8

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.

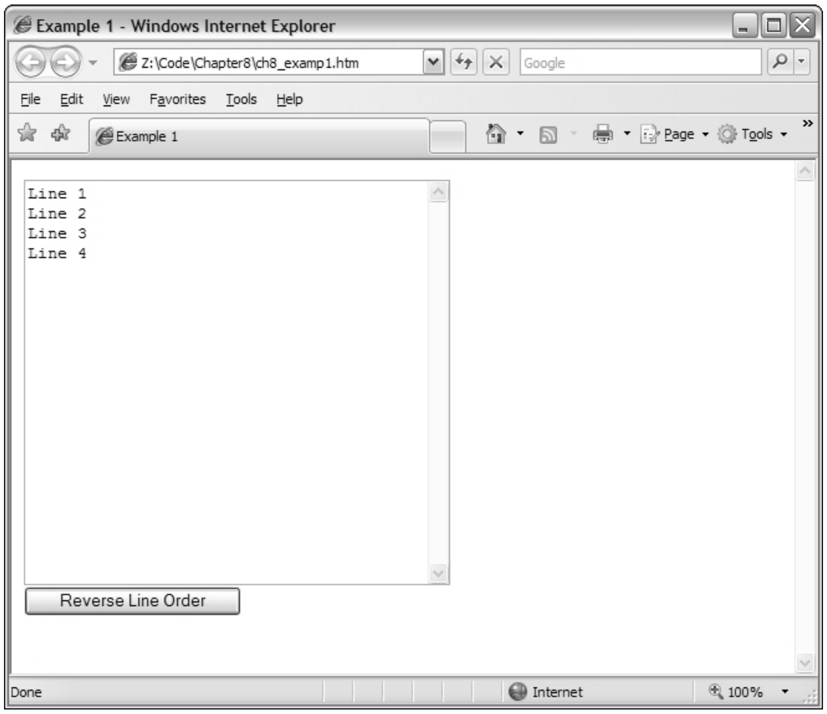
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Example 1</title>
<script language="JavaScript" type="text/JavaScript">
function splitAndReverseText(textAreaControl)
{

   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>
```

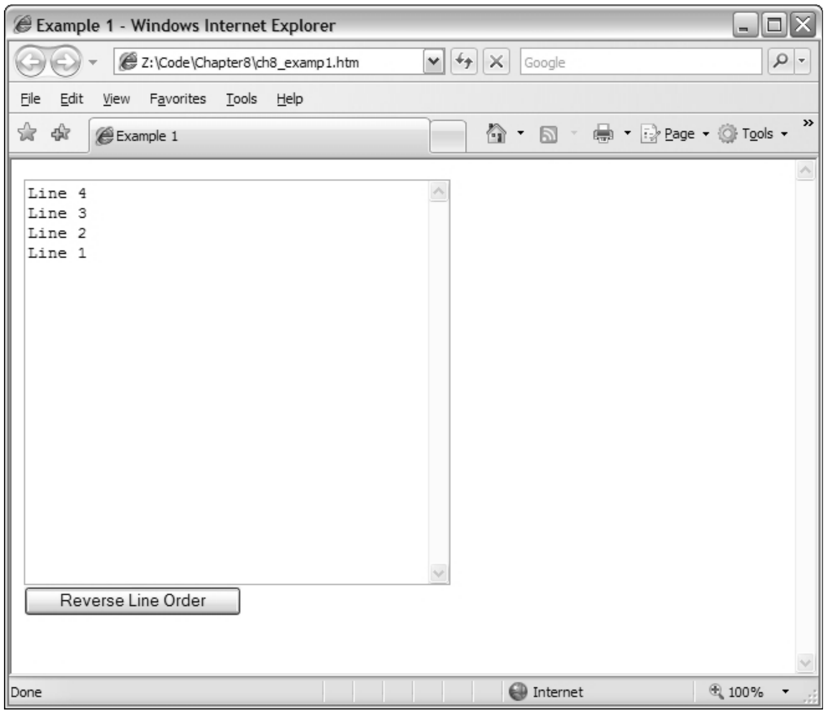
Save this as `ch8_examp1.htm` and load it into your browser. You should see the screen shown in Figure 8-1.


Example 1 - Windows Internet Explorer
Z:\Code\Chapter8\ch8_examp1.htm
Google
File Edit View Favorites Tools Help
Example 1
Page Tools
Reverse Line Order
Done
Internet
100%

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

Figure 8-1

Clicking the Reverse Line Order button reverses the order of the lines, as shown in Figure 8-2.


Example 1 - Windows Internet Explorer
Z:\Code\Chapter8\ch8_examp1.htm
File Edit View Favorites Tools Help
Example 1
Reverse Line Order
Done
Internet
100%

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

Figure 8-2

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

Although this example works on Internet Explorer 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 Internet Explorer.*

How It Works

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 Internet Explorer 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";
      }
   }
```

When you split the string, all your line formatting is removed. So in the `if` statement you add a linefeed (`\n`) onto the end of each string, except for the last string; that is, when the `indexCount` variable is `0`.

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 we 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 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 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.

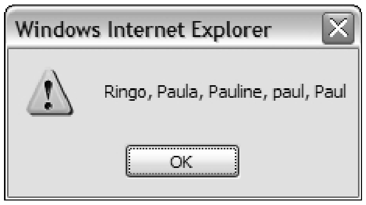
```
myString = myString.replace(myRegExp, "Ringo");
```

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 HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<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 8-3.



```
Ringo, Paula, Pauline, paul, Paul
```

Figure 8-3

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?

Well, 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.

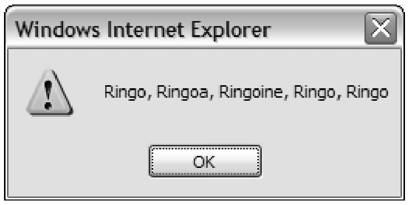
What you want is a global match, which is a search for all possible matches to be made and replaced. To help you out, the `RegExp` object has three attributes you can define. You can see these listed in the following table.

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 our `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 8-4.



Ringo, Ringoa, Ringoine, Ringo, Ringo

Figure 8-4

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.

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, including tab, newline, carriage return, formfeed, and vertical tab	`\s` matches *tab*
`\S`	Any non-whitespace character	`\S` matches `A`, but not the tab character
`.`	Any single character other than the newline character (`\n`)	`.` matches a or 4 or @
`[...]`	Any one of the characters between the brackets	`[abc]` will match a or b or c, but nothing else `[a-z]` will match any character in the range a to z
`[^...]`	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, and not punctuation or symbols like `@`, `%`, and so on.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
<script language="JavaScript" 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>
<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 `ch8_examp2.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.

How It Works

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 our 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 let's 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.

First 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. Next 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 our regular expression actually matches any character not between A and Z or a and z.

Next you add `\d` to indicate any digit character, or any character between 0 and 9.

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

So your expression matches any character that is not between a and z, A and Z, or 0 and 9. Finally, 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.

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 the user know that his 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.

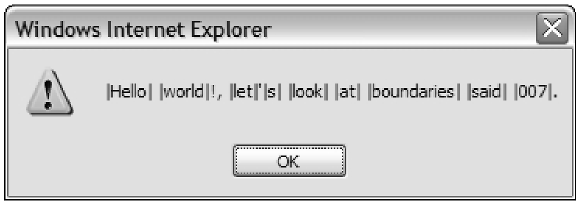
Imagine you had the string `"Hello world!, let's look at boundaries said 007."` defined in the code as follows:

```
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 8-5.



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

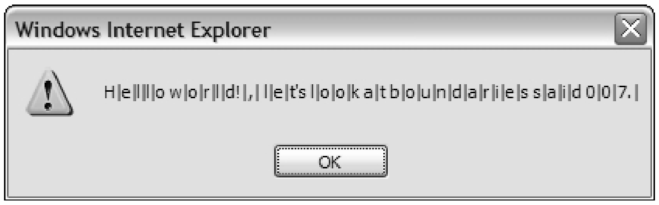
Figure 8-5

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 8-6.



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 8-6

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 our 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 we started looking at regular expressions:

```
<html>
<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>
```

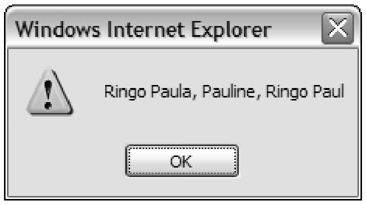
We used this code to convert all instances of `Paul` or `paul` to `Ringo`.

However, we 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 our regular expression to the following:

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

This gives the result shown in Figure 8-7.



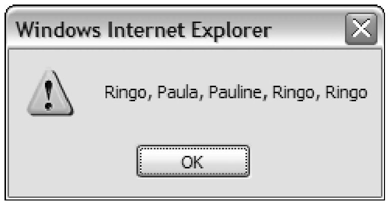
Ringo Paula, Pauline, Ringo Paul

Figure 8-7

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 8-8.



Ringo, Paula, Pauline, Ringo, Ringo

Figure 8-8

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 we 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.

The question is, 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;
```

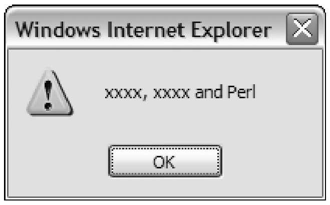
Breaking this expression down, you can see the pattern it requires is as follows:

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 8-9.



```
xxxx, xxxx and Perl
```

Figure 8-9

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 our 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 our 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 we have done here. 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 HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<body>
<script language="JavaScript" 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 `ch8_examp3.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.

How It Works

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.

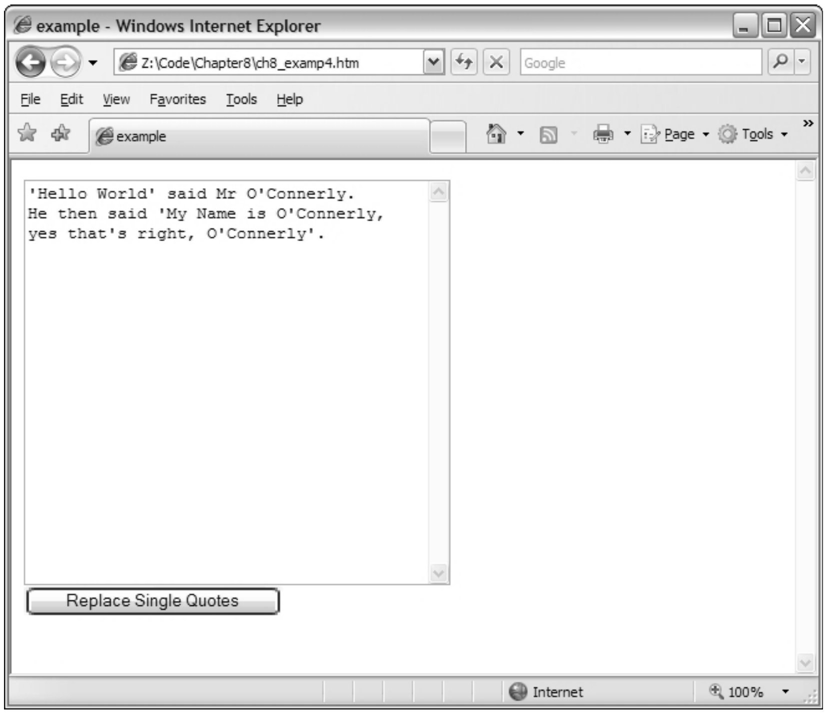
Try It Out Replacing Single Quotes with Double Quotes

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

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<script language="JavaScript" 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 `ch8_examp4.htm`. Load the page into your browser and you should see what is shown in Figure 8-10.

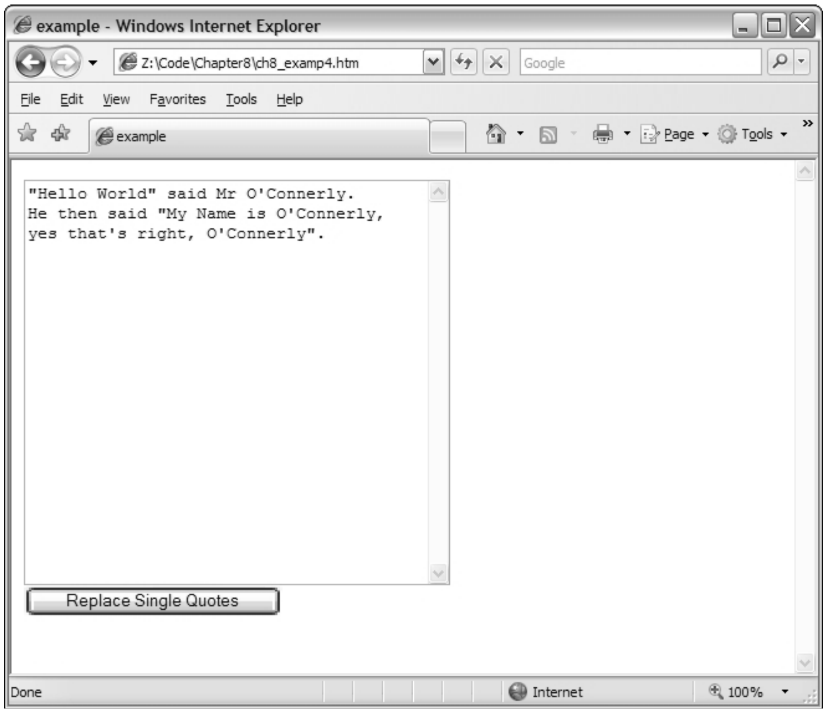


```
'Hello World' said Mr O'Connerly.
He then said 'My Name is O'Connerly,
yes that's right, O'Connerly'.
```

example - Windows Internet Explorer
Z:\Code\Chapter8\ch8_examp4.htm
File Edit View Favorites Tools Help
Replace Single Quotes
Internet
100%

Figure 8-10

Click the Replace Single Quotes button to see the single quotes in the text area replaced as in Figure 8-11.


example - Windows Internet Explorer
Z:\Code\Chapter8\ch8_examp4.htm
Google
File Edit View Favorites Tools Help
example
Page Tools
Replace Single Quotes
Done
Internet
100%

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

Figure 8-11

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

How It Works

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

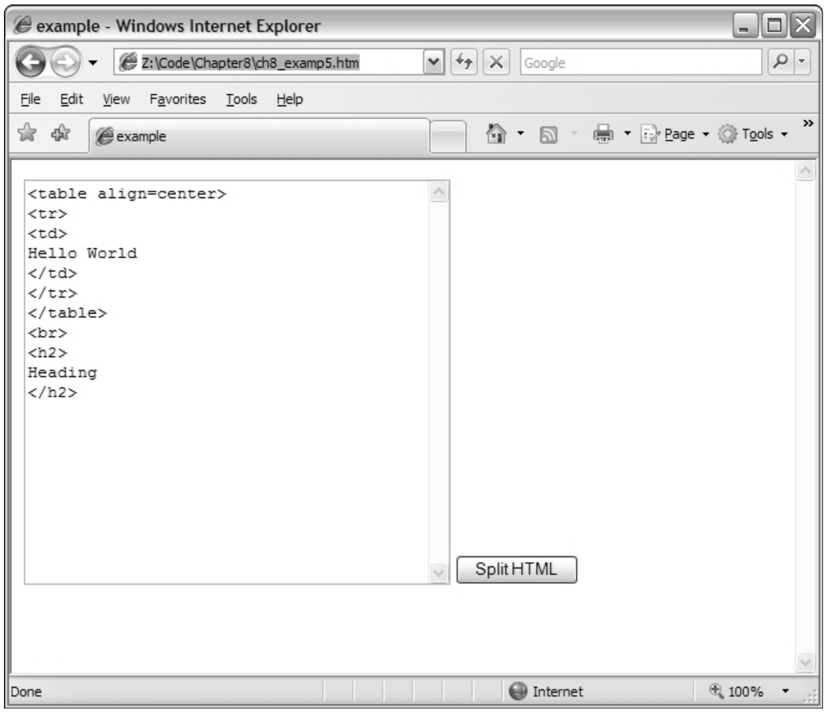
In the next example, you want to take a string of HTML and split it into its component parts. For example, you want the HTML `<P>Hello</P>` to become an array, with the elements having the following contents:

<P>	Hello	</P>

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<script language="JavaScript" 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 `ch8_examp5.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 8-12.


example - Windows Internet Explorer
Z:\Code\Chapter8\ch8_examp5.htm
Google
File Edit View Favorites Tools Help
example
Page Tools
Split HTML
Done
Internet
100%

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

Figure 8-12

How It Works

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");
```

The Trivia Quiz

The goal for the trivia quiz in this chapter is to enable it to set questions with answers that have to be typed in by the user, in addition to the multiple-choice questions you already have. To do this you'll be making use of your newfound knowledge of regular expressions to search the reply that the user types in for a match with the correct answer.

The problem you face with text answers is that a number of possible answers may be correct and you don't want to annoy the user by insisting on only *one* specific version. For example, the answer to the question "Which president was involved in the Watergate scandal?" is Richard Milhous Nixon. However, most people will type Nixon, or maybe Richard Nixon or even R Nixon. Each of these variations is valid, and using regular expressions you can easily check for all of them (or at least many plausible alternatives) in just a few lines of code.

What will you need to change to add this extra functionality? In fact changes are needed in only two pages: the `GlobalFunctions.htm` page and the `AskQuestion.htm` page.

In the `GlobalFunctions.htm` page, you need to define your new questions and answers, change the `getQuestion()` function, which builds up the HTML to display the question to the user, and change the `answerCorrect()` function, which checks whether the user's answer is correct.

In the `AskQuestion.htm` page, you need to change the function `getAnswer()`, which retrieves the user's answer from the page's form.

You'll start by making the changes to `GlobalFunctions.htm` that you created in the last chapter, so open this up in your HTML editor.

All the existing multiple-choice questions that you define near the top of the page can remain in exactly the same format, so there's no need for any changes there. How can this be if you're using regular expressions?

Previously you checked to see that the answer the user selected, such as A, B, C, and so on, was equal to the character in the `answers` array. Well, you can do the same thing here, but using a very simple regular expression that matches the character supplied by the user with the character in the `answers` array. If they match, you know the answer is correct.

Now you'll add the first new text-based question and answer directly underneath the last multiple-choice question in the `GlobalFunctions.htm` file.

```
// define question 4
questions[3] = "In the Simpsons, Bleeding Gums Murphy played which instrument?";
// assign answer for question 4
answers[3] = "\\bsax(ophone)?\\b";
```

The question definition is much simpler for text-based questions than for the multiple-choice questions: it's just the question text itself.

The answer definition is a regular expression. Note that you use `\\b` rather than `\b`, since you'll be creating your regular expressions using `new RegExp()` rather than using the `/` and `/` method. The valid answers to this question are `sax` and `saxophone`, so you need to define your regular expression to match either of those. You'll see later that the case flag will be set so that even `SaxoPhone` is valid, though dubious, English! Let's break it down stage by stage as shown in the following table.

Expression	Description
`\\b`	The `\\b` indicates that the answer must start with a word boundary; in other words, it must be a whole word and not contained inside another word. You do this just in case the user for some reason puts characters before his answer, such as `My answer is saxophone`.
`Sax`	The user's answer must start with the characters `sax`.
`(ophone)?`	You've grouped the pattern `ophone` by putting it in parentheses. By putting the `?` just after it, you are saying that that pattern can appear zero or one time. If the user types `sax`, it appears zero times, and if the user types `saxophone`, it appears once — either way you make a match.
`\\b`	Finally you want the word to end at a word boundary.

The second question you'll create is

```
"Which American president was involved in the Watergate scandal?"
```

The possible correct answers for this are quite numerous and include the following:

```
Richard Milhous Nixon
Richard Nixon
Richard M. Nixon
Richard M Nixon
R Milhous Nixon
R. Milhous Nixon
R. M. Nixon
R M Nixon
R.M. Nixon
RM Nixon
R Nixon
R. Nixon
Nixon
```

This is a fairly exhaustive list of possible correct answers. You could perhaps accept only `Nixon` and `Richard Nixon`, but the longer list makes for a more challenging regular expression.

```
// define question 5
questions[4] = "Which American president was involved in the Watergate scandal?";
// assign answer for question 5
answers[4] = "\\b((Richard |R\\.? ?)(Milhous |M\\.? )?)?Nixon\\b";
```

Add the question-and-answer code under the other questions and answers in the `GlobalFunctions.htm` file.

Let's analyze this regular expression now.

Expression	Description
`\\b`	This indicates that the answer must start with a word boundary, so the answer must be a whole word and not contained inside another word. You do this just in case the user for some reason puts characters before his answer, such as `My answer is President Nixon`.
`((Richard |R\\.? ?)`	This part of the expression is grouped together with the next part, `(Milhous |M\\.? )?)`. The first parenthesis creates the outer group. Inside this is an inner group, which can be one of two patterns. Before the | is the pattern `Richard`, and after it is the pattern `R` followed by an optional dot (`.`) followed by an optional space. So either `Richard` or `R` will match. Since the `.` is a special character in regular expressions, you have to tell JavaScript you mean a literal dot and not a special-character dot. You do this by placing the `\` in front. However, because you are defining this regular expression using the `RegExp()` constructor, you need to place an additional `\` in front.
`(Milhous |M\\.? )?)?`	This is the second subgroup within the outer group. It works in a similar way to the first subgroup, but it's `Milhous` rather than `Richard` and `M` rather than `R` that you are matching. Also, the space after the initial is not optional, since you don't want `RMNixon`. The second `?` outside this inner group indicates that the middle name/initial is optional. The final parenthesis indicates the end of the outer group. The final `?` indicates that the outer group pattern is optional — this is to allow the answer `Nixon` alone to be valid.
`Nixon\\b`	Finally the pattern `Nixon` must be matched, and followed by a word boundary.

That completes the two additional text-based questions. Now you need to alter the question creation function, `getQuestion()`, again inside the file `GlobalFunctions.htm`, as follows:

```
function getQuestion()
{
   if (questions.length != numberOfQuestionsAsked)
   {
      var questionNumber = Math.floor(Math.random() * questions.length);
      while (questionsAsked[questionNumber] == true)
      {
         questionNumber = Math.floor(Math.random() * questions.length);
      }
      var questionLength = questions[questionNumber].length;
      var questionChoice;
      numberOfQuestionsAsked++;
      var questionHTML = "<h4>Question " + numberOfQuestionsAsked +  "</h4>";
      // Check if array or string
      if (typeof questions[questionNumber] == "string")
      {
         questionHTML = questionHTML + "<p>" + questions[questionNumber] + "</p>";
```

```
            questionHTML = questionHTML + "<p><input type=text name=txtAnswer ";
            questionHTML = questionHTML + " maxlength=100 size=35></p>";
            questionHTML = questionHTML + '<script type="text/javascript">';
                           + 'document.QuestionForm.txtAnswer.value = "";<\/script>';
         }
         else
         {
         questionHTML = questionHTML + "<p>" + questions[questionNumber][0];
         questionHTML = questionHTML + "</p>";
         for (questionChoice = 1;questionChoice < questionLength;questionChoice++)
         {
            questionHTML = questionHTML + "<input type=radio ";
            questionHTML = questionHTML + "name=radQuestionChoice";
            if (questionChoice == 1)
            {
               questionHTML = questionHTML + " checked";
            }
            questionHTML = questionHTML + ">" +
               questions[questionNumber][questionChoice];
            questionHTML = questionHTML + "<br>"
         }
         }
         questionHTML = questionHTML + "<br><input type='button' "
         questionHTML = questionHTML + "value='Answer Question'";
         questionHTML = questionHTML + "name=buttonNextQ ";
         questionHTML = questionHTML + "onclick='return buttonCheckQ_onclick()'>";
         currentQNumber = questionNumber;
         questionsAsked[questionNumber] = true;
      }
      else
      {
         questionHTML = "<h3>Quiz Complete</h3>";
         questionHTML = questionHTML + "You got " + numberOfQuestionsCorrect;
         questionHTML = questionHTML + " questions correct out of ";
         questionHTML = questionHTML + numberOfQuestionsAsked;
         questionHTML = questionHTML + "<br><br>Your trivia rating is ";
         switch(Math.round(((numberOfQuestionsCorrect / numberOfQuestionsAsked) * 10)))
         {
            case 0:
            case 1:
            case 2:
            case 3:
               questionHTML = questionHTML + "Beyond embarrassing";
               break;
            case 4:
            case 5:
            case 6:
            case 7:
               questionHTML = questionHTML + "Average";
               break;
            default:
               questionHTML = questionHTML + "Excellent"
         }
         questionHTML = questionHTML + "<br><br><A href='quizpage.htm'><strong>";
```

```
            questionHTML = questionHTML + "Start again</strong></A>";
         }
         return questionHTML;
      }
```

You can see that the `getQuestion()` function is mostly unchanged by your need to ask text-based questions. The only code lines that have changed are the following:

```
            if (typeof questions[questionNumber] == "string")
            {
               questionHTML = questionHTML + "<p>" + questions[questionNumber] + "</p>";
               questionHTML = questionHTML + "<p><input type=text name=txtAnswer ";
               questionHTML = questionHTML + " maxlength=100 size=35></P>";
               // Next line necessary due to bugs in Netscape 7.x
               questionHTML = questionHTML + '<script type="text/javascript">'
                                + 'document.QuestionForm.txtAnswer.value = "";<\/script>';
            }
            else
            {
```

The reason for this change is that the questions for multiple-choice and text-based questions are displayed differently. Having obtained your question number, you then need to check to see if this is a text question or a multiple-choice question. For text-based questions, you store the string containing the text inside the `questions[]` array; for multiple-choice questions, you store an array inside the `questions[]` array, which contains the question and options. You can check to see whether the type of data stored in the `questions[]` array at the index for that particular question is a string type. If it's a string type, you know you have a text-based question; otherwise you can assume it's a multiple-choice question. Note that Netscape 7.*x* has a habit of keeping previously entered data in text fields. This means that when the second text-based question is asked, the answer given for the previous text question is automatically pre-entered.

You use the `typeof` operator as part of the condition in your `if` statement in the following line:

```
      if (typeof questions[questionNumber] == "string")
```

If the condition is `true`, you then create the HTML for the text-based question; otherwise the HTML for a multiple-choice question is created.

The second function inside `GlobalFunctions.htm` that needs to be changed is the `answerCorrect()` function, which actually checks the answer given by the user.

```
      function answerCorrect(questionNumber, answer)
      {
         // declare a variable to hold return value
         var correct = false;
         // if answer provided is same as answer then correct answer is true
         var answerRegExp = new RegExp(answers[questionNumber],"i");
         if (answer.search(answerRegExp) != -1)
         {
            numberOfQuestionsCorrect++;
            correct = true;
         }
      // return whether the answer was correct (true or false)
      return correct;
   }
```

Instead of doing a simple comparison of the user's answer to the value in the `answers[]` array, you're now using regular expressions.

First you create a new `RegExp` object called `answerRegExp` and initialize it to the regular expression stored as a string inside your `answers[]` array. You want to do a case-insensitive match, so you pass the string `i` as the second parameter.

In your `if` statement, you search for the regular-expression answer pattern in the answer given by the user. This answer will be a string for a text-based question or a single character for a multiple-choice question. If a match is found, you'll get the character match position. If no match is found, `-1` is returned. Therefore, if the match value is not `-1`, you know that the user's answer is correct, and the `if` statement's code executes. This increments the value of the variable `numberOfQuestionsCorrect`, and sets the `correct` variable to the value `true`.

That completes the changes to `GlobalFunctions.htm`. Remember to save the file before you close it.

Finally, you have just one more function you need to alter before your changes are complete. This time the function is in the file `AskQuestion.htm`. The function is `getAnswer()`, which is used to retrieve the user's answer from the form on the page. The changes are shown in the following code:

```
   function getAnswer()
   {
      var answer = 0;
      if (document.QuestionForm.elements[0].type == "radio")
      {
         while (document.QuestionForm.radQuestionChoice[answer].checked != true)
            answer++;
         answer =  String.fromCharCode(65 + answer);
      }
      else
      {
         answer = document.QuestionForm.txtAnswer.value;
      }
      return answer;
   }
```

The user's answer can now be given via one of two means: an option being chosen in an option group, or text being entered in a text box. You determine which way was used for this question by using the `type` property of the first control in the form. If the first control is a radio button, you know this is a multiple-choice question; otherwise you assume it's a text-based question.

If it is a multiple-choice question, you obtain the answer, a character, as you did before you added text questions. If it's a text-based question, it's simply a matter of getting the text value from the text control written into the form dynamically by the `getQuestion()` function in the `GlobalFunctions.htm` page.

Save the changes to the page. You're now ready to give your updated trivia quiz a test run. Load `TriviaQuiz.htm` to start the quiz. You should now see the text questions you've created (see Figure 8-13).

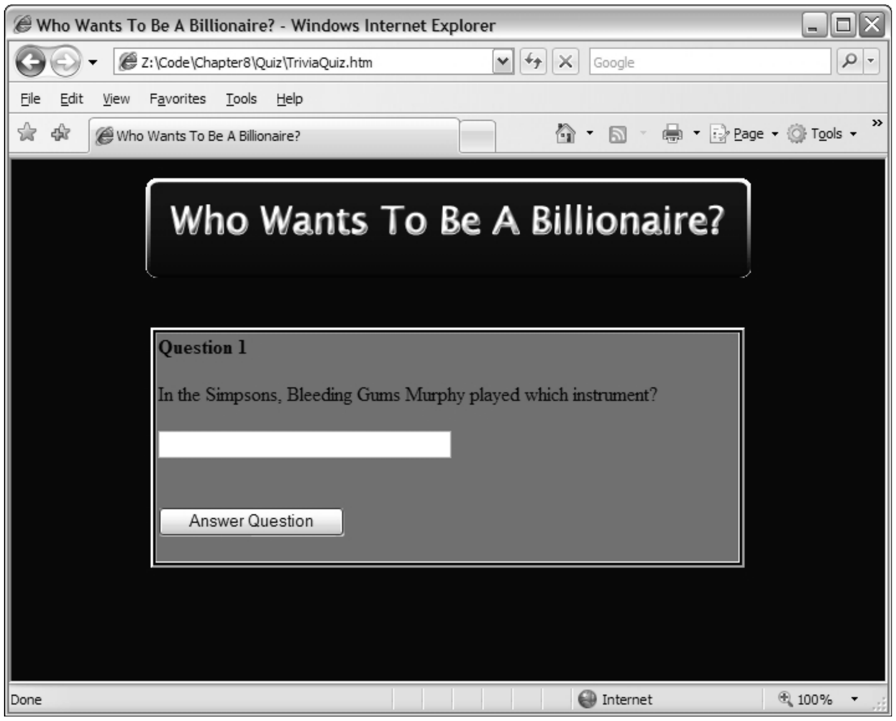

Who Wants To Be A Billionaire? - Windows Internet Explorer
Z:\Code\Chapter8\Quiz\TriviaQuiz.htm
Who Wants To Be A Billionaire?
Question 1
In the Simpsons, Bleeding Gums Murphy played which instrument?
Answer Question
Done
Internet
100%

Figure 8-13

Although you've learned a bit more about regular expressions while altering the trivia quiz, perhaps the most important lesson has been that using general functions, and where possible placing them inside common code modules, makes later changes quite simple. In less than 20 lines, mostly in one file, you have made a significant addition to the quiz.

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.
- You also updated the trivia quiz in this chapter to allow questions to be set that require a text-based response from the user, in addition to the multiple-choice questions you have already seen.

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. You'll be adapting the trivia quiz so that the user can select a time within which it must be completed — enabling him to specify, for example, that five questions must be answered in one minute.

Exercise Questions

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

Question 1

What problem does the code below 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?

Question 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”

Question 3

Imagine you have a web site with a message board. Write a regular expression that would remove barred words. (I’ll let you make up your own words!)