CHAPTER 8

Window and Document Objects

In This Chapter

What the window object does

How to access key window object properties and methods

How to trigger script actions after a document loads

The purposes of the location and history objects

How the document object is created

How to access key document object properties and methods

Now that you have exposure to programming fundamentals, it will be easier to demonstrate how to script document objects. Starting with this lesson, the tutorial turns back to the document object model, diving more deeply into each of the objects you will place in many of your documents.

Document Objects

As a refresher, study the Netscape Navigator document object hierarchy in Figure 8-1. This lesson focuses on objects at or near the top of the hierarchy: window, location, history, and document. The goal is not only to equip you with the basics so you can script simple tasks, but also to prepare you for in-depth examinations of each object and its properties, methods, and event handlers in Part III of this book. I introduce only the basic properties, methods, and event handlers for objects in this tutorial — far more are to be found in Part III. Examples in that part of the book assume you know the programming fundamentals covered in previous lessons.

The Window Object

At the very top of the document object hierarchy is the window object. This object gains that exalted spot in the object food chain because it is the master container for all content you view in the Web browser. As long as a browser window is open — even if no document is loaded in the window — the window object is defined in the current model in memory.

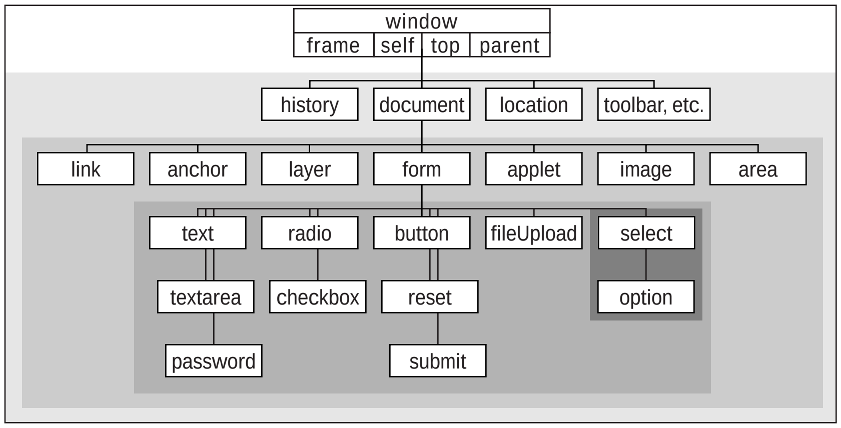

Figure 8-1: The Netscape Navigator 4 document object model

In addition to the content part of the window where documents go, a window's sphere of influence includes the dimensions of the window and all of the "stuff" that surrounds the content area. Netscape calls this area — where scrollbars, toolbars, the status bar, and menu bar on non-Macintosh versions live — a window's *chrome*. Not every browser or every version of Navigator has full scripted control over the chrome of the main browser window, but you can easily script the creation of additional windows sized the way you want and have only the chrome elements you wish to display in that subwindow.

Although the discussion about frames comes in Chapter 11, I can safely say now that each frame is also considered a window object. If you think about it, that makes sense, because each frame can hold a different document. When a script runs in one of those documents, it regards the frame that holds the document as the window object in its view of the object hierarchy.

As you will see here, the window object is a convenient place for the document object model to attach methods that display modal dialog boxes and adjust the text that displays in the status bar at the bottom of the browser window. Another window object method lets you create a separate window that appears on the screen. When you look at all of the properties, methods, and event handlers defined for the window object in Netscape's object model (see Appendix A), it should be clear why they are attached to window objects — visualize their scope and the scope of a browser window.

Accessing window properties and methods

Script references to properties and methods of the window object can be worded in several ways, depending more on whim and style than on specific syntactical requirements. The most logical and common way to compose such references includes the window object in the reference:

```
window.propertyName
window.methodName([parameters])
```

A window object also has a synonym when the script doing the referencing is pointing to the window that houses the document. The synonym is `self`. Reference syntax then becomes

```
self.propertyName
self.methodName([parameters])
```

You can use these initial reference object names interchangeably, but I tend to reserve the use of `self` for more complex scripts that involve multiple frames and windows. The `self` moniker more clearly denotes the current window holding the script's document. To me, it makes the script more readable — by me and by others.

Back in Chapter 4, I indicated that because the window object is always "there" when a script runs, you can omit it from references to any objects inside that window. Therefore, the following syntax models assume properties and methods of the current window:

```
propertyName
methodName([parameters])
```

In fact, as you will see in a few moments, some methods may be more understandable if you omit the window object reference. The methods run just fine either way.

Creating a window

A script does not create the main browser window. A user does that by virtue of launching the browser or by opening a URL or file from the browser's menus (if the window is not already open). But a script can generate any number of subwindows once the main window is open (and contains a document whose script needs to open subwindows).

The method that generates a new window is `window.open()`. This method contains up to three parameters that define window characteristics, such as the URL of the document to load, its name for `TARGET` reference purposes in HTML tags, and physical appearance (size and chrome contingent). I won't go into the details of the parameters here (they're covered in great depth in Chapter 14), but I do want to expose you to an important concept involved with the `window.open()` method.

Consider the following statement that opens a new window to a specific size and with an HTML document from the server:

```
var subWindow =
window.open("definition.html","def","HEIGHT=200,WIDTH=300")
```

The important part of this statement is that it is an assignment statement. Something gets assigned to that variable `subWindow`. What is it? It turns out that when the `window.open()` method runs, it not only opens up that new window according to specifications set as parameters, but it evaluates to a reference to that new window. In programming parlance, the method is said to *return* a value — in this case, a genuine object reference. The value returned by the method is assigned to the variable.

Your script can now use that variable as a valid reference to the second window. If you need to access one of its properties or methods, you must use that reference as part of the complete reference. For example, to close the subwindow from a script in the main window, the reference to the `close()` method for that subwindow would be

```
subWindow.close()
```

If you were to issue `window.close()`, `self.close()`, or just `close()` in the main window's script, the method would close the main window, and not the subwindow. To address another window, then, you must include a reference to that window as part of the complete reference. This will have an impact on your code, because you probably want the variable holding the reference to the subwindow to be valid as long as the main document is loaded into the browser. For that to happen, the variable will have to be initialized as a global variable, rather than inside a function (although its value can be set inside a function). That way, one function can open the window while another closes it.

Listing 8-1 is a page that has a button for opening a blank new window and closing that window from the main window. To view this demonstration, shrink your main browser window to less than full screen. Then when the new window is generated, reposition the windows so you can see the smaller new window when the main window is in front. The key point of Listing 8-1 is that the `newWindow` variable is defined as a global variable so that both functions have access to it. When a variable is declared with no value assignment, its value is null. It turns out that a null value is the same as `false` in a condition, while any value is the same as `true` in a condition. Therefore, in the `closeNewWindow()` function, the condition tests whether the window has been created before issuing the subwindow's `close()` method.

Listing 8-1: **References to Window Objects**

```
<HTML>
<HEAD>
<TITLE>Window Opener and Closer</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var newWindow
function makeNewWindow() {
       newWindow = window.open("","","HEIGHT=300,WIDTH=300")
}
function closeNewWindow() {
       if (newWindow) {
          newWindow.close()
       }
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
<INPUT TYPE="button" VALUE="Create New Window"
onClick="makeNewWindow()">
<INPUT TYPE="button" VALUE="Close New Window"
onClick="closeNewWindow()">
```

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

Window Properties and Methods

The one property and three methods for the window object described in this lesson have an immediate impact on user interaction. They work with all scriptable browsers. Extensive code examples can be found in Part III for each property and method.

window.status property

The status bar at the bottom of the browser window normally displays the URL of a link when you roll the mouse pointer atop it. Other messages also appear in that space during document loading, Java applet initialization, and the like. However, you can use JavaScript to display your own messages in the status bar at times that may be beneficial to your users. For example, rather than display the URL of a link, you can display a friendlier plain-language description of the page at the other end of the link (or a combination of both to accommodate both newbies and geeks).

The `window.status` property can be assigned some other text at any time. To change the text of a link, the action is triggered by an `onMouseOver=` event handler of a link object. A peculiarity of the `onMouseOver=` event handler for setting the status bar is that an additional statement — `return true` — must also be part of the event handler. This is very rare in JavaScript, but required here for the status bar to be successfully overridden by your script.

Due to the simplicity of setting the `window.status` property, it is most common for the script statements to be run as inline scripts in the event handler definition. This is handy for short scripts, because you don't have to specify a separate function or add `<SCRIPT>` tags to your page. You simply add the script statements to the `<A>` tag:

```
<A HREF="http://home.netscape.com" onMouseOver="window.status='Visit
the Netscape Home page (home.netscape.com)'; return true">Netscape</A>
```

Look closely at the script statements assigned to the `onMouseOver=` event handler. The two statements are

```
window.status='Visit the Netscape Home page (home.netscape.com)'
return true
```

When run as inline scripts, the two statements must be separated by a semicolon. Equally important, the entire set of statements is surrounded by double quotes(`"..."`). To nest the string being assigned to the `window.status` property inside the double-quoted script, the string is surrounded by single quotes (`'...'`). You get a lot of return for a little bit of script when you set the status bar. The downside is that scripting this property is how those awful status bar scrolling banners are created. Yech!

window.alert() method

I have already used the `alert()` method many times so far in this tutorial. This window method generates a dialog box that displays whatever text you pass as a parameter (see Figure 8-2). A single OK button (which cannot be changed) lets the user dismiss the alert.

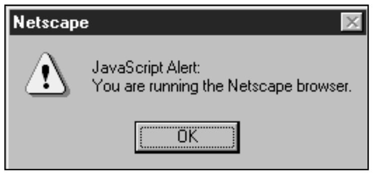

Figure 8-2: A JavaScript alert dialog box

The appearance of this and two other JavaScript dialog boxes (described next) has changed slightly since the first scriptable browsers. In versions prior to Navigator 4 (as shown in Figure 8-2), the browser inserted words clearly indicating that the dialog box was a "JavaScript Alert." This text cannot be altered by script: Only the other message content can be changed.

All three dialog box methods are good cases for using a window object's methods without the reference to the window. Even though the `alert()` method is technically a window object method, no special relationship exists between the dialog box and the window that generates it. In production scripts, I rarely use the full reference:

```
alert("This is a JavaScript alert dialog.")
```

window.confirm() method

The second style of dialog box presents two buttons (Cancel and OK in most versions on most platforms) and is called a confirm dialog (see Figure 8-3). More importantly, this is one of those methods that returns a value: `true` if the user clicks OK, `false` if the user clicks Cancel. You can use this dialog and its returned value as a way to have a user make a decision about how a script will progress.

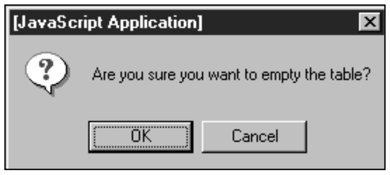

Figure 8-3: A JavaScript confirm dialog box

Because the method always returns a Boolean value, you can use the evaluated value of the entire method as a condition statement in an `if` or `if...else` construction. For example, in the following code fragment, the user is asked about starting the application over. Doing so causes the default page of the site to be loaded into the browser:

```
if (confirm("Are you sure you want to start over?")) {
       location = "index.html"
}
```

window.prompt() method

The final dialog box of the window object, the prompt dialog box (see Figure 8-4), displays a message that you set and provides a text field for the user to enter a response. Two buttons, Cancel and OK, let the user dismiss the dialog box with two opposite expectations: canceling the entire operation or accepting the input typed into the dialog box.

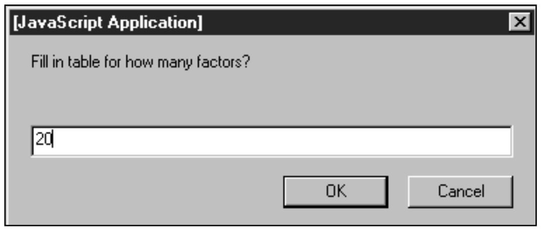

Figure 8-4: A JavaScript prompt dialog box

The `window.prompt()` method has two parameters. The first is the message that acts as a prompt to the user. You can suggest a default answer in the text field by including a string as the second parameter. If you don't want any default answer to appear, then include an empty string (two double quotes without any space between them).

This method returns one value when the user clicks on either button. A click of the Cancel button returns a value of `null`, regardless of what is typed into the field. A click of the OK button returns a string value of the typed entry. Your scripts can use this information in conditions for `if` and `if...else` constructions. A value of `null` is treated as `false` in a condition. It turns out that an empty string is also treated as `false`. Therefore, a condition can easily test for the presence of real characters typed into the field to simplify a condition test, as shown in the following fragment:

```
var answer = prompt("What is your name?","")
if (answer) {
       alert("Hello, " + answer + "!")
}
```

The only time the `alert()` method is called is when the user has entered something into the prompt dialog and clicked the OK button.

onLoad= event handler

The window object reacts to several system and user events, but the one you will probably use most is the event that fires as soon as everything in a page has finished loading. This event waits for images, Java applets, and data files for plug-ins to download fully to the browser. It can be dangerous to script access to elements of a document object while the page loads, because if the object has not yet loaded (perhaps due to a slow network connection or server), a script error will result. The advantage of using the `onLoad=` event to invoke functions is that you are assured that all document objects are in the browser's document object model. All window event handlers are placed inside the `<BODY>` tag. Even though you will come to associate the `<BODY>` tag's attributes with the document object's properties, it is the window object's event handlers that go inside the tag (the document object has no event handlers).

The Location Object

Sometimes an object in the hierarchy represents something that doesn't seem to have a physical presence, like a window or a button. That's the case with the location object. This object represents the URL loaded into the window. This is different from the document object (discussed later in this lesson), because the document is the real content; the location is simply the URL.

Unless you are truly Web-savvy, you may not realize a URL consists of many components that define the address and method of data transfer for a file. Pieces of a URL include the protocol (like `http:`) and the hostname (like `www.giantco.com`). All of these items are accessed as properties of the location object. For the most part, though, your scripts will be interested in only one property: the `href` property, which defines the complete URL.

Setting the `location.href` property is the primary way your scripts navigate to other pages, either in the current window or another frame. You can generally navigate to a page in your own Web site by specifying a relative URL (that is, relative to the currently loaded page), rather than the complete URL with protocol and host info. For pages outside of your domain, you need to specify the complete URL.

A shortcut that works well in Navigator is to omit the reference to the `href` property. You can simply set the location object to a URL (relative or absolute), and Navigator will get to the desired page. Therefore, both of the following statements accomplish the same end:

```
location = "http://www.dannyg.com"
location.href = "http://www.dannyg.com"
```

If the page to be loaded is in another window or frame, the window reference must be part of the statement. For example, if your script opens a new window and assigns its reference to a variable named `newWindow`, the statement that loads a page into that window would be

```
newWindow.location = "http://www.dannyg.com"
```

The History Object

Another object that doesn't have a physical presence on the page is the history object. Each window maintains a list of recent pages that have been visited by the browser. While the history object's list contains the URLs of recently visited pages, those URLs are not generally accessible by script. But methods of the history object allow for navigating backward and forward through the history relative to the currently loaded page.

The Document Object

The document object holds the real content of the page. Properties and methods of the document object generally affect the look and content of the document that occupies the window. Except for some of the advances in Dynamic HTML in Internet Explorer 4, the text contents of a page cannot be accessed or changed once the document has loaded. However, as you saw in your first script of Chapter 3, the `document.write()` method lets a script dynamically create content as the page is loading. A great many of the document object's properties are established by attributes of the `<BODY>` tag. Many other properties are arrays of other objects in the document.

Accessing a document object's properties and methods is straightforward, as shown in the following syntax examples:

```
[window.]document.propertyName
[window.]document.methodName([parameters])
```

The window reference is optional when the script is accessing the document object that contains the script.

document.forms[] property

One of the object types contained by a document is the form object. Because there can conceivably be more than one form in a document, forms are stored as arrays in the `document.forms[]` property. As you recall from the discussion of arrays in Chapter 7, an index number inside the square brackets points to one of the elements in the array. To access the first form in a document, for example, the reference would be

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

In general, however, I recommend that you access a form by way of a name you assign to the form in its `NAME` attribute, as in

```
document.formName
```

Either methodology reaches the same object. When a script needs to reference elements inside a form, the complete address to that object must include the document and form.

document.title property

Not every property about a document object is set in a `<BODY>` tag attribute. If you assign a title to the page in the `<TITLE>` tag set within the Head portion, that title text is reflected by the `document.title` property. A document's title is mostly a cosmetic setting that gives a plain-language name of the page appearing in the browser's title bar, as well as the user's history listing and bookmark of your page.

document.write() method

The `document.write()` method can be used in both immediate scripts to create content in a page as it loads and in deferred scripts that create new content in the same or different window. The method requires one string parameter, which is the HTML content to write to the window or frame. Such string parameters can be variables or any other expressions that evaluate to a string. Very often, the content being written includes HTML tags.

Bear in mind that after a page loads, the browser's output stream is automatically closed. After that, any `document.write()` method issued to the current page opens a new stream that immediately erases the current page (along with any variables or other values in the original document). Therefore, if you wish to replace the current page with script-generated HTML, you need to accumulate that HTML in a variable and perform the writing with just one `document.write()` method. You don't have to explicitly clear a document and open a new data stream: One `document.write()` call does it all.

One last piece of housekeeping advice about the `document.write()` method involves its companion method, `document.close()`. Your script must close the output stream when it has finished writing its content to the window (either the same window or another). After the last `document.write()` method in a deferred script, be sure to include a `document.close()` method. Failure to do this may cause images and forms not to appear; also, any `document.write()` method invoked later will only append to the page, rather than clear the existing content to write anew. To demonstrate the `document.write()` method, I show two versions of the same application. One writes to the same document that contains the script; the other writes to a separate window. Type in each document, save it, and open it in your browser.

Listing 8-2 creates a button that assembles new HTML content for a document, including HTML tags for a new document title and color attribute for the `<BODY>` tag. One `document.write()` statement blasts the entire new content to the same document, obliterating all vestiges of the content of Listing 8-2. The `document.close()` statement, however, is required to properly close the output stream. When you load this document and click the button, notice that the document title in the browser's title bar changes accordingly. As you click back to the original and try the button again, notice that the dynamically written second page loads much faster than even a reload of the original document.

Listing 8-2: **Using document.write() on the Same Window**

```
<HTML>
<HEAD>
<TITLE>Writing to Same Doc</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function reWrite() {
        // assemble content for new window
        var newContent = "<HTML><HEAD><TITLE>A New Doc</TITLE></HEAD>"
        newContent += "<BODY BGCOLOR='aqua'><H1>This document is brand
new.</H1>"
        newContent += "Click the Back button to see original document."
        newContent += "</BODY></HTML>"
        // write HTML to new window document
        document.write(newContent)
        document.close() // close layout stream
}
```

```
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" VALUE="Replace Content" onClick="reWrite()">
</FORM>
</BODY>
</HTML>
```

In Listing 8-3, the situation is a bit more complex because the script generates a subwindow, to which is written an entirely script-generated document. To keep the reference to the new window alive across both functions, the `newWindow` variable is declared as a global variable. As soon as the page loads, the `onLoad=` event handler invokes the `makeNewWindow()` function. This function generates a blank subwindow. I have added a property to the third parameter of the `window.open()` method that instructs the status bar of the subwindow appear.

A button in the page invokes the `subWrite()` method. The first task it performs is to check the `closed` property of the subwindow. This property (which exists only in newer browser versions) returns `true` if the referenced window is closed. If that's the case (in case the user has manually closed the window), the function invokes the `makeNewWindow()` function again to open that window again.

With the window open, new content is assembled as a string variable. As with Listing 8-2, the content is written in one blast (although that isn't necessary for a separate window), followed by a `close()` method. But notice an important difference: both the `open()` and `close()` methods explicitly specify the subwindow.

Listing 8-3: **Using document.write() on Another Window**

```
<HTML>
<HEAD>
<TITLE>Writing to Sub Window</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var newWindow
function makeNewWindow() {
       newWindow = window.open("","","status,height=200,width=300")
}

function subWrite() {
       // make new window if someone has closed it
       if (newWindow.closed) {
          makeNewWindow()
       }
       // assemble content for new window
       var newContent = "<HTML><HEAD><TITLE>A New Doc</TITLE></HEAD>"
       newContent += "<BODY BGCOLOR='coral'><H1>This document is brand
new.</H1>"
```

(continued)

Listing 8-3 *Continued*

```
        newContent += "</BODY></HTML>"
        // write HTML to new window document
        newWindow.document.write(newContent)
        newWindow.document.close() // close layout stream
}
</SCRIPT>
</HEAD>
<BODY onLoad="makeNewWindow()">
<FORM>
<INPUT TYPE="button" VALUE="Write to Subwindow" onClick="subWrite()">
</FORM>
</BODY>
</HTML>
```

The Link Object

Belonging to the document object in the hierarchy is the link object. A document can have any number of links, so references to links, if necessary, are usually made via the array index method:

```
document.links[n].propertyName
```

More commonly, though, links are not scripted. However, there is an important JavaScript component to these objects. When you want the click on a link to execute a script rather than navigate directly to another URL, you can redirect the `HREF` attribute to call a script function. The technique involves a pseudo-URL called the `javascript:` URL. If you place the name of a function after the `javascript:` URL, a scriptable browser runs that function. So as not to mess with the minds of users, the function should probably perform some navigation in the end, but the script can do other things as well, such as simultaneously changing the content of two frames within a frameset.

The syntax for this construction in a link is as follows:

```
<A HREF="javascript:void
functionName([parameter1]...[parameterN])">...</A>
```

The `void` keyword prevents the link from trying to display any value that may be returned from the function. Remember this `javascript:` URL technique for all tags that include `HREF` and `SRC` attributes: If an attribute accepts a URL, it can accept this `javascript:` URL as well. This can come in handy as a way to script actions for client-side image maps that don't necessarily navigate anywhere, but cause something to happen on the page just the same.

The next logical step past the document level in the object hierarchy is the form. That's where you will spend the next lesson.

Exercises

1. Which of the following references are valid and which are not? Explain what is wrong with the invalid references.

   a. `window.document.form[0]`

   b. `self.entryForm.entryField.value`

   c. `document.forms[2].name`

   d. `entryForm.entryField.value`

   e. `newWindow.document.write("Howdy")`

2. Write the JavaScript statement that displays a message in the status bar welcoming visitors to your Web page.
3. Write the JavaScript statement that displays the same message to the document as an `<H1>`-level headline on the page.
4. Create a page that prompts the user for his or her name as the page loads (via a dialog box) and then welcomes the user by name in the body of the page.
5. Create a page with any content you like, but one that automatically displays a dialog box after the page loads to show the user the URL of the current page.

✦ ✦ ✦