The Window Object

CHAPTER 14

In This Chapter

Scripting communication among multiple frames

Creating and managing new windows

Controlling the size, position, and appearance of the browser window

A quick look at the Netscape document object model diagram in Chapter 13 (Figure 13-1) reveals that the window object is the outermost, most global container of all document-related objects in the JavaScript world. All HTML and JavaScript activity takes place inside a window. That window may be a standard Windows, Mac, or Xwindows application-style window, complete with scrollbars, toolbars and other "chrome"; if you have Navigator 4 or Internet Explorer 4 running in certain modes, the window may appear in the guise of the underlying desktop itself. A frame is also a window, even though it doesn't have many accoutrements beyond scrollbars. The window object is where everything begins in JavaScript, and this chapter begins the in-depth investigation of JavaScript objects with the window.

Of all the Navigator document model objects, the window object has by far the most terminology associated with it. This necessitates an abnormally long chapter to keep the discussion in one place. Use the running footers as a navigational aid through this substantial collection of information.

Window Terminology

The window object is often a source of confusion when you first learn about the document object model. A number of synonyms for window objects muck up the works: *top, self, parent,* and *frame*. Aggravating the situation is that these terms are also properties of a window object. Under some conditions, a window is its own parent, but if you define a frameset with two frames, there is only one parent among a total of three window objects. It doesn't take long before the whole subject can make your head hurt.

If you do not use frames in your Web applications, all of these headaches never appear. But if frames are part of your design plan, you should get to know how frames affect the object model.

Frames

The application of frames has become a religious issue among page authors: some swear by them, while others swear at them. I believe there can be compelling reasons to use frames at times. For example, if you have a document that requires considerable scrolling to get through, you may want to maintain a static set of navigation controls visible at all times. By placing those controls — be they links or image maps — in a separate frame, you have made the controls available for immediate access, regardless of the scrolled condition of the main document.

Creating frames

The task of defining frames in a document remains the same whether or not you're using JavaScript. The simplest framesetting document consists of tags that are devoted to setting up the frameset, as follows:

```
<HTML>
<HEAD>
<TITLE>My Frameset</TITLE>
</HEAD>
<FRAMESET>
        <FRAME NAME="Frame1" SRC="document1.html">
        <FRAME NAME="Frame2" SRC="document2.html">
</FRAMESET>
</HTML>
```

The preceding HTML document, which the user never sees, defines the frameset for the entire browser window. Each frame must have a URL reference (specified by the `SRC` attribute) for a document to load into that frame. For scripting purposes, assigning a name to each frame with the `NAME` attribute greatly simplifies scripting frame content.

The frame object model

Perhaps the key to successful frame scripting is understanding that the object model in the browser's memory at any given instant is determined by the HTML tags in the currently loaded documents. All canned object model graphics, such as Figure 13-1 in this book, do not reflect the precise object model for your document or document set.

For a single, frameless document, the object model starts with just one window object, which contains one document, as shown in Figure 14-1. In this simple structure, the window object is the starting point for all references to any loaded object. Because the window is always there — it must be there for a document to load into — a reference to any object in the document can omit a reference to the current window.

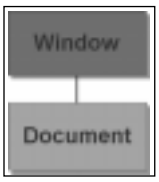

Figure 14-1: The simplest window-document relationship

In a simple two-framed frameset model (Figure 14-2), the browser treats the container of the initial, framesetting document as the *parent* window. The only visible evidence that the document exists is that the framesetting document's title appears in the browser window title bar.

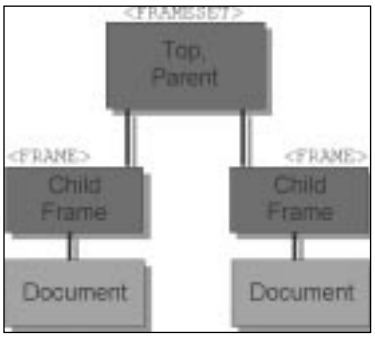

Figure 14-2: The parent and frames are part of the object model.

Each <FRAME> tag inside the <FRAMESET> tag set creates another window object into which a document is loaded. Each of those frames, then, has a document object associated with it. From the point of view of a given document, it has a single window container, just like the model shown in Figure 14-2. And although the parent frame object is not visible to the user, it remains in the object model in memory. The presence of the parent often makes it a convenient repository for variable data that needs to be shared by multiple child frames or must persist between loading of different documents inside a child frame.

In even more complex arrangements, as shown in Figure 14-3, a child frame itself may load a framesetting document. In this situation, the differentiation between the parent and top object starts to come into focus. The top window is the only one in common with all frames in Figure 14-3. As you will see in a moment, when frames need to communicate with other frames (and their documents), you must fashion references to the distant object via the window object they all have in common.

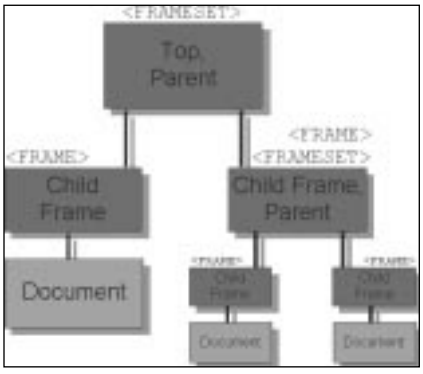

Figure 14-3: Three generations of window objects

Referencing frames

The purpose of an object reference is to help JavaScript locate the desired object in the object model currently held in memory. A reference is a road map for the browser to follow, so that it can track down, say, the value of a particular text field in a particular document. Therefore, when you construct a reference, think about where the script appears in the object model and how the reference can help the browser determine where it should go to find the distant object. In a two-generation scenario such as the one shown in Figure 14-2, three intergenerational references are possible:

- Parent-to-child
- Child-to-parent
- Child-to-child

Assuming that you need to access an object, function, or variable in the relative's frame, the following are the corresponding reference structures: `frameName.objFuncVarName`; `parent.objFuncVarName`; `parent.frameName.objFuncVarName`.

The rule is this: Whenever a reference must point to another frame, begin the reference with the window object that the two destinations have in common. To demonstrate that rule on the complex model in Figure 14-3, if the left-hand child frame's document needs to reference the document at the bottom right of the map, the reference structure is

```
top.frameName.frameName.document. ...
```

Follow the map from the top window object down through two frames to the final document. JavaScript has to take this route, so your reference must help it along.

Top versus parent

After seeing the previous object maps and reference examples, you may be wondering, Why not use `top` as the leading object in all trans-frame references? From an object model point of view, you'll have no problem doing that: A `parent` in a two-generation scenario is also the `top` window. What you can't count on, however, is your framesetting document always being the top window object in someone's browser. Take the instance where a Web site loads other Web sites into one of its frames. At that instant, the `top` window object belongs to someone else. If you always specify `top` in references intended just for your `parent` window, your references won't work and will probably lead to script errors for the user. My advice, then, is to use `parent` in references whenever you mean one generation above the current document.

Preventing framing

You can use your knowledge of `top` and `parent` references to prevent your pages from being displayed inside another Web site's frameset. Your top-level document must check whether it is loaded into its own top or parent window. When a document is in its own top window, a reference to the `top` property of the current window is equal to a reference to the current window (the window synonym `self` seems most grammatically fitting here). If the two values are not

equal, you can script your document to reload itself as a top-level document. When it is critical that your document be a top-level document, include the script in Listing 14-1 in the head portion of your document:

Listing 14-1: Prevention from Getting "Framed"

```
<SCRIPT LANGUAGE="JavaScript">
if (top != self) {
    top.location = location
}
</SCRIPT>
```

Your document may appear momentarily inside the other site's frameset, but then the slate is wiped clean, and your top-level document rules the browser window.

Switching from frames to frameless

Some sites load themselves in a frameset by default and offer users the option of getting rid of the frames. You cannot dynamically change the makeup of a frameset once it has loaded, but you can load the content page of the frameset into the main window. Simply include a button or link whose action loads that document into the top window object:

```
top.location = "mainBody.html"
```

A switch back to the frame version entails nothing more complicated than loading the framesetting document.

Inheritance versus containment

Scripters who have experience in object-oriented programming environments probably expect frames to inherit properties, methods, functions, and variables defined in a parent object. That's *not* the case in JavaScript. You can, however, still access those parent items when you make a call to the item with a complete reference to the parent. For example, if you want to define a deferred function in the framesetting parent document that all frames can share, the scripts in the frames would refer to that function with this reference:

```
parent.myFunc()
```

You can pass arguments to such functions and expect returned values.

Navigator 2 bug: Parent variables

Some bugs linger in Navigator 2 that cause problems when accessing variables in a parent window from one of its children. If a document in one of the child frames unloads, a parent variable value that depends on that frame may get scrambled or disappear. Using a temporary document.cookie for global variable values may be a better solution. For Navigator 3 and up, you should declare parent variables that are updated from child frames as first-class string objects (with the new String() constructor) as described in Chapter 27.

Frame synchronization

A pesky problem for some scripters' plans is that including immediate scripts in the framesetting document is dangerous — if not crash-prone in Navigator 2. Such scripts tend to rely on the presence of documents in the frames being created by this framesetting document. But if the frames have not yet been created and their documents loaded, the immediate scripts will likely crash and burn.

One way to guard against this problem is to trigger all such scripts from the frameset's `onLoad=` event handler. This handler won't trigger until all documents have successfully loaded into the child frames defined by the frameset. At the same time, be careful with `onLoad=` event handlers in the documents going into a frameset's frames. If one of those scripts relies on the presence of a document in another frame (one of its brothers or sisters), you're doomed to eventual failure. Anything coming from a slow network or server to a slow modem can get in the way of other documents loading into frames in the ideal order.

One way to work around this problem is to create a string variable in the parent document to act as a flag for the successful loading of subsidiary frames. When a document loads into a frame, its `onLoad=` event handler can set that flag to a word of your choice to indicate that the document has loaded. A better solution, however, is to construct the code so that the parent's `onLoad=` event handler triggers all the scripts that you want to run after loading. Depending on other frames is a tricky business, but the farther the installed base of Web browsers gets from Navigator 2, the less the associated risk. For example, beginning with Navigator 3, if a user resizes a window, the document does not reload itself, as it used to in Navigator 2. Even so, you still should test your pages thoroughly for any residual effects that may accrue if someone resizes a window or clicks Reload.

Blank frames

Often, you may find it desirable to create a frame in a frameset but not put any document in it until the user has interacted with various controls or other user interface elements in other frames. Navigator has a somewhat empty document in one of its internal URLs (`about:blank`). But with Navigator 2 and 3 on the Macintosh, an Easter egg–style message appears in that window when it displays. This URL is also not guaranteed to be available on non-Netscape browsers. If you need a blank frame, let your framesetting document write a generic HTML document to the frame directly from the `SRC` attribute for the frame, as shown in the skeletal code in Listing 14-2. It requires no additional transactions to load an "empty" HTML document.

Listing 14-2: **Creating a Blank Frame**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
function blank() {
      return "<HTML></HTML>"
}
//-->
```

```
</SCRIPT>
</HEAD>
<FRAMESET>
        <FRAME NAME="Frame1" SRC="someURL.html">
        <FRAME NAME="Frame2" SRC="javascript:'parent.blank()'">
</FRAMESET>
</HTML>
```

Viewing frame source code

Studying other scripters' work is a major learning tool for JavaScript (or any programming language). Beginning with Navigator 3, you can easily view the source code for any frame, including those frames whose content is generated entirely or in part by JavaScript. Click the desired frame to activate it (a subtle border appears just inside the frame on some browser versions, but don't be alarmed if the border doesn't appear). Then select Frame Source from the View menu (or right-click submenu). You can also print or save a selected frame (from the File menu).

Window Object

Properties	*Methods*	*Event Handlers*
closed	alert()	onBlur=
defaultStatus	back()	onDragDrop=
document	blur()	onFocus=
frames[]	captureEvents()	onLoad=
history	clearInterval()	onMove=
innerHeight	clearTimeout()	onResize=
innerWidth	close()	onUnload=
location	confirm()	
locationbar	disableExternalCapture()	
menubar	enableExternalCapture()	
name	find()	
onerror	handleEvent()	
opener	forward()	
outerHeight	home()	
outerWidth	moveBy()	
pageXOffset	moveTo()	

Properties	Methods	Event Handlers
pageYOffset	focus()	
parent	open()	
personalbar	print()	
scrollbars	prompt()	
self	releaseEvents()	
status	resizeBy()	
statusbar	resizeTo()	
toolbar	routeEvent()	
top	scroll()	
window	scrollBy()	
	scrollTo()	
	setInterval()	
	setTimeout()	
	stop()	

Syntax

Creating a window:

```
windowObject = window.open([parameters])
```

Accessing window properties or methods:

```
window.property | method([parameters])
self.property | method([parameters])
windowObject.property | method([parameters])
```

About this object

The window object has the unique position of being at the top of the JavaScript object hierarchy. This exalted location gives the window object a number of properties and behaviors unlike those of any other object.

Chief among its unique characteristics is that because everything takes place in a window, you can usually omit the window object from object references. You've seen this behavior in previous chapters when I invoked document methods such as `document.write()`. The complete reference is `window.document.write()`. But because the activity was taking place in the window that held the document running the script, that window was assumed to be part of the reference. For single-frame windows, this concept is simple enough to grasp.

As previously stated, among the list of properties for the window object is one called `self`. This property is synonymous with the window object itself (which is

why it shows up in hierarchy diagrams as an object). Having a property of an object that is the same name as the object may sound confusing, but this situation is not that uncommon in object-oriented environments. I discuss the reasons why you may want to use the `self` property as the window's object reference in the `self` property description that follows.

As indicated earlier in the syntax definition, you don't always have to specifically create a window object in JavaScript code. When you start your browser, it usually opens a window. That window is a valid window object, even if the window is blank. Therefore, when a user loads your page into the browser, the window object part of that document is automatically created for your script to access as it pleases.

Your script's control over an existing (already open) window's user interface elements varies widely with the browser and browser version for which your application is intended. With the exception of Navigator 4, the only change you can make to an open window is to the status line at the bottom of the browser window. With Navigator 4, however, you can control such properties as the size, location, and "chrome" elements (toolbars and scrollbars, for example) on the fly. Many of these properties can be changed beyond specific safe limits only if you cryptographically sign the scripts (see Chapter 40) and the user grants permission for your scripts to make those modifications.

Window properties are far more flexible on all browsers when your scripts generate a new window (with the `window.open()` method): You can influence the size, toolbar, or other view options of a window. Navigator 4 provides even more options for new windows, including whether the window should remain at a fixed layer among desktop windows and whether the window should even display a title bar. Again, if an option can conceivably be used to deceive a user (for example, hiding one window that monitors activity in another window), signed scripts and user permission are necessary.

The window object is also the level at which a script asks the browser to display any of three styles of dialog boxes (a plain alert dialog box, an OK/Cancel confirmation dialog box, or a prompt for user text entry). Although dialog boxes are extremely helpful for cobbling together debugging tools for your own use (Chapter 45), they can be very disruptive to visitors who navigate through Web sites. Because JavaScript dialog boxes are *modal* (that is, you cannot do anything else in the browser — or anything at all on a Macintosh — until you dismiss the dialog box), use them sparingly, if at all. Remember that some users may create macros on their computers to visit sites unattended. Should such an automated access of your site encounter a modal dialog box, it would be trapped on your page until a human could intervene.

All dialog boxes generated by JavaScript in Netscape browsers identify themselves as being generated by JavaScript (less egregiously so in Navigator 4). This is primarily a security feature to prevent deceitful, unsigned scripts from creating system- or application-style dialog boxes that convince visitors to enter private information. It should also discourage dialog box usage in Web page design. And that's good, because dialog boxes tend to be disruptive.

Why are dialog boxes window methods?

I find it odd that dialog boxes are generated as window methods rather than as methods of the navigator object. These dialogs don't really belong to any window. In fact, their modality prevents the user from accessing any window.

To my way of thinking, these methods (and the ones that create or close windows) belong to an object level one step above the window object in the hierarchy (which would include the properties of the navigator object described in Chapter 25). I don't lose sleep over this setup, though. If the powers that be insist on making these dialog boxes part of the window object, that's how my code will read.

Netscape's JavaScript dialog boxes are not particularly flexible in letting you fill them with text or graphic elements beyond the basics. In fact, you can't even change the text of the dialog buttons or add a button. With Navigator 4, however, you can use signed scripts to generate a window that looks and behaves very much like a modal dialog box. Into that window you can load any HTML you like. Thus, you can use such a window as an entry form, preferences selector, or whatever else makes user interface sense in your application. Internet Explorer 4, on the other hand, has a separate method and set of properties specifically for generating a modal dialog. The two scripted solutions are not compatible with each other.

Properties

closed

Value: Boolean **Gettable:** Yes **Settable:** No

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

When you create a subwindow with the `window.open()` method, you may need to access object properties from that subwindow, such as setting the value of a text field. Access to the subwindow is via the window object reference that is returned by the `window.open()` method, as in the following code fragment:

```
var newWind = window.open("someURL.html","subWind")
...
newWind.document.entryForm.ZIP.value = "00000"
```

In this example, the `newWind` variable is not linked "live" to the window, but is only a reference to that window. If the user should close the window, the `newWind` variable still contains the reference to the now missing window. Thus, any script reference to an object in that missing window will likely cause a script error. What

you need to know before accessing items in a subwindow is whether the window is still open.

The `closed` property returns true if the window object has been closed either by script or by the user. Any time you have a script statement that can be triggered after the user has an opportunity to close the window, test for the `closed` property before executing that statement.

As a workaround for Navigator 2, any property of a closed window reference returns a null value. Thus, you can test whether, say, the `parent` property of the new window is null: If so, the window has already closed. Internet Explorer 3, on the other hand, triggers a scripting error if you attempt to access a property of a closed window — there is no error-free way to detect whether a window is open or closed in Internet Explorer 3. The `window.closed` property is implemented in Internet Explorer 4.

Example

In Listing 14-3, I have created the ultimate cross-platform window opening and closing sample. It takes into account the lack of the `opener` property in Navigator 2, the missing `closed` property in Navigator 2 and Internet Explorer 3, and even provides an ugly but necessary workaround for Internet Explorer 3's inability to gracefully see if a subwindow is still open.

The script begins by initializing a global variable, `newWind`, which is used to hold the object reference to the second window. This value needs to be global so that other functions can reference the window for tasks such as closing. Another global variable, `isIE3`, is a Boolean flag that will let the window closing routines know whether the visitor is using Internet Explorer 3 (see details about the `navigator.appVersion` property in Chapter 25).

For this example, the new window contains some HTML code written dynamically to it, rather than loading an existing HTML file into it. Therefore, the URL parameter of the `window.open()` method is left as an empty string. It is vital, however, to assign a name in the second parameter to accommodate the Internet Explorer 3 workaround for closing the window. After the new window is opened, the script assigns an `opener` property to the object if one is not already assigned (this is needed only for Navigator 2). After that, the script assembles HTML to be written to the new window via one `document.write()` statement. The `document.close()` method closes writing to the document — a different kind of close than a window close.

A second function is responsible for closing the subwindow. To accommodate Internet Explorer 3, the script appears to create another window with the same characteristics as the one opened earlier in the script. This is the trick: If the earlier window exists (with exactly the same parameters and a name other than an empty string), Internet Explorer does not create a new window even with the `window.open()` method executing in plain sight. To the user, nothing unusual appears on the screen. Only if the user has closed the subwindow do things look weird for Internet Explorer 3 users. The `window.open()` method momentarily creates that subwindow. This is necessary because a "living" window object must be available for the upcoming test of window existence (Internet Explorer 3 displays a script error if you try to address a missing window, while Navigator and Internet Explorer 4 simply return friendly null values).

As a final test, an `if` condition looks at two conditions: 1) if the window object has ever been initialized with a value other than null (in case you click the window closing button before ever having created the new window) and 2) if the window's `closed` property is null or false. If either condition is true, the `close()` method is sent to the second window.

Listing 14-3: **Checking Before Closing a Window**

```
<HTML>
<HEAD>
<TITLE>window.closed Property</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// initialize global var for new window object
// so it can be accessed by all functions on the page
var newWind
// set flag to help out with special handling for window closing
var isIE3 = (navigator.appVersion.indexOf("MSIE 3") != -1) ? true :
false
// make the new window and put some stuff in it
function newWindow() {
        var output = ""
        newWind = window.open("","subwindow","HEIGHT=200,WIDTH=200")
        // take care of Navigator 2
        if (newWind.opener == null) {
           newWind.opener = window
        }
        output += "<HTML><BODY><H1>A Sub-window</H1>"
        output += "<FORM><INPUT TYPE='button' VALUE='Close Main Window'"
        output +="onClick='window.opener.close()'></FORM></BODY></HTML>"
        newWind.document.write(output)
        newWind.document.close()
}
// close subwindow, including ugly workaround for IE3
function closeWindow() {
        if (isIE3) {
           // if window is already open, nothing appears to happen
           // but if not, the subwindow flashes momentarily (yech!)
           newWind = window.open("","subwindow","HEIGHT=200,WIDTH=200")
        }
        if (newWind && !newWind.closed) {
           newWind.close()
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" VALUE="Open Window" onClick="newWindow()"><BR>
<INPUT TYPE="button" VALUE="Close it if Still Open" onClick="closeWindow()">
</FORM>
</BODY>
</HTML>
```

To complete the example of the window opening and closing, notice that the subwindow is given a button whose `onClick=` event handler closes the main window. In Navigator 2 and Internet Explorer 3, this occurs without complaint. But in Navigator 3 and up and Internet Explorer 4, the user will likely be presented with an alert asking to confirm the closure of the main browser window.
Related Items: `window.open()` method; `window.close()` method.

defaultStatus

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

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

After a document is loaded into a window or frame, the statusbar's message field can display a string that is visible any time the mouse pointer is not atop an object that takes precedence over the statusbar (such as a link object or an image map). The `window.defaultStatus` property is normally an empty string, but you can set this property at any time. Any setting of this property will be temporarily overridden when a user moves the mouse pointer atop a link object (see `window.status` property for information about customizing this temporary statusbar message).

Probably the most common time to set the `window.defaultStatus` property is when a document loads into a window. You can do this as an immediate script statement that executes from the Head or Body portion of the document or as part of a document's `onLoad=` event handler.

The defaultStatus property does not work well in Navigator 2 or Internet Explorer 3, and experiences problems in Navigator 3, especially on the Macintosh (where the property doesn't change even after loading a different document into the window). Many users simply don't see the statusbar change during Web surfing, so don't put mission-critical information in the statusbar.

Example

Unless you plan to change the default statusbar text while a user spends time at your Web page, the best time to set the property is when the document loads. In Listing 14-4, notice how I also extract this property to reset the statusbar in an `onMouseOut=` event handler. Setting the `status` property to empty also resets the statusbar to the `defaultStatus` setting.

Listing 14-4: Setting the Default Status Message

```
<HTML>
<HEAD>
<TITLE>window.defaultStatus property</TITLE>
<SCRIPT LANGUAGE="JavaScript">
```

(continued)

Listing 14-4 *Continued*

```
window.defaultStatus = "Welcome to my Web site."
</SCRIPT>
</HEAD>
<BODY>
<A HREF="http://home.netscape.com" onMouseOver="window.status = 'Go to
your browser Home page.';return true" onMouseOut="window.status =
'';return true">Home</A><P>
<A HREF="http://home.netscape.com" onMouseOver="window.status = 'Visit
Netscape\'s Home page.';return true" onMouseOut="window.status =
window.defaultStatus;return true">Netscape</A>
</BODY>
</HTML>
```

If you need to display single or double quotes in the statusbar (as in the second link in Listing 14-4), use escape characters (`\'` and `\"`) as part of the strings being assigned to these properties.

Related Items: `window.status` property.

document

Value: Object **Gettable:** Yes **Settable:** No

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

I list the `document` property here primarily for completeness. A window object contains a single document object (although in Navigator 4, a window may also contain layers, each of which has a document object, as described in Chapter 19). The value of the `document` property is the document object, which is not a displayable value. Instead, you use the `document` property as you build references to properties and methods of the document and to other objects contained by the document, such as a form and its elements. To load a different document into a window, use the location object (see Chapter 15). The document object is described in detail in Chapter 16.

Related Items: document object.

frames

Value: Window object **Gettable:** Yes **Settable:** No

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

In a multiframe window, the top or parent window contains any number of separate frames, each of which acts like a full-fledged window object. The `frames` property (note the plural use of the word as a property name) plays a role when a statement must reference an object located in a different frame. For example, if a button in one frame is scripted to display a document in another frame, the button's event handler must be able to tell JavaScript precisely where to display the new HTML document. The `frames` property assists in that task.

To use the `frames` property to communicate from one frame to another, it should be part of a reference that begins with the `parent` or `top` property. This lets JavaScript make the proper journey through the hierarchy of all currently loaded objects to reach the desired object. To find out how many frames are currently active in a window, use this expression:

```
parent.frames.length
```

This expression returns a number indicating how many frames are defined by the parent window. This value does not, however, count further nested frames, should a third generation of frame be defined in the environment. In other words, no single property exists that you can use to determine the total number of frames in the browser window if multiple generations of frames are present.

The browser stores information about all visible frames in a numbered (indexed) array, with the first frame (that is, the topmost `<FRAME>` tag defined in the framesetting document) as number 0:

```
parent.frames[0]
```

Therefore, if the window shows three frames (whose indexes would be `frames[0]`, `frames[1]`, and `frames[2]`, respectively), the reference for retrieving the `title` property of the document in the second frame is

```
parent.frames[1].document.title
```

This reference is a road map that starts at the parent window and extends to the second frame's document and its `title` property. Other than the number of frames defined in a parent window and each frame's name (`top.frames[i].name`), no other values from the frame definitions are directly available from the frame object via scripting.

Using index values for frame references is not always the safest tactic, however, because your frameset design may change over time, in which case the index values will also change. Instead, you should take advantage of the `NAME` attribute of the `<FRAME>` tag, and assign a unique, descriptive name to each frame. Then you can use a frame's name as an alternative to the indexed reference. For example, in Listing 14-5, two frames are assigned with distinctive names. To access the title of a document in the `JustAKid2` frame, the complete object reference is

```
parent.JustAKid2.document.title
```

with the frame name (case-sensitive) substituting for the `frames[1]` array reference. Or, in keeping with JavaScript flexibility, you can use the object name in the array index position:

```
parent.frames["JustAKid2"].document.title
```

The supreme advantage to using frame names in references is that no matter how the frameset may change over time, a reference to a named frame will always find that frame, although its index value (that is, position in the frameset) may change.

Example

Listings 14-5 and 14-6 demonstrate how JavaScript treats values of frame references from objects inside a frame. The same document is loaded into each frame. A script in that document extracts info about the current frame and the entire frameset. Figure 14-4 shows the results after loading the HTML document in Listing 14-3.

Listing 14-5: Framesetting Document for Listing 14-6

```
<HTML>
<HEAD>
<TITLE>window.frames property</TITLE>
</HEAD>
<FRAMESET COLS="50%,50%">
        <FRAME NAME="JustAKid1" SRC="lst14-06.htm">
        <FRAME NAME="JustAKid2" SRC="lst14-06.htm">
</FRAMESET>
</HTML>
```

A call to determine the number (length) of frames returns 0 from the point of view of the current frame referenced. That's because each frame here is a window that has no nested frames within it. But add the `parent` property to the reference, and the scope zooms out to take into account all frames generated by the parent window's document.

Listing 14-6: Showing Various window Properties

```
<HTML>
<HEAD>
<TITLE>Window Revealer II</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function gatherWindowData() {
        var msg = ""
        msg += "<B>From the point of view of this frame:</B><BR>"
        msg += "window.frames.length: " + window.frames.length + "<BR>"
        msg += "window.name: " + window.name + "<P>"
        msg += "<B>From the point of view of the framesetting
document:</B><BR>"
        msg += "parent.frames.length: " + parent.frames.length + "<BR>"
        msg += "parent.frames[0].name: " + parent.frames[0].name
        return msg
}
</SCRIPT>
</HEAD>
```

```
<BODY>
<SCRIPT LANGUAGE="JavaScript">
document.write(gatherWindowData())
</SCRIPT>
</BODY>
</HTML>
```

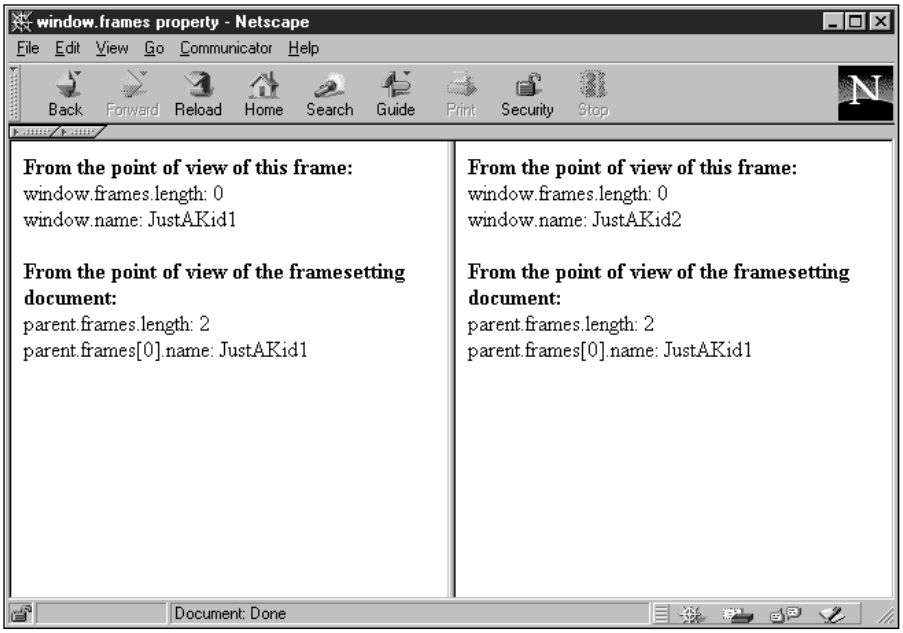

Figure 14-4: Property readouts from both frames loaded from Listing 14-5

The last statement in the example shows how to use the array syntax (brackets) to refer to a specific frame. All array indexes start with 0 for the first entry. Because the document asks for the name of the first frame (`parent.frames[0]`), the response is `JustAKid1` for both frames.

Related Items: `window.parent` property; `window.top` property.

history

Value: Object **Gettable:** Yes **Settable:** No

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

See the discussion of the history object in Chapter 15.

innerHeight
innerWidth
outerHeight
outerWidth

Value: Integer **Gettable:** Yes **Settable:** Yes

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

Navigator 4 lets scripts adjust the height and width of any window, including the main browser window. This can be helpful when your page shows itself best with the browser window sized to a particular height and width. Rather than relying on the user to size the browser window for optimum viewing of your page, you can dictate the size of the window (although the user can always manually resize the main window). And because you can examine the operating system of the visitor via the navigator object (see Chapter 25), you can size a window to adjust for the differences in font and form element rendering on different platforms.

Netscape provides two different points of reference for measuring the height and width of a window: *inner* and *outer*. Both are measured in pixels. The inner measurements are that of the active document area of a window (sometimes known as a window's *content region*). If the optimum display of your document depends on the document display area being a certain number of pixels high and/or wide, the `innerHeight` and `innerWidth` properties are the ones to set.

In contrast, the outer measurements are of the outside boundary of the entire window, including whatever "chrome" is showing in the window: scrollbars, statusbar, and so on. Setting the `outerHeight` and `outerWidth` is generally done in concert with a reading of screen object properties (Chapter 25). Perhaps the most common usage of the outer properties is to set the browser window to fill the available screen area of the visitor's monitor.

A more efficient way of modifying both outer dimensions of a window is with the `window.resizeTo()` method. The method takes width and height as parameters, thus accomplishing a window resizing in one statement. Be aware that resizing a window does not adjust the location of a window. Therefore, just because you set the outer dimensions of a window to the available space returned by the screen object doesn't mean that the window will suddenly fill the available space on the monitor. Application of the `window.moveTo()` method is necessary to ensure the top-left corner of the window is at screen coordinates 0,0.

Despite the freedom that these properties afford the page author, Netscape has built in a minimum size limitation for scripts that are not cryptographically signed. You cannot set these properties such that the outer height and width of the window is smaller than 100 pixels on a side. This is to prevent an unsigned script from setting up a small or nearly invisible window that monitors activity in other windows. With signed scripts, however, windows can be made smaller than 100-by-100 pixels with the user's permission.

Example

In Listing 14-7, a number of buttons let you see the results of setting the `innerHeight`, `innerWidth`, `outerHeight`, and `outerWidth` properties.

Listing 14-7: Setting Window Height and Width

```
<HTML>
<HEAD>
<TITLE>Window Sizer</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// store original outer dimensions as page loads
var originalWidth = window.outerWidth
var originalHeight = window.outerHeight
// generic function to set inner dimensions
function setInner(width, height) {
        window.innerWidth = width
        window.innerHeight = height
}
// generic function to set outer dimensions
function setOuter(width, height) {
        window.outerWidth = width
        window.outerHeight = height
}
// restore window to original dimensions
function restore() {
        window.outerWidth = originalWidth
        window.outerHeight = originalHeight
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<B>Setting Inner Sizes</B><BR>
<INPUT TYPE="button" VALUE="600 Pixels Square"
onClick="setInner(600,600)"><BR>
<INPUT TYPE="button" VALUE="300 Pixels Square"
onClick="setInner(300,300)"><BR>
<INPUT TYPE="button" VALUE="Available Screen Space"
onClick="setInner(screen.availWidth, screen.availHeight)"><BR>
<HR>
<B>Setting Outer Sizes</B><BR>
<INPUT TYPE="button" VALUE="600 Pixels Square"
onClick="setOuter(600,600)"><BR>
<INPUT TYPE="button" VALUE="300 Pixels Square"
onClick="setOuter(300,300)"><BR>
<INPUT TYPE="button" VALUE="Available Screen Space"
onClick="setOuter(screen.availWidth, screen.availHeight)"><BR>
```

(continued)

Listing 14-7 *Continued*

```
<HR>
<INPUT TYPE="button" VALUE="Cinch up for Win95"
onClick="setInner(273,304)"><BR>
<INPUT TYPE="button" VALUE="Cinch up for Mac"
onClick="setInner(273,304)"><BR>
<INPUT TYPE="button" VALUE="Restore Original" onClick="restore()"><BR>
</FORM>
</BODY>
</HTML>
```

As the document loads, it saves the current outer dimensions in global variables. One of the buttons restores the windows to these settings. Two parallel sets of buttons set the inner and outer dimensions to the same pixel values so you can see the effects on the overall window and document area when a script changes the various properties.

Because Navigator 4 displays different-looking buttons in different platforms (as well as other elements), the two buttons contain script instructions to size the window to best display the window contents. Unfortunately, no measure of the active area of a document is available, so the dimension values were determined by trial and error before being hard-wired into the script.

Related Items: `window.resizeTo()` method; `window.moveTo()` method; screen object; navigator object.

location

Value: Object **Gettable:** Yes **Settable:** No

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

See the discussion of the location object in Chapter 15.

locationbar
menubar
personalbar
scrollbars
statusbar
toolbar

Value: Object **Gettable:** Yes **Settable:** Yes (with signed scripts)

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

Beyond the rectangle of the content region of a window (where your documents appear), the Netscape browser window displays an amalgam of bars and other features known collectively as *chrome*. All browsers can elect to remove these chrome items when creating a new window (as part of the third parameter of the `window.open()` method), but until signed scripts were available in Navigator 4, these items could not be turned on and off in the main browser window or any existing window.

Navigator 4 promotes these elements to first-class objects contained by the window object. Figure 14-5 points out where each of the six bars appears in a fully chromed window. The only element that is not part of this scheme is the window's title bar. You can create a new window without a title bar (with a signed script), but you cannot hide and show the title bar on an existing window.

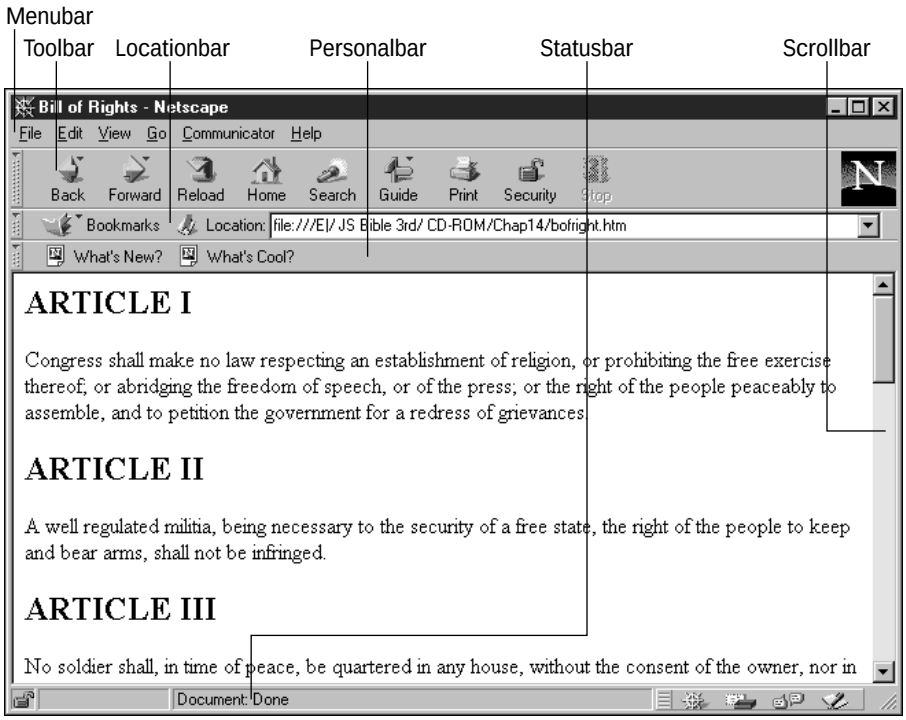

Figure 14-5: Window chrome items

Chrome objects have but one property: `visible`. Reading this Boolean value (possible without signed scripts) lets you inspect the visitor's browser window for the elements currently engaged. There is no intermediate setting or property for the expanded/collapsed state of the toolbar, locationbar, and personalbar.

Changing the visibility of these items on the fly alters the relationship between the inner and outer dimensions of the browser window. If you must carefully size a window to display content, you should adjust the chrome elements before sizing the window. Before you start changing chrome visibility on your page visitors, weigh the decision carefully. Experienced users have fine-tuned the look of their browser windows to just the way they like them. If you mess with that look, you might anger your visitors. Fortunately, changes you make to a chrome element's visibility are not stored to the user's preferences. However, the changes you make survive an unloading of the page. If you change the settings, be sure you first save the initial settings and restore them with an `onUnload=` event handler.

The Macintosh menubar is not part of the browser's window chrome. Therefore, its visibility cannot be adjusted from a script.

Example

In Listing 14-8, you can experiment with the look of a browser window with any of the chrome elements turned on and off. To run this script, you must either sign the scripts or turn on codebase principals (see Chapter 40). Java must also be enabled to use the signed script statements.

As the page loads, it stores the current state of each chrome element. One button for each chrome element triggers the `toggleBar()` function. This function inverts the visible property for the chrome object passed as a parameter to the function. Finally, the Restore button returns visibility to their original settings. Notice that the `restore()` function is also called by the `onUnload=` event handler for the document.

Listing 14-8: **Controlling Window Chrome**

```
<HTML>
<HEAD>
<TITLE>Bars Bars Bars</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// store original outer dimensions as page loads
var originalLocationbar = window.locationbar.visible
var originalMenubar = window.menubar.visible
var originalPersonalbar = window.personalbar.visible
var originalScrollbars = window.scrollbars.visible
var originalStatusbar = window.statusbar.visible
var originalToolbar = window.toolbar.visible

// generic function to set inner dimensions
function toggleBar(bar) {
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
        bar.visible = !bar.visible

netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite")
}
// restore settings
function restore() {
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
```

```
        window.locationbar.visible = originalLocationbar
        window.menubar.visible = originalMenubar
        window.personalbar.visible = originalPersonalbar
        window.scrollbars.visible = originalScrollbars
        window.statusbar.visible = originalStatusbar
        window.toolbar.visible = originalToolbar
netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite")
}
</SCRIPT>
</HEAD>
<BODY onUnload="restore()">
<FORM>
<B>Toggle Window Bars</B><BR>
<INPUT TYPE="button" VALUE="Location Bar"
onClick="toggleBar(window.locationbar)"><BR>
<INPUT TYPE="button" VALUE="Menu Bar"
onClick="toggleBar(window.menubar)"><BR>
<INPUT TYPE="button" VALUE="Personal Bar"
onClick="toggleBar(window.personalbar)"><BR>
<INPUT TYPE="button" VALUE="Scrollbars"
onClick="toggleBar(window.scrollbars)"><BR>
<INPUT TYPE="button" VALUE="Status Bar"
onClick="toggleBar(window.statusbar)"><BR>
<INPUT TYPE="button" VALUE="Tool Bar"
onClick="toggleBar(window.toolbar)"><BR>
<HR>
<INPUT TYPE="button" VALUE="Restore Original Settings"
onClick="restore()"><BR>
</FORM>
</BODY>
</HTML>
```

Related Items: `window.open()` method.

name

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

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

All window objects can have names assigned to them. Names are particularly useful for working with frames, because a good naming scheme for a multiframe environment can help you determine precisely which frame you're working with in references coming from other frames.

The main browser window, however, has no name attached to it by default. Its value is an empty string. There aren't many reasons to assign a name to the window, because JavaScript and HTML provide plenty of other ways to refer to the window object (the `top` property, the `_top` constant for `TARGET` attributes, and the `opener` property from subwindows).

If you want to attach a name to the main window, you can do so by setting the `window.name` property at any time. But be aware that because this is a window property, the life of its value extends beyond the loading and unloading of any given document. Chances are that your scripts would use the reference in only one document or frameset. Unless you restore the default empty string, your programmed window name will be present for any other document that loads later. My suggestion in this regard is to assign a name in a window's or frameset's `onLoad=` event handler and then reset it to empty in a corresponding `onUnload=` event handler:

```
<BODY onLoad="self.name = 'Main'" onUnload="self.name = ''">
```

You can see an example of this application in Listing 14-14, where setting a parent window name is helpful for learning the relationships among parent and child windows.

Related Items: `window.open()` method; `top` property.

onerror

Value: Null, Undefined, or Function Object **Gettable:** Yes **Settable:** Yes

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

Although script error dialog boxes are a scripter's best friend (if you're into debugging, that is), they can be confusing for users who have never seen such dialog boxes. JavaScript lets you turn off the display of script error windows as someone executes a script on your page. The question is: When should you turn off these dialog boxes?

Script errors generally mean that something is wrong with your script. The error may be the result of a coding mistake or, conceivably, a bug in JavaScript (perhaps on a platform version of the browser you haven't been able to test). When such errors occur, often the script won't continue to do what you intended. Hiding the script error from yourself during development would be foolhardy, because you'd never know whether unseen errors are lurking in your code. It can be equally dangerous to turn off error dialog boxes for users who may believe that the page is operating normally, when, in fact, it's not. Some data values may not be calculated or displayed correctly.

That said, I can see some limited instances of when you'd like to keep such dialog windows from appearing. For example, if you know for a fact that a platform-

specific bug trips the error message without harming the execution of the script, you may want to prevent that error alert dialog box from appearing in the files posted to your Web site. You should do this only after extensive testing to ensure that the script ultimately behaves correctly, even with the bug or error.

When the browser starts, the `window.onerror` property is `<undefined>`. In this state, all errors are reported via the normal JavaScript error window. To turn off error dialog boxes, set the `window.onerror` property to null:

```
window.onerror = null
```

You may recognize this syntax as looking like a property version of an event handler described earlier in this chapter. For Netscape browsers, however, no `onError=` event handler exists that you specify in an HTML tag associated with the window object. The error event just happens. (Internet Explorer 4 lets you add an `onError=` event handler to just about every object tag, but these are ignored by Netscape browsers.)

To restore the error dialog boxes, perform a soft or hard reload of the document. Clicking on the Reload button turns them back on.

You can, however, assign a custom function to the `window.onerror` property. This function then handles errors in a more friendly way under your script control. I prefer an even simpler way: Let a global `onerror()` function do the job. Whenever error dialog boxes are turned on (the default behavior), a script error (or Java applet or class exception) invokes the `onerror()` function, passing three parameters:

- Error message
- URL of document causing the error
- Line number of the error

You can essentially trap for all errors and handle them with your own interface (or no user alert dialog box at all). The last line of this function must be `return true` if you do not want the JavaScript script error dialog box to appear.

If you are using LiveConnect to communicate with a Java applet or call up Java class methods directly from your scripts, you can use an `onerror()` function to handle any *exception* that Java may throw. A Java exception is not necessarily a mistake kind of error: some methods assume that the Java code will trap for exceptions to handle special cases (for example, reacting to a user's denial of access when prompted by a signed script dialog). See Chapter 40 for an example of trapping for a specific Java exception via an `onerror()` function.

Example

In Listing 14-9, one button triggers a script that contains an error. I've added an `onerror()` function to process the error so it opens a separate window, filling in a textarea form element (see Figure 14-6). A Submit button is also provided to mail the bug information to a support center e-mail address from Navigator only — an example of how to handle the occurrence of a bug in your scripts. In case you have not yet seen a true JavaScript error dialog box, change the last line of the `onerror()` function to `return false`, then reload the document and trip the error.

Listing 14-9: **Controlling Script Errors**

```
<HTML>
<HEAD>
<TITLE>Error Dialog Control</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
// function with invalid variable value
function goWrong() {
        var x = fred
}
// turn off error dialogs
function errOff() {
        window.onerror = null
}
// turn on error dialogs with hard reload
function errOn() {
        location.reload()
}
function onerror(msg, URL, lineNum) {
        var errWind = window.open("","errors","HEIGHT=270,WIDTH=400")
        var wintxt = "<HTML><BODY BGCOLOR=RED>"
        wintxt += "<B>An error has occurred on this page.  Please
report it to Tech Support.</B>"
        wintxt += "<FORM METHOD=POST ENCTYPE-'text-plain'
ACTION=mailTo:support3@dannyg.com>"
        wintxt += "<TEXTAREA COLS=45 ROWS=8 WRAP=VIRTUAL>"
        wintxt += "Error: " + msg + "\n"
        wintxt += "URL: " + URL + "\n"
        wintxt += "Line: " + lineNum + "\n"
        wintxt += "Client: " + navigator.userAgent + "\n"
        wintxt += "----------------------------------------\n"
        wintxt += "Please describe what you were doing when the error
occurred:"
        wintxt += "</TEXTAREA><P>"
        wintxt += "<INPUT TYPE=SUBMIT VALUE='Send Error Report'>"
        wintxt += "<INPUT TYPE=button VALUE='Close'
onClick='self.close()'>"
        wintxt += "</FORM></BODY></HTML>"
        errWind.document.write(wintxt)
        errWind.document.close()
        return true
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="myform">
<INPUT TYPE="button" VALUE="Cause an Error" onClick="goWrong()"><P>
<INPUT TYPE="button" VALUE="Turn Off Error Dialogs" onClick="errOff()">
<INPUT TYPE="button" VALUE="Turn On Error Dialogs" onClick="errOn()">
</FORM>
</BODY>
</HTML>
```

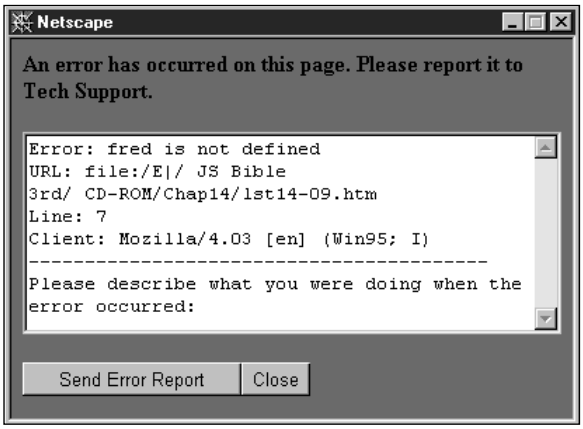

Figure 14-6: An example of a self-reporting error window

Turn off the dialog box by setting the `window.onerror` property to null. I provide a button that performs a hard reload, which, in turn, resets the `window.onerror` property to its default value. With error dialog boxes turned off, the `onerror()` function does not fire.

Related Items: `location.reload()` method; debugging scripts (Chapter 45).

opener

Value: Window object **Gettable:** Yes **Settable:** No

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

Many scripters make the mistake of thinking that a new browser window created with the `window.open()` method has a child-parent relationship similar to the one that frames have with their parents. That's not the case at all. New browser windows, once created, have a very slim link to the window from whence they came: via the `opener` property. The purpose of the `opener` property is to provide scripts in the new window with valid references back to the original window. For example, the original window may contain some variable values or general-purpose functions that a new window at this Web site will want to use. The original window may also have form elements whose settings either are of value to the new window or get set by user interaction in the new window.

Because the value of the `opener` property is a true window object, you can begin references with the property name. Or, you may use the more complete `window.opener` or `self.opener` reference. But the reference must then include some object or property of that original window, such as a window method or a reference to something contained by that window's document.

Although this property was new for Navigator 3 (and was one of the rare Navigator 3 features to be included in Internet Explorer 3), you can make your scripts backward compatible to Navigator 2. For every new window you create, make sure it has an `opener` property as follows:

```
var newWind = window.open()
if (newWind.opener == null) {
        newWind.opener = self
}
```

For Navigator 2, this step adds the `opener` property to the window object reference. Then, no matter which version of JavaScript-enabled Navigator the user has, the `opener` property in the new window's scripts points to the desired original window.

When a script that generates a new window is within a frame, the `opener` property of the subwindow points to that frame. Therefore, if the subwindow needs to communicate with the main window's parent or another frame in the main window, you have to very carefully build a reference to that distant object. For example, if the subwindow needs to get the `checked` property of a checkbox in a sister frame of the one that created the subwindow, the reference would be

```
opener.parent.sisterFrameName.document.formName.checkboxName.checked
```

It is a long way to go, indeed, but building such a reference is always a case of mapping out the path from where the script is to where the destination is, step by step.

Example

To demonstrate the importance of the `opener` property, let's take a look at how a new window can define itself from settings in the main window (Listing 14-10). The `doNew()` function generates a small subwindow and loads the file in Listing 14-11 into the window. Notice the initial conditional statements in `doNew()` to make sure that if the new window already exists, it comes to the front by invoking the new window's `focus()` method. You can see the results in Figure 14-7. Because the `doNew()` function in Listing 14-10 uses window methods and properties not available in Internet Explorer 3, this example does not work correctly in Internet Explorer 3.

Listing 14-10: Contents of a Main Window Document That Generates a Second Window

```
<HTML>
<HEAD>
<TITLE>Master of all Windows</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
var myWind
function doNew() {
        if (!myWind || myWind.closed) {
           myWind = window.open("lst14-
11.htm","subWindow","HEIGHT=200,WIDTH=350")
        } else{
```

```
            // bring existing subwindow to the front
            myWind.focus()
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="input">
Select a color for a new window:
<INPUT TYPE="radio" NAME="color" VALUE="red" CHECKED>Red
<INPUT TYPE="radio" NAME="color" VALUE="yellow">Yellow
<INPUT TYPE="radio" NAME="color" VALUE="blue">Blue
<INPUT TYPE="button" NAME="storage" VALUE="Make a Window"
onClick="doNew()">
<HR>
This field will be filled from an entry in another window:
<INPUT TYPE="text" NAME="entry" SIZE=25>
</FORM>
</BODY>
</HTML>
```

The `window.open()` method doesn't provide parameters for setting the new window's background color, so I let the `getColor()` function in the new window do the job as the document loads. The function uses the `opener` property to find out which radio button on the main page is selected.

Listing 14-11: **References to the opener Property**

```
<HTML>
<HEAD>
<TITLE>New Window on the Block</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function getColor() {
        // shorten the reference
        colorButtons = self.opener.document.forms[0].color
        // see which radio button is checked
        for (var i = 0; i < colorButtons.length; i++) {
            if (colorButtons[i].checked) {
                return colorButtons[i].value
            }
        }
        return "white"
}
</SCRIPT>
</HEAD>
<SCRIPT LANGUAGE="JavaScript">
document.write("<BODY BGCOLOR='" + getColor() + "'>")
</SCRIPT>
<H1>This is a new window.</H1>
<FORM>
```

(continued)

Listing 14-11 *Continued*

```
<INPUT TYPE="button" VALUE="Who's in the Main window?"
onClick="alert(self.opener.document.title)"><P>
Type text here for the main window:
<INPUT TYPE="text" SIZE=25
onChange="self.opener.document.forms[0].entry.value = this.value">
</FORM>
</BODY>
</HTML>
```

In the `getColor()` function, the multiple references to the radio button array would be very long. To simplify the references, the `getColor()` function starts out by assigning the radio button array to a variable I've arbitrarily called `colorButtons`. That shorthand now stands in for lengthy references as I loop through the radio buttons to determine which button is checked and retrieve its `value` property.

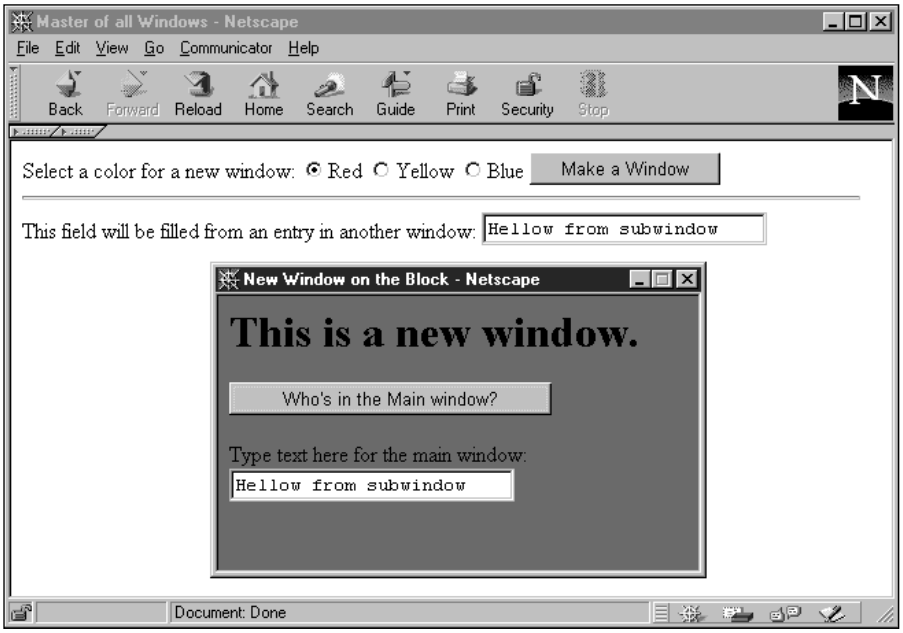

Figure 14-7: The main and subwindows, inextricably linked via the window.opener property

A button in the second window simply fetches the title of the opener window's document. Even if another document loads in the main window in the meantime, the `opener` reference still points to the main window: Its document object, however, will change.

Finally, the second window contains a text object. Enter any text you like there and either tab or click out of the field. The `onChange=` event handler updates the field in the opener's document (provided that document is still loaded).

Related Items: `window.open()` method; `window.focus()` method.

outerHeight
outerWidth

See innerHeight and innerWidth, earlier.

pageXOffset
pageYOffset

Value: Integer **Gettable:** Yes **Settable:** No

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

The top-left corner of the content (inner) region of the browser window is an important geographical point for scrolling documents. When a document is scrolled all the way to the top and flush left in the window (or when a document is small enough to fill the browser window without displaying scrollbars), the document's location is said to be 0,0, meaning zero pixels from the top and zero pixels from the left. If you were to scroll the document, some other coordinate point of the document would be under that top-left corner. That measure is called the *page offset*, and the `pageXOffset` and `pageYOffset` properties let you read the pixel value of the document at the inner window's top-left corner: `pageXOffset` is the horizontal offset, and `pageYOffset` is the vertical offset.

The value of these measures becomes clear if you design navigation buttons in your pages to carefully control paging of content being displayed in the window. For example, you might have a two-frame page in which one of the frames features navigation controls, while the other displays the primary content. The navigation controls take the place of scrollbars, which, for aesthetic reasons, are turned off in the display frame. Scripts connected to the simulated scrolling buttons can determine the `pageYOffset` value of the document and then use the `window.scrollTo()` method to position the document precisely to the next logical division in the document for viewing.

Example

The script in Listing 14-12 is an unusual construction that creates a frameset and creates the content for each of the two frames all within a single HTML document (see "Frame Object" later in this chapter for more details). The purpose of this example is to provide you with a playground to get familiar with the page

offset concept and how the values of these properties correspond to physical activity in a scrollable document.

In the left frame of the frameset are two fields that are ready to show the pixel values of the right frame's `pageXOffset` and `pageYOffset` properties. The content of the right frame is a 30-row table of fixed width (800 pixels). Mouse click events are captured by the document level (see Chapter 39), allowing you to click any table or cell border or outside the table to trigger the `showOffsets()` function in the right frame. That function is a simple script that displays the page offset values in their respective fields in the left frame.

Listing 14-12: **Viewing the page XOffset and page YOffset Properties**

```
<HTML>
<HEAD>
<TITLE>Master of all Windows</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function leftFrame() {
        var output = "<HTML><BODY><H3>Page Offset Values</H3><HR>\n"
        output += "<FORM>PageXOffset:<INPUT TYPE='text' NAME='xOffset'
SIZE=4><BR>\n"
        output += "PageYOffset:<INPUT TYPE='text' NAME='yOffset'
SIZE=4><BR>\n"
        output += "</FORM></BODY></HTML>"
        return output
}

function rightFrame() {
        var output = "<HTML><HEAD><SCRIPT LANGUAGE='JavaScript'>\n"
        output += "function showOffsets() {\n"
        output += "parent.readout.document.forms[0].xOffset.value =
self.pageXOffset\n"
        output += "parent.readout.document.forms[0].yOffset.value =
self.pageYOffset\n}\n"
        output += "document.captureEvents(Event.CLICK)\n"
        output += "document.onclick = showOffsets\n"
        output += "<\/SCRIPT></HEAD><BODY
onClick='showOffsets()'><H3>Content Page</H3>\n"
        output += "Scroll the page and click on a table border to view
page offset values.<BR><HR>\n"
        output += "<TABLE BORDER=5 WIDTH=800>"
        var oneRow = "<TD>Cell 1</TD><TD>Cell 2</TD><TD>Cell
3</TD><TD>Cell 4</TD><TD>Cell 5</TD>"
        for (var i = 1; i <= 30; i++) {
            output += "<TR><TD><B>Row " + i + "</B></TD>" + oneRow +
"</TR>"
        }
        output += "</TABLE></BODY></HTML>"
        return output
}
</SCRIPT>
```

```
</HEAD>
<FRAMESET COLS="30%,70%">
        <FRAME NAME="readout" SRC="javascript:parent.leftFrame()">
        <FRAME NAME="display" SRC="javascript:parent.rightFrame()">
</FRAMESET>
</HTML>
```

To gain an understanding of how the offset values work, scroll the window slightly in the horizontal direction and notice that the `pageXOffset` value increases; the same goes for the `pageYOffset` value as you scroll down. Remember that these values reflect the coordinate in the document that is currently under the top-left corner of the window (frame) holding the document.

Related Items: `window.innerHeight` property; `window.innerWidth` property; `window.scrollBy()` method; `window.scrollTo()` method.

parent

Value: Window object **Gettable:** Yes **Settable:** No

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

The `parent` property (and the `top` property that follows) comes into play primarily when a document is to be displayed as part of a multiframe window. The HTML documents that users see in the frames of a multiframe browser window are distinct from the document that specifies the frameset for the entire window. That document, though still in the browser's memory (and appearing as the URL in the location field of the browser), is not otherwise visible to the user (except in the Document Source view).

If scripts in your visible documents need to reference objects or properties of the frameset window, you can reference those frameset window items with the `parent` property (do not, however, expand the reference by preceding it with the window object, as in `window.parent.propertyName`). In a way, the `parent` property seems to violate the object hierarchy because, from a single frame's document, the property points to a level seemingly higher in precedence. If you didn't specify the `parent` property or instead specified the `self` property from one of these framed documents, the object reference is to the frame only, rather than to the outermost framesetting window object.

A nontraditional but perfectly legal way to use the `parent` object is as a means of storing temporary variables. Thus, you could set up a holding area for individual variable values or even an array of data. These values can then be shared among all documents loaded into the frames, including when documents change inside the frames. You have to be careful, however, when storing data in the parent on the fly (that is in response to user action in the frames). Variables can revert to their

default values (that is, the values set by the parent's own script) if the user resizes the browser window.

A child window can also call a function defined in the parent window. The reference for such a function is

```
parent.functionName([parameters])
```

At first glance, it may seem as though the `parent` and `top` properties point to the same framesetting window object. In an environment consisting of one frameset window and its immediate children, that's true. But if one of the child windows was, itself, another framesetting window, then you wind up with three generations of windows. From the point of view of the "youngest" child (for example, a window defined by the second frameset), the `parent` property points to its immediate parent, whereas the `top` property points to the first framesetting window in this chain.

On the other hand, a new window created via the `window.open()` method has no parent-child relationship to the original window. The new window's `top` and `parent` point to that new window. You can read more about these relationships in the "Frames" section earlier in this chapter.

Example

To demonstrate how various `window object` properties refer to window levels in a multiframe environment, use your browser to load the Listing 14-13 document. It, in turn, sets each of two equal-size frames to the same document: Listing 14-14. This document extracts the values of several window properties, plus the `document.title` properties of two different window references.

Listing 14-13: **Framesetting Document for Listing 14-14**

```
<HTML>
<HEAD>
<TITLE>The Parent Property Example</TITLE>
<SCRIPT LANGUAGE="JavaScript">
self.name = "Framesetter"
</SCRIPT>
</HEAD>
<FRAMESET COLS="50%,50%" onUnload="self.name = ''">
        <FRAME NAME="JustAKid1" SRC="lst14-14.htm">
        <FRAME NAME="JustAKid2" SRC="lst14-14.htm">
</FRAMESET>
</HTML>
```

Listing 14-14: **Revealing Various Window-Related Properties**

```
<HTML>
<HEAD>
<TITLE>Window Revealer II</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function gatherWindowData() {
        var msg = ""
```

```
        msg = msg + "top name: " + top.name + "<BR>"
        msg = msg + "parent name: " + parent.name + "<BR>"
        msg = msg + "parent.document.title: " + parent.document.title +
"<P>"
        msg = msg + "window name: " + window.name + "<BR>"
        msg = msg + "self name: " + self.name + "<BR>"
        msg = msg + "self.document.title: " + self.document.title
        return msg
}
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
document.write(gatherWindowData())
</SCRIPT>
</BODY>
</HTML>
```

In the two frames (Figure 14-8), the references to the `window` and `self` object names return the name assigned to the frame by the frameset definition (`JustAKid1` for the top frame, `JustAKid2` for the bottom frame). In other words, from each frame's point of view, the window object is its own frame. References to `self.document.x` refer only to the document loaded into that window frame. But references to the top and parent windows (which are one and the same in this example) show that those `object` properties are shared among both frames.

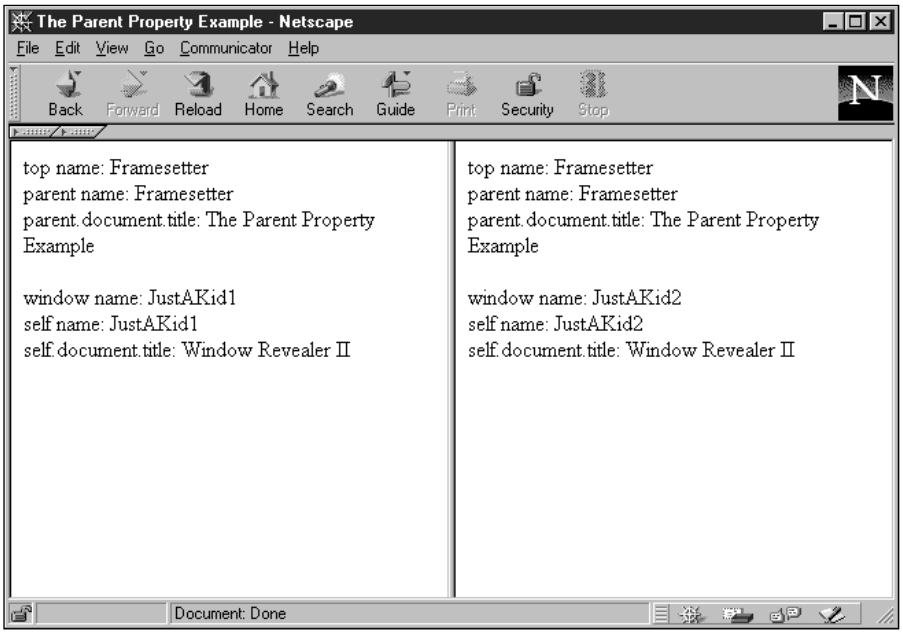

Figure 14-8: After the document in Listing 14-14 loads into the two frames established by Listing 14-13, parent and top properties are shared by both frames.

A couple other fine points are worth highlighting. First, the name of the framesetting window is set as the Listing 14-13 loads, rather than in response to an `onLoad=` event handler in the `<FRAMESET>` tag. The reason for this is that the name must be set in time for the documents loading in the frames to get that value. If I had waited until the frameset's `onLoad=` event handler, the name wouldn't be set until *after* the frame documents had loaded. Second, I restore the parent window's name to an empty string when the framesetting document unloads. This is to prevent future pages from getting confused about the window name.

Related Items: `window.frames` property; `window.self` property; `window.top` property.

personalbar
scrollbar

See locationbar.

self

Value: Window object **Gettable:** Yes **Settable:** No

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

Just as the window object reference is optional, so too is the `self` property when the object reference points to the same window as the one containing the reference. In what may seem to be an unusual construction, the `self` property represents the same object as the window. For instance, to obtain the title of the document in a single-frame window, you can use any of the following three constructions:

```
window.document.title
self.document.title
document.title
```

Although `self` is a property of a window, you should not combine the references within a single-frame window script (for example, don't begin a reference with `window.self`). Specifying the `self` property, though optional for single-frame windows, can help make an object reference crystal clear to someone reading your code (and to you, for that matter). Multiple-frame windows are where you need to pay particular attention to this property.

JavaScript is pretty smart about references to a statement's own window. Therefore, you can generally omit the `self` part of a reference to a same-window document element. But when you intend to display a document in a multiframe window, complete references (including the `self` prefix) to an object make it much easier on anyone who reads or debugs your code to track who is doing what to whom. You are free to retrieve the `self` property of any window. The value that

comes back is an entire window object — a copy of all data that makes up the window (including properties and methods).

Example

Listing 14-15 uses the same operations as Listing 14-4, but substitutes the `self` property for all window object references. The application of this reference is entirely optional, but it can be helpful for reading and debugging scripts if the HTML document is to appear in one frame of a multiframe window — especially if other JavaScript code in this document refers to documents in other frames. The `self` reference helps anyone reading the code know precisely which frame was being addressed.

Listing 14-15: Using the self Property

```
<HTML>
<HEAD>
<TITLE>self Property</TITLE>
<SCRIPT LANGUAGE="JavaScript">
self.defaultStatus = "Welcome to my Web site."
</SCRIPT>
</HEAD>
<BODY>
<A HREF="http://home.netscape.com" onMouseOver="self.status = 'Go to
your browser Home page.';return true" onMouseOut="self.status =
'';return true">Home</A><P>
<A HREF="http://home.netscape.com" onMouseOver="self.status = 'Visit
Netscape\'s Home page.';return true" onMouseOut="self.status =
self.defaultStatus;return true">Netscape</A>
</BODY>
</HTML>
```

Related Items: `window.frames` property; `window.parent` property; `window.top` property.

status

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

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

At the bottom of the browser window is a statusbar. Part of that bar includes an area that normally discloses the document loading progress or the URL of a link that the mouse is pointing to at any given instant. You can control the temporary content of that field by assigning a text string to the window object's `status` property (Figure 14-9). You should adjust the `status` property only in response to

events that have a temporary effect, such as a link or image map area object's `onMouseOver=` event handler. When the `status` property is set in this situation, it overrides any other setting in the statusbar. If the user then moves the mouse pointer away from the object that changes the statusbar, the bar returns to its default setting (which may be empty on some pages).

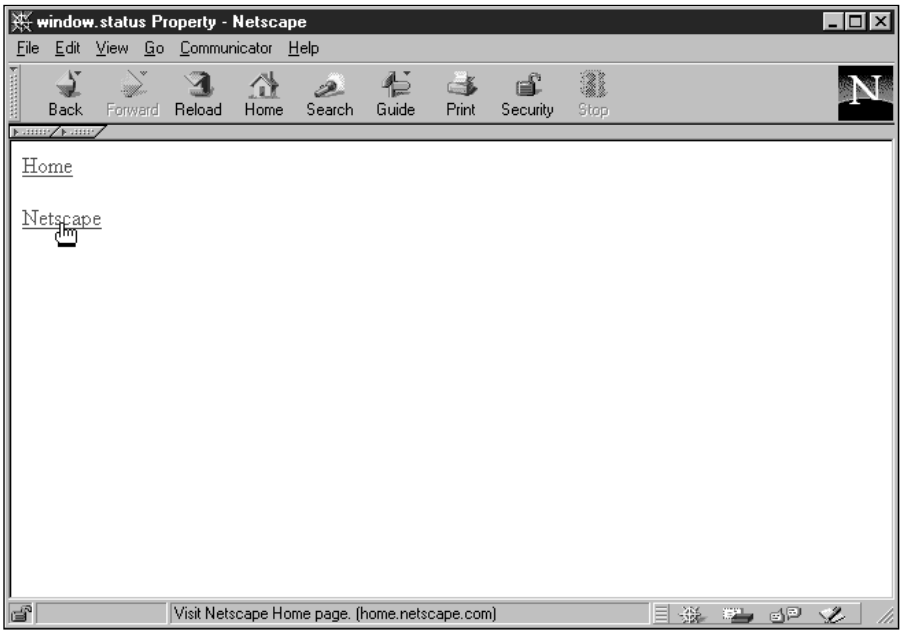

Figure 14-9: The statusbar can be set to display a custom message when the pointer rolls over a link.

Use this window property as a friendlier alternative to displaying the URL of a link as a user rolls the mouse around the page. For example, if you'd rather use the statusbar to explain the nature of the destination of a link, put that text into the statusbar in response to the `onMouseOver=` event handler. But be aware that experienced Web surfers like to see URLs down there. Therefore, consider creating a hybrid message for the statusbar that includes both a friendly description followed by the URL in parentheses. In multiframe environments, you can set the `window.status` property without having to worry about referencing the individual frame.

Example

In Listing 14-16, the `status` property is set in a handler embedded in the `onMouseOver=` attribute of two HTML link tags. Notice that the handler requires a `return true` statement (or any expression that evaluates to `return true`) as the last statement of the handler. This statement is required or the status message will not display.

Listing 14-16: **Links with Custom Statusbar Messages**

```
<HTML>
<HEAD>
<TITLE>window.status Property</TITLE>
</HEAD>
<BODY>
<A HREF="http://www.dannyg.com" onMouseOver="window.status = 'Go to my
Home page. (www.dannyg.com)'; return true">Home</A><P>
<A HREF="http://home.netscape.com" onMouseOver="window.status = 'Visit
Netscape Home page. (home.netscape.com)'; return true">Netscape</A>
</BODY>
</HTML>
```

As a safeguard against platform-specific anomalies that affect the behavior of `onMouseOver=` event handlers and the `window.status` property, you should also include an `onMouseOut=` event handler for links and client-side image map area objects. Such `onMouseOut=` event handlers should set the `status` property to an empty string. This setting ensures that the statusbar message returns to the `defaultStatus` setting when the pointer rolls away from these objects. If you want to write a generalizable function that handles all window status changes, you can do so, but word the `onMouseOver=` attribute carefully so that the event handler evaluates to `return true`. Listing 14-17 shows such an alternative.

Listing 14-17: **Handling Status Message Changes**

```
<HTML>
<HEAD>
<TITLE>Generalizable window.status Property</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function showStatus(msg) {
        window.status = msg
        return true
}
</SCRIPT>
</HEAD>
<BODY>
<A HREF="http://home.netscape.com" onMouseOver="return showStatus('Go
to my Home page.')" onMouseOut="return showStatus('')">Home</A><P>
<A HREF="http://home.netscape.com" onMouseOver="return
showStatus('Visit Netscape Home page.')" onMouseOut="return
showStatus('')">Netscape</A>
</BODY>
</HTML>
```

Notice how the event handlers return the results of the `showStatus()` method to the event handler, allowing the entire handler to evaluate to `return true`.

One final example of setting the statusbar (shown in Listing 14-18) also demonstrates how to create a scrolling banner in the statusbar.

Listing 14-18: **Creating a Scrolling Banner**

```
<HTML>
<HEAD>
<TITLE>Message Scroller</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
var msg = "Welcome to my world..."
var delay = 150
var timerId
var maxCount = 0
var currCount = 1

function scrollMsg() {
        // set the number of times scrolling message is to run
        if (maxCount == 0) {
           maxCount = 3 * msg.length
        }
        window.status = msg
        // keep track of how many characters have scrolled
        currCount++
        // shift first character of msg to end of msg
        msg = msg.substring (1, msg.length) + msg.substring (0, 1)
        // test whether we've reached maximum character count
        if (currCount >= maxCount) {
           timerID = 0          // zero out the timer
           window.status = ""   // clear the status bar
           return               // break out of function
        } else {
           // recursive call to this function
           timerId = setTimeout("scrollMsg()", delay)
        }
}
// -->
</SCRIPT>
</HEAD>
<BODY onLoad="scrollMsg()">
</BODY>
</HTML>
```

Because the statusbar is being set by a standalone function (rather than by an `onMouseOver=` event handler), you do not have to append a `return true` statement to set the `status` property. The `scrollMsg()` function uses more advanced JavaScript concepts, such as the `window.setTimeout()` method (covered later in this chapter) and string methods (covered in Chapter 27). To speed the pace at which the words scroll across the statusbar, reduce the value of `delay`.

Many Web surfers (myself included) don't care for these scrollers that run forever in the statusbar. They can also crash earlier browsers, because the `setTimeout()` method eats application memory in Navigator 2. Use scrolling bars sparingly or design them to run only a few times after the document loads.

Setting the status property with `onMouseOver=` event handlers has had a checkered career along various implementations in Navigator. A script that sets the statusbar is always in competition against the browser itself, which uses the statusbar to report loading progress. Bugs also prevent the bar from clearing itself, even when an `onMouseOut=` event handler sets it to an empty string. The situation improves with each new browser version, but be prepared for anomalies among visitors using older scriptable browsers.

Related Items: `window.defaultStatus` property; `onMouseOver=` event handler; `onMouseOut=` event handler; link object.

statusbar
toolbar

See locationbar.

top

Value: Window object **Gettable:** Yes **Settable:** No

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

The window object's `top` property refers to the topmost window in the JavaScript hierarchy. For a single-frame window, the reference is to the same object as the window itself (including the `self` and `parent` property), so do not include `window` as part of the reference. In a multiframe window, the top window is the one that defines the first frameset (in case of nested framesets). Users don't ever really see the top window in a multiframe environment, but the browser stores it as an object in its memory. The reason is that the top window has the road map to the other frames (if one frame should need to reference an object in a different frame), and its children frames can call upon it. Such a reference looks like

```
top.functionName([parameters])
```

For more about the distinction between the `top` and `parent` properties, see the in-depth discussion about scripting frames at the beginning of this chapter. See also the example of the `parent` property for listings that demonstrate the values of the `top` property.

Related Items: `window.frames` property; `window.self` property; `window.parent` property.

window

Value: Window object **Gettable:** Yes **Settable:** No

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

Listing the `window` property as a separate property may be more confusing than helpful. The `window` property is the same object as the window object. You do not need to use a reference that begins with `window.window`. Although the window object is assumed for many references, you can use `window` as part of a reference to items in the same window or frame as the script statement that makes that reference. You should not, however, use window as a part of a reference involving items higher up in the hierarchy (`top` or `parent`).

Methods

alert(*message*)

Returns: Nothing.

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

An alert dialog box is a modal window that presents a message to the user with a single OK button to dismiss the dialog box. As long as the alert dialog box is showing, no other application or window can be made active. The user must dismiss the dialog box before proceeding with any more work in the browser or on the computer.

The single parameter to the `alert()` method can be a value of any data type, including representations of some unusual data types whose values you don't normally work with in JavaScript (such as complete objects). This makes the alert dialog box a handy tool for debugging JavaScript scripts. Anytime you want to monitor the value of an expression, use that expression as the parameter to a temporary `alert()` method in your code. The script proceeds to that point and then stops to show you the value. (See Chapter 45 for more tips on debugging scripts.)

What is often disturbing to application designers is that all JavaScript-created modal dialog boxes (via the `alert()`, `confirm()`, and `prompt()` methods) identify themselves as being generated by JavaScript or the browser (Internet Explorer 4). The look is particularly annoying in browsers before Navigator 4 and Internet Explorer 4, because the wording appears directly in the dialog box's content area, rather than in the title bar of the dialog box. The purpose of this identification is to

act as a security precaution against unscrupulous scripters who might try to spoof system or browser alert dialog boxes inviting a user to reveal passwords or other private information. These identifying words cannot be overwritten or eliminated by your scripts. If you want more control over a window, generate a separate browser window with `window.open()`. Unless you use signed scripts to create an always raised window, that new window will not be a modal dialog box and could get hidden behind a larger window. Syntax for Internet Explorer 4's special dialog-style window is not part of Navigator 4.

Because the `alert()` method is of a global nature (that is, no particular frame in a multiframe environment derives any benefit from laying claim to the alert dialog box), a common practice is to omit all window object references from the statement that calls the method. Restrict the use of alert dialog boxes in your HTML documents and site designs. The modality of the windows is disruptive to the flow of a user's navigation around your pages. Communicate with users via forms or by writing to separate document window frames.

Example

The parameter for the example in Listing 14-19 is a concatenated string. It joins together two fixed strings and the value of the browser's `appName` property. Loading this document causes the alert dialog box to appear, as shown in several configurations in Figure 14-10. The `JavaScript Alert:` line cannot be deleted from the dialog box in earlier browsers, nor can the title bar be changed in Navigator 4 or Internet Explorer 4.

Listing 14-19: **Displaying an Alert Dialog Box**

```
<HTML>
<HEAD>
<TITLE>window.alert() Method</TITLE>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
alert("You are running the " + navigator.appName + " browser.")
</SCRIPT>
</BODY>
</HTML>
```

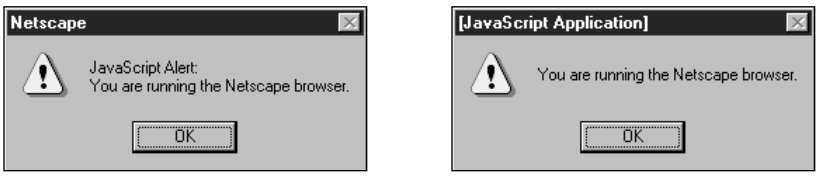

Figure 14-10: Results of the alert() method in Listing 14-19 in Navigator 3 and Navigator 4 for Windows 95

Related Items: `window.confirm()` method; `window.prompt()` method.

back()

Returns: Nothing.

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

The Back button's behavior has gone through transformations since Navigator 2. Some authors like the changes, others do not. In Navigator 2, the history object (and all navigation methods associated with it) assumed the entire browser window would change with a click of the Back or Forward button in the toolbar. With the increased popularity of frames, this mechanism didn't work well if one frame remained static while documents flew in and out of another frame: navigation had to be on a frame-by-frame basis, and that's how the Back and Forward buttons worked in Navigator 3 and now in Navigator 4.

From Navigator 3 onward, each window object (including frames) maintains its own history. Unfortunately, JavaScript doesn't observe this until you get to Navigator 4, and thus for a lot of browsers out there (including Internet Explorer 3 and Internet Explorer 4), the history navigation methods control the global history. The purpose of the `window.back()` method is to offer a scripted version of the global back and forward navigation buttons, while allowing the history object to control navigation strictly within a particular window or frame — as it should. For more information about version compatibility and the back and forward navigation, see the history object in Chapter 15.

Example

Listing 14-20 is a framesetting document for a `back()` and `forward()` method laboratory to help you understand the differences between window and history navigation. All the work is done in the document shown in Listing 14-21.

Listing 14-20: **Navigation Lab Frameset**

```
<HTML>
<HEAD>
<TITLE>Back and Forward</TITLE>
</HEAD>
<FRAMESET COLS="45%,55%">
        <FRAME NAME="controller" SRC="lst14-21.htm">
        <FRAME NAME="display" SRC="lst14-03.htm">
</FRAMESET>
</HTML>
```

The top portion of Listing 14-21 contains simple links to other example files from this chapter. A click on any link loads a different document into the right-hand frame to let you build some history inside the frame.

Listing 14-21: **Navigation Lab Control Panel**

```
<HTML>
<HEAD>
<TITLE>Lab Controls</TITLE>
</HEAD>
<BODY>
<B>Load a series of documents into the right frame by clicking some of
these links (make a note of the sequence you click on):</B><P>
<A HREF="lst14-04.htm" TARGET="display">Listing 14-4</A><BR>
<A HREF="lst14-05.htm" TARGET="display">Listing 14-5</A><BR>
<A HREF="lst14-09.htm" TARGET="display">Listing 14-9</A><BR>
<A HREF="lst14-10.htm" TARGET="display">Listing 14-10</A><BR>
<HR>
<FORM NAME="input">
<B>Click on the various buttons below to see the results in this
frameset:</B><P>
<UL>
<LI>Substitute for toolbar buttons -- <TT>window.back()</TT> and
<TT>window.forward()</TT>:<INPUT TYPE="button" VALUE="Back"
onClick="window.back()"><INPUT TYPE="button" VALUE="Forward"
onClick="window.forward()"><P>

<LI><TT>history.back()</TT> and <TT>history.forward()</TT> for
righthand frame:<INPUT TYPE="button" VALUE="Back"
onClick="parent.display.history.back()"><INPUT TYPE="button"
VALUE="Forward" onClick="parent.display.history.forward()"><P>

<LI><TT>history.back()</TT> for this frame:<INPUT TYPE="button"
VALUE="Back" onClick="history.back()"><P>
</UL>
</FORM>
</BODY>
</HTML>
```

At the bottom are three sets of navigation buttons. All scripting is performed directly in the button event handlers. The button first pair is tied to the `window.back()` and `window.forward()` methods. These work only in Navigator 4. The others are tied to histories of each frame. When you reach the end of a history list (by clicking either of the `history.back()` buttons), navigation ceases, because that frame is out of history. But the `window.back()` button is connected to the browser's global history (the one you see in the Go menu) and can keep going back until the entire frameset of Listing 14-20 is gone. The behavior of the history methods is different outside of Navigator 4.

Related Items: `window.forward()` method; `history.back()` method; `history.forward()` method; `history.go()` method.

blur()

Returns: Nothing.

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

The opposite of `window.focus()` is `window.blur()`, which pushes the referenced window to the back of all other open windows. If other Navigator windows, such as the Mail or News windows, are open, the window receiving the `blur()` method is placed behind these windows as well. As with the `window.focus()` method, make sure that your references signify the correct window. See Listing 14-30 for an example of `window.blur()` in action.

Related Items: `window.open()` method; `window.focus()` method; `window.opener` property.

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 and eventually reaches its intended target. For example, if you click a button, the click event first reaches the window object; then it goes to the document object; and 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 one of these outer containers 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 window object to intercept all events of a particular type, use the `window.captureEvents()` method to turn that facility on.

This 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 window to intercept all click events, the statement is

```
window.captureEvents(Event.CLICK)
```

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

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

Once an event type is captured by the window 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 popup-menu (as a separate layer) for a right-click. If the click comes from the left mouse button, the event is routed to its intended target.

To associate a function with a particular event type captured by a window 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:

```
window.onclick=doClickEvent
```

Note that the function name is assigned only as a reference name (no quotes or parentheses), 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 `window.releaseEvent()` method. See Chapter 33 for details of working with events in this manner.

Example

The page in Listing 14-22 is an exercise in capturing and releasing click events in the window layer. Whenever the window is capturing click events, the `flash()` function runs. In that function, the event is examined so that only if the Control key is also being held down and the name of the button starts with "button" does the document background color flash red. For all click events (that is, those directed at objects on the page capable of their own `onClick=` event handlers), the click is processed with the `routeEvent()` method to make sure the target buttons execute their own `onClick=` event handlers.

Listing 14-22: **Capturing Click Events in the Window**

```
<HTML>
<HEAD>
<TITLE>Window Event Capture</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
// function to run when window captures a click event
function flash(e) {
       if (e.modifiers == Event.CONTROL_MASK &&
e.target.name.indexOf("button") == 0) {
          document.bgColor = "red"
          setTimeout("document.bgColor = 'white'", 500)
       }
       // let event continue to target
       routeEvent(e)
}
// default setting to capture click events
```

(continued)

Listing 14-22 *Continued*

```
window.captureEvents(Event.CLICK)
// assign flash() function to click events captured by window
window.onclick = flash
</SCRIPT>
</HEAD>
<BODY BGCOLOR="white">
<FORM NAME="buttons">
<B>Turn window click event capture on or off (Default is "On")</B><P>
<INPUT NAME="captureOn" TYPE="button" VALUE="Capture On"
onClick="window.captureEvents(Event.CLICK)"> 
<INPUT NAME="captureOff" TYPE="button" VALUE="Capture Off"
onClick="window.releaseEvents(Event.CLICK)">
<HR>
<B>Ctrl+Click on a button to see if clicks are being captured by the
window (background color will flash red):</B><P>
<UL>
<LI><INPUT NAME="button1" TYPE="button" VALUE="Informix"
onClick="alert('You clicked on Informix.')">
<LI><INPUT NAME="button2" TYPE="button" VALUE="Oracle"
onClick="alert('You clicked on Oracle.')">
<LI><INPUT NAME="button3" TYPE="button" VALUE="Sybase"
onClick="alert('You clicked on Sybase.')">
</UL>
</FORM>
</BODY>
</HTML>
```

When you try this page, also turn off window event capture. Now only the buttons' `onClick=` event handlers execute, and the page does not flash red.

Related Items: `window.disableExternalCapture()` method; `window.enableExternalCapture()` method; `window.handleEvent()` method; `window.releaseEvents()` method; `window.routeEvent()` method.

`clearInterval(`*`intervalIDnumber`*`)`

Returns: Nothing.

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

Use the `window.clearInterval()` method to turn off an interval loop action started with the `window.setInterval()` method. The parameter is the ID number returned by the `setInterval()` method. A common application for the JavaScript interval mechanism is animation of an object on a page. If you have multiple

intervals running, each has its own ID value in memory. You can turn off any interval by its ID value. Once an interval loop stops, your script cannot resume that interval: It must start a new one, which will generate a new ID value.

Example

See Listings 14-43 and 14-44 for an example of how `setInterval()` and `clearInterval()` are used together on a page.

Related Items: `window.setInterval()` method; `window.setTimeout()` method; `window.clearTimeout()` method.

`clearTimeout(`*timeoutIDnumber*`)`

Returns: Nothing.

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

Use the `window.clearTimeout()` method in concert with the `window.setTimeout()` method, as described later in this chapter, when you want your script to cancel a timer that is waiting to run its expression. The parameter for this method is the ID number that the `window.setTimeout()` method returns when the timer starts ticking. The `clearTimeout()` method cancels the specified timeout. A good practice is to check your code for instances where user action may negate the need for a running timer — and to stop that timer before it goes off.

Example

The page in Listing 14-23 features one text field and two buttons (Figure 14-11). One button starts a count-down timer coded to last one minute (easily modifiable); the other button interrupts the timer at any time while it is running. When the minute is up, an alert dialog box lets you know.

Listing 14-23: **A Count-Down Timer**

```
<HTML>
<HEAD>
<TITLE>Count Down Timer</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
var running = false
var endTime = null
var timerID = null

function startTimer() {
        running = true
        now = new Date()
```

(continued)

Listing 14-13 *Continued*

```
        now = now.getTime()
        // change last multiple for the number of minutes
        endTime = now + (1000 * 60 * 1)
        showCountDown()
}

function showCountDown() {
        var now = new Date()
        now = now.getTime()
        if (endTime - now <= 0) {
            stopTimer()
            alert("Time is up.  Put down your pencils.")
        } else {
            var delta = new Date(endTime - now)
            var theMin = delta.getMinutes()
            var theSec = delta.getSeconds()
            var theTime = theMin
            theTime += ((theSec < 10) ? ":0" : ":") + theSec
            document.forms[0].timerDisplay.value = theTime
            if (running) {
                timerID = setTimeout("showCountDown()",1000)
            }
        }
}

function stopTimer() {
        clearTimeout(timerID)
        running = false
        document.forms[0].timerDisplay.value = "0:00"
}
//-->
</SCRIPT>
</HEAD>

<BODY>
<FORM>
<INPUT TYPE="button" NAME="startTime" VALUE="Start 1 min. Timer"
onClick="startTimer()">
<INPUT TYPE="button" NAME="clearTime" VALUE="Clear Timer"
onClick="stopTimer()"><P>
<INPUT TYPE="text" NAME="timerDisplay" VALUE="">
</FORM>
</BODY>
</HTML>
```

Notice that the script establishes three variables with global scope in the window: running, `endTime`, and `timerID`. These values are needed inside multiple functions, so they are initialized outside of the functions.

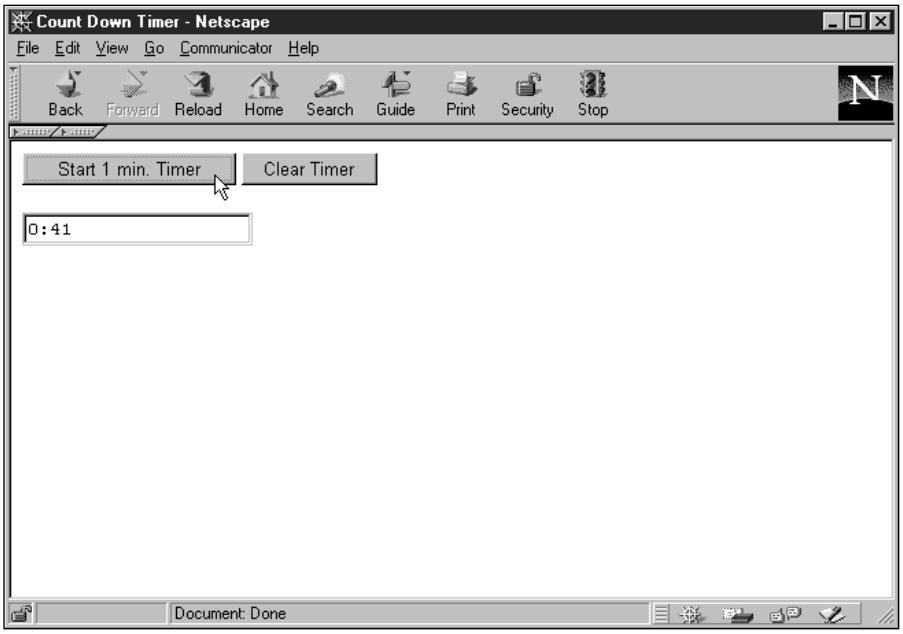

Figure 14-11: The count-down timer page as it displays the time remaining

In the `startTimer()` function, you switch the `running` flag on, meaning that the timer should be going. Using some date functions (Chapter 29), you extract the current time in milliseconds and add the number of milliseconds for the next minute (the extra multiplication by one is the place where you can change the amount to the desired number of minutes). With the end time stored in a global variable, the function now calls another function that compares the current and end times and displays the difference in the text field.

Early in the `showCountDown()` function, check to see if the timer has wound down. If so, you stop the timer and alert the user. Otherwise, the function continues to calculate the difference between the two times and formats the time in mm:ss format. As long as the `running` flag is set to true, the function sets the one-second timeout timer before repeating itself. To stop the timer before it has run out (in the `stopTimer()` function), the most important step is to cancel the timeout running inside the browser. The `clearTimeout()` method uses the global `timerID` value to do that. Then the function turns off the `running` switch and zeros out the display.

When you run the timer, you may occasionally notice that the time skips a second. It's not cheating. It just takes slightly more than one second to wait for the timeout and then finish the calculations for the next second's display. What you're seeing is the display catching up with the real time left.

Related Items: `window.setTimeout()`.

close()

Returns: Nothing.

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

The `window.close()` method closes the browser window referenced by the window object. Most likely, you will use this method to close subwindows created from a main document window. If the call to close the window comes from a window other than the new subwindow, the original window object *must* maintain a record of the subwindow object. You accomplish this by storing the value returned from the `window.open()` method in a global variable that will be available to other objects later (for example, a variable not initialized inside a function). If, on the other hand, an object inside the new subwindow calls the `window.close()` method, the `window` or `self` reference is sufficient.

Be sure to include a window as part of the reference to this method. Failure to do so causes JavaScript to regard the statement as a `document.close()` method, which has different behavior (see Chapter 16). Only the `window.close()` method can close the window via a script. Closing a window, of course, forces the window to trigger an `onUnload=` event handler before the window disappears from view; but once you've initiated the `window.close()` method, you cannot stop it from completing its task.

While I'm on the subject of closing windows, a special case exists when a subwindow tries to close the main window (via a statement such as `self.opener.close()`) when the main window has more than one entry in its session history. As a safety precaution against scripts closing windows they did not create, Navigator 3 and later ask the user whether he or she wants the main window to close (via a Navigator-generated JavaScript confirm dialog box). This security precaution cannot be overridden except in Navigator 4 via a signed script when the user grants permission to control the browser (Chapter 40).

Example

See Listing 14-3 (for the `window.closed` property), which provides an elaborate, cross-platform, bug-accommodating example of applying the `window.close()` method across multiple windows.

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

confirm(*message*)

Returns: True or false.

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

A confirm dialog box presents a message in a modal dialog box along with OK and Cancel buttons. Such a dialog box can be used to ask a question of the user, usually prior to a script performing actions that will not be undoable. Querying a user about proceeding with typical Web navigation in response to user interaction on a form element is generally a disruptive waste of the user's time and attention. But for operations that may reveal a user's identity or send form data to a server, a JavaScript confirm dialog box may make a great deal of sense. Users can also accidentally click buttons, so you should provide avenues for backing out of an operation before it executes.

Because this dialog box returns a Boolean value (OK = `true`; Cancel = `false`), you can use this method as a comparison expression or as an assignment expression. In a comparison expression, you nest the method within any other statement where a Boolean value is required. For example

```
if (confirm("Are you sure?")) {
        alert("OK")
} else {
        alert("Not OK")
}
```

Here, the returned value of the confirm dialog box provides the desired Boolean value type for the `if...else` construction (Chapter 31).

This method can also appear on the right side of an assignment expression, as in

```
var adult = confirm("You certify that you are over 18 years old?")
if (adult) {
        statements for adults
} else {
        statements for children
}
```

You cannot specify other alert icons or labels for the two buttons in JavaScript confirm dialog box windows.

Be careful how you word the question in the confirm dialog box. In Navigator 2 and 3, the buttons are labeled OK and Cancel in Windows browsers; the Mac versions, however, label the buttons Yes and No. If your visitors may be using older Mac Navigators, be sure your questions are logically answered with both sets of button labels. In Navigator 4, all platforms are the same (OK and Cancel).

Example

The example in Listing 14-24 shows the user interface part of how you can use a confirm dialog box to query a user before clearing a table full of user-entered data. The `JavaScript Application` line in the title bar, as shown in Figure 14-12, or the `JavaScript Confirm` legend in earlier browser versions cannot be removed from the dialog box.

Listing 14-24: **The Confirm Dialog Box**

```
<HTML>
<HEAD>
<TITLE>window.confirm() Method</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function clearTable() {
        if (confirm("Are you sure you want to empty the table?")) {
           alert("Emptying the table...") // for demo purposes
           //statements that actually empty the fields
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<!-- other statements that display and populate a large table -->
<INPUT TYPE="button" NAME="clear" VALUE="Reset Table"
onClick="clearTable()">
</FORM>
</BODY>
</HTML>
```

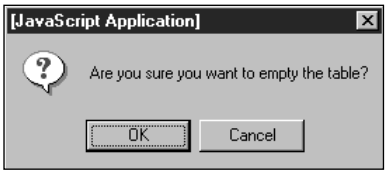

Figure 14-12: A JavaScript confirm dialog box (Navigator 4 Windows 95 format)

Related Items: `window.alert()`; `window.prompt()`; `form.submit()` method.

`disableExternalCapture()`

`enableExternalCapture()`

Returns: Nothing.

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

Security restrictions prevent one frame from monitoring events in another frame (when a different domain is in that second frame) unless the user has granted permission to a signed script. Controlling this cross-frame access

requires two special window object methods: `enableExternalCapture()` and `disableExternalCapture()`.

Putting these methods to work is a little trickier than manipulating the regular `window.captureEvents()` method. You have to turn on external capture in the frame doing the capture, but then set `captureEvents()` and the event handler in the frame whose events you want to capture. Moreover, when a new document loads into the second frame, you must set the `captureEvents()` and event handler for that frame again. See Chapter 40 for details about signed scripts.

Example

A framesetting document in Listing 14-25 loads two frames that let you experiment with both local and external event capture. You must either code sign the page or turn on codebase principals (see Chapter 40) to use this example. In the left frame is the control panel (Listing 14-26) for the laboratory. In the right frame I load Netscape's home page to provide a page with a different domain than your local disk or Web server. You can see what this frameset looks like in Figure 14-13.

Listing 14-25: **Frameset for Capture Laboratory**

```
<HTML>
<HEAD>
<TITLE>window.frames property</TITLE>
</HEAD>
<FRAMESET COLS="40%,60%">
        <FRAME NAME="controls" SRC="lst14-26.htm">
        <FRAME NAME="display" SRC="http://home.netscape.com">
</FRAMESET>
</HTML>
```

The control panel is an extension of the one used to demonstrate the `window.captureEvents()` method earlier in this chapter. In addition to the local window event capture, this new version adds a function that toggles external event capture on and off. To help differentiate the results of the local and external click event capture, the local capture flashes the control panel color in red; the external capture flashes in yellow.

Listing 14-26: **Control Panel for Capture Laboratory**

```
<HTML>
<HEAD>
<TITLE>Window Event Capture</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2" ARCHIVE="lst14-26.jar" ID="main">
// function to run when window captures a click event
function flashRed(e) {
        if (e.modifiers == Event.CONTROL_MASK &&
```

(continued)

Listing 14-26 *Continued*

```
e.target.name.indexOf("button") == 0) {
           document.bgColor = "red"
           setTimeout("document.bgColor = 'white'", 500)
       }
       // let event continue to target
       routeEvent(e)
}
function flashYellow(e) {
       if (e.target.href) {
           document.bgColor = "yellow"
           setTimeout("document.bgColor = 'white'", 500)
       }
       // let event continue to target
       routeEvent(e)
}
function setExternal(on) {
       if (on) {

netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
           window.enableExternalCapture()
           parent.display.captureEvents(Event.CLICK)
           parent.display.onclick=flashYellow
       } else {

netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
           window.disableExternalCapture()
       }
}

// default setting to capture click events
window.captureEvents(Event.CLICK)
// assign flash() function to click events captured by window
window.onclick = flashRed
</SCRIPT>
</HEAD>
<BODY BGCOLOR="white"
onUnload="netscape.security.PrivilegeManager.disablePrivilege('Universal
BrowserWrite')" ID="handler">
<FORM NAME="buttons">
<B>Turn window click event capture on or off (Default is "On")</B><P>
<INPUT NAME="captureOn" TYPE="button" VALUE="Capture On"
onClick="window.captureEvents(Event.CLICK)"> 
<INPUT NAME="captureOff" TYPE="button" VALUE="Capture Off"
onClick="window.releaseEvents(Event.CLICK)">
<HR>
<B>Turn window click event EXTERNAL capture on or off (Default is
"Off")</B><P>
<INPUT NAME="captureOn" TYPE="button" VALUE="External Capture On"
onClick="setExternal(true)"> 
```

```
<INPUT NAME="captureOff" TYPE="button" VALUE="External Capture Off"
onClick="setExternal(false)">
<HR>
<B>Ctrl+Click on a button to see if clicks are being captured by the
window (background color will flash red):</B><P>
<UL>
<LI><INPUT NAME="button1" TYPE="button" VALUE="Informix"
onClick="alert('You clicked on Informix.')">
<LI><INPUT NAME="button2" TYPE="button" VALUE="Oracle"
onClick="alert('You clicked on Oracle.')">
<LI><INPUT NAME="button3" TYPE="button" VALUE="Sybase"
onClick="alert('You clicked on Sybase.')">
</UL>
</FORM>
</BODY>
</HTML>
```

When the frameset initially loads, only the local window event capture is turned on, as before. The two statements at the end of the `<SCRIPT>` tag take care of that, including the one that directs all click events from the control panel window to the `flashRed()` function. In the `<SCRIPT>` tag, I put the attributes relating to signed scripts as a reminder that if this page were to be deployed on a server, the `<SCRIPT>` tag and every event handler in the document would require an ID attribute (see Chapter 40 for details). For the sake of readability, I have omitted the ID attributes for most of the event handlers, assuming that you will be trying this example with the help of codebase principals enabled.

The `setExternal()` function is a single function that toggles the external capture based on the Boolean value it receives as an argument. To turn on external capture, the script first invokes the Java method that requests `UniversalBrowserWrite` permission from the user (no error handling is built into this example to accommodate permission denial). Next, `enableExternalCapture()` is set for the control panel window: the one with the scripts that will be doing the processing of events from the second frame.

The remaining two statements are directed at the other frame. They engage event capturing over there for click events and direct those events to the `flashYellow()` function defined here in the control panel. JavaScript takes care of making the connections that force those external events to run a local function.

When you load this frameset and start clicking around, turn on the external capture, and click any link in the right frame. The control panel should flash yellow momentarily. If that link navigated to another page, you must turn on external capture again; but if the link navigated to an anchor on the same page, the next click will flash the yellow again. All the while, Ctrl+clicking on the lower three control panel buttons causes the background to flash red. Study the code carefully in Listing 14-26 and click around the laboratory frames to see how the two frames are handled.

Related Items: `window.captureEvents()` method; event object; signed scripts.

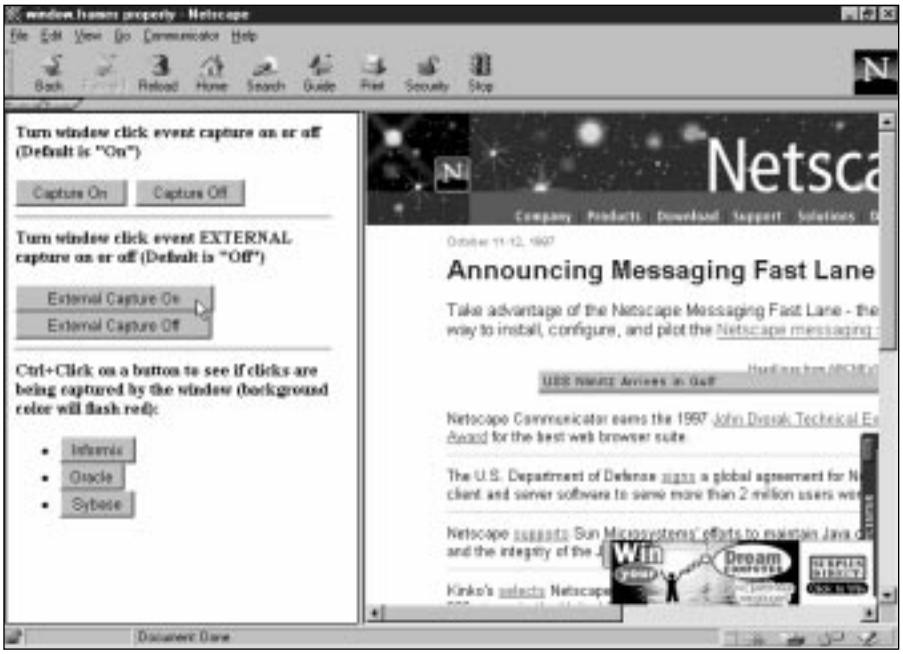

Figure 14-13: The local and external event capture laboratory

find(["*searchString*" [, *matchCaseBoolean*, *searchUpBoolean*]])

Returns: Boolean value for nondialog searches.

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

The `window.find()` method gives you a good amount of control over searching for a text string within a document contained by a window or frame. The action of this method mimics the powers of the browser's Find dialog box, accessible from the Find button in the toolbar.

The easiest way to deploy this command is without any parameters. This displays the browser's Find dialog box, just as if the user had clicked the Find button in the toolbar. If you want a search function available in a window lacking a toolbar, this is the way to go. With no parameters, this function does not return a value.

You can, however, go further to put the search facility more under script control. At the minimum, you can specify a search string as a parameter to the function. The search is based on simple string matching, and is not in any way connected with the regular expression kind of search (see Chapter 30). If the search finds a match,

the browser scrolls to that matching word, and highlights the word, just like using the browser's own Find dialog box. The function also returns a Boolean `true` when a match is found. This function does not allow you to bypass the scrolling and physical highlighting of the found string. If no match is found in the document or no more matches occur in the current search direction (the default direction is from top to bottom), the function returns `false`. This lets you control how the lack of a match is alerted to the user or what action the script should take.

Two optional parameters to the scripted find action let you specify whether the search should be case-sensitive and whether the search direction should be upward from the bottom of the document. These choices are identical to the ones that appear in the browser's Find dialog. Default behavior is case-insensitive searches from top to bottom. If you specify any one of these two optional parameters, you must specify both of them.

Internet Explorer 4 also has a text search facility, but it is implemented in an entirely different way (using its TextRange object and `findText()` method). The visual behavior also differs in that it does not highlight and scroll to a matching string in the text.

Example

Listing 14-27 is a framesetting document for an interface that permits experimentation with the `window.find()` method. The top frame is a control panel for searching in a copy of the Bill of Rights that appears in the bottom frame.

Listing 14-27: **Find() Method Frameset**

```
<HTML>
<HEAD>
<TITLE>window.find() method</TITLE>
</HEAD>
<FRAMESET ROWS="25%,75%">
        <FRAME NAME="controls" SRC="lst14-28.htm">
        <FRAME NAME="display" SRC="bofright.htm">
</FRAMESET>
</HTML>
```

All the action takes place in Listing 14-28, which is the control panel for text searches in the lower frame.

Listing 14-28: **Find() Method Control Panel**

```
<HTML>
<HEAD>
<TITLE>Window Event Capture</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
// function to run when window captures a click event
function findIt(form) {
        var matchString = form.searchTxt.value
```

(continued)

Listing 14-28 *Continued*

```
        var showDialog = form.dialog.checked
        var caseSensitive = form.sensitive.checked
        var backward = form.bkward.checked
        var wind = parent.display
        var success = true

        if (showDialog) {
           wind.find()
        } else {
           if (!matchString) {
              alert("Enter a search string in the field.")
           } else {
              success = wind.find(matchString,caseSensitive,backward)
           }
        }
        if (!success) {
           alert("No (more) matches found.")
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="controls">
<INPUT NAME="finder" TYPE="button" VALUE="Find..."
onClick="findIt(this.form)">
<INPUT NAME="searchTxt" TYPE="text"><BR>
<INPUT NAME="dialog" TYPE="checkbox"
onClick="this.form.sensitive.checked = false; this.form.bkward.checked
= false">Show Find dialog<BR>
<INPUT NAME="sensitive" TYPE="checkbox"
onClick="this.form.dialog.checked = false">Match case<BR>
<INPUT NAME="bkward" TYPE="checkbox" onClick="this.form.dialog.checked
= false">Search up<BR>
</FORM>
</BODY>
</HTML>
```

The control panel contains one text field for input of a string to search for. Three checkboxes let you set whether the browser's Find dialog should appear or the search should be entirely under script control with the two optional parameters. The Find button triggers the `findIt()` function, which assembles `find()` methods based on the checkbox settings and field input.

In the `find()` function, I first extract all the settings and assign them to individual variables. This is primarily for readability later in the function (eliminating all the long references in statements). Notice that the window being targeted for the search is the display frame, not the current window where the controls are.

To start playing with this example, enter the word "article" into the text box and click the Find button. Continue clicking as the highlighted instances of found text come into view. When you reach the end of the document, the function's alert dialog box tells you there are no more matches to be found (at least in the current direction). Activate the "Search up" checkbox and then start finding in the opposite direction. If you activate the "Match case" checkbox, no matches will be found, since the word "article" is in all uppercase letters in the document.

Related Items: None.

focus()

Returns: Nothing.

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

The minute you create another window for the user in your Web site environment, you must pay attention to window layer management. With browser windows so easily activated by the slightest mouse click, a user can lose a smaller window behind a larger one in a snap. Most inexperienced Navigator users won't think to pull down the Window or Communicator menu to see whether the smaller window is still open and then activate it from the menu. If that subwindow is important to your site design, then you should present a button or other device in each window that enables users to safely switch between windows. The `window.focus()` method brings the referenced window to the front of all the windows.

Rather than supply a separate button on your page to bring a hidden window forward, you should build your window-opening functions in such a way that if the window is already open, the function automatically brings that window forward, as shown in the example that follows. This removes the burden of window management from your visitors.

The key to success with this method is making sure that your references to the desired windows are correct. Therefore, be prepared to use the `window.opener` property to refer to the main window if a subwindow needs to bring the main window back into focus. If your windowing environment consists of three or more windows, you have to make sure that you assign a unique name to each window and then use those names if subwindows need to communicate with other subwindows.

Example

To show how both the `window.focus()` method and its opposite, `window.blur()`, operate, Listing 14-29 creates a two-window environment. From each window, you can bring the other window to the front. The main window uses the object returned by `window.open()` to assemble the reference to the new window. In the subwindow (whose content is created entirely on the fly by JavaScript), `self.opener` is summoned to refer to the original window, while `self`

is used to direct the `blur()` method to the subwindow itself. Blurring one window and focusing on another window both have the same result of sending the window to the back of the pile.

Listing 14-29: **The window.focus() and window.blur() Methods**

```
<HTML>
<HEAD>
<TITLE>Focus() and Blur()</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
// declare global variable name
var newWindow = null
function makeNewWindow() {
        // check if window already exists
        if (!newWindow || newWindow.closed) {
            // store new window object in global variable
            newWindow = window.open("","","width=250,height=250")
            // assemble content for new window
            var newContent = "<HTML><HEAD><TITLE>Another Sub
Window</TITLE></HEAD>"
            newContent += "<BODY bgColor='salmon'><H1>A Salmon-Colored
Subwindow.</H1>"
            newContent += "<FORM><INPUT TYPE='button' VALUE='Bring Main
to Front' onClick='self.opener.focus()'>"
            newContent += "<FORM><INPUT TYPE='button' VALUE='Put Me in
Back' onClick='self.blur()'>"
            newContent += "</FORM></BODY></HTML>"
            // write HTML to new window document
            newWindow.document.write(newContent)
            newWindow.document.close()
        } else {
        // window already exists, so bring it forward
        newWindow.focus()
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" NAME="newOne" VALUE="Show New Window"
onClick="makeNewWindow()">
</FORM>
</BODY>
</HTML>
```

A key ingredient to the success of the `makeNewWindow()` function in Listing 14-29 is the first conditional expression. Because `newWind` is initialized as a null value when the page loads, that is its value the first time through the function. But after the subwindow is opened the first time, `newWind` is assigned a value (the subwindow object) that remains intact even if the user closes the window. Thus, the value doesn't revert to null by itself. To catch the possibility that the user has

closed the window, the conditional expression also sees if the window is closed. If it is, a new subwindow is generated, and that new window's reference value is reassigned to the `newWind` variable. On the other hand, if the window reference exists and the window is not closed, that subwindow is brought to the front with the `focus()` method.

Related Items: `window.open()` method; `window.blur()` method; `window.opener` property.

forward()

Returns: Nothing.

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

The Forward button's behavior has followed the same evolution as the Back button's since Navigator 2. In Navigator 2, the history object (and all navigation methods associated with it) assumed the entire browser window would change with a click of the Back or Forward button in the toolbar. With the increased popularity of frames, this mechanism didn't work well if one frame remained static while documents flew in and out of another frame: navigation had to be on a frame-by-frame basis, and that's how the Back and Forward buttons worked in Navigator 3 and now in Navigator 4.

From Navigator 3 onward, each window object (including frames) maintains its own history. Unfortunately, JavaScript doesn't observe this until you get to Navigator 4, and thus for a lot of browsers out there (including Internet Explorer 3 and Internet Explorer 4), the history navigation methods control the global history. The purpose of the `window.forward()` method is to offer a scripted version of the global forward navigation, while allowing the history object to control navigation strictly within a particular window or frame — as it should. For more information about version compatibility and the back and forward navigation, see the history object in Chapter 15.

Example

See the Navigation laboratory example in Listing 14-20 and 14-21 to see the differences among the various navigation methods.

Related Items: `window.back()` method; history object.

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.

Every object that has event handlers associated with it also has a `handleEvent()` method. Thus, if you are capturing click events in a window, you can redirect the events to, say, a particular button or link on the page because both of those objects know what to do with click events. Consider the following code excerpt:

```
<SCRIPT LANGUAGE="JavaScript1.2">
// function to run when window captures a click event
function doClicks(e) {
        // send all clicks to the first link in the document
        document.links[0].handleEvent(e)
}
// set window to capture click events
window.captureEvents(Event.CLICK)
// assign doClick() function to click events captured by window
window.onclick = doClicks
</SCRIPT>
```

The window is set up to capture all click events and invoke the `doClicks()` function each time the user clicks on a clickable item in the window. In the `doClicks()` function is a single statement that instructs the first link in the document to handle the click event being passed as a parameter. The link must have an `onClick=` event handler defined for this to be meaningful. Because the event object is passed along, the link's event handler can examine event properties (for example, location of the click) and perhaps alter some of the link's properties before letting it perform its linking task. The preceding example is really showing how to use `handleEvent()` with a link object, rather than a window object. There is little opportunity for other objects to capture events that normally go to the window, but this method is part of every event-aware object.

Example

See Chapter 33 for details and in-depth examples of working with the event object.

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

home()

Returns: Nothing.

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

Like many of the window methods new to Navigator 4, the `window.home()` method provides a scripted way of replicating the action of a toolbar button: the Home button. The action navigates the browser to whatever URL is set in the browser preferences for home page location. Even if you have the starting page set to a blank page, both the Home button and the `window.home()` method go to the URL. You cannot control the default home page of a visitor's browser. Therefore, I recommend that you use this method only if you provide an alternative interface to the toolbar you have turned off (with a signed script).

Related Items: `window.back()` method; `window.forward()` method; `window.toolbar` property.

moveBy(*deltaX*,*deltaY*)
moveTo(*x*,*y*)

Returns: Nothing.

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

JavaScript (starting with Navigator 4) can adjust the location of a browser window on the screen. This applies to the main window or any subwindow generated by script. The only security restriction that applies is moving the window off screen entirely: you need a signed script and the user's permission to hide a window this way.

You can move a window to an absolute position on the screen or adjust it along the horizontal and/or vertical axis by any number of pixels, irrespective of the absolute pixel position. The coordinate space for the x (horizontal) and y (vertical) position is the entire screen, with the top-left corner representing 0,0. The point of the window you set with the `moveBy()` and `moveTo()` methods is the very top-left corner of the outer edge of the browser window. Therefore, when you move the window to point 0,0, that sets the window flush with the top-left corner of the screen.

If you try to adjust the position of the window such that any edge falls beyond the screen area, the window remains at the edge of the screen — unless you are using a signed script and have the user's permission to adjust the window partially or completely offscreen. It is dangerous to move the only visible browser window entirely off screen, because the user has no way to get it back into view without quitting and relaunching Navigator 4.

The difference between the `moveTo()` and `moveBy()` methods is that one is an absolute move, while the other is relative. Parameters you specify for `moveTo()` are the precise horizontal and vertical pixel counts on the screen where you want the upper-left corner of the window to appear. In contrast, the parameters for `moveBy()` indicate how far to adjust the window location in either direction. If you want to move the window 25 pixels to the right, you must still include both parameters, but the `y` value will be zero:

```
window.moveBy(25,0)
```

To move to the left, the first parameter must be a negative number.

Example

Several examples of using the `window.moveTo()` and `window.moveBy()` methods are shown in Listing 14-30. The page presents four buttons, each of which performs a different kind of browser window movement.

Listing 14-30: Window Boogie

```
<HTML>
<HEAD>
<TITLE>Window Gymnastics</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
// function to run when window captures a click event
function moveOffScreen() {
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
        var maxX = screen.width
        var maxY = screen.height
        window.moveTo(maxX+1, maxY+1)
        setTimeout("window.moveTo(0,0)",500)
netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite")
}
// moves window in a circular motion
function revolve() {
        var winX = (screen.availWidth - window.outerWidth) / 2
        var winY = 50
        window.resizeTo(300,200)
        window.moveTo(winX, winY)

        for (var i = 1; i < 36; i++) {
           winX += Math.cos(i * (Math.PI/18)) * 5
           winY += Math.sin(i * (Math.PI/18)) * 5
           window.moveTo(winX, winY)
        }
}
// moves window in a horizontal zig-zag pattern
function zigzag() {
        window.resizeTo(300,200)
        window.moveTo(0,80)
        var incrementX = 2
        var incrementY = 2
        var floor = screen.availHeight - outerHeight
        var rightEdge = screen.availWidth - outerWidth
        for (var i = 0; i < rightEdge; i += 2) {
           window.moveBy(incrementX, incrementY)
           if (i%60 == 0) {
              incrementY = -incrementY
           }
        }
}
// resizes window to occupy all available screen real estate
function maximize() {
```

```
        window.moveTo(0,0)
        window.outerWidth = screen.availWidth
        window.outerHeight = screen.availHeight
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="buttons">
<B>Window Gymnastics</B><P>
<UL>
<LI><INPUT NAME="offscreen" TYPE="button" VALUE="Disappear a Second"
onClick="moveOffScreen()">
<LI><INPUT NAME="circles" TYPE="button" VALUE="Circular Motion"
onClick="revolve()">
<LI><INPUT NAME="bouncer" TYPE="button" VALUE="Zig Zag"
onClick="zigzag()">
<LI><INPUT NAME="expander" TYPE="button" VALUE="Maximize"
onClick="maximize()">
</UL>
</FORM>
</BODY>
</HTML>
```

The first button requires that you have codebase principals turned on (see Chapter 40) to take advantage of what would normally be a signed script. The `moveOffScreen()` function momentarily moves the window entirely out of view. Notice how the script determines the size of the screen before deciding where to move the window. After the journey off screen, the window comes back into view at the upper-left corner of the screen.

If using the Web sometimes seems like going around in circles, then the second function, `revolve()`, should feel just right. After reducing the size of the window and positioning it near the top center of the screen (notice the calculation to determine the x coordinate for centering the window horizontally), the script uses a bit of math to position the window along 36 places around a perfect circle (at 10-degree increments). This is an example of how to dynamically control a window's position based on math calculations.

To demonstrate the `moveBy()` method, the third function, `zigzag()`, uses a `for` loop to increment the coordinate points to make the window travel in a sawtooth pattern across the screen. The x coordinate continues to increment linearly until the window is at the edge of the screen (also calculated on the fly to accommodate any size monitor). The y coordinate must increase and decrease as that parameter changes direction at various times across the screen.

In the fourth function, you see some practical code (finally) that demonstrates how best to maximize the browser window to fill the entire available screen space on the visitor's monitor. Notice that instead of using the `resizeTo()` method, I set the `outerHeight` and `outerWidth` properties of the window. These settings are less bug prone than the `resizeTo()` method.

Related Items: `window.outerHeight` property; `window.outerWidth` property; `window.resizeBy()` method; `window.resizeTo()` method.

open("*URL*", "*windowName*" [, "*windowFeatures*"])

Returns: A window object representing the newly created window; null if method fails.

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

With the `window.open()` method, a script provides a Web site designer with an immense range of options for the way a second or third Web browser window looks on the user's computer screen. Moreover, most of this control can work with all JavaScript-enabled browsers without the need for signed scripts. Because the interface elements of a new window are easier to envision, I cover those aspects of the `window.open()` method parameters first.

The optional `windowFeatures` parameter is *one string*, consisting of a comma-separated list of assignment expressions (behaving something like HTML tag attributes). If you omit this third parameter, JavaScript creates the same type of new window you'd get from the New Web Browser menu choice in the File menu. But you can control which window elements appear in the new window with the third parameter. Remember this important rule: If you specify *any* of the method's original set of third parameter values, all of those features are turned off unless the parameters specify the features to be switched on. Table 14-1 lists the attributes you can control for a newly created window in all browsers.

Table 14-1
window.open() Method Attributes Controllable via Script

Attribute	*Value*	*Description*
`toolbar`	Boolean	"Back," "Forward," and other buttons in the row
`location`	Boolean	Field displaying the current URL
`directories`	Boolean	"What's New" and other buttons in the row
`status`	Boolean	Statusbar at bottom of window
`menubar`*	Boolean	Menubar at top of window
`scrollbars`	Boolean	Displays scrollbars if document is larger than window
`resizable`**	Boolean	Interface elements that allow resizing by dragging
`copyhistory`	Boolean	Duplicates Go menu history for new window
`width`	pixelCount	Window outer width in pixels
`height`	pixelCount	Window outer height in pixels

* Not on Macintosh because the menubar is not in the browser window; when off in Navigator 4, displays an abbreviated Mac menubar.

** Macintosh windows are always resizable.

Boolean values for true can be either `yes`, `1`, or just the feature name by itself; for false, use a value of `no` or `0`. If you omit any Boolean attributes, they are rendered as false. Therefore, if you want to create a new window that shows only the toolbar and statusbar and is resizable, the method looks like this:

```
window.open("newURL","NewWindow", "toolbar,status,resizable")
```

A new window that does not specify the height and width is set to the default size of the browser window that the browser creates from a File menu's New Web Browser command. In other words, a new window does not automatically inherit the size of the window making the `window.open()` method call. A new window created via a script is positioned somewhat arbitrarily, depending on the operating system platform of the browser. Generally, though, the position is at or near the top-left corner of the screen, just as a new Web browser window would be; window position is not scriptable except in Navigator 4.

Speaking of Navigator 4, this browser version includes a suite of extra features for the `window.open()` method. Those parameters deemed to be security risks require signed scripts and the user's permission before they are recognized. If the user fails to grant permission, the secure parameter is ignored. Table 14-2 shows the extra window features provided by Navigator 4.

Table 14-2
Extra window.open() Method Attributes in Navigator 4

Attribute	*Value*	*Description*
`alwaysLowered`*	Boolean	Always behind other browser windows
`alwaysRaised`*	Boolean	Always in front of other browser windows
`dependent`	Boolean	Subwindow closes if the opener window closes
`hotkeys`	Boolean	If true, disables menu shortcuts (except Quit and Security Info) when menubar is turned off
`innerHeight`**	pixelCount	Content region height; same as old `height` property
`innerWidth`**	pixelCount	Content region width; same as old `width` property
`outerHeight`**	pixelCount	Visible window height
`outerWidth`**	pixelCount	Visible window width
`screenX`**	pixelCount	Horizontal position of top-left corner on screen
`screenY`**	pixelCount	Vertical position of top-left corner on screen
`titlebar`*	Boolean	Title bar and all other border elements
`z-lock`*	Boolean	Window layer is fixed below browser windows

* Requires a signed script

** Requires a signed script to size or position a window beyond safe threshold

A couple of these new attributes have different behaviors on different operating system platforms, due to the way the systems manage their application windows.

For example, the `alwaysLowered`, `alwaysRaised`, and `z-locked` styles can exist in layers that range behind Navigator 4's own windows in the Windows 95 platform; on the Mac, however, such windows are confined to the levels occupied by Navigator 4. The difference is that Windows 95 allows windows from multiple applications to interleave each other, while the Mac keeps each application's windows in contiguous layers.

To apply signed scripts to opening a new window with the secure window features, you must enable `UniversalBrowserWrite` privileges like you do for other signed scripts (see Chapter 40). A code fragment that generates an `alwaysRaised` style window follows:

```
<SCRIPT LANGUAGE="JavaScript" ARCHIVE="myJar.jar" ID="1">
function newRaisedWindow() {
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
        var newWindow =
window.open("","","HEIGHT=100,WIDTH=300,alwaysRaised")

netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite")
        var newContent = "<HTML><BODY><B> On top of spaghetti!"</B>"
        newContent += "<FORM><CENTER><INPUT TYPE='button' VALUE='OK'"
        newContent +=
"onClick='self.close()'></CENTER></FORM></BODY></HTML>"
        newWindow.document.write(newContent)
        newWindow.document.close()
}
</SCRIPT>
```

You can experiment with the look and behavior of new windows with any combination of attributes with the help of the script in Listing 14-31. This page presents a table of all new window Boolean attributes and creates a new 300-by-300 pixel window based on your choices. This page assumes that if you are using Navigator 4 you have codebase principals turned on for signed scripts (see Chapter 40). The interface for this laboratory is shown in Figure 14-14.

Be careful with turning off the title bar and hotkeys. With the title bar off, the content appears to float in space, because absolutely no borders are displayed. With hotkeys still turned on, you can use Ctrl+W to close this borderless window (except on the Mac, for which the hotkeys are always disabled with the title bar off). This is how you can turn a computer into a kiosk by sizing a window to the screen's dimensions and setting the window options to `"titlebar=no, hotkeys=no,alwaysRaised=yes"`.

Listing 14-31: **New Window Laboratory**

```
<HTML>
<HEAD>
<TITLE>window.open() Options</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var isNav4 = (navigator.appName == "Netscape" &&
navigator.appVersion.charAt(0) == 4) ? true : false

function makeNewWind(form) {
```

```
        if (isNav4) {

netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
        }
        var attr = "HEIGHT=300,WIDTH=300"
        for (var i = 0; i < form.elements.length; i++) {
           if (form.elements[i].type == "checkbox") {
              attr += "," + form.elements[i].name + "="
              attr += (form.elements[i].checked) ? "yes" : "no"
           }
        }
        var newWind = window.open("bofright.htm","subwindow",attr)
        if (isNav4) {

netscape.security.PrivilegeManager.disablePrivilege("CanvasAccess")
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<B>Select new window options:</B>
<TABLE BORDER=2>
<TR>
        <TD COLSPAN=2 BGCOLOR="yellow" ALIGN="middle">All Browsers
Features:</TD>
</TR>
<TR>
        <TD><INPUT TYPE="checkbox" NAME="toolbar">toolbar</TD>
        <TD><INPUT TYPE="checkbox" NAME="location">location</TD>
</TR>
<TR>
        <TD><INPUT TYPE="checkbox" NAME="directories">directories</TD>
        <TD><INPUT TYPE="checkbox" NAME="status">status</TD>
</TR>
<TR>
        <TD><INPUT TYPE="checkbox" NAME="menubar">menubar</TD>
        <TD><INPUT TYPE="checkbox" NAME="scrollbars">scrollbars</TD>
</TR>
<TR>
        <TD><INPUT TYPE="checkbox" NAME="resizable">resizable</TD>
        <TD><INPUT TYPE="checkbox" NAME="copyhistory">copyhistory</TD>
</TR>
<TR>
        <TD COLSPAN=2 BGCOLOR="yellow" ALIGN="middle">Communicator
Features:</TD>
</TR>
<TR>
        <TD><INPUT TYPE="checkbox"
NAME="alwaysLowered">alwaysLowered</TD>
        <TD><INPUT TYPE="checkbox" NAME="alwaysRaised">alwaysRaised</TD>
</TR>
```

(continued)

Listing 14-31 *Continued*

```
<TR>
        <TD><INPUT TYPE="checkbox" NAME="dependent">dependent</TD>
        <TD><INPUT TYPE="checkbox" NAME="hotkeys" CHECKED>hotkeys</TD>
</TR>
<TR>
        <TD><INPUT TYPE="checkbox" NAME="titlebar" CHECKED>titlebar</TD>
        <TD><INPUT TYPE="checkbox" NAME="z-lock">z-lock</TD>
</TR>
<TR>
        <TD COLSPAN=2 ALIGN="middle"><INPUT TYPE="button" NAME="forAll"
VALUE="Make New Window" onClick="makeNewWind(this.form)"></TD>
</TR>
</TABLE>
<BR>
</FORM>
</BODY>
</HTML>
```

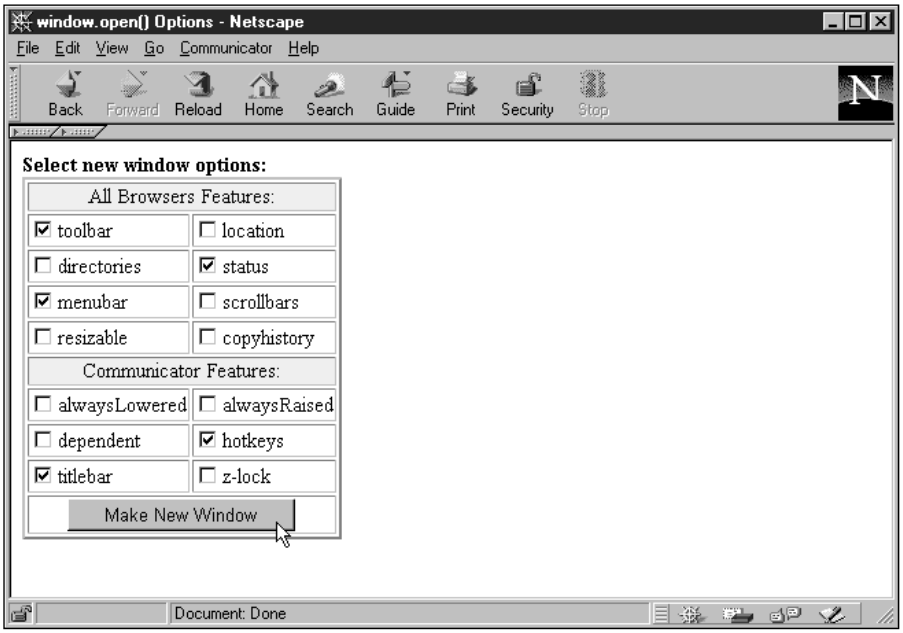

Figure 14-14: A new window attribute laboratory interface

Getting back to the other parameters of `window.open()`, the middle parameter is the name for the new window. Don't confuse this parameter with the document's title, which would normally be set by whatever HTML text determines the content of the window. A window name must be the same style of one-word identifier you

use for other object names and variables. This name is also an entirely different entity than the window object that the `open()` method returns. You don't use the name in your scripts. At most, the name can be used for `TARGET` attributes of links and forms.

A script generally populates a window with one of two kinds of information:

- ✦ An existing HTML document whose URL is known beforehand
- ✦ An HTML page created on the fly

To create a new window that displays an existing HTML document, supply the full URL as the first parameter of the `window.open()` method. If your page is having difficulty loading a URL into a new page (except as noted in the sidebar "A Navigator 2 bug workaround"), try specifying the complete URL of the target document (instead of just the filename).

Leaving the first parameter as an empty string forces the window to open with a blank document, ready to have HTML written to it by your script (or loaded separately by another statement that sets that window's location to a specific URL). If you plan to write the content of the window on the fly, assemble your HTML content as one long string value and then use the `document.write()` method to post that content to the new window. If you plan to append no further writing to the page, also include a `document.close()` method at the end to tell the browser that you're finished with the layout (so that the `Layout:Complete` or `Document:Done` message appears in the statusbar, if your new window has one).

A call to the `window.open()` method returns a value of the new window's object if the window opens successfully. This value is vitally important if your script needs to address elements of that new window (such as when writing to its document). When a script creates a new window, the default window object (which the script normally points to) still contains the document that holds the script. The new window does not, even though it may be on top. Therefore, to further manipulate items within the new window, you need a reference to that new window object. After the new window is open, however, no parent-child relationship exists between the windows.

To handle references to the subwindow properly, you should always assign the result of a `window.open()` method to a global variable. Before writing to the new window the first time, test the variable to make sure that it is not a null value — the window may have failed to open because of low memory, for instance. If everything is okay, you can use that variable as the beginning of a reference to any property or object within the new window. For example

```
newWindow = window.open("","")
if (newWindow != null) {

newWindow.document.write("<HTML><HEAD><TITLE>Hi!</TITLE></HEAD>")
}
```

If you initialize the new window's variable as a global variable (see Chapter 34), any value that the variable receives (even if it gets the value while inside a function) remains in effect as long as the original document stays loaded in the first window. You can come back to that value in another script handler (perhaps some button that closes the subwindow) by making the proper reference to the new window.

A Navigator 2 bug workaround

If you're concerned about backward compatibility with Navigator 2, you should be aware of a bug in the Macintosh and UNIX flavors of the browser. In those versions, if you include a URL as a parameter to `window.open()`, Navigator opens the window but does not load the URL. A second call to the `window.open()` method is required. Moreover, the second parameter must be an empty string if you add any third-parameter settings. Here is a sample listing you can adapt for your own usage:

```
<HTML>
<HEAD>
<TITLE>New Window</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// workaround for window.open() bug on X and Mac platforms
function makeNewWindow() {
     var newWindow =
window.open("http://www.dannyg.com","","status,height=200,width=300")
     if (navigator.appVersion.charAt(0) == "2" &&
navigator.appName == "Netscape") {
           newWindow =
window.open("http://www.dannyg.com","","status,height=200,width=300")
     }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" NAME="newOne" VALUE="Create New Window"
onClick="makeNewWindow()">
</FORM>
</BODY>
</HTML>
```

This workaround can also be used without penalty in Windows versions of Navigator 2 and Navigator 3.

When scripts in the subwindow need to communicate with objects and scripts in the originating window, you must make sure that the subwindow has an `opener` property if the level of JavaScript in the visitor's browser doesn't automatically supply one. See the discussion about the `window.opener` property earlier in this chapter.

Invoking multiple `window.open()` methods with the same window name parameter (the second parameter) does not create additional copies of that window in Netscape browsers (although it does in Internet Explorer 3). JavaScript prevents you from creating two windows with the same name. Nor does a `window.open()` method bring an existing window of that name to the front of the window layers: Use `window.focus()` for that.

Example

In Listing 14-32, I install a button that generates a new window of a specific size that has only the statusbar turned on. The script here shows all the elements necessary to create a new window that has all the right stuff on most platforms. The new window object reference is assigned to a global variable, `newWindow`. Before a new window is generated, the script looks to see if the window has never been generated before (in which case `newWindow` would be null) or, for newer browsers, the window is closed. If either condition is true, the window is created with the `open()` method. Otherwise, the existing window is brought forward with the `focus()` method (Navigator 3 and up; Internet Explorer 4).

As a safeguard against older browsers, the script manually adds an `opener` property to the new window if one is not already assigned by the `open()` method. The current window object reference is assigned to that property.

To build the string that is eventually written to the document, I use the += (add-by-value) operator, which appends the string on the right side of the operator to the string stored in the variable on the left side. In this example, the new window is handed an `<H1>`-level line of text to display.

Listing 14-32: **Creating a New Window**

```
<HTML>
<HEAD>
<TITLE>New Window</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var newWindow
function makeNewWindow() {
        if (!newWindow || newWindow.closed) {
           newWindow = window.open("","","status,height=200,width=300")
           if (!newWindow.opener) {
              newWindow.opener = window
           }
           // assemble content for new window
           var newContent = "<HTML><HEAD><TITLE>One Sub
Window</TITLE></HEAD>"
           newContent += "<BODY><H1>This window is brand new.</H1>"
           newContent += "</BODY></HTML>"
           // write HTML to new window document
           newWindow.document.write(newContent)
           newWindow.document.close() // close layout stream
        } else {
           // window's already open; bring to front
           newWindow.focus()
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" NAME="newOne" VALUE="Create New Window"
 onClick="makeNewWindow()">
```

(continued)

Listing 14-32 *Continued*

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

If you need to create a new window for the lowest common denominator of scriptable browser, you will have to omit the `focus()` method and the `window.closed` property from the script (as well as add the bug workaround described earlier). Or you may prefer to forego a subwindow for all browsers below a certain level. For example, Navigator 3 and up provide a solid foundation for new window features available for Internet Explorer users only from Version 4. But also see Listing 14-3 (in the `window.closed` property discussion) for other ideas about cross-platform authoring for subwindows.

Related Items: `window.close()` method; `window.blur()` method; `window.focus()` method; `window.closed` property.

print()

Returns: Nothing.

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

Like several other new window methods for Navigator 4, `print( )` provides a scripted way of invoking the Print button in the toolbar, whether the toolbar is visible or not. Printing, however, is a little different, because a user — and now a script — can specify that the entire browser window be printed or just a particular frame. If you build a reference to the `print()` method with a reference to a frame, then just that frame will be printed. To prevent a rogue `print()` command from tying up a printer without the user's permission, the `print()` method goes only so far as to present the browser's print dialog box. The user must still click the OK or Print button (depending on the operating system) to send the window or frame content to the printer.

Example

Listing 14-33 is a frameset that loads Listing 14-34 into the top frame and a copy of the Bill of Rights into the bottom frame.

Listing 14-33: **Print Frameset**

```
<HTML>
<HEAD>
<TITLE>window.print() method</TITLE>
```

```
</HEAD>
<FRAMESET ROWS="25%,75%">
        <FRAME NAME="controls" SRC="lst14-34.htm">
        <FRAME NAME="display" SRC="bofright.htm">
</FRAMESET>
</HTML>
```

Two buttons in the top control panel (Listing 14-34) let you print the whole frameset or just the lower frame. To print the entire frameset, the reference includes the parent window; to print the lower frame, the reference is directed at the `parent.display` frame.

Listing 14-34: **Printing Control**

```
<HTML>
<HEAD>
<TITLE>Print()</TITLE>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" NAME="printWhole" VALUE="Print Entire Frameset"
onClick="parent.print()"><P>
<INPUT TYPE="button" NAME="printFrame" VALUE="Print Bottom Frame Only"
onClick="parent.display.print()"><P>
</FORM>
</BODY>
</HTML>
```

If you don't like some facet of the printed output, blame the browser's print engine, and not JavaScript. The `print()` method merely invokes the browser's regular printing routines. Pages whose content is generated entirely by JavaScript print only in Navigator 3 and later (and Internet Explorer 4). A page containing some HTML and some JavaScript-generated content prints only the HTML portion.

Related Items: `window.back()` method; `window.forward()` method; `window.home()` method; `window.find()` method.

`prompt(`*`message, defaultReply`*`)`

Returns: String of text entered by user or null.

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

The third kind of dialog box that JavaScript can display includes a message from the script author, a field for user entry, and two buttons (OK and Cancel, or Yes

and No on Mac versions of Navigator 2 and 3). The script writer can supply a prewritten answer so a user confronted with a prompt dialog box can click OK (or press Enter) to accept that answer without further typing. Supplying both parameters to the `window.prompt()` method is important. Even if you don't want to supply a default answer, enter an empty string as the second parameter:

```
prompt("What is your postal code?","")
```

If you omit the second parameter, JavaScript inserts the string `<undefined>` into the dialog box's field. This will be disconcerting to most Web page visitors.

The value returned by this method is a string in the dialog box's field when the user clicks on the OK button. If you're asking the user to enter a number, remember that the value returned by this method is a string. You may need to perform data-type conversion with the `parseInt()` or `parseFloat()` functions (see Chapter 28) to use the returned values in math calculations.

When the user clicks on the prompt dialog box's OK button without entering any text into a blank field, the returned value is an empty string (""). Clicking on the Cancel button, however, makes the method return a null value. Therefore, the scripter must test for the type of returned value to make sure that the user entered some data that can be processed later in the script, as in

```
var entry = prompt("Enter a number between 1 and 10:","")
if (entry != null) {
        statements to execute with the value
}
```

This script excerpt assigns the results of the prompt dialog box to a variable and executes the nested statements if the returned value of the dialog box is not null
(if the user clicked on the OK button). The rest of the statements then have to include data validation to make sure that the entry was a number within the desired range (see Chapter 37).

It may be tempting to use the prompt dialog box as a handy user input device. But, like the other JavaScript dialog boxes, the modality of the prompt dialog box is disruptive to the user's flow through a document and can also trap automated macros that some users activate to capture Web sites. In forms, HTML fields are better user interface elements for attracting user text entry. Perhaps the safest way to use a prompt dialog box is to have it appear when a user clicks a button element on a page — and then only if the information you require of the user can be provided in a single prompt dialog box. Presenting a sequence of prompt dialog boxes is downright annoying to users.

Example

The function that receives values from the prompt dialog box in Listing 14-35 (see the dialog box in Figure 14-15) does some data-entry validation (but certainly not enough for a commercial site). It first checks to make sure that the returned value is neither null (Cancel) nor an empty string (the user clicked on OK without entering any values). See Chapter 37 for more about data-entry validation.

Listing 14-35: **The Prompt Dialog Box**

```
<HTML>
<HEAD>
<TITLE>window.prompt() Method</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function populateTable() {
        var howMany = prompt("Fill in table for how many factors?","")
        if (howMany != null && howMany != "") {
           alert("Filling the table for " + howMany) // for demo
           //statements that validate the entry and
           //actually populate the fields of the table
        }
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<!-- other statements that display and populate a large table -->
<INPUT TYPE="button" NAME="fill" VALUE="Fill Table..."
onClick="populateTable()">
</FORM>
</BODY>
</HTML>
```

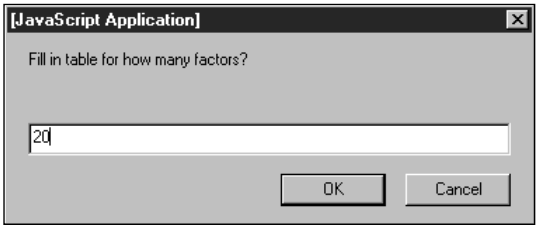

Figure 14-15: The prompt dialog box displayed from Listing 14-35 (Windows 95 format)

Notice one important user interface element in Listing 14-35. Because clicking on the button leads to a dialog box that requires more information from the user, the button's label ends in an ellipsis (or, rather, three periods acting as an ellipsis character). The ellipsis is a common courtesy to let users know that a user interface element leads to a dialog box of some sort. As in similar situations in Windows 95 and Macintosh programs, the user should be able to cancel out of that dialog box and return to the same screen state that existed before the button was clicked.

Related Items: `window.alert()` method; `window.confirm()` method.

releaseEvents(*eventTypeList*)

Returns: Nothing.

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

If your scripts have enabled event capture for the window object (or document 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. Parameters for the `releaseEvents()` method are one or more event types. Each event type is its own entity, so if your window captures three event types at one point, you can release some or all of those event types as the visitor interacts with your page. For example, if the page loads and captures three types of events, as in

```
window.captureEvents(Event.CLICK | Event.KEYPRESS | Event.CHANGE)
```

you can later turn off window event capture for all but the click event:

```
window.releaseEvents(Event.KEYPRESS | Event.CHANGE)
```

The window will still capture and process click events, but keyPress and change events go directly to their target objects.

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

resizeBy(*deltaX*,*deltaY*)

resizeTo(*outerwidth*,*outerheight*)

Returns: Nothing.

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

Starting with Navigator 4, scripts can control the size of the current browser window on the fly. While you can set the individual inner and outer width and height properties of a window, the `resizeBy()` and `resizeTo()` methods let you adjust both axis measurements in one statement. In both instances, all adjustments affect the lower-right corner of the window: To move the top-left corner, use the `window.moveBy()` or `window.moveTo()` methods.

Each resize method requires a different kind of parameter. The `resizeBy()` method adjusts the window *by* a certain number of pixels along one or both axes. Therefore, it is not concerned with the specific size of the window beforehand —

only by how much each axis is to change. For example, to increase the current window size by 100 pixels horizontally and 50 pixels vertically, the statement would be

```
window.resizeBy(100, 50)
```

Both parameters are required, but if you only want to adjust the size in one direction, set the other to zero. This would be the same as adding a value to one of the outer window measurement properties. You may also shrink the window by using negative values for either or both parameters.

There is greater need for the `resizeTo()` method, especially when you know that on a particular platform the window needs adjustment to a specific width and height to best accommodate that platform's display of form elements. Parameters for the `resizeTo()` method are the actual pixel width and height of the outer dimension of the window — the same as the `window.outerWidth` and `window.outerHeight` properties. In practice, I have found that the `resizeTo()` method behaves less accurately than setting the individual properties. If you need to maximize the browser window to the user's monitor, use the script segment shown in the `window.outerHeight` property discussion.

For both methods, you are limited to the viewable area of the screen unless the page uses signed scripts (see Chapter 40). With signed scripts and the users permission, you can adjust windows beyond the available screen borders.

Example

You can experiment with the resize methods with the page in Listing 14-36. Two parts of a form let you enter values for each method. The one for `window.resize()` also lets you enter a number of repetitions to better see the impact of the values. Enter zero and negative values to see how those affect the method.

Listing 14-36: **Window Resize Methods**

```
<HTML>
<HEAD>
<TITLE>Window Resize Methods</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function doResizeBy(form) {
        var x = parseInt(form.resizeByX.value)
        var y = parseInt(form.resizeByY.value)
        var count = parseInt(form.count.value)
        for (var i = 0; i < count; i++) {
           window.resizeBy(x, y)
        }
}
function doResizeTo(form) {
        var x = parseInt(form.resizeToX.value)
        var y = parseInt(form.resizeToY.value)
        window.resizeTo(x, y)
}
</SCRIPT>
</HEAD>
```

(continued)

Listing 14-36 *Continued*

```
<BODY>
<FORM>
<B>Enter the x and y increment, plus how many times the window should
be resized by these increments:</B><BR>
Horiz:<INPUT TYPE="text" NAME="resizeByX" SIZE=4>
Vert:<INPUT TYPE="text" NAME="resizeByY" SIZE=4>
How Many:<INPUT TYPE="text" NAME="count" SIZE=4>
<INPUT TYPE="button" NAME="ResizeBy" VALUE="Show resizeBy()"
onClick="doResizeBy(this.form)">
<HR>
<B>Enter the desired width and height of the current window:</B><BR>
Width:<INPUT TYPE="text" NAME="resizeToX" SIZE=4>
Height:<INPUT TYPE="text" NAME="resizeToY" SIZE=4>
<INPUT TYPE="button" NAME="ResizeTo" VALUE="Show resizeTo()"
onClick="doResizeTo(this.form)">
</FORM>
</BODY>
</HTML>
```

Related Items: `window.outerHeight` property; `window.outerWidth` property; `window.moveTo()` 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, as 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.

Perhaps a more common reason for capturing events at the window (or similar) level is to look for special cases, such as when someone Ctrl+clicks on an element. In this case, even though the window event handler receives all click events, it performs further processing only when the `event.modifiers` property indicates the Ctrl key is also pressed and the `event.target` property reveals the item being clicked is a link rather than a button. All other instances of the click event are routed on their way to their destinations. The event object knows where it's going, so your `routeEvent()` method doesn't have to worry about that.

The parameter for the `routeEvent()` method is the event object that is passed to the function that processes the high-level event, as shown here:

```
function flashRed(e) {
   [statements that filter specific events to flash background color red]
   routeEvent(e)
}
```

The event object, `e`, comes into the function and is passed unmodified to the object that was clicked.

Example

The `window.routeEvent()` method is used in the example for `window.captureEvents()`, Listing 14-22.

Related Items: `window.captureEvents()` method; `window.releaseEvents()` method; `window.handleEvent()` method; event object.

scroll(horizontalCoord, verticalCoord)

Returns: Nothing.

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

Although you can control precious little about the look of an existing window via JavaScript, you can adjust the way a document scrolls inside a window or frame. On the surface, the `window.scroll()` method sounds like a practical power within scripts. Due to the way various platforms render fonts and other visual elements on the screen, the `window.scroll()` method works best when you use absolute positioning of Dynamic HTML. Unless you know the precise pixel location of a desired element to bring into view, the method won't be particularly valuable as an internal navigation device (navigating to anchors with the location object is more precise in this situation).

If you are designing for Navigator 4 and do not require backward compatibility to Navigator 3 or Internet Explorer 4, use the `window.scrollTo()` method instead of the `window.scroll()` method. Both methods perform the same operation, but the new method better fits into the latest direction of the Netscape object model.

The `window.scroll()` method takes two parameters, the horizontal (x) and vertical (y) coordinates of the document that is to be positioned at the top-left corner of the window or frame. You must realize that the window and document have two similar, but independent, coordinate schemes. From the window's point of view, the top-left pixel (of the active area) is point 0,0. All documents also have a 0,0 point: the very top-left of the document. The window's 0,0 point doesn't move, but the document's 0,0 point can move — via manual or scripted scrolling. Although `scroll()` is a window method, it seems to behave more like a document method, as the document appears to reposition itself within the window. Conversely, you can also think of the window moving to bring its 0,0 point to the designated coordinate of the document.

Although you can set values to ones that go beyond the maximum size of the document or to negative values, the results vary from platform to platform. For the moment, the best usage of the `window.scroll()` method is as a means of adjusting the scroll to the very top of a document (`window.scroll(0,0)`) when you want the user to be at a base location in the document. For vertical scrolling within a text-heavy document, an HTML anchor may be a better alternative for now (though it doesn't readjust horizontal scrolling).

Example

To demonstrate the `scroll()` method, Listing 14-37 defines a frameset with a document in the top frame (Listing 14-38) and a control panel in the bottom frame (Listing 14-39). A series of buttons and text fields in the control panel frame directs the scrolling of the document. I've selected an arbitrary, large GIF image to use in the example. To see results of some horizontal scrolling values, you may need to shrink the width of the browser window until a horizontal scrollbar appears in the top frame.

Listing 14-37: **A Frameset for the scroll() Demonstration**

```
<HTML>
<HEAD>
<TITLE>window.scroll() Method</TITLE>
</HEAD>

<FRAMESET ROWS="50%,50%">
        <FRAME SRC="lst14-38.htm" NAME="display">
        <FRAME SRC="lst14-39.htm" NAME="control">
</FRAMESET>
</HTML>
```

Listing 14-38: **The Image to Be Scrolled**

```
<HTML>
<HEAD>
<TITLE>Arch</TITLE>
</HEAD>

<BODY>
<H1>A Picture is Worth...</H1>
<HR>
<CENTER>
<TABLE BORDER=3>
<CAPTION ALIGN=bottom>A Splendid Arch</CAPTION>
<TD>
<IMG SRC="arch.gif">
</TD></TABLE></CENTER>
</BODY>
</HTML>
```

Listing 14-39: **Controls to Adjust Scrolling of the Upper Frame**

```
<HTML>
<HEAD>
<TITLE>Scroll Controller</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
function scroll(x,y) {
       parent.frames[0].scroll(x,y)
}
function customScroll(form) {

parent.frames[0].scroll(parseInt(form.x.value),parseInt(form.y.value))
}
</SCRIPT>
</HEAD>
<BODY>
<H2>Scroll Controller</H2>
<HR>
<FORM NAME="fixed">
Click on a scroll coordinate for the upper frame:<P>
<INPUT TYPE="button" VALUE="0,0" onClick="scroll(0,0)">
<INPUT TYPE="button" VALUE="0,100" onClick="scroll(0,100)">
<INPUT TYPE="button" VALUE="100,0" onClick="scroll(100,0)">
<P>
<INPUT TYPE="button" VALUE="-100,100" onClick="scroll(-100,100)">
<INPUT TYPE="button" VALUE="20,200" onClick="scroll(20,200)">
<INPUT TYPE="button" VALUE="1000,3000" onClick="scroll(1000,3000)">
</FORM>
<HR>
<FORM NAME="custom">
Enter an Horizontal
<INPUT TYPE="text" NAME="x" VALUE="0" SIZE=4>
and Vertical
<INPUT TYPE="text" NAME="y" VALUE="0" SIZE=4>
value.  Then
<INPUT TYPE="button" VALUE="click to scroll"
onClick="customScroll(this.form)">
</FORM>
</BODY>
</HTML>
```

Notice that in the `customScroll()` function, JavaScript must convert the string values from the two text boxes to integers (with the `parseInt()` method) for the `scroll()` method to accept them. Nonnumeric data can produce very odd results. Also be aware that although this example shows how to adjust the scroll values in another frame, you can set such values in the same frame or window as the script, as well as in subwindows, provided that you use the correct object references to the window.

Related Items: `window.scrollBy()` method; `window.scrollTo()` method.

scrollBy(*deltaX*,*deltaY*)
scrollTo(*x*,*y*)

Returns: Nothing.

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

Navigator 4 provides a related pair of window scrolling methods. The `window.scrollTo()` method is the new version of the `window.scroll()` method. The two work identically to position a specific coordinate point of a document at the top-left corner of the inner window region.

In contrast, the `window.scrollBy()` method allows for relative positioning of the document. Parameter values indicate by how many pixels the document should scroll in the window (horizontally and vertically). Negative numbers are allowed if you want to scroll to the left and/or upward. The `scrollBy()` method comes in handy if you elect to hide the scrollbars of a window or frame and offer other types of scrolling controls for your users. For example, to scroll down one screenful of a long document, you can use the `window.innerHeight` to determine what the offset from the current position would be:

```
window.scrollBy(0, window.innerHeight)
```

To scroll upward, use a negative value for the second parameter:

```
window.scrollBy(0, -window.innerHeight)
```

Scrolling the document in the Macintosh exhibits some buggy behavior. At times it appears as though you are allowed to scroll well beyond the document edges. In truth, the document has stopped at the border, but the window or frame may not have refreshed properly.

Example

To work with the `scrollTo()` method, you can use Listings 14-37 through 14-39 (the `window.scroll()` method) but substitute `window.scrollTo()` for `window.scroll()`. The results should be the same. For `scrollBy()`, the example starts with the frameset in Listing 14-40. It loads the same content document as the `window.scroll()` example (Listing 14-38), but the control panel (Listing 14-41) provides input to experiment with the `scrollBy()` method.

Listing 14-40: **Frameset for ScrollBy Controller**

```
<HTML>
<HEAD>
<TITLE>window.scrollBy() Method</TITLE>
</HEAD>

<FRAMESET ROWS="50%,50%">
```

```
        <FRAME SRC="lst14-38.htm" NAME="display">
        <FRAME SRC="lst14-41.htm" NAME="control">
</FRAMESET>
</HTML>
```

Notice in Listing 14-41 that all references to window properties and methods are directed to the display frame. String values retrieved from text fields are converted to number with the `parseInt()` global function.

Listing 14-41: **ScrollBy Controller**

```
<HTML>
<HEAD>
<TITLE>ScrollBy Controller</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function page(direction) {
        var deltaY = parent.display.innerHeight
        if (direction == "up") {
           deltaY = -deltaY
        }
        parent.display.scrollBy(0, deltaY)
}
function customScroll(form) {
        parent.display.scrollBy(parseInt(form.x.value),
parseInt(form.y.value))
}
</SCRIPT>
</HEAD>
<BODY>
<B>ScrollBy Controller</B>
<FORM NAME="custom">
Enter an Horizontal increment
<INPUT TYPE="text" NAME="x" VALUE="0" SIZE=4">
and Vertical
<INPUT TYPE="text" NAME="y" VALUE="0" SIZE=4">
value.<BR>Then
<INPUT TYPE="button" VALUE="click to scrollBy()"
onClick="customScroll(this.form)">
<HR>
<INPUT TYPE="button" VALUE="PageDown" onClick="page('down')">
<INPUT TYPE="button" VALUE="PageUp" onClick="page('up')">

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

Related Items: `window.pageXOffset` property; `window.pageYOffset` property; `window.scroll()` method.

setInterval("*functionOrExpr*", *msecDelay* [, *funcarg1*, ..., *funcargn*])

Returns: Interval ID integer.

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

It is important to understand the distinction between the `setInterval()` and `setTimeout()` methods. Before the `setInterval()` method was part of JavaScript, authors replicated the behavior with `setTimeout()`, but the task often required reworking scripts a bit.

Use `setInterval()` when your script needs to call a function or execute some expression repeatedly with a fixed time delay between calls to that function or expression. The delay is not at all like a wait state in some languages: Other processing does not halt while the delay is in effect. Typical applications include animation by moving an object around the page under controlled speed (instead of letting the JavaScript interpreter whiz the object through its path at CPU-dependent speeds). In a kiosk application, you can use `setInterval()` to advance "slides" that appear in other frames or as layers, perhaps changing the view every ten seconds. Clock displays and countdown timers would also be suitable usage of this method (even though you see examples in this book that use the old-fashioned `setTimeout()` way to perform timer and clock functions).

In contrast, `setTimeout()` is best suited for those times when you need to carry out a function or expression one time in the future — even if that future is only a second or two away. See the discussion of the `setTimeout()` method for details on this application.

The first parameter of the `setInterval()` method is the name of the function or expression to run when the interval elapses. This item must be a quoted string. If the parameter is a function, no function arguments are allowed inside the function's parentheses unless the arguments are literal strings. You can, however, include evaluated function arguments as a comma-delimited list, starting with the third parameter.

Putting function arguments in the final parameters of the `setInterval()` method is unique to Navigator 4. Internet Explorer 4 uses the third parameter to specify the scripting language of the statement or function being invoked in the first parameter. If you are trying to achieve cross-platform compatibility, design a function called by `setInterval()` so no arguments need to be passed. That way, Navigator 4 will ignore the third parameter required for Internet Explorer 4.

The second parameter of this method is the number of milliseconds (1,000 per second) that JavaScript should use as the interval between invocations of the function or expression. Even though the measure is in extremely small units, don't rely on 100 percent accuracy of the intervals. Various other internal processing delays may throw off the timing just a bit.

Like `setTimeout()`, `setInterval()` returns an integer value that is the ID for the interval process. That ID value lets you turn off the process with the `clearInterval()` method. That method takes the ID value as its sole parameter. This mechanism allows for the setting of multiple interval processes running, while giving your scripts the power to stop individual processes at any time without interrupting the others.

Example

The demonstration of the `setInterval()` method entails a two-framed environment. The framesetting document is shown in Listing 14-42.

Listing 14-42: SetInterval() Demonstration Frameset

```
<HTML>
<HEAD>
<TITLE>setInterval() Method</TITLE>
</HEAD>

<FRAMESET ROWS="50%,50%">
        <FRAME SRC="lst14-43.htm" NAME="control">
        <FRAME SRC="bofright.htm" NAME="display">
</FRAMESET>
</HTML>
```

In the top frame is a control panel with several buttons that control the automatic scrolling of the Bill of Rights text document in the bottom frame. Listing 14-43 shows the control panel document. Many functions here control the interval, scrolling jump size, and direction, and they demonstrate several aspects of applying `setInterval()`.

Notice that in the beginning the script establishes a number of global variables. Three of them are parameters that control the scrolling; the last one is for the ID value returned by the `setInterval()` method. The script needs that value to be a global value so a separate function can halt the scrolling with the `clearInterval()` method.

All scrolling is performed by the `autoScroll()` function. Because I want this method ultimately to coexist in a page usable in Internet Explorer 4 (with the help of a JScript segment to handle Internet Explorer 4's different manner of scrolling content), the `autoScroll()` method does not rely on parameters. Instead, all controlling parameters are global variables. In this application, placement of those values in globals helps the page restart autoscrolling with the same parameters as it had when it last ran.

Listing 14-43: SetInterval() Control Panel

```
<HTML>
<HEAD>
<TITLE>ScrollBy Controller</TITLE>
```

(continued)

Listing 14-43 *Continued*

```
<SCRIPT LANGUAGE="JavaScript1.2">
var scrollSpeed = 500
var scrollJump = 1
var scrollDirection = "down"
var intervalID

function autoScroll() {
        if (scrollDirection == "down") {
           scrollJump = Math.abs(scrollJump)
        } else if (scrollDirection == "up" && scrollJump > 0) {
           scrollJump = -scrollJump
        }
        parent.display.scrollBy(0, scrollJump)
        if (parent.display.pageYOffset <= 0) {
           clearInterval(intervalID)
        }
}

function reduceInterval() {
        stopScroll()
        scrollSpeed -= 200
        startScroll()
}
function increaseInterval() {
        stopScroll()
        scrollSpeed += 200
        startScroll()
}
function reduceJump() {
        scrollJump -= 2
}
function increaseJump() {
        scrollJump += 2
}
function swapDirection() {
        scrollDirection = (scrollDirection == "down") ? "up" : "down"
}
function startScroll() {
        parent.display.scrollBy(0, scrollJump)
        intervalID = setInterval("autoScroll()",scrollSpeed)
}
function stopScroll() {
        clearInterval(intervalID)
}
</SCRIPT>
</HEAD>
<BODY onLoad="startScroll()">
<B>AutoScroll by setInterval() Controller</B>
<FORM NAME="custom">
<INPUT TYPE="button" VALUE="Start Scrolling" onClick="startScroll()">
```

```
<INPUT TYPE="button" VALUE="Stop Scrolling" onClick="stopScroll()"><P>
<INPUT TYPE="button" VALUE="Shorter Time Interval"
onClick="reduceInterval()">
<INPUT TYPE="button" VALUE="Longer Time Interval"
onClick="increaseInterval()"><P>
<INPUT TYPE="button" VALUE="Bigger Scroll Jumps"
onClick="increaseJump()">
<INPUT TYPE="button" VALUE="Smaller Scroll Jumps"
onClick="reduceJump()"><P>
<INPUT TYPE="button" VALUE="Change Direction"
onClick="swapDirection()">

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

The `setInterval()` method is invoked inside the `startScroll()` function. This function initially "burps" the page by one `scrollJump` interval so that the test in `autoScroll()` for the page being scrolled all the way to the top doesn't halt a page from scrolling before it gets started. One of the global variables, `scrollSpeed`, is used to fill the delay parameter for `setInterval()`. To change this value on the fly, the script must stop the current interval process, change the `scrollSpeed` value, and start a new process.

The intensely repetitive nature of this application is nicely handled by the `setInterval()` method.

Related Items: `window.clearInterval()` method; `window.setTimeout()` method.

setTimeout("*functionOrExpr*", *msecDelay* [, *funcarg1*, ..., *funcargn*])

Returns: ID value for use with `window.clearTimeout()` method.

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

The name of this method may be misleading, especially if you have done other kinds of programming involving *timeouts*. In JavaScript, a timeout is an amount of time (in milliseconds) *before a stated expression evaluates*. A timeout is not a wait or script delay, but rather a way to tell JavaScript to hold off executing a statement or function for a desired amount of time.

Say that you have a Web page designed to enable users to interact with a variety of buttons or fields within a time limit (this is a Web page running at a free-standing kiosk). You can turn on the timeout of the window so that if no interaction occurs with specific buttons or fields lower in the document after, say, two minutes (120,000 milliseconds), the window reverts to the top of the document

or to a help screen. To tell the window to switch off the timeout when a user does navigate within the allotted time, you need to have any button that the user interacts with call the other side of a `setTimeout()` method — the `clearTimeout()` method — to cancel the current timer. (The `clearTimeout()` method is explained earlier in this chapter.)

The expression that comprises the first parameter of the `window.setTimeout()` method can be either a call to any function or method or a standalone JavaScript statement. The expression evaluates after the time limit expires.

Understanding that this timeout does not halt script execution is very important. In fact, if you use a `setTimeout()` method in the middle of a script, the succeeding statements in the script execute immediately; after the delay time, the statement in the `setTimeout()` method executes. Therefore, I've found that the best way to design a timeout in a script is to plug it in as the *last* statement of a function: Let all other statements execute and then let the `setTimeout()` method appear to halt further execution until the timer goes off. In truth, however, although the timeout is "holding," the user is not prevented from performing other tasks. And once a timeout timer is ticking, you cannot adjust its time. Instead, clear the timeout and start a new one.

It is not uncommon for a `setTimeout()` method to invoke the very function in which it lives. For example, if you have written a Java applet to perform some extra work for your page and you need to connect to it via LiveConnect, your scripts must wait for the applet to load and carry out its initializations. While an `onLoad=` event handler in the document ensures that the applet object is visible to scripts, it doesn't know whether the applet has finished its initializations. A JavaScript function that inspects the applet for a clue might need to poll the applet every 500 milliseconds until the applet sets some internal value indicating all is ready, as shown here:

```
var t
function autoReport() {
      if (!document.myApplet.done) {
            t = setTimeout("autoReport",500)
      } else {
            clearTimeout(t)
            [more statements using applet data]
      }
}
```

JavaScript provides no built-in equivalent for a wait command. The worst alternative is to devise a looping function of your own to trap script execution for a fixed amount of time. In Navigator 3 and up, you can also use LiveConnect (see Chapter 38) to invoke a Java method that freezes the browser's thread for a fixed amount of time. Unfortunately, both of these practices prevent other processes from being carried out, so you should consider reworking your code to rely on a `setTimeout()` method instead.

Example

When you load the HTML page in Listing 14-44, it triggers the `updateTime()` function, which displays the time (in *hh:mm am/pm* format) in the statusbar (Figure 14-16). Instead of showing the seconds incrementing one by one (which

may be distracting to someone trying to read the page), this function alternates the last character of the display between an asterisk and nothing.

Listing 14-44: **Display the Current Time**

```
<HTML>
<HEAD>
<TITLE>Status Bar Clock</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
var flasher = false
// calculate current time, determine flasher state,
// and insert time into status bar every second
function updateTime() {
        var now = new Date()
        var theHour = now.getHours()
        var theMin = now.getMinutes()
        var theTime = "" + ((theHour > 12) ? theHour - 12 : theHour)
        theTime += ((theMin < 10) ? ":0" : ":") + theMin
        theTime  += (theHour >= 12) ? " pm" : " am"
        theTime += ((flasher) ? " " : "*")
        flasher = !flasher
        window.status = theTime
        // recursively call this function every second to keep timer going
        timerID = setTimeout("updateTime()",1000)
}
//-->
</SCRIPT>
</HEAD>

<BODY onLoad="updateTime()">
</BODY>
</HTML>
```

Figure 14-16: A clock ticks in the statusbar.

In this function, the way the `setTimeout()` method works is that once the current time (including the flasher status) appears in the statusbar, the function waits one second (1,000 milliseconds) before calling the same function again. You don't have to clear the `timerID` value in this application because JavaScript does it for you every time the 1,000 milliseconds elapse.

A logical question to ask is whether this application should be using `setInterval()` instead of `setTimeout()`. This is a case in which either one does the job. To use `setInterval()` here would require that the interval process start outside of the `updateTime()` function, because you need only one process running that repeatedly calls `updateTime()`. It would be a cleaner implementation in that regard, instead of the tons of timeout processes spawned by the above listing. On the other hand, it would not run in any browsers before Navigator 4 or Internet Explorer 4, as Listing 14-44 does.

One warning about `setTimeout()` functions that dive into themselves as frequently as this one does: Each call eats up a bit more memory for the browser application in Navigator 2. If you let this clock run for a while, some users may encounter memory difficulties, depending on which operating system they're using. But considering the amount of time the typical user spends on Web pages (even if only 10 or 15 minutes), the function shouldn't present a problem. And any reloading invoked by the user (such as by resizing the window in Navigator 2) frees up memory once again.

Related Items: `window.clearTimeout()` method; `window.setInterval()` method; `window.clearInterval()` method.

stop()

Returns: Nothing.

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

Navigator 4's `stop()` method offers a scripted equivalent of clicking the Stop button in the toolbar. Availability of this method allows you to create your own toolbar on your page and hide the toolbar (in the main window with signed scripts or in a subwindow). For example, if you have an image representing the Stop button in your page, you can surround it with a link whose action stops loading, as in the following:

```
<A HREF="javascript: void stop()"><IMG SRC="myStop.gif" BORDER=0></A>
```

A script cannot stop its own document from loading, but it can stop loading of another frame or window. Similarly, if the current document dynamically loads a new image or a multimedia MIME type file as a separate action, the `stop()` method can halt that process. Even though the `stop()` method is a window method, it is not tied to any specific window or frame: Stop means stop.

Related Items: `window.back()` method; `window.find()` method; `window.forward()` method; `window.home()` method; `window.print()` method.

Event handlers

onBlur=
onFocus=

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

If knowing when a window or frame has been activated or deactivated is important to your page design, you can set event handlers that fire under those activities. For example, you can track how frequently a user switches between two of your windows over a period of time. By saving timestamps triggered by the `onBlur=` event handler in the subwindow and stamps triggered by the `onFocus=` event handler in the main window, you can create a record of the user's window activity.

You should, however, be aware of some potential side effects of scripting these events. As with their counterparts in text objects of forms, these event handlers, when asked to display JavaScript modal dialog boxes (such as the alert dialog box), can cause a nearly infinite loop, because the alert dialog box interrupts the natural action of window or frame blurring. Therefore, I recommend scripting these event handlers to perform less obvious tasks.

Another issue worth noting is that you should adhere to graphical user interface guidelines when dealing with windows. You may, for example, be tempted to close a subwindow when the user activates the main window. That design is unnatural in most GUI universes. Navigator provides a Window menu (part of the Navigator 4 menu in Navigator 4) to help the user bring forward a hidden window, and you can also script subwindows to gain focus. Regardless of how you decide to include these event handlers in your scripts, be sure to test them for inopportune clicks in the affected windows.

Related Items: `window.blur()`; `window.focus()`.

onDragDrop=

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

With closer integration between the computer desktop and browsers these days, it is increasingly possible that shortcuts (or aliases) to Web URLs will be

represented on our desktops and other kinds of documents. Beginning with Navigator 4, you can script awareness of dragging and dropping of such items onto the browser window. The window's dragDrop event fires whenever a user drops a file or other URL-filled object onto the window.

You can add an `onDragDrop=` event handler to the `<BODY>` tag of your document and pass along the event object that has some juicy tidbits about the drop: the object on which the item was dropped and the URL of the item. The function called by the event handler receives the event object information and can process it from there. Because this event is a window event, you don't have to turn on `window.captureEvents()` to get the window to feel the effect of the event.

The juiciest tidbit of the event, the URL of the dropped item, can be retrieved only with a signed script and the user's permission (see Chapter 40). Listing 14-45 shows a simple document that reveals the URL and screen location, as derived from the event objects passed with the dragDrop event. You must have codebase principals turned on to get the full advantage of this listing.

Listing 14-45: **Analyzing a DragDrop Event**

```
<HTML>
<HEAD>
<TITLE>ScrollBy Controller</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function reportDrag(e) {
        var msg = "You dropped the file:\n"

netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead")
        msg += e.data

netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserRead")
        msg += "\nonto the window object at screen location ("
        msg += e.screenX + "," + e.screenY + ")."
        alert(msg)
        return false
}
</SCRIPT>
</HEAD>
<BODY onDragDrop="reportDrag(event)">
<B>Drag and Drop a file onto this window</B>
</BODY>
</HTML>
```

The dragDrop event is the only one so far that uses the `data` property of the event object. That property contains the URL. The `target` property reveals only the window object, but you can access the event object's `screenX` and `screenY` properties to get the location of the mouse release.

Related Items: event object.

onLoad=

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

The load event is sent to the current window at the end of the document loading process (after all text and image elements have been transferred from the source file server to the browser, and after all plug-ins and Java applets have loaded and started running). At that point, the browser's memory contains all the objects and script components in the document that the browser can possibly know about.

The `onLoad=` handler is an attribute of a `<BODY>` tag for a single-frame document or of the `<FRAMESET>` tag for the top window of a multiple-frame document. When the handler is an attribute of a `<FRAMESET>` tag, the event triggers only after all frames defined by that frameset have completely loaded.

Use either of the following scenarios to insert an `onLoad=` handler into a document:

```
<HTML>
<HEAD>
</HEAD>
<BODY [other attributes] onLoad="statementOrFunction">
[body content]
</BODY>
</HTML>
```

```
<HTML>
<HEAD>
</HEAD>
<FRAMESET [other attributes] onLoad="statementOrFunction">
       <FRAME>frame specifications</FRAME>
</FRAMESET>
</HTML>
```

This handler has a special capability when part of a frameset definition: It won't fire until the `onLoad=` event handlers of all child frames in the frameset have fired. Therefore, if some initialization scripts depend on components existing in other frames, trigger them from the frameset's `onLoad=` event handler. This brings up a good general rule of thumb for writing JavaScript: Scripts that execute during a document's loading should contribute to the process of generating the document and its objects. To act immediately on those objects, design additional functions that are called by the `onLoad=` event handler for that window.

The type of operations suited for an `onLoad=` event handler are those that can run quickly and without user intervention. Users shouldn't be penalized by having to wait for considerable post-loading activity to finish before they can interact with your pages. At no time should you present a modal dialog box as part of an

onLoad= handler. Users who design macros on their machines to visit sites unattended may get hung up on a pí8 nthat automatically displays an aler t, confirm, or prompt dialog box. On th noth r hand, an operation such as setting th window.defaultStatus property is a perfect candidat nfor an onLoad= event handler.

As a reminder about a 8 0 neral ule pertaining to JavaScript event handlers, event handlers such as onLoad=, onUnload=, onBlur=, and onFocus= can b nset by assignment (window.onload = myfunction) or by creating functions of the same nam n(function onload()). The majority of your event handlers, however, will likely b ndefined as HTML tí8 attributes.

Related Items: onUnload= handler; window.defaultStatus property.

onMove=

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

If a user drags a window around th scr een, th action trig8 0 rs a mov nevent for the window object. When you assign a function to th event (for example, window.onmove = handleMoves), the function receives an event object whose screenX and screenY properties reveal th coor dinate point (relative to the entire screen) of the top-left corner of th window after th move.

Relat d Items: event object.

onLoad= bugs and anomalies

The onLoad= event has changed its b havior over th life of JavaScript in Navigator . In Navigator 2, th nonLoad= event handler fired whenever the user resized th window . Many developers considered this a bug because the running of such scripts destroyed carefully gath red data since the document orig inally loaded. From Navigator 3 onward (and including Internet Explorer 3), a window resize does not trig8 0 r a load event.

Two onLoad= bugs haunt Navigator 3 when used in conjunction with framesets. The first bug affects only Windows versions. The problem is that the frameset's onLoad= event handler is not necessarily th last on nto fi re among all the frames. It is possible that one fram 's onLoad= event may still not have processed b for nthe fram set's onLoad= event handler goes. This can cause serious problems if your frameset's onLoad= event handler relies on that final fram nb ing fully loaded.

The second bug affects all versions of Navigator 3, but at least a workaround exists. If a fram contains a Java applet, the frameset's onLoad= event handler will fir nbefore the applet has fully loaded and started. But if you place an onLoad= event handler in th applet's document (even a dummy onLoad="" in the <BODY> tí8), the fram set's onLoad= event handler b haves properly .

onResize=

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

If a user resizes a window, the action triggers a resize event for the window object. When you assign a function to the event (for example, `window.resize = handleResizes`), the function receives an event object whose `width` and `height` properties reveal the outer width and height of the entire window. A window resize in Navigator 4 does not trigger the `onLoad=` event handler (although the document content is rendered again to fill the inner window size). Therefore, if you want a script to run both when the document loads and when the user resizes the window (this is how Navigator 2 worked), you can use the `onResize=` event handler to help perform those duties.

Related Items: event object.

onUnload=

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

An unload event reaches the current window just before a document is cleared from view. The most common ways windows are cleared are when new HTML documents are loaded into them or when a script begins writing new HTML on the fly for the window or frame.

Limit the extent of the `onUnload=` event handler to quick operations that do not inhibit the transition from one document to another. Do not invoke any methods that display dialog boxes. You specify `onUnload=` event handlers in the same places in an HTML document as the `onLoad=` handlers: as a `<BODY>` tag attribute for a single-frame window or as a `<FRAMESET>` tag attribute for a multiframe window. Both `onLoad=` and `onUnload=` event handlers can appear in the same `<BODY>` or `<FRAMESET>` tag without causing problems. The `onUnload=` event handler merely stays safely tucked away in the browser's memory, waiting for the unload event to arrive for processing as the document gets ready to clear the window.

Let me pass along one caution about the `onUnload=` event handler. Even though the event fires before the document goes away, don't burden the event handler with time-consuming tasks, such as generating new objects. The document could possibly go away before the function completes, leaving the function looking for objects and values that no longer exist. The best defense is to keep your `onUnload=` event handler processing to a minimum.

Related Items: `onLoad=` handler; `window.defaultStatus` property.

Frame Object

Properties ***Methods*** ***Event Handlers***

Same as window object.

Syntax

Creating a frame:

```
<FRAMESET
        ROWS="ValueList"
        COLS="ValueList"
        [FRAMEBORDER=YES | NO]
        [BORDER=pixelSize]
        [BORDERCOLOR=colorSpecification]
        [onBlur="handlerTextOrFunction"]
        [onDragDrop="handlerTextOrFunction"]
        [onFocus="handlerTextOrFunction"]
        [onLoad="handlerTextOrFunction"]
        [onMove="handlerTextOrFunction"]
        [onResize="handlerTextOrFunction"]
        [onUnload="handlerTextOrFunction"]>
        <FRAME
               SRC="locationOrURL"
               NAME="firstFrameName"
               [FRAMEBORDER= YES | NO]
               [BORDERCOLOR=colorSpecification]
               [MARGINHEIGHT=pixelSize]
               [MARGINWIDTH=pixelSize]
               [NORESIZE]
               [SCROLLING=YES | NO | AUTO]>
        ...
</FRAMESET>
```

Accessing properties or methods of another frame:

```
parent.frameName.property | method([parameters])
parent.frames[i].property | method([parameters])
```

About this object

A frame object behaves exactly like a window object, except that it has been created as part of a frameset by another document. A frame object always has a `top` and a `parent` property different from its `self` property. If you load a document that is normally viewed in a frame into a single browser window, its window is no longer a frame. Consult the earlier discussion about the window object for details on the properties and methods these two objects share.

One other significant difference between a window and frame object occurs in the `onLoad=` and `onUnload=` event handlers. Because each document loading into a frame may have its own `onLoad=` event handler defined in its `<BODY>` definition, the frame containing that document receives the load event after the individual document (and its components) finishes loading. But the frameset that governs the frame receives a separate load event after all frames have finished loading their documents. That event is captured in the `onLoad=` event handler of the `<FRAMESET>` definition (but see specific buggy behavior in the `onLoad=` event handler discussion, earlier in this chapter). The same applies to the `onUnload=` event handlers defined in `<BODY>` and `<FRAMESET>` definitions. Navigator 3 and later provide an easy way to view the source code for a document loaded in a frame. Click anywhere in the frame to select it (if a border is defined for the frame, it will highlight subtly), and then select Frame Source from the View menu.

✦ ✦ ✦