The Document Object

16

CHAPTER

In This Chapter

Accessing arrays of objects contained by the document object

Changing content colors

Writing new document content to a window or frame

User interaction is a vital aspect of client-side JavaScript scripting, and most of the communication between script and user takes place by way of the document object and its components. Understanding the scope of the document object is key to knowing how far you can take JavaScript.

Review the document object's place within the JavaScript object hierarchy. Figure 16-1 clearly shows that the document object is a pivotal point for a large percentage of JavaScript objects.

In fact, the document object and all that it contains is so big that I have divided its discussion into many chapters, each focusing on related object groups. This chapter looks only at the document object, while each of the eight succeeding chapters details objects contained by the document object.

I must stress at the outset that many newcomers to JavaScript have the expectation that they can, on the fly, modify sections of a loaded page's content with ease: replace some text here, change a table cell there. It's very important, however, to understand that except for a limited number of JavaScript objects, Netscape's document object model does not allow a lot of content manipulation after a page has loaded. The items that can be modified on the fly include text object values, textarea object values, images (starting with Navigator 3), and select object list contents

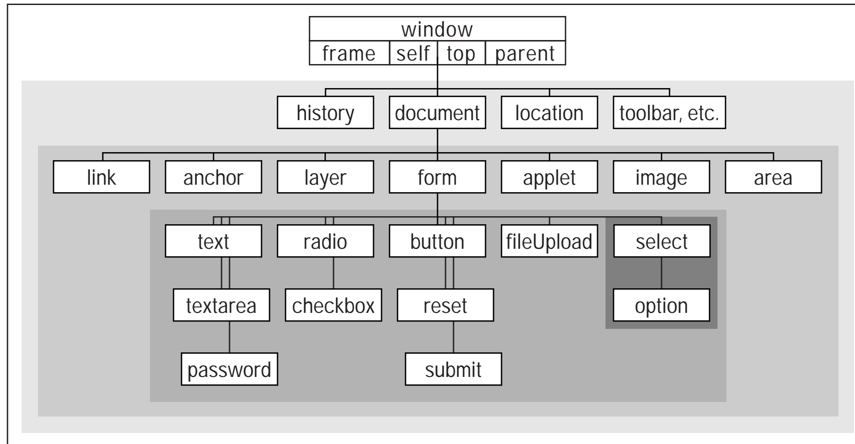


Figure 16-1: The JavaScript object hierarchy

A handful of other invisible properties are modifiable after the fact, but their settings don't survive soft reloads of a document. If your pages need to modify their contents based on user input or timed updates, consider designing your pages so that scripts write the contents; then let the scripts rewrite the entire page with your new settings.

Dynamic HTML and Documents

Navigator 4 and Internet Explorer 4 usher in a new concept called Dynamic HTML (DHTML). I devote Chapters 41 through 43 to the concepts behind DHTML. One of the advantages of this new page construction technique is that more content can, in fact, be altered on the fly after a document has loaded. Many of the roadblocks to creativity in earlier browser versions have been shattered with DHTML. Unfortunately, Netscape and Microsoft are not yet on the same page of the playbook when it comes to implementing scriptable interfaces to DHTML. Some common denominators exist, thanks to the W3C standards body, but both companies have numerous extensions that operate on different principles.

The fundamental difference is in the way each company implements content holders that our scripts can modify. Netscape relies on a new HTML `<LAYER>` tag and layer object; Microsoft has essentially turned every existing content-related tag into an object in the Internet Explorer 4 document object model.

Both methodologies have their merits. I like the ability to change text or HTML for any given element in an Internet Explorer 4 page. At the same time, Netscape's layer object, despite the HTML tag proliferation it brings, is a convenient container for a number of interesting animation effects. Because the point of view of this book is from that of Navigator, my assumption is you are designing primarily (if not exclusively) for a Netscape user audience, with the need to be compatible with Internet Explorer users. Therefore, if you see that I am glossing over a favorite Internet Explorer–only feature of yours, I do so to keep the discussion focused on Navigator applications, not to denigrate Microsoft's accomplishments.

Document Object

Document Object Properties	*Methods*	*Event Handlers*
alinkColor	captureEvents()	(None)
anchors[]	clear()	
applets[]	close()	
bgColor	getSelection()	
cookie	handleEvent()	
cookie	open()	
domain	releaseEvents()	
embeds	routeEvent()	
fgColor	write()	
forms[]	writeln()	
images[]		
lastModified		
layers[]		
linkColor		
links[]		
location		
referrer		
title		
URL		
vlinkColor		

Syntax

Creating a document:

```
<BODY
    [BACKGROUND="backgroundImageURL"]
    [BGCOLOR="#backgroundColor"]
    [TEXT="#foregroundColor"]
    [LINK="#unfollowedLinkColor"]
    [ALINK="#activatedLinkColor"]
    [VLINK="#followedLinkColor"]
    [onClick="handlerTextOrFunction"]
    [onDblClick="handlerTextOrFunction"]
    [onMouseDown="handlerTextOrFunction"]
```

```
        [onMouseUp="handlerTextOrFunction"]
        [onKeyDown="handlerTextOrFunction"]
        [onKeyPress="handlerTextOrFunction"]
        [onKeyUp="handlerTextOrFunction"]
        [onLoad="handlerTextOrFunction"]
        [onUnload="handlerTextOrFunction"]
        [onBlur="handlerTextOrFunction"]
        [onFocus="handlerTextOrFunction"]
        [onMove="handlerTextOrFunction"]
        [onResize="handlerTextOrFunction"]
        [onDragDrop="handlerTextOrFunction"]>
</BODY>
```

Accessing document properties or methods:

```
[window.] document.property | method([parameters])
```

About this object

A document object is the totality of what exists inside the content region of a browser window or window frame (excluding toolbars, status lines, and so on). The document is a combination of the content and interface elements that make the Web page worth visiting.

The officially sanctioned syntax for creating a document object, shown above, may mislead you to think that only elements defined within `<BODY>` tags comprise a document object. In truth, some `<HEAD>` tag information, such as `<TITLE>` and, of course, any scripts inside `<SCRIPT>` tags, are part of the document as well. So are some other values (properties), including the date on which the disk file of the document was last modified and the URL from which the user reached the current document.

Many event handlers defined in the Body, such as `onLoad=` and `onUnload=`, are not document-event handlers but rather window-event handlers. Load and unload events are sent to the window after the document finishes loading and just prior to the document being cleared from the window, respectively. See Chapter 14's discussion about the window object for more details about these and other window events whose event handlers are placed in the `<BODY>` tag.

Another way to create a document is to use the `document.write()` method to blast some or all of an HTML page into a window or frame. The window may be the current window running a script, a subwindow created by the script, or another frame in the current frameset. If you are writing the entire document, it is good practice to write a formal HTML page with all the tags you would normally put into an HTML file on your server.

Properties

alinkColor
vlinkColor
bgColor
fgColor
linkColor

Value: Hexadecimal triplet string **Gettable:** Yes **Settable:** Limited

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

Netscape began using these `<BODY>` attributes for various color settings with Navigator Version 1.1. Many other browsers now accept these attributes, and they are part of HTML Level 3.2. All five settings can be read via scripting, but the ability to change some or all of these properties varies widely with browser and client platform. Table 16-1 shows a summary of which browsers and platforms can set which of the color properties. Notice that only `document.bgColor` is adjustable on the fly in Navigator browsers.

Table 16-1
Setting Document Colors on the Fly (Browser Versions)

	Navigator			*Internet Explorer*		
Color Property	*Windows*	*Mac*	*UNIX*	*Windows*	*Mac*	*UNIX*
`bgColor`	All	4	4	All	All	4
All others	None	None	None	All	All	4

If you experiment with setting `document.bgColor` on Mac or UNIX versions of Navigator 2 and 3, you may be fooled into thinking that the property is being set correctly. While the property value may stick, these platforms do not refresh their windows properly: if you change the color after all content is rendered, the swath of color obscures the content until a reload of the window. The safest, backward-compatible scripted way of setting document color properties is to compose the content of a frame or window and set the `<BODY>` tag color attributes dynamically.

Values for all color properties can be either the common HTML hexadecimal triplet value (for example, `"#00FF00"`) or any of the Netscape color names. Internet Explorer recognizes these plain language color names, as well. But also be aware that some colors work only when the user has the monitor set to 16- or 24-bit color settings.

JavaScript object property names are case-sensitive. This is important for the five property names that begin with lowercase letters and have an uppercase C within them.

Example

I've selected some color values at random to plug into three settings of the ugly colors group for Listing 16-1. The smaller window displays a dummy button so you can see how its display contrasts with color settings. Notice that the script sets the colors of the smaller window by rewriting the entire window's HTML code. After changing colors, the script displays the color values in the original window's textarea. Even though some colors are set with the Netscape color constant values, properties come back in the hexadecimal triplet values. You can experiment to your heart's content by changing color values in the listing. Every time you change the values in the script, save the HTML file and reload it in the browser.

Listing 16-1: Color Sampler

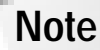
```
<HTML>
<HEAD>
<TITLE>Color Me</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function defaultColors() {
       return "BGCOLOR='#c0c0c0' VLINK='#551a8b' LINK='#0000ff'"
}

function uglyColors() {
       return "BGCOLOR='yellow' VLINK='pink' LINK='lawngreen'"
}
function showColorValues() {
       var result = ""
       result += "bgColor: " + newWindow.document.bgColor + "\n"
       result += "vlinkColor: " + newWindow.document.vlinkColor + "\n"
       result += "linkColor: " + newWindow.document.linkColor + "\n"
       document.forms[0].results.value = result
}
// dynamically writes contents of another window
function drawPage(colorStyle) {
       var thePage = ""
       thePage += "<HTML><HEAD><TITLE>Color Sampler</TITLE></HEAD><BODY
"
       if (colorStyle == "default") {
          thePage += defaultColors()
       } else {
          thePage += uglyColors()
       }
       thePage += ">Just so you can see the variety of items and
color, <A "
       thePage += "HREF='http://www.nowhere.com'>here's a link</A>,
and <A HREF='http://home.netscape.com'> here is another link </A> you
can use on-line to visit and see how its color differs from the
standard link."
```

```
        thePage += "<FORM>"
        thePage += "<INPUT TYPE='button' NAME='sample' VALUE='Just a
Button'>"
        thePage += "</FORM></BODY></HTML>"
        newWindow.document.write(thePage)
        newWindow.document.close()
        showColorValues()
}
// the following works properly only in Windows Navigator
function setColors(colorStyle) {
        if (colorStyle == "default") {
           document.bgColor = "#c0c0c0"
        } else {
           document.bgColor = "yellow"
        }
}
var newWindow = window.open("","","height=150,width=300")
</SCRIPT>
</HEAD>

<BODY>
Try the two color schemes on the document in the small window.
<FORM>
<INPUT TYPE="button" NAME="default" VALUE='Default Colors'
onClick="drawPage('default')">
<INPUT TYPE="button" NAME="weird" VALUE="Ugly Colors"
onClick="drawPage('ugly')"><P>
<TEXTAREA NAME="results" ROWS=3 COLS=20></TEXTAREA><P><HR>
These buttons change the current document, but not correctly on all
platforms<P>
<INPUT TYPE="button" NAME="default" VALUE='Default Colors'
onClick="setColors('default')">
<INPUT TYPE="button" NAME="weird" VALUE="Ugly Colors"
onClick="setColors('ugly')"><P>
</FORM>
<SCRIPT LANGUAGE="JavaScript">
drawPage("default")
</SCRIPT>
</BODY>
```

To satisfy the curiosity of those who want to change the color of a loaded document on the fly, the preceding example includes a pair of buttons that set the color properties of the current document. If you're running browsers and versions capable of this power (see Table 16-1), everything will look fine; but in other platforms, you may lose the buttons and other document content behind the color. You can still click and activate these items, but the color obscures them. Unless you know for sure that users of your Web page use only browsers and clients empowered for background color changes, do not change colors by setting properties of an existing document. And if you set the other color properties for Internet Explorer users, the settings are ignored safely by Navigator.



If you are using Internet Explorer 3 for the Macintosh, you will experience some difficulties with Listing 16-1. The script in the main document loses its connection with the subwindow; it does not redraw the second window with other colors. You can, however, change the colors in the main document. The significant flicker you may experience is related to the way the Mac version redraws content after changing colors.

Related Items: `document.links` property.

anchors

Value: Array of anchor objects **Gettable:** Yes **Settable:** No

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

Anchor objects (described in Chapter 17) are points in an HTML document marked with `<A NAME="">` tags. Anchor objects are referenced in URLs by a trailing hash value. Like other object properties that contain a list of nested objects, the `document.anchors` property (notice the plural) delivers an indexed array of anchors in a document. Use the array references to pinpoint a specific anchor for retrieving any anchor property.

Anchor arrays begin their index counts with 0: The first anchor in a document, then, has the reference `document.anchors[0]`. And, as is true with any built-in array object, you can find out how many entries the array has by checking the `length` property. For example

```
anchorCount = document.anchors.length
```

The `document.anchors` property is read-only (and its array entries come back as null). To script navigation to a particular anchor, assign a value to the `window.location` or `window.location.hash` object, as described in Chapter 15's location object discussion.

Example

In Listing 16-2, I appended an extra script to a listing from Chapter 15 to demonstrate how to extract the number of anchors in the document. The document dynamically writes the number of anchors found in the document. You will not likely ever need to reveal such information to users of your page, and the `document.anchors` property is not one that you will call frequently. The object model defines it automatically as a document property while defining actual anchor objects.

Listing 16-2: Reading the Number of Anchors

```
<HTML>
<HEAD>
```

```
<TITLE>document.anchors Property</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function goNextAnchor(where) {
        window.location.hash = where
}
</SCRIPT>
</HEAD>

<BODY>

<A NAME="start"><H1>Top</H1></A>
<FORM>
<INPUT TYPE="button" NAME="next" VALUE="NEXT"
onClick="goNextAnchor('sec1')">
</FORM>
<HR>

<A NAME="sec1"><H1>Section 1</H1></A>
<FORM>
<INPUT TYPE="button" NAME="next" VALUE="NEXT"
onClick="goNextAnchor('sec2')">
</FORM>
<HR>

<A NAME="sec2"><H1>Section 2</H1></A>
<FORM>
<INPUT TYPE="button" NAME="next" VALUE="NEXT"
onClick="goNextAnchor('sec3')">
</FORM>
<HR>

<A NAME="sec3"><H1>Section 3</H1></A>
<FORM>
<INPUT TYPE="button" NAME="next" VALUE="BACK TO TOP"
onClick="goNextAnchor('start')">
</FORM>
<HR><P>
<SCRIPT LANGUAGE="JavaScript">
document.write("<I>There are " + document.anchors.length + " anchors
defined for this document</I>")
</SCRIPT>
</BODY>
</HTML>
```

Related Items: anchor object; location object; `document.links` property.

applets

Value: Array of applet objects **Gettable:** Yes **Settable:** No

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

The `applets` property refers to Java applets defined in a document by the `<APPLET>` tag. An applet is not officially an object in the document until the applet loads completely.

Most of the work you do with Java applets from JavaScript takes place via the methods and variables defined inside the applet. Although you can reference an applet according to its indexed array position, you will more likely use the applet object's name in the reference to avoid any confusion. For more details, see the discussion of the applet object later in this chapter and the LiveConnect discussion in Chapter 38.

Example

The `document.applets` property is defined automatically as the browser builds the object model for a document that contains applet objects. You will rarely access this property, except to determine how many applet objects a document has.

Related Items: applet object.

cookie

Value: String **Gettable:** Yes **Settable:** Yes

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

The cookie mechanism in Navigator lets you store small pieces of information on the client computer in a reasonably secure manner. In other words, when you need some tidbit of information to persist at the client level while either loading diverse HTML documents or moving from one session to another, the cookie mechanism saves the day. Netscape's technical documentation (much of which is written from the perspective of a server writing to a cookie) can be found on the Web at `http://www.netscape.com/newsref/std/cookie_spec.html`.

The cookie is commonly used as a means to store the username and password you enter into a password-protected Web site. The first time you enter this information into a CGI-governed form, the CGI program has Navigator write the information back to a cookie on your hard disk (usually after encrypting the password). Rather than bothering you to enter the username and password the next time you access the site, the server searches the cookie data stored for that particular server and extracts the username and password for automatic validation processing behind the scenes.

Note

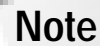
I cover the technical differences between Navigator and Internet Explorer cookies later in this section. But if you are using Internet Explorer 3, be aware that the browser neither reads nor writes cookies when the document accessing the cookie is on the local hard disk. Internet Explorer 4 works with cookies generated by local files.

The cookie file

Allowing some foreign CGI program to read from and write to your hard disk may give you pause, but Navigator doesn't just open up your drive's directory for the world to see (or corrupt). Instead, the cookie mechanism provides access to just one special text file located in a platform-specific spot on your drive.

In Windows versions of Navigator, the cookie file is named cookies.txt and is located in the Navigator directory; Mac users can find the MagicCookie file inside the Netscape folder, which is located within the System Folder:Preferences folder. The cookie file is a text file (but because the MagicCookie file's type is not TEXT, Mac users can open it only via applications capable of opening any kind of file). Internet Explorer uses a different filing system: Each cookie is saved as its own file inside a Cookies directory within system directories.

If curiosity drives you to open the cookie file, I recommend you do so only with a copy saved in another directory or folder. Any alteration to the existing file can mess up whatever valuable cookies are stored there for sites you regularly visit. Inside the file (after a few comment lines warning you not to manually alter the file) are lines of tab-delimited text. Each return-delimited line contains one cookie's information. The cookie file is just like a text listing of a database.



Note

As you experiment with Navigator cookies, you will be tempted to look into the cookie file after a script writes some data to the cookie. The cookie file will not contain the newly written data, because cookies are transferred to disk only when the user quits Navigator; conversely, the cookie file is read into Navigator's memory when it is launched. While you read, write, and delete cookies during a Navigator session, all activity is performed in memory (to speed up the process) to be saved later.

A cookie record

Among the "fields" of each cookie record are the following:

- ✦ Domain of the server that created the cookie
- ✦ Information on whether you need a secure HTTP connection to access the cookie
- ✦ Pathname of URL(s) capable of accessing the cookie
- ✦ Expiration date of the cookie
- ✦ Name of the cookie entry
- ✦ String data associated with the cookie entry

Notice that cookies are domain-specific. In other words, if one domain creates a cookie, another domain cannot access it through Navigator's cookie mechanism behind your back. That reason is why it's generally safe to store what I call *throwaway passwords* (the username/password pairs required to access some free registration-required sites) in cookies. Moreover, sites that store passwords in a cookie usually do so as encrypted strings, making it more difficult for someone to hijack the cookie file from your unattended PC and figure out what your personal password scheme might be.

Cookies also have expiration dates. Because the Navigator cookie file can hold no more than 300 cookies (as dictated by Navigator), the cookie file can get pretty full over the years. Therefore, if a cookie needs to persist past the current Navigator session, it also has an expiration date established by the cookie writer. Navigator cleans out any expired cookies to keep the file from exploding some years hence.

Not all cookies have to last beyond the current session, however. In fact, a scenario in which you use cookies temporarily while working your way through a Web site is quite typical. Many shopping sites employ one or more temporary cookie records to behave as the shopping cart for recording items you intend to purchase. These items are copied to the order form at check-out time. But once you submit the order form to the server, that client-side data has no particular value. As it turns out, if your script does not specify an expiration date, Navigator keeps the cookie fresh in memory without writing it to the cookie file. When you quit Navigator, that cookie data disappears as expected.

JavaScript access

Scripted access of cookies from JavaScript is limited to setting the cookie (with a number of optional parameters) and getting the cookie data (but with none of the parameters).

The JavaScript object model defines cookies as properties of documents, but this description is somewhat misleading. If you use the default path to set a cookie (that is, the current directory of the document whose script sets the cookie in the first place), then all documents in that same directory have read and write access to the cookie. A benefit of this arrangement is that if you have a scripted application that contains multiple documents, all documents in the same directory can share the cookie data. Netscape Navigator, however, imposes a limit of 20 named cookies entries for any domain; Internet Explorer 3 imposes an even more restrictive limit of one cookie (that is, one name-value pair) per domain. If your cookie requirements are extensive, then you need to fashion ways of concatenating cookie data (I do this in the Decision Helper application on the CD-ROM).

Saving cookies

To write cookie data to the cookie file, you use a simple JavaScript assignment operator with the `document.cookie` property. But the formatting of the data is crucial to achieving success. Here is the syntax for assigning a value to a cookie (optional items are in brackets):

```
document.cookie = "cookieName=cookieData
              [; expires=timeInGMTString]
              [; path=pathName]
              [; domain=domainName]
              [; secure]"
```

Examine each of the properties individually.

Name/Data

Each cookie must have a name and a string value (even if that value is an empty string). Such name-value pairs are fairly common in HTML, but they look odd in an

assignment statement. For example, if you want to save the string "Fred" to a cookie named "userName," the JavaScript statement is

```
document.cookie = "userName=Fred"
```

If Navigator sees no existing cookie in the current domain with this name, it automatically creates the cookie entry for you; if the named cookie already exists, Navigator replaces the old data with the new data. Retrieving `document.cookie` at this point yields the following string:

```
userName=Fred
```

You can omit all the other cookie-setting properties, in which case Navigator uses default values, as explained in a following section. For temporary cookies (those that don't have to persist beyond the current Navigator session), the name-value pair is usually all you need.

The entire name-value pair must be a single string with no semicolons, commas, or character spaces. To take care of spaces between words, preprocess the value with the JavaScript `escape()` function, which ASCII-encodes the spaces as `%20` (and then be sure to `unescape()` the value to restore the human-readable spaces when you retrieve the cookie later).

You cannot save a JavaScript array to a cookie. But with the help of the `Array.join()` method, you can convert an array to a string; use `String.split()` to re-create the array after reading the cookie at a later time. These two methods are available in Navigator from Version 3 onward and Internet Explorer 4 onward. If you add extra parameters, notice that all of them are included in the single string assigned to the `document.cookie` property. Also, each of the parameters must be separated by a semicolon and space.

Expires

Expiration dates, when supplied, must be passed as Greenwich mean time (GMT) strings (see Chapter 29 about time data). To calculate an expiration date based on today's date, use the JavaScript Date object as follows:

```
var exp = new Date()
var oneYearFromNow = exp.getTime() + (365 * 24 * 60 * 60 * 1000)
exp.setTime(oneYearFromNow)
```

Then convert the date to the accepted GMT string format:

```
document.cookie = "userName=Fred; expires=" + exp.toGMTString()
```

In the cookie file, the expiration date and time is stored as a numeric value (seconds) but, to set it, you need to supply the time in GMT format. You can delete a cookie before it expires by setting the named cookie's expiration date to a time and date earlier than the current time and date. The safest expiration parameter is

```
expires=Thu, 01-Jan-70 00:00:01 GMT
```

Omitting the expiration date signals Navigator that this cookie is temporary. Navigator never writes it to the cookie file and forgets it the next time you quit Navigator.

Path

For client-side cookies, the default path setting (the current directory) is usually the best choice. You can, of course, create a duplicate copy of a cookie with a separate path (and domain) so the same data is available to a document located in another area of your site (or the Web).

Domain

To help synchronize cookie data with a particular document (or group of documents), Navigator matches the domain of the current document with the domain values of cookie entries in the cookie file. Therefore, if you were to display a list of all cookie data contained in a `document.cookie` property, you would get back all the name-value cookie pairs from the cookie file whose domain parameter matches that of the current document.

Unless you expect the document to be replicated in another server within your domain, you can usually omit the domain parameter when saving a cookie. Navigator automatically supplies the domain of the current document to the cookie file entry. Be aware that a domain setting must have at least two periods, such as

```
.mcom.com
.hotwired.com
```

Or, you can write an entire URL to the domain, including the `http://` protocol (as Navigator does automatically when the domain is not specified).

SECURE

If you omit the `SECURE` parameter when saving a cookie, you imply that the cookie data is accessible to any document or CGI program from your site that meets the other domain- and path-matching properties. For client-side scripting of cookies, you should omit this parameter when saving a cookie.

Retrieving cookie data

Cookie data retrieved via JavaScript is contained in one string, including the whole name-data pair. Even though the cookie file stores other parameters for each cookie, you can only retrieve the name-data pairs via JavaScript. Moreover, in Navigator when two or more (up to a maximum of 20) cookies meet the current domain criteria, these cookies are also lumped into that string, delimited by a semicolon and space. For example, a `document.cookie` string might look like this:

```
userName=Fred; password=NikL2sPacU
```

In other words, you cannot treat named cookies as objects. Instead, you must parse the entire cookie string, extracting the data from the desired name-data pair.

When you know that you're dealing with only one cookie (and that no more will ever be added to the domain), you can customize the extraction based on known data, such as the cookie name. For example, with a cookie name that is seven characters long, you can extract the data with a statement like this:

```
var data =
unescape(document.cookie.substring(7,document.cookie.length))
```

The first parameter of the `substring()` method includes the equals sign to separate the name from the data.

A better approach is to create a general purpose function that can work with single- or multiple-entry cookies. Here is one I use in some of my pages:

```
function getCookieData(label) {
        var labelLen = label.length
        var cLen = document.cookie.length
        var i = 0
        var cEnd
        while (i < cLen) {
           var j = i + labelLen
           if (document.cookie.substring(i,j) == label) {
              cEnd = document.cookie.indexOf(";",j)
              if (cEnd == -1) {
                 cEnd = document.cookie.length
              }
              return unescape(document.cookie.substring(j,cEnd))
           }
           i++
        }
        return ""
}
```

Calls to this function pass the name of the desired cookie as a parameter. The function parses the entire cookie string, chipping away any mismatched entries (through the semicolons) until it finds the cookie name.

If all of this cookie code still makes your head hurt, you can turn to a set of functions devised by experienced JavaScripter and Web site designer Bill Dortch of hIdaho Design. His cookie functions provide generic access to cookies that you can use in all of your cookie-related pages. Listing 16-3 shows Bill's cookie functions, which include a variety of safety nets for date calculation bugs that appeared in some versions of Netscape Navigator 2. Don't be put off by the length of the listing: Most of the lines are comments. Updates to Bill's functions can be found at `http://www.hidaho.com/cookies/cookie.txt`.

Listing 16-3: Bill Dortch's cookie Functions

```
<html>
<head>
<title>Cookie Functions</title>
</head>
<body>
<script language="javascript">
<!-- begin script
//
//  Cookie Functions -- "Night of the Living Cookie" Version (25-Jul-96)
//
//  Written by:  Bill Dortch, hIdaho Design <bdortch@hidaho.com>
//  The following functions are released to the public domain.
//
```

(continued)

Listing 16-3 *(continued)*

```
//  This version takes a more aggressive approach to deleting
//  cookies.  Previous versions set the expiration date to one
//  millisecond prior to the current time; however, this method
//  did not work in Netscape 2.02 (though it does in earlier and
//  later versions), resulting in "zombie" cookies that would not
//  die.  DeleteCookie now sets the expiration date to the earliest
//  usable date (one second into 1970), and sets the cookie's value
//  to null for good measure.
//
//  Also, this version adds optional path and domain parameters to
//  the DeleteCookie function.  If you specify a path and/or domain
//  when creating (setting) a cookie**, you must specify the same
//  path/domain when deleting it, or deletion will not occur.
//
//  The FixCookieDate function must now be called explicitly to
//  correct for the 2.x Mac date bug.  This function should be
//  called *once* after a Date object is created and before it
//  is passed (as an expiration date) to SetCookie.  Because the
//  Mac date bug affects all dates, not just those passed to
//  SetCookie, you might want to make it a habit to call
//  FixCookieDate any time you create a new Date object:
//
//    var theDate = new Date();
//    FixCookieDate (theDate);
//
//  Calling FixCookieDate has no effect on platforms other than
//  the Mac, so there is no need to determine the user's platform
//  prior to calling it.
//
//  This version also incorporates several minor coding improvements.
//
//  **Note that it is possible to set multiple cookies with the same
//  name but different (nested) paths.  For example:
//
//    SetCookie ("color","red",null,"/outer");
//    SetCookie ("color","blue",null,"/outer/inner");
//
//  However, GetCookie cannot distinguish between these and will return
//  the first cookie that matches a given name.  It is therefore
//  recommended that you *not* use the same name for cookies with
//  different paths.  (Bear in mind that there is *always* a path
//  associated with a cookie; if you don't explicitly specify one,
//  the path of the setting document is used.)
//
//  Revision History:
//
//    "Toss Your Cookies" Version (22-Mar-96)
//      - Added FixCookieDate() function to correct for Mac date bug
//
//    "Second Helping" Version (21-Jan-96)
```

```
//       - Added path, domain and secure parameters to SetCookie
//       - Replaced home-rolled encode/decode functions with Netscape's
//         new (then) escape and unescape functions
//
//     "Free Cookies" Version (December 95)
//
//
//  For information on the significance of cookie parameters,
//  and on cookies in general, please refer to the official cookie
//  spec, at:
//
//       http://www.netscape.com/newsref/std/cookie_spec.html
//
//******************************************************************
//
// "Internal" function to return the decoded value of a cookie
//
function getCookieVal (offset) {
  var endstr = document.cookie.indexOf (";", offset);
  if (endstr == -1)
    endstr = document.cookie.length;
  return unescape(document.cookie.substring(offset, endstr));
}
//
//  Function to correct for 2.x Mac date bug.  Call this function to
//  fix a date object prior to passing it to SetCookie.
//  IMPORTANT:  This function should only be called *once* for
//  any given date object!  See example at the end of this document.
//
function FixCookieDate (date) {
  var base = new Date(0);
  var skew = base.getTime(); // dawn of (Unix) time - should be 0
  if (skew > 0)  // Except on the Mac - ahead of its time
    date.setTime (date.getTime() - skew);
}
//
//  Function to return the value of the cookie specified by "name".
//     name - String object containing the cookie name.
//     returns - String object containing the cookie value, or null if
//       the cookie does not exist.
//
function GetCookie (name) {
  var arg = name + "=";
  var alen = arg.length;
  var clen = document.cookie.length;
  var i = 0;
  while (i < clen) {
    var j = i + alen;
    if (document.cookie.substring(i, j) == arg)
      return getCookieVal (j);
       i = document.cookie.indexOf(" ", i) + 1;
    if (i == 0) break;
  }
```

(continued)

Listing 16-3 *(continued)*

```
  return null;
}
//
//  Function to create or update a cookie.
//    name - String object containing the cookie name.
//    value - String object containing the cookie value.  May contain
//      any valid string characters.
//    [expires] - Date object containing the expiration data of the cookie.  If
//      omitted or null, expires the cookie at the end of the current session.
//    [path] - String object indicating the path for which the cookie is valid.
//      If omitted or null, uses the path of the calling document.
//    [domain] - String object indicating the domain for which the cookie is
//      valid. If omitted or null, uses the domain of the calling document.
//    [secure] - Boolean (true/false) value indicating whether cookie transmission
//      requires a secure channel (HTTPS).
//
//  The first two parameters are required.  The others, if supplied, must
//  be passed in the order listed above.  To omit an unused optional field,
//  use null as a place holder.  For example, to call SetCookie using name,
//  value and path, you would code:
//
//      SetCookie ("myCookieName", "myCookieValue", null, "/");
//
//  Note that trailing omitted parameters do not require a placeholder.
//
//  To set a secure cookie for path "/myPath", that expires after the
//  current session, you might code:
//
//      SetCookie (myCookieVar, cookieValueVar, null, "/myPath", null, true);
//
function SetCookie (name,value,expires,path,domain,secure) {
  document.cookie = name + "=" + escape (value) +
    ((expires) ? "; expires=" + expires.toGMTString() : "") +
    ((path) ? "; path=" + path : "") +
    ((domain) ? "; domain=" + domain : "") +
    ((secure) ? "; secure" : "");
}

//  Function to delete a cookie. (Sets expiration date to start of epoch)
//    name -   String object containing the cookie name
//    path -   String object containing the path of the cookie to delete.  This MUST
//             be the same as the path used to create the cookie, or null/omitted if
//               no path was specified when creating the cookie.
//    domain - String object containing the domain of the cookie to delete.  This MUST
//             be the same as the domain used to create the cookie, or null/omitted if
//               no domain was specified when creating the cookie.
//
function DeleteCookie (name,path,domain) {
  if (GetCookie(name)) {
    document.cookie = name + "=" +
```

```
        ((path) ? "; path=" + path : "") +
        ((domain) ? "; domain=" + domain : "") +
        "; expires=Thu, 01-Jan-70 00:00:01 GMT";
    }
  }

  //
  //  Examples
  //
  var expdate = new Date ();
  FixCookieDate (expdate); // Correct for Mac date bug - call only once
  for given Date object!
  expdate.setTime (expdate.getTime() + (24 * 60 * 60 * 1000)); // 24 hrs
  from now
  SetCookie ("ccpath", "http://www.hidaho.com/colorcenter/", expdate);
  SetCookie ("ccname", "hIdaho Design ColorCenter", expdate);
  SetCookie ("tempvar", "This is a temporary cookie.");
  SetCookie ("ubiquitous", "This cookie will work anywhere in this
  domain",null,"/");
  SetCookie ("paranoid", "This cookie requires secure
  communications",expdate,"/",null,true);
  SetCookie ("goner", "This cookie must die!");
  document.write (document.cookie + "<br>");
  DeleteCookie ("goner");
  document.write (document.cookie + "<br>");
  document.write ("ccpath = " + GetCookie("ccpath") + "<br>");
  document.write ("ccname = " + GetCookie("ccname") + "<br>");
  document.write ("tempvar = " + GetCookie("tempvar") + "<br>");
  // end script -->
  </script>
  </body>
  </html>
```

Extra batches

You may design a site that needs more than 20 Netscape cookies for a given domain. For example, in a shopping site, you never know how many items a customer might load into the shopping cart cookie.

Because each named cookie stores plain text, you can create your own text-based data structures to accommodate multiple pieces of information per cookie. (Despite Netscape's information that each cookie can contain up to 4,000 characters, the value of one name-value pair cannot exceed 2,000 characters.) The trick is determining a delimiter character that won't be used by any of the data in the cookie. In Decision Helper (on the CD-ROM), for example, I use a period to separate multiple integers stored in a cookie; Netscape uses colons to separate settings in the custom page cookie data.

With the delimiter character established, you must then write functions that concatenate these "subcookies" into single cookie strings and extract them on the other side. It's a bit more work, but well worth the effort to have the power of persistent data on the client.

Example

Experiment with the last group of statements in Listing 16-3 to create, retrieve, and delete cookies.

Related Items: String object methods (Chapter 27).

domain

Value: String **Gettable:** Yes **Settable:** Yes

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

Security restrictions can get in the way of sites that have more than one server at their domain. Because some objects, especially the location object, prevent access to properties of other servers displayed in other frames, legitimate access to those properties are blocked. For example, it's not uncommon for popular sites to have their usual public access site on a server named something like `www.popular.com`. If a page on that server includes a front end to a site search engine located at `search.popular.com`, visitors who use browsers with these security restrictions will be denied access.

To guard against that eventuality, a script in documents from both servers can instruct the browser to think both servers are the same. In the example above, you would set the `document.domain` property in both documents to `popular.com`. Without specifically setting the property, the default value includes the server name as well, thus causing a mismatch between host names.

Before you start thinking you can spoof your way into other servers, be aware that you can set the `document.domain` property only to servers with the same domain (following the "two-dot" rule) as the document doing the setting. Therefore, documents originating only from `xxx.popular.com` can set their `document.domain` properties to `popular.com` server.

Related Items: `window.open()` method; `window.location` object; security (Chapter 40).

embeds

Value: Array of plug-ins **Gettable:** Yes **Settable:** No

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

Whenever you want to load data that requires a plug-in application to play or display, you use the `<EMBED>` tag. The `document.embeds` property is merely one way to determine the number of such tags defined in the document:

```
var count = document.embeds.length
```

For controlling those plug-ins in Navigator, you can use the LiveConnect technology, described in Chapter 38.

forms

Value: Array **Gettable:** Yes **Settable:** No

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

As I show in Chapter 21, which is dedicated to the form object, an HTML form (anything defined inside a `<FORM>...</FORM>` tag pair) is a JavaScript object unto itself. You can create a valid reference to a form according to its name (assigned via a form's `NAME=` attribute). For example, if a document contains the following form definition

```
<FORM NAME="phoneData">
       input item definitions
</FORM>
```

your scripts can refer to the form object by name:

```
document.phoneData
```

However, a document object also tracks its forms in another way: as a numbered list of forms. This type of list in JavaScript is called an *array,* which means a table consisting of just one column of data. Each row of the table holds a representation of the corresponding form in the document. In the first row of a `document.forms` array, for instance, is the form that loaded first (it was first from the top of the HTML code). If your document defines one form, the `forms` property is an array one entry in length; with three separate forms in the document, the array is three entries long.

To help JavaScript determine which row of the array your script wants to access, you append a pair of brackets to the `forms` property name and insert the row number between the brackets (this is standard JavaScript array notation). This number is formally known as the *index*. JavaScript arrays start their row numbering with 0, so the first entry in the array is referenced as

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

At that point, you're referencing the equivalent of the first form object. Any of its properties or methods are available by appending the desired property or method name to the reference. For example, to retrieve the value of an input text field named "homePhone" from the second form of a document, the reference you use is

```
document.forms[1].homePhone.value
```

One advantage to using the `document.forms` property for addressing a form object or element instead of the actual form name is that you may be able to

generate a library of generalizable scripts that know how to cycle through all available forms in a document and hunt for a form that has some special element and property. The following script fragment (part of a *repeat loop* described more fully in Chapter 31) uses a loop-counting variable (i) to help the script check all forms in a document:

```
for (var i = 0; i < document.forms.length; i++) {
        if (document.forms[i]. ... ) {
           statements
        }
}
```

Each time through the repeat loop, JavaScript substitutes the next higher value for i in the document.forms[i] object reference. Not only does the array counting simplify the task of checking all forms in a document, but this fragment is totally independent of whatever names you assign to forms.

As you saw in the preceding script fragment, there is one more aspect of the document.forms property that you should be aware of. All JavaScript arrays that represent built-in objects have a length property that returns the number of entries in the array. JavaScript counts the length of arrays starting with 1. Therefore, if the document.forms.length property returns a value of 2, the form references for this document would be document.forms[0] and document.forms[1]. If you haven't programmed these kinds of arrays before, the different numbering systems (indexes starting with 0, length counts starting with 1) take some getting used to.

If you use a lot of care in assigning names to objects, you will likely prefer the document.formName style of referencing forms. In this book, you see both indexed array and form name style references. The advantage of using name references is that even if you redesign the page and change the order of forms in the document, references to the named forms will still be valid, whereas the index numbers of the forms will have changed. See also the discussion in Chapter 21 of the form object and how to pass a form's data to a function.

Example

The document in Listing 16-4 is set up to display an alert dialog box that replicates navigation to a particular music site, based on the checked status of the "bluish" checkbox. The user input here is divided into two forms: one form with the checkbox and the other form with the button that does the navigation. A block of copy fills the space in between. Clicking the bottom button (in the second form) triggers the function that fetches the checked property of the "bluish" checkbox, using the document.forms[i] array as part of the address.

Listing 16-4: **Using the document.forms Property**

```
<HTML>
<HEAD>
<TITLE>document.forms example</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function goMusic() {
        if (document.forms[0].bluish.checked) {
           alert("Now going to the Blues music area...")
```

```
        } else {
            alert("Now going to Rock music area...")
        }
}
</SCRIPT>
</HEAD>

<BODY>
<FORM NAME="theBlues">
<INPUT TYPE="checkbox" NAME="bluish">Check here if you've got the
blues.
</FORM>
<HR>
M<BR>
o<BR>
r<BR>
e<BR>
<BR>
C<BR>
o<BR>
p<BR>
y<BR>
<HR>
<FORM NAME="visit">
<INPUT TYPE="button" VALUE="Visit music site" onClick="goMusic()">
</FORM>
</BODY>
</HTML>
```

Related Items: form object.

images

Value: Array of image objects **Gettable:** Yes **Settable:** No

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

With images treated as first-class objects beginning with Navigator 3 and Internet Explorer 4, it's only natural for a document to maintain an array of all the image tags defined on the page (just as it does for links and anchors). The prime importance of having images as objects is that you can modify their content (the source file associated with the rectangular space of the image) on the fly. You can find details about the image object in Chapter 18.

The lack of an image object in Windows versions of Internet Explorer 3 disappointed many authors who wanted to swap images on mouse rollovers. Little known, however, is that the Macintosh version of Internet Explorer (Version 3.01a) has an image object in its object model, working exactly like Navigator 3's image object. The image object is present on all platforms in Internet Explorer 4.

Use image array references to pinpoint a specific image for retrieval of any image property or for assigning a new image file to its `src` property. Image arrays begin their index counts with 0: The first image in a document has the reference `document.images[0]`. And, as with any array object, you can find out how many images the array contains by checking the `length` property. For example

```
imageCount = document.images.length
```

Images can also have names, so if you prefer, you can refer to the image object by its name, as in

```
imageLoaded = document.imageName.complete
```

Example

The `document.images` property is defined automatically as the browser builds the object model for a document that contains image objects. See the discussion about the `image` object in Chapter 18 for reference examples.

Related Items: image object.

lastModified

Value: DateString **Gettable:** Yes **Settable:** No

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

Every disk file maintains a modified timestamp, and most servers expose this information to a browser accessing a file. This information is available by reading the `document.lastModified` property (be sure to observe the uppercase M in the property name). If your server supplies this information to the client, you can use the value of this property to present this information for readers of your Web page. The script automatically updates the value for you, rather than requiring you to hand-code the HTML line every time you modify the home page.

The returned value is not a date object (Chapter 29) but rather a straight string consisting of time and date, as recorded by the document's file system. You can, however, convert the date string to a JavaScript date object and use the date object's methods to extract selected elements for recompilation into readable form. Listing 16-5 shows an example.

Even local file systems don't necessarily provide the correct data for every browser to interpret. For example, in Navigator of all generations for the Macintosh, dates from files stored on local disks come back as something from the 1920s (although Internet Explorer manages to reflect the correct date). But put that same file on a UNIX or NT Web server, and the date appears correctly when accessed via the Net.

Example

Experiment with the `document.lastModified` property with Listing 16-5. But also be prepared for inaccurate readings if the file is located on some servers or local hard disks.

Listing 16-5: document.lastModified Property in Another Format

```
<HTML>
<HEAD>
<TITLE>Time Stamper</TITLE>
</HEAD>
<BODY>
<CENTER> <H1>GiantCo Home Page</H1></CENTER>
<SCRIPT LANGUAGE="JavaScript">
update = new Date(document.lastModified)
theMonth = update.getMonth() + 1
theDate = update.getDate()
theYear = update.getYear()
document.writeln("<I>Last updated:" + theMonth + "/" + theDate + "/" +
theYear + "</I>")
</SCRIPT>
<HR>
</BODY>
</HTML>
```

As noted at great length in Chapter 29's discussion about the date object, you should be aware that date formats vary greatly from country to country. Some of these formats use a different order for date elements. When you hard-code a date format, it may take a form that is unfamiliar to other users of your page.

Related Items: date object (Chapter 29).

layers

Value: String **Gettable:** Yes **Settable:** No

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

The layer object (Chapter 19) is new as of Netscape Navigator 4 and is not part of Internet Explorer 4's object model. Therefore, only the Netscape browser contains a property that reflects the array of layer objects within a document. This is the same kind of array used to refer to other document objects, such as images and applets.

A Netscape layer is a container for content that can be precisely positioned on the page. Layers can be defined with the Netscape-specific `<LAYER>` tag or with W3C standard style-sheet positioning syntax, as explained in Chapter 19. Each layer contains a document object — the true holder of the content displayed in that layer. Layers can be nested within each other, but a reference to `document.layers` reveals only the first level of layers defined in the document. Consider the following HTML skeleton:

```
<HTML>
<BODY>
<LAYER NAME="Europe">
        <LAYER NAME="Germany"></LAYER>
        <LAYER NAME="Netherlands"></LAYER>
</LAYER>
</BODY>
</HTML>
```

From the point of view of the primary document, there is one layer (`Europe`). Therefore, the length of the `document.layers` array is 1. But the Europe layer has a document, in which two more layers are nested. A reference to the array of those nested layers would be

```
document.layers[1].document.layers
```

or

```
document.Europe.document.layers
```

The length of this nested array is two: The `Germany` and `Netherlands` layers. No property exists that reveals the entire set of nested arrays in a document, but you can create a `for` loop to crawl through all nested layers (shown in Listing 16-6).

Example

Listing 16-6 demonstrates how to use the `document.layers` property to crawl through the entire set of nested layers in a document. Using recursion (discussed in Chapter 34), the script builds an indented list of layers in the same hierarchy as the objects themselves and displays the results in an alert dialog. After you load this document (the script is triggered by the `onLoad=` event handler), compare the alert dialog contents against the structure of `<LAYER>` tags in the document.

Listing 16-6: A Layer Crawler

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript1.2">
var output = ""
function crawlLayers(layerArray, indent) {
        for (var i = 0; i < layerArray.length; i++) {
            output += indent + layerArray[i].name + "\n"
            if (layerArray[i].document.layers.length) {
                var newLayerArray = layerArray[i].document.layers
                crawlLayers(newLayerArray, indent + "   ")
            }
        }
        return output
}
function revealLayers() {
        alert(crawlLayers(document.layers, ""))
}
```

```
</SCRIPT>
</HEAD>
<BODY onLoad="revealLayers()">
<LAYER NAME="Europe">
       <LAYER NAME="Germany"></LAYER>
       <LAYER NAME="Netherlands">
          <LAYER NAME="Amsterdam"></LAYER>
          <LAYER NAME="Rotterdam"></LAYER>
       </LAYER>
       <LAYER NAME="France"></LAYER>
</LAYER>
<LAYER NAME="Africa">
       <LAYER NAME="South Africa"></LAYER>
       <LAYER NAME="Ivory Coast"></LAYER>
</LAYER>
</BODY>
</HTML>
```

Related Items: layer object.

links

Value: Array of link objects **Gettable:** Yes **Settable:** No

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

The `document.links` property is similar to the `document.anchors` property, except that the objects maintained by the array are link objects — items created with `<A HREF="">` tags. Use the array references to pinpoint a specific link for retrieving any link property, such as the target window specified in the link's HTML definition.

Link arrays begin their index counts with 0: The first link in a document has the reference `document.links[0]`. And, as with any array object, you can find out how many entries the array has by checking the length property. For example

```
linkCount = document.links.length
```

Entries in the `document.links` property are full-fledged `location` objects.

Example

The `document.links` property is defined automatically as the browser builds the object model for a document that contains link objects. You will rarely access this property, except to determine the number of link objects in the document.

Related Items: link object; `document.anchors` property.

location
URL

Value: String **Gettable:** Yes **Settable:** No (Navigator)

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

The fact that JavaScript frequently reuses the same terms in different contexts may be confusing to the language's newcomers. Such is the case with the `document.location` property. You may wonder how it differs from the `location` object (Chapter 15). In practice, many scripts also get the two confused when references don't include the window object. As a result, a new property name, `document.URL`, was introduced in Navigator 3 and Internet Explorer 4 to take the place of `document.location`. You can still use `document.location`, but the term may eventually disappear from the JavaScript vocabulary (or at least from Netscape's object model). To help you get into the future mindset, the rest of this discussion refers to this property as `document.URL`.

The remaining question is how the `window.location` object and `document.URL` property differ. The answer lies in their respective data types.

A location object, you may recall from Chapter 15, consists of a number of properties about the document currently loaded in a window or frame. Assigning a new URL to the object tells the browser to load that URL into the frame. The `document.URL` property, on the other hand, is simply a string (read-only in Navigator) that reveals the URL of the current document. The value may be important to your script, but the property does not have the "object power" of the `window.location` object. You cannot change (assign another value to) this property value because a document has only one URL: its location on the Net (or your hard disk) where the file exists, and what protocol is required to get it.

This may seem like a fine distinction, and it is. The reference you use (`window.location` object or `document.URL` property) depends on what you are trying to accomplish specifically with the script. If the script is changing the content of a window by loading a new URL, you have no choice but to assign a value to the `window.location` object. Similarly, if the script is concerned with the component parts of a URL, the properties of the location object provide the simplest avenue to that information. To retrieve the URL of a document in string form (whether it is in the current window or in another frame), you can use either the `document.URL` property or the `window.location.href` property.

Example

HTML documents in Listing 16-7 through 16-9 create a test lab that enables you to experiment with viewing the `document.URL` property for different windows and frames in a multiframe environment. Results are displayed in a table, with an additional listing of the `document.title` property to help you identify documents being referred to. The same security restrictions that apply to retrieving

`window.location` **object properties also apply to retrieving the** `document.URL` **property from another window or frame.**

Listing 16-7: Frameset for document.URL Property Reader

```
<HTML>
<HEAD>
<TITLE>document.URL Reader</TITLE>
</HEAD>
<FRAMESET ROWS="60%,40%">
       <FRAME NAME="Frame1" SRC="lst16-09.htm">
       <FRAME NAME="Frame2" SRC="lst16-08.htm">
</FRAMESET>
</HTML>
```

Listing 16-8: document.URL Property Reader

```
<HTML>
<HEAD>
<TITLE>URL Property Reader</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
function fillTopFrame() {
       newURL=prompt("Enter the URL of a document to show in the top
frame:","")
       if (newURL != null && newURL != "") {
       top.frames[0].location = newURL
       }
}

function showLoc(form,item) {
       var windName = item.value
       var theRef = windName + ".document"
       form.dLoc.value = unescape(eval(theRef + ".URL"))
       form.dTitle.value = unescape(eval(theRef + ".title"))
}
</SCRIPT>
</HEAD>

<BODY>
Click the "Open URL" button to enter the location of an HTML document
to display in the upper frame of this window.
<FORM>
<INPUT TYPE="button" NAME="opener" VALUE="Open URL..."
onClick="fillTopFrame()">
</FORM>
<HR>
<FORM>
Select a window or frame to view each document property values.<P>
```

(continued)

Listing 16-8 *(continued)*

```
<INPUT TYPE="radio" NAME="whichFrame" VALUE="parent"
onClick="showLoc(this.form,this)">Parent window
<INPUT TYPE="radio" NAME="whichFrame" VALUE="top.frames[0]"
onClick="showLoc(this.form,this)">Upper frame
<INPUT TYPE="radio" NAME="whichFrame" VALUE="top.frames[1]"
onClick="showLoc(this.form,this)">This frame<P>
<TABLE BORDER=2>
<TR><TD ALIGN=RIGHT>document.URL:</TD>
<TD><TEXTAREA NAME="dLoc" ROWS=3 COLS=30
WRAP="soft"></TEXTAREA></TD></TR>

<TR><TD ALIGN=RIGHT>document.title:</TD>
<TD><TEXTAREA NAME="dTitle" ROWS=3 COLS=30
WRAP="soft"></TEXTAREA></TD></TR>
</TABLE>
</FORM>
</BODY>
</HTML>
```

Listing 16-9: Placeholder for Listing 16-7

```
<HTML>
<HEAD>
<TITLE>Opening Placeholder</TITLE>
</HEAD>
<BODY>
Initial place holder. Experiment with other URLs for this frame (see
below).
</BODY>
</HTML>
```

Related Items: location object; `location.href` property.

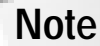
referrer

Value: String **Gettable:** Yes **Settable:** No

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

When a link from one document leads to another, the second document can, under JavaScript control, reveal the URL of the document containing the link. The `document.referrer` property contains a string of that URL. This feature can be a

useful tool for customizing the content of pages based on the previous location the user was visiting within your site. A referrer contains a value only when the user reaches the current page via a link. Any other method of navigation (such as through the history or by manually entering a URL) sets this property to an empty string.



The `document.referrer` property is broken in Windows versions of Internet Explorer 3 and Internet Explorer 4. In the Windows version, the current document's URL is given as the referrer; the proper value is returned in the Macintosh versions.

Example

This demonstration requires two documents. The first document, in Listing 16-10, simply contains one line of text as a link to the second document. In the second document (Listing 16-11), a script verifies the document from which the user came via a link. If the script knows about that link, it displays a message relevant to the experience the user had at the first document. Also try opening Listing 16-11 from the Open File command in the File menu to see how the script won't recognize the referrer.

Listing 16-10: **A Source Document**

```
<HTML>
<HEAD>
<TITLE>document.referrer Property 1</TITLE>
</HEAD>

<BODY>
<H1><A HREF="lst16-11.htm">Visit my sister document</A>
</BODY>
</HTML>
```

Listing 16-11: **Checking document.referrer**

```
<HTML>
<HEAD>
<TITLE>document.referrer Property 2</TITLE>
</HEAD>

<BODY><H1>
<SCRIPT LANGUAGE="JavaScript">
if(document.referrer.length > 0 && document.referrer.indexOf("16-
10.htm") != -1){
       document.write("How is my brother document?")
       } else {
       document.write("Hello, and thank you for stopping by.")
       }
</SCRIPT>
</H1></BODY>
</HTML>
```

Related Items: link object.

`title`

Value: String **Gettable:** Yes **Settable:** No

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

A document's title is the text that appears between the `<TITLE>...</TITLE>` tag pair in an HTML document's Head portion. The title usually appears in the title bar of the browser window in a single-frame presentation. Only the title of the topmost framesetting document appears as the title of a multiframe window. Even so, the `title` property for an individual document appearing in a frame is available via scripting. For example, if two frames are available (`UpperFrame` and `LowerFrame`), a script in the document occupying the `LowerFrame` frame could reference the `title` property of the other frame's document like this:

```
parent.UpperFrame.document.title
```

This property cannot be set by a script except when constructing an entire HTML document via script, including the `<TITLE>` tags.

Note

UNIX versions of Navigator 2 fail to return the `document.title` property value. Also, in Navigator 4 for the Macintosh, if a script creates the content of another frame, the `document.title` property for that dynamically written frame returns the file name of the script that wrote the HTML, even when it writes a valid `<TITLE>` tag set.

Example

See Listings 16-7 through 16-9 for examples of retrieving the `document.title` property from a multiframe window.

Related Items: history object.

Methods

`captureEvents(eventTypeList)`

Returns: Nothing.

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

In Navigator 4, an event filters down from the window object, through the document object, and eventually reaches its target. For example, if you click a button, the click event first reaches the window object; then it goes to the

document object; if the button is defined within a layer, the event also filters through that layer; eventually (in a split second) it reaches the button, where an `onClick=` event handler is ready to act on that click.

The Netscape mechanism allows window, document, and layer objects to intercept events and process them prior to reaching their intended targets (or preventing them from reaching their destinations entirely). But for an outer container to grab an event, your script must instruct it to capture the type of event your application is interested in preprocessing. If you want the document object to intercept all events of a particular type, use the `document.captureEvents()` method to turn that facility on.

The `document.captureEvents()` method takes one or more event types as parameters. An event type is a constant value built inside Navigator 4's event object. One event type exists for every kind of event handler you see in all of Navigator 4's document objects. The syntax is the event object name (`Event`) and the event name in all uppercase letters. For example, if you want the document to intercept all click events, the statement is

```
document.captureEvents(Event.CLICK)
```

For multiple events, add them as parameters, separated by the pipe (|) character:

```
document.captureEvents(Event.MOUSEDOWN | Event.KEYPRESS)
```

Once an event type is captured by the document object, it must have a function ready to deal with the event. For example, perhaps the function looks through all `Event.MOUSEDOWN` events and looks to see if the right mouse button was the one that triggered the event and what form element (if any) is the intended target. The goal is to perhaps display a pop-up menu (as a separate layer) for a right-click. If the click comes from the left mouse button, then the event is routed to its intended target.

To associate a function with a particular event type captured by a document object, assign a function to the event. For example, to assign a custom `doClickEvent()` function to click events captured by the window object, use the following statement:

```
document.onclick=doClickEvent
```

Notice that the function name is assigned only as a reference name, not like an event handler within a tag. The function, itself, is like any function, but it has the added benefit of automatically receiving the event object as a parameter. To turn off event capture for one or more event types, use the `document.releaseEvent()` method. See Chapter 33 for details of working with events in this manner.

Example

See the example for the `window.captureEvents()` method in Chapter 14 (Listing 14-22) to see how to capture events on their way to other objects. You can substitute the `document` reference for the `window` reference in that example to see how the document version of the method works just like the window version. If you understand the mechanism for windows, you understand it for documents. The same is true for the other event methods.

Related Items: `document.handleEvent()` method; `document.releaseEvents()` method; `document.routeEvent()` method; parallel window object event methods.

clear()

Returns: Nothing.

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

Clearing a document and closing a document are two quite different actions. As described in the following `document.close()` section, closing deals with the layout stream previously opened to a document. Frequently, the stream must be closed before all data specified in the HTML of the document appears correctly.

Clearing a document, on the other hand, removes from the browser whatever HTML is written to the document — as well as the object model for that document. You do not have to clear a document prior to opening or writing to another one (JavaScript clears the old one for you). In fact, the `document.clear()` method does not work correctly, even in Navigator 3, and can cause any number of errors or crash problems in early browsers. To get the same result as the one that you expect to get from `document.clear()`, I recommend loading a blank HTML document (that is, one with the simplest tags and no visible content).

Related Items: `document.close()` method; `document.write()` method; `document.writeln()` method.

close()

Returns: Nothing.

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

Whenever a layout stream is opened to a window via the `document.open()` method or either of the document writing methods (which also open the layout stream), you must close the stream once the document has been written. This causes the `Layout:Complete` and `Document:Done` messages to appear in the status line (although you may experience some bugs in the status message on some platforms). The closing step is very important to prepare the window for the next potential round of replenishment with new script-assembled HTML. If you don't close the window, subsequent writing is appended to the bottom of it.

Some or all of the data specified for the window won't display properly until you invoke the `document.close()` method, especially when images are being drawn as part of the document stream. A common symptom is the momentary

appearance and then disappearance of the document parts. If you see such behavior, look for a missing `document.close()` method after the last `document.write()` method.

Example

Before you experiment with this method, be sure you understand the `document.write()` method described later in this chapter. After that, make a separate set of the three documents for that method's example (Listing 16-13 through 16-15 in a different directory or folder). In the `takePulse()` function listing, comment out both the `document.open()` and `document.close()` statements, as shown here:

```
msg += "<P>Make it a great day!</BODY></HTML>"
//parent.frames[1].document.open()
parent.frames[1].document.write(msg)
//parent.frames[1].document.close()
```

Now try the pages on your browser. You will see that each click of the upper button appends text to the bottom frame, without first removing the previous text. The reason is that the previous layout stream was never closed. The document thinks that you're still writing to it. Also, without properly closing the stream, the last line of text may not appear in the most recently written batch.

Related Items: `document.open()` method; `document.clear()` method; `document.write()` method; `document.writeln()` method.

getSelection()

Returns: String.

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

It's likely that many Web browser users aren't aware that they can select and copy body text in a document for pasting into other application documents. Even so, Navigator 4 offers a scripted way of capturing the text selected by a user in a page. The `document.getSelection()` method returns the string of text selected by the user. If nothing is selected, an empty string is the result. Returned values consist only of the visible text on the page, and not the underlying HTML or style of the text.

Example

The document in Listing 16-12 combines document event capture and the `getSelection()` method to display in a textarea object the content of any selection you make from the text on the page.

Listing 16-12: Text Selection

```
<HTML>
<HEAD>
<TITLE>URL Property Reader</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function showSelection() {
        document.forms[0].selectedText.value = document.getSelection()
}
document.captureEvents(Event.MOUSEUP)
document.onmouseup = showSelection
</SCRIPT>
</HEAD>

<BODY>
<B>Select some text and see how JavaScript can capture the
selection:</B>
<HR>
<H2>ARTICLE I</H2>
<P>
Congress shall make no law respecting an establishment of religion, or
prohibiting the free exercise thereof; or abridging the freedom of
speech, or of the press; or the right of the people peaceably to
assemble, and to petition the government for a redress of grievances.
</P>
</HR>
<FORM>
<TEXTAREA NAME="selectedText" ROWS=3 COLS=40 WRAP="virtual"></TEXTAREA>
</FORM>
</BODY>
</HTML>
```

Related Items: None.

handleEvent(event)

Returns: Nothing.

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

When you explicitly capture events in the window, document, or layer object (by invoking the `captureEvents()` method for that object), you can control where the events go after their initial capture. To let an event continue to its original target (for example, a button that was clicked by a user), you use the `routeEvent()` method. But if you want to redirect an event (or class of events) to

a particular event handler elsewhere in the document, use the `handleEvent()` method.

See the discussion of the `handleEvent()` method for the window object in Chapter 14. The behavior of the `handleEvent()` method for all objects is the same.

Related Items: `document.captureEvents()` method; `document.releaseEvents()` method; `document.routeEvent()` method; event object.

open(["mimeType"] [, replace])

Returns: Nothing.

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

Opening a document is different from opening a window. In the case of a window, you're creating a new object, both on the screen and in the browser's memory. Opening a document, on the other hand, tells the browser to get ready to accept some data for display in the window named or implied in the reference to the `document.open()` method. (For example, `parent.frames[1].document.open()` may refer to a different frame in a frameset, whereas `document.open()` implies the current window or frame.) Therefore, the method name may mislead newcomers because the `document.open()` method has nothing to do with loading documents from the Web server or hard disk. Rather, this method is a prelude to sending data to a window via the `document.write()` or `document.writeln()` methods. In a sense, the `document.open()` method merely opens a door; the other methods send the data, and the `document.close()` method closes that door once the page's data has been sent in full.

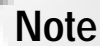
The `document.open()` method is optional because a `document.write()` method that attempts to write to a closed document automatically clears the old document and opens a new one. Whether or not you use the `document.open()` method, be sure to use the `document.close()` method after all the writing has taken place.

An optional parameter to the `document.open()` method lets you specify the nature of the data being sent to the window. A MIME (Multipurpose Internet Mail Extension) type is a specification for transferring and representing multimedia data on the Internet (originally for mail transmission, but now applicable to all Internet data exchanges). You've seen MIME depictions in the list of helper applications in your browser's preferences settings. A MIME type is represented by a pair of data type names separated by a slash (such as `text/html` and `image/gif`). When you specify a MIME type as a parameter to the `document.open()` method, you're instructing the browser about the kind of data it is about to receive, so it knows how to render the data. The values that JavaScript accepts are

```
text/html
text/plain
image/gif
image/jpeg
image/xbm
plugIn
```

If you omit the parameter, JavaScript assumes the most popular type, `text/html` — the kind of data you typically assemble in a script prior to writing to the window. The `text/html` type includes any images that the HTML references. Specifying any of the image types means that you have the raw binary representation of the image you want to appear in the new document — possible, but unlikely.

Another possibility is to direct the output of a `write()` method to a Netscape plug-in. For the *mimeType* parameter, specify the plug-in's MIME type (for example, `application/x-director` for Shockwave). Again, the data you write to a plug-in must be in a form that it knows how to handle. The same mechanism also works for writing data directly to a helper application.


Note

Internet Explorer 3 does not accept any parameters for the `document.open()` method. Internet Explorer 4 accepts only the `text/html` MIME type.

Navigator 4 includes a second, optional parameter to the method: `replace`. This parameter does for the `document.open()` method what the `replace()` method does for the location object. For `document.open()`, it means that the new document you are about to write replaces the previous document in the window or frame from being recorded to that window or frame's history.

Finally, be aware that only Navigator 3 or later enables you to use `document.open()` in the same window or frame as the one containing the script that invokes the `document.open()` method. Attempting to reopen the script's own document with this method in Navigator 2 usually leads to a crash of the browser.

Example

You can see an example of where the `document.open()` method fits in the scheme of dynamically creating content for another frame in the discussion of the `document.write()` method, later in this chapter.

Related Items: `document.close()` method; `document.clear()` method; `document.write()` method; `document.writeln()` method.

releaseEvents(eventTypeList)

Returns: Nothing.

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

If your scripts have enabled event capture for the document object (or window or layer, for that matter), you can turn off that capture with the `releaseEvents()` method. This does not inhibit events from reaching their intended target. In fact, by releasing capture from a higher object, released events don't bother stopping at those higher objects anymore.

See the discussion of the `releaseEvents()` method for the window object in Chapter 14. The behavior of the `releaseEvents()` method for all objects is the same.

Related Items: `document.captureEvents()` method; `document.routeEvent()` method.

routeEvent(event)

Returns: Nothing.

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

If you turn on event capturing in the window, document, or layer object (via their respective `captureEvents()` methods), the event handler you assign to those events really captures those events, preventing them from ever reaching their intended targets. For some page designs this is intentional, for it allows the higher-level object to handle all events of a particular type. But if your goal is to perform some preprocessing of events before they reach their destination, you need a way to pass that event along its regular path. That's what the `routeEvent()` method is for.

See the discussion of the `routeEvent ()` method for the window object in Chapter 14. The behavior of the `routeEvent ()` method for all objects is the same.

write("string1" [,"string2" ... [, "stringn"]]) writeln("string1" [,"string2" ... [, "stringn"]])

Returns: Boolean true if successful.

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

Both of these methods send text to a document for display in its window. The only difference between the two methods is that `document.writeln()` appends a carriage return to the end of the string it sends to the document (but you must still write a `<BR>` to insert a line break).

A common, incorrect conclusion that many JavaScript newcomers make is that these methods enable a script to modify the contents of an existing document. This is not true. Once a document has loaded into a window (or frame), the only text that you can modify without reloading or rewriting the entire page is the content of text and textarea objects (the exception to this is the manner in which Internet Explorer 4's extended object model allows text and HTML substitutions after document loading). In fact, because of bugs on some versions of Navigator 2, attempting to write to an existing document may cause the browser to crash. The behavior was much improved in Navigator 3, and I have more to say about that in a moment.

The two safest ways to use the `document.write()` and `document.writeln()` methods are to

✦ Embed a script in an HTML document to write some or all of the page's content

✦ Send HTML code to either a new window or to a separate frame in a multiframe window

For the first case, you essentially interlace script segments within your HTML. The scripts run as the document loads, writing whatever scripted HTML content you like. This task is exactly what you did in script1.htm in Chapter 3.

In the latter case, a script can gather input from the user in one frame and then algorithmically determine the layout and content destined for another frame. The script assembles the HTML code for the other frame as a string variable (including all necessary HTML tags). Before the script can write anything to the frame, it can optionally open the layout stream (to close the current document in that frame) with the `parent.frameName.document.open()` method. In the next step, a `parent.frameName.document.write()` method pours the entire string into the other frame. Finally, a `parent.frameName.document.close()` method ensures that the total data stream is written to the window. Such a frame looks just the same as if it were created by a source document on the server rather than on the fly in memory. The document object of that window or frame is a full citizen as a JavaScript document object. You can, therefore, even include scripts as part of the HTML specification for one of these temporary HTML pages.

Starting with Navigator 3 and Internet Explorer 3, you can write to the current window, but you should be prepared for the consequences. Once an HTML document (containing the script that is going to write) is loaded, the page's incoming stream has already closed. If you then attempt to apply a series of `document.write()` statements, the first `document.write()` method completely removes all vestiges of the original document. That includes all of its objects and scripted variable values. Therefore, if you try to assemble a new page with a series of `document.write()` statements, the parameters to the method cannot include any object references or variables from the original document: They will be gone before the second `document.write()` statement executes. To get around this potential problem, assemble the content for the new screenful of content as one variable value and then pass that variable as the parameter to a single `document.write()` statement. Also be sure to include a `document.close()` statement in the next line of script.

Assembling HTML in a script to be written via the `document.write()` method often requires skill in concatenating string values and nesting strings. A number of JavaScript string object shortcuts facilitate the formatting of text with HTML tags (see Chapter 27 for details).

Whether your script should send lots of small strings via multiple `document.write()` methods or assemble a larger string to be sent via one `document.write()` method depends partly on the situation (especially when writing to the current document in Navigator 3) and partly on style. From a performance standpoint, a fairly standard procedure is to do more preliminary work in memory and place as few I/O (input/output) calls as possible. On the other hand, it's easier to make a difficult-to-track mistake in string concatenation when

you assemble longer strings. My personal preference is to assemble longer strings, but you should use the system that's most comfortable for you.

You may see another little-known way of passing parameters to these methods. Instead of concatenating string values with the plus operator, you can also bring string values together by separating them with commas. For example, the following two statements produce the same results:

```
document.write("Today is " + new Date())
document.write("Today is ",new Date())
```

Neither form is better than the other, so use the one that feels more comfortable to your existing programming style.

Using the `document.open()`, `document.write()`, and `document.close()` methods to display images in a document requires some small extra steps. First, any URL assignments you write via `document.write()` must be complete (not relative) URL references (especially for users of Navigator 2). Accomplishing this reliably on your HTML authoring computer and the Web server may require you to algorithmically establish the pathname to the current document on the server (see Listing 15-5 in the preceding chapter).

The other image trick is to be sure to specify `HEIGHT` and `WIDTH` attributes for every image, scripted or otherwise. Navigator 2 requires these attributes, and document-rendering performance will be improved on all platforms, because the values help the browser lay out elements even before their details are loaded.

In addition to the `document.write()` example that follows (see Listings 16-13, 16-14, and 16-15), you can find fuller implementations that use this method to assemble images in and bar charts in the bonus applications on the CD-ROM. Because you can assemble any valid HTML as a string to be written to a window or frame, a customized, on-the-fly document can be as elaborate as the most complex HTML document you can imagine.

Example

The example in Listings 16-13 through 16-15 demonstrates several important points about using the `document.write()` or `document.writeln()` methods for writing to another frame. First is the fact that you can write any HTML code to a frame, and the browser accepts it as if the source code came from an HTML file somewhere. In the example, I assemble a complete HTML document, including basic HTML tags for completeness.

Listing 16-13: **Frameset for document.write() Example**

```
<HTML>
<HEAD>
<TITLE>Writin' to the doc</TITLE>
</HEAD>
<FRAMESET ROWS="50%,50%">
       <FRAME NAME="Frame1" SRC="lst16-14.htm">
       <FRAME NAME="Frame2" SRC="lst16-15.htm">
</FRAMESET>
</HTML>
```

Listing 16-14: **document.write() Example**

```
<HTML>
<HEAD>
<TITLE>Document Write Controller</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function takePulse(form) {
       var msg = "<HTML><HEAD><TITLE>On The Fly with " +
form.yourName.value + "</TITLE></HEAD>"
       msg += "<BODY BGCOLOR='salmon'><H1>Good Day " +
form.yourName.value + "!</H1><HR>"
       for (var i = 0; i < form.how.length; i++) {
          if (form.how[i].checked) {
             msg += form.how[i].value
             break
          }
       }
       msg += "<P>Make it a great day!</BODY></HTML>"
       parent.Frame2.document.open()
       parent.Frame2.document.write(msg)
       parent.Frame2.document.close()
}
function getTitle() {
       alert("Lower frame document.title is now:" +
parent.Frame2.document.title)
}
</SCRIPT>
</HEAD>

<BODY>
Fill in a name, and select how that person feels today. Then click
"Write To Below" to see the results in the bottom frame.
<FORM>
Enter your first name:<INPUT TYPE="text" NAME="yourName"
VALUE="Dave"><P>
How are you today? <INPUT TYPE="radio" NAME="how" VALUE="I hope that
feeling continues forever." CHECKED>Swell
<INPUT TYPE="radio" NAME="how" VALUE="You may be on your way to
feeling Swell">Pretty Good
<INPUT TYPE="radio" NAME="how" VALUE="Things can only get better from
here.">So-So<P>
<INPUT TYPE="button" NAME="enter" VALUE="Write To Below"
onClick="takePulse(this.form)">
<HR>
<INPUT TYPE="button" NAME="peek" VALUE="Check Lower Frame Title"
onClick="getTitle()">
</BODY>
</HTML>
```

Listing 16-15: **Placeholder for Listing 16-13**

```
<HTML>
<HEAD>
<TITLE>Placeholder</TITLE>
<BODY>
</BODY>
</HTML>
```

Figure 16-2 shows an example of the frame written by the script.

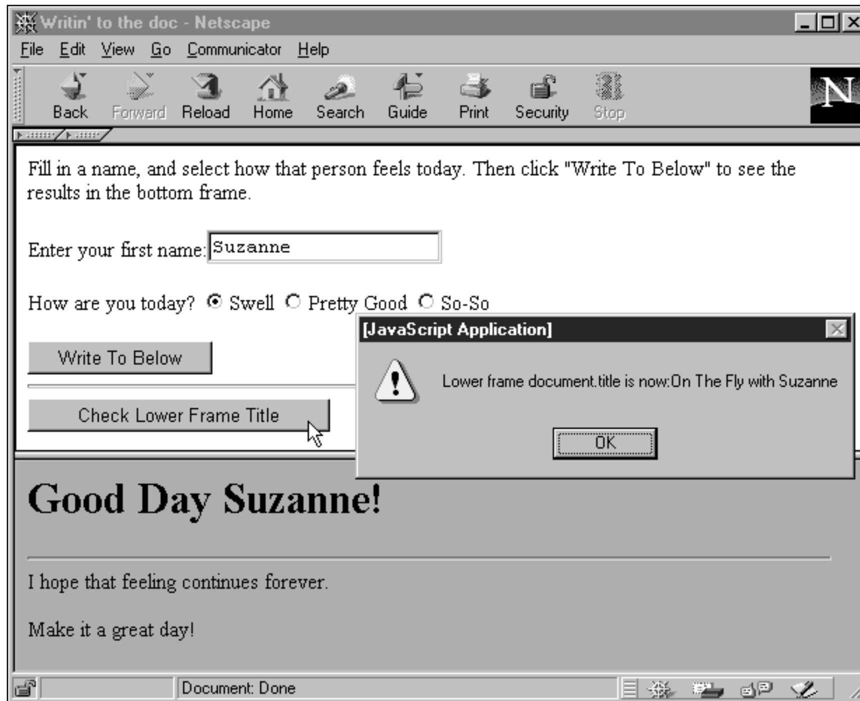


Figure 16-2: Clicking the Write To Below button in the upper frame causes a script to assemble and write HTML for the bottom frame.

A second point to note is that this example customizes the content of the document based on user input. This makes the experience of working with your Web page feel far more interactive to the user — yet you're doing it without any CGI programs running on the server. Although this is a pretty basic computer programming kind of interaction, this capability is relatively new to Web page authoring.

The third point I want to bring home is that the document created in the separate frame by the `document.write()` method is a real JavaScript document object. In this example, for instance, the `<TITLE>` tag of the written document

changes if you redraw the lower frame after changing the entry of the name field in the upper frame. If you click the lower button after updating the bottom frame, you see that the `document.title` property has, indeed, changed to reflect the `<TITLE>` tag written to the browser in the course of displaying the frame's page (except in Macintosh Navigator 4, which exhibits a bug for this property in a dynamically written document). The fact that you can artificially create full-fledged, JavaScript document objects on the fly represents one of the most important powers of serverless CGI scripting (for information delivery to the user) with JavaScript. You have much to take advantage of here if your imagination is up to the task. To print or view the source of a document written with JavaScript, you must use Navigator 3 or later.

Notice that with Navigator 3, you could easily modify Listing 16-14 to write the results to the same frame as the document containing the field and buttons. Instead of specifying the lower frame

```
parent.frames[1].document.open()
parent.frames[1].document.write(msg)
parent.frames[1].document.close()
```

the code simply could have used

```
document.open()
document.write(msg)
document.close()
```

This code would replace the form document with the results and not require any frames in the first place. Because the code assembles all of the content for the new document into one variable value, that data survives the one `document.write()` method.

The frameset document (Listing 16-13) creates a blank frame by loading a blank document (Listing 16-15). An alternative I highly recommend is to have the framesetting document fill the frame with a blank document of its own creation. See "Blank Frames" in Chapter 14 for further details about this technique for Navigator 3 and later.

Related Items: `document.open()`; `document.close()`; `document.clear()`.