

JavaScript and Embedded Objects

Modern browsers support many technologies beyond (X)HTML, CSS, and JavaScript. A wide variety of extra functionality is available in the form of browser plug-ins, ActiveX controls, and Java applets. These technologies provide extended capabilities that can make Web pages appear more like applications than marked-up text. Embedded objects provide a natural complement to the limited capabilities of scripting languages like JavaScript.

Embedded objects come in many forms, but the most popular are multimedia in nature. A good example is Macromedia Flash files, which allow designers to add advanced vector graphics and animation to Web sites. Various other types of embedded video, sound, and live audio are also quite popular. Embedded Java applets are often included in pages that require more advanced graphics, network, or processing functionality.

Browsers provide the bridge that facilitates communication between JavaScript and embedded objects. The way this communication is carried out is essentially non-standardized, although browser vendors adhere to their own ad hoc “standards,” which are in widespread use. Even so, there are numerous concerns when dealing with embedded objects. First, including them makes the assumption that the user’s browser has the capability to handle such objects. Second, even if the user does have an appropriate extension installed, many users find embedded objects annoying because they increase download time while only occasionally improving the overall utility of the site. Third, users with older browsers and users on non-Windows platforms are often unable to use embedded objects because of lack of support.

This chapter introduces the way that JavaScript can be used to interact with embedded objects in most major browsers. Complex integration of objects with JavaScript requires more comprehensive information, which can be found at browser and plug-in vendor sites.

Java

Many think that JavaScript is a boiled-down form of Java because of the similarity in their names. The fact that JavaScript was originally called “LiveScript” suggests the mistake in drawing such a conclusion. While Java and JavaScript are both object-oriented languages, they are both commonly used on the Web, and the syntax of both resembles the syntax of C, they are in truth very different languages. Java is a class-based object-oriented language, whereas JavaScript is prototype-based. Java is strongly typed, whereas JavaScript is weakly typed. Java is compiled into platform-independent bytecode before execution, while JavaScript source code is generally interpreted directly by the browser. Java programs execute in a separate context called a “sandbox,” whereas JavaScript is interpreted in the context of the browser.

This last difference—in execution context—is very important. Java applets are nearly platform-independent, stand-alone programs designed to run in a restricted execution environment. There is a lot of theory that goes into the Java sandbox, but in essence applets run in a “virtual machine” that is somewhat isolated from the user’s browser and operating system. This isolation is designed to preserve platform independence as well as the security of the client’s machine.

Java applets are most often used to implement applications that require comprehensive graphics capabilities and network functionality. Java packages installed on the client machine provide networking code, graphics libraries, and user interface routines, often making it a much more capable language than JavaScript for some tasks. Common applications include applets that display real-time data downloaded from the Web (for example, stock tickers), interactive data browsing tools, site navigation enhancements, games, and scientific tools that perform calculations or act as visualization tools.

Including Applets

Before delving into the details of applet interaction, a brief review of how to include applets in your pages is in order. Traditionally, applets are included with the `<applet>` tag. The tag’s **code** attribute is then set to the URL of the .class file containing the applet, and the **height** and **width** attributes indicate the shape of the rectangle to which the applet’s input and output are confined; for example:

```
<applet code="myhelloworld.class" width="400" height="100"
  name="myhelloworld" id="myhelloworld">
<em>Your browser does not support Java!</em>
</applet>
```

Note how the `<applet>` tag’s **name** attribute (as well as **id** attribute) is also set. Doing so assigns the applet a convenient handle JavaScript can use to access its internals.

Although the use of `<applet>` is widespread, it has been deprecated under HTML 4 and XHTML. More appropriate is the `<object>` tag. It has a similar syntax:

```
<object classid="java:myhelloworld.class" width="400" height="100"
  name="myhelloworld" id="myhelloworld">
<em>Your browser does not support Java!</em>
</object>
```

NOTE *There are some problems with the use of the <object> syntax for including applets, the least of which is lack of support in older browsers. We will use the <applet> syntax, but you should be aware that it is preferable standards-wise to use <object> whenever possible.*

Initial parameters can be included inside the <applet> or <object> tag using the <param> tag, as shown here:

```
<applet code="myhelloworld.class" width="400" height="100"
  name="myhelloworld" id="myhelloworld">
<param name="message" value="Hello world from an initial parameter!" />
<em>Your browser does not support Java!</em>
</applet>
```

Java Detection

Before attempting to manipulate an applet from JavaScript, you must first determine whether the user's browser is Java-enabled. Although the contents of an <applet> tag are displayed to the user whenever Java is turned off or unavailable, you still need to write your JavaScript so that you do not try to interact with an applet that is not running.

The `javaEnabled()` method of the `Navigator` object returns a Boolean indicating whether the user has Java enabled. This method was first made available in IE4 and Netscape 3, the first versions of the browsers that support JavaScript interaction with Java applets. Using a simple `if` statement with this method should provide the most basic Java detection, as shown here:

```
if ( navigator.javaEnabled() )
{
  // do Java related tasks
}
else
  alert("Java is off");
```

Once support for Java is determined, then JavaScript can be used to interact with included applets.

Accessing Applets in JavaScript

The ability to communicate with applets originated with a Netscape technology called LiveConnect that was built into Netscape 3. This technology allows JavaScript, Java, and plug-ins to interact in a coherent manner and automatically handles type conversion of data to a form appropriate to each. Microsoft implemented the same capabilities in IE4, though not under the name LiveConnect. The low-level details of how embedded objects and JavaScript interact are complicated, unique to each browser, and even vary between different versions of the same browser. The important thing is that no matter what it is called, the capability exists in versions of IE4+ (except under Macintosh) and Netscape 3+ (although early versions of Netscape 6 have some problems), and Mozilla-based browsers.

Applets can be accessed through the `applets[]` array of the `Document` object or directly through `Document` using the applet's name. Consider the following HTML:

```
<applet code="myhelloworld.class" width="400" height="100"
  name="myhelloworld" id="myhelloworld">
<em>Your browser does not support Java!</em>
</applet>
```

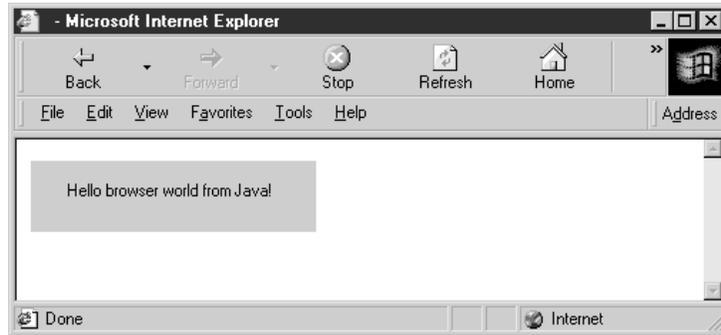
Assuming that this applet is the first to be defined in the document, it can be accessed in all of the following ways, with the last being preferred:

```
document.applets[0]
// or
document.applets["myhelloworld"]
// or the preferred access method
document.myhelloworld
```

The JavaScript properties, defined primarily under the browser object model and later by the DOM, of an `Applet` object are listed in Appendix B and consist of an unsurprising assortment of information reflecting the attributes of the (X)HTML `<applet>` tag for which it was defined. The relevant aspect to this JavaScript-Java communication discussion is the fact that all properties and methods of the applet's class that are declared `public` are also available through the `Applet` object. Consider the following Java class definition for the previous `myhelloworld` example. The output (when embedded as before) is shown in Figure 18-1.

```
import java.applet.Applet;
import java.awt.Graphics;
public class myhelloworld extends Applet
{
    String message;
    public void init()
    {
        message = new String("Hello browser world from Java!");
    }
    public void paint(Graphics myScreen)
    {
        myScreen.drawString(message, 25, 25);
    }
    public void setMessage(String newMessage)
    {
        message = newMessage;
        repaint();
    }
}
```

FIGURE 18-1
The output of the **myhelloworld** applet in Internet Explorer



Now comes the interesting part. Because the `setMessage()` method of the `myhelloworld` class is declared **public**, it is made available in the appropriate **Applet** object. We can invoke it in JavaScript as

```
document.myhelloworld.setMessage("Wow. Check out this new message!");
```

Before proceeding further with this example, it is very important to note that applets often require a significant amount of load time. Not only must the browser download the required code, but it also has to start the Java virtual machine and walk the applet through several initialization phases in preparation for execution. It is for this reason that it is never a good idea to access an applet with JavaScript before making sure that it has begun execution. The best approach is to use an **onload** handler for the **Document** object to indicate that the applet has loaded. Because this handler fires only when the document has completed loading, you can use it to set a flag indicating that the applet is ready for interaction. This technique is illustrated in the following example using the previously defined `myhelloworld` applet:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Applet Interaction Example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<script type="text/javascript">
<!--
var appletReady = false;
function changeMessage(newMessage) {
    if (!navigator.javaEnabled()) {
        alert("Sorry! Java isn't enabled!");
        return;
    }
}
```

```

    if (appletReady)
        document.myhelloworld.setMessage(newMessage);
    else
        alert("Sorry! The applet hasn't finished loading");
}
// -->
</script>
<body onload="appletReady = true;">
<applet code="myhelloworld.class" width="400" height="100"
    name="myhelloworld" id="myhelloworld">
<em>Your browser does not support Java!</em>
</applet>
<form action="#" method="get" onsubmit="return false;" name="inputForm"
    id="inputForm">
<input type="text" name="message" id="message" />
<input type="button" value="Change Message"
    onclick="changeMessage(document.inputForm.message.value);" />
</form>
</body>
</html>

```

The output of this script after changing the message is shown in Figure 18-2.

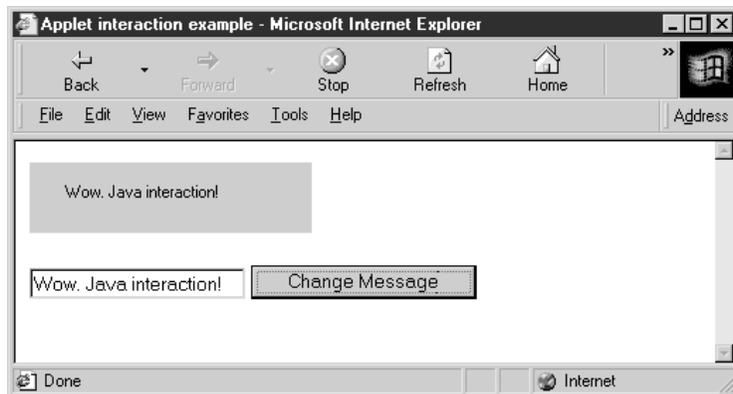
There are tremendous possibilities with this capability. If class instance variables are declared **public**, they can be set or retrieved as you would expect:

```
document.appletName.variableName
```

Inherited variables are, of course, also available.

NOTE *Java applets associated with applets defined in `<object>` tags receive the **public** properties and methods just as those defined in `<applet>` tags do. However, using `<object>` instead of `<applet>` is potentially less cross-browser compatible because Netscape 4 does not expose this HTML element to scripts.*

FIGURE 18-2
JavaScript can call **public** methods of Java applets.



Issues with JavaScript-Driven Applets

Experienced programmers might be asking at this point why one would choose to embed a Java applet alongside JavaScript in a page. One reason might be to avoid having to re-implement code in JavaScript that is readily available in Java. Another reason is that many people feel that user interfaces written in (X)HTML/CSS are easier to implement than in Java (though some people believe the opposite!). One major benefit of using a Web-based interface to drive an embedded applet is that changes to the interface can be made without the hassle of recompiling the Java code.

Discovering Interfaces

Many new programmers wonder how to find out what “hooks” are made available by a particular applet. An easy way to find out is to examine the source code (the .java file) associated with the applet. If it is not available, you can use a **for/in** loop on the appropriate **Applet** object to print out its properties. Anything that is not usually a property of an **Applet** browser object is a part of the interface defined by the applet’s class. However, this method is discouraged because it gives you no information about the type of arguments the applet’s methods expect. Generally, it’s not a good idea to drive an applet from JavaScript unless you know for sure how the interface it exposes should be used.

Type Conversion

The issue of type conversion in method arguments has serious bearing on JavaScript-driven applets. While most primitive JavaScript types are easily converted to their Java counterparts, converting complicated objects can be problematic. If you need to pass user-defined or non-trivial browser objects to applets, close examination of each browser’s type conversion rules is required. A viable option is to convert the JavaScript object to a string before passing it to an applet. The applet can then manually reconstruct the object from the string. A better option might be to retrieve the objects directly using the Java classes mentioned in the following section.

Security

A final issue is the fact that most browsers’ security models will prevent an applet from performing an action at the behest of JavaScript that the script could not otherwise perform on its own. This makes sense when one considers that Java is (in theory) designed to protect the user from malicious code. Experimentation with the restrictions placed on JavaScript-driven applets reveals inconsistent security policies among different browsers and versions.

Accessing JavaScript with Applets

Although it may come as a surprise, it is possible for Java applets to drive JavaScript. Internet Explorer, Netscape, and Mozilla-based browsers are capable of using the **netscape** Java package, which defines a family of class libraries for JavaScript interaction. In particular, the **JObject** class (**netscape.javascript.JObject**) allows an applet to retrieve and manipulate JavaScript objects in the current page. In addition, it affords an applet the ability to execute arbitrary JavaScript in the browser window as if it were a part of the page.

On the (X)HTML side of things, all that is required to enable this functionality is the addition of the **mayscript** attribute to the **<applet>** tag in question. The **mayscript** attribute is a nonstandard security feature used to prevent malicious applets from modifying the

documents in which they are contained. Omitting this attribute (theoretically) prevents the applet from crossing over into “browser space,” though enforcement by browsers is spotty.

While this is a powerful capability, Java-driven JavaScript is rarely used in practice. Details about these classes can be found in Java documentation for the specific browsers.

Plug-ins

Browser *plug-ins* are executable components that extend the browser’s capabilities in a particular way. When the browser encounters an embedded object of a type that it is not prepared to handle (e.g., something that isn’t HTML or other Web file type), the browser might hand the content off to an appropriate plug-in. If no appropriate plug-in is installed, the user is given the option to install one (assuming the page is properly written). Plug-ins consist of executable code for displaying or otherwise processing a particular type of data. In this way, the browser is able to hand special types of data, for example multimedia files, to plug-ins for processing.

Plug-ins are persistent in the browser in the sense that once installed, they remain there unless manually removed by the user. Most browsers come with many plug-ins already installed, so you may have used them without even knowing. Plug-ins were introduced in Netscape 2 but are supported, at least HTML–syntax-wise, by most major browsers, including Opera and Internet Explorer 3 and later. However, the actual component in the case of Internet Explorer is not a plug-in but instead an ActiveX control discussed later in the chapter. Plug-ins are a Netscape-introduced technology supported by many other browsers.

Embedding Content for Plug-Ins

Although never officially a part of any HTML specification, the `<embed>` tag is most often used to include embedded objects for Netscape and Internet Explorer. A Macromedia Flash file might be embedded as follows:

```
<embed id="demo" name="demo"
  src="http://www.javascriptref.com/examples/ch18/flash.swf"
  width="318" height="252" play="true" loop="false"
  pluginspage="http://www.macromedia.com/go/getflashplayer"
  swliveconnect="true"></embed>
```

The result of loading a page with this file is shown in Figure 18-3.

The most important attributes of the `<embed>` tag are `src`, which gives the URL of the embedded object, and `pluginspage`, which indicates to the browser where the required plug-in is to be found if it is not installed in the browser. Plug-in vendors typically make available the embedding syntax, so check their site for the value of `pluginspage`.

Recall that applets embedded with `<object>` tags are passed initial parameters in `<param>` tags. The syntax of `<embed>` is different in that initial parameters are passed using attributes of the element itself. For instance, in the preceding example the `play` attribute tells the plug-in to immediately begin playing the specified file.

FIGURE 18-3
An embedded
Flash file



The `<object>` element is the newer, official way to include embedded objects of any kind in your pages. However, `<object>` is not supported in Netscape browsers prior to version 4, and `<embed>` continues to be supported by new browsers. So it is unlikely that `<object>` will completely supplant `<embed>` any time in the near future. However, `<object>` and `<embed>` are very often used together in order to maximize client compatibility. This technique is illustrated in the later ActiveX section of this chapter.

MIME Types

So how does the browser know what kind of data is appropriate for each plug-in? The answer lies in Multipurpose Internet Mail Extension types, or *MIME types* for short. MIME types are short strings of the form *mediatype/subtype*, where the *mediatype* describes the general nature of the data and the *subtype* describes it more specifically. For example, GIF images have type *image/gif*, which indicates that the data is an image and its specific format is GIF (Graphics Interchange Format). In contrast, CSS files have type *text/css*, which indicates that the file is composed of plain text adhering to CSS specifications. The MIME major media types are *application* (proprietary data format used by some application), *audio*, *image*, *message*, *model*, *multipart*, *text*, and *video*.

Each media type is associated with at most one handler in the browser. Common Web media such as (X)HTML, CSS, plain text, and images are handled by the browser itself. Other media, for example, MPEG video and Macromedia Flash, are associated with the

appropriate plug-in (if it is installed). Keep in mind that a plug-in can handle multiple MIME types (for example, different types of video), but that each MIME type is associated with at most one plug-in. If one type were associated with more than one plug-in, the browser would have to find some way to arbitrate which component actually receives the data.

Detecting Support for MIME Types

Netscape 3+, Opera 4+, and Mozilla-based browsers provide an easy way to examine the ability of the browser to handle particular MIME types. The `mimeTypes[]` property of the `Navigator` object holds an array of `MimeType` objects. Some interesting properties of this object are shown in Table 18-1.

The browser hands embedded objects off to plug-ins according to the data that makes up each of these objects. A good way to think about the process is that the browser looks up MIME types and filename suffixes in the `mimeTypes` array to find the `enabledPlugin` reference to the appropriate plug-in. The programmer can therefore use the `mimeTypes` array to check whether the browser will be able to handle a particular kind of data.

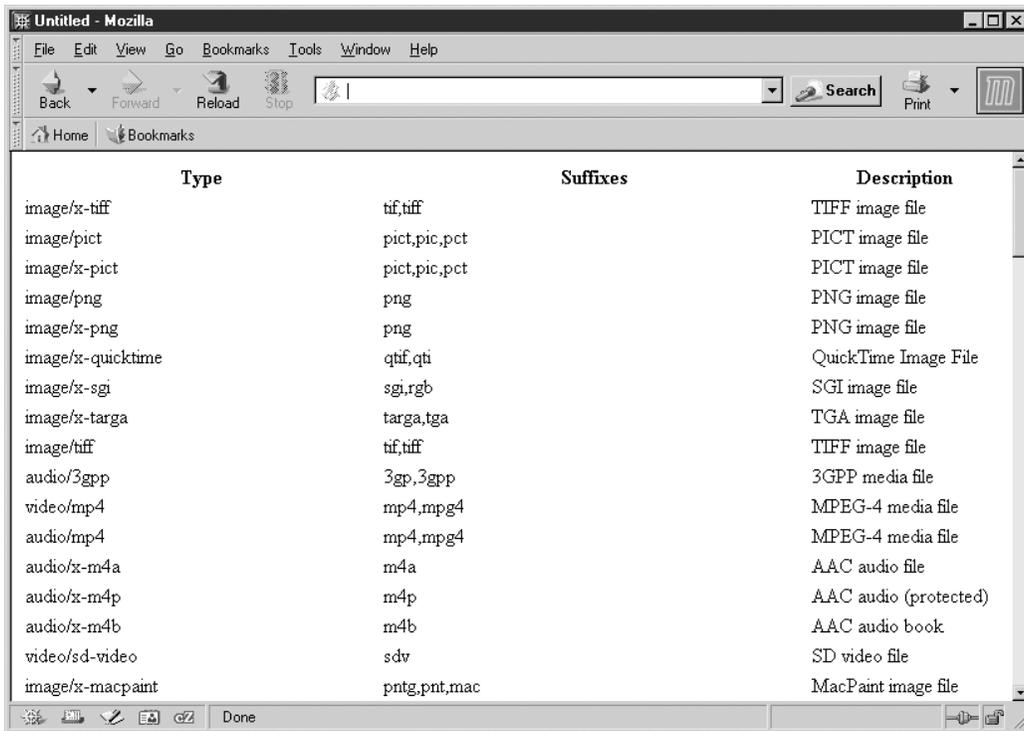
Before delving into this process, it might be insightful to see what MIME types your Netscape browser supports. The following code prints out the contents of the `mimeTypes[]` array.

```
if (navigator.mimeTypes)
{
    document.write("<table><tr><th>Type</th>");
    document.write("<th>Suffixes</th><th>Description</th></tr>");
    for (var i=0; i<navigator.mimeTypes.length; i++)
    {
        document.write("<tr><td>" + navigator.mimeTypes[i].type + "</td>");
        document.write("<td>" + navigator.mimeTypes[i].suffixes + "</td>");
        document.write("<td>" + navigator.mimeTypes[i].description
+ "</td></tr>");
    }
    document.write("</table>");
}
```

Part of the result in a typical installation of Mozilla-based browsers is shown in Figure 18-4. Of course, you can also access similar information by typing **about:plugins** in the location bar of Netscape and Mozilla-based browsers.

Property	Description
description	String describing the type of data the MIME type is associated with
enabledPlugin	Reference to the plug-in associated with this MIME type
suffixes	Array of strings holding the filename suffixes for files associated with this MIME type
type	String holding the MIME type

TABLE 18-1 Properties of the `MimeType` Object



Type	Suffixes	Description
image/x-tiff	tif,tiff	TIFF image file
image/pict	pict,pic,pct	PICT image file
image/x-pict	pict,pic,pct	PICT image file
image/png	png	PNG image file
image/x-png	png	PNG image file
image/x-quicktime	qtif,qti	QuickTime Image File
image/x-sgi	sgi,rgb	SGI image file
image/x-targa	targa,tga	TGA image file
image/tif	tif,tiff	TIFF image file
audio/3gpp	3gp,3gpp	3GPP media file
video/mp4	mp4,mpg4	MPEG-4 media file
audio/mp4	mp4,mpg4	MPEG-4 media file
audio/x-m4a	m4a	AAC audio file
audio/x-m4p	m4p	AAC audio (protected)
audio/x-m4b	m4b	AAC audio book
video/sd-video	sdv	SD video file
image/x-macpaint	pngt,pnt,mac	MacPaint image file

FIGURE 18-4 Contents of the `mimeTypes[]` array in Mozilla

To detect support for a particular data type, you first access the `mimeTypes[]` array by the MIME type string in which you are interested. If a `MimeType` object exists for the desired type, you then make sure that the plug-in is available by checking the `MimeType` object's `enabledPlugin` property. The concept is illustrated by the following code:

```
if (navigator.mimeTypes
    && navigator.mimeTypes["video/mpeg"]
    && navigator.mimeTypes["video/mpeg"].enabledPlugin)
    document.write('<embed src="/movies/mymovie.mpeg" width="300" ' +
        ' height="200"></embed>');
else
    document.write('');
```

If the user's browser has the `mimeTypes[]` array and it supports MPEG video (`video/mpeg`) and the plug-in is enabled, an embedded MPEG video file is written to the document. If these conditions are not fulfilled, then a simple image is written to the page. Note that the `pluginspage` attribute was omitted for brevity because the code has already detected that an appropriate plug-in is installed.

This technique of MIME type detection is used when you care only whether a browser supports a particular kind of data. It gives you no guarantee about the particular plug-in that will handle it. To harness some of the more advanced capabilities that plug-ins provide, you often need to know if a specific vendor's plug-in is in use. This requires a different approach.

Detecting Specific Plug-Ins

In Netscape 3+, Opera 4+, and Mozilla-based browsers, each plug-in installed in the browser has an entry in the `plugins[]` array of the `Navigator` object. Each entry in this array is a `Plugin` object containing information about the specific vendor and version of the component installed. Some interesting properties of the `Plugin` object are listed in Table 18-2.

Each `Plugin` object is an array of the `MimeType` objects that it supports (hence its `length` property). You can visualize the `plugins[]` and `mimeTypes[]` arrays as being cross-connected. Each element in `plugins[]` is an array containing references to one or more elements in `mimeTypes[]`. Each element in `mimeTypes[]` is an object referred to by exactly one element in `plugins[]`, the element referred to by the `MimeType`'s `pluginEnabled` reference.

You can refer to the individual `MimeType` objects in a `Plugin` element by using double-array notation:

```
navigator.plugins[0][2]
```

This example references the third `MimeType` object supported by the first plug-in.

More useful is to index the plug-ins by name. For example, to write all the MIME types supported by the Flash plug-in (if it exists!), you might write

```
if (navigator.plugins["Shockwave Flash"])
{
    for (var i=0; i<navigator.plugins["Shockwave Flash"].length; i++)
        document.write("Flash MimeType: " +
            navigator.plugins["Shockwave Flash"][i].type + "<br />");
}
```

Of course, as with all things plug-in-related, you need to read vendor documentation very carefully in order to determine the *exact* name of the particular plug-in in which you are interested.

Property	Description
description	String describing the nature of the plug-in. Exercise caution with this property because this string can be rather long.
name	String indicating the name of the plug-in.
length	Number indicating the number of MIME types this plug-in is currently supporting.

TABLE 18-2 Some Interesting Properties of the `Plugin` Object

To illustrate the composition of the **Plugin** object more clearly, the following code prints out the contents of the entire `plugins[]` array:

```
for (var i=0; i<navigator.plugins.length; i++)
{
  document.write("Name: " + navigator.plugins[i].name + "<br />");
  document.write("Description: " + navigator.plugins[i].description + "<br />");
  document.write("Supports: ");
  for (var j=0; j<navigator.plugins[i].length; j++)
    document.write(" &nbsp;" + navigator.plugins[i][j].type);
  // the nonbreaking space included so the types are more readable
  document.write("<br /><br />");
}
```

The results are shown in Figure 18-5.

Dealing with Internet Explorer

One thing to be particularly conscious of is that Internet Explorer defines a faux `plugins[]` array as a property of **Navigator**. It does so in order to prevent poorly written Netscape-specific scripts from throwing errors while they probe for plug-ins. Under Internet Explorer, you have some reference to plug-in-related data through the `document.embeds[]` collection. However, probing for MIME types and other functions is not supported, since Explorer actually uses ActiveX controls to achieve the function of plug-ins included via an `<embed>` tag. For more information on using JavaScript with ActiveX, see the section entitled “ActiveX”

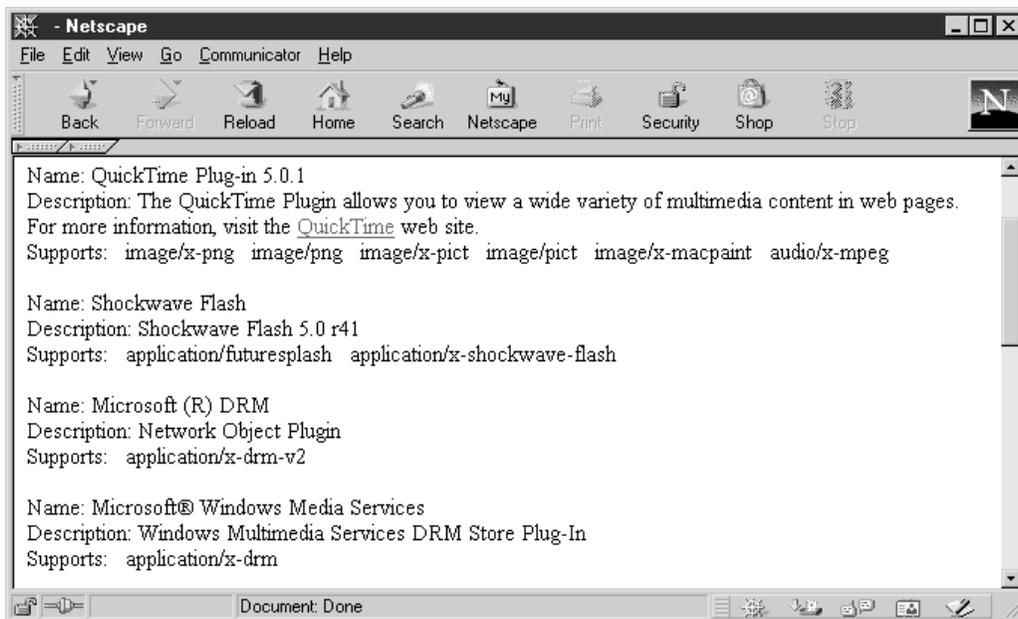


FIGURE 18-5 Example contents of the `navigator.plugins[]` array

later in this chapter. For now, simply consider that to rely solely on information from `navigator.plugins[]` without first doing some browser detection can have some odd or even disastrous consequences.

Interacting with Plug-Ins

By now you might be wondering why one would want to detect whether a specific plug-in will be handling a particular MIME type. The reason is that, like Java applets, plug-ins are LiveConnect-enabled in Netscape 3+, Internet Explorer 4+, and Mozilla-based browsers. This means that plug-ins can implement a public interface through which JavaScript can interact with them. This capability is most commonly used by multimedia plug-ins to provide JavaScript with fine-grained control over how video and audio are played. For example, plug-ins often make methods available to start, stop, and rewind content as well as to control volume, quality, and size settings. The developer can then present the user with form fields that control the behavior of the plug-in through JavaScript.

This capability works in the reverse direction as well. Embedded objects can invoke JavaScript in the browser to control navigation or to manipulate the content of the page. The more advanced aspects of this technology are beyond the scope of this book, but common aspects include functions that plug-ins are programmed to invoke when a particular event occurs. Like a JavaScript event handler, the plug-in will attempt to invoke a function with a specific name at a well-defined time, for example, when the user halts playback of a multimedia file. To prevent namespace collisions with other objects in the page, these methods are typically prefixed with the **name** or **id** attribute of the `<object>` or `<embed>` of the object instance.

As with applets, there remains the issue of how the JavaScript developer knows which methods the plug-in provides and invokes. The primary source for this information is documentation from the plug-in vendor. But be warned: These interfaces are highly specific to vendor, version, and platform. When using LiveConnect capabilities, careful browser and plug-in sensing is usually required.

We now have most of the preliminary information required in order to detect and interact safely with plug-ins. There is, however, one final aspect of defensive programming to cover before jumping into the interaction itself.

Refreshing the Plug-Ins Array

Suppose you have written some custom JavaScript to harness the capabilities provided by a specific plug-in. When users visit your page without the plug-in they are prompted to install it because you have included the proper **pluginspage** attribute in your `<embed>`. Unfortunately, if a user visits your page without the plug-in, agrees to download and install it, and then returns to your page, your JavaScript will not detect that the browser has the required plug-in. The reason is that the `plugins[]` array needs to be refreshed whenever a new plug-in is installed (a browser restart will work as well).

Refreshing the `plugins[]` array is as simple as invoking its **refresh()** method. Doing so causes the browser to check for newly installed plug-ins and to reflect the changes in the `plugins[]` and `mimeTypes[]` arrays. This method takes a Boolean argument indicating whether the browser should reload any current documents containing an `<embed>`. If you supply **true**, the browser causes any documents (and frames) that might be able to take advantage of the new plug-in to reload. If **false** is passed to the method, the `plugins[]`

array is updated, but no documents are reloaded. A typical example of the method's use is found here:

```
<em>If you have just installed the plugin, please <a
href="javascript:navigator.plugins.refresh(true)">reload the page with
plugin support</a></em>
```

Of course, this should be presented only to users of Netscape, Opera, or Mozilla-based browsers where plug-ins are supported in the first place.

Interacting with a Specific Plug-In

Nearly everything that was true of applet interaction remains true for plug-ins as well. Applets are accessed through the **Document** object, using the applet's **name** or **id** attribute. Similarly, the plug-in handling data embedded in the page is accessed by the **name** attribute of the **<embed>** tag that includes it. As with applets, you need to be careful that you do not attempt to access embedded data before it is finished loading. The same technique of using the **onload** handler of the **Document** to set a global flag indicating load completion is often used. However, one major difference between applets and plug-ins is that as far as the DOM specification is concerned, the **<embed>** tag doesn't exist, nor do plug-ins. Despite the fact that their use, particularly in the form of Flash, is so widespread, the specification chooses not to acknowledge their dominance and try to standardize their use.

To illustrate interaction with plug-ins, we show a simple example using a Macromedia Flash file. The first thing to note is that there are two plug-in names corresponding to Flash players capable of LiveConnect interaction. They are "Shockwave Flash" and "Shockwave Flash 2.0." Second, consulting Macromedia's documentation reveals that the **<embed>** tag should have its **swliveconnect** attribute set to **true** (though it does not appear to be required for this example) if you wish to use JavaScript to call into the Flash player.

You can find a list of methods supported by the Flash player at Macromedia's Web site (for example, at <http://www.macromedia.com/support/flash/publishexport/scriptingwithflash/>). The methods we will use in our simple example are **GotoFrame()**, **IsPlaying()**, **Play()**, **Rewind()**, **StopPlay()**, **TotalFrames()**, and **Zoom()**. The following example controls a simple Flash file extolling the wonders of JavaScript.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Simple Flash control example (Netscape and Mozilla only)</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<script type="text/javascript">
<!--
var pluginReady = false;
var pluginAvailable = false;
if (document.all) alert("Demo for netscape only");
function detectPlugin()
{
// if the appropriate plugin exists and is configured
// then it is ok to interact with the plugin
if (navigator.plugins &&
    (navigator.plugins["Shockwave Flash"] &&
```

```

        navigator.plugins["Shockwave Flash"]["application/x-shockwave-flash"])
    ||
    (navigator.plugins["Shockwave Flash 2.0"] &&
     navigator.plugins["Shockwave Flash 2.0"]["application/x-shockwave-flash"]))
    )
    pluginAvailable = true;
}

function changeFrame(i)
{
    if (!pluginReady || !pluginAvailable)
        return;
    if (i>=0 && i<document.demo.TotalFrames())
        // function expects an integer, not a string!
        document.demo.GotoFrame(parseInt(i));
}

function play()
{
    if (!pluginReady || !pluginAvailable)
        return;
    if (!document.demo.IsPlaying())
        document.demo.Play();
}

function stop()
{
    if (!pluginReady || !pluginAvailable)
        return;
    if (document.demo.IsPlaying())
        document.demo.StopPlay();
}

function rewind()
{
    if (!pluginReady || !pluginAvailable)
        return;
    if (document.demo.IsPlaying())
        document.demo.StopPlay();

    document.demo.Rewind();
}

function zoom(percent)
{
    if (!pluginReady || !pluginAvailable)
        return;
    if (percent > 0)
        document.demo.Zoom(parseInt(percent));
    // method expects an integer
}
//-->
</script>
</head>
<body onload="pluginReady=true; detectPlugin();">

<!-- Note: embed tag will not validate against -->
<embed id="demo" name="demo"

```

```

src="http://demos.javascriptref.com/jscript.swf"
width="318" height="300" play="false" loop="false"
pluginspage="http://www.macromedia.com/go/getflashplayer"
swliveconnect="true"></embed>

<form name="controlform" id="controlform" action="#" method="get">
<input type="button" value="Start" onclick="play();" />
<input type="button" value="Stop" onclick="stop();" />
<input type="button" value="Rewind" onclick="rewind();" /><br />
<input type="text" name="whichframe" id="whichframe" />
<input type="button" value="Change Frame"
  onclick="changeFrame(controlform.whichframe.value);" /><br />
<input type="text" name="zoomvalue" id="zoomvalue" />
<input type="button" value="Change Zoom"
  onclick="zoom(controlform.zoomvalue.value);" />
  (greater than 100 to zoom out, less than 100 to zoom in)<br />
</form>
</body>
</html>

```

The example—stopped in the middle of playback and zoomed in—is shown in Figure 18-6.

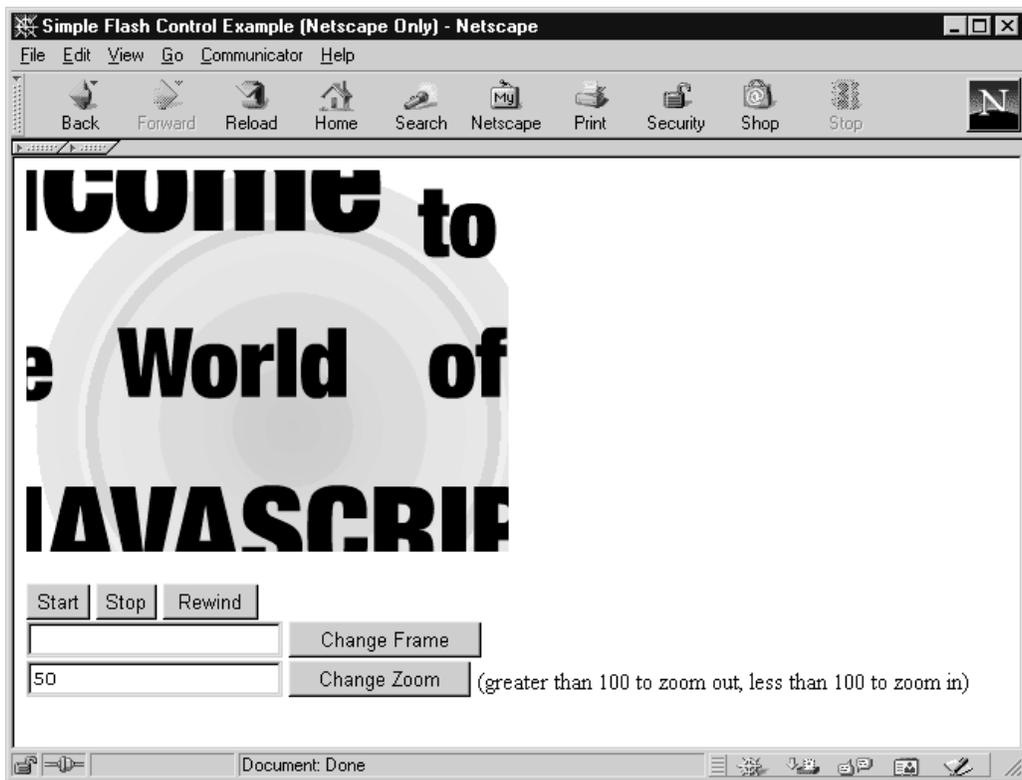


FIGURE 18-6 The scriptable Flash plug-in lets us zoom in on the Flash file.

There exist far more powerful capabilities than the previous example demonstrates. One particularly useful aspect of Flash is that embedded files can issue commands using `FSCommand()` that can be “caught” with JavaScript by defining an appropriately named function. Whenever an embedded Flash file in a LiveConnect-enabled browser issues an `FSCommand()`, the Flash file crosses over into browser territory to invoke the `name_doFSCommand()` method if one exists. The `name` portion of `name_doFSCommand()` corresponds to the `name` or `id` of the element in which the object is defined. In the previous example, the Flash file would look for `demo_doFSCommand()` because the file was included in an `<embed>` with `name` equal to “demo.” Common applications include alerting the script when the data has completed loading and keeping scripts apprised of the playback status of video or audio. As with other more advanced capabilities, details about these kinds of *callback functions* can be obtained from the plug-in vendors.

ActiveX

ActiveX is a Microsoft component object technology enabling Windows programs to load and use other programs or objects at runtime. ActiveX controls are basically subprograms launched by the browser that can interact with page content. For example, if a `<textarea>` provided insufficient editing capabilities for a particular task, the page author might include an ActiveX control that provides an editor interface similar to that of MS Word.

While on the surface ActiveX controls might seem a lot like Java applets, the two technologies are not at all alike. For one, once an ActiveX control is installed on the user’s machine, it is given greater access to the local system. This loosened security stance means that controls can access and change files, and do all manner of other powerful yet potentially unsavory things. Since ActiveX controls are executable code, they are built for a specific operating system and platform. This means that they are minimally supported outside of Internet Explorer, and not at all outside of Windows.

Whereas Java applets are downloaded when they are needed, ActiveX controls are, like plug-ins, persistent once they are installed. This installation process is often automatic, which is both good and bad. It is good in the sense that it obviates the need to have the user manually install a required component. But it is also a security risk because most users could be easily fooled into accepting the installation of a malicious control. We’ll have more to say about the security of ActiveX controls in Chapter 22.

Including ActiveX Controls

An ActiveX control is embedded in the page using an `<object>` tag with the `classid` attribute specifying the GUID (Globally Unique Identifier) of the ActiveX control you wish to instantiate. The syntax is similar to that of the `<object>` syntax for the inclusion of applets. Parameters are passed using `<param>` elements, and anything included between the `<object>`’s opening and closing tags is processed by non-`<object>`-aware browsers; for example:

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#
version=6,0,40,0" name="demoMovie" id="demoMovie" width="318" height="252">
<param name="movie"
value="http://www.javascriptref.com/examples/ch18/flash.swf" />
```

```

<param name="play" value="true" />
<param name="loop" value="false" />
<param name="quality" value="high" />

<em>Your browser does not support ActiveX!</em>

</object>

```

This example defines an embedded Flash file for use with an ActiveX control. In general, ActiveX controls have **classid** attributes beginning with “clsid:.” We saw another possibility in a previous section where the **classid** began with “java:.” In general, the **classid** attribute specifies the unique identifier of the control for which the data is intended. The **classid** value for each ActiveX control is published by the vendor, but it is also commonly inserted by Web development tools such as Macromedia Dreamweaver (www.macromedia.com/dreamweaver).

The final item of note is the **codebase** attribute specifying the version of the ActiveX binary that is required for this particular object. The **classid** and **codebase** attributes serve the function that manual probing of plug-ins does under Netscape. If the user’s machine doesn’t have the required control or version, the user will be prompted to download it from the given location.

Cross-Browser Inclusion of Embedded Objects

By far the best way to ensure the cross-browser compatibility of your pages is to use a combination of ActiveX controls and plug-in syntax. To accomplish this, use an **<object>** intended for IE/Windows ActiveX controls and include within it an **<embed>** intended for Netscape and IE/Macintosh plug-ins. The technique is illustrated in the following example:

```

<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#\
version=6,0,40,0"
  name="demoMovie" id="demoMovie" width="318" height="252">
<param name="movie" value=http://www.javascriptref.com/examples/ch18/flash.swf
  />
<param name="play" value="true" />
<param name="loop" value="false" />
<param name="quality" value="high" />

<embed src="http://www.javascriptref.com/examples/ch18/flash.swf"
  width="318" height="252" play="true" loop="false" quality="high"
  pluginspage="http://www.macromedia.com/go/getflashplayer">
<noembed>
  Error: No Object or Embed Support
</noembed>
</embed>

</object>

```

Browsers that do not understand **<object>** will see the **<embed>**, whereas browsers capable of processing **<object>** will ignore the enclosed **<embed>**. Using **<object>** and **<embed>** in concert maximizes the possibility that the user will be able to process your content.

Interacting with ActiveX Controls

JavaScript can be used to interact with ActiveX controls in a manner quite similar to plug-ins. A control is accessible under the **Document** object according to the **id** of the **<object>** that included it. If the required control isn't available, Internet Explorer automatically installs it (subject to user confirmation) and then makes it available for use.

NOTE You may have to include the *mayscript* attribute in the **<object>** to enable callback functions.

Any methods exposed by the control are callable from JavaScript in the way applet or plug-in functionality is called. Simply invoke the appropriate function of the **<object>** in question. To invoke the **Play()** method of the control in the previous example, you'd write

```
document.demoMovie.Play();
```

As a quick demonstration, we recast the previous example so it works in both Netscape and Internet Explorer browsers.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Cross-browser Flash Control Example </title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<script type="text/javascript">
<!--
var dataReady = false;
var pluginAvailable = false;
function detectPlugin()
{
    if (navigator.plugins &&
        ((navigator.plugins["Shockwave Flash"] &&
            navigator.plugins["Shockwave Flash"]["application/x-shockwave-flash"])
            ||
            (navigator.plugins["Shockwave Flash 2.0"] &&
            navigator.plugins["Shockwave Flash 2.0"]["application/x-shockwave-flash"]))
        pluginAvailable = true;
    return(pluginAvailable);
}

function changeFrame(i)
{
    if (!dataReady)
        return;
    // Some versions of the ActiveX control don't support TotalFrames,
    // so the check is omitted here. However, the control handles values
    // out of range gracefully.
    document.demo.GotoFrame(parseInt(i));
}

function play()
```

```
{
    if (!dataReady)
        return;
    if (!document.demo.IsPlaying())
        document.demo.Play();
}

function stop()
{
    if (!dataReady)
        return;
    if (document.demo.IsPlaying())
        document.demo.StopPlay();
}

function rewind()
{
    if (!dataReady)
        return;
    if (document.demo.IsPlaying())
        document.demo.StopPlay();
    document.demo.Rewind();
}

function zoom(percent)
{
    if (!dataReady)
        return;
    if (percent > 0)
        document.demo.Zoom(parseInt(percent));
}
//-->
</script>
</head>
<body onload="dataReady = true;">
<object id="demo" classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
width="318"
height="300"
codebase="http://active.macromedia.com/flash2/cabs/swflash.cab#version=5,0,0,0">
<param name="movie" value="http://demos.javascriptref.com/jscript.swf" />
<param name="play" value="false" />
<param name="loop" value="false" />
<script type="text/javascript">
<!--
    if (detectPlugin())
    {
        document.write('<embed name="demo" src="http://demos.javascriptref.com/
jscript.swf" width="318" height="300"
play="false" loop="false" pluginspage="http://www.macromedia.com/shockwave/download/
index.cgi?P1_Prod_
Version=ShockwaveFlash5" swliveconnect="true"></embed>');
    }
    else
    {
        // you can write an image in here in a "real" version
        document.write('Macromedia Flash is required for this demo');
    }
}
```

```

//-->
</script>
<noscript>
  JavaScript is required to demonstrate this functionality!
</noscript>
</object>
<form name="controlForm" id="controlForm" onsubmit="return false;" action="#"
method="get">
<input type="button" value="Start" onclick="play();" />
<input type="button" value="Stop" onclick="stop();" />
<input type="button" value="Rewind" onclick="rewind();" /><br />
<input type="text" name="whichFrame" id="whichFrame" />
<input type="button" value="Change Frame"
  onclick="changeFrame(controlForm.whichFrame.value);" /><br />
<input type="text" name="zoomValue" id="zoomValue" />
<input type="button" value="Change Zoom"
  onclick="zoom(controlForm.zoomValue.value)" /> (greater than 100 to zoom out, less
  than 100 to zoom in)<br />
</form>
</body>
</html>

```

You might wonder if ActiveX controls can do everything plug-ins can. The answer: yes, and even more. For example, data handled by ActiveX controls can take full advantage of callback functions, so everything that is possible with a plug-in is possible with ActiveX. Further, because data destined for ActiveX is embedded in **<object>** elements, it can take full advantage of the **<object>** event handlers defined in (X)HTML. Interestingly, there seems to be more robust support for ActiveX in VBScript than in JavaScript. This is most likely a result of the fact that as a Microsoft technology, VBScript is more closely coupled with Microsoft's COM. For more information on ActiveX, see <http://www.microsoft.com/com/tech/activex.asp>.

Summary

Embedded objects provide the means with which you can expand the capabilities of your pages to include advanced processing, network, and multimedia tasks. Web browsers support Java applets and ActiveX controls and/or Netscape plug-ins. JavaScript can interact with all forms of embedded objects to some degree. Typically, the object handling the embedded content is addressable under the **Document** object (as its **id** or **name**). When embedding content, it is recommended to write cross-browser scripts capable of interacting with both ActiveX controls and plug-ins.

This chapter served as an introduction to what is possible with embedded objects. A large part of ActiveX and plug-in capabilities are specific to the browser, vendor, and platform, so the best way to find information about these technologies is from the ActiveX control or plug-in vendors themselves. Because of the large number of browser bugs and documentation inconsistencies, often interaction with embedded objects is best carried out through a JavaScript library written with these subtleties in mind. Many such libraries can be found on the Web.

Embedded objects provide a way to *enhance* your site, not replace it. Pages should always degrade gracefully, so that they can be used by those on alternative platforms or who choose not to install plug-in, ActiveX, or Java technology. Sites that *require* a specific technology are very frustrating to use for the segment of the population that prefers an operating system and browser configuration other than Windows/Internet Explorer. As we discussed in the last chapter, detection techniques should always be employed to avoid locking users out of sites based upon technology limitations or client differences.