15

CHAPTER

Plug-ins, ActiveX Controls, and Java Applets

In the last chapter, we saw how scripts can be added to XHTML documents. Scripts can manipulate a variety of form elements and, in the case of so-called Dynamic HTML, the page elements themselves. Scripts also are used to access embedded binary objects. As discussed in Chapter 9, embedded objects can be used to bring multimedia, such as sounds and movies, to the Web. They also can be used to add small executable programs to a page. Binary objects come in many forms, including Netscape plug-ins, Microsoft ActiveX controls, and Java applets. Each of these requires special markup. In the future, all included media types will be added with the **object** element. However, until such standardization is universally supported, it is useful to study each individual technology with particular emphasis on how it can intersect with markup.

Scripting, Programming, and Objects

You might wonder why this chapter is separate from the last one. With both scripts and embedded objects, the interactivity takes place on the client side. What's the difference? Why distinguish between scripting and objects? Remember the point of Web client-side scripting—small bits of interpreted code used to add a bit of functionality to a page or fill the gaps in an application. Scripting is *not* necessarily as complex or general as programming, although it often seems as if it is. Programming is more generalized than scripting; programming enables you to create just about anything that you can imagine, although it tends to be more complex in some sense than scripting. Think about checking the data fields of a form; you need only a few lines of JavaScript to make sure the fields are filled. Now consider trying to create something sophisticated, such as a full-blown video game within a Web page. This takes more than a few lines of JavaScript code, and probably should be programmed in a language such as Java, C/C++, or Visual Basic. In fact, building objects is not trivial. It can require significant knowledge of programming. Fortunately for most casual Web page designers, putting together a custom object probably isn't necessary as existing components can be used. This chapter discusses each of the object technologies, as well as how such objects can be inserted into a Web page in conjunction with HTML/XHTML markup and scripting.

Plug-ins

Plug-ins such as the Flash player, QuickTime player, and others, are small helper programs (components) that extend the browser to support new functionality. Plug-ins are primarily a Netscape technology and have been around since Netscape Navigator 2. They are supported by some other browsers, notably Opera (www.opera.com). The **embed** element used to reference plug-ins is also supported under Internet Explorer, although it does result in the launch of an ActiveX control (a similar Microsoft technology discussed later in the chapter). Although plug-ins can go a long way toward extending the possible capabilities of a browser, the technology does have its drawbacks. Users must locate and download plug-ins, install them, and occasionally even restart their browsers. Many users find this rather complicated. Beginning with Netscape 4.*x*, some installation relief was found with somewhat self-installing plug-ins and other features, but plug-ins can still be occasionally troublesome for users. To further combat this problem, many of the most commonly requested plug-ins, such as Macromedia's Flash, are being included as a standard feature with Netscape and other browsers. However, even if installation were not such a problem, plug-ins are not available on every machine; an executable program, or *binary*, must be created for each particular operating system. Because of this machine-specific approach, many plug-ins work only on Windows-based systems. A decreasing number of plug-ins work on Macintosh and even less on Linux or UNIX. Finally, each plug-in installed on a system is a persistent extension to the browser, and takes up memory and disk space.

The benefit of plug-ins is that they can be well integrated into Web pages. You include them by using the **<embed>** or **<object>** tags. Typically, the **<embed>** syntax is used, but the **<object>** syntax is the preferred method because it is part of the XHTML specification and will, therefore, validate. Eventually, **<object>** will supplant **<embed>** completely, but for now it is very common in Web pages. In general, the **embed** element takes a **src** attribute to specify the URL of the included binary object. The **height** and **width** attributes often are used to indicate the pixel dimensions of the included object, if it is visible. To embed a short Audio Video Interleaved (AVI) format movie called welcome.avi that can be viewed by a video plug-in such as those generally installed with many browsers, use the following markup fragment:

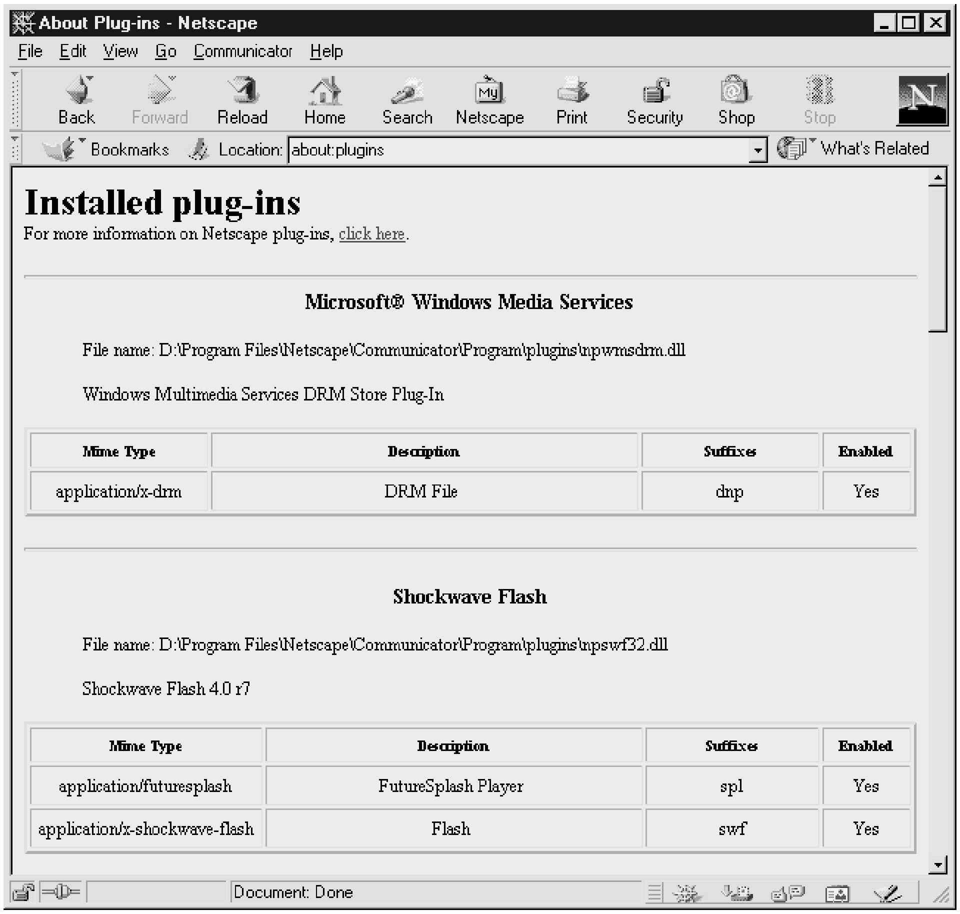
```
<embed src="welcome.avi" height="100" width="100"></embed>
```

The **<embed>** tag displays the plug-in (in this case a movie) as part of the HTML/XHTML document.

A browser can have many plug-ins installed. To check which plug-ins are installed in Netscape, the user can enter a strange URL, such as **about:plugins**, or look under the browser's Help menu for the option "About Plug-ins." The browser will show a list of plug-ins that are installed, the associated MIME type that will invoke each plug-in, and information as to whether that plug-in is enabled. Figure 15-1 shows an example of the plug-in information page.

<embed> Syntax

The primary way to load plug-ins for Netscape browsers is to use the nonstandard **<embed>** tag. It is preferable to use the **object** element, which is part of the HTML and XHTML specification, but **<object>** does not always work well, particularly in older


About Plug-ins - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: about:plugins What's Related
Document: Done

Installed plug-ins

For more information on Netscape plug-ins, click here.

Microsoft® Windows Media Services

File name: D:\Program Files\Netscape\Communicator\Program\plugins\npwmsdrm.dll

Windows Multimedia Services DRM Store Plug-In

Mime Type	Description	Suffixes	Enabled
application/x-drm	DRM File	dnp	Yes

Shockwave Flash

File name: D:\Program Files\Netscape\Communicator\Program\plugins\npswf32.dll

Shockwave Flash 4.0 r7

Mime Type	Description	Suffixes	Enabled
application/futuresplash	FutureSplash Player	spl	Yes
application/x-shockwave-flash	Flash	swf	Yes

FIGURE 15-1 About Plug-ins listing

browsers. Furthermore, many developers and editors seem stuck on the **<embed>** syntax. For backward compatibility, you might have to use both forms, as shown later in this chapter. You can find the general syntax of an **<embed>** tag in the element reference in Appendix A.

The most important attribute for an **<embed>** tag probably is **src**, which is set to the URL of the data object that is to be passed to the plug-in and embedded in the page. The browser generally determines the MIME type of the file—and thus the plug-in to pass the data to—by the filename suffix. For example, a file such as test1.dcr would be mapped to a MIME type of application/x-director and passed to a Shockwave for Director plug-in. In some cases, however, the plug-in to use with a particular **<embed>** tag is not obvious. The plug-in might not need to use an **src** attribute if it reads all of its data at run time or doesn't need any external data.

Because plug-ins are rectangular, embedded objects similar to images, the **embed** element has many of the same attributes as the **img** element:

- **align** Use to align the object relative to the page and allow text to flow around the object. To achieve the desired text layout, you may have to use the **br** element with the **clear** attribute.
- **hspace** and **vspace** Use to set the buffer region, in pixels, between the embedded object and the surrounding text.
- **border** Use to set a border for the plug-in, in pixels. As with images, setting this attribute to zero may be useful when using the embedded object as a link.
- **height** and **width** Use to set the vertical and horizontal size of the embedded object, typically in pixels, although you can express them as percentage values. Values for **height** and **width** should always be set, unless the **hidden** attribute is used. Setting the **hidden** attribute to **true** in an **<embed>** tag causes the plug-in to be hidden and overrides any **height** and **width** settings, as well as any effect the object might have on layout.

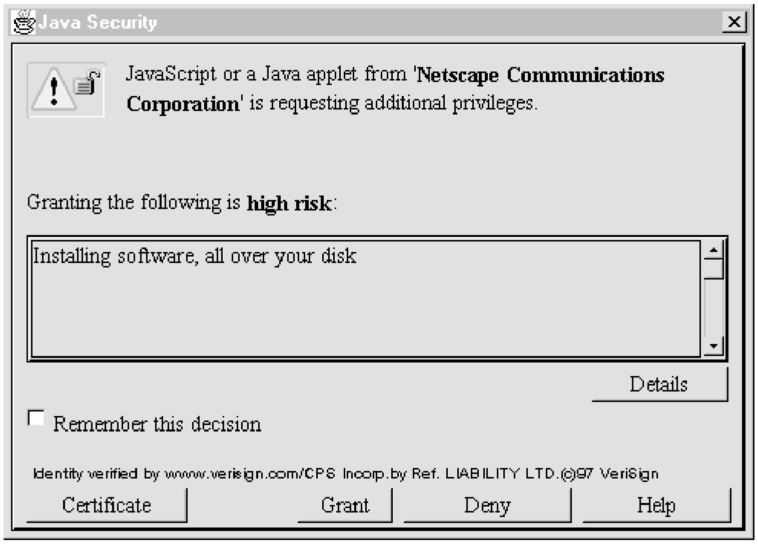
Custom Plug-in Attributes

In addition to the standard attributes for **<embed>**, plug-ins might have custom attributes to communicate specialized information to the plug-in code. For example, a movie player plug-in may have a **loop** attribute to indicate how many times to loop the movie. Remember that under HTML or XHTML, the browser ignores all nonstandard attributes when parsing the markup. All other attributes are passed to the plug-in, allowing the plug-in to examine the list for any custom attributes that could modify its behavior. Enumerating all the possible custom attributes here is simply not possible as each particular plug-in used can have a variety of custom attributes. You should be certain to look at the documentation for whatever plug-in you are going to use.

Attributes for Installation of Plug-ins

If embedded data in a Web page has no associated plug-in, the user will need to install a plug-in to address it. However, making users figure out which plug-in to install isn't a very good idea; instead, set the **pluginspage** attribute equal to a URL that indicates the instructions for installing the needed plug-in. This way, if the browser encounters an **embed** element that it can't handle, it visits the specified page and provides information on how to download and install the plug-in. Starting with Netscape 4, however, this attribute automatically points to a special Netscape plug-in finder page.

Netscape 4 and beyond simplify the plug-in installation process by introducing the JAR Installation Manager (JIM), which is used to install Java Archive files (JARs). JAR files are a collection of files, including plug-ins, which can be automatically downloaded and installed. Set the **pluginurl** attribute for an **<embed>** tag to the URL of a JAR file containing the plug-in that is needed. If the user doesn't have the appropriate plug-in already installed, the browser invokes JIM with the specified JAR file and begins the download and installation process. The user has control over this process. The downloaded objects can be signed—a type of authentication—to help users avoid downloading malicious code. Figure 15-2 shows a sample JIM window under Netscape 4.



JavaScript or a Java applet from '**Netscape Communications Corporation**' is requesting additional privileges.

Granting the following is **high risk**:

Installing software, all over your disk

☐ Remember this decision

Identity verified by www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign

FIGURE 15-2 JIM window under Netscape 4

In Netscape 4 or greater, the **pluginurl** attribute takes precedence over **pluginspage**. However, if neither attribute is used, the Netscape browser should default to a plug-in finder page.

<noembed>

Some browsers don't understand Netscape's plug-in architecture, or even the **<embed>** tag. Rather than lock out these browsers from a Web page, the **<noembed>** tag enables you to provide some alternative text or marked-up content. In the following short example, an AVI video is embedded in the page. The **<noembed>** tag contains an image, which in turn has an alternative text reading set with the **alt** attribute. Note how the example degrades from a very sophisticated setting all the way down to a text-only environment:

```
<embed src="welcome.avi" height="100" width="100" />
<noembed>
   <img src="welcome.gif" alt="Welcome to Demo Company, Inc." />
</noembed>
```

One potential problem with the **<noembed>** approach occurs when a browser supports plug-ins but lacks the specific plug-in to deal with the included binary object. In this case, the user is presented with a broken puzzle-piece icon or a similar icon, and then is directed to a page to download the missing plug-in. As discussed previously, you should always set the **pluginurl** or **pluginspage** attribute to start the user on the process of getting the plug-in needed to view the content.

<object> Syntax for Plug-ins

The **<object>** tag can also be used to include a variety of object types in a Web page, including Netscape plug-ins. Like an **<embed>** tag, an **<object>** tag's attributes determine the type of object to include, as well as the type and location of the plug-in. The primary attribute for **<object>** when referencing plug-ins is **data,** which represents the URL of the object's data and is equivalent to the **src** attribute of **<embed>**. Like the **<embed>** tag, the **type** attribute represents the MIME type of the object's data. This sometimes can be inferred from the value of the **data** attribute. The **codebase** attribute, which is similar to the **pluginspage** attribute, represents the URL of the plug-in. The **classid** attribute is used to specify the URL to use to install the plug-in, by using the JIM. If no **classid** attribute is specified and the object can't be handled, the object is ignored, and any nested markup is displayed as an alternative rendering. The **id** attribute is used to set the name of the object for scripting. If the browser can't handle the type, or can't determine the type, it can't embed the object. Subsequent HTML is parsed as normal. The following is an example of using Netscape's old LiveAudio plug-in under Netscape 4 with the **<object>** syntax:

```
<object data="click.wav" type="audio/wav" height="60" width="144"
        autostart="false">
   <b>Sorry, no LiveAudio installed...</b>
</object>
```

Page authors should be cautious when referencing plug-ins with an **<object>** tag because compatibility issues with Microsoft Internet Explorer can arise. For the complete syntax of the **object** element, refer to the element reference in Appendix A.

Scripting and Plug-ins

Plug-ins can be accessed from a scripting language. Each plug-in used in a document can be referenced in Netscape's browsers as an element of the JavaScript **embeds[ ]** collection, which is part of the document object, as discussed in the previous chapter. Under Netscape, you can determine which plug-ins are available in the browser by using the **plugins[ ]** collection, which is part of the navigator object in JavaScript. The following markup displays the plug-ins that are installed in a Netscape browser:

```
<!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" lang="en">
<head>
<title>Print Plug-ins</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<h2 align="center">Plug-ins Installed</h2>
<hr />
<script type="text/javascript">
<!--
if (navigator.appName == "Microsoft Internet Explorer")
 document.write("Plug-ins[] collection not supported under IE");
else
 {
   var num_plugins = navigator.plugins.length;
   for (var count=0; count < num_plugins; count++)
     document.write(navigator.plugins[count].name + "<br />");
 }
//-->
</script>
</body>
</html>
```

Note that this example will not display the plug-ins under Internet Explorer because IE doesn't support the **plugins[ ]** collection. Under Netscape, however, you can use some simple if-then logic in JavaScript to determine which markup to output depending on if a particular plug-in is loaded in the browser or not.

Once plug-ins are used in a page, they always should be named using the **name** and **id** attributes so they can be accessed easily from JavaScript. For example, the markup

```
<embed src="welcome.avi" name="welcomemovie" id="welcomemovie"
height="100" width="100"></embed>
```

gives this instance of the LiveVideo plug-in the name WelcomeMovie. After the plug-in is named, it can be accessed from JavaScript as **document.welcomemovie**. If it is the second plug-in in the page, it also could be referenced as **document.embeds[1]**. Why not "index 2"? Arrays in JavaScript, which is how collections are implemented, start numbering at zero, so **document.embeds[0]** references the first plug-in, **document.embeds[1]** references the second plug-in, and so on.

After you name an occurrence of a plug-in in a page, you might be able to manipulate the plug-in's actions even after you load the page. Netscape browsers, starting with the 3.*x* generation, include a technology called *LiveConnect* that enables JavaScript to communicate with Java applets and plug-ins. However, only plug-ins written to support LiveConnect can be manipulated using JavaScript. Fortunately, many plug-ins such as Macromedia Flash support LiveConnect. This simple example shows how LiveConnect works by using form buttons to start and stop the playing of a Flash movie in a page using 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" lang="en">
<head>
<title>Flash JavaScript Control Example</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--

var loaded=false;
function playFlash(id)
{
 var flashFile = eval("window.document."+id);
  if (!loaded)
   {
    while (!loaded)
     {
        if (flashFile.PercentLoaded() == 100)
         {
           flashFile.Play();
           loaded = true;
          }
```

```
      }
    }
   else
    flashFile.Play();
}

function stopFlash(id)
{
  var flashFile = eval("window.document."+id);
  flashFile.StopPlay();
}
//-->
</script>
</head>
<body>
<h2 align="center">Plug-in and JavaScript Interaction</h2>
<embed src="example.swf" quality="high"
       pluginspage="http://www.macromedia.com/go/getflashplayer"
       type="application/x-shockwave-flash"
       width="400" height="250"
       id="example" name="example"
       swliveconnect="true">
  <noembed>
      You need Flash and Netscape for this demo
  </noembed>
</embed>
<hr />
<form action="#">
<input type="button" name="Button1" value="Start Flash" onclick="playFlash('example');" />
<input type="button" name="Button2" value="Stop Flash" onclick="stopFlash('example');" />
</form>
</body>
</html>
```

NOTE *The **swliveconnect="true"** attribute is very important and will ensure that this example works. Without this attribute, the demo should not work. Even so, the preceding example is very specific to Netscape and it may not work even in some Mozilla variants. Also, it might work under Internet Explorer, but there is no guarantee of this. Furthermore, the implementation of LiveConnect is buggy and might not work under all versions of Navigator. Still, the example illustrates the basics of a script talking to a plug-in.*

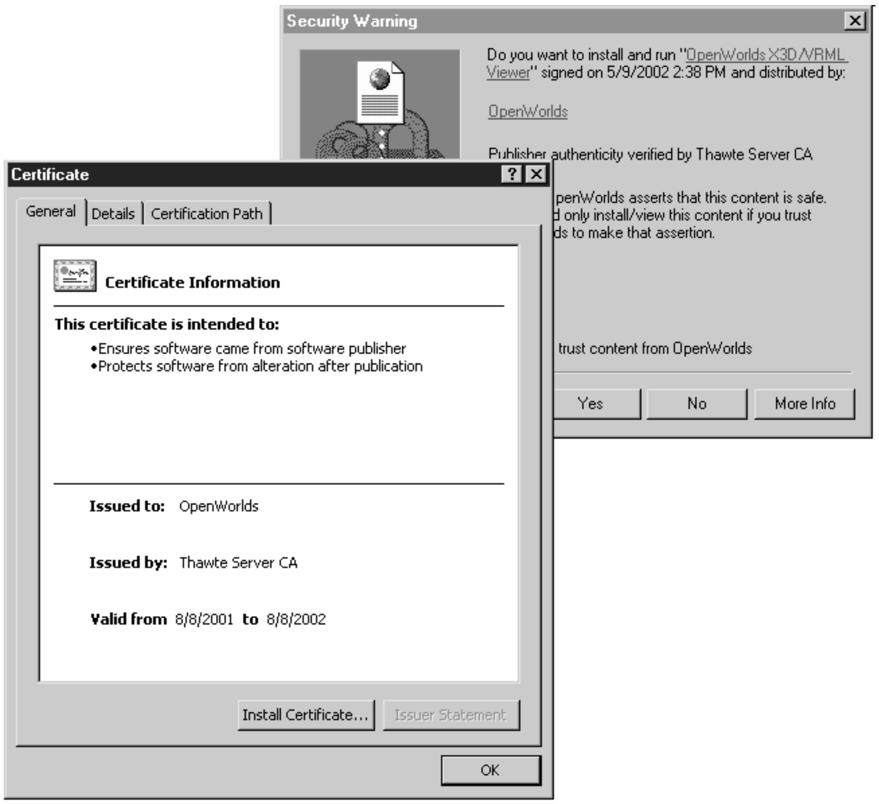
Tying together plug-ins by using a scripting language in conjunction with LiveConnect hints at the power of such component models as Netscape's plug-ins. However, Netscape plug-ins often are passed over in favor of ActiveX or Java applets for general programming tasks, and plug-ins often are regulated to handling new media forms as this example just demonstrated.

ActiveX Controls

ActiveX (http://www.microsoft.com/com/tech/activex.asp), which is the Internet portion of the Component Object Model (COM), is Microsoft's component technology for creating small components, or *controls*, within a Web page. It is intended to distribute these controls via the Internet to add new functionality to browsers such as Internet Explorer. ActiveX controls are more similar to generalized programmed components than plug-ins because they can reside beyond the browser within container programs such as Microsoft Office. ActiveX controls are similar to Netscape plug-ins insofar as they are persistent and machine specific. Although this makes resource use a problem, installation is not an issue: the components download and install automatically.

Security is a big concern for ActiveX controls. Because these small pieces of code could potentially have full access to a user's system, they could cause serious damage. This capability, combined with automatic installation, creates a serious problem with ActiveX. End users might be quick to click a button to install new functionality, only to accidentally get their hard drives erased. To address this problem, Microsoft provides authentication information to indicate who wrote a control, in the form of code signing and a certificate, as shown in Figure 15-3.

Certificates provide only some indication that the control creator is reputable; they do nothing to prevent a control from actually doing something malicious. Safe Web browsing should be practiced by accepting controls only from reputable sources.


Security Warning

Do you want to install and run "OpenWorlds X3D/VRML Viewer" signed on 5/9/2002 2:38 PM and distributed by:

OpenWorlds

Publisher authenticity verified by Thawte Server CA

penWorlds asserts that this content is safe.
d only install/view this content if you trust
ds to make that assertion.

trust content from OpenWorlds

Certificate

Certificate Information

This certificate is intended to:
- Ensures software came from software publisher
- Protects software from alteration after publication

Issued to: OpenWorlds

Issued by: Thawte Server CA

Valid from 8/8/2001 to 8/8/2002

FIGURE 15-3 ActiveX signed-code certificate

Adding Controls to Web Pages

Adding an ActiveX control to a Web page requires the use of the **<object>** tag. The basic form of <**object>** for an ActiveX control is as follows:

```
<object classid="CLSID:class-identifier"
        height="pixels or percentage"
        width="pixels or percentage"
        id="unique identifier">

Parameters and alternative text rendering

</object>
```

When you insert ActiveX controls, **classid** is the most important attribute for an **<object>** tag. The value of **classid** identifies the object to include. Each ActiveX control has a class identifier of the form "**CLSID**: *class-identifier*," where the value for *class-identifier* is a complex string, such as the following, which uniquely identifies the control:

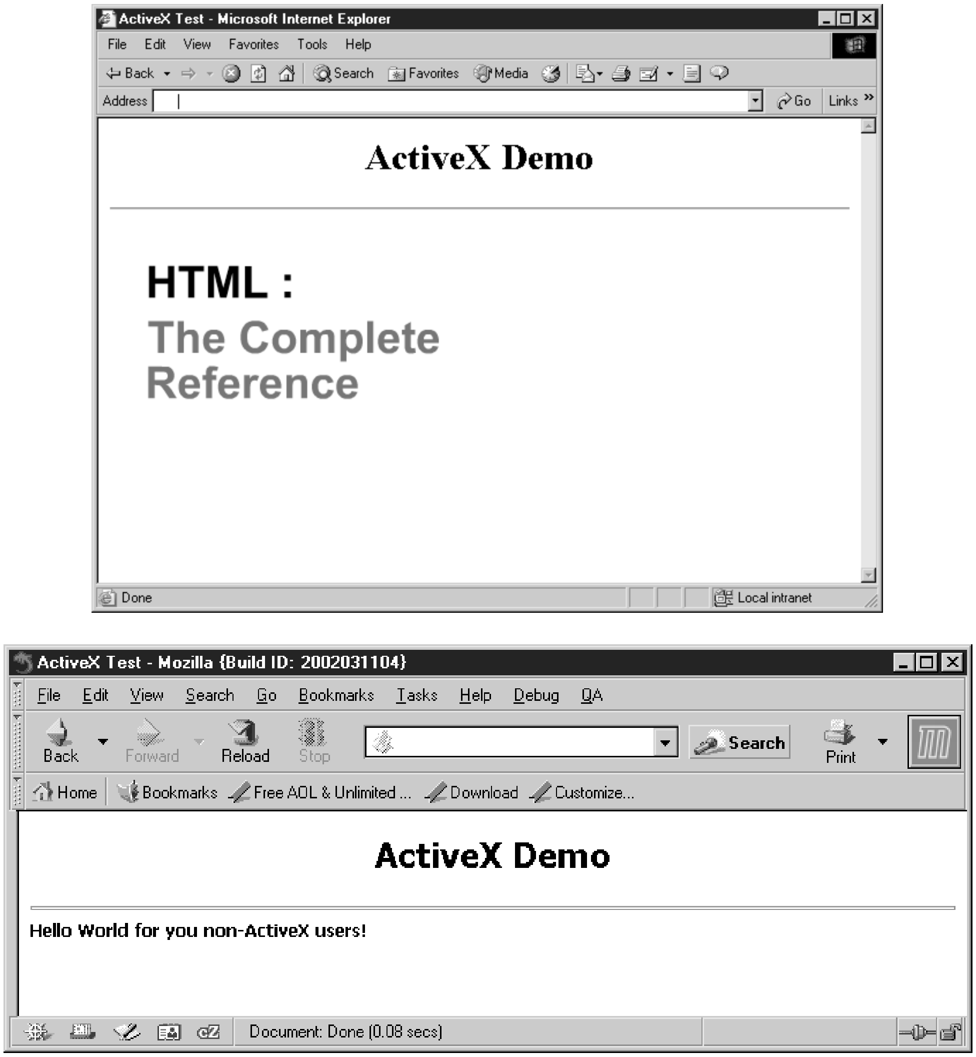
```
D27CDB6E-AE6D-11cf-96B8-444553540000
```

This is the identifier for the ActiveX implementation of the Flash Player. The other important attributes for the basic form of **<object>** when used with ActiveX controls include **height** and **width**, which are set to the pixel dimensions of the included control, and **id**, which associates a unique identifier with the control for scripting purposes. Between the **<object>** and **</object>** tags are various **<param>** tags that specify information to pass to the control, and alternative content and markup that displays in non-ActiveX-aware browsers. The following is a complete example that uses an **<object>** tag to insert an ActiveX control into a Web page. The markup specifies a Flash file. Figure 15-4 shows the rendering of the control under Internet Explorer and Mozilla, which does not support ActiveX.

```
<!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" lang="en">
<head>
<title>ActiveX Test</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>

<h1 align="center">ActiveX Demo</h1>
<hr />

<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        codebase="http://download.macromedia.com/pub/
                  shockwave/cabs/flash/swflash.cab#version=6,0,0,0"
        id="flash1" name="flash1"
        width="320" height="240">

   <param name="movie" value="http://www.htmlref.com/flash/example.swf" />
   <param name="quality" value="high" />
```

```
    <b>Hello World for you non-ActiveX users!</b>

</object>
</body>
</html>
```

After you look at the previous markup, you may have questions about how to determine the **classid** value for the control and the associated **<param>** values that can be set. However, providing a chart for all the controls and their associated identifiers isn't necessary. Many Web development tools, including Macromedia Dreamweaver and Microsoft Visual Studio


ActiveX Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go Links
Done
Local intranet
ActiveX Test - Mozilla {Build ID: 2002031104}
File Edit View Search Go Bookmarks Tasks Help Debug QA
Back Forward Reload Stop Search Print
Home Bookmarks Free AOL & Unlimited ... Download Customize...
Document: Done (0.08 secs)

ActiveX Demo

HTML :
The Complete
Reference

ActiveX Demo

Hello World for you non-ActiveX users!

FIGURE 15-4 Rendering of ActiveX control under Internet Explorer and Mozilla

support the automated insertion of controls into a page, as well as configuration of the various control properties. If you are required to insert one by hand, hopefully the vendor of the control has provided documentation that you can consult to find the appropriate **classid**.

Installing ActiveX Controls

As mentioned previously, the most important attribute in the **<object>** syntax probably is **classid**, which is used to identify the particular object to include. For example, the syntax "CLSID:class-identifier" is for registered ActiveX controls. Generally, however, when the **object** element supports other included items well, **classid** might be set to other forms, such as "java: Blink.class," as discussed later in the chapter. Explorer also allows the use of the **code** attribute for the **<object>** tag; **code** is used to set the URL of the Java class file to include.

ActiveX and plug-ins are similar in the sense that both are persistent, platform-specific components. ActiveX controls, however, are easy to download and install. This installation, or running of ActiveX controls, can be described as a series of steps:

1. The browser loads an HTML page that references an ActiveX control with the **<object>** tag and its associated **classid** attribute.
2. The browser checks the system registry to see whether the control specified by the **classid** value is installed; this control takes the form "CLSID: some-id-number."
3. If the control is installed, the browser compares the **codebase** version attribute stored in the registry against the **codebase** version attribute specified in the tag. If a newer version is specified in the page, a newer control is needed.
4. If the control is not installed or a newer control is needed, the value of the **codebase** attribute is used to determine the location of the control to download. The **codetype** attribute also can be used to set the MIME type of the object to download. Most inclusions of ActiveX controls avoid this because it tends to default to the MIME type application/octet-stream.

For security reasons, the browser checks to see whether the code is signed before the download and installation begins. If the code is not signed, the user is warned. If the code is signed, the user may be presented with an Authenticode certificate bearing the identity of the author of the control. Based on these criteria, the user can allow or deny the installation of the control on his or her system. If the user accepts the control, it is automatically downloaded, installed, and invoked in the page for its specific function. Finally, the control is stored persistently on the client machine for further invocation. This process can be avoided when the **declare** attribute is present. The **declare** attribute is used to indicate whether the **<object>** is being defined only and not actually instantiated until later **<object>** occurrences, which will start the installation process.

NOTE *The W3C HTML 4 specification also indicates use of the* ***standby*** *attribute, which can be used to specify a message to display as the object is being downloaded. This currently is not supported by most browsers.*

Passing Data to ActiveX Controls

Unlike plug-ins, ActiveX controls do not use special attributes to pass data. Instead, they use a completely different element, called **param**, which is enclosed within an **<object>** tag. You can pass parameters to a control by using **<param>** tags, as shown here:

```
<object classid="CLSID:control-classid-here"
        id="label1" height="65" width="325">
   <param name="Caption" value="Hello World" />
   <param name="FontName" value="Arial" />
   <param name="FontSize" value="36" />

</object>
```

In this case, the **Caption** parameter is set to Hello World, the **FontName** parameter is set to Arial, and the **FontSize** parameter is set to 36 points. This is just a generic example to illustrate the idea; the previous example with Flash illustrated setting the source of the *movie* and its *quality* via **<param>** tags.

ActiveX Controls and Scripting

Similar to plug-ins, you can control ActiveX controls by using a scripting language such as JavaScript or VBScript. Before a control can be scripted, however, it must be named by using the **id** attribute. After it is named, scripting code for a particular event can be set for the control so that it can respond to events such as user clicks or mouse movements. The following simple example shows the previous Flash demo using ActiveX style **<object>** syntax with only minor modifications to the script to make it more like traditional Explorer JavaScript syntax:

```
<!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" lang="en">
<head>
<title>Flash JavaScript Control Example</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
var loaded=false;

function playFlash(id)
{
  var flashFile = eval("window.document.all."+id);
  if (!loaded)
     {
        while (!loaded)
       {
         if (flashFile.PercentLoaded() == 100)
         {
           flashFile.Play();
           loaded = true;
         }
       }
     }
   else
    flashFile.Play();
}

function stopFlash(id)
{
  var flashFile = eval("window.document.all."+id);
  flashFile.StopPlay();
}
//-->
</script>
</head>
<body>
<h2 align="center">Plug-in and JavaScript Interaction</h2>

<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        codebase="http://download.macromedia.com/pub/shockwave
                  /cabs/flash/swflash.cab#version=6,0,0,0"
        id="example" name="example"
        width="320" height="240">

   <param name="movie" value="http://www.htmlref.com/flash/example.swf" />
   <param name="quality" value="high" />
   <param name="swliveconnect" value="true" />

   <b>Hello World for you non-ActiveX users!</b>

</object>

<form action="#">
<input type="button" name="Button1" value="Start Flash" onclick="playFlash('example');" />
<input type="button" name="Button2" value="Stop Flash" onclick="stopFlash('example');" />
</form>
</body>
</html>
```

NOTE *This example won't work in anything other than Internet Explorer 3 or better running on a Windows-based system.*

Using ActiveX

Developers can access an abundance of available controls for various purposes. Today, most of the controls used are for playing media such as Flash movies or wrap-around images. While Microsoft used to include a variety of controls with its applications, including Internet Explorer, most of these are being phased out in favor of technologies that are part of the .NET framework. Yet despite this, ActiveX still survives and some developers even write their own browser controls and helper objects in Visual Basic, C++, and other high-level languages. ActiveX and related technologies are part of a larger Microsoft development framework that has undergone numerous name changes over the years and at the time of this writing is known as the .NET platform. Even that might change by the time you read this, so for the very latest information on development for the Microsoft platform, see http://msdn.microsoft.com.

Java Applets

Whereas both Microsoft's ActiveX and Netscape's plug-ins are platform and browser specific, Sun Microsystems' Java technology (http://java.sun.com) aimed to provide a platform-neutral development language, allowing programs to be written once and deployed on any machine, browser, or operating system that supports the Java virtual machine (JVM). Java uses small Java programs, called *applets*, that were first introduced by Sun's HotJava browser. Today, applets are supported by most Web browsers, including Netscape Navigator and Microsoft Internet Explorer. Of course, the nirvana of perfect cross-platform development never really materialized. Many versions of the Windows operating system do not ship with Java virtual machines and the technology never really took off in public Web sites. However, they are still used and Java applets continue to play an important role even in client-side development, particularly within controlled environments such as intranets.

Applets are written in the Java language and compiled to a machine-independent byte-code, which is downloaded automatically to the Java-capable browser and run within the browser environment. But even with a fast processor, the end system might appear to run the byte-code slowly compared to a natively compiled application because the byte code must be interpreted by the Java Virtual Machine. Even with recent Just-In-Time (JIT) compilers in newer browsers, Java often doesn't deliver performance equal to natively compiled applications upon startup. Once running, Java applets and applications perform well. However, even if compilation weren't an issue, current Java applets generally aren't persistent; they may have to be downloaded again in the future. Java-enabled browsers act like thin-client applications because they add code only when they need it. In this sense, the browser doesn't become bloated with added features, but expands and contracts upon use.

Security in Java has been considered from the outset. Because programs are downloaded and run automatically, a malicious program could be downloaded and run without the user being able to stop it. Under the first implementation of the technology, Java applets had little access to resources outside the browser's environment. Within Web pages, applets can't write to local disks or perform other harmful functions. This framework has been referred to as the *Java sandbox*. Developers who want to provide Java functions outside of the sandbox must write Java applications, which run as separate applications from browsers. Other Internet programming technologies (for example, ActiveX) provide little or no safety from damaging programs.

Oddly, Java developers often want to add just these types of insecure features, as well as such powerful features as persistence and inter-object communication. In fact, under new browsers, extended access can be requested for signed Java applets. (A *signed applet* enables users to determine who authored its code, and to accept or reject the applet accordingly.) Java applets can securely request limited disk access, limited disk access and network usage, limited disk read access and unlimited disk write access, and unrestricted access. Users downloading an applet that is requesting any enhanced privileges are presented with a dialog box that outlines the requested access and presents the applet's credentials in the form of its digital signature. The user then can approve or reject the applet's request. If the user doesn't approve the request, the applet can continue to run, but it can't perform the denied actions.

Java code looks very much like C++. The following code fragment shows a simple example of a Java applet:

```
import java.applet.Applet;
import java.awt.Graphics;

public class helloworld extends Applet
{
    public void paint(Graphics g)
     {
        g.drawString("Hello World", 50, 25);
     }
}
```

You can save this previous example into a file named helloworld.java and then send the code through a Java compiler (such as JavaSoft's javac) to produce a class file called helloworld.class, which can be used on a Web page to display the phrase "Hello World." You can use an **<applet>** tag to add a Java applet to a Web page. As with the **<embed>** tag, you must indicate the object to add. In this case, use the **code** attribute to indicate the URL of the Java class file to load. Because this is an included object, the **height** and **width** attributes should also be set. The following example includes the HelloWorld applet in a Web page. Figure 15-5 shows the rendering of the Java example under Netscape 4 with Java turned on and Java turned off.

```
<!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" lang="en">
<head>
<title>Java Hello World</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<h1 align="center">Java Applet Demo</h1>
<hr />
<applet code="helloworld.class"
        height="50" width="175">

 <h1>Hello World for you non-Java-aware browsers</h1>

</applet>
</body>
</html>
```

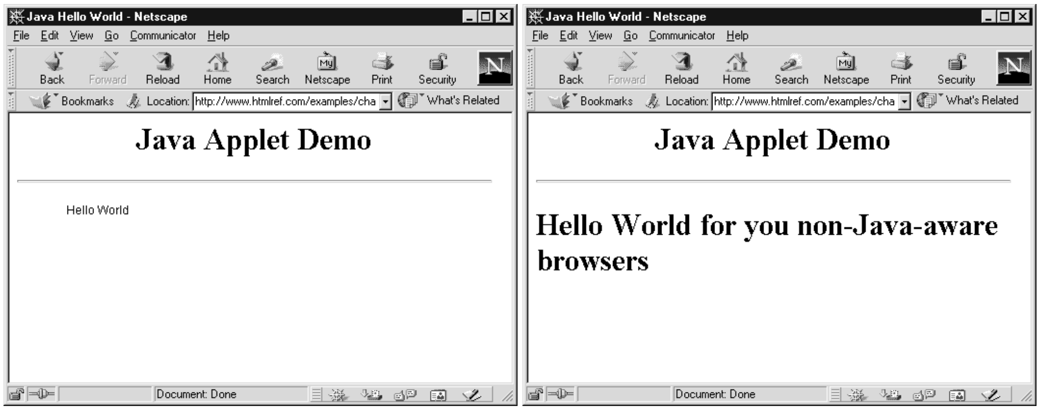
In the preceding code example, between **<applet>** and **</applet>** is an alternative rendering for browsers that don't support Java or the **applet** element, or that have Java support disabled.

<applet> Syntax

Because Java applets are included objects, just like Netscape plug-ins, the syntax for the **applet** element is similar to the **embed** element, particularly for things such as alignment and sizing. The complete syntax for **<applet>** is shown in the element reference in Appendix A.

The most important attribute for the **applet** element probably is **code**, which is set to the URL of the Java class to load into the page. The **codebase** attribute can be set to the URL of the directory that contains the Java classes; otherwise, the current document's URL is used for any relative URLs.

Because Java applets are rectangular, embedded objects similar to images or plug-ins, an **<applet>** tag has many of the same attributes as images and plug-ins, including **align**, **height**, **width**, **hspace**, and **vspace**.


Java Hello World - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
Bookmarks Location: http://www.htmlref.com/examples/cha What's Related
Document: Done

Java Applet Demo

Hello World

Java Applet Demo

Hello World for you non-Java-aware browsers

FIGURE 15-5 Java example under Netscape 4 with Java turned on and off

The **archive** attribute can be used to include many classes into a single archive file, which then can be downloaded to the local disk. The file specified by the **archive** attribute can be a compressed ZIP file (.zip) or a Java Archive (.jar), which can be made with a JAR packaging utility. For example,

```
<applet archive="bunchofclasses.zip"
        code="sampleApp.class"
        width="560"
        height="270">
</applet>
```

downloads all the classes in bunchofclasses.zip. After the file is downloaded, the **code** attribute is examined and the archive is checked to see whether sampleApp.class exists there. If not, it is fetched from the network. Due to the expense of fetching many class files by using HTTP, ideally, you should attempt to archive all potentially used classes and send them simultaneously. You also can derive some caching benefit by using the **archive** attribute because it keeps class files in the user's cache or a temporary directory. According to the HTML specification, the **archive** attribute can take a comma-separated list of archive files.

Passing Data to Java Applets

Unlike plug-ins, Java applets don't use special attributes to pass data. Instead, like ActiveX control's syntax, they use a different tag called **<param>**, which is enclosed within an **<applet>** tag, as the way to pass on information. You could extend the HelloWorld applet to allow the message output to be modified by using the **<param>** tag to pass in a message, as shown here:

```
<applet code="helloworld.class" width="50" height="175">

<param name="Message" value="Hello World in Java!" />

<h1>Hello World for you non-Java-aware browsers</h1>

</applet>
```

The following is the basic transitional XHTML syntax for **<param>**. Note that it is the same for Java applets and ActiveX controls:

```
<param name="Object property name"
       value="Value to pass in with object name"
       valuetype="DATA | REF | OBJECT"
       type="MIME Type"
       id="document-wide unique id" />
```

The **name** attribute for **<param>** is used to specify the name of the object property that is being set; in the preceding example, the name is "Message." If you are using a pre-made Java applet, the various property names should be specified in the documentation for the applet. The actual value to be assigned to the property is set by the **value** attribute. The **valuetype** attribute specifies the meaning of the **value** attribute. The data passed to an attribute typically takes the form of a string. Setting the **valuetype** attribute to **data** results in the default action. Setting **valuetype** to **ref** indicates that the data assigned to the **value** attribute is a URL that references an external file to load for the attribute. The last value for **valuetype** is **object**, which indicates that **value** is set to the name of an applet or object located somewhere else within the document. The data in the applet or object can be referenced to allow objects to "talk" to each other.

The **param** elements for a particular Java applet occur within the **<applet>** tag; a Java applet can have many **param** elements. The **applet** element also can enclose regular XHTML markup and other textual content that provides an alternative rendering for non-Java-capable browsers. When alternative content is found within the **<applet>**, the **<param>** tags should be placed before the other content. Note that you also can set the **alt** attribute for the **applet** element to provide a short description. Authors should use the text contained within the element as the alternative text, and not the **alt** attribute.

Java Applets and Scripting

Java applets can control scripts in a Web page. Supposedly, the inclusion of the **mayscript** attribute in an **<applet>** tag permits the applet to access JavaScript. When dealing with applets retrieved from other sources, you can use the **mayscript** attribute to prevent the applet from accessing JavaScript without the user's knowledge. If an applet attempts to access JavaScript when this attribute has not been specified, a run-time exception should occur. In practice, however, it appears that browsers do not necessarily care about the absence of **mayscript**.

Probably more interesting for page designers is the fact that scripts can control or even modify Java applets that are embedded in a page. For the applet to be accessed, it should be named using the **name** attribute as well as the **id** attribute. Providing a unique name for the applet allows scripts to access the applet and its public interfaces. The name also can be used by other applets to allow the applets to communicate with each other. JavaScript in Netscape 3 and above, as well as in Internet Explorer 4 and above, allows access to the applets in a page via the **applets[ ]** collection, which is a property of the document object. When an applet is named, it can be accessed through JavaScript as **document.*appletname***, such as **document.myApplet**, or through the array of applets in the document such as **document.applets[0]** or **document.applets["myApplet"]**. If the Java applet has public properties exposed, they can be modified from a script in a Web page. The following simple

Java code takes the "Hello World" example from earlier in the chapter and expands it with a **setMessage** method, which can be used to change the message displayed in the applet:

```
import java.applet.Applet;
import java.awt.Graphics;
public class newhelloworld extends Applet
{
    String theMessage;

    public void init()
     {
       theMessage = new String("Hello World");
     }
    public void paint(Graphics g)
     {
        g.drawString(theMessage, 50, 25);
     }
    public void setMessage(String message)
     {
       theMessage = message;
       repaint();
     }
}
```

If this Java code is compiled into a class file as explained earlier, it can be included in a Web page and accessed via JavaScript, as shown next. The following example markup shows how a form could be used to collect data from the user and update the applet in real time:

```
<!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" lang="en">
<head>
<title>Java and Scripting Demo</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
function setMessage()
{
  var message = document.TestForm.NewMessage.value;
  document.NewHello.setMessage(message);
}
//-->
</script>
</head>
<body>
<h1 align="center">Java and Scripting Demo</h1>
<hr />

<applet code="newhelloworld.class"
        name="NewHello"
        height="50" width="175">
```

```
 <h1>You need Java for this example.</h1>

</applet>

<br /><br />

<form action="#" name="TestForm" id="TestForm">
  <input type="text" size="15" maxlength="15" name="NewMessage" />
  <input type="button" value="Set Message" onclick="setMessage()" />
</form>
</body>
</html>
```

<object> Syntax for Java Applets

The strict variants of HTML and XHTML indicate that the **applet** element has been deprecated and that **object** should be used instead. The following is the most basic XHTML syntax for inserting an object, such as a Java applet:

```
<object classid="URL of object to include"
        height="pixels or percentage"
        width="pixels or percentage">

  Parameters and alternative text

</object>
```

For the complete **<object>** syntax, see the element reference in Appendix A.

Notice that the **classid** attribute is used to specify the URL of the object to include. In the case of Java applets, you should use **java:**. For ActiveX controls, use **clsid:**. The following example rewrites the first simple Java applet example to use **<object>** and to work under strict XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>
<title>Java Hello World</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<h1>Java Applet Demo</h1>
<hr />
<div>

<object classid="java:helloworld.class"
        height="50" width="175">

  <h1>Hello World for you non-Java-aware browsers</h1>
</object>

</div>
```

```
</body>
</html>
```

Because of the fragmentation of the Java community, Sun has made some attempts to bring together the syntax of Java applets using a Java plug-in. The specific syntax for this plug-in under Netscape and Internet Explorer includes both **<object>** and **<embed>** forms. Readers interested in this syntax for applet inclusion should go straight to Sun's Java support site for the latest syntax (http://java.sun.com) because the syntax continues to change.

Cross-Platform <object> Syntax Today and Tomorrow

Although the whole point of Java applets is to deal with cross-platform compatibility issues, Microsoft ActiveX controls and Netscape plug-ins are extremely platform- and browser-dependent. While this might suggest that using Java applets would be the way to go, in many cases, it is more likely that ActiveX controls and Netscape plug-ins can be referenced side-by-side in a Web page before Java applets are even considered. For example, consider the **<object>** syntax, **<embed>** syntax, and **<noembed>** syntax combined in the following way:

```
<object classid="XXX" id="object1" name="object1"
        height="100" width="100">
    <param name="sample param" value="sample" />
    <!--   other param elements here -->
    <embed src="XXX" id="plug1" name="plug1" height="100" width="100">
          <noembed>
            Sorry, your browser supports neither ActiveX nor plug-ins.
          </noembed>
    </embed>
</object>
```

In this case, the ActiveX control is tried first, then the plug-in, and finally the content within the **<noembed>** would be used if none of the other worked. Other methods for addressing cross platform issues might include using JavaScript to detect what browser is being run and then outputting the appropriate markup to reference an ActiveX control, Java applet, or maybe even some other alternative form. The point is that it is possible with some careful thought to cover all the possible situations and until the syntax and technology for including objects is straightened out, this is the only reasonable approach to handling cross-browser issues, short of locking out users from a page or falling back to less interactive or less motivating technology.

Although the future of cross-browser object support sounds enticing, it has yet to materialize. For example, according to the strict versions of XHTML including the upcoming XHTML 2.0, **<object>** will be the main way to add any form of object to a Web page, whether it's an image, image map, sound, video, ActiveX control, Java applet, or anything else. This approach seems appropriate, but before rushing out to use **<object>**, understand the ramifications—little backwards compatibility.

Even today, while standardized, the syntax is not consistently supported. For example, according to even the HTML 4 specification, the **object** element can be used to include markup from another file by using the **data** attribute. Imagine specifying a header file called header.html with the contents shown here:

```
<h1 style="text-align: center; background-color: #FFA500;">
I am an included heading!</h1>
```

This file then could be included in a Web page by using an **<object>** tag, like so:

```
<object data="header.html">
Header not included
</object>
```

This example should pull in the contents of the file header.htm in browsers that support this feature, and display "Header not included" in all others. The example here demonstrates this in use:

```
<!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" lang="en">
<head>
<title>Object Inclusion Test</title>
</head>
<body>

<div style="background-color: yellow; border-style: solid;
text-align: center; margin: 5px;"> Before Object </div>

<object data="header.html" style="width: 100%;
border-style: dashed; border-width: 1px;">
Header not included
</object>

<div style="background-color: yellow; border-style: solid;
text-align: center; margin: 5px;">  After Object </div>

</body>
</html>
```

A rendering of Internet Explorer 6 and Mozilla shown in Figure 15-6 demonstrates the drastic difference in browsers that do and do not support the full **<object>** syntax.

Eventually, **<object>** will be used in a generalized sense and maybe object technologies will be supported through very simple markup. For now, you should carefully use the **applet**, **embed,** and **object** elements to include components and media objects other than images in pages along with any required scripting to avoid locking users out of viewing your Web page.

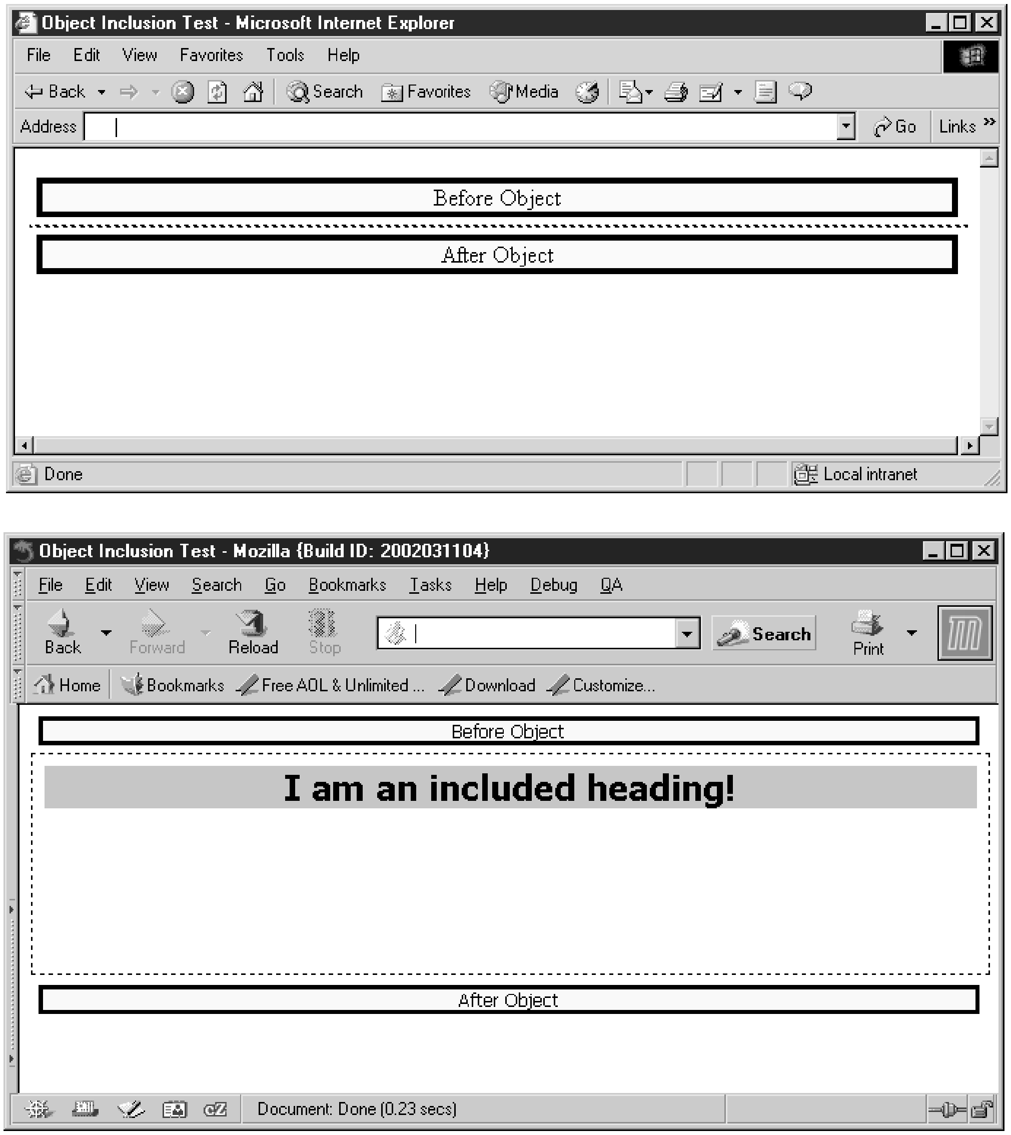

Object Inclusion Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address Go Links
Before Object
After Object
Done
Local intranet
Object Inclusion Test - Mozilla {Build ID: 2002031104}
File Edit View Search Go Bookmarks Tasks Help Debug QA
Back Forward Reload Stop Search Print
Home Bookmarks Free AOL & Unlimited ... Download Customize...
Before Object
I am an included heading!
After Object
Document: Done (0.23 secs)

FIGURE 15-6 <object> for file inclusion in Internet Explorer and Mozilla

Summary

With the inclusion of programmed objects such as ActiveX controls, Java applets, and Netscape plug-ins, Web pages can become complex, living documents. Choosing the appropriate component technology is not very straightforward. Netscape plug-ins are very popular for including media elements such as Flash animations, video, or sound files. Unfortunately, they are platform-specific, and largely limited to Netscape browsers, although most other browsers can handle their syntax and come up with something equivalent. The preferred solution in the Microsoft world is ActiveX controls. ActiveX controls are just as platform-specific as Netscape plug-ins, and have some potential security issues. Solving the cross-platform problem requires complex page scripting or the use of Java applets that provide cross-platform object support, typically at the expense of performance. Either way, the page rendering should degrade gracefully if the user can't support the particular object technology. Eventually, the syntax for all included media will be handled with an **<object>** tag, but for now, **<embed>** and **<applet>** should be used as well to provide backward compatibility for including plug-ins and Java applets in a Web page.