Beginning JavaScript®, Fifth Edition
By Jeremy McPeak and Paul Wilton

6

String Manipulation

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Using the String object's advanced methods to manipulate strings
- Matching substrings follow a specific pattern
- Validating useful pieces of information, such as telephone numbers, e-mail addresses, and postal codes

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `http://www.wiley.com/go/BeginningJavaScript5E` on the Download Code tab. You can also view all of the examples and related files at `http://beginningjs.com`.

In Chapter 5 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 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 you can use 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 plaintext. You 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 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>

<html lang="en">
<head>
    <title>Chapter 6, Example 1</title>
</head>
    <body>
        <script>
            var values = prompt("Please enter a set of comma separated values.",
                "Apples,Oranges,Bananas");

            function splitAndReverseText(csv) {
                var parts = csv.split(",");
                parts.reverse();

                var reversedString = parts.join(",");

                alert(reversedString);
            }

            splitAndReverseText(values);
        </script>
    </body>
</html>
```

Save this as `ch6_example1.html` and load it into your browser. Use the default value in the prompt box, click OK, and you should see the screen shown in Figure 6-1.

Try other comma-separated values to test it further.

The key to how this code works is the function `splitAndReverseText()`. It accepts a string value that should contain one or more commas. You start by splitting the value contained within `csv` using the `split()` method and putting the resulting array inside the `parts` variable:

```
function splitAndReverseText(csv) {
    var parts = csv.split(",");
```

This uses a comma as the separator. You then reverse the array of string parts using the `Array` object's `reverse()` method:

```
    parts.reverse();
```

With the array now reversed, it's just a simple matter of creating the new string. You can easily accomplish this with the `Array` object's `join()` method:

```
    var reversedString = parts.join(",");
```

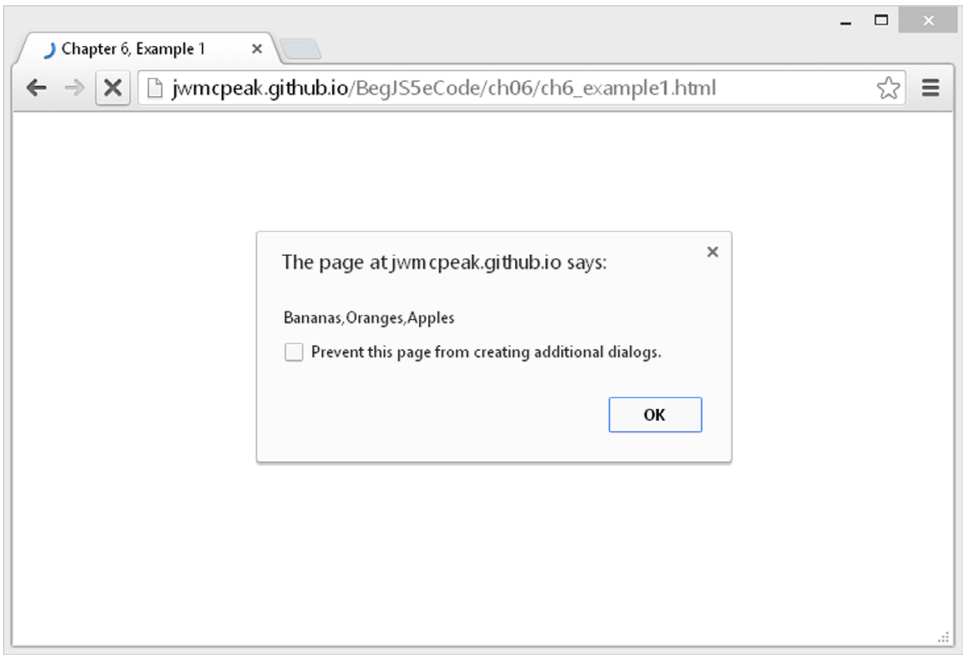

FIGURE 6-1

Remember from Chapter 5 that the `join()` method converts an array into a string, separating each element with the specified separator.

Finally, you display the new string in an `alert` box:

```
    alert(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:

```
Var 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 plaintext, the `search()` method provides no real benefit over methods like `indexOf()`, which you've already seen. However, you see later that the power of this method becomes apparent when you use regular expressions.

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 plaintext 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. Most modern programming languages support regular expressions, as do lots of applications, such as WebMatrix, Sublime Text, and Dreamweaver, in which the Find facility allows regular expressions to be used. 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. You have 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 is 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 use the first method. The main reason for using the second method is that it allows the regular expression to be determined at run time (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 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 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>

<html lang="en">
<head>
    <title>Chapter 6, Figure 2</title>
</head>
<body>
    <script>
        var myString = "Paul, Paula, Pauline, paul, Paul";
        var myRegExp = /Paul/;

        myString = myString.replace(myRegExp, "Ringo");
        alert(myString);
    </script>
</body>
</html>
```

You can save and run this code. You will see the screen shown in Figure 6-2.

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.

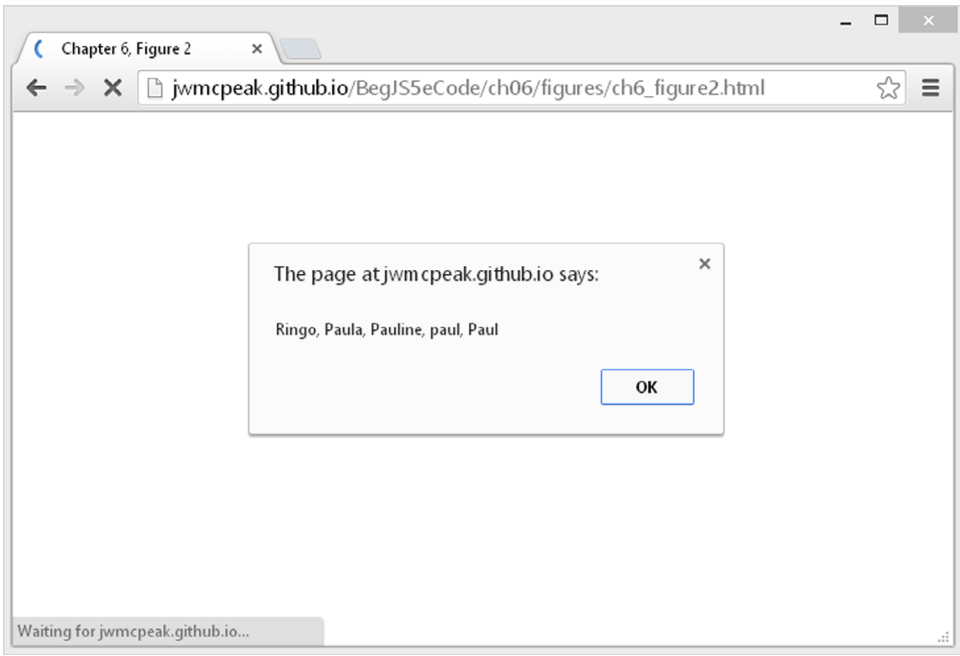

FIGURE 6-2

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, as 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. 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 learn more about these attribute 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 6-3.

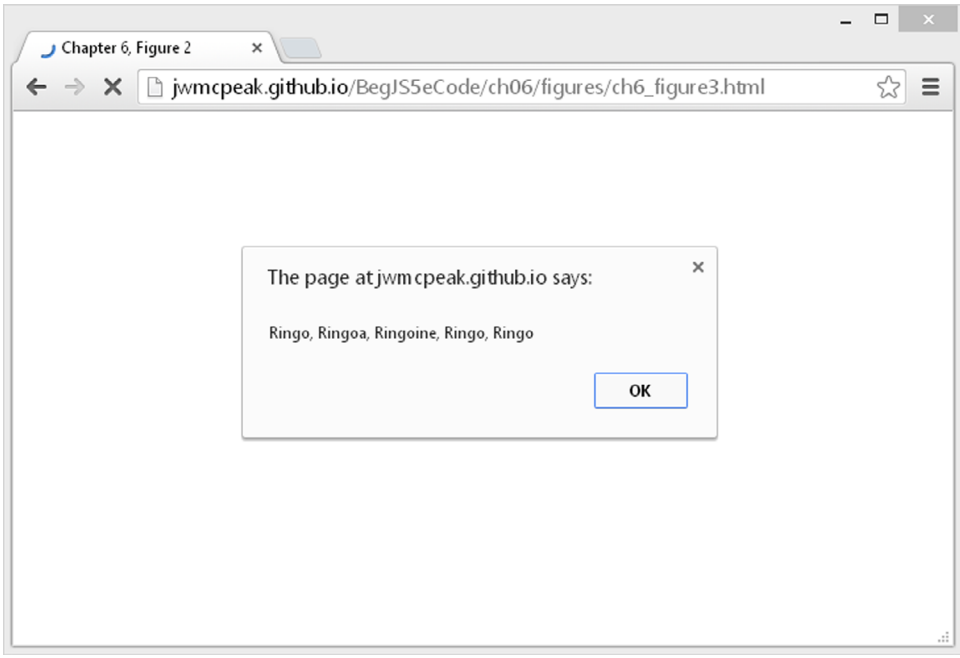

FIGURE 6-3

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 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 look at three types of special characters in this section.

Text, Numbers, and Punctuation

The first group of special characters 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 newline character (`\n`)	`.` matches a or 4 or @.
`[...]`	Any one of the characters between the brackets `[a-z]` matches any character in the range a to z	`[abc]` matches a or b or c, but nothing else.
`[^...]`	Any one character, but not one of those inside the brackets	`[^abc]` matches any character except a or b or c. `[^a-z]` matches 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 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 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>

<html lang="en">
<head>
    <title>Chapter 6, Example 2</title>
</head>
<body>
    <script>
        var input = prompt("Please enter a pass phrase.", "");

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

        if (isValid(input)) {
            alert("Your passphrase contains only valid characters");
        } else {
            alert("Your passphrase contains one or more invalid characters");
        }
    </script>
</body>
</html>
```

Save the page as `ch6_example2.html`, and then load it into your browser. Type just letters, numbers, and spaces into the prompt box, click OK, 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 `isValid()` function. As its name implies, it checks the validity of the passphrase:

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

The function takes just one parameter: the text you want to validate. 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. You 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 matches 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 matches 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 `isValid()` but without regular expressions:

```
function isValid2(text) {
   var returnValue = true;
   var validChars = "abcdefghijklmnopqrstuvwxyz1234567890 ";

   for (var charIndex = 0; charIndex < text.length;charIndex++) {
      if (validChars.indexOf(text.charAt(charIndex).toLowerCase()) < 0) {
         returnValue = false;
         break;
      }
   }
   return returnValue;
}
```

This is probably as small as the non-regular expression version can be, and yet it's still several lines longer than `isValid()`.

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 10 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: you use an `if...else` statement to display the appropriate message to the user. If the passphrase is valid, an `alert` box tells the user that all is fine:

```
if (isValid(input)) {
    alert("Your passphrase contains only valid characters");
}
```

If it isn't, another `alert` box tells users 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, you have 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 includes 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 6-4.

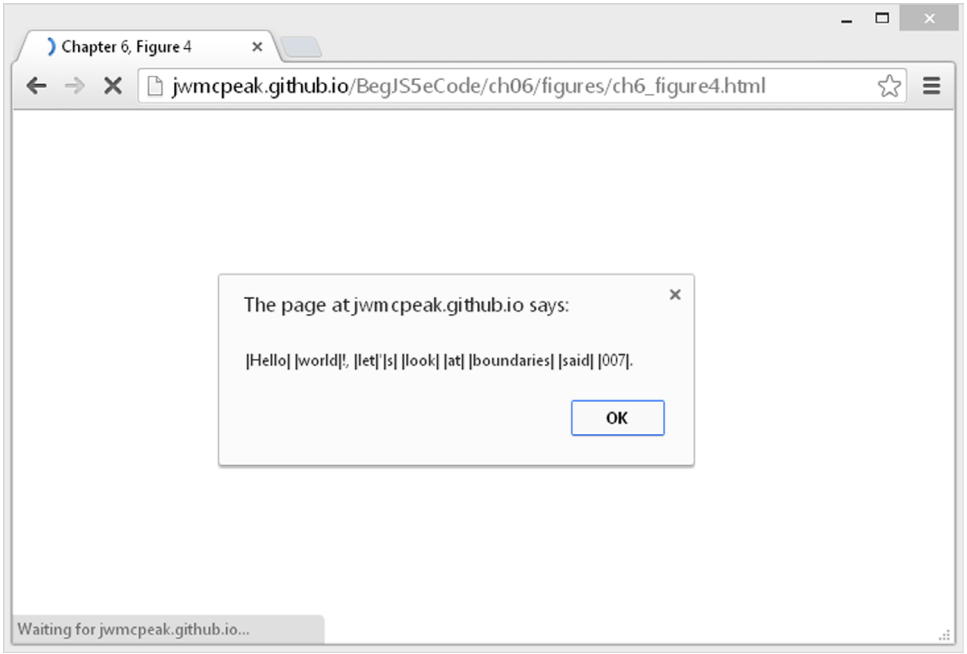

FIGURE 6-4

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 6-5.

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>

<html lang="en">
<head>
```

```
    <title>Chapter 6, Figure 2</title>
</head>
<body>
    <script>
        var myString = "Paul, Paula, Pauline, paul, Paul";
        var myRegExp = /Paul/gi;

        myString = myString.replace(myRegExp, "Ringo");
        alert(myString);
    </script>
</body>
</html>
```

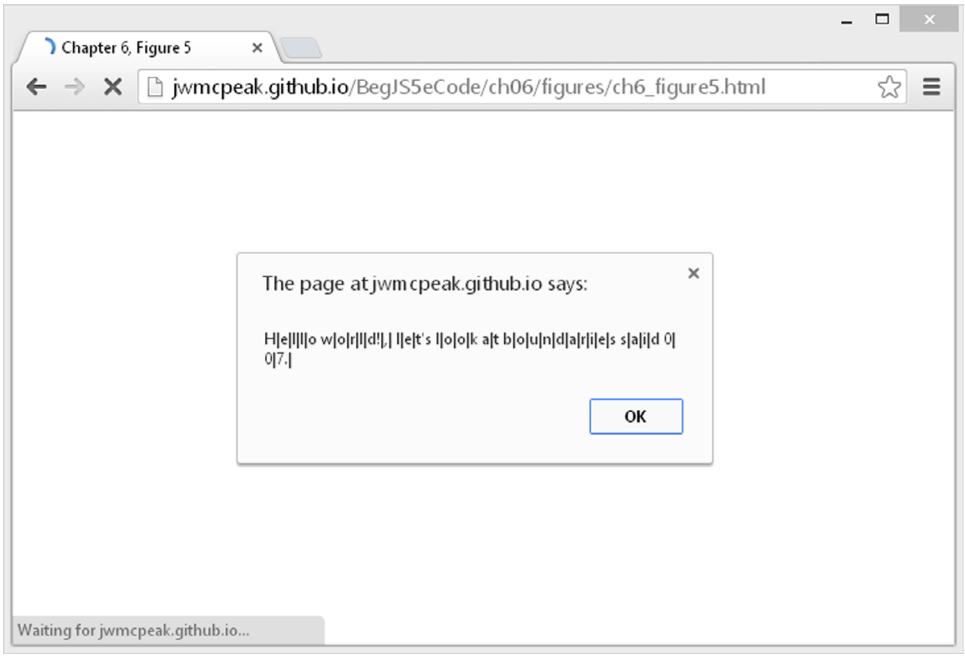

FIGURE 6-5

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 6-6.

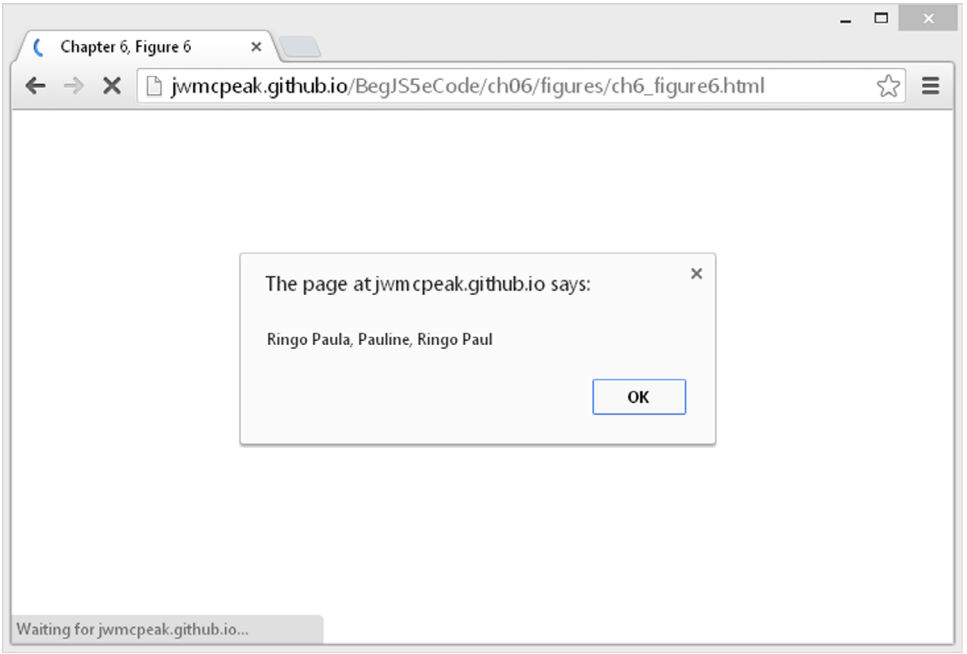

FIGURE 6-6

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 6-7.

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";
```

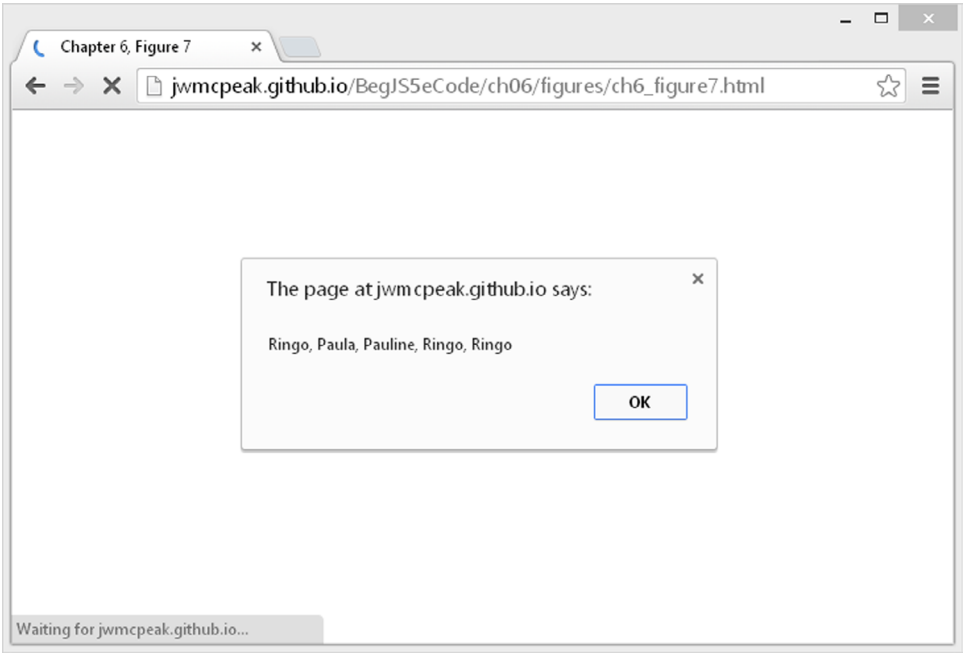

FIGURE 6-7

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 PHP";
```

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 instance of the characters `Java` or `VB`, and ending in `Script`:

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

Breaking down this expression, 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 PHP";
var myRegExp = /\b(VB)?(Java)?Script\b/gi;
myString = myString.replace(myRegExp, "xxxx");
alert(myString);
```

The output of this code is shown in Figure 6-8.

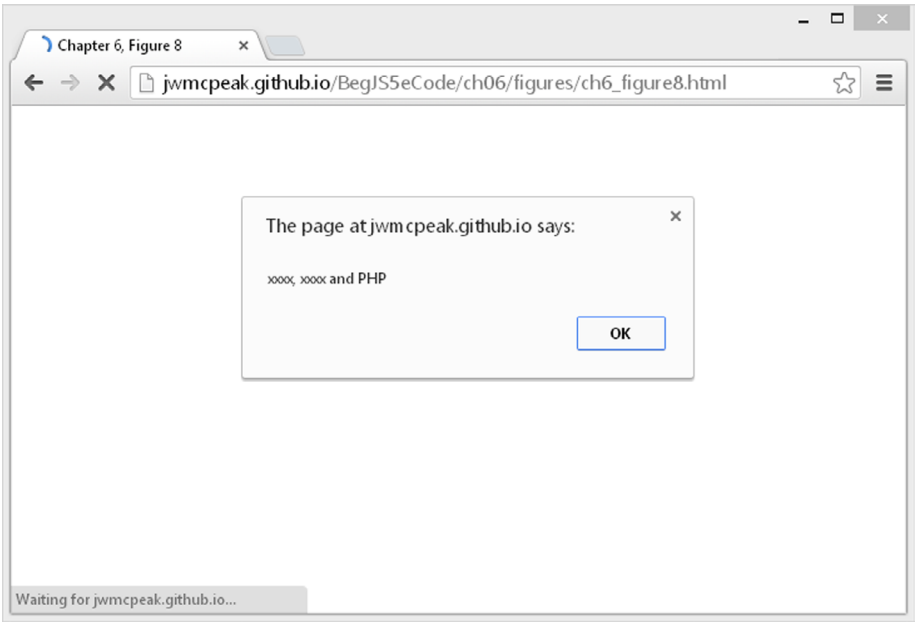

FIGURE: 6-8

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, you can refer to the first group 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? Because 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. You can do this 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 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 look at the String object's split(), replace(), search(), and match() methods with plenty more examples of regular expression syntax.

THE STRING OBJECT

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 in this section you 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 you 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>

<html lang="en">
<head>
    <title>Chapter 6, Example 3</title>
</head>
<body>
    <script>
        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 `ch6_example3.html` 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 `$` sign and the group's number. Each group in a regular expression is given a number from `1` to `99`; any groups greater than `99` are not accessible. To refer to a group, you write `$` 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 = "2012, 2013, 2014";
```

If you wanted to change this to `"the year 2012, the year 2013, the year 2014"`, 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 2012, the year 2013, the year 2014"`.

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

```
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>

<html lang="en">
<head>
    <title>Chapter 6, Example 4</title>
</head>
<body>
    <script>
        var text = "He then said 'My Name is O'Connerly, yes " +
                   "that's right, O'Connerly'";

        document.write("Original: " + text + "<br/>");

        var myRegExp = /\B'|'\B/g;
        text = text.replace(myRegExp, '"');

        document.write("Corrected: " + text);
    </script>
</body>
</html>
```

Save the page as `ch6 _ example4.html`. Load the page into your browser and you should see what is shown in Figure 6-9.

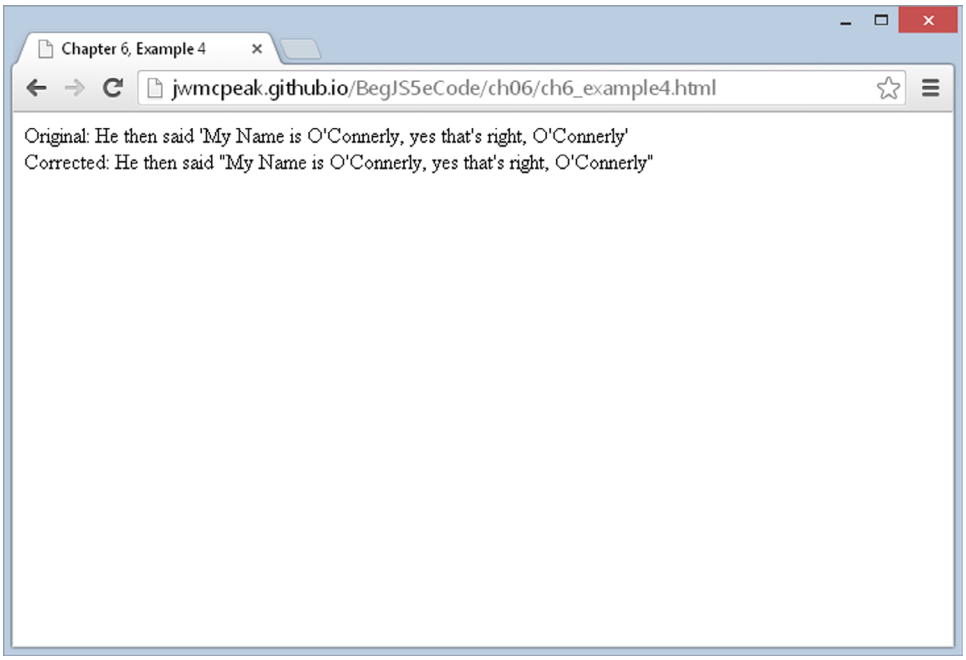

FIGURE 6-9

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.

The workhorses of this code are two simple lines:

```
var myRegExp = /\B'|'\B/g;
text = text.replace(myRegExp, '"');
```

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.

The second line of code uses the `replace()` method to do the character pattern search and replace, and assigns the new value to the `text` variable.

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, `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 2012, 2013 and 2014";
```

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) {
  for (var index = 0; index < resultsArray.length; index++) {
```

```
        alert(resultsArray[index]);
    }
}
```

This would result in three `alert` boxes containing the numbers `2012`, `2013`, and `2014`.

TRY IT OUT Splitting HTML

In this 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:

	Hello	

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Chapter 6, Example 5</title>
</head>
    <body>
        <div id="output"></div>
        <script>
            var html = "<h2>Hello World!</h2>" +
                "<p>We love JavaScript!</p>";

            var regex = /<[^>\r\n]+>|[^<>\r\n]+/g;
            var results = html.match(regex);

            document.getElementById("output").innerText = results.join("\r\n");
        </script>
    </body>
</html>
```

Save this file as `ch6_example5.html`. When you load the page into your browser you'll see that the string of HTML is split, with each element's tags and content displayed on separate lines, as shown in Figure 6-10.

Once again, the code that makes all of this work consists of just a few lines. You first create a `RegExp` object and initialize it to your regular expression:

```
var regex = /<[^>\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. 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 `>`.

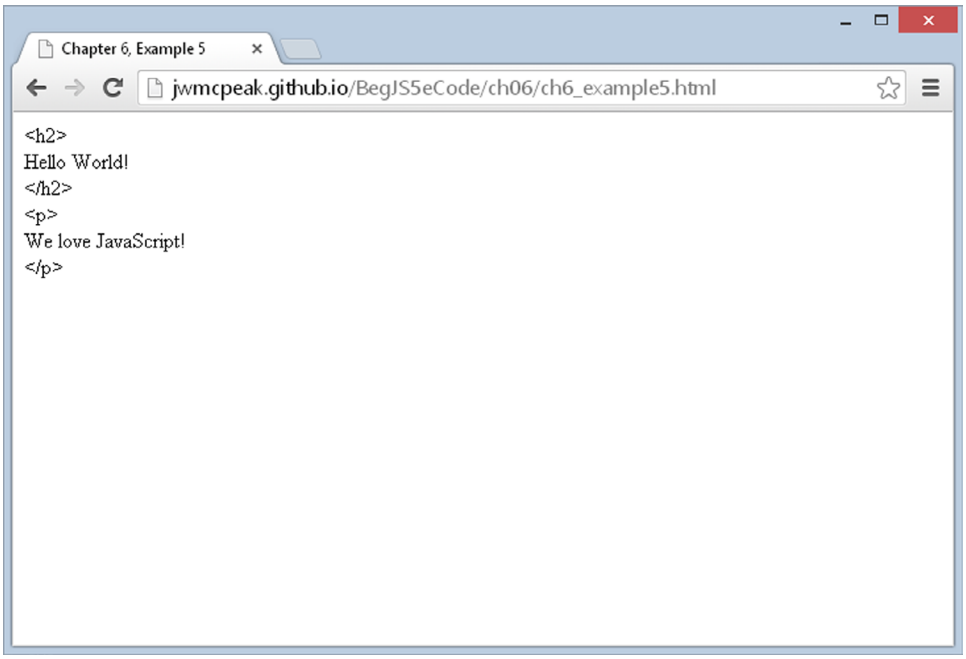

FIGURE 6-10

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 plaintext.

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 `results` variable to the `Array` object returned by the `match()` method:

```
var results = html.match(regex);
```

The remainder of the code populates a `<div/>` element with the split HTML:

```
document.getElementById("output").innerText = results.join("\r\n");
```

This code uses features you haven't yet seen. It essentially retrieves the element that has an `id` value of `output`; this is the `<div/>` element at the top of the body. The `innerText` property enables you to set the text inside of the `<div/>` element. You learn more in later chapters.

You then 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.

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, on some occasions 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 define regular expressions with the / and / method, after the final / you could add 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 want—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 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 `ch6 _ example6.js`.

To test the code, you need a simple page:

```
<!DOCTYPE html>

<html lang="en">
```

```
<head>
    <title>Chapter 6, Example 6</title>
</head>
    <body>
        <script src="ch6_example6.js"></script>
        <script>
            var phoneNumber = prompt("Please enter a phone number.", "");

            if (isValidTelephoneNumber(phoneNumber)) {
                alert("Valid Phone Number");
            } else {
                alert("Invalid Phone Number");
            }

            var postalCode = prompt("Please enter a postal code.", "");

            if (isValidPostalCode(postalCode)) {
                alert("Valid Postal Code");
            } else {
                alert("Invalid Postal Code");
            }

            var email = prompt("Please enter an email address.", "");

            if (isValidEmail(email)) {
                alert("Valid Email Address");
            } else {
                alert("Invalid Email Address");
            }
        </script>
    </body>
</html>
```

Save this as `ch6_example6.html` and load it into your browser, and you'll be prompted to enter a phone number. Enter a valid telephone number (for example, +1 (123) 123 4567), and you'll see a message that states whether or not the phone number is valid.

You'll then be prompted to enter a postal code and an e-mail. Enter those values to test those functions. 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.
- A valid number can be entered in different ways (for example, with or without the national or international code).

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 your 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, this won't work in some countries, 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're matching a plus sign `(\+)` followed by one to three digits `(\d{1,3})` and an optional space `(?)`. Remember that because 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 time, 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, because

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*** *The initial space and ? mean "match zero or one space." This is followed by three or four digits (*`\d{3,4}`*)—although U.S. numbers always have three digits, UK numbers often have four. 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 because 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 period (note the `\` character, because `.` 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, let's look once again at the `isValidTelephoneNumber()` function in `ch6_example6.js`:

```
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 multiple 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 United Kingdom and the United States:

```
^(\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 for 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 SE2V 3ER, with a letter at the end of the first part. Currently, only some of those postcodes start with SE, WC, and W, but that may change. Valid examples of UK postcodes 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])$
```

Just for reference, let's look once again at the `isValidPostalCode()` function:

```
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, 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 it contains many variations to cope with. It's best to start by breaking it down. First, note that the latter two versions are exceptionally rare and not provided for in the regular expression you'll create.

Second, you need to break up the e-mail address into separate parts. Let's 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 must follow certain rules. 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, we 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 the engine will match against only 128 characters. 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 square bracket and backslash characters (`[`, `]`, and `\`) 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, here's the `isValidEmail()` function for reference:

```
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 );
}
```

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

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 that allow any of certain groups of characters, such as digits, words, or non-word characters, to be matched.

- You can also use special characters 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 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.

EXERCISES

You can find suggested solutions to these questions 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 website with a message board. Write a regular expression that would remove barred words. (You can make up your own words!)