Navigator and Other Environment Objects

25
CHAPTER

In This Chapter

Determining which browser the user has

Branching scripts according to the user's operating system

Verifying that a specific plug-in is installed

JavaScript devotes one category of its objects to browser software (also called the client by some users). These objects are not part of any windows or documents that you see, but rather apply to the inner workings of the browser software.

In addition to providing some of the same information that CGI programs on the server extract as environment variables, these browser-level objects also include information about how well-equipped the browser is with regard to plug-ins and Java. And one object newly defined for Navigator 4 reveals information about the user's video monitor, which may influence the way your scripts define information displayed on the page.

The objects in this chapter don't show up on the JavaScript object hierarchy diagrams except as a free-standing group derived from the navigator and screen objects (see Appendix A). Internet Explorer 4's object model, however, places its corresponding objects under the window object. Because the `window` reference is optional, you can omit it for Internet Explorer 4 and wind up with a cross-compatible script. With the added built-in capabilities that each generation of browser introduces, these small independent objects are sure to grow even more in importance.

Navigator Object

Properties	*Methods*	*Event Handlers*
`appName`	`javaEnabled()`	(None)
`appCodeName`	`preference()`	
`appVersion`	`taintEnabled()`	
`language`		
`mimeTypes[]`		
`platform`		
`plugins[]`		
`userAgent`		

Syntax

Accessing navigator object properties and methods:

```
navigator.property | method()
```

About this object

In Chapter 14, I repeatedly mentioned that the window object is the top banana of the JavaScript object hierarchy. In other programming environments, you will likely find an application level higher than the window. You may think that an object known as the navigator object would be that all-encompassing object. That is not the case, however.

Although Netscape originally defined the navigator object for the Navigator 2 browser, Microsoft Internet Explorer also supports the object in its object model. Despite the object name seeming to be tied to a specific commercial browser, other software vendors appear to find the object vocabulary an acceptable standard among their JavaScript implementations.

The properties of the navigator object deal with the browser program the user runs to view documents. Properties include those for extracting the version of the browser and the platform of the client running the browser. Because so many properties of the navigator object are similar, I treat four of them together.

Properties

appName

appCodeName

appVersion

userAgent

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

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

The best way to see what these properties hold is to view them from two different versions of Navigator 4: The Windows 95 version and the Macintosh version (Figure 25-1).

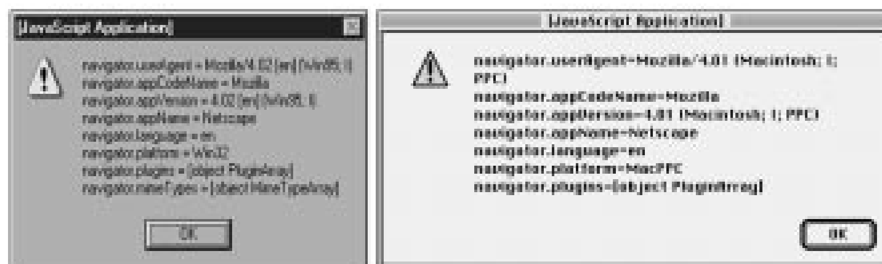


Figure 25-1: Property dumps for the navigator object under Windows 95 (left) and Macintosh (right) versions of Navigator 4

The `appName` and `appCodeName` properties are simply the official name and the internal code name for the browser application. For Netscape browsers, the `appName` value is `Netscape`; for Internet Explorer, the value is `Microsoft Internet Explorer`. Both browsers bear the Netscape code name, `Mozilla`, as the `appCodeName` property.

The `appVersion` and `userAgent` properties provide more meaningful detail. They contain not only the browser's version number, but also data about the platform of the browser and the country for which the browser is released (in Navigator, "I" stands for "international," which has a less rigid encryption feature built into it than the "U" version for the United States). It seems that with each generation of browser, more information is revealed in the `userAgent` property. For example in Navigator 4, a code for the localized language of the browser is also part of the property. The `userAgent` property is a string similar to the

USER_AGENT header that the browser sends to the server at certain points during the connection process between client and server.

Because the userAgent property reveals more information about a visitor's browser than other properties, viewing a selection of such values from many generations and types of browsers will help you learn how it works. Table 25-1 shows some of the values that your scripts are likely to see. This table does not include, of course, the many values that would not be reflected by browsers that do not support JavaScript.

Table 25-1
Typical navigator.userAgent Values

navigator.userAgent	***Description***
Mozilla/4.02 [en] (Win95; I; Nav)	Navigator-only version of Communicator 4.02, English edition for Windows 95, and export encryption
Mozilla/4.01 [fr] (Win95; I)	Navigator 4.01, French edition for Windows 95, export encryption
Mozilla/4.01 [en] (WinNT; U)	Navigator 4.01, English edition for Windows NT with U.S. encryption
Mozilla/4.01 (Macintosh; I; 68K)	Navigator 4.01 for 68xxx processor on Macintosh
Mozilla/4.0 (compatible; MSIE 4.0b2; Windows NT)	Internet Explorer 4.0b2 (Preview 2) on via proxy gatewayCERN-HTTPD/3.0 libwww/2.17 Windows NT; accessed via a gateway
Mozilla/3.01Gold (Win95; I)	Navigator 3.01 Gold for Windows 95
Mozilla/3.01 (Macintosh; I; PPC)	Navigator 3.01 for PowerPC Macintosh
Mozilla/3.01 (X11; I; HP-UX A.09.05 9000/720)	Navigator 3.01 for HP-UX on RS-9000
Mozilla/3.01 (X11; I; IRIX 6.2 IP22)	Navigator 3.01 for IRIX
Mozilla/3.01 (X11; I; SunOS 5.4 sun4m)	Navigator 3.01 for SunOS 5.4
Mozilla/3.01-C-MACOS8 (Macintosh; I; PPC)	Navigator 3.01 running on a MacOS8-equipped PowerPC Macintosh

navigator.userAgent	*Description*
Mozilla/3.01Gold [de] (Win16; I)	Navigator 3.01, German edition for Windows 3.0x
Mozilla/3.0 (Win95; I)	Navigator 3 for Windows 95
Mozilla/3.0 WebTV/1.2 (compatible; MSIE 2.0)	IE 2 built into a WebTV box, emulating Navigator 3 (its scripting compatibility with Navigator 3 is in question)
Mozilla/3.0 (compatible; MSIE 3.01; Navigator 3	IE 3.01 on a PowerPC Macintosh, emulating (which it does better than Windows versions) Mac_PowerPC)
Mozilla/2.0 (compatible; MSIE 3.0; AOL 3.0; Windows 3.1)	IE 3 (version for America Online software Version 3) for Windows 3.1, emulating Navigator 2
Mozilla/2.0 (compatible; MSIE 3.02; Update a; Windows 95)	IE 3.02, Update a for Windows 95, emulating Navigator 2
Mozilla/2.0 (compatible; MSIE 3.01; Windows 95)	IE 3.01 for Windows 95, emulating Navigator 2
Mozilla/2.0 (compatible; MSIE 3.0B; Windows NT)	IE 3 (beta) emulating Navigator 2

Because you will mostly call upon these properties to define a special case for a particular browser version or platform, choose the property that most easily provides the information you need to extract. For example, as you can see from Table 25-1, assuming that all `userAgent` values starting with "Mozilla" are Netscape browsers would be wrong. The `appName` property is a better choice to examine for this browser distinction.

Be careful about values for `navigator.appVersion`, especially with regard to what we know as Internet Explorer 3. If you look through Table 25-1, you will see that Internet Explorer 3's `userAgent` string reveals itself to be a Navigator 2 compatible. The `navigator.appVersion` string for such versions of Internet Explorer starts with the `userAgent` string after the Mozilla/designation. Therefore, if your scripts for distinguishing browser versions look only at the first character of the `navigator.appVersion` value to obtain the browser generation, the script might think it is working with Internet Explorer 2, rather than 3.0*x*. In truth, the Windows versions of Internet Explorer 3 are closer to the scriptability of Navigator 2 than to Navigator 3, so the value returned from the property is an honest self-appraisal. Still, if your scripts rely on knowing for sure that the version is Internet Explorer 3.0*x*, you should use the `string.indexOf()` method to look for the presence of the unique string `"MSIE 3"` in either the `navigator.appVerion` or `navigator.userAgent` values.

Speaking of compatibility and browser versions, the question often arises whether your scripts should distinguish among incremental releases within a browser's generation (for example, 3.0, 3.01, 3.02, and so on). The latest incremental release occasionally contains bug fixes and (rarely) new features that you may rely on. If that is the case, then I suggest looking for this information when the page loads and recommend to the user that he or she download the latest browser version. Beyond that, I suggest scripting for the latest version of a given generation, and don't bother branching for incremental releases.

Even scripting for only the browser generation can get you in trouble if you fail to think ahead. It was common, for example, for scripted pages to look at the first character of the `navigator.appVersion` value to determine if the visitor had the latest browser:

```
if (navigator.appVersion.charAt(0) == "3") {
        do scripted stuff
}
```

What this tactic misses is the inevitable evolution of browser versions. When someone visits with generation 4, the scripting is ignored. It is better to test for the minimum version by turning that character into a number and comparing accordingly:

```
if (parseInt(navigator.appVersion.charAt(0)) >= 3) {
        do scripted stuff
}
```

See Chapter 13 for more information about designing pages for cross-platform deployment.

Example

Listing 25-1 provides a number of reusable functions that your scripts can use to determine a variety of information about the currently running browser. All functions return a Boolean value in line with the pseudo-question presented in the function's name. For example, the `isWindows()` function returns true if the browser is any type of Windows browser and false if it's not (in Internet Explorer 3, the values are zero for false and -1 for true, but those values are perfectly usable in `if` condition phrases). Some functions extract one or more characters from `navigator` properties to determine the browser's version number. If this kind of browser detection occurs frequently in your pages, consider moving these functions into an external .js source library for inclusion in your pages (see Chapter 13).

When you load this page, it presents fields that display the results of each function depending on the type of browser and client operating system you are using at the time.

Listing 25-1: Functions to Examine Browsers

```
<HTML>
<HEAD>
<TITLE>UserAgent Property Library</TITLE>
<SCRIPT LANGUAGE="JavaScript">
```

```
function isWindows() {
        return (navigator.appVersion.indexOf("Win") != -1)
}

function isWin95NT() {
        return (isWindows() && (navigator.appVersion.indexOf("Win16")
== -1 && navigator.appVersion.indexOf("Windows 3.1") == -1))
}

function isMac() {
        return (navigator.appVersion.indexOf("Mac") != -1)
}

function isMacPPC() {
        return (isMac() && (navigator.appVersion.indexOf("PPC") != -1
|| navigator.appVersion.indexOf("PowerPC") != -1))
}

function isUnix() {
        return (navigator.appVersion.indexOf("X11") != -1)
}

function isNav() {
        return (navigator.appName == "Netscape")
}

function isGeneration2() {
        return (navigator.appVersion.charAt(0) == "2")
}

function isGeneration3() {
        return (navigator.appVersion.charAt(0) == "3")
}

function isGeneration3Min() {
        return (parseInt(navigator.appVersion.charAt(0)) >= 3)
}
function isNav4_02() {
        return (isNav() && (navigator.appVersion.substring(0,4) ==
"4.02") )
}

function isMSIE3Min() {
        return (navigator.appVersion.indexOf("MSIE") != -1)
}

function checkBrowser() {
        var form = document.forms[0]
        form.brand.value = isNav()
        form.win.value = isWindows()
        form.win32.value = isWin95NT()
        form.mac.value = isMac()
```

(continued)

Listing 25-1 *(continued)*

```
        form.ppc.value = isMacPPC()
        form.unix.value = isUnix()
        form.ver3Only.value = isGeneration3()
        form.ver3Up.value = isGeneration3Min()
        form.Nav4_02.value = isNav4_02()
        form.MSIE3.value = isMSIE3Min()
}
</SCRIPT>
</HEAD>

<BODY onLoad="checkBrowser()">
<H1>About This Browser</H1>
<FORM>
<H2>Brand</H2>
Netscape Navigator:<INPUT TYPE="text" NAME="brand" SIZE=5>
<HR>
<H2>Platform</H2>
Windows:<INPUT TYPE="text" NAME="win" SIZE=5>
Windows 95/NT:<INPUT TYPE="text" NAME="win32" SIZE=5><P>
Macintosh: <INPUT TYPE="text" NAME="mac" SIZE=5>
Mac PPC:<INPUT TYPE="text" NAME="ppc" SIZE=5><P>
Unix:<INPUT TYPE="text" NAME="unix" SIZE=5><P>
<HR>
<H2>Version</H2>
3.0x Only:<INPUT TYPE="text" NAME="ver3Only" SIZE=5><P>
3 or Later: <INPUT TYPE="text" NAME="ver3Up" SIZE=5><P>
Navigator 4.02: <INPUT TYPE="text" NAME="Nav4_02" SIZE=5><P>
MSIE 3 or Later: <INPUT TYPE="text" NAME="MSIE3" SIZE=5><P>
</FORM>
</BODY>
</HTML>
```

Sometimes you may need to use more than one of these functions together. For example, if you want to create a special situation for the `window.open()` bug that afflicts UNIX and Macintosh versions of Navigator 2, then you have to put your Boolean operator logic powers to work to construct a fuller examination of the browser:

```
function isWindowBuggy() {
        return (isGeneration2() && (isMac() || isUnix()))
}
```

Related Items: None.

language

Value: Two-character String **Gettable:** Yes **Settable:** No

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

Navigator 4 and later include a navigator property that reflects the identifier for a localized language version of the program (it has nothing to do with scripting or programming language). These short strings resemble but are not identical to the URL suffixes for countries. In the `navigator.userAgent` property value, the language appears inside brackets, but the raw property value is without the brackets. Table 25-2 shows a sampling of languages and their designations.

Table 25-2
Sample navigator.language Values

navigator.language	*Language*
`en`	English
`de`	German
`es`	Spanish
`fr`	French
`ja`	Japanese
`da`	Danish
`it`	Italian
`ko`	Korean
`nl`	Dutch
`pt`	Brazilian Portuguese
`sv`	Swedish

The assumption you can make is that a user of a particular language version of Navigator will also be interested in content in the same language. If your site offers multiple language paths, then you can use this property setting to automate the navigation to the proper section for the user.

Related Items: `navigator.userAgent` property.

mimeTypes[]

Value: Array of MIME types **Gettable:** Yes **Settable:** No

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

A MIME (*Multipurpose Internet Mail Extension*) type is a file format for information that travels across the Internet. Browsers usually have a limited capability for displaying or playing information beyond HTML text and one or two image standards (.gif and .jpg are the most common formats). To fill in the gap, browsers maintain an internal list of MIME types with corresponding instructions on what to do when information of a particular MIME type arrives at the client. For example, when a CGI program serves up an audio stream in an audio format, the browser locates that MIME type in its table (the MIME type is among the first chunk of information to reach the browser from the server) and then launches a helper application or activates a plug-in capable of playing that MIME type. Your browser is not equipped to display every MIME type, but it does know how to alert you when you don't have the helper application or plug-in needed to handle an incoming file. For instance, the browser may ask if you want to save the file for later use or switch to a Web page containing more information about the necessary plug-in.

The `mimeTypes[]` property of the navigator object is simply the array of MIME types about which your browser knows (see "MimeType Object" later in this chapter). Navigator 3 and up come with dozens of MIME types already listed in their tables (even if the browser doesn't have the capability to handle all those items automatically). If you have third-party plug-ins in Navigator's plug-ins directory/folder or helper applications registered with Navigator, that array contains these new entries as well.

If your Web pages are media-rich, you likely will want to be sure that each visitor's browser is capable of playing the media your page has to offer. With JavaScript and Navigator, you can cycle through the `mimeTypes[]` array to find a match for the MIME type of your media. Then use the properties of the mimeType object (detailed later in this chapter) to ensure the optimum plug-in is available. If your media still requires a helper application instead of a plug-in, the array only lists the MIME type; thus, you won't be able to determine whether a helper application has been assigned to this MIME type from the array list.

Example

For examples of this property and details about using the mimeType object, see the discussion of the object later in this chapter. A number of simple examples showing how to use this property to see whether the navigator object has a particular MIME type do not go far enough to determine whether a plug-in is installed and enabled to play the incoming data.

Related Items: `navigator.plugin[]` property; `mimeType` object.

platform

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

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

Another part of the `navigator.userAgent` is retrievable as a separate entity starting with Navigator 4. The `navigator.platform` value reflects the operating system according to the codes established by Netscape for its `userAgent` values. Table 25-3 lists values of the most popular operating systems.

In the long list of browser detection functions in Listing 25-1, I elected not to use the `navigator.platform` property because it is limited to Navigator 4 and later, while the other properties in that listing are available to all scriptable browsers.

Table 25-3
Sample navigator.platform Values

navigator.platform	*Operating System*
Win95	Windows 95
WinNT	Windows NT
Win16	Windows 3.*x*
Mac68k	Mac (680x0 CPU)
MacPPC	Mac (PowerPC CPU)
SunOS	Solaris

Notice that the `navigator.platform` property does not go into versioning of the operating system. Only the raw name is provided.

Related Items: `navigator.userAgent` property.

plugins[]

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

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

You rarely find users involved with Web page design for Navigator who have not heard about *plug-ins* — the technology that enables you to embed new media types and foreign file formats directly into Web documents. For instance, instead of requiring you to view a video clip in a separate window atop the main browser window, a plug-in enables you to make that viewer as much a part of the page design

as a static image. The same goes for audio players, 3-D animation, chat sessions — even the display of Microsoft Office documents, such as PowerPoint and Word.

Whenever you launch Navigator, you probably watch the status of its loading process in the splash screen. One of the messages that appears is "Registering Plug-ins." During that brief moment, Navigator creates its built-in list of available plug-ins to include all the plug-in files contained in a special directory/folder (the name varies with the operating system, but is obvious). The items registered at launch time are the ones listed in the `navigator.plugins[]` array. Each plug-in is, itself, an object with several properties.

Being able to have your scripts investigate the visitor's browser for a particular installed plug-in is a valuable capability if you want to guide the user through the process of downloading and installing a plug-in, if the system does not currently have it.

Example

For examples of this property and for details about using the plugin object, see "Plugin Object" later in this chapter.

Related Items: `navigator.mimeTypes[]` property; plugin object.

Methods

`javaEnabled()`

Returns: Boolean.

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

Although Navigator and Internet Explorer 4 ship with Java support turned on, a user can easily turn it off in a preferences dialog box. Some corporate installations may also turn off Java as the default setting for their users. If your pages specify Java applets, you don't normally have to worry about this property, because the applet tag's alternate text fills the page in the places where the applet would normally go. But if you are scripting applets from JavaScript (via LiveConnect, Chapter 38), you won't want your scripts making calls to applets or Java classes if you have Java support turned off. In a similar vein, if you are creating a page with JavaScript, you can fashion two quite different layouts, depending on whether Java is available.

Navigator's `javaEnabled()` method returns a Boolean value reflecting the Preferences setting. This value does not necessarily reflect Java support in the browser, but rather whether Java is turned on inside a JavaScript 1.1–level or later browser. A script cannot change the `navigator` setting, but its value does change immediately upon toggling the Preference setting.

Related Items: `navigator.preference()` method; LiveConnect (Chapter 38).

preference(*name* [, *val*])

Returns: Preference Value.

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

Browser preferences are normally set by the user. Until Navigator 4 and the advent of signed scripts, almost all of their settings had been completely out of view of scripts, even when it might make sense to expose them. But with signed scripts and the `navigator.preference()` method, many preferences are now viewable and settable with the user's permission. These preferences were exposed to scripting primarily for the purposes of centralized configuration administration for enterprise installations. I don't recommend altering a public Web site visitor's browser preferences, even if given permission to do so — the user may not know how much trouble you can cause.

When you want to read a particular preference setting, you pass only the preference name parameter with the method. Reading a preference requires a signed script with the target of `UniversalPreferencesRead` (see Chapter 40). To change a preference, pass both the preference name and the value (with a signed script target of `UniversalPreferencesWrite`).

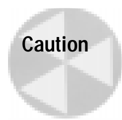
Table 25-4 shows a handful of scriptable preferences in Navigator 4 (learn more about these settings at `http://developer.netscape.com/library/documentation/deploymt/jsprefs.htm`). Each item has a corresponding entry in the preferences window in Navigator 4 (shown in parentheses). Notice that the preference name uses dot syntax that mimics the hierarchical structure of the Navigator 3 preferences menu, rather than the Navigator 4 preferences organization. The cookie security level is a single preference value with a matrix of integer values indicating the level.

Table 25-4
navigator.preference Values

navigator.preference	*Value*	*Preference Dialog Listing*
`general.always_load_images`	Boolean	(Advanced) Automatically load images
`security.enable_java`	Boolean	(Advanced) Enable Java
`javascript.enabled`	Boolean	(Advanced) Enable JavaScript
`browser.enable_style_sheets`	Boolean	(Advanced) Enable style sheets
`autoupdate.enabled`	Boolean	(Advanced) Enable AutoInstall

(continued)

navigator.preference	Value	Preference Dialog Listing
network.cookie.cookieBehavior	0	(Advanced) Accept all cookies
network.cookie.cookieBehavior	1	(Advanced) Accept only cookies that get sent back to the originating server
network.cookie.cookieBehavior	2	(Advanced) Disable cookies
network.cookie.warnAboutCookies	Boolean	(Advanced) Warn me before accepting a cookie



One preference to watch out for is the one that disables JavaScript. If you disable JavaScript, the only way for JavaScript to be reenabled is by the user manually changing the setting in his or her Navigator preferences dialog box.

Example

The page in Listing 25-2 displays checkboxes for each of the preferences listed in Table 25-4. To run this script without signing the scripts, turn on codebase principals, as directed in Chapter 40.

One function reads all the preferences and sets the checkbox values accordingly. Another function sets a preference when you click its checkbox. Because of the interaction among three of the cookie settings, it is easier to have the script rerun the `showPreferences()` function after each setting, rather than trying to manually control the properties of the three checkboxes. Rerunning that function also helps verify that the preference was set.

Listing 25-2: Reading and Writing Browser Preferences

```
<HTML>
<HEAD>
<TITLE>Reading/Writing Browser Preferences</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function setPreference(pref, value) {

netscape.security.PrivilegeManager.enablePrivilege("UniversalPreference
sWrite")
        navigator.preference(pref, value)

netscape.security.PrivilegeManager.disablePrivilege("UniversalPreferenc
esWrite")
        showPreferences()
}

function showPreferences() {
        var form = document.forms[0]

netscape.security.PrivilegeManager.enablePrivilege("UniversalPreference
sRead")
```

```
        form.imgLoad.checked =
navigator.preference("general.always_load_images")
        form.javaEnable.checked =
navigator.preference("security.enable_java")
        form.jsEnable.checked =
navigator.preference("javascript.enabled")
        form.ssEnable.checked =
navigator.preference("browser.enable_style_sheets")
        form.autoIEnable.checked =
navigator.preference("autoupdate.enabled")
        var cookieSetting =
navigator.preference("network.cookie.cookieBehavior")
        for (var i = 0; i < 3; i++) {
           form.elements["cookie" + i].checked = (i == cookieSetting) ?
true : false
        }
        form.cookieWarn.checked =
navigator.preference("network.cookie.warnAboutCookies")

netscape.security.PrivilegeManager.disablePrivilege("UniversalPreferenc
esRead")
}
</SCRIPT>
</HEAD>

<BODY onLoad="showPreferences()">
<B>Browser Preferences Settings</B>
<HR>
<FORM>
<INPUT TYPE="checkbox" NAME="imgLoad"
onClick="setPreference('general.always_load_images',this.checked)">Auto
matically Load Images<BR>
<INPUT TYPE="checkbox" NAME="javaEnable"
onClick="setPreference('security.enable_java',this.checked)">Java
Enabled<BR>
<INPUT TYPE="checkbox" NAME="jsEnable"
onClick="setPreference('javascript.enabled',this.checked)">JavaScript
Enabled<BR>
<INPUT TYPE="checkbox" NAME="ssEnable"
onClick="setPreference('browser.enable_style_sheets',this.checked)">Sty
le Sheets Enabled<BR>
<INPUT TYPE="checkbox" NAME="autoIEnable"
onClick="setPreference('autoupdate.enabled',this.checked)">AutoInstall
Enabled<BR>
<INPUT TYPE="checkbox" NAME="cookie0"
onClick="setPreference('network.cookie.cookieBehavior',0)">Accept All
Cookies<BR>
<INPUT TYPE="checkbox" NAME="cookie1"
onClick="setPreference('network.cookie.cookieBehavior',1)">Accept Only
Cookies Sent Back to Server<BR>
<INPUT TYPE="checkbox" NAME="cookie2"
onClick="setPreference('network.cookie.cookieBehavior',2)">Disable
Cookies<BR>
```

(continued)

Listing 25-2 *(continued)*

```
<INPUT TYPE="checkbox" NAME="cookieWarn"
onClick="setPreference('network.cookie.warnAboutCookies',this.checked)"
>Warn Before Accepting Cookies<BR>
</FORM>
</BODY>
</HTML>
```

Related Items: `navigator.javaEnabled()` method.

taintEnabled()

Returns: Boolean.

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

Navigator 3 featured a partially implemented security feature called *data tainting,* which was turned off by default. This feature has been replaced by signed scripts, but for the sake of backward compatibility, the `navigator.taintEnabled()` method is available in more modern browsers that don't employ tainting (in which case the method returns false). Do not employ this method in your scripts.

MimeType Object

Properties	*Methods*	*Event Handlers*
`description`	(None)	(None)
`enabledPlugin`		
`type`		
`suffixes`		

Syntax

Accessing `mimeType` properties:

navigator.mimeTypes[*i*].*property*

About this object

A browser's mimeType object is essentially an entry in the internal array of MIME types about which the browser knows. Navigator 3 and later, for example, ships with an internal list of more than five dozen MIME types. Only a handful of these types are associated with helper applications or plug-ins. But add to that plug-ins and other helpers you've added over time, and the number of MIME types can grow to more than a hundred.

The MIME type for the data is usually among the first bits of information to arrive at a browser from the server. A MIME type consists of two pieces of information: type and subtype. The traditional way of representing these pieces is as a pair separated by a slash, as in

```
text/html
image/gif
audio/wav
audio/x-midi
video/quicktime
application/x-zip-compressed
```

If a file does not contain the MIME type "header" (or a CGI program sending the file does not precede the transmission with the MIME type string), the browser receives the data as a `text/plain` MIME type. When the file is loaded from a local hard drive, the browser looks to the filename's extension (the suffix after the period) to figure out the file's type.

Regardless of the way it determines the MIME type of the incoming data, the browser then acts according to instructions it maintains internally. You can see these settings by looking at the Applications or Helpers preference settings (depending on which version of Navigator you use). In Navigator 4, a click on a file type description reveals the extension and MIME type registered for that type, as well as whether the file is processed by an application or a plug-in.

By having the mimeType object available to JavaScript, your page can query a visitor's browser to discover not only whether it has a particular MIME type listed currently, but ultimately whether the browser has a corresponding plug-in installed and enabled. In such queries, the mimeType and plugin objects work together to help scripts make these determinations.

Because of the close relationships between mimeType and plugin objects, I save the examples of using these objects and their properties for the end of the chapter. There, you can see how to build functions into your scripts that enable you to examine how well a visitor's Netscape browser is equipped for either a MIME type or data that requires a specific plug-in (Internet Explorer's JScript implementation does not provide facilities for examining this information). In the meantime, be sure that you understand the properties of both objects.

Properties

description

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

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

While registering themselves with the browser at launch time, plug-ins can provide the browser with an extra field of information: a plain-language description of the plug-in. For example, the Microsoft video plug-in, whose MIME types are video/msvideo and video/x-msvideo, supplies the following description field:

```
Video for Windows
```

When you select About Plug-ins from the Help menu, you can see the descriptions of the various installed plug-ins. This information is useful primarily for the kind of display you see in About Plug-ins, rather than as a way for your scripts to compare values.

When a MIME type does not have a plug-in associated with it (either no plug-in is installed or a helper application is used instead), you often see the `type` property repeated in the description field.

Related Items: None.

enabledPlugin

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

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

The descriptions of the mimeType and plugin objects seem to come full circle when you reach the `mimeType.enabledPlugin` property, the reason being that the property is a vital link between a known MIME type and the plug-in that the browser engages when data of that type arrives.

Knowing which plug-in is associated with a MIME type becomes very important when you have more than one plug-in capable of playing a given MIME type. For example, the Crescendo MIDI audio plug-in can take the place of the default LiveAudio plug-in if you set up your browser that way. Therefore, all MIDI data streams are played through the Crescendo plug-in. If you prefer to have your Web page's MIDI sound played only through LiveAudio, your script needs to know which plug-in is set to receive your data and perhaps alert the user accordingly. These kinds of conflicts are not common (at least at this early stage of plug-in development), because each plug-in developer with a new type of data tries to choose a MIME type that is unique. But you have no guarantee of such uniqueness even today, so a careful check of MIME type and plug-in is highly recommended if you want your page to look professional.

The `enabledPlugin` property evaluates to a plugin object. Therefore, you can dig a bit deeper with this information to fetch the `name` or `filename` properties of a

plug-in directly from a mimeType object. Go through the following steps to see how this all works (Windows users with Navigator 4 should use Version 4.02 or later):

1. Choose Open Location or Open Page from the File menu and enter

```
javascript:
```

2. In the "javascript typein" field, enter

```
navigator.mimeTypes[0].type
```

   You then see the MIME type and subtype for the first entry in your browser's `mimeTypes[]` array (the exact item varies depending on the items registered in your browser).

3. Edit the typein field to read

```
navigator.mimeTypes[0].enabledPlugin
```

   This statement returns a plugin object for the first entry in the `mimeTypes[]` array.

4. Add the name property to the end, so that the field reads

```
navigator.mimeTypes[0].enabledPlugin.name
```

The result is the name of the plug-in that your browser invokes whenever it receives data of the type shown as the result of step 2.

Example

See "Looking for MIME and Plug-ins" later in this chapter.

Related Items: None.

type

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

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

A mimeType object's `type` property is the combination of the type and subtype commonly used to identify the kind of data coming from the server. CGI programs, for example, typically precede a data transmission with a special header string in the following format:

```
Content-type: <type>/<subtype>
```

This string prompts a browser to look up how to treat an incoming data stream of this kind. As you see later in this chapter, knowing whether a particular MIME type is

listed in the `navigator.mimeTypes[]` array is not enough. A good script must dig deeper to uncover additional information about what is truly available for your data.

Example

See "Looking for MIME and Plug-ins" later in this chapter.

Related Items: `description` property.

suffixes

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

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

Every MIME type has one or more filename extensions, or suffixes, associated with it. You can read this information for any mimeType object via the `suffixes` property. The value of this property is a string. If the MIME type has more than one suffix associated with it, the string contains a comma-delimited listing, as in

```
mpg, mpeg, mpe
```

Multiple versions of a suffix have no distinction between them. Those MIME types that are best described in four or more characters (derived from a meaningful acronym, such as mpeg) have three-character versions to accommodate the "8-dot-3" filename conventions of MS-DOS and its derivatives.

Example

See "Looking for MIME and Plug-ins" later in this chapter.

Related Items: None.

Plugin Object

Properties	*Methods*	*Event Handlers*
`name`	`refresh()`	(None)
`filename`		
`description`		
`length`		

Syntax

Accessing plug-in properties or method:

```
navigator.plugins[i].property | method()
```

About this object

Understanding the distinction between the data embedded in documents that summon the powers of plug-ins and those items that browsers consider to be plug-ins is important. The former are made part of the document object by way of `<EMBED>` tags. If you want to control the plug-in via LiveConnect, you can gain access through the `document.embedName` object (see Chapter 38).

The concern here, however, deals with the way the plug-ins work from the browser's perspective: The software items registered with the browser at launch time stand ready for any matching MIME type that comes from the Net. One of the main purposes of having these objects scriptable is to let your scripts determine whether a desired plug-in is currently registered with the browser and even to help with installing a plug-in.

The close association between the plugin and mimeType object, demonstrated by the `mimeType.enabledPlugin` property, is equally visible coming from the direction of the plug-in. A plugin object evaluates to an array of MIME types that the plug-in interprets. Let's experiment to make this association clear (Windows users with Navigator 4 should use Version 4.02 or later):

1. Select Open Location from the File menu and enter

```
javascript:
```

2. In the "javascript typein" field, enter

```
navigator.plugins["LiveAudio"].length
```

   Instead of the typical index value for the array notation, you use the actual name of the LiveAudio plug-in. This expression evaluates to 7, meaning that the `navigator.plugins["LiveAudio"]` array entry contains an array of seven items — the MIME types it recognizes.

3. Edit the typein field to read

```
navigator.plugins["LiveAudio"][0].type
```

That's not a typo: Because one array evaluates to a different array, the second set of square brackets is not separated from the first set by a period. In other words, this statement evaluates to the `type` property of the first mimeType object contained by the LiveAudio plug-in.

I doubt that you will have to use this kind of construction much, because if you know the name of the desired plug-in, you know what MIME types it already supports. In most cases, you come at the search from the MIME type direction and look for a specific enabled plug-in. See "Looking for MIME and Plug-ins" later for details on how to use the plug-in object in a production setting.

Properties

name

filename

description

length

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

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

The first three properties of the plugin object provide descriptive information about the plug-in file. The plug-in developer supplies the name and description. It's unclear whether future versions of plug-ins will differentiate themselves from earlier ones via either of these fields. That may be important if you are transferring data to a plug-in that requires a later version of the software. In the meantime, however, these items can play a role in helping a site visitor understand that a plug-in other than the ideal one (for a given MIME type) currently has an enabled setting.

Be aware that along the way, Netscape forgot to tell plug-in authors in the early days to assign the same name to every platform version of a plug-in. Be prepared for discrepancies across platforms.

Another piece of information available from a script is the plug-in's filename. On some platforms, such as Windows, this data comes in the form of a complete pathname to the plug-in DLL file; on other platforms, only the plug-in file's name appears.

Finally, the `length` property of a plugin object counts the number of MIME types that the plug-in recognizes. Although you could use this information to loop through all possible MIME types for a plug-in, a more instructive way would probably be to have your scripts approach the issue via the MIME type, as discussed later in this chapter.

Example

See "Looking for MIME and Plug-ins" later in this chapter.

Related Items: `mimeType.description` property.

Methods

refresh()

Returns: Nothing.

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

You may have guessed that Navigator determines its list of installed plug-ins while it launches. If you were to drop a new plug-in file into the plug-ins directory/folder, you would have to quit Navigator and relaunch it before it would see the new plug-in file. But that isn't a very friendly approach if you take pains to guide a user through the downloading and installation of a new plug-in file. The minute the user quits the browser, you have a slim chance of getting that person right back. That's where the `refresh()` method comes in.

The `refresh()` method is directed primarily at the browser, but the syntax of the call reminds the browser to refresh just the plug-ins:

```
navigator.plugins.refresh()
```

Interestingly, this command works only for adding a plug-in to the existing collection. If the user removes a plug-in and invokes this method, the removed one stays in the `navigator.plugins[]` array, although it may not be available for use. Only quitting and relaunching Navigator makes a plug-in removal take full effect.

I think it's a good idea to have a routine ready that leads a user through downloading and installing a plug-in not normally delivered with Navigator. Doing so through a separate window or a branch of your Web site's organization is ideal. Always include a `refresh()` method before returning to the point where the plug-in is required for your special media.

Example

If you want to experiment with this method, follow this sequence:

1. Move one of the plug-in files from the Navigator plug-ins directory/folder (whatever it may be called on your operating system) to another directory/folder.
2. Quit and restart Navigator.
3. Select About Plug-ins from the Help menu and make note of the plug-ins listed in this screen.
4. Move the plug-in file back to the plug-ins directory/folder.
5. Open Location `javascript:`
6. In the "javascript typein" field, type

   ```
   navigator.plugins.refresh()
   ```

7. Select About Plug-ins again from the Help menu.

The restored plug-in should now appear as part of the list. Starting with Navigator 4, plug-ins can be constructed to take advantage of automatic installation via the SmartUpdate technique. If you are a plug-in developer, consult Netscape's developer Web site (`http://developer.netscape.com`) for details on

adding this feature to your plug-in. SmartUpdate obviates the need for MIME type or plug-in checking as well as the `refresh()` method for visitors using Navigator 4.

Related Items: None.

Looking for MIME and Plug-ins

If you go to great lengths to add new media and data types to your Web pages, then you certainly want your visitors to reap the benefits of those additions. But you cannot be guaranteed that they have the requisite plug-ins installed to accommodate that fancy data. Fortunately, if your audience is dependent on Navigator 3 (and not Navigator 4 with its SmartUpdate auto-installation of plug-ins), you can use JavaScript to inspect the browser to find out if the desired software is ready for your data.

The value of this inspection capability is that you can maintain better control of your site visitors who don't yet have the necessary plug-in in Navigator 3. Rather than merely providing a link to the plug-in's download site, you can build a more complete interface around the downloading and installation of the plug-in, without losing your visitor. I have some suggestions about such an interface at the end of this discussion.

How you go about inspecting a visitor's plug-in library depends on what information you have about the data file or stream and how precise you must be in locating a particular plug-in. Some plug-ins may override MIME type settings that you would expect to be in a browser. For example, a new audio plug-in may take over for LiveAudio when it's installed by the user (often without the user's explicit permission). Another issue that complicates matters is that the same plug-in may have a different name (`navigator.plugins[i].name` property) depending on the operating system. Therefore, searching your script for the presence of a plug-in by name is not good enough if the name may be different on the Macintosh version versus the Windows 95 version. With luck, this naming discrepancy will resolve itself over time as plug-in developers understand the scripter's need for consistency across platforms.

To help you jump-start the process in your scripts, I discuss three utility functions you can use without modification in your own scripts. These functions are excerpts from a long listing (Listing 25-3), which is located in its entirety on the book's CD-ROM. The pieces not shown here are merely user interface elements to let you experiment with these functions.

Note

The scripts in Listing 25-3 reveal a bug in the Windows version of Navigator 4 that crashes Navigator when the script attempts to retrieve mimeType objects of certain types too quickly (in a `for` loop). The script does work fine in Windows for Navigator 3. Also, to date I have not found a compatible workaround for any version of Internet Explorer.

Verifying a MIME type

Listing 25-3a is a function whose narrow purpose is to compare any MIME type definition (in the `<type>/<subtype>` format as a string) against the browser's internal list of MIME types. The function does more, however, than simply look for a match. The reason is that the browser can easily list MIME types for which no

plug-in is installed (as is the case with dozens of MIME types in the default list provided in Navigator).

Listing 25-3a: **Verifying a MIME type**

```
// Pass "<type>/<subtype>" string to this function to find
// out if the MIME type is registered with this browser
// and that at least some plug-in is enabled for that type.
function mimeIsReady(mime_type) {
        for (var i = 0; i < navigator.mimeTypes.length; i++) {
            if (navigator.mimeTypes[i].type == mime_type) {
                if (navigator.mimeTypes[i].enabledPlugin != null) {
                    return true
                }
            }
        }
        return false
}
```

The real power of this function comes in the most nested `if` statement. For script execution to reach this point, the MIME type in question has been found to be listed in the browser's massive `mimeTypes[]` array. What you really need to know is whether a plug-in is enabled for that particular MIME type. If the script passes that final test, then you can safely report back that the browser supports the MIME type. If, on the other hand, no match has been found after cycling through all listed MIME types, the function returns false.

Verifying a plug-in

Next, in Listing 25-3b, you let JavaScript see if the browser has a specific plug-in registered in the `navigator.plugins[]` array. This method approaches the installation question from a different angle. Instead of coming with a known MIME type, you come with a known plug-in. But because more than one registered plug-in can support a given MIME type, this function explores one step further to see whether at least one of the plug-in's MIME types (of any kind) is enabled in the browser.

Listing 25-3b: **Verifying a Plug-in**

```
// Pass the name of a plug-in for this function to see
// if the plug-in is registered with this browser and
// that it is enabled for at least one MIME type of any kind.
function pluginIsReady(plug_in) {
        for (var i = 0; i < navigator.plugins.length; i++) {
            if (navigator.plugins[i].name.toLowerCase() ==
plug_in.toLowerCase()) {
                for (var j = 0; j < navigator.plugins[i].length; j++) {
                    if (navigator.plugins[i][j].enabledPlugin) {
```

(continued)

Listing 25-3b *(continued)*

```
                    return true
                }
            }
            return false
        }
    }
    return false
}
```

The parameter for the `pluginIsReady()` function is a string consisting of the plug-in's name as it appears in boldface in the About Plug-ins listing (from the Navigator Help menu). The script loops through all registered plug-ins for a match against this string (converting both strings to all lowercase to help overcome discrepancies in capitalization).

Next comes a second repeat loop, which looks through the MIME types associated with a plug-in (in this case, only a plug-in whose name matches the parameter). Notice the use of the strange, double-array syntax for the most nested `if` statement: For a given plug-in (denoted by the `i` index), you have to loop through all items in the MIME types array (`j`) connected to that plug-in. The conditional phrase for the last `if` statement has an implied comparison against null (see another way of explicitly showing the null comparison in Listing 25-3a). The conditional statement evaluates to either an object or a null, which JavaScript can accept as `true` or `false`, respectively. The point is that if an enabled plug-in is found for the given MIME type of the given plug-in, then this function returns true.

I must emphasize that before using this function, make sure that the plug-in is internally named the same way on all platforms for which a plug-in version exists. Such is not the case with many early plug-ins.

Verifying both plug-in and MIME type

The last utility function (Listing 25-3c) is the safest way of determining whether a visitor's browser is equipped with the "right stuff" to play your media. This function requires both a MIME type and plug-in name as parameters and also makes sure that both items are supported and enabled in the browser before returning a true.

Listing 25-3c: **Verifying Plug-in and MIME type**

```
// Pass "<type>/<subtype>" and plug-in name strings for this
// function to see if both the MIME type and plug-in are
// registered with this browser, and that the plug-in is
// enabled for the desired MIME type.
function mimeAndPluginReady(mimetype,plug_in) {
```

```
        for (var i = 0; i < navigator.plugins.length; i++) {
            if (navigator.plugins[i].name.toLowerCase() ==
plug_in.toLowerCase()) {
                for (var j = 0; j < navigator.plugins[i].length; j++) {
                    var mimeObj = navigator.plugins[i][j]
                    if (mimeObj.enabledPlugin && (mimeObj.type ==
mimetype)) {
                            return true
                    }
                }
                return false
            }
        }
        return false
}
```

This function resembles the one in Listing 25-3b until you reach the most nested statements. Here, instead of looking for any old MIME type, you insist on the existence of an explicit match between the MIME type passed as a parameter and an enabled MIME type associated with the plug-in. Because this function relies on a plug-in's name, the same cautions about checking for name consistency across platforms applies here. To see how these functions work on your browser, open the complete file (lst25-03.htm) from the CD-ROM.

Managing plug-in installation (Navigator 3)

If your scripts determine that a visitor is using Navigator 3 and does not have the plug-in your data expects, you may want to consider providing an electronic guide to installing the plug-in. One way to do this is to open a new frameset (in the main window). One frame would contain step-by-step instructions with links to the plug-in's download site. The download site's page would appear in the other frame of this temporary window. The steps must take into account all installation requirements for every platform, or, alternatively, you can create a separate installation document for each unique class of platform. Macintosh files, for instance, frequently must be decoded from binhex format and then uncompressed before you move them into the plug-ins folder. Other plug-ins have their own, separate installation program. The final step should include a call to

```
navigator.plugins.refresh()
```

to make sure that the browser updates its internal listings. After that, the script can go back to the `document.referrer`, which should be the page that sent the visitor to the installation pages. All in all, the process is cumbersome — it's not like downloading a Java applet. But if you provide some guidance, you stand a better chance of the user coming back to play your cool media.

Screen Object

Properties	*Methods*	*Event Handlers*
`availHeight`	(None)	(None)
`availWidth`		
`colorDepth`		
`height`		
`pixelDepth`		
`width`		

Syntax

Accessing `screen` properties:

```
screen.property
```

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

About this object

Navigator 4 provides a screen object that lets your scripts inquire about the size and color settings of the video monitor used to display a page. Properties are carefully designed to reveal not only the raw width and height of the monitor (in pixels), but what the available width and height are once you take into account the operating system's screen-hogging interface elements (for example, the Windows 95/NT Taskbar and the Mac menubar).

Some of these property values are also accessible in Navigator 3 if you use LiveConnect to access Java classes directly. Example code for this approach is supplied in the individual property listings.

Internet Explorer 4 provides a screen object, although it appears as an element of the window object in the Internet Explorer 4 object model. Only three properties of the Internet Explorer 4 screen object — `height`, `width`, and `colorDepth` — are the same syntax as Navigator 4's screen object.

availHeight
availWidth
height
width

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

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

With Navigator 4's additional window sizing methods, your scripts may want to know how large the user's monitor is. This is particularly important if you are setting up an application to run in kiosk mode, which occupies the entire screen. Two pairs of properties let scripts extract the dimensions of the screen. All dimensions are in pixels.

The gross height and width of the monitor is available from the `screen.height` and `screen.width` properties. Thus, a monitor rated as an 800 × 600 monitor returns values of 800 and 600 for width and height, respectively. These properties are available in Internet Explorer 4, as well.

But the gross size is not always completely available as displayable area for a window. To the rescue come the `screen.availWidth` and `screen.availHeight` properties. For example, Windows 95 and NT 4 display the Taskbar. The default location for this bar is at the bottom of the window, but users can reorient it along any edge of the screen. If the default behavior of always showing the Taskbar is in force, the bar takes away from the screen real estate available for window display (unless you intentionally size or position a window so that part of the window extends under the bar). When along the top or bottom edge of the screen, the Taskbar occupies 28 vertical pixels; when positioned along one of the sides, the bar occupies 60 horizontal pixels. On the Macintosh platform, the 20-pixel-deep menubar occupies a top strip of the screen. While windows can be positioned and sized so they are partially covered by the menubar, it is not a good idea to open a window in or move a window into that location.

You can use the available screen size values as settings for window properties. For example, to maximize a window, you must position the window at the top left of the screen and then set the outer window dimensions to the available sizes, as follows:

```
function maximize() {
        window.moveTo(0,0)
        window.outerWidth = screen.availWidth
        window.outerHeight = screen.availHeight
}
```

The above function positions the window appropriately on the Macintosh just below the menubar so that the window is not obscured by the menubar. If, however, the client is Windows 95/NT and the user has positioned the Taskbar at the top of the screen, the window will be partially hidden under the Taskbar (you cannot query the available screen space's coordinates).

For Navigator 3, you can use LiveConnect to access a native Java class that reveals the overall screen size (not the available screen size). If the user is running Navigator 3 and Java is enabled, the following script fragment can be placed in the Head portion of your document to set variables with screen width and height:

```
var toolkit = java.awt.Toolkit.getDefaultToolkit()
var screen = toolkit.getScreenSize()
```

The screen variable is an object whose properties (`width` and `height`) contain the pixel measures of the current screen. This LiveConnect technique also works in Navigator 4 (but not in Internet Explorer 3), so you can use one screen size method for all late-model Navigators. In fact, you can also extract the screen resolution (pixels per inch) in the same manner. The following statement, added after the ones above, sets the variable `resolution` to that value:

```
var resolution = toolkit.getScreenResolution()
```

Related Items: `window.innerHeight` property; `window.innerWidth` property; `window.outerHeight` property; `window.outerWidth` property; `window.moveTo()` method; `window.resizeTo()` method.

colorDepth

pixelDepth

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

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

You can design a page for Navigator 4 with different color models in mind, because your scripts can query the client to find out how many colors the user has set the monitor to display. This can be helpful if you have more subtle color schemes that require 16-bit color settings or images tailored to specific palette sizes.

Both `screen.colorDepth` and `screen.pixelDepth` properties return the number of bits that the color client computer's video display control panel is set to. The `screen.colorDepth` value may take into account a custom color palette, so in general I prefer to rely only on the `screen.pixelDepth` value (Internet Explorer 4 supports only the `screen.colorDepth` property of this pair). You can use this value to determine which of two image versions to load, as shown in the following script fragment that runs as the document loads:

```
if (screen.colorDepth > 8 ) {
       document.write("<IMG SRC='logoHI.jpg' HEIGHT='60' WIDTH='100'")
} else {
       document.write("<IMG SRC='logoLO.jpg' HEIGHT='60' WIDTH='100'")
}
```

In this example, the logoHI.jpg image is designed for 16-bit displays or better, while the colors in logoLO.jpg have been tuned for 8-bit display.

While LiveConnect in Navigator 3 has a way to extract what appears to be the `pixelDepth` equivalent, the Java implementation is flawed. You do not always get the correct value, so I don't recommend relying on this tactic for Navigator 3 users.

Related Items: None.