Chapter 43: Data-Entry Validation

In This Chapter

Validating data as it is being entered

Validating data immediately prior to submission

Organizing complex data validation tasks

Give users a field in which to enter data and you can be sure that some users will enter the wrong kind of data. Often the "mistake" is accidental — a slip of the pinkie on the keyboard; other times, users intentionally type the incorrect entry to test the robustness of your application. Whether you solicit a user's entry for client-side scripting purposes or for input into a server-based CGI or database, you should use JavaScript on the client to handle validation of the user's entry. Even for a form connected to a CGI script, it's far more efficient (from the perspective of bandwidth, server load, and execution speed) to let client-side JavaScript get the data straight before your server program deals with it.

Real-time versus Batch Validation

You have two opportunities to perform data-entry validation in a form: as the user enters data into a form and just before the form is submitted. I recommend you take advantage of both of these opportunities.

Real-time validation triggers

The most convenient time to catch an error is immediately after the user makes it — especially for a long form that requests a wide variety of information. You can make the user's experience less frustrating if you catch an entry mistake just after the user enters the information: his or her attention is already focused on the nature of the content (or some paper source material may already be in front of the user). It is much easier for the user to address the same information entry right away.

A valid question for the page author is how to trigger the real-time validation. Backward-compatible text boxes have two potential event handlers for this purpose: `onChange` and `onBlur`. I personally avoid `onBlur` event handlers, especially ones that can display an alert dialog box (as a data-entry validation is likely to do). Because a good validation routine brings focus to the errant text box, you can get some odd behavior with the interaction of the `focus()` method and the `onBlur` event handler. Users who wish to continue past an invalid field are locked in a seemingly endless loop.

The problem with using `onChange` as the validation trigger is that a user can defeat the validation. A change event occurs only when the text of a field indeed changes when the user tabs or clicks out of the field. If the user is alerted about some bad entry in a field and doesn't fix the error, the change event doesn't fire again. In some respects, this is good because a user may have a legitimate reason for passing by a particular form field initially with the intention of returning to the entry later. Because a user can defeat the `onChange` event handler trigger, I recommend you also perform batch validation prior to submission.

In NN4+ and IE4+, you also have the luxury of letting keyboard events trigger validations. This is most helpful when you want to prevent some character(s) from being entered into a field. For example, if a field is supposed to contain only a positive integer value, you can use the `onKeyPress` event handler of the text box to verify that the character just typed is a number. If the character is not a number, the event is trapped and no character reaches the text box. You should also alert the user in some way about what's going on. Listing 43-1 demonstrates a simplified version of this kind of keyboard trapping, compatible with NN4+ and IE4+ event models. The message to the user is displayed in the statusbar. Displaying the message there has the advantage of being less intrusive than an alert dialog box (and keeps the text insertion cursor in the text box), but it also means that users might not see the message. The `onSubmit` event handler in the listing prevents a press of the Enter key in this one-field form from reloading this sample page.

Listing 43-1
Allowing Only Numbers into a Text Box

```
<HTML>
<HEAD>
<TITLE>Letting Only Numbers Pass to a Form Field</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function checkIt(evt) {
    evt = (evt) ? evt : window.event
    var charCode = (evt.which) ? evt.which : evt.keyCode
    if (charCode > 31 && (charCode < 48 || charCode > 57)) {
        status = "This field accepts numbers only."
        return false
    }
    status = ""
    return true
}
</SCRIPT>
</HEAD>

<BODY>
<H1>Letting Only Numbers Pass to a Form Field</H1>
<HR>
<FORM onSubmit="return false">
Enter any positive integer: <INPUT TYPE="text" NAME="numeric"
    onKeyPress="return checkIt(event)">
</FORM>
</BODY>
</HTML>
```

Keyboard event monitoring isn't practical for most validation actions, however. For example, if the user is supposed to enter an e-mail address, you need to validate the complete entry for the presence of an `@` symbol (via the `onChange` event handler). On the other hand, you can be granular about your validations and use both the `onChange` and `onKeyPress` event handlers; you employ the latter for blocking invalid characters in e-mail addresses (such as spaces).

Batch mode validation

In all scriptable browsers, the `onSubmit` event handler cancels the submission if the handler evaluates to `return false`. Additional submission event cancelers include setting the IE4+ `event.returnValue` property to `false` and invoking the *`evt`*`.preventDefault()` method in NN6 (see Chapter 29 on event objects for details). You can see an example of the basic `return false` behavior in Listing 23-4 of Chapter 23. That example uses the results of a `window.confirm()` dialog box to determine the return value of the event handler. But you can also use a return value from a series of individual text box validation functions. If any one of the validations fails, the user is alerted and the submission is canceled.

Before you worry about two versions of validation routines loading down the scripts in your page, you'll be happy to know that you can reuse the same validation routines in both the real-time and batch validations. Later in this chapter, I demonstrate what I call "industrial-strength" data-entry validation adapted from a real intranet application. But before you get there, you should learn about general validation techniques that you can apply to both types of validations.

Designing Filters

The job of writing data validation routines essentially involves designing filters that weed out characters or entries that don't fit your programming scheme. Whenever your filter detects an incorrect entry, it should alert the user about the nature of the problem and enable the user to correct the entry.

Before you put a text or TEXTAREA object into your document that invites users to enter data, you must decide if any possible entry can disturb the execution of the rest of your scripts. For example, if your script must have a number from that field to perform calculations, you must filter out any entry that contains letters or punctuation — except for periods if the program can accept floating-point numbers. Your task is to anticipate every possible entry users can make and allow only those entries your scripts can use.

Not every entry field needs a data validation filter. For example, you may prompt a user for information that is eventually stored as a `document.cookie` or in a string database field on the server for future retrieval. If no further processing takes place on that information, you may not have to worry about the specific contents of that field.

One other design consideration is whether a text field is even the proper user interface element for the data required of the user. If the range of choices for a user entry is small (a dozen or fewer), a more sensible method is to avoid the data-entry problem altogether by turning that field into a SELECT element. Your HTML attributes for the object ensure that you control the kind of entry made to that object. As long as your script knows how to deal with each of the options defined for that object, you're in the clear.

Building a Library of Filter Functions

A number of basic data-validation processes function repeatedly in form-intensive HTML pages. Filters for integers only, numbers only, empty entries, alphabet letters only, and the like are put to use every day. If you maintain a library of generalizable functions for each of your data-validation tasks, you can drop these functions into your scripts at a moment's notice and be assured that they will work. For NN3+ and IE4+, you can also create the library of validation functions as a separate `.js` library file and link the scripts into any HTML file that needs them.

Making validation functions generalizable requires careful choice of wording and logic so that they return Boolean values that make syntactical sense when called from elsewhere in your scripts. As you see later in this chapter, when you build a larger framework around smaller functions, each function is usually called as part of an `if...else` conditional statement. Therefore, assign a name that fits logically as part of an `if` clause in plain language. For example, you can name a function that checks whether an entry is empty `isEmpty()`. The calling statement for this function is:

```
if (isEmpty(value)) { ... }
```

From a plain-language perspective, the expectation is that the function returns `true` if the passed value is empty. With this design, the statements nested in the `if` construction handle empty entry fields. I revisit this design later in this chapter when I start stacking multiple-function calls together in a larger validation routine.

To get you started with your library of validation functions, this chapter provides some building blocks that you can learn from and use as starting points for more specific filters of your own design. Some of these functions are put to use in the JavaScript application in Chapter 50.

isEmpty()

This first function, shown in Listing 43-2, checks to see if the incoming value is either empty or `null`. Adding a check for `null` means that you can use this function for purposes other than just text-object validation. For example, if another function defines three parameter variables, but the calling function passes only two, the third variable is set to `null`. If the script then performs a data-validation check on all parameters, the `isEmpty()` function responds that the `null` value is devoid of data.

Listing 43-2

Is an Entry Empty or Null?

```
// general purpose function to see if an input value has been
// entered at all
function isEmpty(inputStr) {
    if (inputStr == null || inputStr == "") {
        return true
    }
    return false
}
```

This function uses a Boolean OR operator (`||`) to test for the existence of a `null` value or an empty string in the value passed to the function. Because the name of the function implies a `true` response if the entry is empty, that value is the one that returns to the calling statement if either condition is true. Because a `return` statement halts further processing of a function, the `return false` statement lies outside of the `if` construction. If processing reaches this statement, the `inputStr` value has failed the test.

If this seems like convoluted logic — `return true` when the value is empty — you can also define a function that returns the inverse values. You can name it `isNotEmpty()`. As it turns out, however, typical processing of an empty entry is better served when the test returns a `true` than when the value is empty — aiding the `if` construction that calls the function in the first place.

isPosInteger()

This next function examines each character of the value to make sure that only numbers from 0 through 9 with no punctuation or other symbols exist. The goal of the function in Listing 43-3 is to weed out any value that is not a positive integer.

Listing 43-3

Test for Positive Integers

```
// general purpose function to see if a suspected numeric input
// is a positive integer
function isPosInteger(inputVal) {
    inputStr = inputVal.toString()
    for (var i = 0; i < inputStr.length; i++) {
        var oneChar = inputStr.charAt(i)
        if (oneChar < "0" || oneChar > "9") {
            return false
        }
    }
    return true
}
```

Notice that this function makes no assumption about the data type of the value that is passed as a parameter. If the value had come directly from a text object, it would already be a string and the line that forced data conversion to a string would be unnecessary. But to generalize the function, the conversion is included to accommodate the possibility that it may be called from another statement that has a numeric value to check.

The function requires you to convert the input value to a string because it performs a character-by-character analysis of the data. A `for` loop picks apart the value one character at a time. Rather than force the script to invoke the *`string`*`.charAt()` method twice for each time through the loop (inside the `if` condition), one statement assigns the results of the method to a variable, which is then used twice in the `if` condition. Placing the results of the `charAt()` method into a variable makes the `if` condition shorter and easier to read and also is microscopically more efficient.

In the `if` condition, the ASCII value of each character is compared against the range of 0 through 9. This method is safer than comparing numeric values of the single characters because one of the characters can be nonnumeric. (You can encounter all kinds of other problems trying to convert that character to a number for numeric comparison.) The ASCII value, on the other hand, is neutral about the meaning of a character: If the ASCII value is less than 0 or greater than 9, the character is not valid for a genuine positive integer. The function bounces the call with a false reply. On the other hand, if the `for` loop completes its traversal of all characters in the value without a hitch, the function returns `true`.

You may wonder why this validation function doesn't use the `parseInt()` global function (Chapter 42). That function returns `NaN` only if the first character of the input string is not a number. But because `parseInt()` and `parseFloat()` peel off any initial numeric values from a string, neither returns `NaN` if the input string is, for example, `35a`.

isInteger()

The next possibility includes the entry of a negative integer value. Listing 43-4 shows that you must add an extra check for a leading negation sign.

Listing 43-4
Checking for Leading Minus Sign

```
// general purpose function to see if a suspected numeric input
// is a positive or negative integer
function isInteger(inputVal) {
    inputStr = inputVal.toString()
    for (var i = 0; i < inputStr.length; i++) {
        var oneChar = inputStr.charAt(i)
        if (i == 0 && oneChar == "-") {
            continue
        }
        if (oneChar < "0" || oneChar > "9") {
            return false
        }
    }
    return true
}
```

When a script can accept a negative integer, the filter must enable the leading minus sign to pass unscathed. You cannot just add the minus sign to the `if` condition of Listing 43-3

because you can accept that symbol only when it appears in the first position of the value — anywhere else makes the value an invalid number. To handle the possibility of a leading minus sign, you add another `if` statement whose condition looks for a special combination: the first character of the string (as indexed by the loop-counting variable) and the minus character. If both of these conditions are met, execution immediately loops back around to the update expression of the `for` loop (because of the `continue` statement) rather than carrying out the second `if` statement, which would obviously fail. By putting the `i == 0` comparison operation at the front of the condition, you ensure the entire condition short circuits to `false` for all subsequent iterations through the loop.

isNumber()

The final numeric filter function in this series enables any integer or floating-point number to pass while filtering out all others (Listing 43-5). All that distinguishes an integer from a floating-point number for data-validation purposes is the decimal point.

Listing 43-5
Testing for a Decimal Point

```
// general purpose function to see if a suspected numeric input
// is a positive or negative number
function isNumber(inputVal) {
    oneDecimal = false
    inputStr = inputVal.toString()
    for (var i = 0; i < inputStr.length; i++) {
        var oneChar = inputStr.charAt(i)
        if (i == 0 && oneChar == "-") {
            continue
        }
        if (oneChar == "." && !oneDecimal) {
            oneDecimal = true
            continue
        }
        if (oneChar < "0" || oneChar > "9") {
            return false
        }
    }
    return true
}
```

Anticipating the worst, however, the function cannot simply treat a decimal point at any position within the string as a valid character. Such an act assumes that no one would ever enter more than one decimal point into a numeric text field. Only one decimal point is allowed for this function (as well as for JavaScript math). Therefore, you add a Boolean flag variable (`oneDecimal`) to the function and a separate `if` condition that sets that flag to `true` when the function encounters the first decimal point. Should another decimal point appear in the string, the final `if` statement gets a crack at the character. Because the character falls outside the ASCII range of 0 through 9, it fails the entire function.

If you want to accept only positive floating-point numbers, you can make a new version of this function by removing the statement that lets the leading minus sign through. Be

aware that this function works only for values that are not represented in exponential notation.

Custom validation functions

The listings shown so far in this chapter should give you plenty of source material to use in writing customized validation functions for your applications. Listing 43-6 shows an example of such an application-specific variation (extracted from the application in Chapter 50).

Listing 43-6

A Custom Validation Function

```
// function to determine if value is in acceptable range
// for this application
function inRange(inputStr) {
    num = parseInt(inputStr)
    if (num < 1 || num > 586 && num < 596 || num > 599 && num < 700 || num > 728)
{
        return false
    }
    return true
}
```

For this application, you need to see if an entry falls within multiple ranges of acceptable numbers. The first statement of the `inRange()` function converts the incoming value to a number (via the `parseInt()` function) so that the value can be compared numerically against maximum and minimum values of several ranges within the database. Following the logic of the previous validation functions, the `if` condition looks for values outside the acceptable range, so it can alert the user and return a `false` value.

The `if` condition is quite a long sequence of operators. As you noticed in the list of operator precedence (Chapter 40), the Boolean AND operator (`&&`) has precedence over the Boolean OR operator (`||`). Therefore, the AND expressions evaluate first, followed by the OR expressions. Parentheses may help you better visualize what's going on in that monster condition:

```
if (num < 1 || (num > 586 && num < 596) ||(num > 599 && num < 700) || num > 728)
```

In other words, you exclude four possible ranges from consideration:

* Values less than 1
* Values between 586 and 596
* Values between 599 and 700
* Values greater than 728

Any value for which any one of these tests is true yields a Boolean `false` from this function. Combining all these tests into a single condition statement eliminates the need to construct an otherwise complex series of nested `if` constructions.

Combining Validation Functions

When you design a page that requests a particular kind of text input from a user, you often need to call more than one data-validation function to handle the entire job. For example, if you merely want to test for a positive integer entry, your validation should test for the presence of any entry as well as the validation as an integer.

After you know the kind of permissible data that your script will use after validation, you're ready to plot the sequence of data validation. Because each page's validation task is different, I supply some guidelines to follow in this planning rather than prescribe a fixed route for all to take.

My preferred sequence is to start with examinations that require less work and increase the intensity of validation detective work with succeeding functions. I borrow this tactic from real life: When a lamp fails to turn on, I look for a pulled plug or a burned-out lightbulb before tearing the lamp's wiring apart to look for a short.

Using the data-validation sequence from the data-entry field (which must be a three-digit number within a specified range) in Chapter 50, I start with the test that requires the least amount of work: Is there an entry at all? After my script is ensured an entry of some kind exists, it next checks whether that entry is "all numbers as requested of the user." If so, the script compares the number against the ranges of numbers in the database.

To make this sequence work together efficiently, I create a master validation function consisting of nested `if...else` statements. Each `if` condition calls one of the generalized data-validation functions. Listing 43-7 shows the master validation function.

Listing 43-7
Master Validation Function

```
// Master value validator routine
function isValid(inputStr) {
    if (isEmpty(inputStr)) {
        alert("Please enter a number into the field before clicking the button.")
        return false
    } else {
        if (!isNumber(inputStr)) {
            alert("Please make sure entries are numbers only.")
            return false
        } else {
            if (!inRange(inputStr)) {
                var msg = "Sorry, the number you entered is not part of our
database."
                msg += "Try another three-digit number."
                alert(msg)
                return false
```

```
            }
        }
    }
    return true
}
```

This function, in turn, is called by the function that controls most of the work in this application. All that the main function wants to know is whether the entered number is valid. The details of validation are handed off to the `isValid()` function and its special-purpose validation testers.

I construct the logic in Listing 43-7 so that if the input value fails to be valid, the `isValid()` function alerts the user of the problem and returns `false`. That means I have to watch my `true`s and `false`s very carefully.

In the first validation test, an empty value is a bad thing; thus, when the `isEmpty()` function returns `true`, the `isValid()` function returns `false` because an empty string is not a valid entry. In the second test, a number value is good so the logic has to flip 180 degrees. The `isValid()` function returns `false` only if the `isNumber()` function returns `false`. But because `isNumber()` returns `true` when the value is a number, I switch the condition to test for the opposite results of the `isNumber()` function by negating the function name (preceding the function with the Boolean NOT (`!`) operator). This operator works only with a value that evaluates to a Boolean expression — which the `isNumber()` function always does. The final test for being within the desired range works on the same basis as `isNumber()`, using the Boolean NOT operator to turn the results of the `inRange()` function into the method that works best for this sequence.

Finally, if all validation tests fail to find bad or missing data, the entire `isValid()` function returns `true`. The statement that calls this function can now proceed with processing, ensured that the value entered by the user will work.

There is one additional point worth reinforcing, especially for newcomers. Although all these functions seem to be passing around the same input string as a parameter, notice that any changes made to the value (such as converting it to a string or number) are kept private to each function. These subfunctions never touch the original value in the calling function — they work only with copies of the original value . Therefore, even after the data validation takes place, the original value is in its original form and ready to go.

Date and Time Validation

You can scarcely open a bigger can of cultural worms than when trying to program around the various date and time formats in use around the world. If you have ever looked through the possible settings in your computer’s operating system, you can begin to understand the difficulty of this issue.

Trying to write JavaScript that accommodates all of the world's date and time formats for validation is an enormous, if not wasteful, challenge. It's one thing to validate that a text box contains data in the form `xx/xx/xxxx`, but there are also valid value concerns that can get very messy on an international basis. For example, while North America typically uses the `mm/dd/yyyy` format, a large portion of the rest of the world uses `dd/mm/yyyy` (with different delimiter characters, as well). Therefore, how should your validation routine treat the entry `20/03/2002`? Is it incorrect because there are not 20 months in a year; or is it correct as March 20th? To query a user for this kind of information, I suggest you divide the components into individually validated fields (separate text objects for hours and minutes) or make SELECT element entries whose individual values can be assembled at submit time into a hidden date field for processing by the database that needs the date information. (Alternately, you can let your server CGI handle the conversion.)

Despite my encouragement to "divide and conquer" date entries, there may be situations in which you feel it's safe to provide a single text box for date entry (perhaps for a form that is used on a corporate intranet strictly by users in one country). You see some more sophisticated code later in this chapter, but a "quick-and-dirty" solution runs along these lines:

1. Use the entered data (for example, in `mm/dd/yyyy` format) as a value passed to the `new Date()` constructor function.

2. From the newly created date object, extract each of the three components (month, day, and year) into separate numeric values (with the help of `parseInt()`).

3. Compare each of the extracted values against the corresponding date, month, and year values returned by the date object's `getDate()`, `getMonth()`, and `getFullYear()` methods (adjusting for zero-based values of `getMonth()`).

4. If all three pairs of values match, then the entry is apparently valid.

Listing 43-8 puts this action sequence to work. The `validDate()` function receives a reference to the field being checked. A copy of the field's value is made into a date object, and its components are read. If any part of the date conversion or component extraction fails (because of improperly formatted data or unexpected characters), one or more of the variable values becomes `NaN`. This code assumes that the user enters a date in the `mm/dd/yyyy` format, which is the sequence that the `Date` object constructor expects its data. If the user enters `dd/mm/yyyy`, the validation fails for any day beyond the 12th.

Listing 43-8
Simple Date Validation

```
<HTML>
<HEAD>
<TITLE>Simple Date Validation</TITLE>
<SCRIPT LANGUAGE="JavaScript">
```

```
function validDate(fld) {
    var testMo, testDay, testYr, inpMo, inpDay, inpYr, msg
    var inp = fld.value
    status = ""
    // attempt to create date object from input data
    var testDate = new Date(inp)
    // extract pieces from date object
    testMo = testDate.getMonth() + 1
    testDay = testDate.getDate()
    testYr = testDate.getFullYear()
    // extract components of input data
    inpMo = parseInt(inp.substring(0, inp.indexOf("/")), 10)
    inpDay = parseInt(inp.substring((inp.indexOf("/") + 1), inp.lastIndexOf("/")),
10)
    inpYr = parseInt(inp.substring((inp.lastIndexOf("/") + 1), inp.length), 10)
    // make sure parseInt() succeeded on input components
    if (isNaN(inpMo) || isNaN(inpDay) || isNaN(inpYr)) {
        msg = "There is some problem with your date entry."
    }
    // make sure conversion to date object succeeded
    if (isNaN(testMo) || isNaN(testDay) || isNaN(testYr)) {
        msg = "Couldn't convert your entry to a valid date. Try again."
    }
    // make sure values match
    if (testMo != inpMo || testDay != inpDay || testYr != inpYr) {
        msg = "Check the range of your date value."
    }
    if (msg) {
        // there's a message, so something failed
        alert(msg)
        // work around IE timing problem with alert by
        // invoking a focus/select function through setTimeout();
        // must pass along reference of fld (as string)
        setTimeout("doSelection(document.forms['" +
        fld.form.name + "'].elements['" + fld.name + "'])", 0)
        return false
    } else {
        // everything's OK; if browser supports new date method,
        // show just date string in status bar
        status = (testDate.toLocaleDateString) ? testDate.toLocaleDateString() :
            "Date OK"
        return true
    }
}

// separate function to accommodate IE timing problem
function doSelection(fld) {
    fld.focus()
    fld.select()
}
</SCRIPT>
</HEAD>

<BODY>
<H1>Simple Date Validation</H1>
<HR>
<FORM NAME="entryForm" onSubmit="return false">
Enter any date (mm/dd/yyyy): <INPUT TYPE="text" NAME="startDate"
    onChange="validDate(this)">
</FORM>
</BODY>
</HTML>
```

Selecting Text Fields for Reentry

During both real-time and batch validations, it is especially helpful to the user if your code — upon discovering an invalid entry — not only brings focus to the subject text field, but also selects the content for the user. By preselecting the entire field, you make it easy for the user to just retype the data into the field for another attempt (or to begin using the left and right arrow keys to move the insertion cursor for editing). The reverse type on the field text also helps bring attention to the field. (Not all operating systems display a special rectangle around a focused text field.)

Form fields have both `focus()` and `select()` methods, which you should invoke for the subject field in that order. IE for Windows, however, exhibits undesirable behavior when trying to focus and select a field immediately after you close an alert dialog box. In most cases, the field does not keep its focus or selection. This is a timing problem, but one that you can cure by processing the focus and select actions through a `setTimeout()` method. The bottom of the script code of Listing 43-9 demonstrates how to do this.

Method calls to the form field reside in a separate function (called `doSelection()` in this example). Obviously, the methods need a reference to the desired field, so the `doSelection()` function requires access to that reference. You can use a global variable to accomplish this (set the value in the validation function; read it in the `doSelection()` function), but globals are not elegant solutions to passing transient data. Even though the validation function receives a reference to the field, that is an object reference, and the `setTimeout()` function's first parameter cannot be anything but a string value. Therefore, the reference to the text field provides access to names of both the form and field. The names fill in as index values for arrays so that the assembled string (upon being invoked) evaluates to a valid object reference:

```
"doSelection(document.forms['" + fld.form.name + "'].elements['" + fld.name +
"'])"
```

Notice the generous use of built-in `forms` and `elements` object arrays, which allow the form and field names to assemble the reference without resorting to the onerous `eval()` function.

For timing problems such as this one, no additional time is truly needed to let IE recover from whatever ails it. Thus, the time parameter is set to 0 milliseconds. Using the `setTimeout()` portal is enough to make everything work. There is no penalty for using this construction with NN or IE/Mac, even though they don't need it.

An "Industrial-Strength" Validation Solution

I had the privilege of working on a substantial intranet project that included dozens of forms, often with two or three different kinds of forms displayed simultaneously within a frameset. Data-entry accuracy was essential to the validity of the entire application. My task was to devise a data-entry validation strategy that not only ensured accurate entry of

data types for the underlying (SQL) database, but also intelligently prompted users who made mistakes in their data entry.

Structure

From the start, the validation routines were to be in a client-side library linked in from an external `.js` file. That would allow all forms to share the validation functions. Because there were multiple forms displayed in a frameset, it would prove too costly in download time and memory requirements to include the `validations.js` file in every frame's document. Therefore, the library was moved to load in with the frameset. The `<SCRIPT SRC="validations.js"></SCRIPT>` tag set went in the Head portion of the framesetting document.

This logical placement presented a small challenge for the workings of the validations because there had to be two-way conversations between a validation function (in the frameset) and a form element (nested in a frame). The mechanism required that a reference to the frame containing the form element be passed as part of the validation routine so that the validation script could make corrections, automatic formatting, and erroneous field selections from the frameset document's script. (In other words, the frameset script needed a path back to the form element making the validation call.)

Dispatch mechanism

From the specification drawn up for the application, it is clear that there are more than two dozen specific types of validations across all the forms. Moreover, multiple programmers work on different forms. It is helpful to standardize the way validations are called, regardless of the validation type (number, string, date, phone number, and so on).

My idea was to create one `validate()` function that contained parameters for the current frame, the current form element, and the type of validation to perform. This would make it clear to anyone reading the code later that an event handler calling `validate()` performed validation, and the details of the code were in the `validations.js` library file.

In `validations.js`, I converted a string name of a validation type into the name of the function that performs the validation in order to make this idea work. As a bridge between the two, I created what I called a *dispatch lookup table* for all the primary validation routines that would be called from the forms. Each entry of the lookup table had a label consisting of the name of the validation and a method that invoked the function. Listing 43-9 shows an excerpt of the entire lookup table creation mechanism.

Listing 43-9
Creating the Dispatch Lookup Table

```
/*
   Begin validation dispatching mechanism
```

```
*/
function dispatcher(validationFunc) {
   this.doValidate = validationFunc
}
var dispatchLookup = new Array()
dispatchLookup["isNotEmpty"] = new dispatcher(isNotEmpty)
dispatchLookup["isPositiveInteger"] = new dispatcher(isPositiveInteger)
dispatchLookup["isDollarsOnly8"] = new dispatcher(isDollarsOnly8)
dispatchLookup["isUSState"] = new dispatcher(isUSState)
dispatchLookup["isZip"] = new dispatcher(isZip)
dispatchLookup["isExpandedZip"] = new dispatcher(isExpandedZip)
dispatchLookup["isPhone"] = new dispatcher(isPhone)
dispatchLookup["isConfirmed"] = new dispatcher(isConfirmed)
dispatchLookup["isNY"] = new dispatcher(isNY)
dispatchLookup["isNum16"] = new dispatcher(isNum16)
dispatchLookup["isM90_M20Date"] = new dispatcher(isM90_M20Date)
dispatchLookup["isM70_0Date"] = new dispatcher(isM70_0Date)
dispatchLookup["isM5_P10Date"] = new dispatcher(isM5_P10Date)
dispatchLookup[“isDateFormat”] = new dispatcher(isDateFormat)
```

Each entry of the array is assigned a `dispatcher` object, whose custom object constructor assigns a function reference to the object's `doValidate()` method. For these assignment statements to work, their corresponding functions must be defined earlier in the document. You can see some of these functions later in this section.

The link between the form elements and the dispatch lookup table is the `validate()` function, shown in Listing 43-10. A call to `validate()` requires a minimum of three parameters, as shown in the following example:

```
<INPUT TYPE="text" NAME="phone" SIZE="10"
onChange="parent.validate(window, this, 'isPhone')">
```

The first is a reference to the frame containing the document that is calling the function (passed as a reference to the current window). The second parameter is a reference to the form control element itself (using the `this` operator). After that, you see one or more individual validation function names as strings. This last design allows more than one type of validation to take place with each call to `validate()` (for example, in case a field must check for a data type and check that the field is not empty).

Listing 43-10
Main Validation Function

```
// main validation function called by form event handlers
function validate(frame, field, method) {
    gFrame = frame
    gField =  window.frames[frame.name].document.forms[0].elements[field.name]
    var args = validate.arguments
    for (i = 2; i < args.length; i++) {
        if (!dispatchLookup[args[i]].doValidate()) {
            return false
        }
    }
    return true
}
```

In the `validate()` function, the frame reference is assigned to a global variable that is declared at the top of the `validations.js` file. Validation functions in this library

need this information to build a reference back to a companion field required of some validations (explained later in this section). A second global variable contains a reference to the calling form element. Because the form element reference by itself does not contain information about the frame in which it lives, the script must build a reference out of the information passed as parameters. The reference must work from the framesetting document down to the frame, its form, and form element name. Therefore, I use the `frame` and `field` object references to get their respective names (within the `frames` and `elements` arrays) to assemble the text field's object reference; the resulting value is assigned to the `gField` global variable. I choose to use global variables in this case because passing these two values to numerous nested validation functions could be difficult to track reliably. Instead, the only parameter passed to specific validation functions is the value under test.

Next, the script creates an array of all arguments passed to the `validate()` function. A `for` loop starts with an index value of `2`, the third parameter containing the first validation function name. For each one, the named item's `doValidate()` method is called. If the validation fails, this function returns `false`; but if all validations succeed, then this function returns `true`. Later you see that this function's returned value is the one that allows or disallows a form submission.

Sample validations

Above the dispatching mechanism in the `validations.js` are the validation functions themselves. Many of the named validation functions have supporting utility functions that other named validation functions often use. Because of the eventual large size of this library file (the production version was about 40 kilobytes), I organized the functions into two groups: the named functions first, and the utility functions below them (but still before the dispatching mechanism at the bottom of the document).

To demonstrate how some of the more common data types are validated for this application, I show several validation functions and, where necessary, their supporting utility functions. Figure 43-1 shows a sample form that takes advantage of these validations. (You have a chance to try it later in this chapter.) When you are dealing with critical corporate data, you must go to extreme lengths to ensure valid data. And to help users see their mistakes quickly, you need to build some intelligence into validations where possible.

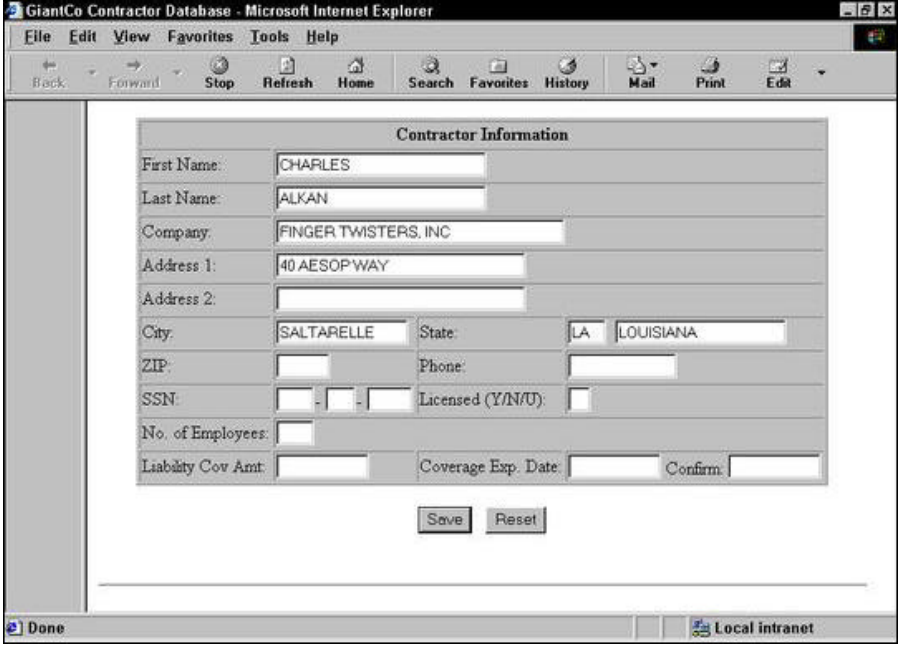

Figure 43-1
Sample form for industrial-strength validations

U.S. state name

The design specification for forms that require entry of a U.S. state calls for entry of the state's two-character abbreviation. A companion field to the right displays the entire state name as user feedback verification. The onChange event handler not only calls the validation, but it also feeds the focus to the field following the expanded state field so users are less likely to type into it.

Before the validation can even get to the expansion part, it must first validate that the entry is a valid, two-letter abbreviation. Because I need both the abbreviation and the full state name for this validation, I create an array of all the states using each state abbreviation as the index label for each entry. Listing 43-11 shows that array creation.

Listing 43-11
Creating a U.S. States Array

```
// States array
var USStates = new Array(51)
USStates["AL"] = "ALABAMA"
USStates["AK"] = "ALASKA"
USStates["AZ"] = "ARIZONA"
USStates["AR"] = "ARKANSAS"
USStates["CA"] = "CALIFORNIA"
USStates["CO"] = "COLORADO"
USStates["CT"] = "CONNECTICUT"
USStates["DE"] = "DELAWARE"
USStates["DC"] = "DISTRICT OF COLUMBIA"
USStates["FL"] = "FLORIDA"
USStates["GA"] = "GEORGIA"
```

```
USStates["HI"] = "HAWAII"
USStates["ID"] = "IDAHO"
USStates["IL"] = "ILLINOIS"
USStates["IN"] = "INDIANA"
USStates["IA"] = "IOWA"
USStates["KS"] = "KANSAS"
USStates["KY"] = "KENTUCKY"
USStates["LA"] = "LOUISIANA"
USStates["ME"] = "MAINE"
USStates["MD"] = "MARYLAND"
USStates["MA"] = "MASSACHUSETTS"
USStates["MI"] = "MICHIGAN"
USStates["MN"] = "MINNESOTA"
USStates["MS"] = "MISSISSIPPI"
USStates["MO"] = "MISSOURI"
USStates["MT"] = "MONTANA"
USStates["NE"] = "NEBRASKA"
USStates["NV"] = "NEVADA"
USStates["NH"] = "NEW HAMPSHIRE"
USStates["NJ"] = "NEW JERSEY"
USStates["NM"] = "NEW MEXICO"
USStates["NY"] = "NEW YORK"
USStates["NC"] = "NORTH CAROLINA"
USStates["ND"] = "NORTH DAKOTA"
USStates["OH"] = "OHIO"
USStates["OK"] = "OKLAHOMA"
USStates["OR"] = "OREGON"
USStates["PA"] = "PENNSYLVANIA"
USStates["RI"] = "RHODE ISLAND"
USStates["SC"] = "SOUTH CAROLINA"
USStates["SD"] = "SOUTH DAKOTA"
USStates["TN"] = "TENNESSEE"
USStates["TX"] = "TEXAS"
USStates["UT"] = "UTAH"
USStates["VT"] = "VERMONT"
USStates["VA"] = "VIRGINIA"
USStates["WA"] = "WASHINGTON"
USStates["WV"] = "WEST VIRGINIA"
USStates["WI"] = "WISCONSIN"
USStates[“WY”] = " WYOMING"
```

The existence of this array comes in handy in determining if the user enters a valid, two-state abbreviation. Listing 43-12 shows the actual `isUSState()` validation function that puts this array to work.

The function’s first task is to assign an uppercase version of the entered value to a local variable (`inputStr`), which is the value being analyzed throughout the rest of the function. If the user enters something in the field (`length > 0`) but no entry in the `USStates` array exists for that value, the entry is not a valid state abbreviation. Time to go to work to help out the user.

Listing 43-12

Validation Function for U.S. States

```
// input value is a U.S. state abbreviation; set entered value to all uppercase
// also set companion field (NAME="<xxx>_expand") to full state name
function isUSState() {
    var inputStr = gField.value.toUpperCase()
    if (inputStr.length > 0 && USStates[inputStr] == null) {
        var msg = ""
```

```
        var firstChar = inputStr.charAt(0)
        if (firstChar == "A") {
            msg += "\n(Alabama = AL; Alaska = AK; Arizona = AZ; Arkansas = AR)"
        }
        if (firstChar == "D") {
            msg += "\n(Delaware = DE; District of Columbia = DC)"
        }
        if (firstChar == "I") {
            msg += "\n(Idaho = ID; Illinois = IL; Indiana = IN; Iowa = IA)"
        }
        if (firstChar == "M") {
            msg += "\n(Maine = ME; Maryland = MD; Massachusetts = MA;  " +
                       "Michigan = MI; Minnesota = MN; Mississippi = MS; " +
                       "Missouri = MO; Montana = MT)"
        }
        if (firstChar == "N") {
            msg += "\n(Nebraska = NE; Nevada = NV)"
        }
        alert("Check the spelling of the state abbreviation." + msg)
        gField.focus()
        gField.select()
        return false
    }
    gField.value = inputStr
    var expandField =
window.frames[gFrame.name].document.forms[0].elements[gField.name + "_expand"]
    expandField.value = USStates[inputStr]
    return true
}
```

The function assumes that the user tried to enter a valid state abbreviation but either had incorrect source material or momentarily forgot a particular state's abbreviation. Therefore, the function examines the first letter of the entry. If that first letter is any one of the five identified as causing the most difficulty, a legend for all states beginning with that letter is assigned to the `msg` variable (for running on newer browsers only, a `switch` construction is preferred). An alert message displays the generic alert, plus any special legend if one is assigned to the `msg` variable. When the user closes the alert, the field has focus and its text is selected. (This application runs solely on Navigator, so the IE `setTimeout()` workaround isn't needed — but you can add it very easily, especially thanks to the global variable reference for the field.) The function returns `false` at this point.

If, on the other hand, the abbreviation entry is a valid one, the field is handed the uppercase version of the entry. The script then uses the two global variables set in `validate()` to create a reference to the expanded display field (whose name must be the same as the entry field plus `"_expand"`). That expanded display field is then supplied the `USStates` array entry value corresponding to the abbreviation label. All is well with this validation, so it returns `true`.

You can see here that the so-called validation routine is doing far more than simply checking validity of the data. By communicating with the field, converting its contents to uppercase, and talking to another field in the form, a simple call to the validation function yields a lot of mileage.

Date validation

Many of the forms in this application have date fields. In fact, dates are an important part of the data maintained in the database behind the forms. All users of this application are familiar with standard date formats in use in the United States, so I don't have to worry about the possibility of cultural variations in date formats. Even so, I want the date entry to accommodate the common date formats, such as `mmddyyyy`, `mm/dd/yyyy`, and `mm-dd-yyyy` (as well as accommodate two-digit year entries spanning 1930 to 2029).

The plan also calls for going further in helping users enter dates within certain ranges. For example, a field used for a birthdate (the listings are for medical professionals) should recommend dates starting no more than 90 years, and no less than 20 years, from the current date. And to keep this application running well into the future, the ranges should be on a sliding scale from the current year, no matter when it might be. Whatever the case, the date range validation is only a recommendation and not a transaction stopper.

Rather than create separate validation functions for each date field, I create a system of reusable validation functions for each date range (several fields on different forms require the same date ranges). Each one of these individual functions calls a single, generic date-validation function that handles the date-range checking. Listing 43-13 shows a few examples of these individual range-checking functions.

Listing 43-13
Date Range Validations

```
// Date Minus 90/Minus 20
function isM90_M20Date() {
    if (gField.value.length == 0) return true
    var thisYear = getTheYear()
    return isDate((thisYear - 90),(thisYear - 20))
}

// Date Minus 70/Minus 0
function isM70_0Date() {
    if (gField.value.length == 0) return true
    var thisYear = getTheYear()
    return isDate((thisYear - 70),(thisYear))
}

// Date Minus 5/Plus 10
function isM5_P10Date() {
    if (gField.value.length == 0) return true
    var thisYear = getTheYear()
    return isDate((thisYear - 5),(thisYear + 10))
}
```

The naming convention I create for the functions includes the two range components relative to the current date. A letter "M" means the range boundary is minus a number of years from the current date; "P" means the range is plus a number of years. If the boundary should be the current year, a zero is used. Therefore, the `isM5_P10Date()` function performs range checking for boundaries between five years before and 10 years after the current year.

Before performing any range checking, each function makes sure there is some value to validate. If the field entry is empty, the function returns `true`. This is fine here because dates are not required when the data is unknown.

Next, the functions get the current four-digit year. The code here had to work originally with browsers that did not have the `getFullYear()` method available yet. Therefore, the Y2K fix described in Chapter 36 was built into the application:

```
function getTheYear() {
    var thisYear = (new Date()).getYear()
    thisYear = (thisYear < 100)? thisYear + 1900: thisYear
    return thisYear
}
```

The final call from the range validations is to a common `isDate()` function, which handles not only the date range validation but also the validation for valid dates (for example, making sure that September has only 30 days). Listing 43-14 shows this monster-sized function. Because of the length of this function, I interlace commentary within the code listing.

Listing 43-14
Primary Date Validation Function

```
// date field validation (called by other validation functions that specify
minYear/maxYear)
function isDate(minYear,maxYear,minDays,maxDays) {
    var inputStr = gField.value
```

To make it easier to work with dates supplied with delimiters, I first convert hyphen delimiters to slash delimiters. The pre-regular expression `replaceString()` function is the same one described in Chapter 34; it is located in the utility functions part of the `validations.js` file.

```
    // convert hyphen delimiters to slashes
    while (inputStr.indexOf("-") != -1) {
        inputStr = replaceString(inputStr,"-","/")
    }
```

For validating whether the gross format is OK, I check whether zero or two delimiters appear. If the value contains only one delimiter, then the overall formatting is not acceptable. The error alert shows models for acceptable date-entry formats.

```
    var delim1 = inputStr.indexOf("/")
    var delim2 = inputStr.lastIndexOf("/")
    if (delim1 != -1 && delim1 == delim2) {
        // there is only one delimiter in the string
        alert("The date entry is not in an acceptable format.\n\nYou can enter
dates in the following formats: mmddyyyy, mm/dd/yyyy, or mm-dd-yyyy.  (If the
month or date data is not available, enter \'01\' in the appropriate location.)")
        gField.focus()
        gField.select()
        return false
    }
```

If there are two delimiters, I tear apart the string into components for month, day, and year. Because two-digit entries can begin with zeros, I make sure the `parseInt()` functions specify base-10 conversions.

```
    if (delim1 != -1) {
        // there are delimiters; extract component values
        var mm = parseInt(inputStr.substring(0,delim1),10)
        var dd = parseInt(inputStr.substring(delim1 + 1,delim2),10)
        var yyyy = parseInt(inputStr.substring(delim2 + 1, inputStr.length),10)
```

For no delimiters, I tear apart the string and assume two-digit entries for the month and day and two or four digits for the year.

```
    } else {
        // there are no delimiters; extract component values
        var mm = parseInt(inputStr.substring(0,2),10)
        var dd = parseInt(inputStr.substring(2,4),10)
        var yyyy = parseInt(inputStr.substring(4,inputStr.length),10)
    }
```

The `parseInt()` functions reveal whether any entry is not a number by returning `NaN`, so I check whether any of the three values is not a number. If so, then an alert signals the formatting problem and supplies acceptable models.

```
    if (isNaN(mm) || isNaN(dd) || isNaN(yyyy)) {
        // there is a non-numeric character in one of the component values
        alert("The date entry is not in an acceptable format.\n\nYou can enter
dates in the following formats: mmddyyyy, mm/dd/yyyy, or mm-dd-yyyy.")
        gField.focus()
        gField.select()
        return false
    }
```

Next, I perform some gross range validation on the month and date to make sure that months are entered from 1 to 12 and dates from 1 to 31. I take care of aligning exact month lengths later.

```
    if (mm < 1 || mm > 12) {
        // month value is not 1 thru 12
        alert("Months must be entered between the range of 01 (January) and 12
(December).")
        gField.focus()
        gField.select()
        return false
    }
    if (dd < 1 || dd > 31) {
        // date value is not 1 thru 31
        alert("Days must be entered between the range of 01 and a maximum of 31
(depending on the month and year).")
        gField.focus()
        gField.select()
        return false
    }

    // validate year, allowing for checks between year ranges
    // passed as parameters from other validation functions
```

Before getting too deep into the year validation, I convert any two-digit year within the specified range to its four-digit equivalent.

```
    if (yyyy < 100) {
        // entered value is two digits, which we allow for 1930-2029
```

```
        if (yyyy >= 30) {
            yyyy += 1900
        } else {
            yyyy += 2000
        }
    }

    var today = new Date()
```

I designed this function to work with a pair of year ranges or date ranges (so many days before and/or after today). If the function is passed date ranges, then the first two parameters must be passed as `null`. This first batch of code works with the date ranges (because the `minYear` parameter is `null`).

```
    if (!minYear) {
        // function called with specific day range parameters
        var dateStr = new String(monthDayFormat(mm) + "/" + monthDayFormat(dd) +
        "/" + yyyy)
        var testDate = new Date(dateStr)
        if (testDate.getTime() < (today.getTime() + (minDays * 24 * 60 * 60 *
1000))) {
            alert("The most likely range for this entry begins " + minDays +
            " days from today.")
        }
        if (testDate.getTime() > today.getTime() + (maxDays * 24 * 60 * 60 *
1000)) {
            alert("The most likely range for this entry ends " + maxDays +
            " days from today.")
        }
```

You can also pass hard-wired, four-digit years as parameters. The following branch compares the entered year against the range specified by those passed year values.

```
    } else if (minYear && maxYear) {
        // function called with specific year range parameters
        if (yyyy < minYear || yyyy > maxYear) {
            // entered year is outside of range passed from calling function
            alert("The most likely range for this entry is between the years " +
minYear + " and " + maxYear + ".  If your source data indicates a date outside
this range, then enter that date.")
        }
    } else {
```

For year parameters passed as positive or negative year differences, I begin processing by getting the four-digit year for today's date. Then I compare the entered year against the passed range values. If the entry is outside the desired range, an alert reveals the preferred year range within which the entry should fall. But the function does not return any value here because an out-of-range value is not critical for this application.

```
        // default year range (now set to (this year - 100) and (this year + 25))
        var thisYear = today.getYear()
        if (thisYear < 100) {
        thisYear += 1900
        }
        if (yyyy < minYear || yyyy > maxYear) {
            alert("It is unusual for a date entry to be before " + minYear + " or
after " + maxYear + ". Please verify this entry.")
        }
    }
```

One more important validation is to make sure that the entered date is valid for the month and year. Therefore, the various date components are passed to functions to check against month lengths, including the special calculations for the varying length of February. Listing 43-15 shows these functions. The alert messages they display are smart enough to inform the user what the maximum date is for the entered month and year.

```
    if (!checkMonthLength(mm,dd)) {
        gField.focus()
        gField.select()
        return false
    }
    if (mm == 2) {
        if (!checkLeapMonth(mm,dd,yyyy)) {
            gField.focus()
            gField.select()
            return false
        }
    }
```

The final task is to reassemble the date components into a format that the database wants for its date storage and stuff it into the form field. If the user enters an all-number or hyphen-delimited date, it is automatically reformatted and displayed as a slash-delimited, four-digit-year date.

```
    // put the Informix-friendly format back into the field
    gField.value = monthDayFormat(mm) + "/" + monthDayFormat(dd) + "/" + yyyy
    return true
}
```

A utility function invoked multiple times in the previous function converts a single-digit month or day number to a string that might have a leading zero:

```
// convert month or day number to string,
// padding with leading zero if needed
function monthDayFormat(val) {
    if (isNaN(val) || val == 0) {
        return "01"
    } else if (val < 10) {
        return "0" + val
    }
    return "" + val
}
```

Listing 43-15

Functions to Check Month Lengths

```
// check the entered month for too high a value
function checkMonthLength(mm,dd) {
    var months = new
Array("","January","February","March","April","May","June","July",
        "August","September","October","November","December")
    if ((mm == 4 || mm == 6 || mm == 9 || mm == 11) && dd > 30) {
        alert(months[mm] + " has only 30 days.")
        return false
    } else if (dd > 31) {
        alert(months[mm] + " has only 31 days.")
        return false
    }
    return true
}
```

```
// check the entered February date for too high a value
function checkLeapMonth(mm,dd,yyyy) {
    if (yyyy % 4 > 0 && dd > 28) {
        alert("February of " + yyyy + " has only 28 days.")
        return false
    } else if (dd > 29) {
        alert("February of " + yyyy + " has only 29 days.")
        return false
    }
    return true
}
```

This is a rather extensive date-validation routine, but it demonstrates how thorough you must be when a database relies on accurate entries. The more prompting and assistance you can give to users to ferret out problems with invalid entries, the happier those users will be.

Cross-confirmation fields

The final validation type that I cover here is probably not a common request, but it demonstrates how the dispatch mechanism created at the outset expands so easily to accommodate this enhanced client request. The situation is that some fields (mostly dates in this application) are deemed critical pieces of data because this data triggers other processes from the database. As a further check to ensure entry of accurate data, a number of values are set up for entry twice in separate fields — and the fields have to match exactly. In many ways, this mirrors the two passes you are often requested to make when you set a password: enter two copies and let the computer compare them to make sure you typed what you intended to type.

I established a system that places only one burden on the many programmers working on the forms: while you can name the primary field anything you want (to help alignment with database column names, for example), you must name the secondary field the same plus `"_xcfm"` — which stands for *cross-confirm*. Then, pass the `isConfirmed` validation name to the `validate()` function after the date range validation name, as follows:

```
onChange="parent.validate(window, this, 'isM5_P10Date','isConfirmed')"
```

In other words, after the entered value is initially checked against a required date range, the `isConfirmed()` validation function compares the fully vetted, properly formatted date in the current field against its parallel entry.

Listing 43-16 shows the one function in `validations.js` that handles the confirmation in both directions. After assigning a copy of the entry field value to the `inputStr` variable, the function next sets a Boolean flag (`primary`) that lets the rest of the script know if the entry field is the primary or secondary field. If the string `"_xcfm"` is missing from the field name, then the entry field is the primary field.

For the primary field branch, the script assembles the name of the secondary field and compares the content of the secondary field's value against the `inputStr` value. If they

are not the same, the user is entering a new value into the primary field and the script empties the secondary field to force reentry to verify that the user enters the proper data.

For the secondary field entry branch, the script assembles a reference to the primary field by stripping away the final five characters of the secondary field's name. I can use the `lastIndexOf()` string method instead of the longer way involving the string's length; but after experiencing so many platform-specific problems with `lastIndexOf()` in Navigator, I decided to play it safe. Finally, the two values are compared, with an appropriate alert displayed if they don't match.

Listing 43-16

Cross-Confirmation Validation

```
// checks an entry against a parallel, duplicate entry to
// confirm that correct data has been entered
// Parallel field name must be the main field name plus "_xcfm"
function isConfirmed() {
    var inputStr = gField.value
    // flag for whether field under test is primary (true) or confirmation field
    var primary = (gField.name.indexOf("_xcfm") == -1)
    if (primary) {
        // clear the confirmation field if primary field is changed
        var xcfmField =
window.frames[gFrame.name].document.forms[0].elements[gField.name + "_ xcfm"]
        var xcfmValue = xcfmField.value
        if (inputStr != xcfmValue) {
            xcfmField.value = ""
            return true
        }
    } else {
        var xcfmField =
window.frames[gFrame.name].document.forms[0].elements[gField.name.substring(0,(gFi
eld.name.length-5))]
        var xcfmValue = xcfmField.value
        if (inputStr != xcfmValue) {
            alert("The main and confirmation entry field contents do not match.
Both fields must have EXACTLY the same content to be accepted by the database.")
            gField.focus()
            gField.select()
            return false
        }
    }
    return true
 }
```

Last-minute check

Every validation event handler is designed to return `true` if the validation succeeds. This comes in handy for the batch validation that performs one final check of the entries triggered by the form's `onSubmit` event handler. This event handler calls a `checkForm()` function and passes the form control object as a parameter. That parameter helps create a reference to the form element that is passed to each validation function.

Because successful validations return `true`, you can nest consecutive validation tests so that the most nested statement of the construction is `return true` because all validations have succeeded. The form's `onSubmit` event handler is as follows:

```
onSubmit="return checkForm(this)"
```

And the following code fragment is an example of a `checkForm()` function. A separate `isDateFormat()` validation function called here checks whether the field contains an entry in the proper format — meaning that it has likely survived the range checking and format shifting of the real-time validation check.

```
function checkForm(form) {
    if (parent.validate(window, form.birthdate, “isDateFormat”)) {
        if (parent.validate(window, form.phone, “isPhone”)) {
            if (parent.validate(window, form.name, “isNotEmpty”)) {
                return true
            }
        }
    }
    return false
}
```

If any one validation fails, the field is given focus and its content is selected (controlled by the individual validation function). In addition, the `checkForm()` function returns `false`. This, in turn, cancels the form submission.

Try it out

Listing 43-17 is a definition for a frameset that not only loads the validation routines described in this section, but also loads a page with a form that exercises the validations in real-time and batch mode just prior to submission. The form appears earlier in this chapter in Figure 43-1.

Listing 43-17
Frameset for Trying validation.js

```
<HTML>
<HEAD>
<TITLE>GiantCo Contractor Database</TITLE>
<SCRIPT LANGUAGE="JavaScript" SRC="validation.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript">
function blank() {
    return "<HTML><BODY BGCOLOR='lightsteelblue'></BODY></HTML>"
}
</SCRIPT>
</HEAD>
    <FRAMESET FRAMEBORDER COLS="20%,80%">
        <FRAME NAME="toc" SRC="javascript:parent.blank()">
        <FRAME NAME="entries" SRC="lst43-18.htm">
    </FRAMESET>
</FRAMESET>
</HTML>
```

The application scenario for the form is the entry of data into a company's contractor database. Some fields are required, and the date field must be cross-confirmed with a second entry of the same data. If the form passes its final validation prior to submission,

the form reloads and you see a readout of the form data that would have been submitted from the previous form had the ACTION been set to a server CGI program URI.

Plan for Data Validation

I devoted this entire chapter to the subject of data validation because it represents the one area of error checking that almost all JavaScript authors should be concerned with. If your scripts (client-side or server-side) perform processing on user entries, you want to prevent script errors at all costs.