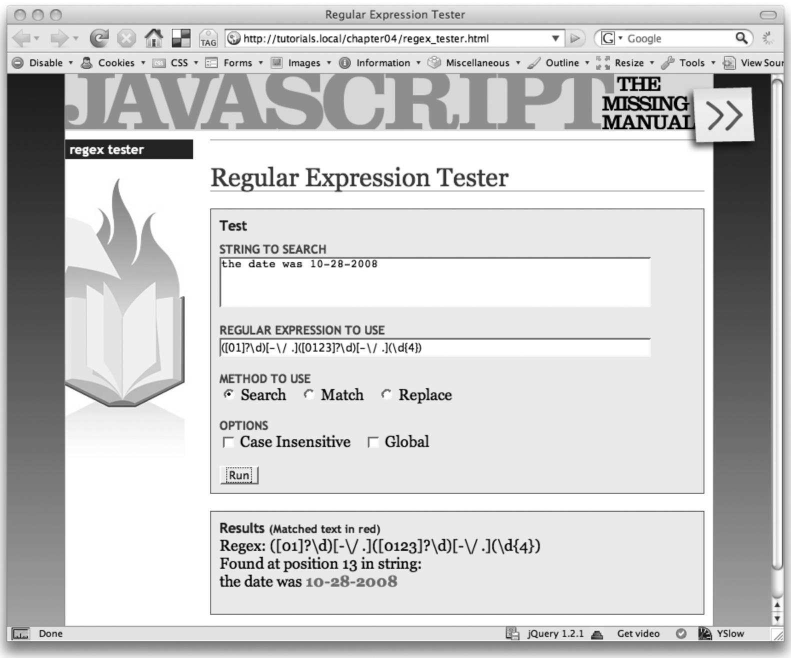

Regular Expression Tester

Test

STRING TO SEARCH

```
the date was 10-28-2008
```

REGULAR EXPRESSION TO USE

```
([01]?\d)[-\/ .]([0123]?\d)[-\/ .](\d{4})
```

METHOD TO USE

[✓ Search] Match Replace

OPTIONS

Case Insensitive Global

Run

Results (Matched text in red)
Regex: ([01]?\d)[-\/ .]([0123]?\d)[-\/ .](\d{4})
Found at position 13 in string:
the date was 10-28-2008

Figure 4-5:
This sample page, included with the tutorial files, lets you test out regular expressions using different methods—like Search or Match—and try different options such as case-insensitive or global searches.

Numbers

Numbers are an important part of programming. They let you perform tasks like calculating a total sales cost, determining the distance between two points, or simulating the roll of a die by generating a random number from 1 to 6. JavaScript gives you many different ways of working with numbers.

Changing a String to a Number

When you create a variable, you can store a number in it like this:

```
var a = 3.25;
```

However, there are times when a number is actually a string. For example, if you use the *prompt( )* method (page 55) to get visitor input, even if someone types 3.25, you'll end up with a string that contains a number. In other words, the result will be '3.25' (a string) and not 3.25 (a number). Frequently, this method doesn't cause a problem, since the JavaScript interpreter usually converts a string to a number when it seems like a number is called for. For example:

```
var a = '3';
var b = '4';
alert(a*b); // 12
```

In this example, even though the variables *a* and *b* are both strings, the JavaScript interpreter converts them to numbers to perform the multiplication (3×4) and return the result: 12.

However, when you use the + operator, the JavaScript interpreter won't make that conversion, and you can end up with some strange results:

```
var a = '3';
var b = '4';
alert(a+b); // 34
```

In this case, both *a* and *b* are strings; the + operator not only does mathematical addition, it also combines (concatenates) two strings together (see page 49). So instead of adding 3 and 4 to get 7, in this example, you end up with two strings fused together: 34.

When you need to convert a string to a number, JavaScript provides several ways:

• *Number( )* converts whatever string is passed to it into a number, like this:

```
var a = '3';
a = Number(a); // a is now the number 3
```

  So the problem of adding two strings that contain numbers could be fixed like this:

```
var a = '3';
var b = '4';
var total = Number(a) + Number(b); // 7
```

  A faster technique is the + operator, which does the same thing as *Number( )*. Just add a + in front of a variable containing a string, and the JavaScript interpreter converts the string to a number.

```
var a = '3';
var b = '4';
var total = +a + +b // 7
```

  The downside of either of these two techniques is that if the string contains anything except numbers, a single period or a + or – sign at the beginning of the string, you'll end up with a nonnumber, or the JavaScript value *NaN*, which means "not a number" (see page 50).

• *parseInt( )* tries to convert a string to a number as well. However, unlike *Number( )*, *parseInt( )* will try to change even a string with letters to a number, as long as the string begins with numbers. This command can come in handy when you get a string like '20 years' as the response to a question about someone's age:

```
var age = '20 years';
age = parseInt(age,10); //20
```

The *parseInt( )* method looks for either a number or a + or – sign at the beginning of the string and continues to look for numbers until it encounters a non-number. So in the above example, it returns the number 20 and ignores the other part of the string, ' years'.

Note: You're probably wondering what the 10 is doing in *parseInt(age,10);*. JavaScript can handle Octal numbers (which are based on 8 different digits 0-7, unlike decimal numbers which are based on 10 different digits 0-9); when you add the *,10* to *parseInt()*, you're telling the JavaScript interpreter to treat whatever the input is as a decimal number. That way, JavaScript correctly interprets a string like '08' in a prompt window or form field—decimally. For example, in this code *age* would be equal to 0:

```
var age = '08 years';
age = parseInt(age);
```

However, in the following code the variable *age* would hold the value 8:

```
var age = '08 years';
age = parseInt(age,10);
```

In other words, always add the *,10* when using the *parseInt()* method.

This method is also helpful when dealing with CSS units. For example, if you want to find the width of an element on a page (you'll learn how to do that on page 186), you often end up with a string like this: '200px' (meaning 200 pixels wide). Using the *parseInt( )* method, you can retrieve just the number value and then perform some operation on that value.

- *parseFloat( )* is like *parseInt( ),* but you use it when a string might contain a decimal point. For example, if you have a string like '4.5 acres' you can use *parseFloat( )* to retrieve the entire value including decimal places:

  ```
  var space = '4.5 acres';
  space = parseFloat(space); // 4.5
  ```

  If you used *parseInt( )* for the above example, you'd end up with just the number 4, since *parseInt( )* only tries to retrieve whole numbers (integers).

Which of the above methods you use depends on the situation: If your goal is to add two numbers, but they're strings, then use *Number( )* or + operator. However, if you want to extract a number from a string that might include letters, like '200px' or '1.5em', then use *parseInt( )* to capture whole numbers (200, for example) or *parseFloat( )* to capture numbers with decimals (1.5, for example).

Testing for Numbers

When using JavaScript to manipulate user input, you often need to verify that the information supplied by the visitor is of the correct type. For example, if you ask for people's years of birth, you want to make sure they supply a number. Likewise, when you're performing a mathematical calculation, if the data you use for the calculation isn't a number, then your script might break.

To verify that a string is a number, use the *isNaN( )* method. This method takes a string as an argument and tests whether the string is "not a number." If the string contains anything except a plus or minus (for positive and negative numbers) followed by numbers and an optional decimal value, it's considered "not a number," so the string '-23.25' is a number, but the string '24 pixels' is not. This method returns either *true* (if the string is not a number) or *false* (if it is a number). You can use *isNaN( )* as part of a conditional statement like this:

```
var x = '10'; // is a number
if (isNaN(x)) {
  // is NOT a number
} else {
  // it is a number
}
```

Rounding Numbers

JavaScript provides a way to round a fractional number to an integer—for example, rounding 4.5 up to 5. Rounding comes in handy when you're performing a calculation that must result in a whole number. For example, say you're using JavaScript to dynamically set a pixel height of a <div> tag on the page based on the height of the browser window. In other words, the height of the <div> is calculated using the window's height. Any calculation you make might result in a decimal value (like 300.25), but since there's no such thing as .25 pixels, you need to round the final calculation to the nearest integer (300, for example).

You can round a number using the *round( )* method of the Math object. The syntax for this looks a little unusual:

```
Math.round(number)
```

You pass a number (or variable containing a number) to the *round( )* method, and it returns an integer. If the original number has a decimal place with a value below .5, the number is rounded down; if the decimal place is .5 or above, it is rounded up. For example, 4.4 would round down to 4, while 4.5 rounds up to 5.

```
var decimalNum = 10.25;
var roundedNum = Math.round(decimalNum); // 10
```

Note: JavaScript provides two other methods for rounding numbers *Math.ceil()* and *Math.floor()*. You use them just like *Math.round()*, but *Math.ceil()* always rounds the number up (for example, *Math.ceil(4.0001)* returns 5), while *Math.floor()* always rounds the number down: *Math.floor(4.99999)* returns 4. To keep these two methods clear in your mind, think a *ceil*ing is up, and a *floor* is down.

Formatting Currency Values

When calculating product costs or shopping cart totals, you'll usually include the cost, plus two decimals out, like this: 9.99. But even if the monetary value is a whole number, it's common to add two zeros, like this: 10.00. And a currency value like 8.9 is written as 8.90. Unfortunately, JavaScript doesn't see numbers that way: it leaves the trailing zeros off (10 instead of 10.00, and 8.9 instead of 8.90, for example).

Fortunately, there's a method for numbers called *toFixed( ),* which lets you convert a number to a string that matches the number of decimal places you want. To use it, add a period after a number (or after the name of a variable containing a number), followed by *toFixed(2)*:

```
var cost = 10;
var printCost = '$' + cost.toFixed(2); // $10.00
```

The number you pass the *toFixed( )* method determines how many decimal places to go out to. For currency, use 2 to end up with numbers like 10.00 or 9.90; if you use 3, you end up with 3 decimal places, like 10.000 or 9.900.

If the number starts off with more decimal places than you specify, the number is rounded to the number of decimal places specified. For example:

```
var cost = 10.289;
var printCost = '$' + cost.toFixed(2); // $10.29
```

In this case, the 10.289 is rounded up to 10.29.

Note: The *toFixed()* method only works with numbers. So if you use a string, you end up with an error:

```
var cost='10';//a string
var printCost='$' + cost.toFixed(2);//error
```

To get around this problem, you need to convert the string to a number as described on page 134, like this:

```
var cost='10';//a string
cost = +cost;
var printCost='$' + cost.toFixed(2);//$10.00
```

Creating a Random Number

Random numbers can help add variety to a program. For example, say you have an array of questions for a quiz program (like Script 3.3 on page 106). Instead of asking the same questions in the same order each time, you can randomly select one question in the array. Or, you could use JavaScript to randomly select the name of a graphic file from an array and display a different image each time the page loads. Both of these tasks require a random number.

JavaScript provides the *Math.random( )* method for creating random numbers. This method returns a randomly generated number between 0 and 1 (for example .9716907176080688 or .10345038010895868). While you might not have much need for numbers like those, you can use some simple math operations to generate a whole number from 0 to another number. For example, to generate a number from 0 to 9, you'd use this code:

```
Math.floor(Math.random( )*10);
```

This code breaks down into two parts. The part inside the *Math.floor( )* method—*Math.random( )*10*—generates a random number between 0 and 10. That will generate numbers like 4.190788392268892; and since the random number is *between* 0 and 10, it never *is* 10. To get a whole number, the random result is passed to the *Math.floor( )* method, which rounds any decimal number down to the nearest whole number, so 3.4448588848 becomes 3 and .1111939498984 becomes 0.

If you want to get a random number between 1 and another number, just multiply the *random( )* method (case issue) by the uppermost number and add one to the total. For example, if you want to simulate a die roll to get a number from 1 to 6:

```
var roll = Math.floor(Math.random( )*6 +1); // 1,2,3,4,5 or 6
```

Randomly selecting an array element

You can use the *Math.random( )* method to randomly select an item from an array. As discussed on page 59, each item in an array is accessed using an index number. The first item in an array uses an index value of 0, and the last item in the array is accessed with an index number that's 1 minus the total number of items in the array. Using the *Math.random( )* method makes it really easy to randomly select an array item:

```
var people = ['Ron','Sally','Tricia','Bob']; //create an array
var random = Math.floor(Math.random( ) * people.length);
var rndPerson = people[random]; //
```

The first line of this code creates an array with four names. The second line does two things: First, it generates a random number between 0 and the number of items in the array (*people.length*)—in this example, a number *between* 0 and 4. Then it uses the *Math.floor( )* method to round down to the nearest integer, so it will produce the number 0, 1, 2, or 3. Finally, it uses that number to access one element from the array and store it in a variable named *rndPerson*.

A function for selecting a random number

Functions are a great way to create useful, reusable snippets of code (page 97). If you use random numbers frequently, you might want a simple function to help you select a random number between any two numbers—for example, a number between 1 and 6, or 100 and 1,000. The following function is called using two arguments; The first is the bottom possible value (1 for example), and the second is the largest possible value (6 for example):

```
function rndNum(from, to) {
  return Math.floor((Math.random( )*(to - from + 1)) + from);
}
```

To use this function, add it to your Web page (as described on page 98), and then call it like this:

```
var dieRoll = rndNum(1,6); // get a number between 1 and 6
```

Dates and Times

If you want to keep track of the current date or time, turn to JavaScript's *Date* object. This special JavaScript object lets you determine the year, month, day of the week, hour, and more. To use it, you create a variable and store a new *Date* object inside it like this:

```
var now = new Date( );
```

The *new Date( )* command creates a *Date* object containing the current date and time. Once created, you can access different pieces of time and date information using various date-related methods as listed in Table 4-5. For example, to get the current year use the *getFullYear( )* method like this:

```
var now = new Date( );
var year = now.getFullYear( );
```

Note: *new Date()* retrieves the current time and date as determined by each visitor's computer. In other words, if someone hasn't correctly set their computer's clock, then the date and time won't be accurate.

Table 4-5. *Methods for accessing parts of the Date object*

Method	What it returns
`getFullYear( )`	The year: 2008, for example.
`getMonth( )`	The month as an integer between 0 and 11: 0 is January and 11 is December.
`getDate( )`	The day of the month—a number between 1 and 31.
`getDay( )`	The day of the week as a number between 0 and 6. 0 is Sunday, and 6 is Saturday.
`getHours( )`	Number of hours on a 24-hour clock (i.e. a number between 0 and 23). For example, 11p.m. is 23.
`getMinutes( )`	Number of minutes between 0 and 59.
`getSeconds( )`	Number of seconds between 0 and 59.
`getTime( )`	Total number of milliseconds since January 1, 1970 at midnight (see box on page 142).

Getting the Month

To retrieve the month for a *Date* object, use the *getMonth( )* method, which returns the month's number:

```
var now = new Date( );
var month = now.getMonth( );
```

However, instead of returning a number that makes sense to us humans (as in 1 meaning January), this method returns a number that's one less. For example, January is 0, February is 1, and so on. If you want to retrieve a number that matches how we think of months, just add 1 like this:

```
var now = new Date( );
var month = now.getMonth( )+1;//matches the real month
```

There's no built-in JavaScript command that tells you the name of a month. Fortunately, JavaScript's strange way of numbering months comes in handy when you want to determine the actual name of the month. You can accomplish that by first creating an array of month names, then accessing a name using the index number for that month:

```
var months = ['January','February','March','April','May',
              'June','July','August','September',
              'October','November','December'];
var now = new Date( );
var month = months[now.getMonth( )];
```

The first line creates an array with all twelve month names, in the order they occur (January–December). Remember that to access an array item you use an index number, and that arrays are numbered starting with 0 (see page 59). So to access the first item of the array *months*, you use *months[0]*. So, by using the *getMonth( )* method, you can retrieve a number to use as an index for the *months* array and thus retrieve the name for that month.

Getting the Day of the Week

The *getDay( )* method retrieves the day of the week. And as with the *getMonth( )* method, the JavaScript interpreter returns a number that's one less than what you'd expect: 0 is considered Sunday, the first day of the week, while Saturday is 6. Since the name of the day of the week is usually more useful for your visitors, you can use an array to store the day names and use the *getDay( )* method to access the particular day in the array, like this:

```
var days = ['Sunday','Monday','Tuesday','Wednesday',
            'Thursday','Friday','Saturday'];
var now = new Date( );
var dayOfWeek = days[now.getDay( )];
```

In the tutorial on page 146, you'll see use both the *getDay( )* and *getMonth( )* techniques to create a useful function for creating a human-readable date.

Getting the Time

The *Date* object also contains the current time, so you can display the current time on a Web page or use the time to determine if the visitor is viewing the page in the a.m. or p.m. You can then do something with that information, like display a background image of the sun during the day, or the moon at night.

POWER USERS' CLINIC

The Date Object Behind the Scenes

JavaScript lets you access particular elements of the *Date* object, such as the year or the day of the month. However, the JavaScript interpreter actually thinks of a date as the number of *milliseconds* that have passed since midnight on January 1, 1970. For example, Wednesday, July 2, 2008 is actually 1214982000000 to the JavaScript interpreter.

That isn't a joke: As far as JavaScript is concerned, the beginning of time was January 1, 1970. That date (called the "Unix epoch") was arbitrarily chosen in the 70s by programmers creating the Unix operating system, so they could all agree on a way of keeping track of time. Since then, this way of tracking a date has become common in many programming languages and platforms.

Whenever you use a *Date* method like *getFullYear()*, the JavaScript interpreter does the math to figure out (based on how many seconds have elapsed since January 1, 1970) what year it is. If you want to see the number of milliseconds for a particular date, you use the *getTime()* method:

```
var sometime = new Date( );
var msElapsed = sometime.getTime( );
```

Tracking dates and times as milliseconds makes it easier to calculate differences between dates. For example, you can determine the amount of time until next New Year's day by first getting the number of milliseconds that will have elapsed from 1/1/1970 to when next year rolls around and then subtracting the number of milliseconds that have elapsed from 1/1/1970 to today:

```
// milliseconds from 1/1/1970 to today
var today = new Date( );
// milliseconds from 1/1/1970 to next new
year
var nextYear = new Date(2009, 0, 1);
// calculate milliseconds from today to
next year
var  timeDiff = nextYear - today;
```

The result of subtracting two dates is the number of milliseconds difference between the two. If you want to convert that into something useful, just divide it by the number of milliseconds in a day (to determine how many days) or the number of milliseconds in an hour (to determine how many hours), and so on.

```
var second = 1000; // 1000 milliseconds in
a second
var minute = 60*second; // 60 seconds in a
minute
var hour = 60*minute; // 60 minutes in an
hour
var day = 24*hour; // 24 hours in a day
var totalDays = timeDiff/day; // total
number of days
```

(In this example, you may have noticed a different way to create a date: *new Date(2009,0,1)*. You can read more about this method on page 145.)

You can use the *getHours( )*, *getMinutes( )*, and *getSeconds( )* methods to get the hours, minutes, and seconds. So to display the time on a Web page, add the following in the HTML where you wish the time to appear:

```
var now = new Date( );
var hours = now.getHours( );
var minutes = now.getMinutes( );
var seconds = now.getSeconds( );
document.write(hours + ":" + minutes + ":" + seconds);
```

This code produces output like 6:35:56 to indicate 6 a.m., 35 minutes, and 56 seconds. However, it will also produce output that you might not like, like 18:4:9 to indicate 4 minutes and 9 seconds after 6 p.m. One problem is that most people reading this book, unless they're in the military, don't use the 24-hour clock. They don't recognize 18 as meaning 6 p.m. An even bigger problem is that times should be formatted with two digits for minutes and seconds (even if they're a number less than 10), like this: 6:04:09. Fortunately, it's not difficult to adjust the above script to match those requirements.

Changing hours to a.m. and p.m.

To change hours from a 24-hour clock to a 12-hour clock, you need to do a couple of things. First, you need to determine if the time is in the morning (so you can add 'am' after the time) or in the afternoon (to append 'pm'). Second, you need to convert any hours greater than 12 to their 12-hour clock equivalent (for example, change 14 to 2 p.m.).

Here's the code to do that:

```
1    var now = new Date();
2    var hour = now.getHours();
3    if (hour < 12) {
4      meridiem = 'am';
5    } else {
6      meridiem = 'pm';
7    }
8    hour = hour % 12;
9    if (hour==0) {
10     hour = 12;
11   }
12   hour = hour + ' ' + meridiem;
```

Note: The column of numbers at the far left is just line numbering to make it easier for you to follow the discussion below. Don't type these numbers into your own code!

Lines 1 and 2 grab the current date and time and store the current hour into a variable named *hour*. Lines 3–7 determine if the hour is in the afternoon or morning; if the hour is less than 12 (the hour after midnight is 0), then it's the morning (a.m.); otherwise, it's the afternoon (p.m.).

Line 8 introduces a mathematical operator called *modulus* and represented by a percent (%) sign. It returns the remainder of a division operation. For example, 2 divides into 5 two times (2×2 is 4), with 1 left over. In other words, 5 % 2 is 1. So in this case, if the hour is 18, 18 % 12 results in 6 (12 goes into 18 once with a remainder of 6). 18 is 6 p.m., which is what you want. If the first number is smaller than the number divided into it (for example, 8 divided by 12), then the result is the original number. For example, 8 % 12 just returns 8; in other words, the modulus operator doesn't change the hours before noon.

Lines 9–11 take care of two possible outcomes with the modulus operator. If the hour is 12 (noon) or 0 (after midnight), then the modulus operator returns 0. In this case, *hour* is just set to 12 for either 12 p.m. or 12 a.m.

Finally, line 12 combines the reformatted hour with a space and either “am” or “pm”, so the result is displayed as, for example, “6 am” or “6 pm”.

Padding single digits

As discussed on page 142, when the minutes or seconds values are less than 10, you can end up with weird output like 7:3:2 p.m. To change this output to the more common 7:03:02 p.m., you need to add a 0 in front of the single digit. It’s easy with a basic conditional statement:

```
1    var minutes = now.getMinutes( );
2    if (minutes<10) {
3      minutes = '0' + minutes;
4    }
```

Line 1 grabs the minutes in the current time, which in this example could be 33 or 3. Line 2 simply checks if the number is less than 10, meaning the minute is a single digit and needs a 0 in front of it. Line 3 is a bit tricky, since you can’t normally add a 0 in front of a number: 0 + 2 equals 2, not 02. However, you can combine strings in this way so *'0' + minutes* means combine the string '0' with the value in the *minutes* variable. As discussed on page 50, when you add a string to a number, the JavaScript interpreter converts the number to a string as well, so you end up with a string like '08'.

You can put all of these parts together to create a simple function to output times in formats like 7:32:04 p.m., or 4:02:34 a.m., or even leave off seconds altogether for a time like 7:23 p.m.:

```
function printTime(secs) {
    var sep = ':'; //seperator character
    var hours,minutes,seconds,time;
    var now = new Date( );
    hours = now.getHours( );
    if (hours < 12) {
        meridiem = 'am';
    } else {
        meridiem = 'pm';
    }
    hours = hours % 12;
    if (hours==0) {
        hours = 12;
    }
```

```
        time = hours;
        minutes = now.getMinutes( );
        if (minutes<10) {
            minutes = '0' + minutes;
        }
        time += sep + minutes;
        if (secs) {
            seconds = now.getSeconds( );
            if (seconds<10) {
                seconds = '0' + seconds;
            }
            time += sep + seconds;
        }
        return time + ' ' + meridiem;
    }
```

You'll find this function in the file *printTime.js* in the *chapter04* folder in the Tutorials. You can see it in action by opening the file *time.html* (in that same folder) in a Web browser. To use the function, either attach the *printTime.js* file to a Web page (see page 23), or copy the function into a Web page or another external JavaScript file (page 22). To get the time, just call the function like this: *printTime( )*, or, if you want the seconds displayed as well, *printTime(true)*. The function will return a string containing the current time in the proper format.

Creating a Date Other Than Today

So far, you've seen how to use *new Date( )* to capture the current date and time on a visitor's computer. But what if you want to create a *Date* object for next Thanksgiving or New Year's? JavaScript lets you create a date other than today in a few different ways. You might want to do this if you'd like to do a calculation between two dates: for example, "How many days until the new year?" (Also see the box on page 142.)

When using the *Date( )* method, you can also specify a date and time in the future or past. The basic format is this:

```
new Date(year,month,day,hour,minutes,seconds,milliseconds);
```

For example, to create a *Date* for noon on New Year's Day 2010, you could do this:

```
var ny2010 = new Date(2010,0,1,12,0,0,0);
```

This code translates to "create a new Date object for January 1, 2010 at 12 o'clock, 0 minutes, 0 seconds, and 0 milliseconds." You must supply at least a year and month, but if you don't need to specify an exact time, you can leave off milliseconds, seconds, minutes, and so on. For example, to just create a date object for January 1, 2010, you could do this:

```
var ny2010 = new Date(2010,0,1);
```

Note: Remember that JavaScript uses 0 for January, 1 for February, and so on, as described on page 141.

Creating a date that's one week from today

As discussed in the box on page 142, the JavaScript interpreter actually treats a date as the number of milliseconds that have elapsed since Jan 1, 1970. Another way to create a date is to pass a value representing the number of milliseconds for that date:

```
new Date(milliseconds);
```

So another way to create a date for January 1, 2010 would be like this:

```
var ny2010 = new Date(1262332800000);
```

Of course, since most of us aren't human calculators, you probably wouldn't think of a date like this. However, milliseconds come in very handy when you're creating a new date that's a certain amount of time from another date. For example, when setting a cookie using JavaScript, you need to specify a date at which point that cookie is deleted from a visitor's browser. To make sure a cookie disappears after one week, you need to specify a date that's one week from today.

To create a date that's one week from now, you could do the following:

```
var now = new Date( ); // today
var nowMS = now.getTime( ); // get # milliseconds for today
var week = 1000*60*60*24*7; // milliseconds in one week
var oneWeekFromNow = new Date(nowMS + week);
```

The first line stores the current date and time in a variable named *now*. Next, the *getTime( )* method extracts the number of milliseconds that have elapsed from January 1, 1970 to today. The third line calculates the total number of milliseconds in a single week (1000 milliseconds * 60 seconds * 60 minutes * 24 hours * 7 days). Finally, the code creates a new date by adding the number of milliseconds in a week to today.

Tutorial

To wrap up this chapter, you'll create a useful function for outputting a date in several different human-friendly formats. The function will be flexible enough to let you print out a date, as in "January 1, 2009," "1/1/09," or "Monday, February 2, 2009." In addition, you'll use some of the date and string methods covered in this chapter to build this function.

Overview

As with any program you write, it's good to start with a clear picture of what you want to accomplish and the steps necessary to get it done. For this program, you want to output the date in many different formats, and you want the function to be easy to use.