CHAPTER

4

Working with Words, Numbers, and Dates

Storing information in a variable or an array is just the first step in effectively using data in your programs. As you read in the last chapter, you can use data to make decisions in a program ("Is the score 0?"). You'll also frequently manipulate data by either searching through it (trying to find a particular word in a sentence, for example), manipulating it (rounding a number to a nearest integer), or reformatting it to display properly (formatting a number like 430 to appear in the proper monetary format, like $430.00).

This chapter will show you how to accomplish common tasks when working with strings and numbers. In addition, it'll introduce the JavaScript Date object, which lets you determine the current date and time on a visitor's computer.

A Quick Object Lesson

So far in this book, you've learned that you can write something to a Web page with the *document.write( )* command, and to determine how many items are in an array, you type the name of the array followed by a period and the word "length," like so: *days.length*. You're probably wondering what those periods are about. You've made it through three chapters without learning the particulars of this feature of JavaScript syntax, and it's time to address them.

You can conceptualize many of the elements of the JavaScript language, as well as elements of a Web page, as *objects*. The real world, of course, is filled with objects too, such as a dog or a car. Most objects are made up of different parts: a dog has a tail, a head, and four legs; a car has doors, wheels, headlights, a horn, and so on. An object might also do something—a car can transport passengers, a dog can

bark. In fact, even a part of an object can do something: for example, a tail can wag, and a horn can honk. Table 4-1 illustrates one way to show the relationships between objects, their parts, and actions.

***Table 4-1.** A simplified view of the world*

Object	Parts	Actions
dog		bark
	tail	wag
car		transport
	horn	honk

The world of JavaScript is also filled with objects: a browser window, a document, an array, a string, and a date are just a few examples. Like real-world objects, Java-Script objects are also made up of different parts. In programming-speak, the parts of an object are called *properties*. The actions an object can perform are called *methods*, which are basically functions (like the ones you created in the previous chapter) that are specific to an object (see Table 4-2).

Note: You can always tell a method from a property because methods end in parentheses: *write( )*, for example.

Each object in JavaScript has its own set of properties and methods. For example, the array object has a property named *length*, and the document object has a method named *write( )*. To access an object's property or execute one of its meth-ods, you use *dot-syntax*—those periods! The dot (period) connects the object with its property or method. For example, *document.write( )* means "run the *write( )* method of the document object." If the real world worked like that, you'd have a dog wag his tail like this: *dog.tail.wag( )* (of course, in the real world, a doggy treat works a lot better).

***Table 4-2.** Some methods and properties of an array object (see page 56 for more information on arrays)*

An array object	Property	Method
['Bob', 'Jalia', 'Sonia']	length	
		push()
		pop()
		shift()

And just as you might own several dogs in the real world, your JavaScript pro-grams can have multiple versions of the same kind of object. For example, say you create two simple variables like this:

```
var first_name = 'Jack';
var last_name = 'Hearts';
```

You've actually created two different *string* objects. Strings (as you'll see in this chapter) have their own set of properties and methods, which are different from the methods and properties of other objects, like arrays. When you create an object (also called creating an *instance* of that object) you can access all of the properties and methods for that object. You've actually been doing that in the last few chapters without even realizing it. For example, you can create an array like this:

```
var names = ['Jack', 'Queen', 'King'];
```

The variable *names* is an instance of an array object. To find out the number of items in that array, you access that array's *length* property using dot notation:

```
names.length
```

Likewise, you can add an item to the end of that array by using the array object's *push( )* method like this (see page 61 for a refresher on array methods):

```
names.push('Ace');
```

Whenever you create a new variable and store a value into it, you're really creating a new instance of a particular type of object. So each of these lines of JavaScript create different types of JavaScript objects:

```
var first_name = 'Bob'; // a string object
var age = 32; // a number object
var valid = false; // a Boolean object
var data = ['Julia', 22, true]; // an array object composed of other objects
```

In fact, when you change the type of information stored in a variable, you change the type of object it is as well. For example, if you create a variable named *data* that stores an array, then store a number in the variable, you've changed that variable's type from an array to a number object:

```
var data = ['Julia', 22, true]; // an array object composed of other objects
data = 32; //changes to number object
```

The concepts of objects, properties, methods, and dot-syntax may seem a little weird at first glance. However, since they are a fundamental part of how JavaScript works, you'll get used to them pretty quickly.

Tip: As you continue reading this book, keep in mind these few facts:

- The world of JavaScript is populated with lots of different types of objects.
- Each object has its own properties and methods.
- You access an object's property or activate an object's method using dot-syntax: *document.write( )*, for example.

Strings

Strings are the most common type of data you'll work with: input from form fields, the path to an image, a URL, and HTML that you wish to replace on a page are all examples of the letters, symbols, and numbers that make up strings. Consequently, JavaScript provides lot of methods for working with and manipulating strings.

Determining the Length of a String

There are times when you want to know how many characters are in a string. For example, say you want to make sure that when someone creates an account on your top secret Web site, they create a new password that's more than 6 letters but no more than 15. Strings have a *length* property that gives you just this kind of information. Add a period after the name of the variable, followed by *length* to get the number of characters in the string: *name.length*.

For example, to make sure a password has the proper number of characters, you could use a conditional statement to test the password's length like this:

```
var password = 'sesame';
if (password.length <= 6) {
  alert('That password is too short.');
} else if (password.length > 15) {
  alert('That password is too long.');
}
```

Note: In the above example, the password is just directly assigned to the variable *var password = 'sesame'*. In a real world scenario, you'd get the password from a form field, as described on page 312.

Changing the Case of a String

JavaScript provides two methods to convert strings to all uppercase or all lowercase, so you can change 'hello' to 'HELLO' or 'NOT' to 'not'. Why, you might ask? Converting letters in a string to the same case makes comparing two strings easier. For example, say you created a Quiz program like the one from last chapter (see page 106) and one of the questions is "Who was the first American to win the Tour De France?" You might have some code like this to check the quiz-taker's answer:

```
var correctAnswer = 'Greg LeMond';
var response = prompt('Who was the first American to win the Tour De↵
France?', '');
if (response == correctAnswer) {
  // correct
} else {
  // incorrect
}
```

The answer is Greg LeMond, but what if the person taking the quiz typed Greg Lemond? The condition would look like this: 'Greg Lemond' == 'Greg LeMond'. Since JavaScript treats uppercase letters as different than lowercase letters, the lowercase 'm' in Lemond wouldn't match the 'M' in LeMond, so the quiz-taker would have gotten this question wrong. The same would happen if her key-caps key was down and she typed GREG LEMOND.

To get around this difficulty, you can convert both strings to the same case and then compare them:

```
if (response.toUpperCase( ) == correctAnswer.toUpperCase( )) {
  // correct
} else {
  // incorrect
}
```

In this case, the conditional statement converts both the quiz-taker's answer and the correct answer to uppercase, so 'Greg Lemond' becomes 'GREG LEMOND' and 'Greg LeMond' becomes 'GREG LEMOND'.

To get the string all lowercase, use the *toLowerCase( )* method like this:

```
var answer = 'Greg LeMond';
alert(answer.toLowerCase( )); // 'greg lemond'
```

Note that neither of these methods actually alters the original string stored in the variable—they just return that string in either all uppercase or all lowercase. So in the above example, *answer* still contains 'Greg LeMond' even after the alert appears. (In other words, these methods work just like a function that returns some other value as described on page 102.)

Searching a String: *indexOf()* Technique

JavaScript provides several techniques for searching for a word, number, or other series of characters inside a string. Searching can come in handy, for example, if you want to know which Web browser a visitor is using to view your Web site. Every Web browser identifies information about itself in a string containing a lot of different statistics. You can see that string for yourself by adding this bit of JavaScript to a page and previewing it in a Web browser:

```
<script type="text/javascript">
alert(navigator.userAgent);
</script>
```

Navigator is one of a Web browser's objects, and *userAgent* is a property of the navigator object. The *userAgent* property contains a long string of information; for example, on Internet Explorer 7 running on Windows XP, the *userAgent* property is: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1). So, if you want to see if the Web browser was IE 7, you can just search the *userAgent* string for "MSIE 7".

One method of searching a string is the *indexOf( )* method. Basically, after the string you add a period, *indexOf( )* and supply the string you're looking for. The basic structure looks like this:

```
string.indexOf('string to look for')
```

The *indexOf( )* method returns a number: if the search string isn't found, the method returns –1. So if you want to check for Internet Explorer, you can do this:

```
var browser = navigator.userAgent; // this is a string
if (browser.indexOf('MSIE') != -1) {
  // this is Internet Explorer
}
```

In this case, if *indexOf( )* doesn't locate the string 'MSIE' in the *userAgent* string, it will return –1, so the condition tests to see if the result is not (*!=*) –1.

When the *indexOf( )* method *does* find the searched for string, it returns a number that's equal to the starting position of the searched for string. The following example makes things a lot clearer:

```
var quote = 'To be, or not to be.'
var searchPosition = quote.indexOf('To be'); // returns 0
```

Here, *indexOf( )*searches for the position of 'To be' inside the string 'To be, or not to be.' The larger string begins with 'To be', so *indexOf( )* finds 'To be' at the first position. But in the wacky way of programming, the first position is considered 0, the second letter (o) is at position 1, and the third letter (a space in this case) is 2 (as explained on page 59, arrays are counted in the same way).

The *indexOf( )* method searches from the beginning of the string. You can also search from the end of the string by using the *lastIndexOf( )* method. For example, in the Shakespeare quote, the word 'be' appears in two places, so you can locate the first 'be' using *indexOf( )* and the last 'be' with *lastIndexOf( )*:

```
var quote = "To be, or not to be."
var firstPosition = quote.indexOf('be'); // returns 3
var lastPosition = quote.lastIndexOf('be'); // returns 17
```

The results of those two methods are pictured in Figure 4-1. In both cases, if 'be' didn't exist anywhere in the string, the result would be –1, and if there's only one instance of the searched-for word, *indexOf( )* and *lastIndexOf( )* will return the same value—the starting position of the searched for string within the larger string.

Extracting Part of a String with *slice()*

To extract part of a string, use the *slice( )* method. This method returns a portion of a string. For example, say you had a string like "http://www.sawmac.com" and

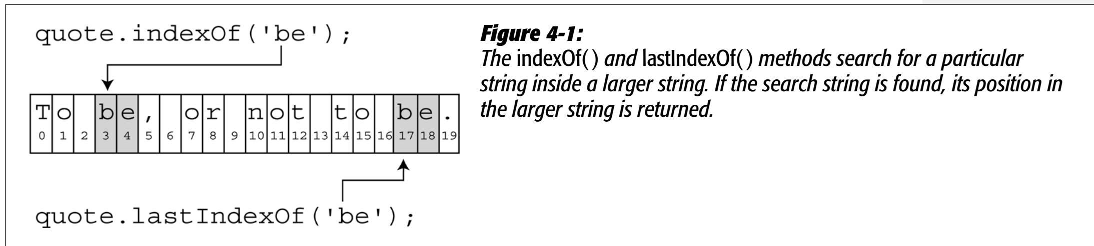

Figure 4-1:
The indexOf( ) *and* lastIndexOf( ) *methods search for a particular string inside a larger string. If the search string is found, its position in the larger string is returned.*

you wanted to eliminate the http:// part. One way to do this is to extract every character in the string that follows the http:// like this:

```
var url = 'http://www.sawmac.com';
var domain = url.slice(7); // www.sawmac.com
```

The *slice( )* method requires a *number* that indicates the starting *index* position for the extracted string (see Figure 4-2). In this example—*url.slice(7)*—the 7 indicates the eighth letter in the string (remember, the first letter is at position 0). The method returns all of the characters starting at the specified index position to the end of the string.

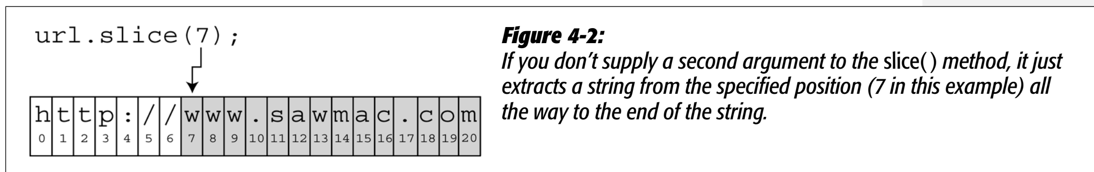

Figure 4-2:
If you don't supply a second argument to the slice( ) *method, it just extracts a string from the specified position (7 in this example) all the way to the end of the string.*

You can also extract a specific number of characters within a string by supplying a second argument to the *slice( )* method. Here's the basic structure of the *slice( )* method:

```
string.slice(start, end);
```

The *start* value is a number that indicates the first character of the extracted string; the end *value* is a little confusing—it's not the position of the last letter of the extracted string; it's actually the position of the last letter + 1. For example, if you wanted to extract the first five letters of the string 'To be, or not to be.', you would specify 0 as the first argument, and 5 as the second argument. As you can see in Figure 4-3, 0 is the first letter in the string, and 5 is the sixth letter, but the last letter specified is not extracted from the string. In other words, the character specified by the second argument is *never* part of the extracted string.

Tip: If you want to extract a specific number of characters from a string, just add that number to the starting value. For example, if you want to retrieve the first 10 letters of a string, the first argument would be 0 (the first letter) and the last would be 0 + 10 or just 10: *slice(0,10)*.

You can also specify negative numbers, for example *quote.slice(-6,-1)*. A negative number counts backwards from the end of the string, as pictured in Figure 4-3.

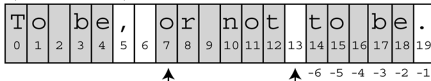

Figure 4-3: *The* slice( ) *method extracts a portion of a string. The actual string is not changed in any way. For instance, the string contained in the quote variable in this example isn't changed by* quote.slice(0,5)*. The method simply returns the extracted string, which you can store inside a variable, display in an alert box, or even pass as an argument to a function.*

Tip: If you want, say, to extract a string that includes all of the letters from the 6th letter from the end of the string all the way to the end, you leave off the second argument:

```
quote.slice(-6);
```

Finding Patterns in Strings

Sometimes you wish to search a string, not for an exact value, but for a specific pattern of characters. For example, say you want to make sure when a visitor fills out an order form, he supplies a phone number in the correct format. You're not actually looking for a specific phone number like 503-555-0212. Instead you're looking for a general pattern: 3 numbers, a hyphen, three numbers, another hyphen, and four numbers. You'd like to check the value the visitor entered, and if it matches the pattern (for example, it's 415-555-3843, 408-555-3782, or 212-555-4828, and so on) then everything's OK. But if it doesn't match that pattern (for example, the visitor typed 823lkjxdfglkj) then you'd like to post a message like "Hey buddy, don't try to fool us!"

JavaScript lets you use *regular expressions* to find patterns within a string. A regular expression is a series of characters that define a pattern that you wish to search for. As with many programming terms, the name "regular expression" is a bit misleading. For example, here's what a common regular expression looks like:

```
/^[-\w.]+@([a-zA-Z0-9][-a-zA-Z0-9]+\.)+[a-zA-Z]{2,4}$/
```

Unless you're a super-alien from Omicron 9, there's nothing very *regular*-looking about a regular expression. To create a pattern you use characters like *, +, ?, and \ *w*, which are translated by the JavaScript interpreter to match real characters in a string like letters, numbers, and so on.

Note: Pros often shorten the name regular expression to *regex*.

Creating and Using a Basic Regular Expression

To create a regular expression in JavaScript, you must create a regular expression object, which is a series of characters between two forward slashes. For example, to create a regular expression that matches the word 'hello', you'd type this:

```
var myMatch = /hello/;
```

Just as an opening and closing quote mark creates a string, the opening / and closing / create a regular expression.

There are several string methods that take advantage of regular expressions (you'll learn about them starting on page 131), but the most basic method is the *search( )* method. It works very much like the *indexOf( )* method, but instead of trying to find one string inside another, larger string, it searches for a pattern (a regular expression) inside a string. For example, say you want to find 'To be' inside the string 'To be or not to be.' You saw how to do that with the *indexOf( )* method on page 117, but here's how you can do the same thing with a regular expression:

```
var myRegEx = /To be/; // no quotes around regular expression
var quote = 'To be or not to be.';
var foundPosition = quote.search(myRegEx);  // returns 0
```

If the *search( )* method finds a match, it returns the position of the first letter matched, and if it doesn't find a match, it returns –1. So in the above example, the variable *foundPosition* is 0, since 'To be' begins at the very beginning (the first letter) of the string.

As you'll recall from page 117, *indexOf( )* method works in the same way. You might be thinking that if the two methods are the same, why bother with regular expressions? The benefit of regular expressions is that they can find patterns in a string, so they can make much more complicated and subtle comparisons than the *indexOf( )* method, which always looks for a match to an exact string. For example, you could use the *indexOf( )* method to find out if a string contains the Web address *http://www.missingmanuals.com/*, but you'd have to use a regular expression to find any text that matches the format of a URL—exactly the kind of thing you want to do when verifying if someone supplied a Web address when posting a comment to your blog.

However, to master regular expressions, you need to learn the often confusing symbols required to construct a regular expression.

Building a Regular Expression

While a regular expression can be made up of a word or words, more often you'll use a combination of letters and special symbols to define a pattern that you hope to match. Regular expressions provide different symbols to indicate different types of characters. For example, a single period (.) represents a single character, any character, while *\w* matches any letter or number (but not spaces, or symbols like $ or %). Table 4-3 provides a list of the most common pattern-matching characters.

Note: If this entire discussion of "regular" expressions is making your head hurt, you'll be glad to know this book provides some useful regular expressions (see page 126) that you can copy and use in your own scripts (without really knowing how they work).

Learning regular expressions is a topic better presented by example, so the rest of this section walks you through a few examples of regular expressions to help you wrap your mind around this topic. Assume you want to match five numbers in a row—perhaps to check if there's a U. S. Zip code in a string:

1. **Match one number.**

   The first step is simply to figure out how to match one number. If you refer to Table 4-3, you'll see that there's a special regex symbol for this, *\d*, which matches any single number.

2. **Match five numbers in a row.**

   Since *\d* matches a single number, a simple way to match five numbers is with this regular expression: *\d\d\d\d\d*. (Page 124, however, covers a more compact way to write this.)

3. **Match only five numbers.**

   A regular expression is like a precision-guided missile: It sets its target on the first part of a string that it matches. So, you sometimes end up with a match that's part of a complete word or set of characters. This regular expression matches the first five numbers in a row that it encounters. For example, it will match 12345 in the number 12345678998. Obviously, 12345678998 isn't a Zip code, so you need a regex that targets just five numbers.

   The *\b* character (called the *word boundary* character) matches any nonletter or nonnumber character, so you could rewrite your regular expression like this: *\b\d\d\d\d\d\b*. You can also use the ^ character to match the beginning of a string and the $ character to match the end of a string. This trick comes in handy if you want the entire string to match your regular expression. For example, if someone typed "kjasdflkjsdf 88888 lksadflkjsdkfjl" in a Zip code field on an order form, you might want to ask the visitor to clarify (and fix) their Zip code before ordering. After all, you're really looking for something like 97213 (with no other characters in the string). In this case, the regex would be *^\d\d\d\d\d$*.

Note: Zip codes can have more than five numbers. The ZIP + 4 format includes a dash and four additional numbers after the first five, like this: 97213-1234. For a regular expression to handle this possibility, see page 126.

4. **Put your regex into action in JavaScript.**

   Assume you've already captured a user's input into a variable named *zip*, and you want to test to see if the input is in the form of a valid five-number Zip code:

```
var zipTest = /^\d\d\d\d\d$/; //create regex
if (zip.search(zipTest) == -1) {
  alert('This is not a valid zip code');
} else {
  // is valid format
}
```

Table 4-3.

<table>
<tr><th>Character</th><th>Matches</th></tr>
<tr><td>.</td><td>Any one character—will match a letter, number, space, or other symbol</td></tr>
<tr><td>\w</td><td>Any word character including a–z, A–Z, the numbers 0–9, and the underscore character: _.</td></tr>
<tr><td>\W</td><td>Any character that's not a word character. It's the exact opposite of \w.</td></tr>
<tr><td>\d</td><td>Any digit 0–9.</td></tr>
<tr><td>\D</td><td>Any character except a digit. The opposite of \d.</td></tr>
<tr><td>\s</td><td>A space, tab, carriage return, or new line.</td></tr>
<tr><td>\S</td><td>Anything but a space, tab, carriage return, or new line.</td></tr>
<tr><td>^</td><td>The beginning of a string. This is useful for making sure no other characters come before whatever you're trying to match.</td></tr>
<tr><td>$</td><td>The end of a string. Use $ to make sure the characters you wish to match are at the end of a string. For example, /com$/ matches the string "com", but only when it's the last three letters of the string. In other words, /com$/ would match "com" in the string "Infocom," but not 'com' in 'communication'.</td></tr>
<tr><td>\b</td><td>A space, beginning of the string, end of string, or any nonletter or number character such as +, =, or '. Use \b to match the beginning or ending of a word, even if that word is at the beginning or ending of a string.</td></tr>
<tr><td>[ ]</td><td>Any one character between the brackets. For example, [aeiou] matches any one of those letters in a string. For a range of characters, use a hyphen: [a-z] matches any one lower case letter; [0-9] matches any one number (the same as \d.)</td></tr>
<tr><td>[^]</td><td>Any character except one in brackets. For example, [^aeiouAEIOU] will match any character that isn't a vowel. [^0-9] matches any character that's not a number (the same as \D).</td></tr>
<tr><td>|</td><td>Either the characters before or after the | character. For example, a|b will match either a or b, but not both. (See page 130 for an example of this symbol in action.)</td></tr>
<tr><td>\</td><td>Used to escape any special regex symbol (*,.,\,/, for instance) to search for a literal example of the symbol in a string. For example, . in regex-speak means "any character," but if you want to really match a period in a string you need to escape it, like this: \. .</td></tr>
</table>

The regex example in these steps works, but it seems like a lot of work to type *\d* five times. What if you want to match 100 numbers in a row? Fortunately, JavaScript includes several symbols that can match multiple occurrences of the same character. Table 4-4 includes a list of these symbols. You place the symbol directly *after* the character you wish to match.

For example, to match five numbers, you can write *\d{5}*. The *\d* matches one number, then the *{5}* tells the JavaScript interpreter to match five numbers. So *\d{100}* would match 100 digits in a row.

Let's go through another example. Say you wanted to find the name of any GIF file in a string. In addition, you want to extract the file name and perhaps use it somehow in your script (for example, you can use the *match( )* method described on page 131). In other words, you want to find any string that matches the basic pattern of a GIF file name, such as *logo.gif, banner.gif* or *ad.gif*.

1. **Identify the common pattern between these names.**

   To build a regular expression, you first need to know what pattern of characters you're searching for. Here, since you're after GIFs, you know all the file names will end in .gif. In other words, there can be any number of letters or numbers or other characters before .gif.

2. **Find .gif.**

   Since you're after the literal string '.gif', you might think that part of the regular expression would just be *.gif*. However, if you check out Table 4-3, you'll see that a period has special meaning as a "match any character" character. So *.gif* would match ".gif," but it would also match "tgif." A period matches any single character so in addition to matching a period, it will also match the "t" in tgif. To create a regex with a literal period, add a slash before it; so \. translates to "find me the period symbol". So the regex to find .gif would be *\.gif*.

3. **Find any number of characters before .gif.**

   To find any number of characters, you can use .*, which translates to "find one character (.) zero or more times (*)." That regular expression matches all of the letters in any string. However, if you used that to create a regular expression like *.*\.gif*, you could end up matching more than just a file name. For example, if you have the string 'the file is logo.gif', the regex *.*\.gif* will match the entire string, when what you really want is just logo.gif. To do that, use the *\S* character, which matches any nonspace character: *\S*\.gif* matches just logo.gif in the string.

4. **Make the search case-insensitive.**

   There's one more wrinkle in this regular expression: it only finds files that end in .gif, but .GIF is also a valid file extension, so this regex wouldn't pick up on a name like logo.GIF. To make a regular expression ignore the difference between

upper and lowercase letters, you use the *i* argument when you create the regular expression:

```
/\S*\.gif/i
```

Notice that the *i* goes outside of the pattern and to the right of the / that defines the end of the regular expression pattern.

5. **Put it into action:**

```
var testString = 'The file is logo.gif'; // the string to test
var gifRegex = /\S*\.gif/i; // the regular expression
var results = testString.match(gifRegex);
var file = results[0]; // logo.gif
```

This code pulls out the file name from the string. (You'll learn how the *match( )* method works on page 131.)

Grouping Parts of a Pattern

You can use parentheses to create a subgroup within a pattern. Subgroups come in very handy when using any of the characters in Table 4-4 to match multiple instances of the same pattern.

Table 4-4. *Characters used for matching multiple occurrences of the same character or pattern*

Character	Matches
?	Zero or one occurrences of the previous item, meaning the previous item is optional, but if it does appear, it can only appear once. For example the regex *colou?r* will match both "color" and "colour," but not "colouur."
+	One or more occurrences of the previous item. The previous item must appear at least once.
*	Zero or more occurrences of the previous item. The previous item is optional and may appear any number of times. For example, *.** matches any number of characters.
{n}	An exact number of occurrences of the previous item. For example \d{3} only matches three numbers in a row.
{n, }	The previous item *n* or more times. For example, a{2,} will match the letter "a" two or more times: that would match "aa" in the word "aardvark" and "aaaa" in the word "aaaahhhh."
{n,m}	The previous item at least *n* times but no more than *m* times. So \d{3,4} will match three or four numbers in a row (but not two numbers in a row, nor five numbers in a row).

For example, say you want to see if a string contains either "Apr" or "April"—both of those begin with "Apr," so you know that you want to match that, but you can't just match "Apr," since you'd also match the "Apr" in "Apricot" or "Aprimecorp." So, you must match "Apr" followed by a space or other word ending (that's the *\b* regular expression character described in Table 4-3) or April followed by a word

ending. In other words, the "il" is optional. Here's how you could do that using parentheses:

```
var sentence = 'April is the cruelest month.';
var aprMatch = /Apr(il)?\b/;
if (sentence.search(aprMatch) != -1) {
  // found Apr or April
} else {
   //not found
}
```

The regular expression used here—*/Apr(il)?\b/*—makes the "Apr" required, but the subpattern—*(il)*—optional (that *?* character means zero or one time). Finally, the *\b* matches the end of a word, so you won't match "Apricot" or "Aprilshowers." (See the box on page 133 for another use of subpatterns.)

Tip: You can find a complete library of regular expressions at *www.regexlib.com*. At this Web site, you'll find a regular expression for any situation.

Useful Regular Expressions

Creating a regular expression has its challenges. Not only do you need to understand how the different regular expression characters work, but you then must figure out the proper pattern for different possible searches. For example, if you want to find a match for a Zip code, you need to take into account the fact that a Zip code may be just five numbers (97213) or 5+4 (97213-3333). To get you started on the path to using regular expressions, here are a few common ones.

Note: If you don't feel like typing these regular expressions (and who could blame you), you'll find them already set up for you in a file named *example_regex.txt* in the *chapter04* folder that's part of the tutorial download. (See page 27 for information on downloading the tutorial files.)

U.S. Zip code

Postal codes vary from country to country, but in the United States they appear as either five numbers, or five numbers followed by a hyphen and four numbers. Here's the regex that matches both those options:

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

Tip: For regular expressions that match the postal codes of other countries visit *http://regexlib.com/Search.aspx?k=postal+code*.

That regular expression breaks down into the following smaller pieces:

- *\d{5}* matches five digits, as in 97213.