Forms and Form Elements

CHAPTER 9

In This Chapter

What the form object represents

How to access key form object properties and methods

How text, button, and select objects work

How to submit forms from a script

How to pass information from form elements to functions

Most interactivity between a Web page and the user takes place inside a form. That's where a lot of the cool HTML stuff lives: text fields, buttons, checkboxes, option lists, and so on. As you can tell from the by now familiar object hierarchy diagram (refer back to Figure 8-1), a form is always contained by a document. Even so, the document object must be part of the reference to the form and its elements.

The Form Object

A form object can be referenced either by its position in the array of forms contained by a document or by name (if you assign a name to the form in the <FORM> tag). Even if only one form appears in the document, it is a member of a one-element array, and is referenced as follows:

```
document.forms[0]
```

Notice that the array reference uses the plural version of the word, followed by a set of square brackets containing the index number of the element (zero always being first). But if you assign a name to the form, simply plug the form's name into the reference:

```
document.formName
```

Form as object and container

A form has a relatively small set of properties, methods, and event handlers. Almost all of the properties are the same as the attributes for forms. Navigator allows scripts to change these properties under script control (Internet Explorer allows this only starting with Version 4), which gives your scripts potentially significant power to direct the behavior of a form submission in response to user selections on the page.

A form is contained by a document, and it in turn contains any number of elements. All of those interactive elements that let users enter information or make selections belong to the form object. This relationship mirrors the HTML tag

organization, where items such as <INPUT> tags are nested between the <FORM> and </FORM> tag "bookends."

Creating a form

Forms are created entirely from standard HTML tags in the page. You can set attributes for NAME, TARGET, ACTION, METHOD, and ENCTYPE. Each of these is a property of a form object, accessed by all lowercase versions of those words, as in

```
document.forms[0].action
document.formName.action
```

To change any of these properties, simply assign new values to them:

```
document.forms[0].action = "http://www.giantco.com"
```

form.elements[] property

In addition to keeping track of each type of element inside a form, the browser also maintains a list of all elements within a form. This list is another array, with items listed according to the order in which their HTML tags appear in the source code. It is generally more efficient to create references to elements directly, using their names. However, there are times when a script needs to look through all of the elements in a form. This would be especially true if the content of a form were to change with each loading of the page because the number of text fields changes based on the user's browser type.

The following code fragment shows the form.elements[] property at work in a for repeat loop that looks at every element in a form to set the contents of text fields to an empty string. The script cannot simply barge through the form and set every element's content to an empty string, because some elements may be buttons, which don't have a property that can be set to an empty string.

```
var form = window.document.forms[0]
for (var i = 0; i < form.elements.length; i++) {
        if (form.elements[i].type = "text") {
            form.elements[i].value = ""
        }
}
```

In the first statement, I create a variable — form — that holds a reference to the first form of the document. I do this so that when I make many references to form elements later in the script, the length of those references will be much shorter (and marginally faster). I can use the form variable as a shortcut to building references to items more deeply nested in the form.

Next I start looping through the items in the elements array for the form. Each form element has a type property, which reveals what kind of form element it is: text, button, radio, checkbox, and so on. I'm interested in finding elements whose type is "text." For each of those, I set the value property to an empty string.

I will come back to forms later in this chapter to show you how to submit a form without a Submit button and how client-side form validation works.

Text Objects

Each of the four text-related HTML form elements — text, textarea, password, and hidden — is an element in the document object hierarchy. All but the hidden object display themselves in the page, allowing users to enter information. These objects are also used to display text information that changes in the course of using a page (although Dynamic HTML in Internet Explorer 4 also allows the scripted change of body text in a document).

To make these objects scriptable in a page, you do nothing special to their normal HTML tags — with the possible exception of assigning a `NAME` attribute. I strongly recommend assigning unique names to every form element if your scripts will either be getting or setting properties or invoking their methods.

For the visible objects in this category, event handlers are triggered from many user actions, such as giving a field focus (getting the text insertion pointer in the field) and changing text (entering new text and leaving the field). Most of your text field actions will be triggered by the change of text (the `onChange=` event handler). In the level 4 browsers, new event handlers fire in response to individual keystrokes as well.

Without a doubt, the single most used property of a text element is the `value` property. This property represents the current contents of the text element. Its content can be retrieved and set by a script at any time. Content of the `value` property is always a string. This may require conversion to numbers (Chapter 6) if text fields are used to enter values for some math operations.

Text object behavior

Many scripters look to JavaScript to solve what are perceived as shortcomings or behavioral anomalies with text objects in forms. I want to single these out early in your scripting experience so that you are not confused by them later.

First, the font, font size, font style, and text alignment of a text object's content cannot be altered by script, any more than they can be altered by most versions of HTML. This situation will probably change, but for all browsers scriptable in early 1998, these characteristics are not modifiable.

Second, Navigator forms practice a behavior that was recommended as an informal standard by Web pioneers. When a form contains only one text object, a press of the Enter key while the text object has focus automatically submits the form. For two or more fields, another way is needed to submit the form (for example, a Submit button). This one-field submission scheme works well in many cases, such as the search page at most Web search sites. But if you are experimenting with simple forms containing only one field, the form will be submitted with a press of the Enter key. Submitting a form that has no other action or target specified means the page performs an unconditional reload — wiping out any information entered into the form. You can, however, cancel the submission through an `onSubmit=` event handler in the form, as shown later in this lesson.

To demonstrate how a text field's `value` property can be read and written, Listing 9-1 provides a complete HTML page with a single entry field. Its `onChange=` event handler invokes the `upperMe()` function, which converts the text to uppercase. In the `upperMe()` function, the first statement assigns the text object reference to a more convenient variable, `field`. A lot goes on in the second statement of the function. The right side of the assignment statement performs a couple of key tasks. Working from left to right, the reference to the `value` property of the object (`field.value`) evaluates to whatever content is in the text field at that instant. That string is then handed over to one of JavaScript's string functions, `toUpperCase()`, which converts the value to uppercase. Therefore, the entire right side of the assignment statement evaluates to an uppercase version of the field's content. But that, in and of itself, does not change the field's contents. That's why the uppercase string is assigned to the `value` property of the field.

Listing 9-1: **Getting and Setting a Text Object's Value Property**

```
<HTML>
<HEAD>
<TITLE>Text Object value Property</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function upperMe() {
       var field = document.forms[0].converter
       field.value = field.value.toUpperCase()
}
</SCRIPT>
</HEAD>
<BODY>
<FORM onSubmit="return false">
<INPUT TYPE="text" NAME="converter" VALUE="sample"
onChange="upperMe()">
</FORM>
</BODY>
</HTML>
```

Later in this chapter, I will show you how to reduce even further the need for explicit references in functions such as `upperMe()` in Listing 9-1. In the meantime, notice for a moment the `onSubmit=` event handler in the `<FORM>` tag. I will get more deeply into this event handler later in this chapter, but I want to point out the construction that prevents a single-field form from being submitted when the Enter key is pressed.

The Button Object

I have used the button form element in many examples up to this point in the tutorial. The button is one of the simplest objects to script. It has only a few properties, which are rarely accessed or modified in day-to-day scripts. Like the text object, the visual aspects of the button are governed not by HTML or scripts, but by the operating system and browser being used by the page visitor. By far the

most useful event handler of the button object is the onClick= event handler. It fires whenever the user clicks on the button. Simple enough. No magic here.

The Checkbox Object

A checkbox is also a simple form object, but some of the properties may not be entirely intuitive. Unlike the value property of a plain button object (the text of the button label), the value property of a checkbox is any other text you want associated with the object. This text does not appear on the page in any fashion, but the property (initially set via the VALUE tag attribute) might be important to a script that wants to know more about the checkbox.

The key property of a checkbox object is whether or not the box is checked. The checked property is a Boolean value: true if the box is checked, false if not. When you see that a property is a Boolean, that's a clue that the value might be usable in an if or if...else condition expression. In Listing 9-2, the value of the checked property determines which alert box is shown to the user.

Listing 9-2: **The Checkbox Object's Checked Property**

```
<HTML>
<HEAD>
<TITLE>Checkbox Inspector</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function inspectBox() {
        if (document.forms[0].checkThis.checked) {
           alert("The box is checked.")
        } else {
           alert("The box is not checked at the moment.")
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="checkbox" NAME="checkThis">Check here<BR>
<INPUT TYPE="button" VALUE="Inspect Box" onClick="inspectBox()">
</FORM>
</BODY>
</HTML>
```

Checkboxes are generally used as preferences setters, rather than as action inducers. While a checkbox object has an onClick= event handler, a click of a checkbox should never do anything drastic, such as navigate to another page.

The Radio Object

Setting up a group of radio objects for scripting requires a bit more work. To let the browser manage the highlighting and unhighlighting of a related group of

buttons, you must assign the same name to each of the buttons in the group. You can have multiple groups within a form, but each member of the same group must have the same name.

Assigning the same name to a form element forces the browser to manage the elements differently than if they each had a unique name. Instead, the browser maintains an array list of objects with the same name. The name assigned to the group becomes the name of the array. Some properties apply to the group as a whole; other properties apply to individual buttons within the group, and must be addressed via array index references. For example, you can find out how many buttons are in a group by reading the `length` property of the group:

```
document.forms[0].groupName.length
```

But if you want to find out if a particular button is currently highlighted — via the same `checked` property used for the checkbox — you must access the button element individually:

```
document.forms[0].groupName[0].checked
```

Listing 9-3 demonstrates several aspects of the radio button object, including how to look through a group of buttons to find out which one is checked and how to use the `VALUE` attribute and corresponding property for meaningful work.

The page includes three radio buttons and a plain button. Each radio button's `VALUE` attribute contains the full name of one of the Three Stooges. When the user clicks the button, the `onClick=` event handler invokes the `fullName()` function. In that function, the first statement creates a shortcut reference to the form. Next, a `for` repeat loop is set up to look through all of the buttons in the `stooges` radio button group. An `if` construction looks at the `checked` property of each button. When a button is highlighted, the `break` statement bails out of the `for` loop, leaving the value of the `i` loop counter at the number when the loop broke ranks. The alert dialog box then uses a reference to the `value` property of the `i`'th button so that the full name can be displayed in the alert.

Listing 9-3: **Scripting a Group of Radio Objects**

```
<HTML>
<HEAD>
<TITLE>Extracting Highlighted Radio Button</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function fullName() {
        var form = document.forms[0]
        for (var i = 0; i < form.stooges.length; i++) {
            if (form.stooges[i].checked) {
                break
            }
        }
        alert("You chose " + form.stooges[i].value + ".")
}
</SCRIPT>
</HEAD>
```

```
<BODY>
<FORM>
<B>Select your favorite Stooge:</B>
<INPUT TYPE="radio" NAME="stooges" VALUE="Moe Howard" CHECKED>Moe
<INPUT TYPE="radio" NAME="stooges" VALUE="Larry Fine" >Larry
<INPUT TYPE="radio" NAME="stooges" VALUE="Curly Howard" >Curly<BR>
<INPUT TYPE="button" NAME="Viewer" VALUE="View Full Name..."
onClick="fullName()">
</FORM>
</BODY>
</HTML>
```

As you will learn about form elements in later chapters of this book, the browser's tendency to create arrays out of identically named objects of the same type (except for Internet Explorer 3) can be a benefit to scripts that work with, say, columns of fields in an HTML order form.

The Select Object

The most complex form element to script is the select object. As you can see from the object hierarchy diagram (Figure 9-1), the select object is really a compound object: an object that contains an array of option objects. Moreover, the object can be established in HTML to display itself as either a pop-up list or a scrolling list, the latter configurable to accept multiple selections by users. For the sake of simplicity at this stage, this lesson focuses on deployment as a pop-up list allowing only single selections.

Some properties belong to the entire select object; others belong to individual options inside the select object. If your goal is to determine which item was selected by the user, you must use properties of both the select object and the selected option.

The most important property of the select object itself is the `selectedIndex` property, accessed as follows:

```
document.form[0].selectName.selectedIndex
```

This value is the index number of the item that is currently selected. As with most index counting schemes in JavaScript, the first item (the one at the top of the list) has an index of zero. The `selectedIndex` value is critical for letting you access properties of the selected item. Two important properties of an option item are `text` and `value`, accessed as follows:

```
document.forms[0].selectName.options[n].text
document.forms[0].selectName.options[n].value
```

The `text` property is the string that appears on screen in the select object. It is unusual for this information to be exposed as a form property, because in the HTML that generates a select object, the text is defined outside of the `<OPTION>` tag. But inside the `<OPTION>` tag you can set a `VALUE` attribute, which, like the radio buttons shown earlier, lets you associate some hidden string information with each visible entry in the list.

To extract the `value` or `text` property of a selected option most efficiently, you can use the select object's `selectedIndex` property as an index value to the option. References for this kind of operation get pretty long, so take the time to understand what's happening here. In the following function, the first statement creates a shortcut reference to the select object. The `selectedIndex` property of the select object is then substituted for the index value of the options array of that same object:

```
function inspect() {
        var list = document.forms[0].choices
        var chosenItem = list.options[list.selectedIndex].text
}
```

To bring a select object to life, use the `onChange=` event handler. As soon as a user makes a new selection in the list, this event handler runs the script associated with that event handler (except for Windows versions of Navigator 2, whose `onChange=` event handler doesn't work for select objects). Listing 9-4 shows a common application for a select object. Its text entries describe places to go in and out of a Web site, while the `VALUE` attributes hold the URLs for those locations. When a user makes a selection in the list, the `onChange=` event handler triggers a script that extracts the `value` property of the selected option, and assigns that value to the location object to effect the navigation. Under JavaScript control, this kind of navigation doesn't need a separate Go button on the page.

Listing 9-4: **Navigating with a Select Object**

```
<HTML>
<HEAD>
<TITLE>Select Navigation</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function goThere() {
        var list = document.forms[0].urlList
        location = list.options[list.selectedIndex].value
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Choose a place to go:
<SELECT NAME="urlList" onChange="goThere()">
        <OPTION SELECTED VALUE="index.html">Home Page
        <OPTION VALUE="store.html">Shop Our Store
        <OPTION VALUE="policies">Shipping Policies
        <OPTION VALUE="http://www.yahoo.com">Search the Web
</SELECT>
</FORM>
</BODY>
</HTML>
```

There is much more to the select object, including the ability to change the contents of a list in newer browsers. Chapter 24 covers the object in depth.

Passing Form Data and Elements to Functions

In all of the examples so far in this lesson, when an event handler invoked a function that worked with form elements, the form or form element was explicitly referenced in the function. But valuable shortcuts do exist for transferring information about the form or form element directly to the function without dealing with those typically long references that start with the window or document object level.

JavaScript features a keyword — `this` — that always refers to whatever object contains the script in which the keyword is used. Thus, in an `onChange=` event handler for a text field, you can pass a reference to the text object to the function by inserting the `this` keyword as a parameter to the function:

```
<INPUT TYPE="text" NAME="entry" onChange="upperMe(this)">
```

At the receiving end, the function defines a parameter variable that turns that reference into a variable that the rest of the function can use:

```
function upperMe(field) {
        statement[s]
}
```

The name you assign to the parameter variable is purely arbitrary, but it is helpful to give it a name that expresses what the reference is. Importantly, this reference is a "live" connection back to the object. Therefore, statements in the script can get and set property values at will.

For other functions, you may wish to receive a reference to the entire form, rather than just the object calling the function. This is certainly true if the function is called by a button whose function needs to access other elements of the same form. To pass the entire form, you reference the `form` property of the object, still using the `this` keyword:

```
<INPUT TYPE="button" VALUE="Click Here" onClick="inspect(this.form)">
```

The function definition should then have a parameter variable ready to be assigned to the form object reference. Again, the name of the variable is up to you. I tend to use the variable name `form` as a way to remind me exactly what kind of object is referenced.

```
function inspect(form) {
        statement[s]
}
```

Listing 9-5 demonstrates passing both an individual form element and the entire form in the performance of two separate acts. This page makes believe it is connected to a database of Beatles songs. When you click the Process Data button, it passes the form object, which the `processData()` function uses to access the group of radio buttons inside a `for` loop. Additional references using the passed

form object extract the `value` properties of the selected radio button and the text field.

The text field has its own event handler, which passes just the text field to the `verifySong()` function. Notice how short the reference is to reach the `value` property of the `song` field inside the function.

Listing 9-5: **Passing a Form Object and Form Element to Functions**

```
<HTML>
<HEAD>
<TITLE>Beatle Picker</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function processData(form) {
        for (var i = 0; i < form.Beatles.length; i++) {
            if (form.Beatles[i].checked) {
                break
            }
        }
        // assign values to variables for convenience
        var beatle = form.Beatles[i].value
        var song = form.song.value
        alert("Checking whether " + song + " features " + beatle +
"...")
}

function verifySong(entry) {
        var song = entry.value
        alert("Checking whether " + song + " is a Beatles tune...")
}
</SCRIPT>
</HEAD>

<BODY>
<FORM onSubmit="return false">
Choose your favorite Beatle:
<INPUT TYPE="radio" NAME="Beatles" VALUE="John Lennon" CHECKED>John
<INPUT TYPE="radio" NAME="Beatles" VALUE="Paul McCartney">Paul
<INPUT TYPE="radio" NAME="Beatles" VALUE="George Harrison">George
<INPUT TYPE="radio" NAME="Beatles" VALUE="Ringo Starr">Ringo<P>

Enter the name of your favorite Beatles song:<BR>
<INPUT TYPE="text" NAME="song" VALUE = "Eleanor Rigby"
onChange="verifySong(this)"><P>
<INPUT TYPE="button" NAME="process" VALUE="Process Request..."
onClick="processData(this.form)">
</FORM>
</BODY>
</HTML>
```

Get to know the usage of the `this` keyword in passing form and form element objects to functions. The technique not only saves you typing in your code, but assures accuracy in references to those objects.

Submitting Forms

If you have worked with Web pages and forms before, you are familiar with how simple it is to add a Submit-style button that sends the form to your server. However, design goals for your page may rule out the use of ugly system buttons. If you'd rather display a pretty image, the link tag surrounding that image should use the `javascript:` URL technique to invoke a script that submits the form.

The scripted equivalent of submitting a form is the form object's `submit()` method. All you need in the statement is a reference to the form and this method, as in

```
document.forms[0].submit()
```

One limitation might inhibit your plans to secretly have a script send you an e-mail message from every visitor who comes to your Web site. If the form's `ACTION` attribute is set to a `mailTo:` URL, JavaScript will not pass along the `submit()` method to the form. Of course, Internet Explorer does not allow e-mailing of forms through any machinations.

Before a form is submitted, you may wish to perform some last-second validation of data in the form or other scripting (for example, changing the form's `action` property based on user choices). This can be done in a function invoked by the form's `onSubmit=` event handler. Specific validation routines are beyond the scope of this tutorial (although explained in substantial detail in Chapter 37), but I want to show you how the `onSubmit=` event handler works.

In all but the first generation of scriptable browsers from Microsoft and Netscape, you can let the results of a validation function cancel a submission if the validation shows some data to be incorrect or a field is empty. To control submission, the `onSubmit=` event handler must evaluate to `return true` (to allow submission to continue) or `return false` (to cancel submission). This is a bit tricky at first, because it involves more than just having the function called by the event handler return `true` or `false`. The `return` keyword must also be part of the final evaluation.

Listing 9-6 shows a page with a simple validation routine that ensures all fields have something in them before allowing submission to continue. Notice how the `onSubmit=` event handler (which passes the form object as a parameter — in this case the `this` keyword points to the form object) includes the `return` keyword before the function name. When the function returns its `true` or `false` value, the event handler evaluates to the requisite `return true` or `return false`.

Listing 9-6: **Last-minute Checking before Form Submission**

```
<HTML>
<HEAD>
<TITLE>Validator</TITLE>
```

(continued)

Listing 9-6 *Continued*

```
<SCRIPT LANGUAGE="JavaScript">
function checkForm(form) {
        for (var i = 0; i < form.elements.length; i++) {
            if (form.elements[i].value == "") {
                alert("Fill out ALL fields.")
                return false
            }
        }
        return true
}
</SCRIPT>
</HEAD>

<BODY>
<FORM onSubmit="return checkForm(this)">
Please enter all requested information:<BR>
First Name:<INPUT TYPE="text" NAME="firstName" ><BR>
Last Name:<INPUT TYPE="text" NAME="lastName" ><BR>
Rank:<INPUT TYPE="text" NAME="rank" ><BR>
Serial Number:<INPUT TYPE="text" NAME="serialNumber" ><BR>

<INPUT TYPE="submit">
</FORM>
</BODY>
</HTML>
```

One quirky bit of behavior involving the `submit()` method and `onSubmit=` event handler needs explanation. While you might think (and logically so, in my opinion) that the `submit()` method would be exactly the scripted equivalent of a click of a real Submit button, it's not. In Navigator, the `submit()` method does not cause the form's `onSubmit=` event handler to fire at all. If you want to perform validation on a form submitted via the `submit()` method, invoke the validation in the script that calls the `submit()` method.

So much for the basics of forms and form elements. In the next lesson, you step away from HTML for a moment to look at more advanced JavaScript core language items: strings, math, and dates.

Exercises

1. Rework Listings 9-1, 9-2, 9-3, and 9-4 so that the script functions all receive the most efficient form or form element references from the invoking event handler.
2. Modify Listing 9-6 so that instead of the submission being made by a Submit button, the submission is performed from a hyperlink. Be sure to include the form validation in the process.

3. In the following HTML tag, what kind of information do you think is being passed with the event handler? Write a function that displays in an alert dialog box the information being passed.

```
<INPUT TYPE="text" NAME="phone" onChange="format(this.value)">
```

4. A document contains two forms, named `specifications` and `accessories`. In the `accessories` form is a field named `acc1`. Write two different statements that set the contents of that field to `Leather Carrying Case`.

5. Create a page that includes a select object to change the background color of the current page. The property that needs to be set is `document.bgColor`, and the three values you should offer as options are `red`, `yellow`, and `green`. In the select object, the colors should be displayed as Stop, Caution, and Go. Note: If you are using a Macintosh or UNIX version of Navigator, you must be using Version 4 or later for this exercise.

✦ ✦ ✦