CHAPTER 28

The Date Object

✦ ✦ ✦ ✦

In This Chapter

Working with date and time values in JavaScript

Performing date calculations

Validating date entry form fields

✦ ✦ ✦ ✦

Perhaps the most untapped power of JavaScript is its date and time handling. Scripters passed over the Date object with good cause in the early days of JavaScript, because in earlier versions of scriptable browsers, significant bugs and platform-specific anomalies made date and time programming hazardous without significant testing. Even with the improved bug situation, working with dates requires a working knowledge of the world's time zones and their relationships with the standard reference point, Greenwich mean time (GMT).

Now that date- and time-handling has improved in the latest browsers, I hope more scripters look into incorporating these kinds of calculations into their pages. In the bonus applications on the CD-ROM, I show you an application that lets your Web site highlight the areas that have been updated since each visitor's last surf ride through your pages.

Before getting to the JavaScript part of date discussions, however, I summarize key facts about time zones and their impact on scripting date and time on a browser. If you're not sure what GMT and UTC mean, the following section is for you.

Time Zones and GMT

By international agreement, the world is divided into distinct time zones that allow the inhabitants of each zone to say with confidence that when the Sun appears directly overhead, it is roughly noon, squarely in the middle of the day. The current time in the zone is what we set our clocks to — the local time.

That's fine when your entire existence and scope of life go no further than the width of your own time zone. But with instant communication among all parts of the world, your scope reaches well beyond local time. Periodically you must be aware of the local time in other zones. After all, if you live in New York, you don't want to wake up someone in Los Angeles before dawn with a phone call from your office.

Note

From here on I speak of the Sun "moving" as if Earth were the center of the solar system. I do so for the convenience of our daily perception of the Sun arcing across what appears to us as a stationary sky. In point of fact, I believe Copernicus, so delete that e-mail you were about to send me.

From the point of view of the time zone over which the Sun is positioned at any given instant, all time zones to the east have already had their noon, so it is later in the day for them — one hour later per time zone (except for those few time zones offset by fractions of an hour). That's why when U.S. television networks broadcast simultaneously to the eastern and central time zones, the announced schedule for a program is "10 eastern, 9 central."

Many international businesses must coordinate time schedules of far-flung events. Doing so and taking into account the numerous time zone differences (not to mention seasonal national variations such as daylight saving time) would be a nightmare. To help everyone out, a standard reference point was devised: the time zone running through the celestial observatory at Greenwich, England (pronounced GREN-itch). This time zone is called Greenwich mean time, or GMT for short. The "mean" part comes from the fact that on the exact opposite side of the globe (through the Pacific Ocean) is the international date line, another world standard that decrees where the first instance of the next calendar day appears on the planet. Thus, GMT is located at the middle, or mean, of the full circuit of the day. Not that many years ago, GMT was given another abbreviation that is not based on any one language of the planet. The abbreviation is UTC (pronounced as its letters: yu-tee-see), and the English version is Coordinated Universal Time. Whenever you see UTC, it is for all practical purposes the same as GMT.

If your personal computer's system clock is set correctly, the machine ticks away in GMT time. But because you set your local time zone in the appropriate control panel, all file time stamps and clock displays are in your local time. The machine knows what the offset time is between your local time and GMT. For daylight saving time, you may have to check a preference setting so that the offset is adjusted accordingly; in Windows 95, the operating system knows when the changeover occurs and prompts you if it's OK to change the offset. In any case, if you travel across time zones with a laptop, you should change the computer's time zone setting, not its clock.

JavaScript's inner handling of date and time works a lot like the PC clock (on which your programs rely). Date values that you generate in a script are stored internally in GMT time; however, almost all the displays and extracted values are in the local time of the visitor (not the Web site server). And remember that the date values are created on the visitor's machine by virtue of your script generating that value — you don't send "living" Date objects to the client from the server. This is perhaps the most difficult concept to grasp as you work with JavaScript date and time.

Whenever you program time and date in JavaScript for a public Web page, you must take the world view. This requires knowing that the visitor's computer settings determine the accuracy of the conversion between GMT time and local time. It also means you'll have to do some testing by changing your PC's clock to times in other parts of the world and making believe you are temporarily in those remote locations. This isn't always easy to do. It reminds me of the time I was visiting Sydney, Australia. I was turning in for the night and switched on the television in the hotel. This hotel received a live satellite relay of a long-running U.S. television program, *Today*. The program broadcast from New York was for the morning of the same day I was just finishing in Sydney. Yes, this time zone stuff can make your head hurt.

The Date Object

Borrowing heavily from the Java world, JavaScript includes a Date object (with a capital "D") and a large collection of methods to help you extract parts of dates and times, as well as assign dates and times. Understanding the object model for dates is vital to successful scripting of such data.

JavaScript's defining of a Date object should clue in experienced scripters that they're dealing with far more than a bunch of functions that dish out data from their desktop computer's clock and let them transform the values into various formats for calculation and display. By referring to a Date object, JavaScript may mislead you into thinking that you work with just one such object. Not at all. Any time you want to perform calculations or display dates or times, you create a new Date object in the browser's memory. This object is associated only with the current page, just like the array objects described in Chapter 29.

Another important point about the creation of a Date object is that it is not a ticking clock. Instead it is a snapshot of an exact millisecond in time, whether it be for the instant at which you generate the object or for a specific time in the past or future you need for calculations. If you need to have a live clock ticking away, your scripts will repeatedly create new Date objects to grab up-to-the-millisecond snapshots of your computer's clock.

In the process of creating a Date object, you assign that object to a variable name (behind the scenes, the variable is really just a pointer to the spot in memory where the Date object's data is stored). In other words, that variable becomes the Date object for further manipulation in your scripts: The variable becomes part of the dot-syntax-style reference to all possible methods for dates (Date objects have no event handlers).

One important point to remember is that despite its name, every Date object contains information about date and time. Therefore, even if you're concerned only about the date part of an object's data, time data is standing by as well. As you learn in a bit, the time element can catch you off-guard for some operations.

Creating a Date object

The statement that asks JavaScript to make an object for your script includes the special object construction keyword `new`. The basic syntax for generating a new Date object is as follows:

```
var dateObjectName = new Date([parameters])
```

The Date object evaluates to an object rather than to some string or numeric value.

With the Date object's reference safely tucked away in the variable name, you access all date-oriented methods in the dot-syntax fashion with which you're already familiar:

```
var result = dateObjectName.method()
```

With variables such as `result`, your scripts perform calculations or displays of the Date object's data (some methods extract pieces of the date and time data from the object). If you then want to put some new value into the Date object (such as adding a year to the Date object), you assign the new value to the object by way of the method that lets you set the value:

```
dateObjectName.method(newValue)
```

This example doesn't look like the typical JavaScript assignment statement, which has an equals sign operator. But this statement is the way in which methods that set Date object data work.

You cannot get very far into scripting dates without digging into time zone arithmetic. Although JavaScript may render the string equivalent of a Date object in your local time zone, the internal storage is strictly GMT.

Even though I haven't yet introduced you to details of the Date object's methods, I use two of them to demonstrate adding one year to today's date.

```
oneDate = new Date() // creates object with current GMT date
theYear = oneDate.getYear() // theYear is now storing the value 98
theYear = theYear + 1 // theYear now is 99
oneDate.setYear(theYear) // new year value now in the object
```

At the end of this sequence, the `oneDate` object automatically adjusts all the other settings that it knows has today's date for the next year. The day of the week, for example, will be different, and JavaScript takes care of that for you, should you need to extract that data. With next year's data in the `oneDate` object, you may now want to extract that new date as a string value for display in a field on the page or submit it quietly to a CGI program on the server.

The issue of parameters for creating a new Date object is a bit complex, mostly because of the flexibility that JavaScript offers the scripter. Recall that the job of the `new Date()` statement is to create a place in memory for all data that a date needs to store. What is missing from that task is the data — what date and time to enter into that memory spot. That's where the parameters come in.

If you leave the parameters empty, JavaScript takes that to mean you want today's date and the current time to be assigned to that new Date object. JavaScript isn't any smarter, of course, than the setting of the internal clock of your page visitor's personal computer. If the clock isn't correct, JavaScript won't do any better of a job identifying the date and time.

Caution

Remember that when you create a new Date object, it contains the current time as well. The fact that the current date may include a time of 16:03:19 (in 24-hour time) may throw off things such as days-between-dates calculations. Be careful.

To create a Date object for a specific date or time, you have five ways to send values as a parameter to the `new Date()` constructor function:

```
new Date("Month dd, yyyy hh:mm:ss")
new Date("Month dd, yyyy")
new Date(yy,mm,dd,hh,mm,ss)
new Date(yy,mm,dd)

new Date(milliseconds)
```

These five schemes break down into two styles — a long string versus a comma-delimited list of data — each with optional time settings. If you omit time settings, they are set to 0 (midnight) in the Date object for whatever date you entered. You cannot omit date values from the parameters — every Date object must have a real date attached to it, whether you need it or not.

In the long string version, the month is spelled out in full in English. No abbreviations are allowed. The rest of the data is filled with numbers representing the date, year, hours, minutes, and seconds. For single-digit values, you can use either a one- or two-digit version (such as 4:05:00). Hours, minutes, and seconds are separated by colons.

The short version is strictly a nonquoted list of integer values in the order indicated. JavaScript cannot know that a 30 means the date when you accidentally place it in the month slot.

Date prototype property

Like a number of JavaScript objects, the Date object has a `prototype` property, which enables you to apply new properties and methods to every Date object created in the current page. You can see examples of how this works in discussions of the `prototype` property for string and array objects (Chapters 26 and 29, respectively).

Date methods

The bulk of a Date object's methods are for extracting parts of the date and time information and for changing the date and time stored in the object. These two sets of methods are easily identifiable because they all begin with the word "get" or "set." Table 28-1 lists all of the Date object's methods. Your initial focus will be on the first sixteen methods, which deal with components of the Date object's data.

Table 28-1
Date Object Methods

Method	*Value Range*	*Description*
`dateObj.getTime()`	0-…	Milliseconds since 1/1/70 00:00:00 GMT
`dateObj.getYear()`	70-…	Specified year minus 1900
`dateObj.getMonth()`	0-11	Month within the year (January = 0)
`dateObj.getDate()`	1-31	Date within the month
`dateObj.getDay()`	0-6	Day of week (Sunday = 0)
`dateObj.getHours()`	0-23	Hour of the day in 24-hour time
`dateObj.getMinutes()`	0-59	Minute of the specified hour
`dateObj.getSeconds()`	0-59	Second within the specified minute
`dateObj.setTime(`*`val`*`)`	0-…	Milliseconds since 1/1/70 00:00:00 GMT
`dateObj.setYear(`*`val`*`)`	70-…	Specified year minus 1900
`dateObj.setMonth(`*`val`*`)`	0-11	Month within the year (January = 0)

(continued)

Table 28-1 *(continued)*

Method	*Value Range*	*Description*
`dateObj.setDate(val)`	1-31	Date within the month
`dateObj.setDay(val)`	0-6	Day of week (Sunday = 0)
`dateObj.setHours(val)`	0-23	Hour of the day in 24-hour time
`dateObj.setMinutes(val)`	0-59	Minute of the specified hour
`dateObj.setSeconds(val)`	0-59	Second within the specified minute
`dateObj.getTimezoneOffset()`	0-...	Minutes offset from GMT/UTC
`dateObj.toGMTString()`		Date string in universal format
`dateObj.toLocaleString()`		Date string in your system's format
`Date.parse("dateString")`		Converts string date to milliseconds
`Date.UTC(date values)`		Generates a date value from GMT values

JavaScript maintains its date information in the form of a count of milliseconds (thousandths of a second) starting from January 1, 1970, in the GMT time zone. Dates before that starting point are stored as negative values (but see the section on bugs and gremlins later in this chapter). Regardless of the country you live in or the date and time formats specified for your computer, the millisecond is the JavaScript universal measure of time. Any calculations that involve adding or subtracting times and dates should be performed in the millisecond values to ensure accuracy. Therefore, though you may never display the milliseconds value in a field or dialog box, your scripts will probably work with them from time to time in variables. To derive the millisecond equivalent for any date and time stored in a Date object, use the `dateObj.getTime()` method, as in

```
startDate = new Date()
started = startDate.getTime()
```

Although the method has the word "time" in its name, the fact that the value is the total number of milliseconds from January 1, 1970, means the value also conveys a date.

Other Date object `get` methods extract a specific element of the object. You have to exercise some care here, because some values begin counting with 0 when you may not expect it. For example, January is month 0 in JavaScript's scheme; December is month 11. Hours, minutes, and seconds all begin with 0, which, in the end, is logical. Calendar dates, however, use the actual number that would show up on the wall calendar: The first day of the month is date value 1. For the twentieth century years, the year value is whatever the actual year number is, minus 1900. For 1996, that means the year value is 96. But for years before 1900 and after 1999, JavaScript uses a different formula, showing the full year value. This means you have to check whether a year value is less than 100 and add 1900 to it before displaying that year:

```
var today = new Date()
var thisYear = today.getYear()
if (thisYear < 100) {
        thisYear += 1900
}
```

This assumes, of course, you won't be working with years before A.D. 100. This could cause terrible confusion.

To adjust any one of the elements of a date value, use the corresponding `set` method in an assignment statement. If the new value forces the adjustment of other elements, JavaScript takes care of that. For example, consider the following sequence and how some values are changed for us:

```
myBirthday = new Date("September 11, 1996")
result = myBirthday.getDay() // result = 3, a Wednesday
myBirthday.setYear(97) // bump up to next year
result = myBirthday.getDay() // result = 4, a Thursday
```

Because the same date in the following year is on a different day, JavaScript tracks that for you.

Accommodating time zones

Understanding the `dateObj.getTimezoneOffset()` method involves both your operating system's time control panel setting and an internationally recognized (in computerdom, anyway) format for representing dates and times. If you have ignored the control panel stuff about setting your local time zone, the values you get for this property may be off for most dates and times. In the eastern part of North America, for instance, the eastern standard time zone is five hours earlier than Greenwich mean time. With the `getTimezoneOffset()` method producing a value of minutes' difference between GMT and the PC's time zone, the five hours difference of eastern standard time is rendered as a value of 300 minutes. On the Windows platform, the value automatically changes to reflect changes in daylight saving time in the user's area (if applicable). Offsets to the east of GMT (to the date line) are expressed as negative values.

Dates as strings

When you generate a Date object, JavaScript automatically applies the `toString()` method to the object if you attempt to display that date either in a page or alert box. The format of this string varies with browser and operating system platform. For example, in Navigator 4 for Windows 95, the string is in the following format:

```
Thu Aug 28 11:43:34 Pacific Daylight Time 1997
```

But in the same version for Macintosh, the string is

```
Thu Aug 28 11:43:34 1997
```

Internet Explorer tends to follow the latter format. The point is not to rely on a specific format and character location of this string for the components of dates. Use the Date object methods to extract Date object components.

JavaScript does, however, provide two methods that return the Date object in more constant string formats. One, `dateObj.toGMTString()`, converts the date and time to the GMT equivalent on the way to the variable you use to store the extracted data. Here is what such data looks like:

```
Wed, 07 Aug 1996 03:25:28 GMT
```

If you're not familiar with the workings of GMT and how such conversions can present unexpected dates, you should exercise great care in testing your application. Eight o'clock on a Friday evening in California in the winter is four o'clock on Saturday morning GMT.

If time zone conversions make your head hurt, you can use the second string method, `dateObj.toLocaleString()`. In Navigator 3 for North American Windows users, the returned value looks like this:

```
08/06/96 20:25:28
```

Friendly date formats

What neither of the two string conversion methods address, however, is a way for the scripter to easily extract string segments to assemble custom date strings. For example, you cannot derive any data directly from a Date object that lets you display the object as

```
Friday, August 9, 1996
```

To accomplish this kind of string generation, you have to create your own functions. Listing 28-1 demonstrates one method of creating this kind of string from a Date object (in a form compatible with Navigator 2 and Internet Explorer 3 arrays).

Listing 28-1: **Creating a Friendly Date String**

```
<HTML>
<HEAD>
<TITLE>Date String Maker</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function MakeArray(n) {
        this.length = n
        return this
}
monthNames = new MakeArray(12)
monthNames[1] = "January"
monthNames[2] = "February"
monthNames[3] = "March"
monthNames[4] = "April"
monthNames[5] = "May"
monthNames[6] = "June"
monthNames[7] = "July"
monthNames[8] = "August"
monthNames[9] = "September"
monthNames[10] = "October"
```

```
monthNames[11] = "November"
monthNames[12] = "December"

dayNames = new MakeArray(7)
dayNames[1] = "Sunday"
dayNames[2] = "Monday"
dayNames[3] = "Tuesday"
dayNames[4] = "Wednesday"
dayNames[5] = "Thursday"
dayNames[6] = "Friday"
dayNames[7] = "Saturday"

function customDateString(oneDate) {
        var theDay = dayNames[oneDate.getDay() + 1]
        var theMonth = monthNames[oneDate.getMonth() + 1]
        var theYear = oneDate.getYear() + 1900
        return theDay + ", " + theMonth + " " + oneDate.getDate() +
", " + theYear
}
</SCRIPT>
</HEAD>

<BODY>
<H1> Welcome!</H1>
<SCRIPT LANGUAGE="JavaScript">
document.write(customDateString(new Date()))
</SCRIPT>

<HR>
</BODY>
</HTML>
```

Assuming the user has the PC's clock set correctly (a big assumption), the date appearing just below the opening headline is the current date — making it appear as though the document had been updated today.

More conversions

The final two methods related to Date objects are in the category known as *static methods*. Unlike all other methods, these two do not act on Date objects. Rather, they convert dates from string or numeric forms into millisecond values of those dates. The primary beneficiary of these actions is the `dateObj.setTime()` method, which requires a millisecond measure of a date as a parameter. This is the method you would use to throw an entirely different date into an existing Date object.

`Date.parse()` accepts date strings similar to the ones you've seen in this section, including the internationally approved version. `Date.UTC()`, on the other hand, requires the comma-delimited list of values (in proper order: yy,mm,dd,hh,mm,ss) in the GMT zone. Because setting all other properties via the Date object's methods interpret those values to mean the computer's local time zone, the `Date.UTC()` method gives you a way to hard-code a GMT time. Here is an example that creates a new Date object for 6 p.m. on March 4, 1996, GMT:

```
newObj = new Date(Date.UTC(96,2,4,18,0,0))
result = newObj.toString() // result = "Tue, Mar 04 10:00:00 Pacific
Standard Time 1996"
```

Because I then convert the object to a string, the local time is what comes as a result: the Pacific standard time zone equivalent of the GMT time entered.

New methods

The ECMA-262 language standard defines additional Date object methods to make it easier to set and get object components in GMT time. These methods are in Internet Explorer 4, but only in the Windows version of Navigator 4 as this is being written. Eventually all browsers will have the full set.

These new methods are similar to the component-oriented ones in the language since the beginning. The difference is that "UTC" is made part of the method name. For example, to the pair of `dateObj.getTime()` and `dateObj.setTime()` methods are added `dateObj.getUTCTime()` and `dateObj.setUTCTime()`. These UTC versions will be available for getting and setting all components.

Another improvement is the addition of `dateObj.getFullYear()` and `dateObj.setFullYear()`. These two new methods provide a workaround for the messy year situation currently in force — that twentieth-century year values are two-digit values. With the new methods, all years are treated with the proper number of digits for the year's full representation.

Date and time arithmetic

You may need to perform some math with dates for any number of reasons. Perhaps you need to calculate a date at some fixed number of days or weeks in the future or figure out the number of days between two dates. When calculations of these types are required, remember the *lingua franca* of JavaScript date values: the milliseconds.

What you may need to do in your date-intensive scripts is establish some variable values representing the number of milliseconds for minutes, hours, days, or weeks, and then use those variables in your calculations. Here is an example that establishes some practical variable values, building on each other:

```
var oneMinute = 60 * 1000
var oneHour = oneMinute * 60
var oneDay = oneHour * 24
var oneWeek = oneDay * 7
```

With these values established in a script, I can use one to calculate the date one week from today:

```
targetDate = new Date()
dateInMs = targetDate.getTime()
dateInMs += oneWeek
targetDate.setTime(dateInMs)
```

In another example, I use components of a Date object to assist in deciding what kind of greeting message to place in a document, based on the local time of the user's PC clock. Listing 28-2 adds to the scripting from Listing 28-1, bringing some quasi-intelligence to the proceedings. Again, this script uses the older array creation mechanism to be compatible with Navigator 2 and Internet Explorer 3.

Listing 28-2: **A Dynamic Welcome Message**

```
<HTML>
<HEAD>
<TITLE>Date String Maker</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function MakeArray(n) {
       this.length = n
       return this
}
monthNames = new MakeArray(12)
monthNames[1] = "January"
monthNames[2] = "February"
monthNames[3] = "March"
monthNames[4] = "April"
monthNames[5] = "May"
monthNames[6] = "June"
monthNames[7] = "July"
monthNames[8] = "August"
monthNames[9] = "September"
monthNames[10] = "October"
monthNames[11] = "November"
monthNames[12] = "December"
dayNames = new MakeArray(7)
dayNames[1] = "Sunday"
dayNames[2] = "Monday"
dayNames[3] = "Tuesday"
dayNames[4] = "Wednesday"
dayNames[5] = "Thursday"
dayNames[6] = "Friday"
dayNames[7] = "Saturday"

function customDateString(oneDate) {
       var theDay = dayNames[oneDate.getDay() + 1]
       var theMonth = monthNames[oneDate.getMonth() + 1]
       var theYear = oneDate.getYear() + 1900
       return theDay + ", " + theMonth + " " + oneDate.getDate() +
", " + theYear
}
function dayPart(oneDate) {
       var theHour = oneDate.getHours()
       if (theHour <6 )
          return "wee hours"
       if (theHour < 12)
          return "morning"
       if (theHour < 18)
          return "afternoon"
       return "evening"
}
```

(continued)

Listing 28-2 *(continued)*

```
</SCRIPT>
</HEAD>

<BODY>
<H1> Welcome!</H1>
<SCRIPT LANGUAGE="JavaScript">
today = new Date()
var header = (customDateString(today)).italics()
header += "<BR>We hope you are enjoying the "
header += dayPart(today) + "."
document.write(header)
</SCRIPT>
<HR>
</BODY>
</HTML>
```

I've divided the day into four parts and presented a different greeting for each part of the day. The greeting that plays is based, simply enough, on the hour element of a Date object representing the time the page is loaded into the browser. Because this greeting is embedded in the page, the greeting does not change no matter how long the user stays logged on to the page.

Date bugs and gremlins

Each generation of Navigator improves the stability and reliability of scripted Date objects. Unfortunately, Navigator 2 has enough bugs and crash problems across many platforms to make scripting complex world-time applications for this browser impossible. The Macintosh version also has bugs that throw off dates by as much as a full day. I recommend avoiding Navigator 2 on all platforms for serious date and time scripting.

The situation is much improved for Navigator 3. Still, some bugs persist. One bug in particular affects Macintosh versions of Navigator. Whenever you create a new Date object with daylight saving time engaged in the Date and Time control panel, the browser automatically adds one hour to the object. See the time-based application in Chapter 52 of the bonus applications on the CD-ROM for an example of how to counteract the effects of typical time bugs. Also afflicting the Macintosh in Navigator 3 is a faulty calculation of the time zone offset for all time zones east of GMT. Instead of generating these values as negative numbers (getting lower and lower as you head east), the offset values increase continuously as you head west from Greenwich. While the Western Hemisphere is fine, the values continue to increase past the international date line, rather than switch over to the negative values.

Internet Explorer 3 isn't free of problems. It cannot handle dates before January 1, 1970 (GMT). Attempts to generate a date before that one results in that base date as the value. It also completely miscalculates the time zone offset, following the erroneous pattern of Navigator 2. Even Navigators 3 and 4 have problems with historic dates. You are asking for trouble if the date extends beyond January 1, A.D. 1. Internet Explorer 4, on the other hand, appears to sail very well into ancient history.

You should be aware of one more discrepancy between Mac and Windows versions of Navigator. In Windows, if you generate a Date object for a date in another part of the year, the browser sets the time zone offset for that object according to the time zone setting for that time of year. On the Mac, the current setting of the control panel governs whether the normal or daylight saving time offset is applied to the date, regardless of the actual date within the year. This affects Navigator 3 and 4, and can throw off calculations from other parts of the year by one hour.

It may sound as though the road to Date object scripting is filled with land mines. While date and time scripting is far from hassle free, you can put it to good use with careful planning and a lot of testing. Look to the example application on the CD-ROM for ideas on implementing date and time code into your pages.

Validating Date Entries in Forms

Given the bug horror stories in the previous section, you may wonder how you can ever perform data-entry validation for dates in forms. The problem is not in the calculations as it is in the wide variety of acceptable date formats around the world. No matter how well you instruct users to enter dates in a particular format, many will follow their own habits and conventions. Moreover, how can you know whether an entry of 03/04/98 is the North American March 4, 1998, or the European April 3, 1998? The answer is: You can't.

My recommendation is to divide a date field into three components: month, day, and year. Let the user enter values into each field, and validate each field individually for its valid range. Listing 28-3 shows an example of how this is done. The page includes a form that is to be validated before it is submitted. Each component field does its own range checking on the fly as the user enters values. But because this kind of validation can be defeated, the page includes one further check triggered by the form's `onSubmit=` event handler. If any field is out of whack, the form submission is canceled.

Listing 28-3: **Date Validation in a Form**

```
<HTML>
<HEAD>
<TITLE>Date Entry Validation</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
// **BEGIN GENERIC VALIDATION FUNCTIONS**
// general purpose function to see if an input value has been entered
at all
function isEmpty(inputStr) {
        if (inputStr == "" || inputStr == null) {
            return true
        }
        return false
}
```

(continued)

Listing 28-3 *(continued)*

```
// function to determine if value is in acceptable range for this
application
function inRange(inputStr, lo, hi) {
       var num = parseInt(inputStr, 10)
       if (num < lo || num > hi) {
          return false
       }
       return true
}
// **END GENERIC VALIDATION FUNCTIONS**

function validateMonth(field, bypassUpdate) {
       var input = parseInt(field.value, 10)
       if (isEmpty(input)) {
          alert("Be sure to enter a month value.")
          select(field)
          return false
       } else {
          if (isNaN(input)) {
             alert("Entries must be numbers only.")
             select(field)
             return false
          } else {
             if (!inRange(input,1,12)) {
                alert("Enter a number between 1 (January) and 12
(December).")
                select(field)
                return false
             }
          }
       }
       if (!bypassUpdate) {
          calcDate()
       }
       return true
}

function validateDate(field) {
       var input = parseInt(field.value, 10)
       if (isEmpty(input)) {
          alert("Be sure to enter a date value.")
          select(field)
          return false
       } else {
          if (isNaN(input)) {
             alert("Entries must be numbers only.")
             select(field)
             return false
          } else {
            var monthField = document.birthdate.month
            if (!validateMonth(monthField, true)) return false
            var monthVal = parseInt(monthField.value, 10)
            var monthMax = new
Array(31,31,29,31,30,31,30,31,31,30,31,30,31)
            var top = monthMax[monthVal]
            if (!inRange(input,1,top)) {
               alert("Enter a number between 1 and " + top + ".")
               select(field)
               return false
            }
         }
      }
      calcDate()
      return true
}

function validateYear(field) {
      var input = parseInt(field.value, 10)
      if (isEmpty(input)) {
         alert("Be sure to enter a month value.")
         select(field)
         return false
      } else {
         if (isNaN(input)) {
            alert("Entries must be numbers only.")
            select(field)
            return false
         } else {
            if (!inRange(input,1900,2005)) {
               alert("Enter a number between 1900 and 2005.")
               select(field)
               return false
            }
         }
      }
      calcDate()
      return true
}

function select(field) {
      field.focus()
      field.select()
}

function calcDate() {
      var mm = parseInt(document.birthdate.month.value, 10)
      var dd = parseInt(document.birthdate.date.value, 10)
      var yy = parseInt(document.birthdate.year.value, 10)
      document.birthdate.fullDate.value = mm + "/" + dd + "/" + yy
}
```

(continued)

Listing 28-3 *(continued)*

```
function checkForm(form) {
        if (validateMonth(form.month)) {
           if (validateDate(form.date)) {
              if (validateYear(form.year)) {
                 return true
              }
           }
        }
        return false
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="birthdate" ACTION="mailto:fun@dannyg.com" METHOD=POST
onSubmit="return checkForm(this)">
Please enter your birthdate...<BR>
Month:<INPUT TYPE="text" NAME="month" VALUE=1 SIZE=2
onChange="validateMonth(this)">
Date:<INPUT TYPE="text" NAME="date" VALUE=1 SIZE=2
onChange="validateDate(this)">
Year:<INPUT TYPE="text" NAME="year" VALUE=1900 SIZE=4
onChange="validateYear(this)">
<P>
Thank you for entering:<INPUT TYPE="text" NAME="fullDate" SIZE=10><P>
<INPUT TYPE="submit"> <INPUT TYPE="Reset">
</FORM>
</BODY>
</HTML>
```

The page shows the three entry fields as well as a field that would normally be hidden on a form to be submitted to a CGI program. On the server end, the CGI program would respond only to the hidden field with the complete date, which is in a format for entry into, say, an Informix database. Note that in the above example, the form's action is set to a `mailto`: URL, which means it won't work with Internet Explorer.

Not every date entry validation must be divided in this way. For example, an intranet application can be more demanding in the way users are to enter data. Therefore, you can have a single field for date entry, but the parsing required for such a validation is quite different from that shown in Listing 28-3. See Chapter 37 for an example of such a one-field date validation routine.

✦ ✦ ✦