CHAPTER 4

Browser and Document Objects

In This Chapter

What client-side scripts do

What happens when a document loads

How the browser creates objects

How scripts refer to objects

How to find out what is scriptable in an object

This chapter marks the first of nine tutorial chapters (which compose Part II) tailored to authors who have at least basic grounding in HTML concepts. You will see several practical applications of JavaScript and begin to see how a JavaScript-enabled browser turns familiar HTML elements into objects that your scripts control.

Scripts Run the Show

If you have authored in plain HTML, you are familiar with how HTML tags influence the way content is rendered on a page when viewed in the browser. As the page loads, the browser recognizes tags (by virtue of their containing angle brackets) as formatting instructions. Instructions are read from the top of the document downward, and elements defined in the HTML document appear on screen in the same order in which they are entered in the document. As an author, you do a little work one time up front — adding the tags — and the browser does a lot more work every time a visitor loads the page into a browser.

Assume for a moment that one of the elements on the page is a text field inside a form. The user is supposed to enter some text in the text field and then click the Submit button to send that information back to the Web server. If that information must be an Internet e-mail address, how do you ensure the user included the "@" symbol in the address?

One way is to have a Common Gateway Interface (CGI) program on the server scan the submitted form data after the user has clicked the Submit button and the form information has been transferred to the server. If the user omitted or forgot the "@" symbol, the CGI program resends the page, but this time with an instruction to include the symbol in the address. Nothing is wrong with this exchange, but it means a significant delay for the user to find out that the address does not contain the crucial symbol. Moreover, the Web server has

had to expend some of its resources to perform the validation and communicate back to the visitor. If the Web site is a busy one, the server may be trying to perform hundreds of these validations at any given moment, probably slowing the response time to the user even more.

Now imagine if the document containing that text field had some intelligence built into it that could make sure the text field entry contains the "@" symbol before ever sending one bit (literally!) of data to the server. That kind of intelligence would have to be embedded in the document in some fashion — downloaded with the page's content so it can stand ready to jump into action when called upon. The browser must know how to run that embedded program. Some user action must start the program, perhaps when the user clicks the Submit button. As the program runs, if it detects a lack of the "@" symbol, an alert message should appear to bring the problem to the user's attention; the same program should also be able to decide if the actual submission can proceed, or if it should wait until a valid e-mail address is entered into the field.

This kind of presubmission data entry validation is but one of the practical ways JavaScript adds intelligence to an HTML document. Looking at this example, you might recognize that a script must know how to look into what has been typed in a text field; a script must also know how to let a submission continue or how to abort the submission. A browser capable of running JavaScript programs conveniently treats elements such as the text field as *objects*. A JavaScript script controls the action and behavior of objects — most of which you see on the screen in the browser window.

JavaScript in Action

By adding lines of JavaScript code to your HTML documents, you control on-screen objects as your applications require. To give you an idea of the scope of application you can create with JavaScript, I show you several applications from the CD-ROM (in the folder named Bonus Applications Chapters). I strongly suggest you open the applications and play with them in your browser. Links to the application files from the CD-ROM can be found on the page tutor1.htm in the book listings folder. I also provide URLs to the examples at my Web site.

Interactive user interfaces

HTML hyperlinks do a fine job, but they're not necessarily the most engaging way to present a table of contents to a large site or document. With a bit of JavaScript, it is possible to create an interactive, expandable table of contents listing that displays the hierarchy of a large body of material (see Figure 4-1). Just like the text listings in operating system file management windows, the expandable table of contents lets the user see as much or as little as possible, while providing a shortcut to the Big Picture of the entire data collection.

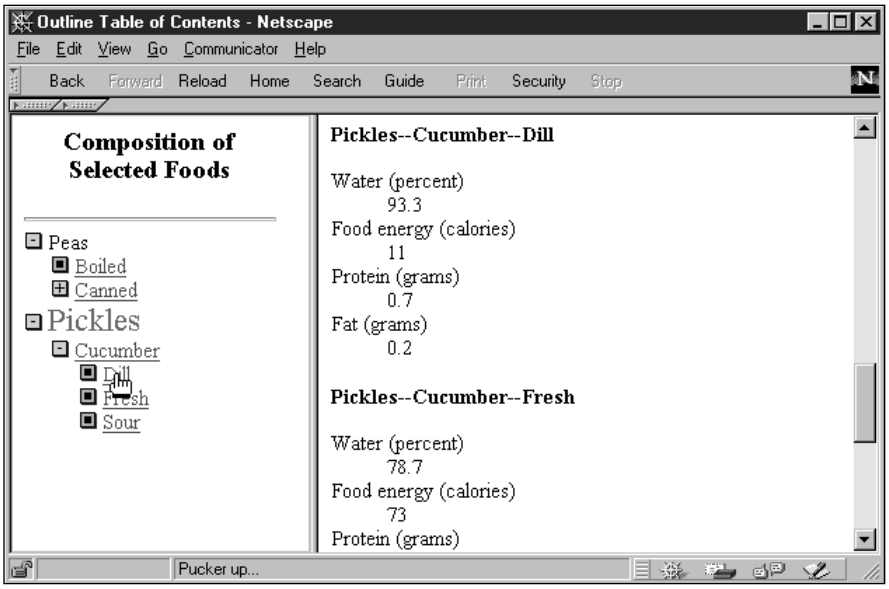

Figure 4-1: An expandable table of contents (http://www.dannyg.com/javascript/ol2/index.htm)

Click on a gray widget icon to expand the items underneath. An endpoint item has an orange and black widget icon. Items in the outline can be links to other pages or descriptive information. You also maintain the same kind of font control over each entry as you would expect from HTML. While such outlines have been created with server CGIs in the past, the response time between clicks is terribly slow. By placing all of the smarts behind the outline inside the page, it downloads once and runs quickly after each click.

As demonstrated in the detailed description of this outline in the application Outline-Style Table of Contents (Chapter 50 of the bonus applications chapters on the CD-ROM), the scriptable workings can be implemented within straight HTML for Navigator 2 and 3 and in Dynamic HTML for Navigator 4 and Internet Explorer 4. Either way you do it, the quick response and action on the screen makes for a more engaging experience for Web surfers who are in a hurry to scout your site.

Small data lookup

A common application on the Web is having a CGI program present a page that visitors use to access large databases on the server. Large data collections are best left on the server, where search engines and other technologies are the best fit. But if your page acts as a "front end" to a small data collection lookup, you can consider embedding that data collection in the document (out of view) and letting JavaScript act as the intermediary between user and data.

I've done just that in a Social Security prefix lookup system shown in Figure 4-2. I converted a printed table of about 55 entries into a JavaScript table that occupies only a few hundred bytes. When the visitor types the three-character prefix of his or her Social Security number into the field and clicks the Search button, a script

behind the scenes compares that number against the 55 or so ranges in the table. When the script finds a match, it displays the corresponding state of registration in a second field.

If the application were stored on the server and the data stored in a server database, each click of the Search button would mean a delay of many seconds, as the server processes the request, gets the data from the database, and reformulates the page with the result for the user. Built instead as a JavaScript application, once the page downloads the first time, any number of lookups are instantaneous.

Figure 4-2: Looking up data in a small table (http://www.dannyg.com/javascript/ssn2/ssbirthplace.htm)

Forms validation

I've already used data entry form validation as an example of when JavaScript is a good fit. In fact the data entry field in the Social Security lookup page (see Figure 4-2) includes scripting to check the validity of the entered number. Just as a CGI program for this task would have to verify that the entry was a three-digit number, so, too, must the JavaScript program verify the entered value. If a mistake appears in the entry — perhaps a finger slipped and hit a letter key — the visitor is advised of the problem and directed to try another entry. The validation script even preselects the text in the entry field for the visitor so that typing a new value replaces the old.

Interactive data

JavaScript opens opportunities for turning static information into interactive information. Figure 4-3 shows a graphical calculator for determining the value of an electrical component (called a resistor) whose only markings are colored bars.

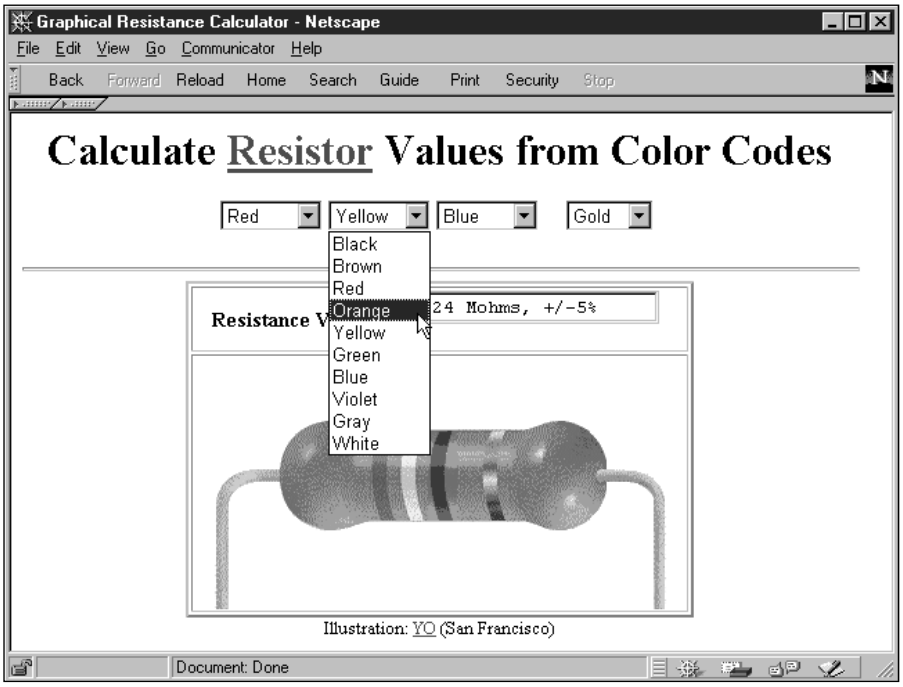

Figure 4-3: An interactive graphical calculator (http://www.dannyg.com/javascript/res2/resistor.htm)

The image in the bottom half of the page is composed of seven images in vertical slices all bunched up against each other. As the visitor selects a color from a pop-up list near the top, the associated image slice changes to the selected color and the resistance value is calculated and displayed.

Again, with the page loaded, response time is instantaneous, whereas a server-based version of this calculator would take many seconds between color changes. Moreover, JavaScript provides the power to preload all possible images while the main page loaded. Therefore, with only a slight extra delay to download all images with the page, no further delay occurs when a visitor chooses a new color. Not only is the application practical (for its intended audience), but it's just plain fun to play with.

Multiple frames

While frames are the domain of HTML, they suddenly become more powerful with some JavaScript behind them. The Decision Helper application shown in Figure 4-4 takes this notion to the extreme.

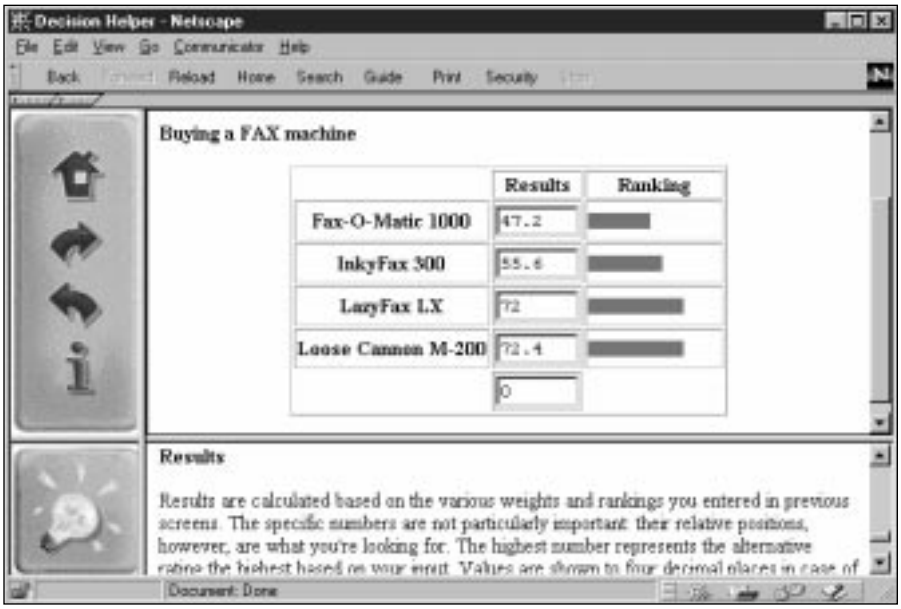

Figure 4-4: The Decision Helper (http://www.dannyg.com/javascript/dh2/)

The Decision Helper is a full-fledged application that includes four input screens and one screen that displays the results of some fairly complex calculations based on the input screens. Results are shown both in numbers and in a bar graph form, as shown in Figure 4-4.

Interaction among three of the four frames requires JavaScript. For example, if the user clicks on one of the directional arrows in the top-left frame, not only does the top-right frame change to another document, but the instructions document in the bottom-right frame shifts to the anchor point that parallels the content of the input screen. Scripting behind the top-right frame documents uses various techniques to preserve entry information as the user navigates through the sequence of input pages. These are the same techniques you might use to build an online product catalog and shopping cart — accumulating the customer's selections from various catalog pages, and then bringing them together in the checkout order form.

Certainly this application could be fashioned out of a CGI program on the server. But the high level of interaction and calculation required would turn this now speedy application into a glacially slow exchange of information between user and server.

Dynamic HTML

Starting with the level 4 browsers from both Netscape and Microsoft, more and more content on the page can be modified with the help of client-side scripts. In Figure 4-5, for example, scripts in the page control the dragging of map pieces in the puzzle. Highlight colors change as you click on the state maps, instruction

panels fly in from the edge of the screen, and another item appears when you place all the states in their proper positions.

The browser feature that makes this level of script control possible is *Dynamic HTML*. JavaScript becomes the vital connection between the user and dynamically respositionable elements on the screen. Not even a CGI program could help this application, since you need immediate programmatic control in the page to respond to user mouse motion and instantaneous changes to screen elements.

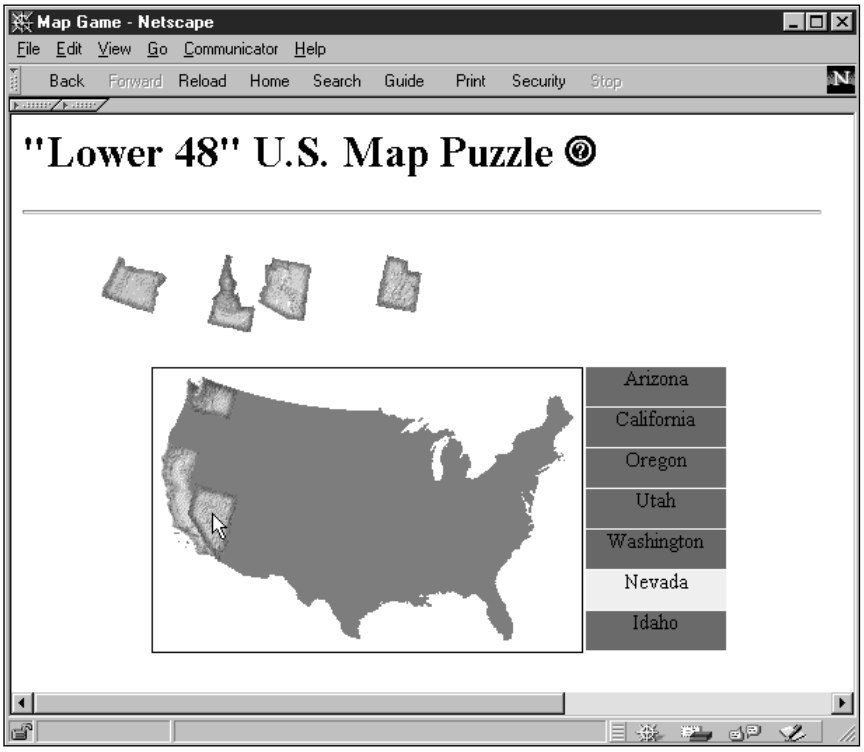

Figure 4-5: A map game in scriptable Dynamic HTML (http://www.dannyg.com/javascript/puzzle/mapgame.htm)

When to use JavaScript

The preceding examples demonstrate a wide range of applications for JavaScript, but by no means do they come close to exhausting JavaScript's possibilities. When faced with a Web application task, I look to client-side JavaScript for help with the following requirements:

- **Data entry validation:** If form fields need to be filled out for processing on the server, I let client-side scripts prequalify the data entered by the user.
- **Server-less CGIs:** I use this term to describe processes that, were it not for JavaScript, would be programmed as CGIs on the server, yielding slow

performance because of the interactivity required between the program and user. This includes tasks such as small data collection lookup, modification of images, and generation of HTML in other frames and windows based on user input.

- **Dynamic HTML interactivity:** It's one thing to use DHTML's abilities to precisely position elements on the page — you don't need scripting for that. But if you intend to make the content dance on the page, scripting makes that happen.
- **CGI prototyping:** Sometimes you may want a CGI program to be at the root of your application because it reduces the potential incompatibilities among browser brands and versions. It may be easier to create a prototype of the CGI in client-side JavaScript. Use the opportunity to polish the user interface before implementing the application as a CGI.
- **Offloading a busy server:** If you have a highly trafficked Web site, it may be beneficial to convert frequently used CGI processes to client-side JavaScript scripts. Once a page is downloaded, the server is freed to serve other visitors. Not only does this lighten server load, but users experience quicker response to the application embedded in the page.
- **Adding life to otherwise dead pages:** HTML by itself is pretty "flat." Adding a blinking chunk of text doesn't help much; animated GIF images more often distract from than contribute to the user experience at your site. But if you can dream up ways to add some interactive zip to your page, it may engage the user and encourage a recommendation to friends or repeat visits.
- **Creating "Web pages that think":** If you let your imagination soar, you may develop new, intriguing ways to make your pages appear to be smart. For example, in the application Intelligent "Updated" Flags (Chapter 52 on the CD-ROM) you will see how, without a server CGI or database, an HTML page can "remember" when a visitor last came to the page; then any items that have been updated since the last visit — regardless of the number of updatings you've done to the page — are flagged for that visitor. That's the kind of subtle, thinking Web page that best displays JavaScript's powers.

The Document Object Model

Before you can truly start scripting, you should have a good feel for the kinds of objects you will be scripting. A scriptable browser does a lot of the work of creating software objects that generally represent the visible objects you see in an HTML page in the browser window. Obvious objects include images and form elements. However there may be other objects that aren't so obvious by looking at a page, but make perfect sense when you consider the HTML tags used to generate a page's content.

To help scripts control these objects — and to help authors see some method to the madness of potentially dozens of objects on a page — the browser makers define a *document object model*. A model is like a prototype or plan for the organization of objects in a page. Figure 4-6 shows the document object model that

Netscape has defined for Navigator 4. Internet Explorer contains almost all of the objects in Figure 4-6, but Microsoft's model is more extensive. At this stage of the learning process, it is not important to memorize every last object in the model, but rather to get a general feel for what's going on. This model appears again in subsequent chapters.

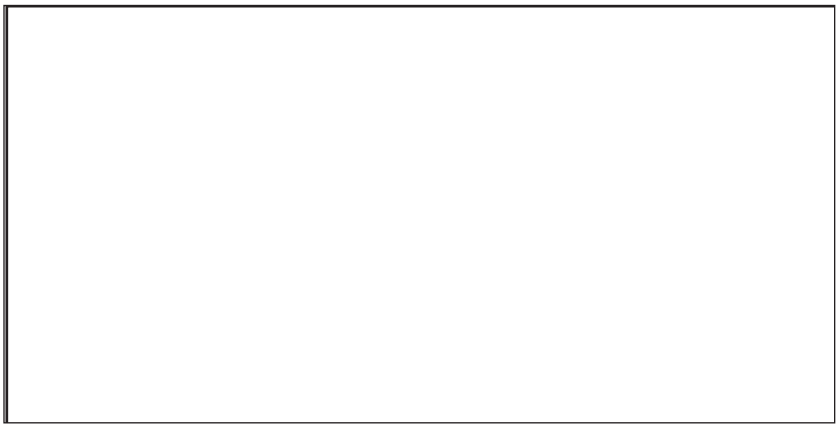

Figure 4-6: Netscape Navigator 4 document object model

One misconception you must avoid at the outset is that the model shown in Figure 4-6 is the model for every document that loads into the browser. On the contrary — it represents an idealized version of a document that includes one of every possible type of object that Navigator 4 knows about. In a moment, I will show you how the document object model stored in the browser at any given instant reflects the HTML in the document. But for now, I want to impress an important aspect of the structure of the idealized model: its hierarchy.

Containment Hierarchy

Notice in Figure 4-6 that objects are grouped together in various levels designated by the density of the gray background. Objects are organized in a hierarchy, not unlike the hierarchy of a company's organization chart of job positions. At the top is the president. Reporting to the president are several vice presidents. One of the vice presidents manages a sales force that is divided into geographical regions. Each region has a manager who reports to the vice president of sales; each region then has several salespeople. If the president wanted to communicate to a salesperson who handles a big account, the protocol would call for the president to route the message through the hierarchy — to the vice president of sales; to the sales manager; to the salesperson. The hierarchy clearly defines each unit's role and relationship to the other units.

This hierarchical structure applies to the organization of objects in a document. Allow me to highlight the key objects and explain their relationships to others.

- **Window object:** At the top of the hierarchy is the window. This object represents the content area of the browser window where HTML documents

appear. In a multiple-frame environment, each frame is also a window, but don't concern yourself with this just yet. Because all document action takes place inside the window, the window is the outermost element of the object hierarchy. Its physical borders contain the document.

- **Document object:** Each HTML document that gets loaded into a window becomes a document object. Its position in the object hierarchy is an important one, as you can see in Figure 4-6. The document object contains by far the most other kinds of objects in the model. This makes perfect sense when you think about it: The document contains the content that you are likely to script.
- **Form object:** Users don't see the beginning and ending of forms on a page, only their elements. But a form is a distinct grouping of content inside an HTML document. Everything that is inside the <FORM>. . .</FORM> tag set is part of the form object. A document might have more than one pair of <FORM> tags if that's what the page design calls for. If so, the map of the objects for that particular document would have two form objects instead of the one that appears in Figure 4-6.
- **Form elements:** Just as your HTML defines form elements within the confines of the <FORM>. . .</FORM> tag pair, so does a form object contain all the elements defined for that object. Each one of those form elements — text fields, buttons, radio buttons, checkboxes, and the like — is a separate object. Unlike the one-of-everything model shown in Figure 4-6, the precise model for any document depends on the HTML tags in the document.

When a Document Loads

Programming languages, such as JavaScript, are convenient intermediaries between your mental image of how a program works and the true inner workings of the computer. Inside the machine, every word of a program code listing influences the storage and movement of bits (the legendary 1s and 0s of the computer's binary universe) from one RAM storage slot to another. Languages and object models are inside the computer (or, in the case of JavaScript, inside the browser's area of the computer) to make it easier for programmers to visualize how a program works and what its results will be. The relationship reminds me a lot of knowing how to drive an automobile from point A to point B without knowing exactly how an internal combustion engine, steering linkages, and all that other internal "stuff" works. By controlling high-level objects such as the ignition key, gear shift, gas pedal, brake, and steering wheel, I can get the results I need.

Of course programming is not exactly like driving a car with an automatic transmission. Even scripting requires the equivalent of opening the hood and perhaps knowing how to check the transmission fluid or change the oil. Therefore, now it's time to open the hood and watch what happens to the document object model as a page loads into the browser.

A simple document

Figure 4-7 shows the HTML and corresponding object model for a very simple document. When this page loads, the browser maintains in its memory a map of the objects generated by the HTML tags in the document. The window object is always there for every document. Every window object also contains an object called the *location* object (it stores information about the URL of the document being loaded). I'll skip that object for now, but acknowledge its presence (as a dimmed box in the diagram) because it is part of the model in the browser memory. Finally, since a document has been loaded, the browser generates a document object in its current map.

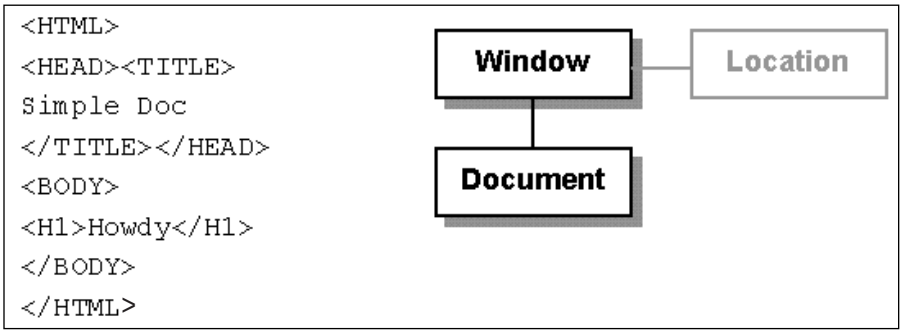

Figure 4-7: A simple document and object map

Add a form

Now I modify the HTML file to include a blank <FORM> tag set and reload the document. Figure 4-8 shows what happens to both the HTML (changes in boldface) and the object map as constructed by the browser. Even though no content appears in the form, the <FORM> tags are enough to tell the browser to create that form object. Also note that the form object is contained by the document in the hierarchy of objects in the current map. This mirrors the structure of the idealized map shown back in Figure 4-6.

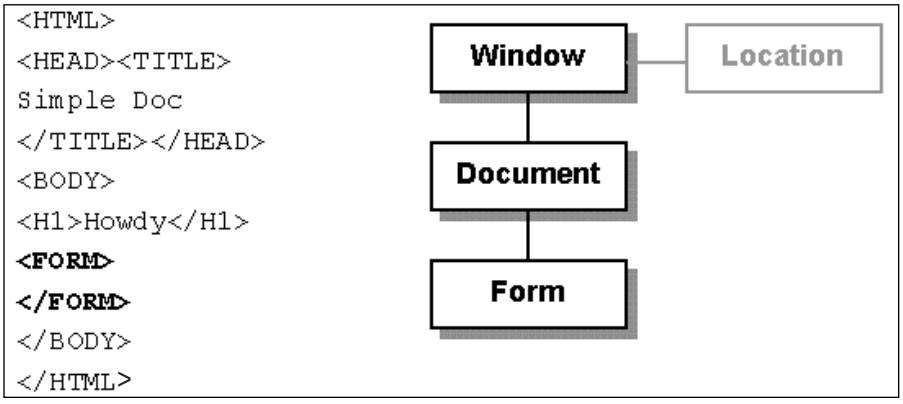

Figure 4-8: Adding a form

Add a text input element

I modify and reload the HTML file again, this time including an <INPUT> tag that defines a text field form element, shown in Figure 4-9. As mentioned earlier, the containment structure of the HTML (the <INPUT> tag goes inside a <FORM> tag set) is reflected in the object map for the revised document. Therefore, the window contains a document; the document contains a form; and the form contains a text input element.

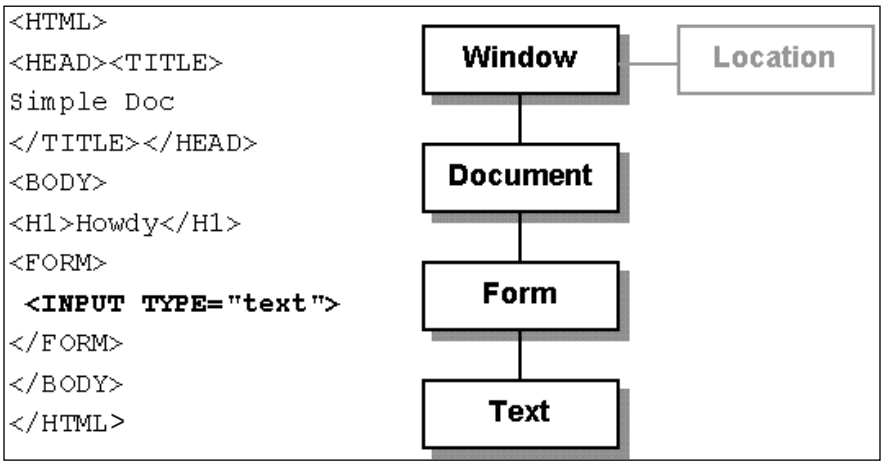

Figure 4-9: Adding a text input element to the form

Add a button element

The last modification I make to the file is to add a button input element to the same form as the text input element earlier (see Figure 4-10). Notice that the HTML for the button is contained by the same <FORM> tag set as the text field. As a result, the object map hierarchy shows both the text field and button being contained by the same form object. If the map were a corporate organization chart, the employees represented by the Text and Button boxes would be at the same level, reporting to the same boss.

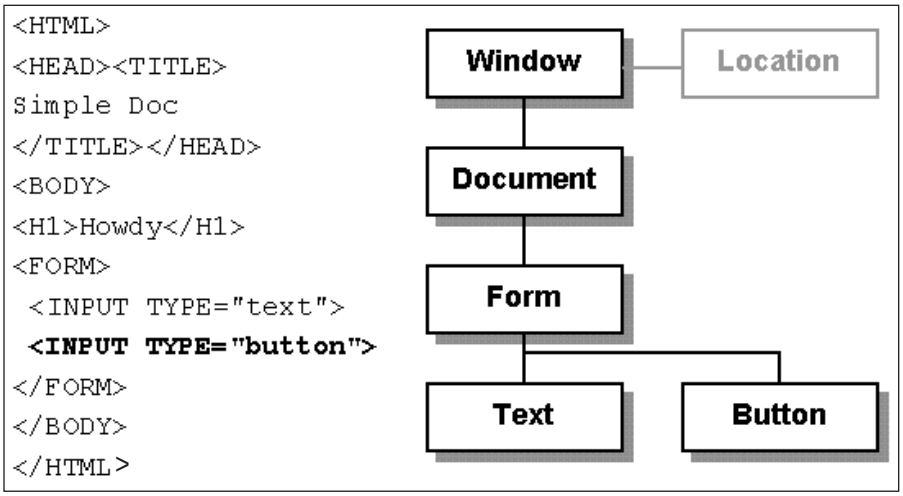

Figure 4-10: Adding a button element to the same form

Now that you see how objects are created in memory in response to HTML tags, the next step is to figure out how scripts can communicate with these objects. After all, scripting is mostly about controlling these objects.

Object References

After a document has loaded into the browser, all of its objects are safely stored in memory in the containment hierarchy structure specified by the browser's document object model. For a script to control one of those objects there must be a way to point to an object and find out something about it: "Hey, Mr. Text Field, what has the user typed in there?"

The JavaScript language uses the containment hierarchy structure to let scripts get in touch with any object in a document. For a moment, make believe you are the browser with a document loaded into your memory. You have this road map of objects handy. If a script needs you to locate one of those objects, it would be a big help if the script showed you what route to follow in the map to reach that object. There must be some way for the script to denote which object he means: an *object reference*.

Object naming

The biggest aid in creating script references to objects is assigning names to every scriptable object in your HTML. Scriptable browsers such as modern versions of Navigator and Internet Explorer acknowledge an optional tag attribute called `NAME`. The attribute lets you assign a unique name to each object. Here are some examples of `NAME` attributes being added to typical tags:

```
<FORM NAME="dataEntry" METHOD=GET>
<INPUT TYPE="text" NAME="entry">
<FRAME SRC="info.html" NAME="main">
```

The only rules about object names (also called *identifiers*) are that they

- ✦ May not contain spaces
- ✦ Should not contain punctuation except for the underscore character
- ✦ Must be inside quotes when assigned to the NAME attribute
- ✦ Must not start with a numeric character

Think of assigning names the same as sticking name tags on everyone attending a conference meeting. The name of the object, however, is only one part of the actual reference that the browser needs to locate the object. For each object, the reference must include the steps along the object hierarchy starting at the top down to the object, no matter how many levels of containment are involved. In other words, the browser cannot pick out an object by name only. A reference includes the names of each object along the path from the window to the object. In the JavaScript language, each successive object name along the route is separated from the other by a period.

To demonstrate what real references look like within the context of an object model you've already seen, I retrace the same model steps shown earlier in the following sections, but this time show the reference to each object as the document acquires more objects.

A simple document

I start with the model whose only objects are the window (and its location object) and document from the simple HTML file. Figure 4-11 shows the object map and references for the two main objects. Every document resides in a window, so to reference the window object, you start with the object name, `window`. Also fixed in this reference is the document, since there can be only one document per window (or frame). Therefore, a reference to the document object is `window.document`.

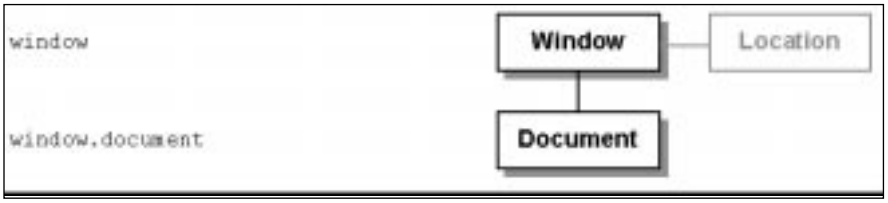

Figure 4-11: References to the window and document

Add a form

Modifying the document to include the empty <FORM> tag generates the form object in the map. If I did the job right, the <FORM> tag would also include a NAME attribute. The reference to the form object, as shown in Figure 4-12, starts with the window, wends through the document, and reaches the form, which I would call by name: `window.document.`*`formName`* (the italics meaning that in a real script, I would substitute the form's name for *`formName`*).

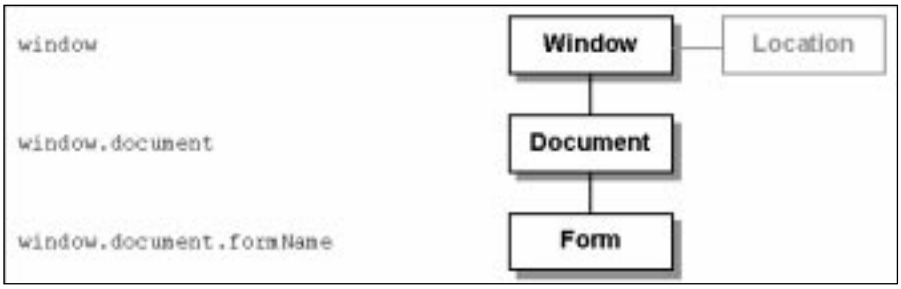

Figure 4-12: Reference to the form object

Add a text input element

As the hierarchy gets deeper, the object reference gets longer. In Figure 4-13, I have added a text input object to the form. The reference to this deeply nested object still starts at the window level, and works its way down to the name I assigned to the object in its <INPUT> tag: `window.document.`*`formName.textName`*.

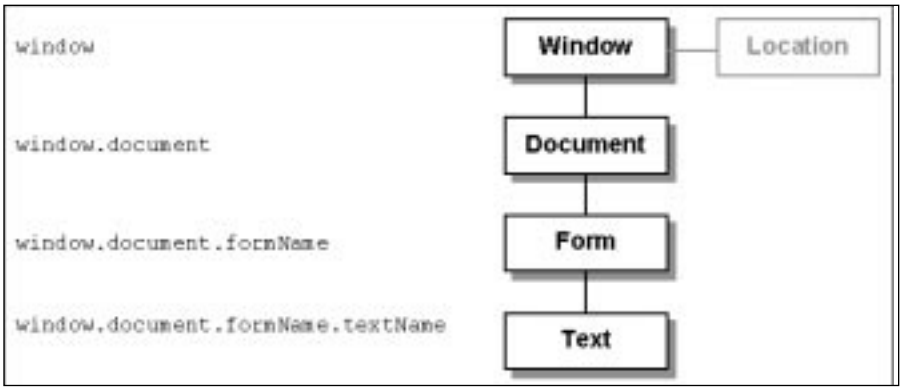

Figure 4-13: Reference to the text field object

Add a button element

When I add a button to the same form as the text object, the reference stays the same length (see Figure 4-14). All that changes is the last part of the reference, where the button name goes in place of the text field name: `window.document.`*`formName.buttonName`*.

About the Dot Syntax

JavaScript uses the period to separate components of a hierarchical reference. This convention is adopted from Java, which, in turn, based this formatting on the C language. Every reference typically starts with the most global scope — the window for client-side JavaScript — and narrows focus with each “dot” (.) delimiter.

If you have not programmed before, don't be put off by the dot syntax. You are probably already using it, such as when you access Usenet newsgroups. The methodology for organizing the thousands of newsgroups is to group them in a hierarchy that makes it relatively easy to both find a newsgroup and visualize where the newsgroup you're currently reading is located in the scheme of things.

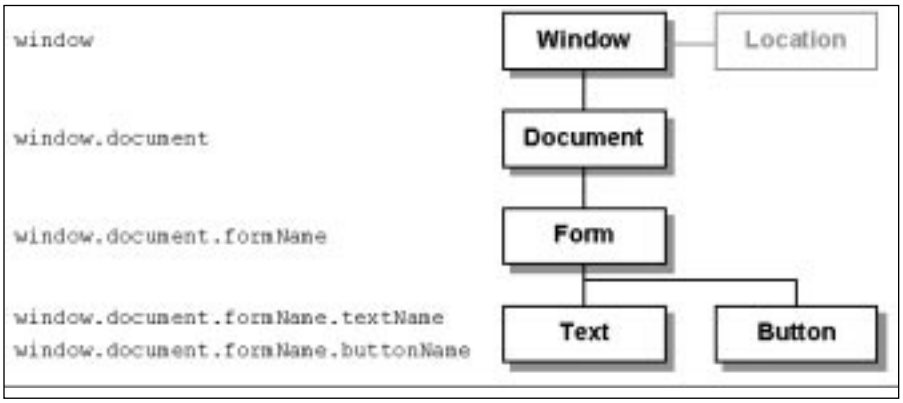

Figure 4-14: Reference to the button object

Newsgroup Organization Model

Let me briefly dissect a typical newsgroup address, to help you understand dot syntax: `rec.sport.skating.inline`. The first entry (at the left edge) defines the basic group — recreation — among all the newsgroup categories. Other group categories, such as `comp.` and `alt.`, have their own sections and do not overlap with what goes on in the `rec` section. Within the `rec` section are dozens of subsections, one of which is `sport`. That distinguishes all the sport-related groups from, say, the automobile or music groups within recreational newsgroups.

Like most broad newsgroup categories, `rec.sport` has many subcategories, each one devoted to a particular sport. In this case, it is skating. Other sport newsgroups include `rec.sport.rugby` and `rec.sport.snowboarding`. Even within the `rec.sport.skating` category, a further subdivision exists to help narrow the subject matter for participants. Therefore, a separate newsgroup just for inline skaters exists, just as a group for roller-skating exists (`rec.sport.skating.roller`). As a narrower definition is needed for a category, a new level is formed by adding a dot and a word to differentiate that subgroup from the thousands of newsgroups on the Net. When you ask your newsgroup software to view messages in the `rec.sport.skating.inline` group, you're giving it a map to follow in the newsgroup hierarchy to go directly to a single newsgroup.

Another benefit of this syntactical method is that names for subcategories can be reused within other categories, if necessary. For example, with this naming scheme, it is possible to have two similarly named subcategories in two separate newsgroup classifications, such as `rec.radio.scanners` and `alt.radio.scanners`. When you ask to visit one, the hierarchical address, starting with the `rec.` or `alt.` classification, ensures you get to the desired place. Neither

collection of messages is automatically connected with the other (although subscribers frequently cross-post to both newsgroups).

For complete newbies to the Net, this dot syntax can be intimidating. Because the system was designed to run on UNIX servers (the UNIX operating system is written in C), the application of a C-like syntax for newsgroup addressing is hardly surprising.

What Defines an Object?

When an HTML tag defines an object, the browser creates a slot for it in memory. But an object is far more complex internally than, say, a mere number stored in memory. The purpose of an object is to represent some "thing." Because in JavaScript you deal with items that appear in a browser window, an object may be an input text field, a button, or the whole HTML document. Outside of the pared-down world of a JavaScript browser, an object can also represent abstract entities, such as a calendar program's appointment entry or a paragraph in an object-oriented word processor.

Every object is unique in some way, even if it should look identical to you in the browser. Three very important facets of an object define what it is, what it looks like, how it behaves, and how scripts control it. Those three facets are *properties*, *methods*, and *event handlers*. They play such key roles in your future JavaScript efforts that the Object Road Map in Appendix A summarizes the properties, methods, and event handlers for each object. You might want to take a quick peek at that road map if for no other reason than to gain an appreciation for the size of the scripting vocabulary that consists of these object features.

Properties

Any physical object you hold in your hand has a collection of characteristics that defines it. A coin, for example, has a shape, diameter, thickness, color, weight, embossed images on each side, and any number of other attributes that distinguish a coin from, say, a feather. Each of those features is called a *property*. Each property has a value of some kind attached to it (even if the value is empty or null). For example, the shape property of a coin might be "circle" — in this case, a text value. In contrast, the denomination property is most likely a numeric value.

You may not have known it, but if you've written HTML for use in a scriptable browser, you have been setting object properties all along without writing one iota of JavaScript. Tag attributes are the most common way an HTML object's properties are set. The presence of JavaScript often adds optional attributes whose initial values can be set when the document loads. For example, the following HTML tag defines a button object that assigns two property values:

```
<INPUT TYPE="button" NAME="clicker" VALUE="Hit Me...">
```

In JavaScript parlance, then, the `name` property holds the word "clicker," while the `value` property is the text that appears on the button label, "Hit Me...". In truth, a button has more properties than just these, but you don't have to set every property for every object. Most properties have default values that are automatically assigned if nothing special is set in the HTML or later from a script.

The contents of some properties can change while a document is loaded and the user interacts with the page. Consider the following text input tag:

```
<INPUT TYPE="text" NAME="entry" VALUE="User Name?">
```

The `name` property of this object is the word "entry." When the page loads, the text of the `VALUE` attribute setting is placed in the text field — the automatic behavior of an HTML text field when the `VALUE` attribute is specified. But if a user enters some other text into the text field, the `value` property changes — not in the HTML, but in the memory copy of the object model that the browser maintains. Therefore, if a script queries the text field about the content of the `value` property, the browser yields the current setting of the property, which may not be the one specified by the HTML if a user changes the text.

To gain access to an object's property, you use the same kind of dot syntax, hierarchical addressing scheme you saw earlier for objects. Since a property is contained by its object, the reference to it consists of the reference to the object plus one more extension naming the property. Therefore, for the button and text object tags just shown, references to various properties would be

```
document.formName.clicker.name
document.formName.clicker.value
document.formName.entry.value
```

You may be wondering what happened to the `window` part of the reference. It turns out that since there can only be one document contained in a window, references to objects inside the document can omit the `window` portion and start the reference with `document`. The document object, however, cannot be omitted from the reference. Notice, too, that the button and text field both have a property named `value`. These properties represent very different attributes for each object. For the button, the property determines the button label; for the text field, the property reflects the current text in the field. You now see how the (sometimes lengthy) hierarchical referencing scheme helps the browser locate exactly the object and property your script needs. No two items in a document can have identical references.

Methods

If a property is like a descriptive adjective for an object, then a method is a verb. A method is all about action related to the object. A method either does something to the object or with the object that affects other parts of a script or document. They are commands of a sort, but whose behaviors are tied to a particular object.

An object can have any number of methods associated with it (including none at all). Method names are like property names except that the format for methods insists that they end with a pair of parentheses. To set a method into motion, a JavaScript statement must include a reference to it, via its object, as in the following examples:

```
document.orderForm.submit()
document.orderForm.entry.select()
```

The first is a scripted way of clicking on a Submit button to send a form to a server. The second selects the text inside a text field named `entry` (which is contained by a form named `orderForm`).

Sometimes a method requires that additional information be sent along with it so that it can do its job. Each chunk of information passed with the method is called a *parameter* or *argument* (you can use the terms interchangeably). You saw examples of passing a parameter in your first script in Chapter 3. Two script statements invoked the `write()` method of the document object:

```
document.write("This browser is version " + navigator.appVersion)
document.write(" of <B>" + navigator.appName + "</B>.")
```

As the page loaded into the browser, each `document.write()` method sent whatever text is inside the parentheses to the current document. In both cases, the content being sent as a parameter consisted of straight text (inside quotes) and the values of two object properties: the `appVersion` and `appName` properties of the navigator object (the navigator object does not appear in the object hierarchy diagram of Figure 4-6, because in Navigator, this object exists outside of the document object model).

As you learn more about the details of JavaScript and the document objects you can script, pay close attention to the range of methods defined for each object. They reveal a lot about what an object is capable of doing under script control.

Event handlers

One last characteristic of a JavaScript object is the event handler. Events are actions that take place in a document, usually as the result of user activity. Common examples of user actions that trigger events include clicking on a button or typing a character into a text field. Some events, such as the act of loading a document into the browser window or experiencing a network error while an image loads, are not so obvious.

Almost every JavaScript object in a document receives events of one kind or another — summarized for your convenience in the Object Road Map of Appendix A. What determines whether the object will do anything in response to the event is an extra attribute you enter into the object's HTML definition. The attribute consists of the event name, an equals sign (just like any HTML attribute), followed by instructions about what to do when the particular event fires. Listing 4-1 shows a very simple document that displays a single button with one event handler defined for it.

Listing 4-1: **A Simple Button with an Event Handler**

```
<HTML>
<BODY>
<FORM>
<INPUT TYPE="button" VALUE="Click Me" onClick="window.alert ('Ouch!')">
</FORM>
</BODY>
</HTML>
```

The form definition contains what, for the most part, looks like a standard input item. But notice the last attribute, `onClick="window.alert('Ouch!')"`. Button objects, as you see in their complete descriptions in Chapter 23, react to mouse clicks. When a user clicks on the button, the browser sends a click event to the button. In this button's definition, the attribute says that whenever the button receives that message, it should invoke one of the window object's methods, `alert()`. The alert method displays a simple alert dialog box whose content is whatever text is passed as a parameter to the method. Like most arguments to HTML attributes, the attribute setting to the right of the equals sign goes inside quotes. If additional quotes are necessary, as in the case of the text to be passed along with the event handler, those inner quotes can be single quotes. In actuality, JavaScript doesn't distinguish between single or double quotes but does require that each set be of the same type. Therefore, the attribute could have also been written

```
onClick='alert("Ouch!")'
```

Exercises

1. Which of the following applications are well-suited to client-side JavaScript, and why or why not?
   a. Music jukebox
   b. Web-site visit counter
   c. Chat room
   d. Graphical Fahrenheit-to-Celsius temperature calculator
   e. All of the above
   f. None of the above

2. General Motors has five divisions for its automobile brands: Chevrolet, Pontiac, Oldsmobile, Buick, and Cadillac. Each brand has several models of automobile. Following this hierarchy model, write the dot-syntax equivalent reference to the following three vehicle models:
   a. Chevrolet Malibu
   b. Pontiac Firebird
   c. Pontiac Bonneville

3. Which of the following object names are valid in JavaScript? For each one that is not valid, explain why.

a. lastName

b. company_name

c. 1stLineAddress

d. zip code

e. today's_date

4. An HTML document contains tags for one image and one form. The form contains tags for three text boxes, one checkbox, a Submit button, and a Reset button. Using the object hierarchy diagram from Figure 4-6 for reference, draw a diagram of the object model that Navigator would create in its memory for these objects. Give names to the image, form, text fields, and checkbox, and write the references to each of those objects.

5. Write the HTML tag for a button input element whose name is "Hi", whose visible label reads "Howdy", and whose `onClick=` event handler displays an alert dialog box that says "Hello to you, too!"

✦ ✦ ✦