CHAPTER 37

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, the incorrect entry is made intentionally 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 bandwidth, server load, and execution speed perspectives 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 field and just before the form is submitted. I recommend you do both.

Real-time validation triggers

The most convenient time to catch an error is immediately after the user has made 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 has entered 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. Text boxes have two potential event handlers for this purpose: `onChange=` and `onBlur=`. I personally avoid `onBlur=` event handlers, especially ones that could 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 must move on past an invalid field will be locked in a seemingly endless loop.

The problem with using onChange= as the validation trigger is that it can be defeated by a user. A change event occurs only when the text of a field has, indeed, changed 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 won'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 coming back to the entry later. Since the onChange= event handler trigger can be defeated, I recommend you also perform batch validation prior to submission.

Batch mode validation

In all scriptable browsers, the onSubmit= event handler cancels the submission if the handler evaluates to return false. You can see an example of this behavior in Listing 21-4 in Chapter 21. That example uses the results of a window.confirm() dialog box to determine the return value of the event handler. But you can use a return value from a series of individual text box validation functions, as well. 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 can be applied to both types of validations.

Designing Filters

The job of writing data validation routines is essentially one of 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 kind of entry is possible that will 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 could make and let through only those 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 retrieval later. 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 user entry is small (a dozen or fewer), a more sensible method may be to avoid the

data-entry problem altogether by turning that field into a select object. 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 are used 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 them into your scripts at a moment's notice and be assured that they will work. For Navigator 3 or later and Internet Explorer 4 or later, 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, a function that checks whether an entry is empty might be named `isEmpty()`. The calling statement for this function would be

```
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 the case in which the entry field is empty. I come back to 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, I provide a few in this chapter that you can both 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 48.

isEmpty()

The first function, shown in Listing 37-1, checks to see if the incoming value is either empty or a null value. Adding a check for a 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 37-1: **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 goes back 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, it means that the `inputStr` value 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 could 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 called the function in the first place.

isPosInteger()

The next function examines each character of the value to make sure that only the numbers from 0 through 9 with no punctuation or other symbols exist. The goal of the function in Listing 37-2 is to weed out any value that is not a positive integer.

Listing 37-2: **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 passed as a parameter. If the value had come directly from a text object, it would already be a string, and the line that forces 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 the input value to be converted 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. It 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 could be nonnumeric. You would 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 true 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.

isInteger()

The next possibility includes the entry of a negative integer value. Listing 37-3 shows that you must add an extra check for a leading negation sign.

Listing 37-3: **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 37-2 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 take care of the possibility, 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` operation at the front of the condition, you ensure the entire condition will short-circuit 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 37-4). All that distinguishes an integer from a floating-point number for data validation purposes is the decimal point.

Listing 37-4: **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 just add a comparison for a decimal (actually, for *not* a decimal) to the condition that compares ASCII values of each character. Such an act assumes that no one would ever enter more than one decimal point into a 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 has 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, 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.

For validations that don't have to accommodate Navigator 2, you can use an even quicker way to test for a valid number. If you pass the value (whether it be a string or a number) through the `parseFloat()` global function (see Chapter 35), the returned value is `NaN` if the conversion is not successful. You can then use the `isNaN()` function to perform the test, as follows:

```
if (isNaN(parseFloat(inputValue))) {
        alert("The value you entered is not a number.")
        return false
}
return true
```

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. An example of such an application-specific variation (extracted from the bonus application in Chapter 48 on the CD-ROM) is shown in Listing 37-5.

Listing 37-5: **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 value is converted to a number (via the `parseInt()` function) so it can be numerically compared 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 that were 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 32), 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 both the presence of any entry and 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 48 on the CD-ROM, 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 created a master validation function consisting of nested `if...else` statements. Each `if` condition calls one of the generalized data validation functions. Listing 37-6 shows the master validation function.

Listing 37-6: **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)) {
                    alert("Sorry, the number you entered is not part of
our database.  Try another three-digit number.")
                    return false
                }
            }
        }
        return true
}
```

This function, in turn, is called by the function that controls most of the work in this application. All it 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 constructed the logic in Listing 37-6 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 trues and falses very carefully.

In the first validation test, being empty 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, being a number 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 a `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 called this function can now proceed with processing, ensured that the value entered by the user will work.

One additional point worth reinforcing, especially for newcomers, is that 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. The original value in the calling function is never touched by these subfunctions — only copies of the original value. Therefore, even after the data validation takes place, the original value is in its original form, ready to go.

Date and Time Validation

You can scarcely open a bigger can of cultural worms than you do when you try 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 the issue.

Trying to write JavaScript that accommodates all of the world's date and time formats for validation would be an enormous, if not wasteful, challenge. My suggestion for querying a user for this kind of information is to either divide the components into individually validated fields (separate text objects for hours and minutes) or, for dates, to make entries select objects.

In the long run, I believe the answer will be a future Java applet or Dynamic HTML component that your scripts will call. The applet will display a clock and calendar on which the user clicks and drags control-panel-style widgets to select dates and times. The values from those settings will then be passed back to your scripts as a valid date object. In the meantime, divide and conquer.

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 being 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 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 the validation functions to be shared by all forms. 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 page 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 must be two-way conversations between a validation function (in the frameset) and a form element (nested in a frame). As you will see in a moment, the mechanism required that the frame containing the form element had to be passed as part of the validation routine so that corrections, automatic formatting, and erroneous field selections could be made from the frameset document's script (that is, 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 was clear that there would be more than two dozen specific types of validations across all the forms. Moreover, multiple programmers would be working on different forms. It would be

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 would contain 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()` was performing validation, and the details of the code would be in the validations.js library file.

To make this idea work meant that in validations.js I had to convert a string name of a validation type into the name of the function that performs the validation. 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 has a label consisting of the name of the validation and a method that invokes the function. Listing 37-7 shows an excerpt of the entire lookup table creation mechanism.

Listing 37-7: **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 constructor assigns a function reference to the object's `doValidate()` method. For each of these assignment statements to work, the function must be defined earlier in the document. You will 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 37-8. 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); second is a reference to the form element itself (using the `this` property); after that come 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 both for a datatype and that the datatype is not empty).

Listing 37-8: **Main Validation Function**

```
// main validation function called by form event handlers
function validate(frame, field, method) {
        gFrame = frame
        gField =  eval("window." + frame.name + ".document.forms[0]." +
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 will 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 an `eval()` function to derive the object reference and assign it to the `gField` global variable.

Next, the script creates an array of all arguments passed to the `validate()` function. A `for` loop starts with index value 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 will 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 often get reused by other named validation functions. Due to 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, 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 were validated for this application, I show several validation functions and, where necessary, their supporting utility functions. As you will see, 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.

U.S. state name

The design specification for forms that required entry of a state of the U.S. called for entry of the two-character abbreviation. A companion field to the right would display the entire state name as user feedback verification. The `onChange=` event handler not only called the validation, but it also fed the focus to the field following the expanded state field so users would be 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 would need both the abbreviation and the full expanded state for this validation, I created an array of all the states, using the state abbreviation as the index label for each entry. Listing 37-9 shows that array creation. If the design were only for Navigator 4, I would have used the literal format for creating such an object to save characters (see Chapter 34).

Listing 37-9: 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"
```

(continued)

Listing 37-9 *(continued)*

```
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 to determine if the user entered a valid two-state abbreviation. Listing 37-10 shows the actual `isUSState()` validation function that puts this array to work.

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

Listing 37-10: **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 = eval("window." + gFrame.name +
".document.forms[0]." + 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 (this would be a great place for a Navigator 4 `switch` construction). An alert message displays the generic alert, plus any special legend if one has been assigned to the `msg` variable. When the user closes the alert, the field will have focus (except for a Navigator 3 bug in UNIX platforms) and its text selected. 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 always 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 then 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 being maintained in the database that is behind the forms. Since all users of this application are familiar with standard date formats in use in the United States, I didn't have to worry about the possibility of cultural variations in date formats. Even so, I wanted the date entry to be accommodating to the common date formats, such as mmddyyyy, mm/dd/yyyy, and mm-dd-yyyy (as well as accommodating two-digit year entries spanning 1930 to 2029).

The plan also called for going further in helping users enter dates within certain ranges. For example, a field that is used for a birthdate (the listings were for medical professionals) should recommend dates starting no more than 90 years from the current date and no less than 20. 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 would be only a recommendation, and not a transaction stopper.

Rather than create separate validation functions for each date field, I created a system of reusable validation functions for each date range (several fields on different forms required the same date ranges). Each one of these individual functions calls a single generic date validation function that handles the date range checking. Listing 37-11 shows a few examples of these individual range-checking functions.

Listing 37-11: **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 created 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 ten 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 always required when the data is unknown.

Next, the functions get the current four-digit year. Given the different ways Navigator 3 treats years before and after 2000, I call a separate utility function to return the four-digit version, as shown here:

```
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 validation for valid dates (for example, making sure that September has only 30 days). Listing 37-12 shows this monster-sized function. Due to the length of this function, I will interlace commentary within the code listing.

Listing 37-12: **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 `replaceString()` function is the same one described in Chapter 26 and 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 not, 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()
```

(continued)

Listing 37-12 *(continued)*

```
            gField.select()
            return false
        }
```

If there are delimiters, I tear apart the string into components for month, day, and year. Because two-digit entries might 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, assuming two-digit entries for month and year, and either 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)
        }
```

Since the `parseInt()` functions would reveal whether any entry is not a number by returning `NaN`, 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, making sure months are entered from 1 to 12 and dates from 1 to 31. I'll 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 either 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 would be 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.

(continued)

Listing 37-12 *(continued)*

```
        } 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, since 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. These functions are shown in Listing 37-13. 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 had entered 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
}
```

Listing 37-13: **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","Augu
st","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 a user to ferret out problems with an invalid entry, the happier those users will be.

Cross-confirmation fields

The final validation type that I'll be covering here is probably not a common request, but it demonstrates how the dispatch mechanism created at the outset

expanded so easily to accommodate this enhanced client request. The situation was that some fields (mostly dates in this application) were deemed critical pieces of data because these data triggered other processes from the database. As a further check to ensure entry of accurate data, a number of values were set up to be entered twice in separate fields, and the fields had 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 placed only one burden on the many programmers working on the forms: while the primary field could be named anything you want (to help alignment with database column names, for example), the secondary field must be named 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, the `isConfirmed()` validation function will compare the fully vetted, properly formatted date in the current field against its parallel entry.

Listing 37-14 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 was the primary or secondary field: If the string "_xcfm" is missing from the field name, then it is the primary.

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, it means the user is entering a new value into the primary field, so the script empties the secondary field to force reentry to verify that the user is entering 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 could have used 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. Then the two values are compared, with an appropriate alert displayed if they don't match.

Listing 37-14: **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 =  eval("window." + gFrame.name +
".document.forms[0]." + gField.name + "_xcfm")
```

```
            var xcfmValue = xcfmField.value
            if (inputStr != xcfmValue) {
               xcfmField.value = ""
               return true
            }
         } else {
            var xcfmField = eval("window." + gFrame.name +
".document.forms[0]." + gField.name.substring(0,(gField.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 is successful. This comes in handy for the batch validation that performs one final check of the entries triggered by the form's `onSubmit=` event handler. The event handler calls a function called `checkForm()` and passes the form 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 selected (controlled by the individual validation function), and the `checkForm()` function returns `false`. This, in turn, cancels the form submission.

Plan for Data Validation

I devoted an 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. Data-entry validation is your last line of defense against user-induced errors.

✦ ✦ ✦