CHAPTER 13

JavaScript Essentials

In This Chapter

Understanding the document object model

Where scripts go in your documents

How to develop a single application for multiple browser versions

The purpose of most JavaScript scripts is to make a Web page interactive, whether or not a program is running on the server to enhance that interactivity. Making a page interactive means tracking user action and responding with some visible change on the page. The avenues for communication between user and script include familiar on-screen elements, such as fields and buttons, as well as dynamic content in level 4 browsers.

To assist the scripter in working with these elements, the browser and JavaScript implements them as software objects. These objects have *properties* that often define the visual appearance of the object. Objects also have *methods*, which are the actions or commands that an object can carry out. Finally, these objects have *event handlers* that trigger the scripts you write in response to an action in the document (usually instigated by the user). This chapter begins by helping you understand that the realm of possibilities for each JavaScript object is the key to knowing how you can use JavaScript to convert an idea into a useful Web page. From there you see how to add scripts to an HTML page and learn several techniques to develop applications for a wide range of browsers — including nonscriptable browsers.

Language and Document Objects

As described briefly in Chapter 2, the environment you will come to know as JavaScript scripting is composed of two quite separate pieces: the core JavaScript language and the document object models of "things" you script with the language. By separating the core language from the objects it works with, the language's designers created a programming language that can be linked to a variety of object collections for different environments. In fact, even within Web applications, the same core JavaScript language is applied to the browser document object model and the server-side object model for processing requests and communicating with databases. Everything an author knows about the core language can be applied equally to client-side and server-side JavaScript.

Core language standard — ECMAScript

Although the JavaScript language was first developed by Netscape, Microsoft incorporated its version of the language, called JScript, in Internet Explorer 3. The core language part of JScript is essentially identical to that of JavaScript in Netscape Navigator, albeit usually one generation behind. Where the two browsers greatly differ, however, is in their document object models.

As mentioned in Chapter 2, standards efforts are in various stages to address potential incompatibilities among browsers. The core language is the first component to achieve standard status. Through the European standards body called ECMA, a formal standard for the language has been agreed to and published. This language, dubbed ECMAScript, is roughly the same as JavaScript 1.1 in Netscape Navigator 3. The standard defines how various data types are treated, how operators work, what various data-specific syntax looks like, and other language characteristics. You can view the ECMA-262 specification at `http://www.ecma.ch`. If you are a student of programming languages, you will find the document fascinating; if you simply want to script your pages, you will probably find the minutia mind-boggling.

Both Netscape and Microsoft have pledged to make their browsers compliant with this standard. Because of the fortuitous timing of the completion of the standard and the formal release of Internet Explorer 4, the latest version of the Microsoft browser is the first to comply with ECMAScript (with a couple minor omissions). Navigator 3 and 4 are already very close to the standard.

Document object standard

A recommendation for a document object model standard has yet to be finalized as this book goes to press. A comparison of the object models in Navigator 4 and Internet Explorer 4 reveals what appears to be diverging strategies — at least for now. While Navigator currently adheres to a fairly simple model, as previewed in earlier chapters and discussed in more detail later in this chapter, Internet Explorer 4 defines an object model that is at once extremely flexible and significantly more complicated to work with.

A key beneficiary of the added complexity in Internet Explorer 4's object model is the scripter who wishes to make a lot of changes to content on the fly — beyond the kind of object positioning possible with standard Cascading Style Sheet-Positioning (see Chapter 41). Internet Explorer 4's page rendering engine is fast enough to reflow content when a script changes the text inside a tag or the size of an object. Virtually everything that can be defined in HTML with a tag is an object in Internet Explorer 4. Moreover, properties for each of these objects include the body text that appears in the document and the very HTML for the element itself — all of which can be modified on the fly without reloading the document.

Standards efforts on the document object model are under way under the auspices of the W3C, the same organization that oversees the HTML specification. While both Netscape and Microsoft again pledge to support industry standards, each company is taking its own route until this particular standard is established, and both are not bashful about including their own proprietary extensions.

Later in this chapter, I say more about how to address issues of incompatibility among browser versions and brands. Suffice it to say that because this book

focuses on Netscape's implementation of the JavaScript language, it also focuses on its document object model. Throughout the rest of this chapter, when I refer to objects and the object hierarchy, I am talking about document objects, as opposed to core language objects.

The Object Hierarchy

In the tutorial chapters of Part II, I speak of the JavaScript *object hierarchy*. In other object-oriented languages, object hierarchy plays a much greater role than it does in JavaScript (you don't have to worry about related terms, such as classes, inheritance, and instances, in JavaScript). Even so, you cannot ignore the hierarchy concept, because much of your code relies on your ability to write *references* to objects that depend on their position within the hierarchy.

Calling these objects *JavaScript objects* is not entirely accurate either. These are really browser document objects: you just happen to use the JavaScript language to bring them to life. Technically speaking, JavaScript objects are the ones that apply to data types and other core language objects separate from the document. These objects are covered in many chapters, beginning with Chapter 26. The more you can keep document and core language objects separate in your head, the more quickly you'll be able to deal with browser brand compatibility issues.

Hierarchy as road map

For the programmer, the primary role of the document object hierarchy is to provide scripts with a way to reference a particular object among all the objects that a browser window can contain. The hierarchy acts as a road map the script can use to know precisely which object to address.

Consider, for a moment, a scene in which you and your friend Tony are in a high school classroom. It's getting hot and stuffy as the afternoon sun pours in through the wall of windows on the west side of the room. You say to Tony, "Would you please open a window?" and motion your head toward a particular window in the room. In programming terms, you've issued a command to an object (whether or not Tony appreciates the comparison). This human interaction has many advantages over anything you can do in programming. First, by making eye contact with Tony before you spoke, he knew that he was the intended recipient of the command. Second, your body language passed along some parameters with that command, pointing ever so subtly to a particular window on a particular wall.

If, instead, you were in the principal's office using the public address system, and you broadcasted the same command, "Would you please open a window?" no one would know what you meant. Issuing a command without directing it to an object is a waste of time, because every object would think, "That can't be meant for me." To accomplish the same goal as your one-on-one command, the broadcasted command would have to be something like "Would Tony Jeffries in Room 312 please open the middle window on the west wall?"

Let's convert this last command to JavaScript "dot" syntax form (see Chapter 4). Recall from the tutorial that a reference to an object starts with the most global point of view and narrows to the most specific point of view. From the point of view of the principal's office, the location hierarchy of the target object is

```
room312.Jeffries.Tony
```

You could also say that Tony's knowledge about how to open a window is one of Tony's methods. The complete reference to Tony and his method then becomes

```
room312.Jeffries.Tony.openWindow()
```

Your job isn't complete yet. The method requires a parameter detailing which window to open. In this case, the window you want is the middle window of the west wall of Room 312. Or, from the hierarchical point of view of the principal's office, it becomes

```
room312.westWall.middleWindow
```

This object road map is the parameter for Tony's `openWindow()` method. Therefore, the entire command that comes over the PA system should be

```
room312.Jeffries.Tony.openWindow(room312.westWall.middleWindow)
```

If, instead of barking out orders while sitting in the principal's office, you were attempting the same task via radio from an orbiting space shuttle to all the inhabitants on Earth, imagine how laborious your object hierarchy would be. The complete reference to Tony's `openWindow()` method and the window that you want opened would have to be mighty long to distinguish the desired objects from the billions of objects within the space shuttle's view.

The point is that the smaller the scope of the object-oriented world you're programming, the more you can assume about the location of objects. For JavaScript, the scope is no wider than the browser itself. In other words, every object that a JavaScript script can work with is within the browser application. A script does not access anything about your computer hardware, operating system, or desktop, or any other stuff beyond the browser program.

The JavaScript document object road map

Figure 13-1 shows the complete JavaScript document object hierarchy as implemented in Netscape Navigator 4. Notice that the window object is the topmost object in the entire scheme. Everything you script in JavaScript is in the browser's window — whether it be the window itself or a form element.

Of all the objects shown in Figure 13-1, you are likely to work most of the time with the ones that appear in boldface. Objects whose names appear in italics are synonyms for the window object, and are used only in some circumstances.

Pay attention to the shading of the concentric rectangles. Every object in the same shaded area is at the same level relative to the window object. When a link from an object extends to the next darker shaded rectangle, that object contains all the objects in darker areas. There exists at most one of these links between levels. A window object contains a document object; a document object contains a form object; a form object contains many different kinds of form elements.

Study Figure 13-1 to establish a mental model for the scriptable elements of a Web page. After you script these objects a few times, the object hierarchy will become second nature to you — even if you don't necessarily remember every detail (property, method, and event handler) of every object. At least you will know where to look for information.

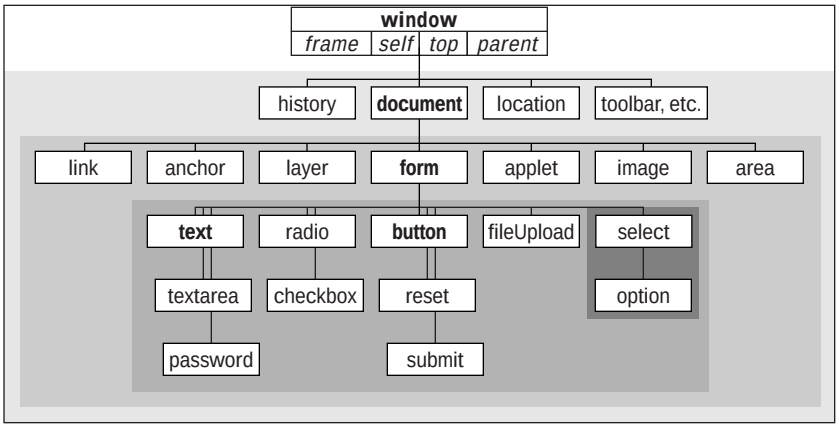

Figure 13-1: The JavaScript document object hierarchy in Navigator 4

Creating JavaScript Objects

Most of the objects that a browser creates for you are established when an HTML document loads into the browser. The same kind of HTML code you've used to create links, anchors, and input elements tells a JavaScript-enhanced browser to create those objects in memory. The objects are there whether or not your scripts call them into action.

The only visible differences to the HTML code for defining those objects are the one or more optional attributes specifically dedicated to JavaScript. By and large, these attributes specify the event you want the user interface element to react to and what JavaScript should do when the user takes that action. By relying on the document's HTML code to perform the object generation, you can spend more time figuring out how to do things with those objects or have them do things for you.

Bear in mind that objects are created *in their load order*, which is why you should put most, if not all, deferred function definitions in the document's Head. And if you're creating a multiframe environment, a script in one frame cannot communicate to another frame's objects until both frames load. This trips up a lot of scripters who are creating multiframe and multiwindow sites (more in Chapter 14).

Object Properties

A property generally defines a particular, current setting of an object. The setting may reflect a visible attribute of an object, such as a document's background color; it may also contain information that is not so obvious, such as the action and method of a form when it is submitted.

Document objects have most of their properties assigned by the attribute settings of the HTML tags that generate the objects. Thus, a property may be a string (for example, a name) or a number (for example, a size). A property can also be an array, such as an array of images contained by a document. If the HTML does not include all attributes, the browser usually fills in a default value for both the attribute and the corresponding JavaScript property.

A note to experienced object-oriented programmers

Although the JavaScript hierarchy appears to have a class-subclass relationship, many of the traditional aspects of a true, object-oriented environment don't apply to the document object model. The JavaScript document object hierarchy is a *containment* hierarchy, not an *inheritance* hierarchy. No object inherits properties or methods of an object higher up the chain. Nor is there any automatic message passing from object to object in any direction. Therefore, you cannot invoke a window's method by sending a message to it via a document or form object. All object references must be explicit.

Predefined document objects are generated only when the HTML code containing their definitions loads into the browser. Many properties, methods, and event handlers cannot be modified once you load the document into the browser. In Chapter 34, you learn how to create your own objects, but those objects cannot be the type that present new visual elements on the page that go beyond what HTML, Java applets, and plug-ins can portray.

When used in script statements, property names are case-sensitive. Therefore, if you see a property name listed as `bgColor`, you must use it in a script statement with that exact combination of lowercase and uppercase letters. But when you are setting an initial value of a property by way of an HTML attribute, the attribute name (like all of HTML) is not case-sensitive. Thus, `<BODY BGCOLOR="white">` and `<body bgcolor="white">` both set the same property value.

Each property determines its own read-write status. Some objects are read-only, whereas others can be changed on the fly by assigning a new value to them. For example, to put some new text into a text object, you assign a string to the object's `value` property:

```
document.forms[0].phone.value = "555-1212"
```

Once an object contained by the document exists (that is, its HTML has loaded into the document), you can also add one or more custom properties to that object. This can be helpful if you wish to associate some additional data with an object for later retrieval. To add such a property, simply specify it in the same statement that assigns a value to it:

```
document.forms[0].phone.delimiter = "-"
```

Any property you set survives as long as the document remains loaded in the window and scripts do not overwrite the object.

Object Methods

An object's method is a command that a script can give to that object. Some methods return values, but that is not a prerequisite for a method. Also, not every object has methods defined for it. In a majority of cases, invoking a method from a script causes some action to take place. It may be an obvious action, such as resizing a window, or something more subtle, such as processing a mouse click.

All methods have parentheses after them and always appear at the end of an object's reference. When a method accepts or requires parameters, their values go inside the parentheses.

While an object has its methods predefined by the object model, you can also assign one or more additional methods to an object that already exists (that is, its HTML has loaded into the document). To do this, a script in the document (or in another window or frame accessible by the document) must define a JavaScript function, then assign that function to a new property name of the object. In the following Navigator 4 script example, the `fullScreen()` function invokes one window object method and adjusts two window object properties. By assigning the function reference to the new `window.maximize` property, I have defined a `maximize()` method for the window object. Thus, a button's event handler can call that method directly.

```
function fullScreen() {
        this.moveTo(0,0)
        this.outerWidth = screen.availWidth
        this.outerHeight = screen.availHeight
}
window.maximize = fullScreen
...
<INPUT TYPE="button" VALUE="Maximize Window"
onClick="window.maximize()">
```

Object Event Handlers

Event handlers specify how an object reacts to an event, whether the event is triggered by a user action (for example, a button click) or a browser action (for example, the completion of a document load). Going back to the earliest JavaScript-enabled browser, event handlers were defined inside HTML tags as extra attributes. They included the name of the attribute, followed by an equals sign (working as an assignment operator) and a string containing the script statement(s) or function(s) to execute when the event occurs. Event handlers also have other forms. In Navigator 3 and later, event handlers have corresponding methods for their objects and every event handler is a property of its object.

Event handlers as methods

Consider a button object whose sole event handler is `onClick=`. This means whenever the button receives a click event, the button triggers the JavaScript expression or function call assigned to that event handler in the button's HTML definition:

```
<INPUT TYPE="button" NAME="clicker" VALUE="Click Me" onClick="doIt()">
```

Normally, that click event is the result of a user physically clicking on the button in the page. You can also trigger the event handler with a script by calling the event handler as if it were a method of the object:

```
document.formName.clicker.onclick()
```

Notice that when summoning an event handler as a method, the method name is all lowercase, regardless of the case used in the event handler attribute within the original HTML tag. This lowercase reference is a requirement.

Invoking an event handler this way is different from using a method to replicate the physical action denoted by the event. For example, imagine a page containing three simple text fields. One of those fields has an `onFocus=` event handler defined for it. Physically tabbing to or clicking in that field brings focus to the field and thereby triggers its `onFocus=` event handler. If the field did not have focus, a button could invoke that field's `onFocus=` event handler by referencing it as a method:

```
document.formName.fieldName.onfocus()
```

This scripted action does not bring physical focus to the field. The field's own `focus()` method, however, does that under script control.

A byproduct of an event handler's capability to act like a method is that you can define the action of an event handler by defining a function with the event handler's name. For example, instead of specifying an `onLoad=` event handler in a document's `<BODY>` tag, you can define a function like this:

```
function onload() {
        statements
}
```

This capability is particularly helpful if you want event handler actions confined to a script running in Navigator 3 or later. Your scripts won't require special traps for Navigator 2 or Internet Explorer 3.

Event handlers as properties

Although event handlers are normally defined in an object's HTML tag, you also have the power to change the event handler from Navigator 3 onward, the reason being that an event handler is treated like any other object property whose value you can both retrieve and modify. The value of an event handler property looks like a function definition. For example, given this HTML definition:

```
<INPUT TYPE="text" NAME="entry" onFocus="doIt()">
```

the value of the object's `onfocus` (all lowercase) property is

```
function onfocus() {
        doIt()
}
```

You can, however, assign an entirely different function to an event handler by assigning a function reference to the property. Such references don't include the parentheses that are part of the function's definition (you see this again much later in Chapter 34, when you assign functions to object properties).

Using the same text field definition you just looked at, assign a different function to the event handler, because based on user input elsewhere in the document, you want the field to behave differently when it receives the focus. If you define a function like this:

```
function doSomethingElse() {
        statements
}
```

you can then assign the function to the field with this assignment statement:

```
document.formName.entry.onfocus = doSomethingElse
```

Because the new function reference is written in JavaScript, you must observe case. In Navigator 3, the handler name must be all lowercase; in Navigator 4, you can also use the more familiar interCap version (for example, `document.formName.entry.onFocus = doSomethingElse`).

Be aware, however, that as with several settable object properties that don't manifest themselves visually, any change you make to an event handler property disappears with a document reload. Therefore, I advise you not to make such changes except as part of a script that also invokes the event handler like a method.

Because every event handler operates as both property and method, I won't list these properties and methods as part of each object's definition in the next chapters. You can be assured this feature works for every JavaScript object that has an event handler from Navigator 3 onward.

Embedding Scripts in Documents

JavaScript offers three ways to include scripts or scripted elements in your documents — the `<SCRIPT>` tag, the external library, and the JavaScript entity for HTML attributes, discussed in the following sections. Not all approaches are available in all versions of Navigator, but you do have everything at your disposal for Navigator 3 onward and for some versions of Internet Explorer 3 onward.

<SCRIPT> tags

The simplest and most compatible way to include script statements in a document is inside a `<SCRIPT>...</SCRIPT>` tag set. You can have any number of such tag sets in your document. For example, you can define some functions in the Head section to be called by event handlers in HTML tags within the Body section. Another tag set could be within the Body to dynamically write part of the content of the page. Place only script statements and comments between the start and end tags of the tag set. Do not place any HTML tags inside unless they are part of a parameter to a `document.write()` statement that creates content for the page.

Every opening `<SCRIPT>` tag should specify the `LANGUAGE` attribute. Because the `<SCRIPT>` tag is a generic tag indicating that the contained statements are to be interpreted as executable script and not renderable HTML, the tag is designed to accommodate any scripting language the browser knows about.

All scriptable browsers (from Navigator 2 onward and Internet Explorer 3 onward) recognize the `LANGUAGE="JavaScript"` attribute setting. However, more recent browsers typically acknowledge additional versions of JavaScript or, in the case of Internet Explorer, other languages. For example, the JavaScript interpreter built into Navigator 3 knows the JavaScript 1.1 version of the language; Navigator 4 and Internet Explorer 4 include the JavaScript 1.2 version. For versions beyond the

original JavaScript, you specify the language version by appending the version number after the language name, without any spaces, as in

```
<SCRIPT LANGUAGE="JavaScript1.1">
<SCRIPT LANGUAGE="JavaScript1.2">
```

How you use these later-version attributes depends on the content of the scripts and your intended audience. For example, while Navigator 4 is JavaScript 1.2–compatible, it works with all versions of the JavaScript `LANGUAGE` attribute. Even features of the language new in JavaScript 1.2 will be executed if the `LANGUAGE` attribute is set to only `"JavaScript"`. On rare occasions (detailed in the succeeding chapters) the behavior of the language change, in a Navigator 4 browser if you specify the JavaScript 1.2 language instead of an earlier version.

Writing scripts for a variety of browser versions requires a bit of care, especially when the scripts may contain language features available only in newer browsers. As demonstrated in an extensive discussion about browser detection later in this chapter, there may be a need to include multiple versions of a script function, each in its own `<SCRIPT>` tag with a different `LANGUAGE` attribute.

JavaScript versus JScript and VBScript

As previously explained, Internet Explorer's version of JavaScript is called JScript. As a result, Internet Explorer's default script language is JScript. While Internet Explorer acknowledges the `LANGUAGE="JavaScript"` attribute, Netscape Navigator ignores the `LANGUAGE="JScript"` attribute. Therefore, if you are writing scripts that must work in both Internet Explorer and Netscape Navigator, you can specify one language (`"JavaScript"`) and count on both browsers interpreting the code correctly (assuming you take into account the other compatibility issues).

An entirely different issue is Internet Explorer's other scripting language, VBScript. This language is a derivative of Visual Basic and offers no interoperability with JavaScript scripts (even though both languages work with the same Internet Explorer object model). You can mix scripts from both languages in the same document, but their tag sets must be completely separate, with the `LANGUAGE` attributes clearly specifying the language for the `<SCRIPT>` tag.

Hiding script statements from older browsers

As more versions of scriptable browsers spread among the user community, the installed base of older, nonscriptable browsers diminishes. However, public Web sites can still attract a variety of browsers that date back to the World Wide Web Stone Age (before A.D.1996).

Browsers from those olden days do not know about the `<SCRIPT>` tag. Normally, browsers ignore tags they don't understand. That's fine when a tag is just one line of HTML, but a `<SCRIPT>` tag sets off any number of script statement lines in a document. Old browsers don't know to expect a closing `</SCRIPT>` tag. Therefore, their natural inclination is to render any lines they encounter after the opening `<SCRIPT>` tag. Unfortunately, this places script statements squarely in the document — surely to confuse anyone who sees such gibberish on the page.

You can, however, exercise a technique that tricks most older browsers into ignoring the script statements. The trick is surrounding the script statements — inside the `<SCRIPT>` tag set — with HTML comment markers. An HTML comment

begins with the sequence <!-- and ends with -->. Therefore, you should embed these comment sequences in your scripts according to the following format:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
script statements here
//-->
</SCRIPT>
```

JavaScript interpreters also know to ignore a line that begins with the HTML beginning comment sequence, but the interpreter needs a little help with the ending sequence. The close of the HTML comment starts with a JavaScript comment sequence (//). This tells JavaScript to ignore the line; but a nonscriptable browser sees the ending HTML symbols and begins rendering the page with the next HTML tag or other text in the document. Since an older browser won't know what the </SCRIPT> tag is, the tag is ignored, and rendering begins after that.

Even with this subterfuge, not all browsers handle HTML comment tags gracefully. Some older America Online browsers will display the script statements no matter what you do. Fortunately, these browsers are disappearing.

If your pages are being designed for public access, include these HTML comment lines in all your <SCRIPT> tag sets. Make sure they go *inside* the tags, not outside. Also note that most of the script examples in this part of the book do not include these comments, for the sake of saving space in the listings.

Hiding scripts entirely?

It may be misleading to say that this HTML comment technique hides scripts from older browsers. In truth, the comments hide the scripts from being rendered by the browsers. The tags and script statements, however, are still downloaded to the browser and appear in the source when viewed by the user.

A common wish among authors is to truly hide scripts from visitors to a page. Client-side JavaScript must be downloaded with the page and is, therefore, visible in the source view of pages. There are, of course, some tricks you can implement that may disguise client-side scripts from prying eyes. The most easily implemented technique is to let the downloaded page contain no visible elements, only scripts that assemble the page that the visitor sees. Source code for such a page is simply the HTML for the page. But that page will not be interactive, because no scripting would be attached unless it were written as part of the page — defeating the goal of hiding scripts. If you are worried about your scripts being "stolen" by other scripters, a perfect way to prevent it is to include a copyright notification in your page's source code. Not only are your scripts visible to the world, but so, too, would a thief's scripts be. This way you can easily see if someone has lifted your scripts verbatim.

Script libraries

If you do a lot of scripting or script a lot of pages for a complex Web application, you will certainly develop some functions and techniques that could be used for several pages. Rather than duplicate the code in all of those pages (and go through the nightmare of making changes to all copies for new features or bug fixes), you can create separate script library files and link them to your pages.

Such an external script file contains nothing but JavaScript code — no <SCRIPT> tags, no HTML. The script file you create must be a text-only file, such as an HTML source file, but its filename must end with the two-character extension .js. To instruct the browser to load the external file at a particular point in your regular HTML file, you add a SRC attribute to the <SCRIPT> tag, as follows:

```
<SCRIPT LANGUAGE="JavaScript" SRC="hotscript.js"></SCRIPT>
```

This kind of tag should go at the top of the document so it loads before any other in-document <SCRIPT> tags load. If you load more than one external library, include a series of these tag sets at the top of the document.

Note two features about this tag construction: First, the <SCRIPT> . . . </SCRIPT> tag pair is required, even though nothing appears between them. You can mix <SCRIPT> tag sets that specify external libraries and include in-document scripts in the same document.

How you reference the source file in the SRC attribute depends on its physical location and your HTML coding style. In the preceding example, the .js file is in the same directory as the HTML file containing the tag. But if you want to refer to an absolute URL, the protocol for the file is http://, just like with an HTML file:

```
<SCRIPT LANGUAGE="JavaScript" SRC="http://www.cool.com/hotscript.js">
</SCRIPT>
```

A very important prerequisite for using script libraries with your documents is your Web server software must know how to map files with the .js extension to a MIME type of application/x-javascript. If you plan to deploy JavaScript in this manner, be sure to test a sample on your Web server beforehand and arrange for any server adjustments that may be necessary.

When a user chooses Document Source from Navigator's View menu, code from a .js file does not appear in the window, even though the browser treats the loaded script as part of the current document. However, the name or URL of the .js file is plainly visible (displayed exactly as you wrote it for the SRC attribute). Anyone can then open that file with the browser (using the http:// protocol) and view the .js file's source code. In other words, an external JavaScript source file is no more hidden from view than JavaScript embedded directly in an HTML file.

Navigator 3 exhibits a bug if you specify an external .js library file in a tag that specifies JavaScript 1.2 as the language. Unfortunately, Navigator 3 ignores the language version and loads the external file no matter what language is specified in that tag. Therefore, if you don't want those scripts to run in Navigator 3, surround the scripts in the external file in a version-checking if clause:

```
if (parseInt(navigator.appVersion) > 3) {
        statements to run here
}
```

Compatibility issues

On the Netscape Navigator side, the external capability was new with Navigator 3. Therefore, the SRC attribute is ignored in Navigator 2, and none of the external scripts become part of the document.

The situation is more clouded on the Internet Explorer side. When Internet Explorer 3 shipped for Windows, the external script library feature was not

available. By most accounts, Internet Explorer Version 3.02 included support for external libraries, but I have heard reports that this is not the case. I know that the Version 3.02 installed on my Windows 95 computers loads external libraries from .js files. It may be a wise tactic to specify a complete URL for the .js file, as this has been known to assist Internet Explorer 3 in locating the script library file associated with an HTML file.

JavaScript entities

Another feature that started with Navigator 3 was the *JavaScript entity*. The idea behind this technique was to provide a way for the browser to use script expressions to fill in the data for any HTML tag attribute. Netscape entities are strings that allow special characters or symbols to be embedded in HTML. They begin with an ampersand symbol (&) and end with a semicolon (;). For example, the `&c;` entity is rendered in Netscape browsers as a copyright symbol (©).

To assign a JavaScript expression to an entity, the entity still begins and ends like all entities, but the expression is surrounded by curly braces. For example, consider a document containing a function that returns the current day of the week, as follows:

```
function today() {
        var days = new
Array("Sunday","Monday","Tuesday","Wednesday","Thursday","Friday",
"Saturday")
        var today = new Date()
        return days[today.getDay()]
}
```

This function can be assigned to a JavaScript entity such that the label of a button is created with the returned value of the function:

```
<INPUT TYPE=button VALUE=&{today()}; onCLick="handleToday()">
```

Expressions can be used to fulfill only attribute assignments, not other parts related to a tag, such as the text for a document title or link. Those items can still be dynamically generated via `document.write()` statements as the document loads.

Browser Version Detection

Without question the biggest challenge facing client-side scripters is how to program an application that accommodates a wide variety of browser versions and brands, each one of which can bring its own quirks and bugs. Happy is the intranet developer who knows for a fact that the company has standardized its computers with a particular brand and version of browser. But that is a rarity, especially in light of the concept of the *extranet* — private corporate networks and applications that open up for access to the company's suppliers and customers.

Having dealt with this problem since the original scripted browser, Navigator 2, had to work alongside a hoard of nonscriptable browsers, I have identified several paths that an application developer can follow. Unless you decide to be autocratic about browser requirements for using your site, you must make compromises in

desired functionality or provide multiple paths in your Web site for two or more classes of browsers. Plenty of Web surfers are still using the text-only, server-based browser called Lynx. In this section, I give you several ideas about how to approach development in an increasingly fragmented browser world.

Is JavaScript on?

Very often, the first decision an application must make is whether the client accessing the site is JavaScript-enabled. Non–JavaScript-enabled browsers fall into two categories: a) JavaScript-capable browsers that have JavaScript turned off in the preferences; and b) browsers that have no JavaScript interpreter built in.

Using the <NOSCRIPT> tag

Except for some of the earliest versions of Netscape Navigator, all JavaScript-capable browsers have a preferences setting to turn off JavaScript (and a separate one for Java). You should know that even though JavaScript is turned on by default in Navigator, many institutional deployments have it turned off by default. The reasons behind this MIS deployment decision vary from scares about Java security violations incorrectly associated with JavaScript, valid JavaScript security concerns on some browser versions, and the fact that some firewalls try to filter JavaScript lines from incoming HTML streams.

All JavaScript-capable browsers include a set of <NOSCRIPT>. . .</NOSCRIPT> tags to balance the <SCRIPT>. . .</SCRIPT> tag set. If one of these browsers has JavaScript turned off, the <SCRIPT> tag is ignored, but the <NOSCRIPT> tag is observed. As with the <NOFRAMES> tag, you can use the body of a <NOSCRIPT> tag set to display HTML that lets users know JavaScript is turned off and therefore the full benefit of the page won't be available unless JavaScript is turned on. Listing 13-1 shows a skeletal HTML page that uses these tags.

Listing 13-1: Employing the <NOSCRIPT> Tag

```
<HEAD>
<TITLE>Some Document</TITLE>
<SCRIPT LANGUAGE="JavaScript">
       // script statements
</SCRIPT>
<NOSCRIPT>
<B>Your browser has JavaScript turned off.</B><BR>
You will experience a more enjoyable time at this Web site if you turn
JavaScript on.<BR>
Open your browser preferences, and enable JavaScript.<BR>
You do not have to restart your browser or your computer after you
enable JavaScript. Simply click the 'Reload' button.
<HR>
</NOSCRIPT>
</HEAD>
<BODY>
<H2>The body of your document.</H2>
</BODY>
</HTML>
```

You can display any standard HTML within the <NOSCRIPT> tag set. An icon image might be a colorful way to draw the user's attention to the special advice at the top of the page. If your document is designed to dynamically create content in one or more places in the document, you may have to include a <NOSCRIPT> tag set after more than one <SCRIPT> tag set to let users know what they're missing. Do not include the HTML comment tags that you use in hiding JavaScript statements from older browsers. Their presence inside the <NOSCRIPT> tags prevents the HTML from rendering.

Other nonscriptable browsers

Unfortunately, browsers that don't know about JavaScript also don't know about the <NOSCRIPT> tag. They generally do not render the content of those tags.

At this point I must point out that newcomers to scripting frequently want to know what script to write to detect whether JavaScript is turned on. Because scripters are so ready to write a script to work around all situations, it takes some thought to realize that a non-JavaScript browser cannot execute such a script: If no JavaScript interpreter exists in the browser (or it is turned off), the script is ignored. I suppose that the existence of a JavaScript-accessible method for Java detection — the `navigator.javaEnabled()` method — promises a parallel method for JavaScript. But logic fails to deliver on that unspoken promise.

Another desire is to have JavaScript substitute document content when the browser is JavaScript-enabled. But you cannot create a script to write scripted content where normal HTML is destined to appear. All HTML in a document appears in the browser, while scripted content can only be additive.

You can use this additive scripting to create unusual effects when displaying different links and (with a caveat) body text for scriptable and nonscriptable browsers. Listing 13-2 shows a short document that uses HTML comment symbols to trick nonscriptable browsers into displaying a link to Netscape's Web site and two lines of text. A scriptable browser takes advantage of a behavior that allows only the nearest <A> tag to be associated with a closing </A> tag. Therefore, the Netscape link isn't rendered at all, but the link to my Web site is. For the body text, the script assigns the same text color to a segment of HTML body text as the document's background. While the colored text is camouflaged in a scriptable browser (and some other text written to the document), the "hidden" text is still in the document, but not visible.

Listing 13-2: **Rendering Different Content for Scriptable and Nonscriptable Browsers**

```
<HTML>
<BODY BGCOLOR="yellow">
<A HREF="http://www.yahoo.com">
<SCRIPT>
<!--
document.writeln("<A HREF=\"http://www.excite.com\">")
//-->
</SCRIPT>
Where?</A>
```

(continued)

Listing 13-2 *Continued*

```
<HR>
<SCRIPT>
<!--
document.write("Howdy from the script!<FONT COLOR='yellow';>")
//-->
</SCRIPT>
<!--
If you can read this, JavaScript is not available.
<SCRIPT>
document.write("</FONT>")
//-->
</SCRIPT>
<BR>
Here's some stuff afterward.
</BODY>
</HTML>
```

Scripting for different browsers

The number of solutions for accommodating different client browsers is large because the specific compatibility need might be as simple as letting a link navigate to a scripted page for script-enabled browsers or as involved as setting up distinct areas of your application for different browser classes. The first step in planning for compatibility is determining what your goal is for various visitor classes.

Establishing goals

Once you have mapped out your application, you must then look at the implementation details to see which browser is required for the most advanced aspect of the application. For example, if the design calls for image swapping on mouse rollovers, that feature requires Navigator 3 or later and Internet Explorer 4 or later. In implementing Dynamic HTML features, you also have to decide if the functionality you specified can be made to work on both Navigator 4 and Internet Explorer 4 through some cross-platform scripting (see Chapter 41).

After you determine the lowest common denominator for the optimum experience, you then must decide how gracefully you want to degrade the application for visitors whose browsers do not meet the common denominator. For example, if you plan a page or site that requires Navigator 4 or Internet Explorer 4 for all the bells and whistles, you could provide an escape path with content in a simple format that every browser from Lynx to Navigator 3 would get. Or perhaps you would provide for users of older scriptable browsers a third offering with limited scriptability that works on all scriptable browsers.

Creating an application or site that has multiple paths for viewing the same content may sound good at the outset, but don't forget that maintenance chores lie ahead as the site evolves. Will you have the time, budget, and inclination to keep all paths up to date? Despite any good intentions a designer of a new Web site may

have, in my experience the likelihood that a site will be properly maintained diminishes rapidly with the complexity of the maintenance task.

Implementing a branching index page

If you decide to offer two or more paths into your application or content, one place you can start visitors down their individual paths is at the default page for your site. Numerous techniques are available that can redirect visitors to the appropriate perceived starting point of the site.

One design to avoid is placing the decision about the navigation path in the hands of the visitor. Offering buttons or links that describe the browser requirements may work for users who are HTML and browser geeks, but the average consumers surfing the Web these days likely won't have a clue about what level of HTML their browsers support or whether they are JavaScript-enabled. It is incumbent upon the index page designer to automate the navigation task as much as possible.

A branching index page has almost no content. It is not the "home page" per se of the site, rather merely a portal to the entire Web site. Its job is to redirect users to what appears to be the home page for the site. Listing 13-3 shows what such a branching index page looks like.

Listing 13-3: A Branching Index Page

```
<HTML>
<HEAD>
  <TITLE>GiantCo On The Web</TITLE>
  <SCRIPT LANGUAGE="JavaScript">
  <!--
      window.location = "home1.html"
  //-->
  </SCRIPT>
  <META HTTP-EQUIV="REFRESH" CONTENT="1;
URL=http://www.giantco.com/home2.html">
</HEAD>

<BODY>
<CENTER>
   <A HREF="home2.html"><IMG SRC="images/giantcoLogo.gif" HEIGHT=60
WIDTH=120 BORDER=0 ALT="GiantCo Home Page"></A>
</CENTER>
</BODY>
</HTML>
```

Notice that the only visible content is an image surrounded by a standard link. The <BODY> tag contains no background color or art. A single script statement is located in the Head. A <META> tag is also in the Head to automate navigation for some users. To see how a variety of browsers would respond to this page, here are what three different classes of browser would do with Listing 13-3:

A JavaScript-enabled browser. Although the entire page loads momentarily (flashing the company logo for a brief moment), the browser executes the script statement that loads home1.html into the window. This executes within about a

second after the page loads. In the meantime, the image has been preloaded into the browser's memory cache. This image should be one that is reused in home1.html so the download time isn't wasted on a one-time image. If your pages require a specific brand or minimum version number, this would be the place to filter out browsers that don't meet the criteria (which may include the installation of a particular plug-in). Use the properties of the navigator object (Chapter 25) to allow only those browsers meeting your design minimum to navigate to the scripted home page. All others fall through to the next execution possibility.

A modern browser with JavaScript turned off or missing. Several modern browsers recognize the special format of the <META> tag as one that loads a URL into the current window after a stated number of seconds. In Listing 13-3 that interval is one second. The <META> tag is executed only if the browser ignores the <SCRIPT> tag. Therefore, any scriptable browser that has JavaScript turned off or browser that knows <META> tags but no scripting will follow the refresh command for the <META> tag. If you utilize this tag, be very careful to observe the tricky formatting of the CONTENT attribute value. The number of seconds is followed by a semicolon and the subattribute URL. A complete URL for your nonscriptable home page version is required for this subattribute. Importantly, the entire CONTENT attribute value is inside one set of quotes.

Older graphical browsers, PDA browsers, and Lynx. The last category includes graphical browsers some would call "brain-dead" as well as intentionally stripped down browsers. Lynx is designed to work in a text-only VT-100 terminal screen; personal digital assistants (PDAs) such as the Apple Newton and Windows CE devices have browsers optimized for usage through slow modems and viewing on small screens. If such browsers do not understand the <META> tag for refreshing content, they will land at this page with no further automatic processing. But by creating an image that acts as a link, the tendency will be to click on it to continue. The link then leads to the nonscriptable home page. Also note that the ALT attribute for the image is supplied. This takes care of Lynx and PDA browsers (with image loading off), because these browsers show the ALT attribute text in lieu of the image. Users click on the text to navigate to the URL referenced in the link tag.

I have a good reason to keep the background of the branching index page plain. For those whose browsers automatically lead them to a content-filled home page, the browser window will flash from a set background color to the browser's default background color before the new home page and its background color appear. By keeping the initial content to only the company logo, less screen flashing and obvious navigation are visible to the user.

One link — alternate destinations

Another filtering technique is available directly from links. With the exceptions of Navigator 2 and Internet Explorer 3, a link can navigate to one destination via a link's onClick= event handler and to another via the HREF attribute if the browser is not scriptable.

The trick is to include an extra `return false` statement in the onClick= event handler. This statement cancels the link action of the HREF attribute. For example, if a nonscriptable browser is to go to one version of a page at the click of a link and the scriptable browser should go to another, the link tag would be as follows:

```
<A HREF="nonJSCatalog.html" onClick="location='JSCatalog.html';return
false">Product Catalog</A>
```

Only nonscriptable browsers, Navigator 2, and Internet Explorer 3 go to the nonJSCatalog.html page; all others go to the JSCatalog.html page.

Multiple-level scripts

Each new JavaScript level brings more functionality to the language. You can use the LANGUAGE attribute of the <SCRIPT> tag to provide road maps for the execution of functions according to the power available in the browser. For example, consider a button whose event handler invokes a function. That function can be written in such a way that users of each JavaScript version get special treatment with regard to unique features of that version. To make sure all scriptable browsers handle the event handler gracefully, you can create multiple versions of the function, each wrapped inside its own <SCRIPT> tag specifying a particular language version.

Listing 13-4 shows the outline of a page that presents different versions of the same event handler. For this technique to work properly, you must lay out the <SCRIPT> tags in ascending order of JavaScript version. In other words, the last function that the browser knows how to read (according to the LANGUAGE version) is the one that gets executed. In Listing 13-4, for instance, Navigator 3 gets only as far as the middle version and executes only that one.

Listing 13-4: **Multiple Script Versions**

```
<HTML>
<HEAD>
   <SCRIPT LANGUAGE="JavaScript">
   <!--
      function doIt() {
        // statements for JavaScript 1.0 browsers
      }
    //-->
   </SCRIPT>
   <SCRIPT LANGUAGE="JavaScript1.1">
   <!--
      function doIt() {
        // statements for JavaScript 1.1 browsers
      }
    //-->
    </SCRIPT>
    <SCRIPT LANGUAGE="JavaScript1.2">
    <!--
      function doIt() {
        // statements for JavaScript 1.2 browsers
      }
    //-->
    </SCRIPT>
</HEAD>
<BODY>
<FORM>
    <INPUT TYPE=button VALUE="Click Me" onClick="doIt()">
</FORM>
```

(continued)

Listing 12-2 *Continued*

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

If you use this technique, you must define an event handler for each level below the highest version you intend to support. For example, failure to include a version for JavaScript 1.0 in Listing 13-4 would result in a script error for users of Navigator 2 and Internet Explorer 3. If you don't want an older browser to execute a function (because the browser doesn't support the functionality required for the action), include a *dummy function* (a function definition with no statements) in the lower-level <SCRIPT> tag to catch the event handlers of less-capable browsers.

Scripted event handler properties

Along the same lines of Listing 13-4, you can define event handlers for objects within separate language versions. For example, in Listing 13-5, a button is assigned an event handler within the context of a JavaScript 1.1–level script. Navigator 2 and Internet Explorer 3 users won't have their button's event handler set, because the HTML tag doesn't have an event handler. In fact, you could also specify a different function for JavaScript 1.2–level browsers by including another <SCRIPT> tag below the one that currently sets the event handler. The new tag would specify JavaScript 1.2 as the language and assign a different function to the event handler.

Listing 13-5: **Event Handler Assignments**

```
<HTML>
<HEAD>
    <SCRIPT LANGUAGE="JavaScript1.1">
    <!--
       function doIt() {
          // statements for JavaScript 1.1 browsers
       }
     //-->
     </SCRIPT>
</HEAD>
<BODY>
<FORM>
    <INPUT TYPE=button NAME=janeButton VALUE="Click Me">
    <SCRIPT LANGUAGE="JavaScript1.1">
    <!--
       document.forms[0]. janeButton.onclick=doIt
    //-->
    </SCRIPT>
</FORM>
</BODY>
</HTML>
```

Designing for Compatibility

Each new major release of a browser brings compatibility problems for page authors. It's not so much that old scripts break in the new versions (well-written scripts rarely break in new versions). No, the problems center on the new features that attract designers and how to handle visitors who have not yet advanced to the latest and greatest browser version.

Adding to these problems are numerous bugs, particularly in first-generation browsers from both Netscape and Microsoft. Worse still, some of these bugs affect only one operating system platform among the many supported by the browser. Even if you have access to all the browsers for testing, the process of finding the errors, tracking down the bugs, and implementing workarounds that won't break later browsers can be a frustrating process, even when you've scripted pages from the earliest days and have a long memory for ancient bug reports.

Catering only to the lowest common denominator can more than double your development time, due to the expanded testing matrix necessary to ensure a good working page in all operating systems on all versions. Decide how important the scripted functionality you're employing in a page is for every user. If you want some functionality that works only in a later browser, then you may have to be a bit autocratic in defining the minimum browser for scripted access to your page — any lesser browser gets shunted to a simpler presentation of your site's data.

Another possibility is to make a portion of the site accessible to most, if not all, browsers, restricting the scripting to only the occasional enhancement that nonscriptable browser users won't miss. But once the application reaches a certain point in the navigation flow, then a more capable browser is necessary to get to the really good stuff. This kind of design is a carefully planned strategy that lets the site welcome all users up to a point, but then enables the application to shine for users of, say, Navigator 3 or higher.

The ideal page is one that displays useful content on any browser, but whose scripting enhances the experience of the page visitor — perhaps in the way of more efficient site navigation or interactivity with the page's content. That is certainly a worthy goal to aspire to. But even if you can achieve this nirvana on only some pages, you will reduce the need for defining entirely separate, difficult-to-maintain paths for browsers of varying capabilities.

Dealing with beta browsers

If you have crafted a skillfully scripted Web page or site, you may be concerned when a prerelease version of a browser available to the public causes script errors or other compatibility problems to appear on your page. Do yourself a favor — don't overreact to bugs and errors that occur in prerelease browser versions. If your code is well written, it should work with any new generation of browser. If the code doesn't work correctly, consider the browser to be buggy. Report the bug (preferably with a script sample) to the browser maker.

It is often difficult to prevent yourself from getting caught up in browser makers' enthusiasm for a new release. But remember that a prerelease version is not a shipping version. Users who visit your page with prerelease browsers should know that there may be bugs in the browser. That your code does not work with a prerelease version is not a sin, nor is it worth losing sleep over. Just be sure to report the bug so the problem doesn't make it into the release version.

Browser version headaches

As described more fully in Chapter 25's discussion of the navigator object, your scripts can easily determine which browser is the one running the script. However, the properties that reveal the version don't always tell the whole story about Internet Explorer 3.

For one thing, the Windows and Macintosh versions of the same major browser version (3.0*x*) implement slightly different object models. The Mac version includes the ever-popular image object for mouse rollover image swapping; the Windows version does not, and any attempt to use such code in the Windows version results in script errors.

Next, the first release of Internet Explorer 3 for the Macintosh was not scriptable at all — the JavaScript interpreter was left out. Macintosh Version 3.01 was the first Mac version to be scriptable. Even among minor generation releases of Internet Explorer 3 for Windows, Microsoft implemented some new features here and there.

But probably the most troublesome problem is that an improved JavaScript interpreter (in the JScript.dll file) underwent substantial improvements between Version 1 and Version 2 for Windows. Many copies of browser Version 3.02 for Windows shipped with Version 1 of the .dll. Some users updated their browsers if they knew to download the new .dll from Microsoft. Unfortunately, the interpreter version is not reflected in any navigator object property. A nasty Catch-22 in this regard is that Version 2 of the interpreter includes a new property that lets you examine the interpreter version, but testing for that property in a browser that has Version 1 of the interpreter installed results in an error message.

Due to the insecurity of knowing exactly what will and won't work in a browser that identifies itself as Internet Explorer 3.0*x*, you might decide to redirect all users of Internet Explorer 3 to pages in your application that include no scripting. But before you think I'm bashing Internet Explorer 3, you should also consider doing the same redirection for Navigator 2 users due to the number of platform-specific bugs that littered that first round of JavaScript. JavaScript Versions 1.1 and 1.2 are much more stable and reliable platforms on which to build scriptable applications. If you have an opportunity to study the access logs of your Web site, analyze the proportion of different browser versions over several days before deciding where you set your lowest common denominator for scripted access.

Compatibility ratings in reference chapters

With the proliferation of scriptable browser versions since Navigator 2, it is important to know up front whether a particular object, property, method, or event handler is supported in the lowest common denominator you are designing for. Therefore, in the rest of the chapters of this reference part of the book, I include frequent compatibility charts, like the following example:

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	(✔)	✔	✔

The first three columns represent Navigator Versions 2, 3, and 4, respectively. For Internet Explorer, two columns appear for Version 3. One, marked IE3/J1, represents the combination of Internet Explorer 3 and JScript.dll Version 1; IE3/J2 represents Internet Explorer 3 and JScript.dll Version 2. The first shipping version of Internet Explorer 4 comes with JScript.dll Version 3, which is the latest one available as this book goes to press. A checkmark means the feature is compatible with the designated browser. You will also occasionally see one or more of the checkmarks surrounded in parentheses. This means some bug or partial implementation for that browser is explained in the body text.

I also recommend that you print out the JavaScript Road Map file shown in Appendix A. The map is on the companion CD-ROM in Adobe Acrobat format. This quick reference clearly shows each object's properties, methods, and event handlers, along with keys to the browser version in which each language item is supported. You should find the printout to be as valuable a day-to-day resource.

Object Definitions in This Book

Throughout the rest of Part III, I present detailed descriptions of each document object (and many core language objects). Each object definition begins with a summary listing of its properties, methods, and event handlers, giving you a sense of the scope of everything a particular object has to offer. From there, I go into detailed explanations of when and how to use each term.

Whenever a syntax definition appears, note the few conventions used to designate items, such as the optional parameters and placeholders that describe the kind of data that belongs in those slots. All syntax definitions and code examples are in HTML, so expect to see plenty of HTML tags in the familiar angle brackets (<>). Listing 13-6 shows the button object definition.

Listing 13-6: **Sample Object Definition**

```
<INPUT
        TYPE="button" | "submit" | "reset"
        NAME="buttonName"
        VALUE="labelText"
        [onClick="handlerTextOrFunction"]
        [onMouseDown="handlerTextOrFunction"]
[onMouseUp="handlerTextOrFunction"] >
```

You should recognize most of this definition as the typical form for an HTML element. The entire definition is surrounded by angle brackets. Some attribute parameters (for `NAME=`, `VALUE=`, and the event handlers) let you assign names and other values meaningful to your script: In this case, the attribute parameters represent how you name the button, what text you want to appear on the button's label, and what you want the button to do when someone clicks it. Those placeholders appear in italics to remind you that you need to fill in those blanks with your own names. The placeholders attempt to describe the nature of the parameter. This is nothing more than what you'd find in a good HTML guide.

The last three attributes of the definition in Listing 13-6 appear inside square brackets. This convention means the attributes are optional. In this case, if you omit the event handler attributes, the button is just another HTML button like others you've probably specified before (although it won't do anything when clicked).

In other definitions, especially parameters for methods, the values to be supplied must be a particular data type (string, number, or Boolean). When the value must be of a specific type, the placeholder for that parameter indicates the data type in its name. For example, look at the history object's `go()` method:

```
go (deltaNumber | "TitleOrURL")
```

In this rare instance of JavaScript accepting either one of two data types (shown on either side of the "|" character), the method can accept either a number or a string (denoted by the quotes) containing a valid title or URL in the window's history list.

Some methods return values after they execute. For example, one window method displays a dialog box that prompts a user to enter text. When the user clicks on the OK button, the text entered into the dialog box is passed back as a value that can be assigned to a variable or used directly as a parameter for another method. Each method listed in the object definitions indicates what kind of value, if any, is returned by that method.

The last item of note about these definitions concerns the way you handle property values. Every property has a value that must be one of the valid JavaScript data types. Whether a property returns a Boolean, a string, or a number can be important for your scripting statements. Therefore, I note the kind of value required for each property.

✦ ✦ ✦