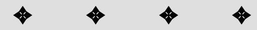
CHAPTER 28

The Navigator and Other Environment Objects

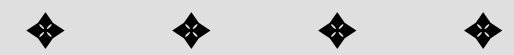


In This Chapter

Determining which browser the user has

Branching scripts according to the user's operating system

Detecting plug-in support



Client-side scripting primarily focuses on the document inside a browser window and the content of the document. As discussed in Chapter 16, the window, too, is an important part of how you apply JavaScript on the client. But stepping out even one more level is the browser application itself. Scripts sometimes need to know about the browser and the computing environment in which it runs so that they can tailor dynamic content for the current browser and operating system.

To that end, browsers provide objects that expose as much about the client computer and the browser as is feasible within accepted principles of preserving a user's privacy. In addition to providing some of the same information that CGI programs on the server receive as environment variables, these browser-level objects also include information about how well equipped the browser is with regard to plug-ins and Java. Another object defined for NN4+ and IE4+ reveals information about the user's video monitor, which may influence the way your scripts calculate information displayed on the page.

The objects in this chapter don't show up on the document object hierarchy diagrams, except as freestanding groups (see Appendix A). The IE4+ object model, however, incorporates these environmental objects as properties of the `window` object. Because the `window` reference is optional, you can omit it for IE and wind up with a cross-browser, compatible script in many cases.

Where the IE (for Windows anyway) and NN environments diverge significantly is in the way scripts can find out whether a particular plug-in or support for a particular MIME type is available in the current browser. As you learn in this chapter, the IE for Windows methodology can be a bit roundabout. And yet the Macintosh version of IE5+ has adopted the approach initiated by NN3. Go figure.

clientInformation Object (IE4+) and navigator Object (All)

Properties	*Methods*	*Event Handlers*
`appCodeName`	`javaEnabled()`	
`appMinorVersion`	`preference()`	
`appName`	`taintEnabled()`	
`appVersion`		
`browserLanguage`		
`cookieEnabled`		
`cpuClass`		
`language`		
`mimeTypes`		
`onLine`		
`oscpu`		
`platform`		
`plugins`		
`product`		
`productSub`		
`securityPolicy`		
`systemLanguage`		
`userAgent`		
`userLanguage`		
`userProfile`		

Properties	Methods	Event Handlers
vendor		
vendorSub		

Syntax

Accessing clientInformation and navigator object properties and methods:

```
(All)        navigator.property | method()
(IE4+/NN6)    [window.]navigator.property | method()
(IE4+)       [window.]clientInformation.property | method()
```

About this object

In Chapter 16, I repeatedly mention that the window object is the top banana of the document object hierarchy. In other programming environments, you likely can find a level higher than the window — perhaps referred to as the *application level*. You may think that an object known as the navigator object is that all-encompassing object. That is not the case, however.

Although Netscape originally invented the navigator object for the Navigator 2 browser, Microsoft Internet Explorer also supports this object in its object model. For those who exhibit partisan feelings toward Microsoft, IE4+ provides an alternate object — clientInformation — that acts as an alias to the navigator object. You are free to use the IE-specific terminology if your development is intended only for IE browsers. All properties and methods of the navigator and clientInformation objects are identical. In the rest of this section, all references to the navigator object also apply to the clientInformation object.

Be aware that the number of properties for this object has grown with virtually every browser version. Moreover, other than some basic items that have been around since the early days, most of the more recent properties are browser-specific. Observe the compatibility ratings for each of the following properties very carefully.

Most of 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 related to one another, I begin this discussion by grouping four of the most popular ones together.

Properties

appCodeName
appName
appVersion
userAgent

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility	✓	✓	✓	✓	✓	✓	✓	✓	✓

These four properties reveal just about everything that browser-sniffing code needs to know about the user's browser brand, version, and other tidbits. Of these four, only the last three are particularly valuable. The first property in the list, `appCodeName`, defines a class of client that encompasses essentially every standard browser. The value returned by browsers, `Mozilla`, is the code name of the first browser engine on which NN and IE browsers at one time were based (the NCSA Mosaic browser). This information does nothing to help your scripts distinguish among browser flavors, so you can ignore the property. But the other three properties are the ones with all the goodies.

The `appName` property returns the official name for the browser application. For Netscape browsers, the `appName` value is `Netscape`; for Internet Explorer, the value is `Microsoft Internet Explorer`.

The `appVersion` and `userAgent` properties provide more meaningful detail. I start with the `appVersion` property because it is revealing and, at times, misleading.

Using the appVersion property

A typical `appVersion` property value looks like the following (one from NN6, one from IE5):

```
5.0 (Windows; en-US)
4.0 (compatible; MSIE 5.5; Windows 98; compat; DigExt)
```

Because most version decisions are based on numeric comparisons (for example, the version is equal to or greater than 4), you frequently need to extract just the number part of the string returned by the `appVersion` property. The cleanest way to do this is via the `parseInt()` or `parseFloat()` methods. Use

```
parseInt(navigator.appVersion)
```

if you are interested only in the number to the left of the decimal; to get the complete leading floating-point number, use

```
parseFloat(navigator.appVersion)
```

All other characters after the leading numbers are ignored.

Also notice that the number does not always accurately represent the version of the browser at hand. For instance, IE5.5 reports that it is Version 4.0. The number is more indicative of a broad generation number rather than a specific browser version number. In other words, the browser exhibits characteristics of the first browsers to wear the `appVersion` of 4 (IE 4.0, it turns out). While this means that IE5.5 can use everything that is in the language and object model of IE4, this obviously doesn't help your script to know if the browser is capable of IE5.5 scripting features.

At the same time, however, buried elsewhere in the `appVersion` string is the wording `MSIE 5.5`—the "true" version of the browser. IE uses this technique to distinguish the actual version number from the generational number. Therefore, for IE, you may have to dig deeper by using string methods such as `indexOf()` to see if the `appVersion` contains the desired string. For example, to see if the browser is a variant of IE5, you can test for just `"MSIE 5"` as follows:

```
var isIE5x = navigator.appVersion.indexOf("MSIE 5") != -1
```

Or to know if the browser is IE5.5, include more of the string:

```
var isIE5_5 = navigator.appVersion.indexOf("MSIE 5.5") != -1
```

There is a hazard in doing this kind of testing, however. Going forward, your code will break if future versions of IE have larger version numbers. Therefore, if you want to use IE5 features with an IE6 browser (assuming such a browser becomes available), your testing for the presence of `"MSIE 5"` fails and the script thinks that it cannot use IE5 features even though they most certainly would be available in IE6. To find out if the current IE browser is the same or newer than a particular version, you must use JavaScript string parsing to deal with the `MSIE x.x` substring of the `appVersion` (or `userAgent`) property. The following example shows one function that extracts the precise IE version name and another function that confirms whether the version is at least IE5.0 for Windows.

```
var ua = navigator.userAgent
function getIEVersion() {
    var IEOffset = ua.indexOf("MSIE ")
    return parseFloat(ua.substring(IEOffset + 5, ua.indexOf(";", IEOffset)))
}
function qualifyBrowser() {
    var qualified = false
    if (navigator.appName == "Microsoft Internet Explorer") {
        if (parseInt(getIEVersion()) >= 5) {
            if (ua.indexOf("Windows") != -1) {
                qualified = true
            }
        }
    }
    if (!qualified) {
        var msg = "These scripts are currently certified to run on:\n"
        msg += " - MS Internet Explorer 5.0 or later for Windows\n"
        alert(msg)
    }
    return qualified
}
```

As clever as the code above looks, using it assumes that the version string surrounding the `MSIE` characters will be immutable in the future. We do not have that kind of guarantee, so you have to remain vigilant for possible changes in future versions.

Thus, with each browser generation's pollution of the `appVersion` and `userAgent` properties, the properties become increasingly less useful for browser sniffing — unless you wish to burden your code with a lot of general-purpose sniffing code, very little of which any one browser uses.

Even NN is not free of problems. For example, the main numbering in the `appVersion` property for NN6 is `5` (in other words, the fifth generation of Mozilla). Buried elsewhere in the property value is the string `Netscape6`. A potentially thornier problem arises due to Netscape's decision to eliminate some nonstandard NN4 DOM features from the NN6 DOM (layer objects and some event object behaviors). Many scripters followed the previously recommended technique of "prepare for the future" by using an `appVersion` of 4 as a minimum:

```
var isNN4 = parseInt(navigator.appVersion) >= 4
```

But any code that relies on the `isNN4` variable to branch to code that talks to the dead-end NN4 objects and properties breaks when it runs in NN6.

The bottom line question is, "What do I do for browser version detection?" Unfortunately, there are dozens of answers to that question, depending on what you need browser detection to do and what level of code you produce.

At one end of the spectrum is code that tries to be many things to many browsers, implementing multiple levels of features for many different generations of browser. This is clearly the most difficult tactic, and you have to create quite a long list of variables for the conditions for which you establish branches. Some branches may work on one combination of browsers, while you may need to split other branches differently because the scripted features have more browser-specific implementations.

At the other end of the spectrum is the code that tries to support, say, only IE5+ and NN6+ with W3C DOM-compatible syntax to the extent that both browser families implement the object model features. Life for this scripter is much easier in that the amount of branching is little or none depending on what the scripts do with the objects.

Between these two extremes, situations call for many different solutions. Object detection (for example, seeing if `document.images` exists before manipulating image objects) is a good solution at times, but not so much for determining the browser version as for knowing whether some code that addresses those objects works. As described in Chapter 14, it is hazardous to use the existence of, say, `document.all` as an indicator that the browser is IE4+. Some other browser in the future may also implement the `document.all` property, but not necessarily all the other IE4+ objects and syntax. Code that thinks it's running in IE4+ just because `document.all` exists can easily break if `document.all` is implemented in another browser but not all the rest of the IE4+ DOM. Using object detection to branch code that addresses the detected objects is, however, very desirable in the long run because it frees your code from getting trapped in the ever-changing browser version game.

Don't write off the `appVersion` and `userAgent` properties entirely. The combination of features that you script may benefit from some of the data in that string, especially when the decisions are made in concert with the `navigator.appName` property. A number of other properties implemented in IE4+ and NN6 can also provide the sufficient clues for your code to perform the branching that your application needs. For instance, it may be very helpful to your scripts to know whether the `navigator.platform` property informs them that they are running in a Windows or Macintosh environment because of the way each operating system renders fonts.

userAgent property details

The string returned by the `navigator.userAgent` property contains a more complete rundown of the browser. 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.

Unfortunately, there is no standard for the way information in the `userAgent` property is formatted. It may be instructive, however, to view what kinds of values come from a variety of browsers on different platforms. Table 28-1 shows some of

the values that your scripts are likely to see. This table does not include, of course, the many values that are not reflected by browsers that do not support JavaScript. The purpose of the table is to show you just a sampling of data that the property can contain from a variety of browsers and operating systems (particularly enlightening if you do not have access to Macintosh or UNIX computers).

Table 28-1 Typical navigator.userAgent Values

navigator.userAgent	*Description*
`Mozilla/5.0 (Windows; U; Win98; en-US)Netscape6/6.0`	Navigator 6 for Windows, running under Windows 98; U.S. English edition and U.S. encryption
`Mozilla/4.74 [en] (X11; U; Linux 2.2.154mdksmp i686)`	Navigator 4.74, English edition for Linux with U.S. encryption
`Mozilla/4.73 (Macintosh; U; PPC)`	Navigator 4.73 for PowerPC Macintosh with U.S. encryption
`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/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; SunOS 5.4 sun4m)`	Navigator 3.01 for SunOS 5.4
`Mozilla/3.01Gold [de] (Win16; I)`	Navigator 3.01, German edition for Windows 3.0*x*
`Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)`	IE 5.0 for Windows 98 with digital signature
`Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)`	IE 5.5 running under Windows NT 5.0
`Mozilla/4.0 (compatible; MSIE 5.0; Mac_PowerPC)`	IE 5.0 running on a PowerPC-equipped Macintosh

navigator.userAgent	*Description*
`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/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 3.0B; Windows NT)`	IE 3 (beta) emulating Navigator 2

Because the `userAgent` property contains a lot of the same information as the `appVersion` property, the same cautions just described apply to the `userAgent` string and the environment data it returns.

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 on which you may rely. If that is the case, then I suggest you look 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 not bothering with branching for incremental releases.

See Chapters 13 and 14 for more information about designing pages for cross-platform deployment.

Example (with Listing 28-1) on the CD-ROM

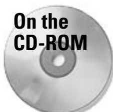
Related Items: `appMinorVersion`, `cpuClass`, `oscpu`, `platform` properties.

appMinorVersion

Value: One-Character String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

In IE parlance, the *minor version* is indicated by the first digit to the right of the decimal in a full version number. But the "version number" referred to here is the number that the `navigator.appVersion` property reports, not the actual version of the browser. For example, although IE5.5 seems to have a version number of 5 and a minor version number of 5, the `appVersion` reports Version 4.0. In this case, the `minorAppVersion` reports 0. Thus, you cannot use the `appMinorVersion` property to detect differences between, say, IE5 and IE5.5. That information is buried deeper within the string returned by `appVersion` and `userAgent`.



Example on the CD-ROM

Related Item: `appVersion` property.

browserLanguage

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `browserLanguage` property in IE4+ (and the `language` property in NN4+) returns the identifier for a localized language version of the program (it has nothing to do with scripting or programming language). The value of the `browserLanguage` property almost always is the same as the other IE language-related properties, unless the user changes the Windows control panel for regional settings after installing IE. In that case, `browserLanguage` returns the original language of the browser application, while the other properties report the language indicated in the system-level preferences panel.

Users of the multilanguage version of Windows 2000 can choose alternate languages for menus and dialog boxes. The `browserLanguage` property returns the language you choose for those settings.

These short strings may resemble, but are not identical to, the URL suffixes for countries. Moreover, when a language has multiple dialects, the dialect can also be a part of the identifier. For example, `en` is the identifier for English. However, `en-us` (or `en-US`) represents the American dialect of English, while `en-gb` (or `en-GB`) represents the dialect recognized in Great Britain. NN sometimes includes these values as part of the `userAgent` data as well. Table 28-2 shows a sampling of language identifiers used for all language-related properties of the `navigator` object.

Table 28-2 **Sample navigator.browserLanguage 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

You can assume that a user of a particular language version of the browser or system is also 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`, `navigator.language`, `navigator.systemLanguage`, `navigator.userLanguage` properties.

cookieEnabled

Value: Boolean Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

The `cookieEnabled` property allows your scripts to determine easily if the browser has cookie functionality turned on. You can surround cookie-related statements with an `if` construction as follows:

```
if (navigator.cookieEnabled) {
    // do cookie stuff here
}
```

This works reliably only on browsers that implement the property. Because older browsers do not have this `navigator` object property, the `if` condition appears `false` (even though cookies may be turned on).

You can still check for cookie functionality in older browsers, but only clumsily. The technique entails assigning a "dummy" cookie value to the `document.cookie` property and attempting to read back the cookie value. If the value is there, then cookies are enabled.

Example on the CD-ROM

Related Item: `document.cookie` property.

cpuClass

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `cpuClass` property returns one of several fixed strings that identifies the family of central processing units running IE. Possible values and their meanings are as follows:

cpuClass	*Description*
`x86`	Intel processor (and some emulators)
`PPC`	Motorola Power PC processor (for example, Macintosh)
`68K`	Motorola 68000-family processor (for example, Macintosh)
`Alpha`	Digital Equipment Alpha processor
`Other`	Other processors, such as SPARC

The processor is not a good guide to determining the operating system because you can run multiple operating systems on most of the preceding processor fami-

lies. Moreover, the `cpuClass` value represents the processor that the browser "thinks" it is running on. For example, when a Windows version of IE is hosted by the *Virtual PC* emulator on a PowerPC Macintosh, the `cpuClass` is reported as `x86` even though the actual hardware processor is `PPC`.

Example on the CD-ROM

Related Item: `navigator.oscpu` property.

language

Value: String — Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓					

The NN4+ `language` property returns the language code for the browser application. While the comparable IE property (`navigator.browserLanguage`) has morphed in later versions to focus on the operating system language, NN's property deals exclusively with the language for which the browser application is written.

Related Item: `navigator.browserLanguage` property.

mimeTypes

Value: Array of mimeType objects — Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
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 the "MimeType object" section later in this chapter). NN3+ 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 want to be sure that each visitor's browser is capable of playing the media your page has to offer. With JavaScript and NN3+, 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 can't determine whether a helper application is assigned to this MIME type from the array list.

You may have noticed that the preceding discussion focuses on Netscape Navigator, yet the compatibility chart shows that IE4+ supports the `mimeTypes` property. The actual situation is more complex. The Windows version of IE4+ supports this property only in so far as to return an empty array. In other words, the property is defined, but it does not contain `mimeType` objects — a nonexistent object in IE for Windows. But on the Macintosh side, IE5+ supports the way Netscape Navigator allows script inspection of MIME types and plug-ins. To see ways of determining plug-in support for IE/Windows, see the section "Plug-in detection in IE/Windows" later in this chapter.

Example on the CD-ROM

Related Item: `navigator.plugins` property; `mimeType` object.

onLine

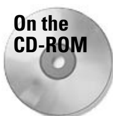
Value: Boolean Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `onLine` property lets scripts determine the state of the offline browsing setting for the browser. Bear in mind that this property does not reveal whether the page is accessed via the Net or a local hard disk. The browser can be in online mode and still access a local page; in this case, the `onLine` property returns `true`.

With the offline browsing capabilities of IE4+, users may prefer to download copies of pages they wish to reference frequently (perhaps on a disconnected laptop computer). In such cases, your pages may want to avoid network-reliant content when accessed offline. For example, if your page includes a link to a live audio feed, you can dynamically generate that link with JavaScript — but do so only if the user is online:

```
if (navigator.onLine) {
    document.write("<A HREF='broadcast.rna'>Listen to Audio</A>")
}
```



Example on the CD-ROM

Related Items: None.

oscpu

Value: String — Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The NN6 `oscpu` property returns a string that reveals OS- or CPU-related information about the user's environment. The precise string varies widely with the client OS. For instance, a Windows 98 machine reports `Win98`, while a Macintosh reports `PPC`. The string formats for Windows NT versions are not standardized, so they offer values such as `WinNT4.0` and `Windows NT 5.0`. UNIX platforms reveal more details, such as the system version and hardware.

Example on the CD-ROM

Related Item: `navigator.cpuClass` property.

platform

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓			✓	✓	✓

The `navigator.platform` value reflects the operating system according to the codes established initially by Netscape for its `userAgent` values. Table 28-3 lists typical values of several operating systems.

In the long list of browser detection functions in Listing 28-1, I elected not to use the `navigator.platform` property because it is not backward-compatible. Meanwhile, the other properties in that listing are available to all scriptable browsers.

Table 28-3 Sample navigator.platform Values

navigator.platform	*Operating System*
`Win98`	Windows 98
`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.

Example on the CD-ROM

Related Item: `navigator.userAgent` property.

plugins

Value: Array of Plug-in Objects Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓			(✓)	(✓)	(✓)

You rarely find users involved with Web page design 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.

When many browsers launch, they create an internal list of available plug-ins located in a special directory/folder (the name varies with the browser and operating system). The `navigator.plugins` array lists the items registered at launch time. Each plug-in is, itself, an object with several properties.

The Windows version of IE4+ supports this property only to return an empty array. In other words, the property is defined, but it does not contain `plugin` objects — a nonexistent object in IE for Windows. But on the Macintosh side, IE5+ supports the way Netscape Navigator allows script inspection of MIME types and plug-ins. To see ways of determining plug-in support for IE/Windows, see the section “Plug-in detection in IE/Windows” later in this chapter.

Having 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 have it currently).

Example

For examples of the `plugins` property and for details about using the plugin object, see the section “plugin object” later in this chapter. Also see Chapter 32 on embedded element objects.

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

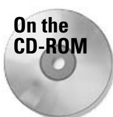
product
productSub
vendor
vendorSub

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

With the browser engine behind Navigator 6 being developed in an Open Source environment, any number of vendors might adapt the engine for any number of browser products. Some distributors of the browser, such as ISPs and computer manufacturers, may also tailor the browser slightly for their customers. These four properties can reveal some of the pedigree of the browser currently running scripts on the page.

Two categories of properties — one for the product, one for the vendor — each have a pair of fields (a primary and secondary field) that can be populated as the vendor sees fit. Some of this information may contain data, such as an identifying number of the *build* (development version) used to generate the product. A script at a computer maker's Web site page may look for a particular series of values in these properties to welcome the customer or to advise the customer of a later build version that is recommended as an upgrade.



Example on the CD-ROM

Related Item: `navigator.userAgent` property.

securityPolicy

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓					

The Netscape-specific `securityPolicy` property returns a string that indicates which cryptographic scheme is implemented in the current browser. Typical string values include `US and CA domestic policy` and `export policy`. Each policy indicates the number of bits used for encryption, usually governed by technology export laws. While the property returns a value in NN4, it returns only an empty string in the first release of NN6. The corresponding IE property is `document.security`.

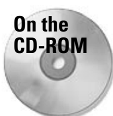
Related Item: `document.security` property.

systemLanguage
userLanguage

Value: Language Code String — Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

These two IE-specific properties report the language code of the written language specified for the operating system. For most operating system versions, these two values are the same. Some Windows versions enable you to set system preferences differently for the base operating system and the language for a given user. Both of these property values can differ from the `navigator.browserLanguage` property if the user downloads and installs the browser with the system set to one language and then changes the system settings to another language.



Example on the CD-ROM

Related Item: `navigator.browserLanguage` property.

userAgent

See `appCodeName`.

userLanguage

See `systemLanguage`.

userProfile

Value: `userProfile` Object Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `userProfile` property returns a reference to the IE `userProfile` object. This object provides scripted access to a limited range of user profile settings with the user's permission. For details, see the `userProfile` object discussion later in this chapter.

Related Item: `userProfile` object.

vendor
vendorSub

See `product`.

Methods

javaEnabled()

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility		✓	✓	✓			✓	✓	✓

Although most modern browsers ship with Java support turned on, a user can easily turn it off in a preferences dialog box (or even elect not to install it with the browser). 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 normally goes. But if you script applets from JavaScript (via LiveConnect, Chapter 44), you don't want your scripts making calls to applets or Java classes if Java support is turned off. In a similar vein, if you create a page with JavaScript, you can fashion two different layouts depending on the availability of Java.

The `navigator.javaEnabled()` method returns a Boolean value reflecting the preferences setting. This value does not reflect Java support in the browser necessarily (and especially not the Java version supported), but rather whether Java is turned on inside the browsers for which this method is supported. A script cannot change the browser's preference setting, but its value does change immediately upon toggling the Preference setting.

Related Items: `navigator.preference()` method; LiveConnect (Chapter 44).

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

Returns: Preference value

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓					

The user normally sets browser preferences. Until NN4 and the advent of signed scripts, almost all settings were completely out of view of scripts — even when it made sense to expose them. But with signed scripts and the `navigator.preference()` method, many NN 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 the browser preferences of a public Web site visitor, even if given permission to do so — the user may not know how much trouble you can cause.

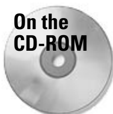
When you want to read a particular preference setting, you pass only the preference name parameter with the method. Reading a preference requires a signed script with the target of `UniversalPreferencesRead` (see Chapter 46). To change a preference, pass both the preference name and the value (with a signed script target of `UniversalPreferencesWrite`).

Table 28-4 shows a handful of scriptable preferences in NN4+ (learn more about these settings at `http://developer.netscape.com/docs/manuals/communicator/preferences/`). Most items have corresponding entries in the preferences window in NN4+ (shown in parentheses). Notice that the preference name uses dot syntax. The cookie security level is a single preference value with a matrix of integer values indicating the level.

Table 28-4 navigator.preference() Values Sampler

navigator.preference	*Value*	*Preference Dialog Listing*
`general.always_load_images`	Boolean	(Advanced) Automatically loads images
`security.enable_java`	Boolean	(Advanced) Enables Java
`javascript.enabled`	Boolean	(Advanced) Enables JavaScript
`browser.enable_style_sheets`	Boolean	(Advanced) Enables style sheets
`autoupdate.enabled`	Boolean	(Advanced) Enables AutoInstall
`navigator.preference`	Value	Preference Dialog Listing
`network.cookie.cookieBehavior`	0	(Advanced) Accepts all cookies
`network.cookie.cookieBehavior`	1	(Advanced) Accepts only cookies that get sent back to the originating server
`network.cookie.cookieBehavior`	2	(Advanced) Disables cookies
`network.cookie.warnAboutCookies`	Boolean	(Advanced) Warns you before accepting a cookie

One preference to watch out for is the one that disables JavaScript. If you disable JavaScript, only the user can reenable JavaScript by manually changing the setting in the Navigator preferences dialog box.



Example (with Listing 28-2) on the CD-ROM

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

taintEnabled()

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility		✓	✓	✓			✓	✓	✓

Navigator 3 featured a partially implemented security feature called *data tainting*, which was turned off by default. This feature was replaced by signed scripts; but

for backward compatibility, the `navigator.taintEnabled()` method is available in more modern browsers that don't employ tainting (in which case, the method always returns `false`). Do not employ this method in your scripts.

mimeType Object

Properties	*Methods*	*Event Handlers*
`description`		
`enabledPlugin`		
`type`		
`suffixes`		

Syntax

Accessing mimeType properties:

```
navigator.mimeTypes[i].property
navigator.mimeTypes["MIMEtype"].property
```

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility		✓	✓	✓				(✓)	(✓)

About this object

A mimeType object is essentially an entry in the internal array of MIME types about which the browser knows. NN3+, 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 list all of the plug-ins and other helpers you've added, 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
video/quicktime
application/pdf
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 you load the file 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 preferences settings usually associated with the name "Applications."

By having the mimeType object available to JavaScript, your page can query a visitor's NN3+ or IE5+/Mac browser to discover whether it has a particular MIME type listed currently and 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. (For plug-in detection for IE/Windows, see the section "Plug-in detection in IE/Windows" later in this chapter.)

Because of the close relationship between mimeType and plugin objects, I save the examples of using these objects and their properties for a section later in this chapter. There you can see how to build functions into your scripts that enable you to examine how well a visitor's NN3+ and IE5+/Mac browser is equipped for either a MIME type or data that requires a specific plug-in. In the meantime, be sure that you understand the properties of both objects.

Properties

description

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility		✓	✓	✓				(✓)	(✓)

While registering with the browser at launch time, plug-ins provide the browser with an extra field of information: a plain-language description of the plug-in. If a

particular MIME type has a plug-in associated with it and enabled for it, the plug-in's description passes through to become the description of the mimeType object. For example, the Adobe Acrobat plug-in (whose MIME type is `application/pdf`) supplies the following description fields:

```
(NN3/NN4)    Acrobat
(NN6)        Acrobat (*.pdf)
```

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 Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility		✓	✓	✓				(✓)	(✓)

The descriptions of the mimeType and plugin objects seem to come full circle when you reach the `mimeType.enabledPlugin` property. The reason is 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 is 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 audio plug-in if you set up your browser that way. Therefore, all MIDI data streams play through the Crescendo plug-in. If you prefer to have your Web page's MIDI sound played only through another plug-in, such as LiveAudio in NN, 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, except where there is strong competition for players of various audio and video media. For other kinds of content, each plug-in developer typically creates a new type of data that has a unique MIME type. But you have no guarantee of such uniqueness, so I highly recommend a careful check of MIME type and plug-in 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. You can use The Evaluator (with NN3+ and IE5+/Mac) to study the relationship between mimeType and plugin objects:

1. Enter the following statement into the bottom text box to examine the properties of a mimeType object:

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

   Notice that the `enabledPlugin` property returns an object.

2. Inspect the plugin object from the bottom text box.

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

   You then see properties and values for a plugin object (described later in this chapter).

3. Check the plugin object for a different mimeType object by using a different index value:

   ```
   navigator.mimeTypes[7].enabledPlugin
   ```

The `mimeTypes` array index values vary almost with every browser, depending on what the user has installed. Therefore, do not rely on the index position in a script to assume that a particular mimeType object is in that position on all browsers.

Example

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

Related Item: `plugin` object.

type

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
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.

The `type` property has a special place in the mimeType object in that its string value can act as the index to the `navigator.mimeTypes` array. Therefore, to get straight to the mimeType object for, say, the `audio/wav` MIME type, your script can reference it directly through the `mimeTypes` array:

```
navigator.mimeTypes["audio/wav"]
```

This same reference can then get you straight to the enabled plug-in (if any) for the MIME type:

```
navigator.mimeTypes["audio/wav"].enabledPlugin
```

Example

See the section “Looking for MIME Types and Plug-ins” later in this chapter.

Related Item: `description` property.

suffixes

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
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 among 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 the section “Looking for MIME Types and Plug-ins” later in this chapter.

Related Items: None.

plugin Object

Properties	*Methods*	*Event Handlers*
`name`	`refresh()`	
`filename`		
`description`		
`length`		

Syntax

Accessing plugin object properties or method:

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

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility		✓	✓	✓				(✓)	(✓)

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 44).

The subject here, however, is 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 to help with installing a plug-in.

The close association between the plugin and mimeType objects, 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. Use The Evaluator (Chapter 13) to experiment with MIME types from the point of view of a plug-in. Begin by finding the name of the plug-in that your browser uses for a common audio MIME type:

1. Enter the following statement into the top text box:

   ```
   navigator.mimeTypes["audio/wav"].enabledPlugin.name
   ```

   If you use NN3+, the value returned is probably `"LiveAudio"`; for IE5+/Mac, the name is probably a version of QuickTime. Copy the name into the clipboard so that you can use it in subsequent statements. The remaining examples show `"LiveAudio"` where you should paste in your plug-in's name.

2. Enter the following statement into the top text box:

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

   Instead of the typical index value for the array notation, use the actual name of the plug-in. This expression evaluates to a number indicating the total number of different MIME types that the plug-in recognizes.

3. Look at the first MIME type specified for the plug-in by entering the following statement into the top text box:

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

The two successive pairs of square brackets is not a typo: Because the entry in the `plugins` array evaluates to an array itself, the second set of square brackets describes the index of the array returned by `plugins["LiveAudio"]`—a period does not separate the sets of brackets. 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; 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 the section "Looking for MIME Types and Plug-ins" later in this chapter for details on how to use the plugin object in a production setting.

Properties

name
filename
description
length

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
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. Thus, while there is no explicit property that defines a plug-in's version number, that information may be part of the string returned by the `name` or `description` properties.

Be aware that plug-in authors may not assign the same name to every OS platform version of a plug-in. Be prepared for discrepancies across platforms. You should hope that the plug-in that you're interested in has a uniform name across platforms because the value of the `name` property can function as an index to the `navigator.plugins` array to access a particular plugin object directly.

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 OS platforms, only the plug-in filename appears.

Finally, the `length` property of a plugin object counts the number of MIME types that the plug-in recognizes (but is not enabled for necessarily). Although you can use this information to loop through all possible MIME types for a plug-in, a more instructive way is to have your scripts approach the issue via the MIME type (as discussed later in this chapter).

Example

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

Related Item: `mimeType.description` property.

Methods

refresh()

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility		✓	✓	✓				(✓)	(✓)

You may have guessed that many browsers determine their lists of installed plug-ins while they launch. If you drop a new plug-in file into the plug-ins directory/folder, you have to quit the browser and relaunch it before the browser sees the new plug-in file. But that isn't a very friendly approach if you take pains to guide a user through downloading and installing 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 the act of quitting and relaunching the browser makes a plug-in removal take full effect.

Related Items: None.

Looking for MIME Types 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 guarantee that they have the requisite plug-ins installed to accommodate that fancy data. Most modern browser versions provide a bit of internal "smarts" by noticing when data requiring an uninstalled plug-in is about to load and trying to

help the user install a missing plug-in. You may wish, however, to take more control over the process by examining the user's browser plug-in functionality prior to loading the external data file.

The best source of information, when available, is the software developer of the plug-in. Macromedia, for example, provides numerous technical notes on its Web site (`www.macromedia.com`) about plug-in detection for its various plug-ins and versions. Unfortunately, that kind of assistance is not always easy to find from other vendors.

A lot of the discussion thus far in this chapter addresses the objects that make plug-in and MIME type support detection possible in some browsers. Netscape for NN3 initially introduced these objects. Since then, they have been adopted by IE5 for the Macintosh only. Microsoft makes it possible — but not easy — to determine whether a particular plug-in is available for IE/Windows. The approach for IE/Windows is entirely different from what I have covered so far; if you wish to perform cross-browser detection, you have to branch your code accordingly. I outline each approach next in its own section, starting with the NN3+ and IE5+/Mac way.

Overview: using mimeType and plugin objects

The value of performing your own inspection of plug-in support is that you can maintain better control of your site visitors who don't have the necessary plug-in yet. Rather than merely providing a link to the plug-in's download site, you can build a more complete interface around the download 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 normally expect to find in a browser. For example, a newly installed audio plug-in may take over for Netscape's LiveAudio plug-in (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 differs from the Macintosh version to the Windows version. With luck, this naming discrepancy will resolve itself over time as plug-in developers understand the scripter's need for consistency across platforms.

One other point that can help you decide the precise approach to take is which information about the plug-in — support for the data MIME type or the presence of a particular plug-in — is important to your page and scripts. If your scripts rely on the existence of a plug-in that you can script via LiveConnect, then be sure that the plug-in is present and enabled for the desired MIME type (so that the plug-in is ensured of loading when it encounters a reference to the URL of the external data).

But if you care only that a plug-in of any kind supports your data's MIME type, then you can simply make sure that any plug-in is enabled for your MIME type.

To help you jump-start the process in your scripts, I discuss three utility functions you can use in your own scripts. These functions are excerpts from a long listing (Listing 28-3), which is located in its entirety on the book's CD-ROM. The pieces not shown here are merely user interface elements that enable you to experiment with these functions.

Verifying a MIME type

Listing 28-3a is a function whose narrow purpose is to determine if the browser currently has plug-in support enabled for a given MIME type (in the *type/subtype* format as a string). The first `if` construction verifies that there is a mimeType object for the supplied MIME type string. If such an object exists, then the next `if` construction determines whether the `enabledPlugin` property of the mimeType object returns a valid object. If so, the function returns `true` — meaning that the MIME type has a plug-in (of unknown supplier) available to play the external media.

Listing 28-3a: **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) {
    if (navigator.mimeTypes[mime_type]) {
        if (navigator.mimeTypes[mime_type].enabledPlugin) {
            return true
        }
    }
    return false
}
```

Verifying a plug-in

In Listing 28-3b, 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 querying the browser about a known MIME type, the function inquires about the presence of 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 28-3b: **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) {
    plug_in = plug_in.toLowerCase()
    for (var i = 0; i < navigator.plugins.length; i++) {
        if (navigator.plugins[i].name.toLowerCase().indexOf(plug_in) != -1) {
            for (var j = 0; j < navigator.plugins[i].length; j++) {
                if (navigator.plugins[i][j].enabledPlugin) {
                    return true
                }
            }
            return false
        }
    }
    return false
}
```

The parameter for the `pluginIsReady()` function is a string consisting of the plug-in's name. As discussed earlier, the precise name may vary from OS to OS or from version to version. The function here assumes that you aren't concerned about plug-in versioning. It also assumes (with reasonably good experience behind the assumption) that a brand-name plug-in contains a string with the brand in it. Thus, the `pluginIsRead()` function simply looks for the existence of the passed name within the plugin object's `name` property. For example, this function accepts `"QuickTime"` as a parameter and agrees that there is a match with the plug-in named `"QuickTime Plug-in 4.1.1"`. The script loops through all registered plug-ins for a substring comparison (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 contains the parameter string). 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 28-3a). The conditional statement evaluates to either an object or `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`.

Verifying both plug-in and MIME type

The last utility function (Listing 28-3c) 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 `true`.

Listing 28-3c: **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(mime_type, plug_in) {
    if (mimeIsReady(mime_type)) {
        var plugInOfRecord = navigator.mimeTypes[mime_type].enabledPlugin
        plug_in = plug_in.toLowerCase()
        for (var i = 0; i < navigator.plugins.length; i++) {
            if (navigator.plugins[i].name.toLowerCase().indexOf(plug_in) != -1) {
                if (navigator.plugins[i] == plugInOfRecord) {
                    return true
                }
            }
        }
    }
    return false
}
```

This function starts by calling the `mimeIsReady()` function from Listing 28-3a. After that, the function resembles the one in Listing 28-3b 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. To see how these functions work on your NN3+ or IE5+/Mac browser, open the complete file (`lst28-03.htm`) from the CD-ROM. The actual listing also includes code that branches around IE for Windows and other browsers that don't support this way of inspecting MIME types and plug-ins.

Managing manual plug-in installation

If your scripts determine that a visitor 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 can contain step-by-step instructions with links to the plug-in's download site. The download site's page can 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. For instance, you must decode Macintosh files frequently from binhex format and then uncompress them 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 return to the `document.referrer`, which should be the page that sends 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 returning to play your cool media. Also consider letting the browser's own updating facilities handle the job (albeit not as smoothly in many cases) by simply loading the data into the page ready or not.

"Plug-in" detection in IE/Windows

IE4+ provides some built-in facilities that may take the place of plug-in detection in some circumstances. First of all, it's important to recognize that IE/Windows does not use the term "plug-in" in the same way that Netscape and IE/Mac use it. Due to the integration between IE and the Windows operating system, IE/Windows employs system-wide ActiveX controls to handle the job of rendering external content. Some of these controls are designed to be accessed from outside their walls, thus allowing client-side scripts to get and set properties or invoke methods built into the controls. These controls behave a lot like plug-ins, so you frequently see them referenced as "plug-ins," as they are in this book.

IE/Windows prefers the `<OBJECT>` tag for both loading the plug-in (ActiveX control) and assigning external content to it. One of the attributes of the OBJECT element is `CLASSID`, which points to a monstrously long string of hexadecimal numbers known as the GUID (Globally Unique Identifier). When the browser encounters one of these GUIDs, it looks into the Windows Registry to get the path to the actual plug-in file. If the plug-in is not installed on the user's machine, then the object doesn't load and any other HTML nested inside the <OBJECT> tag renders instead. Thus, you can display a static image placeholder or HTML message about the lack of the plug-in. But plug-in detection comes in most handy when your scripts need to communicate with the plug-in, such as directing an embedded Windows Media Player plug-in to change sound files or to play. When you build code around a scriptable plug-in, your scripts should make sure that the plug-in object is indeed present so they don't generate errors.

The idea of using the `<OBJECT>` tag instead of the `<EMBED>` tag is that the `<OBJECT>` tag loads a specific plug-in, whereas the MIME type of the data referenced by the `<EMBED>` tag lets the browser determine which plug-in to use for that MIME type. It's not uncommon, therefore, to see an `<OBJECT>` tag definition surround an `<EMBED>` tag — both referencing the same external data file. If the optimum plug-in fails to load, the `<EMBED>` tag is observed, and the browser tries to find any plug-in for the file's MIME type.

With an OBJECT element as part of the HTML page, the element itself is a valid object — even if the plug-in fails to load. Therefore, you must do more to validate the existence of the loaded plug-in than simply test for the existence of the OBJECT

element. To that end, you need to know at least one scriptable property of the plug-in. Unfortunately, not all scriptable plug-ins are fully documented, so you occasionally must perform some detective work to determine which scriptable properties are available. While you're on the search for clues, you can also determine the version of the plug-in and make it a minimum version that your OBJECT element allows to load.

Tracking down plug-in details

Not everyone has access to the Microsoft programming development environments (for example, Visual Basic) through which you can find out all kinds of information about an installed ActiveX control. If you don't have access, you can still dig deep to get most (if not all) of the information you need. The tools you can use include the Windows Registry Editor (`regedit`), The Evaluator (Chapter 13), and, of course, your text editor and IE4+/Windows browser. The following steps take you through finding out everything you need to know about the Windows Media Player control.

1. If you don't know the GUID for the Media Player (most people get it by copying someone else's code that employs it), you can use the Registry Editor (`regedit.exe`) to find it. Open the Registry Editor (in Win95/98/NT, choose Run from the Start menu and enter `regedit`; if that option is not available in your Windows version, search for the file named `regedit`).
2. Expand the `HKEY_CLASSES_ROOT` folder.
3. Scroll down to the nested folder named `CLSID`, and click that folder.
4. Choose Edit/Find, and enter **`Windows Media Player`**. If you were searching for a different plug-in, you would enter an identifying name (usually the product name) in this place.
5. Keep pressing F3 (Find Next) until the editor lands upon a folder whose default value (in the right side of the Registry Editor window) shows `Windows Media Player`.
6. The number inside curly braces next to the highlighted folder is the plug-in's GUID. Right-click the number and choose Copy Key Name. Paste the number into your document somewhere for future reference. Eventually, it will be part of the value assigned to the `CLASSID` attribute of the OBJECT element.
7. Expand the highlighted folder.
8. Click the folder named `InprocServer32`. The default value should show a pathname to the actual ActiveX control for the Windows Media Player plug-in.
9. Right-click the (`Default`) name for the path and choose Modify. The full pathname is visible in an editable field.
10. Armed with this pathname information, open My Computer and locate the actual file inside a directory listing.
11. Right-click the file and choose Properties.
12. Click the Version tab (if present).

13. Copy the version number (generally four sets of numbers delimited by commas), and paste it into your document for future reference. Eventually, it will be assigned to the `CODEBASE` attribute of the OBJECT element.

    You are now ready to try loading the plug-in as an object and look for properties you can test for.

14. Add an OBJECT tag to The Evaluator source code. This can go inside the HEAD or just before the `</BODY>` tag. For example, your tag should look something like the following:

```
<OBJECT ID="wmp" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=1,0,0,0">
</OBJECT>
```

    Copy and paste the numbers for the GUID and version. Two points to watch out for: First, be sure that the GUID value is preceded by `CLSID:` in the value assigned to `CLASSID`; second, be sure the version numbers are preceded by the prefix shown.

15. Load (or reload) the page in IE4+/Windows.

    At this point, the `wmp` object should exist. If the associated plug-in loads successfully, then the `wmp` object's properties include properties exposed by the plug-in.

16. Enter **wmp** into the bottom text box to inspect properties of the `wmp` object. Be patient: It may take many seconds for the retrieval of all properties.

    In case you can't readily distinguish between the OBJECT element object properties and properties of the scriptable plug-in, scroll down to the `wmp.innerHTML` property and its values. When an object loads successfully, any parameters that it accepts are reflected in the `innerHTML` for the OBJECT element. Each PARAM element has a name — the name of one of the scriptable properties of the plug-in.

17. Look for one of the properties that has some kind of value by default (in other words, other than an empty string or `false`). In Windows Media Player, this can be `CreationDate`. Use this property as an object detection condition in scripts that need to access the Windows Media Player properties or methods:

```
if (wmp && wmp.CreationDate) {
    // statements that "talk to" plug-in
}
```

Setting a minimum version number

The four numbers that you grab in Step 13 in the previous section represent the version of the plug-in as installed on your computer. Unless you have a way of verifying that your external content runs on earlier versions of the plug-in (if there are earlier versions), you can safely specify *your* version as the minimum.

Specificity rankings for the four numbers of a version decrease as you move from left to right. For example, Version 1,0,25,2 is later than 1,0,0,0; Version 2,0,0,0 is later

than both of them. If you specify 1,0,25,2, and the user has 1,0,24,0 installed, the plug-in does not load and the object isn't available for scripting. On the other hand, a user with 1,0,26,0 has the object present because the CODEBASE attribute for the version specifies a minimum allowable version to load.

When an object requires VBScript

Not all objects that load via the OBJECT element are scriptable through JScript. Occasionally, an object is designed so that its properties are exposed only to VBScript. This happens, for example, with the Microsoft Windows Media Rights Manager (DRM) object. To find out if the browser (operating system) is equipped with DRM, your page loads the object via the OBJECT element as usual; however, a separate VBScript section must access the object to test for the existence of one of its properties. Because script segments written in either language can access each other, this isn't a problem provided you know what the property or method is for the object. The following fragment from the Head section of a document demonstrates how JavaScript and VBScript can interact so that JavaScript code can branch based on the availability of DRM:

```
<HEAD>
<OBJECT ID="drmObj" HEIGHT="1" WIDTH="1"
CLASSID="CLSID:760C4B83-E211-11D2-BF3E-00805FBE84A6"></OBJECT>
<SCRIPT LANGUAGE="VBScript">
function hasDRM()
    on error resume next
    drmObj.StoreLicense("")
    if (err.number = 0) then
        hasDRM = true
    else
        hasDRM = false
    end if
end function
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript">
var gHasDRM
if (drmObj && hasDRM()) {
    gHasDRM = true
} else {
    gHasDRM = false
}
</SCRIPT>
</HEAD>
```

The JavaScript segment sets a Boolean global variable to indicate whether the object has loaded correctly. Part of the job is accomplished via the hasDRM() function in the VBScript segment. From VBScript, the drmObj object responds to the StoreLicense() method call, but it throws a VBScript error indicating that no parameter was sent along with the method. Any subsequent scripts in this page can use the gHasDRM global variable as a conditional expression before performing any actions requiring the object (which works in tandem with the Windows Media Player).

screen Object

Properties	Methods	Event Handlers
availHeight		
availLeft		
availTop		
availWidth		
bufferDepth		
colorDepth		
fontSmoothingEnabled		
height		
pixelDepth		
updateInterval		
width		

Syntax

Accessing `screen` object properties:

```
(All)         screen.property
(IE4+/NN6)    [window.]navigator.screen.property
```

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓			✓	✓	✓

About this object

Browsers other than from the earliest generations provide 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 also what the available width and height are once you take into account the operating system's screen-hogging interface elements (for example, the Windows taskbar and the Mac menu bar).

You can also access some of these property values in Navigator 3 if you use LiveConnect to access Java classes directly. Example code for this approach appears in the individual property listings.

Internet Explorer 4 provides a `screen` object, although it appears as a property of the `window` object in the IE4+ object model. Only three properties of the IE4+ `screen` object — `height`, `width`, and `colorDepth` — share the same syntax as NN4+'s `screen` object.

Properties

availHeight
availWidth
height
width

Value: Integer Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓			✓	✓	✓

With the availability of window sizing methods in version 4 browsers and later, your scripts may want to know how large the user's monitor is. This is particularly important if you set 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.

You can extract the gross height and width of the monitor 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.

But not every pixel of the screen's gross size is available as displayable area for a window. To the rescue come the `screen.availWidth` and `screen.availHeight` properties. For example, 32-bit Windows operating systems 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 menu bar occupies a top strip of the screen. While you can position and size windows so the menu bar partially covers them, 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 arrange a window so that it occupies all available space on the monitor, 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.resizeTo(screen.availWidth, screen.availHeight)
}
```

The preceding function positions the window appropriately on the Macintosh just below the menu bar so that the menu bar does not obscure the window. If, however, the client is running Windows and the user positions the taskbar at the top of the screen, the window is partially hidden under the taskbar (you cannot query the available screen space's coordinates). Also in Windows, the appearance is not exactly the same as a maximized window. See the discussion of the `window.resizeTo()` method in Chapter 16 for more details. Note that IE/Mac generally returns a value for `screen.availHeight` that is about 24 pixels fewer than the actual available height (even after taking into account the Mac menu bar).

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 runs Navigator 3 and Java is enabled, you can place the following script fragment in the Head portion of your document to set variables with screen width and height:

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

The `screenSize` variable is an object whose properties (`width` and `height`) contain the pixel measures of the current screen. This LiveConnect technique works only in NN3+ (IE does not provide direct access to Java classes). In fact, you can also extract the screen resolution (pixels per inch) in the same manner. The following statement, added after the preceding ones, sets the variable resolution to that value:

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

Related Items: `window.innerHeight`, `window.innerWidth`, `window.outerHeight`, `window.outerWidth` properties; `window.moveTo()`, `window.resizeTo()` methods.

availLeft
availTop

Value: Integer Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓					

The `availLeft` and `availTop` properties return the pixel measure of where (on the Windows OS) the available space of the screen begins. The only time these values are anything other than zero is when a user positions the taskbar along the left or top edges of the screen. For example, if the user positions the taskbar along the top of the screen, you do not want to position a window any higher than the 28 pixels occupied by the taskbar. Oddly, the `availTop` measure does not take into account the Macintosh menu bar, but Mac browsers treat the 0,0 coordinate for a window movement to be just below the menu bar anyway. Therefore, for NN4+, you can use the `availLeft` and `availTop` properties to move the window in a position where you can resize it to occupy the screen:

```
window.moveTo(screen.availLeft, screen.availTop)
window.resizeTo(screen.availWidth, screen.availHeight)
```

There are no corresponding properties for IE.

Example on the CD-ROM

Related Items: `screen.availWidth`, `screen.availHeight` properties; `window.moveTo()` method.

bufferDepth

Value: Integer Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

By default, IE does not use any offscreen buffering of page content. But adjusting the `bufferDepth` property enables you to turn on offscreen buffering and control

the color depth of the buffer. Using offscreen buffering may improve the smoothness of path-oriented animation through positioning.

The default value (buffering turned off) is `0`. By setting the property to `-1`, you instruct IE to set the color depth of the offscreen buffer to the same color depth as the screen (as set in the control panel). This should be the optimum value, but you can also force the offscreen buffer to have one of the following bit depths: `1`, `4`, `8`, `15`, `16`, `24`, or `32`.

Related Items: `screen.colorDepth`, `screen.pixelDepth` properties.

colorDepth
pixelDepth

Value: Integer Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility			✓	✓			✓	✓	✓

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

Both the `screen.colorDepth` and `screen.pixelDepth` properties return the number of color bits to which the color client computer's video display control panel is set. The `screen.colorDepth` value may take into account a custom color palette; so for NN4+, you may prefer to rely only on the `screen.pixelDepth` value. (IE4+, however, 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` are tuned for 8-bit display.

While LiveConnect in NN3 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 that NN3 users rely on this tactic.

Related Item: `screen.bufferDepth` property.

fontSmoothingEnabled

Value: Boolean Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Some versions of the Windows OS have a Display control panel setting for "Smooth Edges" on screen fonts. The `fontSmoothingEnabled` property lets your script see the state of that setting. This setting can affect, for example, which style sheet you enable because it has font specifications that look good only when smoothing is enabled. A default installation of Windows has this feature turned off. This property is not available on non-Windows versions of IE.

Related Items: None.

updateInterval

Value: Integer Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `updateInterval` property is the number of milliseconds between screen updates. The default value of zero lets IE arbitrate among the demands for screen updates in a highly animated setting. If you set this value to a large number, then more screen updates are accumulated in a buffer — preventing some animated steps from updating the screen.

Related Items: None.

userProfile Object

Properties	Methods	Event Handlers
	addReadRequest()	
	clearRequest()	
	doReadRequest()	
	getAttribute()	

Syntax

Accessing `userProfile` object methods:

```
(IE4+)  [window.]navigator.userProfile.method()
```

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

About this object

The `userProfile` object is an IE-specific (and Windows, at that) property that acts as the gateway to the user profile information that the client computer collects from the user. You can retrieve none of this information via JavaScript without permission from the user. Access to this information is performed in a strict sequence, part of which enables you to define how the request for this private information is worded when the user is presented with the request.

User profile data consists of nearly 30 fields of personal information about the user's contact information. Each of these fields has a name, which by and large conforms to the vCard standard. Your scripts can request one or more specific fields from the list, rather than having to deal with the entire set of fields.

The sequence for accessing this data entails four basic steps:

1. Put the request for each vCard field into a queue that is maintained in the browser's memory (via the `addReadRequest()` method).

2. Execute the batch request, which displays a detailed dialog box to the user (via the `doReadRequest()` method). If a user profile is in effect, the user sees which fields you are requesting plus the data in the vCard. The user then has the chance to deselect one or more of your choices — or disallow access completely.
3. Get each attribute by name (via the `getAttribute()` method). You invoke this method once for each vCard field.
4. Clear the queue of requests (via the `clearRequest()` method).

Returned values are strings. Thus, you can prefill the customer information for an order form or capture the information in hidden fields that are submitted with a visible form.

Listing 28-4 demonstrates the use of the four key methods of the `userProfile` object. After the page loads, it attempts to extract the data from every vCard field and displays both the attribute name and the value as associated with the current user profile in a table. Notice that the names of the attributes are hard-wired because the object does not provide a list of implemented attributes.

Listing 28-4: **Accessing userProfile Data**

```
<HTML>
<HEAD>
<TITLE>userProfile Object</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var attrs = ["Business.City","Business.Country","Business.Fax",
             "Business.Phone","Business.State","Business.StreetAddress",
             "Business.URL","Business.Zipcode","Cellular","Company",
             "Department","DisplayName","Email","FirstName",
             "Gender","Home.City","Home.Country","Home.Fax",
             "Home.Phone","Home.State","Home.StreetAddress",
             "Home.Zipcode","Homepage","JobTitle","LastName",
             "MiddleName","Notes","Office","Pager"]

function loadTable() {
    // make sure this executes only in IE4+ for Windows
    if ((navigator.userAgent.indexOf("Win") != -1) && navigator.userProfile) {
        var newRow, newCell, attrValue
        // queue up requests for every vCard attribute
        for (var i = 0; i < attrs.length; i++) {
            navigator.userProfile.addReadRequest("vCard." + attrs[i])
        }
        // dispatch the request to let user accept or deny access
```

Continued

Listing 28-4 *(continued)*

```
        navigator.userProfile.doReadRequest(1, "JavaScript Bible")
        // append rows to the table with attribute/value pairs
        for (var j = 0; j < attrs.length; j++) {
            newRow = document.all.attrTable.insertRow(-1)
            newRow.bgColor = "#FFFF99"
            newCell = newRow.insertCell(0)
            newCell.innerText = "vCard." + attrs[j]
            newCell = newRow.insertCell(1)
            // get the actual value
            attrValue = navigator.userProfile.getAttribute("vCard." + attrs[j])
            newCell.innerHTML = (attrValue) ? attrValue : " "
        }
        // clean up after ourselves
        navigator.userProfile.clearRequest()
    } else {
        alert("This example requires IE4+ for Windows.")
    }
}
</SCRIPT>
</HEAD>
<BODY onLoad="loadTable()">
<H1>userProfile Object</H1>
<HR>
<TABLE ID="attrTable" BORDER=1 CELLPADDING=5>
<TR BGCOLOR="#CCFFFF">
    <TH>vCard Property<TH>Value
</TR>

</TABLE>
</BODY>
</HTML>
```

It appears that the newer the version of Windows that the user runs, the more likely that user profile data is available. Even so, there may be little more than name and address data for those users who are careful not to fill out optional fields of Microsoft Web site forms requesting personal information.

Comparable information may be available from NN4+ users on any OS platform via signed scripts that access LDAP preferences. See the discussion earlier in this chapter about the `navigator.preference()` method.

Methods

addReadRequest("*attributeName*")

Returns: Boolean.

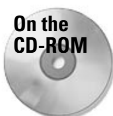
	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Before the user is asked for permission to reveal any personal information, you must queue up requests — even if there is just one field in which you are interested. For each field, use the `addReadRequest()` method and specify as the parameter a string of the attribute name. Acceptable attribute names are as follows:

```
vCard.Business.City
vCard.Business.Country
vCard.Business.Fax
vCard.Business.Phone
vCard.Business.State
vCard.Business.StreetAddress
vCard.Business.URL
vCard.Business.Zipcode
vCard.Cellular
vCard.Company
vCard.Department
vCard.DisplayName
vCard.Email
vCard.FirstName
vCard.Gender
vCard.Home.City
vCard.Home.Country
vCard.Home.Fax
vCard.Home.Phone
vCard.Home.State
vCard.Home.StreetAddress
vCard.Home.Zipcode
vCard.Homepage
vCard.JobTitle
vCard.LastName
vCard.MiddleName
vCard.Notes
vCard.Office
vCard.Pager
```

All attribute values are case-insensitive.

This method returns a Boolean value of `true` if the addition to the queue succeeds. A returned value of `false` usually means that the attribute value is not valid or that a request for that attribute name is already in the queue. If you fail to clear the queue after compiling one list of attributes, attempts to read the attribute result in a return value of `false`.



Example on the CD-ROM

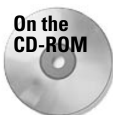
Related Items: `clearRequest()`, `doReadRequest()`, and `getAttribute()` methods.

clearRequest()

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

After retrieving the attributes whose names are stacked in the request queue, invoke the `clearRequest()` method to empty the queue. It is always good programming practice to clean up after yourself, especially when security concerns are involved.



Example on the CD-ROM

Related Items: `addReadRequest()`, `doReadRequest()`, and `getAttribute()` methods.

doReadRequest(*reasonCode*, *identification*[, *domain*[, *path*[, *expiration*]]])

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

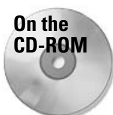
Once the names of the desired vCard attributes are stacked in the queue (via the `addReadRequest()` method), invoke the `doReadRequest()` method to prompt the user for the permission that your scripts need to gain access to the data. The user sees a detailed dialog box that lists the vCard fields you are requesting, as well as a description about your reason for wanting the data and who you are.

The first required parameter is an integer representing one of the standard descriptions as defined by the Internet Privacy Working Group. Associated text is displayed in the permission request dialog box that the user sees. The codes and their strings are as follows:

Code	***Description String***
0	Used for system administration.
1	Used for research and/or product development.
2	Used for completion and support of current transaction.
3	Used to customize the content and design of a site.
4	Used to improve the content of the site, including advertisements.
5	Used for notifying visitors about updates to the site.
6	Used for contacting visitors for marketing of services or products.
7	Used for linking other collected information.
8	Used by site for other purposes.
9	Disclosed to others for customization or improvement of the content and design of the site.
10	Disclosed to others, who may contact you, for marketing of services and/or products.
11	Disclosed to others, who may contact you, for marketing of services and/or products; you have the opportunity to ask a site not to do this.
12	Disclosed to others for any other purpose.

While these description strings are fixed, you do have an opportunity to include some customized information in the second parameter. The parameter is intended to enable you to identify the Web site or organization requesting the information. Standards recommendations suggest you include a URL to the site, as well. In any case, the second parameter can be any string. But it is not treated like HTML, so do not attempt to include a clickable link here.

Two optional parameters enable you to specify a domain and path within that domain for which the user permissions are to apply. Both of these parameters closely mirror their usage in cookies, but they also depend on the capability to set an expiration date via the fifth parameter. Through IE5.5, however, the expiration date parameter is ignored. Therefore, permissions expire when the user quits the browser (just like temporary cookies do).



Example on the CD-ROM

Related Items: `addReadRequest()`, `clearRequest()`, and `getAttribute()` methods.

getAttribute("*attributeName*")

Returns: String.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `getAttribute()` method attempts to retrieve the vCard data based on the items queued via the `addReadRequest()` method. A permission dialog box provides the user an opportunity to choose which of the requested items to reveal or to deny all access to the information. Only one attribute name is permitted as a parameter to the `getAttribute()` method, requiring that you invoke the method for each attribute you want to fetch.

Example on the CD-ROM

Related Items: `addReadRequest()`, `clearRequest()`, and `doReadRequest()` methods.