(myRegExp,"$1");
```

Now imagine that you change that code, so that you create the `RegExp` object like this:

```
var myRegExp = new RegExp("(\b\w+\b) \1");
```

Why would this not work, and how could you rectify the problem?

2. Write a regular expression that finds all of the occurrences of the word "a" in the following sentence and replaces them with "the":

   "a dog walked in off a street and ordered a finest beer"

   The sentence should become:

   "the dog walked in off the street and ordered the finest beer"

3. Imagine you have a web site with a message board. Write a regular expression that would remove barred words. (You can make up your own words!)

Date, Time, and Timers

Chapter 5 discussed that the concepts of date and time are embodied in JavaScript through the `Date` object. You looked at some of the properties and methods of the `Date` object, including the following:

- ❑ The methods `getDate()`, `getDay()`, `getMonth()`, and `getFullYear()` enable you to retrieve date values from inside a `Date` object.
- ❑ The `setDate()`, `setMonth()`, and `setFullYear()` methods enable you to set the date values of an existing `Date` object.
- ❑ The `getHours()`, `getMinutes()`, `getSeconds()`, and `getMilliseconds()` methods retrieve the time values in a `Date` object.
- ❑ The `setHours()`, `setMinutes()`, `setSeconds()`, and `setMilliseconds()` methods enable you to set the time values of an existing `Date` object.

One thing not covered in that chapter is the idea that the time depends on your location around the world. In this chapter you'll be correcting that omission by looking at date and time in relation to *world time*.

For example, imagine you have a chat room on your web site and want to organize a chat for a certain date and time. Simply stating 15:30 is not good enough if your web site attracts international visitors. The time 15:30 could be Eastern Standard Time, Pacific Standard Time, the time in the United Kingdom, or even the time in Kuala Lumpur. You could of course say 15:30 EST and let your visitors work out what that means, but even that isn't foolproof. There is an EST in Australia as well as in the United States. Wouldn't it be great if you could automatically convert the time to the user's time zone? In this chapter you'll see how.

In addition to looking at world time, you'll also be looking at how to create a *timer* in a web page. You'll see that by using the timer you can trigger code, either at regular intervals or just once (for example, five seconds after the page has loaded). You'll see how you can use timers to add a real-time clock to a web page and how to create scrolling text in the status bar. Timers can also be useful for creating animations or special effects in your web applications. Finally, you'll be using the timer to enable the users of your trivia quiz to give themselves a time limit for answering the questions.

World Time

The concept of *now* means the same point in time everywhere in the world. However, when that point in time is represented by numbers, those numbers differ depending on where you are. What is needed is a standard number to represent that moment in time. This is achieved through Coordinated Universal Time (UTC), which is an international basis of civil and scientific time and was implemented in 1964. It was previously known as GMT (Greenwich Mean Time), and, indeed, at 0:00 UTC it is midnight in Greenwich, London.

The following table shows local times around the world at 0:00 UTC time.

San ÅtFrancisco	**New York (EST)**	**Greenwich, London**	**Berlin, Germany**	**Tokyo, Japan**
4:00 pm	7:00 pm	0:00 (midnight)	1:00 am	9:00 am

Note that the times given are winter times — no daylight savings hours are taken into account.

The support for UTC in JavaScript comes from a number of methods of the `Date` object that are similar to those you have already seen. For each of the set-date- and get-date–type methods you've seen so far, there is a UTC equivalent. For example, whereas `setHours()` sets the local hour in a `Date` object, `setUTCHours()` does the same thing for UTC time. You'll be looking at these methods in more detail in the next section.

In addition, three more methods of the `Date` object involve world time.

You have the methods `toUTCString()` and `toLocaleString()`, which return the date and time stored in the `Date` object as a string based on either UTC or local time. Most modern browsers also have these additional methods: `toLocaleTimeString()`, `toTimeString()`, `toLocaleDateString()`, and `toDateString()`.

If you simply want to find out the difference in minutes between the current locale's time and UTC, you can use the `getTimezoneOffset()` method. If the time zone is behind UTC, such as in the United States, it will return a positive number. If the time zone is ahead, such as in Australia or Japan, it will return a negative number.

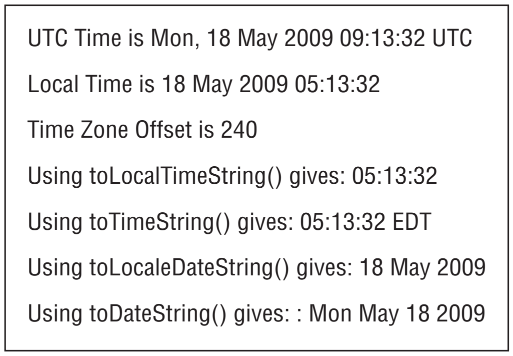
Try It Out The World Time Method of the Date Object

In the following code you use the `toLocaleString()`, `toUTCString()`, `getTimezoneOffset()`, `toLocaleTimeString()`, `toTimeString()`, `toLocaleDateString()`, and `toDateString()` methods and write their values out to the page.

```
<!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">

<head>
<title>example 1</title>

</head>
```

```
<body>
<div id="DisplayResultsDiv"></div>

<script type="text/javascript">
  var localTime = new Date();
  var resultsHTML = '<p>UTC Time is ' + localTime.toUTCString() + '</p>';
  resultsHTML += 'Local Time is ' + localTime.toLocaleString() + '</p>';

  resultsHTML += '<p>Time Zone Offset is ' + localTime.getTimezoneOffset() +
    '</p>';
  resultsHTML += '<p>Using toLocalTimeString() gives: '
                  + localTime.toLocaleTimeString() + '</p>';
  resultsHTML += '<p>Using toTimeString() gives: '
                  + localTime.toTimeString() + '</p>';
  resultsHTML += '<p>Using toLocaleDateString() gives: '
                  + localTime.toLocaleDateString() + '</p>';
  resultsHTML += '<p>Using toDateString() gives: : '
     + localTime.toDateString() + '</p>';
  document.getElementById('DisplayResultsDiv').innerHTML = resultsHTML;

</script>

</body>
</html>
```

Save this as `timetest.htm` and load it into your browser. What you see, of course, depends on which time zone your computer is set to, but your browser should show something similar to Figure 10-1.



```
UTC Time is Mon, 18 May 2009 09:13:32 UTC

Local Time is 18 May 2009 05:13:32

Time Zone Offset is 240

Using toLocalTimeString() gives: 05:13:32

Using toTimeString() gives: 05:13:32 EDT

Using toLocaleDateString() gives: 18 May 2009

Using toDateString() gives: : Mon May 18 2009
```

Figure 10-1

Here the computer's time is set to 05:13:32 a.m. on May 18, 2009, in America's Eastern Standard Time (for example, New York).

So how does this work? At the top of the page's script block, you have just:

```
var localTime = new Date();
```

This creates a new `Date` object and initializes it to the current date and time based on the client computer's clock. (Note that the `Date` object simply stores the number of milliseconds between the date and time on your computer's clock and midnight UTC on January 1, 1970.)

Within the rest of the script block, you obtain the results from various time and date functions. The results are stored in variable `resultsHTML`, and this is then displayed in the page using the last line and the `innerHTML` property.

In the following line, you store the string returned by the `toUTCString()` method in the `resultsHTML` variable:

```
var resultsHTML = '<p>UTC Time is ' + localTime.toUTCString() + '</p>';
```

This converts the date and time stored inside the `localTime Date` object to the equivalent UTC date and time.

Then the following line stores a string with the local date and time value:

```
resultsHTML += 'Local Time is ' + localTime.toLocaleString() + '</p>';
```

Since this time is just based on the user's computer's clock, the string returned by this method also adjusts for Daylight Savings Time (as long as the clock adjusts for it).

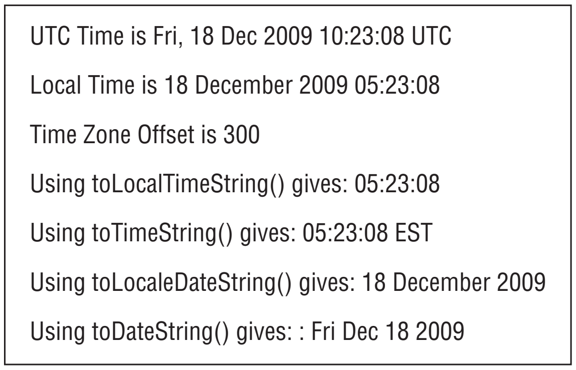
Next, this code stores a string with the difference, in minutes, between the local time zone's time and that of UTC.

```
resultsHTML += '<p>Time Zone Offset is ' + localTime.getTimezoneOffset() +
  '</p>';
```

You may notice in Figure 10-1 that the difference between New York time and UTC time is written to be 240 minutes, or 4 hours. Yet in the previous table, you saw that New York time is 5 hours behind UTC. So what is happening?

Well, in New York on May 18, daylight savings hours are in use. While in the summer it's 8:00 p.m. in New York when it's 0:00 UTC, in the winter it's 7:00 p.m. in New York when it's 0:00 UTC. Therefore, in the summer the `getTimezoneOffset()` method returns `240`, whereas in the winter the `getTimezoneOffset()` method returns `300`.

To illustrate this, compare Figure 10-1 to Figure 10-2, where the date on the computer's clock has been advanced to December, which is in the winter when daylight savings is not in effect:



```
UTC Time is Fri, 18 Dec 2009 10:23:08 UTC

Local Time is 18 December 2009 05:23:08

Time Zone Offset is 300

Using toLocalTimeString() gives: 05:23:08

Using toTimeString() gives: 05:23:08 EST

Using toLocaleDateString() gives: 18 December 2009

Using toDateString() gives: : Fri Dec 18 2009
```

Figure 10-2

The next two methods are `toLocaleTimeString()` and `toTimeString()`, as follows:

```
resultsHTML += '<p>Using toLocalTimeString() gives: ' +
                  localTime.toLocaleTimeString() + '</p>';
resultsHTML += '<p>Using toTimeString() gives: ' +
                  localTime.toTimeString() + '</p>';
```

These methods display just the time part of the date and time held in the `Date` object. The `toLocaleTimeString()` method displays the time as specified by the user on his computer. The second method displays the time but also gives an indication of the time zone (in the example, EST for Eastern Standard Time in America).

The final two methods display the date part of the date and time. The `toLocaleDateString()` displays the date in the format the user has specified on his computer. On Windows operating systems, this is set in the regional settings of the PC's Control Panel. However, because it relies on the user's PC setup, the look of the date varies from computer to computer. The `toDateString()` method displays the current date contained in the PC date in a standard format.

Of course, this example relies on the fact that the user's computer's clock is set correctly, not something you can be 100 percent sure of — it's amazing how many users have their local time zone settings set completely wrong.

Setting and Getting a Date Object's UTC Date and Time

When you create a new `Date` object, you can either initialize it with a value or let JavaScript set it to the current date and time. Either way, JavaScript assumes you are setting the *local* time values. If you want to specify UTC time, you need to use the `setUTC` type methods, such as `setUTCHours()`.

The following are the seven methods for setting UTC date and time:

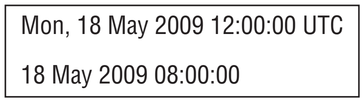
- `setUTCDate()`
- `setUTCFullYear()`
- `setUTCHours()`
- `setUTCMilliseconds()`
- `setUTCMinutes()`
- `setUTCMonth()`
- `setUTCSeconds()`

The names pretty much give away exactly what each of the methods does, so let's launch straight into a simple example, which sets the UTC time.

```
<!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">

<head>
<title>example 2</title>

</head>
<body>
<div id="DisplayResultsDiv"></div>

<script type="text/javascript">
```

```
        var myDate = new Date();
        myDate.setUTCHours(12);
        myDate.setUTCMinutes(0);
        myDate.setUTCSeconds(0);
        var resultsHTML = '<p>' +  myDate.toUTCString() + '</p>';
        resultsHTML += '<p>' +  myDate.toLocaleString() + '</p>';

        document.getElementById('DisplayResultsDiv').innerHTML = resultsHTML;

</script>

</body>
</html>
```

Save this as `settimetest.htm`. When you load it in your browser, you should see something like that shown in Figure 10-3 in your web page, although the actual date will depend on the current date and where you are in the world.



```
Mon, 18 May 2009 12:00:00 UTC

18 May 2009 08:00:00
```

Figure 10-3

You might want to change your computer's time zone and time of year to see how it varies in different regions and with daylight savings changes. For example, although I'm in the United Kingdom, I have changed the settings on my computer for this example to Eastern Standard Time in the U.S. In Windows you can make the changes by opening the Control Panel and then double-clicking the Date/ Time icon.

So how does this example work? You declare a variable, `myDate`, and set it to a new `Date` object. Because you haven't initialized the `Date` object to any value, it contains the local current date and time.

Then, using the `setUTC` methods, you set the hours, minutes, and seconds so that the time is 12:00:00 UTC (midday, not midnight).

Now, when you write out the value of `myDate` as a UTC string, you get 12:00:00 and today's date. When you write out the value of the `Date` object as a local string, you get today's date and a time that is the UTC time 12:00:00 converted to the equivalent local time. The local values you'll see, of course, depend on your time zone. For example, New Yorkers will see 08:00:00 during the summer and 07:00:00 during the winter because of daylight savings. In the United Kingdom, in the winter you'll see 12:00:00, but in the summer you'll see 13:00:00.

For getting UTC dates and times, you have the same functions you would use for setting UTC dates and times, except that this time, for example, it's `getUTCHours()` , not `setUTCHours()`.

- `getUTCDate()`
- `getUTCDay()`

- ❑ `getUTCFullYear()`
- ❑ `getUTCHours()`
- ❑ `getUTCMilliseconds()`
- ❑ `getUTCMinutes()`
- ❑ `getUTCMonth()`
- ❑ `getUTCSeconds()`

Notice that this time there is an additional method, `getUTCDay()`. This works in the same way as the `getDay()` method and returns the day of the week as a number, from `0` for Sunday to `6` for Saturday. Because the day of the week is decided by the day of the month, the month, and the year, there is no `setUTCDay()` method.

Before moving on to look at timers, let's use your newly gained knowledge of the `Date` object and world time to create a world time converter. Later in this chapter, when you've learned how to use timers, you'll update the example to produce a world time clock.

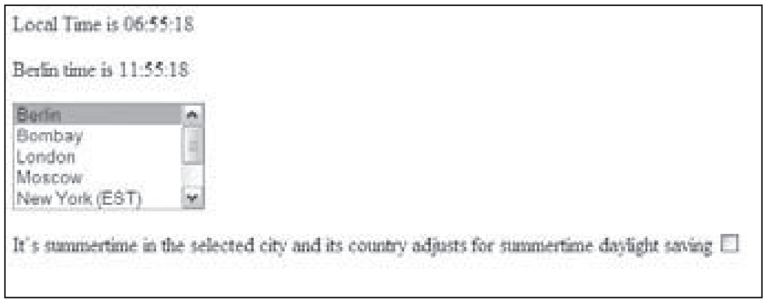
Try It Out World Time Converter (Part I)

The World Time Converter lets you calculate the time in different countries:

```
<!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">

<head>
<title>example 3</title>

<script type="text/javascript">
var timeDiff;
var selectedCity;
var daylightSavingAdjust = 0;
function updateTimeZone()
{
   var lstCity = document.form1.lstCity;
   timeDiff = lstCity.options[lstCity.selectedIndex].value;
   selectedCity = lstCity.options[lstCity.selectedIndex].text;
   updateTime();
}
function getTimeString(dateObject)
{
   var timeString;
   var hours = dateObject.getHours();
   if (hours < 10)
      hours = "0" + hours;
   var minutes = dateObject.getMinutes();
   if (minutes < 10)
      minutes = "0" + minutes;
   var seconds = dateObject.getSeconds()
   if (seconds < 10)
      seconds = "0" + seconds;
   timeString = hours + ":" + minutes + ":" + seconds;
   return timeString;
}
function updateTime()
{
   var nowTime = new Date();
   var resultsText = '<p>Local Time is ' + getTimeString(nowTime) + '</p>';

   nowTime.setMinutes(nowTime.getMinutes() + nowTime.getTimezoneOffset() +
      parseInt(timeDiff) + daylightSavingAdjust);

   resultsText += '<p>' + selectedCity + ' time is ' +
                    getTimeString(nowTime) + '</p>';

   document.getElementById('ConversionResultsDIV').innerHTML = resultsText;

}
function chkDaylightSaving_onclick()
{
   if (document.form1.chkDaylightSaving.checked)
   {
      daylightSavingAdjust = 60;
   }
   else
   {
      daylightSavingAdjust = 0;
   }
   updateTime();
}
</script>
</head>
<body onload="updateTimeZone()">

<div id="ConversionResultsDIV"></div>

<form name="form1">
<select size="5" name="lstCity" onchange="updateTimeZone();">
<option value="60" selected>Berlin
<option value="330">Bombay
<option value="0">London
<option value="180">Moscow
<option value="-300">New York (EST)
<option value="60">Paris
<option value="-480">San Francisco (PST)
<option value="600">Sydney
</select>
<p>
It's summertime in the selected city
and its country adjusts for summertime daylight saving
<input type="checkbox" name="chkDaylightSaving"
   onclick="return chkDaylightSaving_onclick()">
</p>
</form>


</body>
</html>
```

Save this page as `WorldTimeConverter.htm` and then load the page into your browser.

The form layout looks something like the one shown in Figure 10-4. Whenever the user clicks a city in the list, her local time and the equivalent time in the selected city are shown. In the example shown in Figure 10-4, the local region is set to Eastern Standard Time in the U.S. (for a city such as New York), and the selected city is Berlin, with the daylight savings box checked.



Local Time is 06:55:18

Berlin time is 11:55:18

It's summertime in the selected city and its country adjusts for summertime daylight saving

Figure 10-4

It's worth pointing out that this is just an example and not a totally foolproof one, because of the problems presented by daylight savings. Some countries don't have it, others do at fixed times of year, and yet others do but at varying times of the year. This makes it difficult to predict accurately when a country will have its daylight savings period. You have tried to solve this problem by adding a check box for the user to click if the city she chooses from the list is using daylight savings hours (which you assume will put the time in the city forward by one hour).

In addition, don't forget that some users may not even have their regional settings set correctly — there's no easy way around this problem.

In the body of the World Time Converter page is a form in which you've defined a list box using a `<select>` element.

```
<select size="5" name="lstCity" onchange="updateTimeZone();">
<option value="60" selected>Berlin
<option value="330">Bombay
<option value="0">London
<option value="180">Moscow
<option value="-300">New York (EST)
<option value="60">Paris
<option value="-480">San Francisco (PST)
<option value="600">Sydney
</select>
```

Each of the options displays the city's name in the list box and has its value set to the difference in minutes between that city's time zone (in winter) and UTC. So London, which uses UTC, has a value of `0`. Paris, which is an hour ahead of UTC, has a value of `60` (that is, `60` minutes). New York, which is five hours behind UTC, has a value of `-300`.

You'll see that you have captured the `change` event of the `<select>` element and connected it to the function `updateTimeZone()` defined in a script block in the head of the page. This function involves three global variables defined at the top of the script block.

```
var timeDiff;
var selectedCity;
var daylightSavingAdjust = 0;
```

The function `updateTimeZone()` updates two of these, setting the variable `timeDiff` to the value of the list's selected option (that is, the time difference between the selected city and UTC time) and the variable `selectedCity` to the text shown for the selected option (that is, the selected city).

```
function updateTimeZone()
{
   var lstCity = document.form1.lstCity;
   timeDiff = lstCity.options[lstCity.selectedIndex].value;
   selectedCity = lstCity.options[lstCity.selectedIndex].text;
```

In the final part of the function `updateTimeZone()`, the function `updateTime()` is called, as shown in the following:

```
   updateTime();
}
```

Before you go on to look at this function, you return to the final part of the form on the page. This is a check box, which the user clicks if the city she has chosen from the select list is in the summertime of a country that uses daylight savings hours.

```
<input type="checkbox" name="chkDaylightSaving"
   onclick="return chkDaylightSaving_onclick()">
```

As you can see, this check box's `click` event is connected to another function, `chkDaylightSaving_onclick()`.

```
function chkDaylightSaving_onclick()
{
   if (document.form1.chkDaylightSaving.checked)
   {
      daylightSavingAdjust = 60;
   }
   else
   {
      daylightSavingAdjust = 0;
   }
```

Inside the `if` statement, the code accesses the check box's `checked` property, which returns `true` if it is checked and `false` otherwise. If it has been checked, you set the global variable `daylightSavingAdjust` to `60` for summertime daylight savings; otherwise it's set to `0`.

```
   updateTime();
}
```

At the end of this function (as at the end of the function `updateTimeZone()` you saw earlier), the `updateTime()` function is called. You'll look at that next.

In the function `updateTime()`, you write the current local time and the equivalent time in the selected city to the results DIV with ID `ConversionResultsDIV`, which you defined in the frameset page.

You start at the top of the function by creating a new `Date` object, which is stored in the variable `nowTime`. The `Date` object will be initialized to the current local time.

```
function updateTime()
{
   var nowTime = new Date();
```

Next, to make your code more compact and easier to understand, you define a variable, `resultsText`, which will store the conversion results prior to them being written to the `/ConversionResultsDIV DIV` object contained in the page.

The first thing you store in variable `resultsText` is the local time based on the new `Date` object you just created. However, you want the time to be nicely formatted as *hours:minutes:seconds*, so you've written another function, `getTimeString()`, which does this for you. You'll look at that shortly.

```
var resultsText = '<p>Local Time is ' + getTimeString(nowTime) + '</p>';
```

Having stored the current time to your `resultsText` variable, you now need to calculate what the time would be in the selected city before also storing that to the `resultsText` variable.

You saw in Chapter 5 that if you set the value of a `Date` object's individual parts (such as hours, minutes, and seconds) to a value beyond their normal range, JavaScript assumes you want to adjust the date, hours, or minutes to take this into account. For example, if you set the hours to `36`, JavaScript simply changes the hours to `12` and adds one day to the date stored inside the `Date` object. You use this to your benefit in the following line:

```
    nowTime.setMinutes(nowTime.getMinutes() + nowTime.getTimezoneOffset() +
       parseInt(timeDiff) + daylightSavingAdjust);
```

Let's break this line down to see how it works. Suppose that you're in New York, with the local summer time of 5:11, and you want to know what time it is in Berlin. How does your line of code calculate this?

First, you get the minutes of the current local time; it's 5:11, so `nowTime.getMinutes()` returns `11`.

Then you get the difference, in minutes, between the user's local time and UTC using `nowTime .getTimezoneOffset()`. If you are in New York, which is different from UTC by 4 hours during the summer, this is 240 minutes.

Then you get the integer value of the time difference between the standard winter time in the selected city and UTC time, which is stored in the variable `timeDiff`. You've used `parseInt()`here because it's one of the few situations where JavaScript gets confused and assumes you want to join two strings together rather than treat the values as numbers and add them together. Remember that you got `timeDiff` from an HTML element's value, and that an HTML element's values are strings, even when they hold characters that are digits. Since you want the time in Berlin, which is 60 minutes different from UTC time, this value will be `60`.

Finally, you add the value of `daylightSavingsAdjust`. This variable is set in the function `chkdaylightsaving_onclick()`, which was discussed earlier. Since it's summer where you are and Berlin uses daylight savings hours, this value is `60`.

So you have the following:

```
11 + 240 + 60 + 60 = 371
```

Therefore `nowTime.setMinutes()` is setting the minutes to `371`. Clearly, there's no such thing as 371 minutes past the hour, so instead JavaScript assumes you mean 6 hours and 11 minutes after 5:00, that being 11:11 — the time in Berlin that you wanted.

Finally, the `updateTime()` function updates the `resultsText` variable and then writes the results to the `ConversionResultsDIV`.

```
    resultsText += '<p>' + selectedCity + ' time is ' +
                     getTimeString(nowTime) + '</p>';

    document.getElementById('ConversionResultsDIV').innerHTML = resultsText;
}
```

In the `updateTime()` function, you saw that it uses the function `getTimeString()` to format the time string. Let's look at that function now. This function is passed a `Date` object as a parameter and uses it to create a string with the format *hours*:*minutes*:*seconds*.

```
function getTimeString(dateObject)
{
   var timeString;
   var hours = dateObject.getHours();
   if (hours < 10)
      hours = "0" + hours;
   var minutes = dateObject.getMinutes();
   if (minutes < 10)
      minutes = "0" + minutes;
   var seconds = dateObject.getSeconds()
   if (seconds < 10)
      seconds = "0" + seconds;
   timeString = hours + ":" + minutes + ":" + seconds;
   return timeString;
}
```

Why do you need this function? Well, you can't just use this:

```
getHours() + ":" + getMinutes() + ":" + getSeconds()
```

That won't take care of those times when any of the three results of these functions is less than 10. For example, 1 minute past noon would look like 12:1:00 rather than 12:01:00.

The function therefore gets the values for hours, minutes, and seconds and checks each to see if it is below 10. If it is, a zero is added to the front of the string. When all the values have been retrieved, they are concatenated in the variable `timeString` before being returned to the calling function.

In the next section, you're going to look at how, by adding a timer, you can make the displayed time update every second like a clock.

Timers in a Web Page

You can create two types of timers: one-shot timers and continually firing timers. The *one-shot timer* triggers just once after a certain period of time, and the second type of timer continually triggers at set intervals. You will investigate each of these types of timers in the next two sections.

Within reasonable limits, you can have as many timers as you want and can set them going at any point in your code, such as at the window `onload` event or at the click of a button. Common uses for timers include advertisement banner pictures that change at regular intervals or display the changing time in a web page. Also all sorts of animations done with DHTML need `setTimeout()` or `setInterval()` — you'll be looking at DHTML later on in the book.

One-Shot Timer

Setting a one-shot timer is very easy: you just use the `window` object's `setTimeout()` method.

```
window.setTimeout("your JavaScript code", milliseconds_delay)
```

The method `setTimeout()` takes two parameters. The first is the JavaScript code you want executed, and the second is the delay, in milliseconds (thousandths of a second), until the code is executed.

The method returns a value (an integer), which is the timer's unique ID. If you decide later that you want to stop the timer firing, you use this ID to tell JavaScript which timer you are referring to.

For example, to set a timer that fires three seconds after the page has loaded, you could use the following code:

```
<!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">

<head>
<script type="text/javascript">
var timerID;
function window_onload()
{
   timerID = setTimeout("alert('Times Up!')",3000);
   alert('Timer Set');
}
</script>
</head>
<body onload="window_onload()">
</body>
</html>
```

Save this file as `timertest.htm`, and load it into your browser. In this page a message box appears 3,000 milliseconds (that is, 3 seconds) after the `onload` event of the window has fired.

The `setTimeout()` method can also take a direct reference to a function instead of a JavaScript string. For example if you have a function called `myFunction` then you call `setTimeout()` like this:

```
window.setTimeout(myFunction, milliseconds_delay)
```

Although `setTimeout()` is a method of the `window` object, you'll remember that because the `window` object is at the top of the hierarchy, you don't need to use its name when referring to its properties and methods. Hence, you can use `setTimeout()` instead of `window.setTimeout()`.

It's important to note that setting a timer does not stop the script from continuing to execute. The timer runs in the background and fires when its time is up. In the meantime the page runs as usual, and any script after you start the timer's countdown will run immediately. So, in this example, the alert box telling you that the timer has been set appears immediately after the code setting the timer has been executed.

What if you decided that you wanted to stop the timer before it fired?

To clear a timer you use the `window` object's `clearTimeout()` method. This takes just one parameter, the unique timer ID that the `setTimeout()` method returns.

Let's alter the preceding example and provide a button that you can click to stop the timer.

```
<!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">

<head>
<script type="text/javascript">
var timerID;
function window_onload()
{
   timerID = setTimeout("alert('Times Up!')",3000);
   alert('Timer Set');
}

function butStopTimer_onclick()
{
   clearTimeout(timerID);
   alert("Timer has been cleared");
}

</script>
</head>
<body onload="window_onload()">

<form name="form1">
<input type="button" value="Stop Timer" name="butStopTimer"
   onclick="return butStopTimer_onclick()" />
</form>


</body>
</html>
```

Save this as `timertest2.htm` and load it into your browser. Now if you click the Stop Timer button before the three seconds are up, the timer will be cleared. This is because the button is connected to the `butStopTimer_onclick()` function, which uses the timer's ID `timerID` with the `clearTimeout()` method of the `window` object.

Try It Out Updating a Banner Advertisement

You'll now look at a bigger example using the `setTimeout()` method. The following example creates a web page with an image banner advertisement that changes every few seconds.

```
<!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">

<head>
<script language=JavaScript type="text/javascript">
var currentImgNumber = 1;
var numberOfImages = 3;
```

```
function window_onload()
{
   setTimeout("switchImage()",3000);
}
function switchImage()
{
   currentImgNumber++;
   document.imgAdvert.src = 'AdvertImage' + currentImgNumber + '.jpg';
   if (currentImgNumber < numberOfImages)
   {
      setTimeout("switchImage()",3000);   }
   }
</script>
</head>
<body onload="window_onload()">
<img src="AdvertImage1.jpg" name="imgAdvert" />



</body>
</html>
```

After you've typed in the code, save the page as `adverts.htm`. You'll also need to create three images named `AdvertImage1.jpg`, `AdvertImage2.jpg`, and `AdvertImage3.jpg` (alternatively, the three images are supplied with the downloadable code for the book).

When the page is loaded, you start with a view of `AdvertImage1.jpg`, as shown in Figure 10-5.

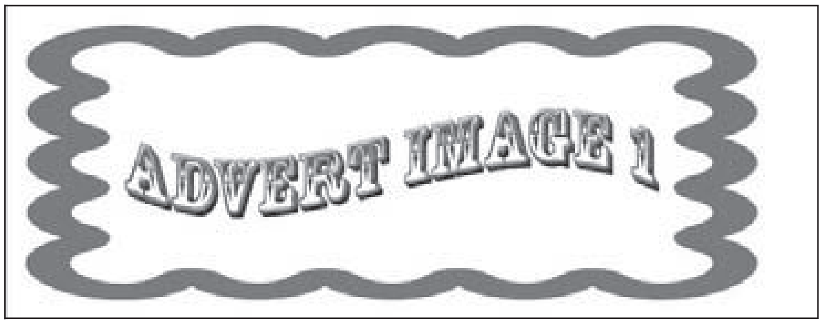


Figure 10-5

In three seconds, this changes to the second image, shown in Figure 10-6.

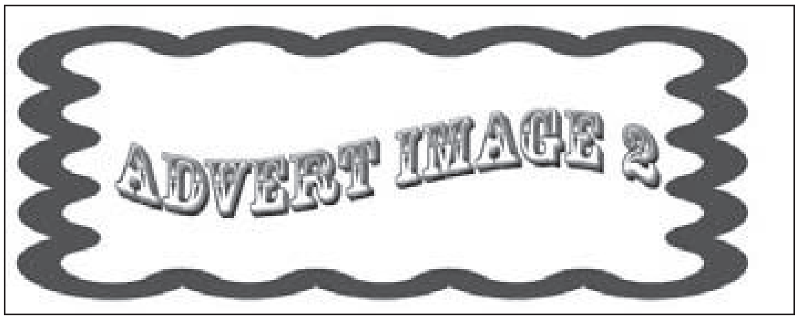


Figure 10-6

Finally, three seconds later, a third and final image loads, shown in Figure 10-7.

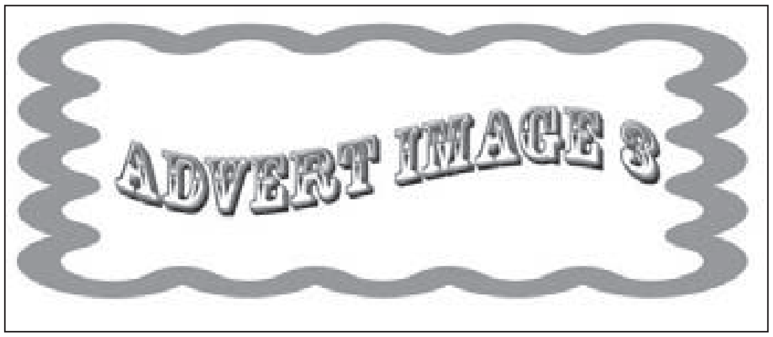


Figure 10-7

When the page loads, the `<img>` tag has its `src` attribute set to the first image.

```
<img src="AdvertImage1.jpg" name="imgAdvert" />
```

Within the `<body>` tag, you connect the `window` object's `onload` event handler to the function `window_onload()`.

```
function window_onload()
{
   setTimeout("switchImage()",3000)
}
```

In this function, you use the `setTimeout()` method to start a timer running that will call the function `switchImage()` in three seconds. Since you don't have to clear the timer, you haven't bothered to save the timer ID returned by the `setTimeout()` method.

The `switchImage()` function changes the value of the `src` property of the `img` object corresponding to the `<img>` tag in your page.

```
function switchImage()
{
   currentImgNumber++;
   document.imgAdvert.src = 'AdvertImage' + currentImgNumber + '.jpg';
```

Your advertisement images are numbered from one to three: `AdvertImage1.jpg`, `AdvertImage2.jpg`, and `AdvertImage3.jpg`. You keep track of the number of the advertisement image currently loaded in the page in the global variable `currentImgNumber`, which you defined at the top of the script block and initialized to `1`. To get the next image you simply increment that variable by one, and then update the image loaded by setting the `src` property of the `img` object, using the variable `currentImgNumber` to build up its full name.

```
   if (currentImgNumber < numberOfImages)
   {
      setTimeout('switchImage()',3000);
   }
}
```

You have three advertisement images you want to show. In the `if` statement you check to see whether `currentImgNumber`, which is the number of the current image, is less than three. If it is, it means there are more images to show, and so you set another timer going, identical to the one you set in the `window` object's `onload` event handler. This timer will call this function again in three seconds.

In earlier browsers, this was the only method of creating a timer that fired continually at regular intervals. However, in most current browsers such as IE6+ and Firefox, you'll see next that there's an easier way.

Setting a Timer that Fires at Regular Intervals

Modern browsers saw new methods added to the `window` object for setting timers, namely the `setInterval()` and `clearInterval()` methods. These work in a very similar way to `setTimeout()` and `clearTimeout()`, except that the timer fires continually at regular intervals rather than just once.

The method `setInterval()` takes the same parameters as `setTimeout()`, except that the second parameter now specifies the interval, in milliseconds, between each firing of the timer, rather than just the length of time before the timer fires.

For example, to set a timer that fires the function `myFunction()` every five seconds, the code would be as follows:

```
var myTimerID = setInterval("myFunction()",5000);
```

As with `setTimeout()`, the `setInterval()` method returns a unique timer ID that you'll need if you want to clear the timer with `clearInterval()`, which works identically to `clearTimeout()`. So to stop the timer started in the preceding code, you would use the following:

```
clearInterval(myTimerID);
```

Try It Out World Time Converter (Part 2)

Let's change the world time example that you saw earlier, so that it displays a local time and selected city time as a continually updating clock.

You'll be making changes to the `WorldTimeConverter.htm` file, so open that in your text editor. Add the following function before the functions that are already defined:

```
var daylightSavingAdjust = 0;
function window_onload()
{
   updateTimeZone();
   window.setInterval("updateTime()",1000);
}
function updateTimeZone()
{
```

Next edit the `<body>` tag so it looks like this:

```
<body onload="return window_onload()">
```

Resave the file, and then load `WorldTimeConverter.htm` into your browser. The page should look the same as the previous version of the time converter, except that the time is updated every second.

The changes you made were short and simple. In the function `window_onload()`, you have added a timer that will call the `updateTime()` function every 1,000 milliseconds — that is, every second. It'll keep doing this until you leave the page. Previously your `updateTime()` function was called only when the user clicked either a different city in the list box or the summertime check box.

The `window_onload()` function is connected to the `window` object's `onload` event in the `<body>` tag, so after the page has loaded your clock starts running.

That completes your look at this example and also your introduction to timers.

Summary

You started the chapter by looking at Coordinated Universal Time (UTC), which is an international standard time. You then looked at how to create timers in web pages.

The particular points covered were the following:

- The `Date` object enables you to set and get UTC time in a way similar to setting a `Date` object's local time by using methods (such as `setUTCHours()` and `getUTCHours()`) for setting and getting UTC hours with similar methods for months, years, minutes, seconds, and so on.
- A useful tool in international time conversion is the `getTimezoneOffset()` method, which returns the difference, in minutes, between the user's local time and UTC. One pitfall of this is that you are assuming the user has correctly set his time zone on his computer. If not, `getTimezoneOffset()` is rendered useless, as will be any local date and time methods if the user's clock is incorrectly set.
- Using the `setTimeout()` method, you found you could start a timer going that would fire just once after a certain number of milliseconds. `setTimeout()` takes two parameters: the first is the code you want executed, and the second is the delay before that code is executed. It returns a value, the unique timer ID that you can use if you later want to reference the timer; for example, to stop it before it fires, you use the `clearTimeout()` method.
- To create a timer that fires at regular intervals, you used the `setInterval()` method, which works in the same way as `setTimeout()`, except that it keeps firing unless the user leaves the page or you call the `clearInterval()` method.

In the next chapter, you'll be looking at a way of storing information on the user's computer using something called a cookie. Although they may not be powerful enough to hold a user's life history, they are certainly enough for us to keep track of a user's visits to the website and what pages they view when they visit. With that information, you can provide a more customized experience for the user.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

1. Create a web page with an advertisement image at the top. When the page loads, select a random image for that advertisement. Every four seconds, make the image change to a different one and ensure a different advertisement is selected until all the advertisement images have been seen.
2. Create a form that gets the user's date of birth. Then, using that information, tell them on what day of the week they were born.