Chapter 44: Scripting Java Applets and Plug-ins

In This Chapter

Communicating with Java applets from scripts

Accessing scripts and objects from Java applets

Controlling scriptable plug-ins

Netscape was the first to implement the facility enabling JavaScript scripts, Java applets, and plug-ins to communicate with each other under one technology umbrella, called LiveConnect (first implemented in NN3). Microsoft met the challenge and implemented a large part of that technology for IE4/Windows, but of course without using the Netscape-trademarked name for the technology. The name is a convenient way to refer to the capability, so you find it used throughout this chapter applying to both NN and IE browsers that support such facilities. This chapter focuses on the scripting side of LiveConnect: approaching applets and plug-ins from scripts and accessing scripts from Java applets.

Except for the part about talking to scripts from inside a Java applet, I don't assume you have any knowledge of Java programming. The primary goal here is to help you understand how to control applets and plug-ins (including ActiveX controls in IE/Windows) from your scripts. If you're in a position to develop specifications for applets, you also learn what to ask of your Java programmers. But if you are also a Java applet programmer, you learn the necessary skills to get your applets in touch with HTML pages and scripts.

LiveConnect Overview

Before you delve too deeply into the subject, you should be aware that LiveConnect features are not available in all modern browsers, much to the chagrin of many. The following browsers do not support this technology:

* IE/Macintosh (at least through Version 5)
* NN4.6 (due to an oversight when the version was released)
* NN6.0 (work is afoot to include it in later versions)

Such a broad swath of browsers not supporting the feature (especially the IE for Macintosh, which has been factory-installed as the default browser on millions of Macs), makes it difficult to design a public Web application that relies on LiveConnect features. Design your pages accordingly.

The internal mechanisms that allow scripts to communicate with applets and plug-ins are quite different for NN and IE. NN3 and NN4 relied exclusively on the Java virtual machine (JVM) that shipped with most OS platform versions of the browsers. In NN4+, the JVM doesn't load until it is needed, sometimes causing a brief delay in initial execution. For the most part, though, the underlying Java engine is invisible to the scripter (you) and certainly to the visitors of your sites. At most, visitors see statusbar messages about applets loading and running.

IE/Windows, on the other hand, has its own internal architecture for communicating between processes. To Windows, most processes are treated as components that have properties and methods accessible to other components.

Whether you use the technology to communicate with a Java applet or an ActiveX control, the advantage to you as an author is that LiveConnect extends the document object model to include objects and data types that are not a part of the HTML world. HTML, for instance, does not have a form element that displays real-time stock ticker data; nor does HTML have the capability to treat a sound file like anything more than a URL to be handed off to a helper application. With LiveConnect, however, your scripts can treat the applet that displays the stock ticker as an object whose properties and methods can be modified after the applet loads; scripts can also tell the sound when to play or pause by controlling the plug-in that manages the incoming sound file.

Why Control Java Applets?

A question I often hear from experienced Java programmers is, "Why bother controlling an applet via a script when you can build all the interactivity you want into the applet itself?" This question is valid if you come from the Java world, but it takes a viewpoint from the HTML and scripting world to fully answer it.

Java applets exist in their own private rectangles, remaining largely oblivious to the HTML surroundings on the page. Applet designers who don't have extensive Web page experience tend to regard their applets as the center of the universe rather than as components of HTML pages.

As a scripter, on the other hand, you may want to use those applets as powerful components to spiff up the overall presentation. Using applets as prewritten objects enables you to make simple changes to the HTML pages — including the geographic layout of elements and images — at the last minute, without having to rewrite and recompile Java program code. If you want to update the look with an entirely new graphical navigation or control bar, you can do it directly via HTML and scripting.

When it comes to designing or selecting applets for inclusion into my scripted page, I prefer using applet interfaces that confine themselves to data display, putting any control of the data into the hands of the script, rather than using onscreen buttons in the applet rectangle. I believe this setup enables much greater last-minute flexibility in the page

design — not to mention consistency with HTML form element interfaces — than putting everything inside the applet rectangle.

A Little Java

If you plan to look at a Java applet's scripted capabilities, you can't escape having to know a little about Java applets and some terminology. The discussion goes more deeply into object orientation than you have seen with JavaScript, but I'll try to be gentle.

Java building blocks classes

One part of Java that closely resembles JavaScript is that Java programming deals with objects, much the way JavaScript deals with a page's objects. Java objects, however, are not the familiar HTML objects but rather more basic building blocks, such as tools that draw to the screen and data streams. But both languages also have some non-HTML kinds of objects in common: strings, arrays, numbers, and so on.

Every Java object is known as a class — a term from the object-orientation world. When you use a Java compiler to generate an applet, that applet is also a class, which happens to incorporate many Java classes, such as strings, image areas, font objects, and the like. The applet file you see on the disk is called a class file, and its file extension is `.class`. This file is the one you specify for the CODE attribute of an <APPLET> tag.

Java methods

Most JavaScript objects have methods attached to them that define what actions the objects are capable of performing. A string object, for instance, has the `toUpperCase()` method that converts the string to all uppercase letters. Java classes also have methods. Many methods are predefined in the base Java classes embedded inside the virtual machine. But inside a Java applet, the author can write methods that either override the base method or deal exclusively with a new class created in the program. These methods are, in a way, like the functions you write in JavaScript for a page.

Not all methods, however, are created the same. Java lets authors determine how visible a method is to outsiders. The types of methods that you, as a scripter, are interested in are the ones declared as public methods. You can access such methods from JavaScript via a syntax that falls very much in line with what you already know. For example, a common public method in applets stops an applet's main process. Such a Java method may look such as this:

```
public void stop() {
    if(thread != null) {
        thread.stop();
        thread = null;
    }
}
```

The `void` keyword simply means that this method does not return any values (compilers need to know this stuff). Assuming that you have one applet loaded in your page, the JavaScript call to this applet method is

```
document.applets[0].stop()
```

Listing 44-1 shows how all this works with the <APPLET> tag for a scriptable digital clock applet example. The script includes calls to two of the applet's methods: to stop and to start the clock.

Listing 44-1
Stopping and Starting an Applet

```
<HTML>
<HEAD>
<TITLE>A Script That Could Stop a Clock</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function pauseClock() {
    document.clock1.stop()
}
function restartClock() {
    document.clock1.start()
}
</SCRIPT>
<BODY>
<H1>Simple control over an applet</H1>
<HR>
<APPLET CODE="ScriptableClock.class" NAME="clock1" WIDTH=500 HEIGHT=45>
<PARAM NAME=bgColor VALUE="Green">
<PARAM NAME=fgColor VALUE="Blue">
</APPLET>
<P>
<FORM NAME="widgets1">
<INPUT TYPE="button" VALUE="Pause Clock" onClick="pauseClock()">
<INPUT TYPE="button" VALUE="Restart Clock" onClick="restartClock()">
</FORM>
</BODY>
</HTML>
```

The syntax for accessing the method (in the two functions) is just like JavaScript in that the references to the applet's methods include the applet object (`clock1` in the example), which is contained by the `document` object.

Java applet "properties"

The Java equivalents of JavaScript object properties are called *public instance variables*. These variables are akin to JavaScript global variables. If you have access to some Java source code, you can recognize a public instance variable by its public keyword:

```
public String fontName
```

Java authors must specify a variable's data type when declaring any variable. That's why the `String` data type appears in the preceding example.

Your scripts can access these variables with the same kind of syntax that you use to access JavaScript object properties. If the `fontName` variable in `ScriptableClock.class`

had been defined as a public variable (it is not), you could access or set its value directly, as shown in the following example.

```
var theFont = document.applets[0].fontName
document.applets[0].fontName = "Courier"
```

Accessing Java fields

In a bit of confusing lingo, public variables and methods are often referred to as *fields*. These elements are not the kind of text entry fields that you see on the screen; rather, they're like slots (another term used in Java) where you can slip in your requests and data. Remember these terms, because they may appear from time to time in error messages as you begin scripting applets.

Scripting Applets in Real Life

Because the purpose of scripting an applet is to gain access to the inner sanctum of a compiled program, the program should be designed to handle such rummaging around by scripters. If you can't acquire a copy of the source code or don't have any other documentation about the scriptable parts of the applet, you may have a difficult time knowing what to script and how to do it.

Although the applet's methods are reflected as properties in an applet object, writing a `for...in` loop to examine these methods tells you perhaps too much. Figure 44-1 shows a partial listing of such an examination of the ScriptableClock applet. This applet has only public methods (no variables), but the full listing shows the dozens of fields accessible in the applet. What you probably won't recognize, unless you have programmed in Java, is that within the listing are dozens of fields belonging to the Java classes that automatically become a part of the applet during compilation. From this listing, you have no way to distinguish the fields defined or overridden in the applet code from the base Java fields.

fieldName	fieldValue
	[JavaMethod ScriptableClock.]
getInfo	[JavaMethod ScriptableClock.getInfo]
setColor	[JavaMethod ScriptableClock.setColor]
setFont	[JavaMethod ScriptableClock.setFont]
setTimeZone	[JavaMethod ScriptableClock.setTimeZone]
paint	[JavaMethod ScriptableClock.paint]
run	[JavaMethod ScriptableClock.run]
stop	[JavaMethod ScriptableClock.stop]
start	[JavaMethod ScriptableClock.start]
init	[JavaMethod ScriptableClock.init]
destroy	[JavaMethod ScriptableClock.destroy]
play	[JavaMethod ScriptableClock.play]
getParameterInfo	[JavaMethod ScriptableClock.getParameterInfo]
getAppletInfo	[JavaMethod ScriptableClock.getAppletInfo]
getAudioClip	[JavaMethod ScriptableClock.getAudioClip]

Figure 44-1
Partial listing of fields from ScriptableClock

Getting to scriptable methods

If you write your own applets or are fortunate enough to have the source code for an existing applet, the safest way to modify the applet variable settings or the running processes is through applet methods. Although setting a public variable value may enable you to make a desired change, you don't know how that change may impact other parts of the applet. An applet designed for scriptability should have a number of methods defined that enable you to make scripted changes to variable values.

To view a sample of an applet designed for scriptability, open the Java source code file for Listing 44-2 from the CD-ROM. A portion of that program listing is shown in the following example.

Listing 44-2
Partial Listing for ScriptableClock.java

```
/*
    Begin public methods for getting
    and setting data via LiveConnect
*/
public void setTimeZone(String zone) {
    stop();
    timeZone = (zone.startsWith("GMT")) ? true : false;
    start();
}

public void setFont(String newFont, String newStyle, String newSize) {
    stop();
    if (newFont != null && newFont != "")
        fontName = newFont;
    if (newStyle != null && newStyle != "")
        setFontStyle(newStyle);
    if (newSize != null && newSize != "")
        setFontSize(newSize);
    displayFont = new Font(fontName, fontStyle, fontSize);
    start();
}

public void setColor(String newbgColor, String newfgColor) {
    stop();
    bgColor = parseColor(newbgColor);
    fgColor = parseColor(newfgColor);
    start();
}

public String getInfo() {
    String result = "Info about ScriptableClock.class\r\n";
    result += "Version/Date: 1.0d1/2 May 1996\r\n";
    result += "Author: Danny Goodman (dannyg@dannyg.com)\r\n";
    result += "Public Variables:\r\n";
    result += "   (None)\r\n\r\n";
    result += "Public Methods:\r\n";
    result += "   setTimeZone(\"GMT\" | \"Locale\")\r\n";
    result += "   setFont(\"fontName\",\"Plain\" |\"Bold\" | \"Italic\",
\"fontSize\")\r\n";
    result += "   setColor(\"bgColorName\", \"fgColorName\")\r\n";
```

```
    result += "       colors: Black, White, Red, Green, Blue, Yellow\r\n";
    return result;
}
/*
    End public methods for scripted access.
*/
```

The methods shown in Listing 44-2 are defined specifically for scripted access. In this case, they safely stop the applet thread before changing any values. The last method is one I recommend to applet authors. The method returns a small bit of documentation containing information about the kind of methods that the applet likes to have scripted and what you can have as the passed parameter values.

Now that you see the amount of scriptable information in this applet, look at Listing 44-3, which takes advantage of that scriptability by providing several HTML form elements as user controls for the clock. The results are shown in Figure 44-2.

Listing 44-3
A More Fully Scripted Clock

```
<HTML>
<HEAD>
<TITLE>Clock with Lots o' Widgets</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">

function setTimeZone(popup) {
    var choice = popup.options[popup.selectedIndex].value
    document.clock2.setTimeZone(choice)
}

function setColor(form) {
    var bg =
form.backgroundColor.options[form.backgroundColor.selectedIndex].value
    var fg =
form.foregroundColor.options[form.foregroundColor.selectedIndex].value
    document.clock2.setColor(bg, fg)
}

function setFont(form) {
    var fontName = form.theFont.options[form.theFont.selectedIndex].value
    var fontStyle = form.theStyle.options[form.theStyle.selectedIndex].value
    var fontSize = form.theSize.options[form.theSize.selectedIndex].value
    document.clock2.setFont(fontName, fontStyle, fontSize)
}
function getAppletInfo(form) {
    form.details.value = document.clock2.getInfo()
}

function showSource() {
    var newWindow = window.open("ScriptableClock.java","",
    "WIDTH=450,HEIGHT=300,RESIZABLE,SCROLLBARS")
}

</SCRIPT>
</HEAD>
<BODY>
<APPLET CODE="ScriptableClock.class" NAME="clock2" WIDTH=500 HEIGHT=45>
<PARAM NAME=bgColor VALUE="Black">
<PARAM NAME=fgColor VALUE="Red">
</APPLET>

<P>
```

```
<FORM NAME="widgets2">
Select Time Zone:
<SELECT NAME="zone" onChange="setTimeZone(this)">
    <OPTION SELECTED VALUE="Locale">Local Time
    <OPTION VALUE="GMT">Greenwich Mean Time
</SELECT><P>
Select Background Color:
<SELECT NAME="backgroundColor" onChange="setColor(this.form)">
    <OPTION VALUE="White">White
    <OPTION SELECTED VALUE="Black">Black
    <OPTION VALUE="Red">Red
    <OPTION VALUE="Green">Green
    <OPTION VALUE="Blue">Blue
    <OPTION VALUE="Yellow">Yellow
</SELECT>
Select Color Text Color:
<SELECT NAME="foregroundColor" onChange="setColor(this.form)">
    <OPTION VALUE="White">White
    <OPTION VALUE="Black">Black
    <OPTION SELECTED VALUE="Red">Red
    <OPTION VALUE="Green">Green
    <OPTION VALUE="Blue">Blue
    <OPTION VALUE="Yellow">Yellow
</SELECT><P>
Select Font:
<SELECT NAME="theFont" onChange="setFont(this.form)">
    <OPTION SELECTED VALUE="TimesRoman">Times Roman
    <OPTION VALUE="Helvetica">Helvetica
    <OPTION VALUE="Courier">Courier
    <OPTION VALUE="Arial">Arial
</SELECT><BR>
Select Font Style:
<SELECT NAME="theStyle" onChange="setFont(this.form)">
    <OPTION SELECTED VALUE="Plain">Plain
    <OPTION VALUE="Bold">Bold
    <OPTION VALUE="Italic">Italic
</SELECT><BR>
Select Font Size:
<SELECT NAME="theSize" onChange="setFont(this.form)">
    <OPTION VALUE="12">12
    <OPTION VALUE="18">18
    <OPTION SELECTED VALUE="24">24
    <OPTION VALUE="30">30
</SELECT><P>
<HR>
<INPUT TYPE="button" NAME="getInfo" VALUE="Applet Info…"
onClick="getAppletInfo(this.form)">
<P>
<TEXTAREA NAME="details" ROWS=11 COLS=70></TEXTAREA>
</FORM>
<HR>
</BODY>
</HTML>
```

Very little of the code here controls the applet — only the handful of functions near the top. The rest of the code makes up the HTML user interface for the form element controls. After you open this document from the CD-ROM, be sure to click the Applet Info button to see the methods that you can script and the way that the parameter values from the JavaScript side match up with the parameters on the Java method side.

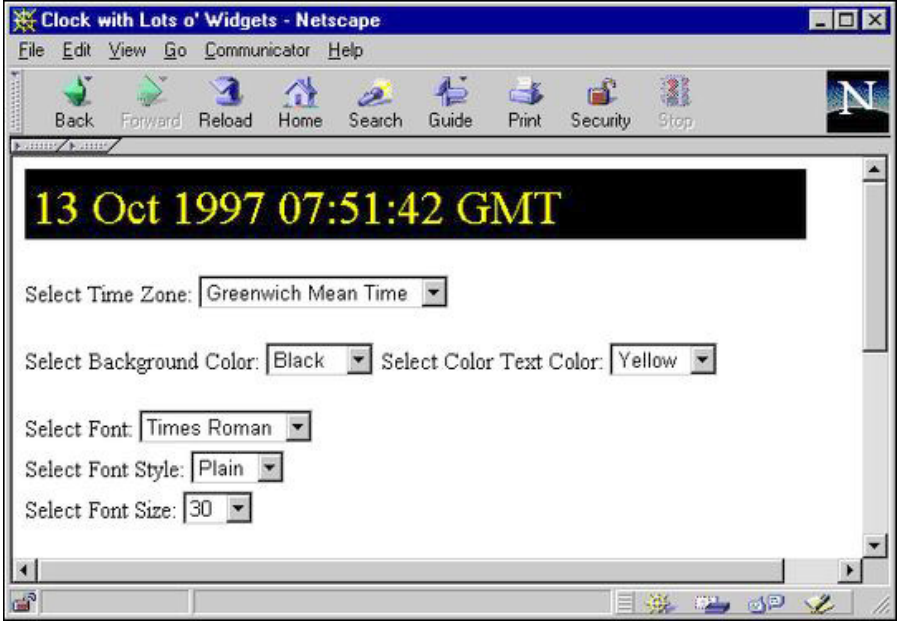

Figure 44-2
Scripting more of the ScriptableClock applet

Applet limitations

Because of concerns about security breaches via LiveConnect, Netscape clamps down on some powers that would be nice to have via a scripted applet. The most noticeable barrier is the one that prevents applets from accessing the network under scripted control. Therefore, even though a Java applet has no difficulty reading or writing text files from the server, such capabilities — even if built into an applet of your own design — won't be carried out if triggered by a JavaScript call to the applet.

Some clever hacks used to be posted on the Web, but they were rather cumbersome to implement and may no longer work on more modern browsers. You can also program the Java applet to fetch a text file after it starts up and then script the access of that value from JavaScript (as described in the following section). Signed scripts (Chapter 46) and applets can break through these security barriers after the user has given explicit permission to do so.

Faceless applets

Until LiveConnect came along, Java applets were generally written to show off data and graphics — to play a big role in the presentation on the page. But if you prefer to let an applet do the heavy algorithmic lifting for your pages while the HTML form elements and images (or Dynamic HTML facilities of newer browsers) do the user interface, you essentially need what I call a *faceless applet*.

The method for embedding a faceless applet into your page is the same as embedding any applet: Use the <APPLET> tag. But specify only 1 pixel for both the HEIGHT and WIDTH attributes (0 has strange side effects). This setting creates a dot on the screen, which, depending on your page's background color, may be completely invisible to page visitors. Place it at the bottom of the page, if you like.

To show how nicely this method can work, Listing 44-4 provides the Java source code for a simple applet that retrieves a specific text file and stores the results in a Java variable available for fetching by the JavaScript shown in Listing 44-5. The HTML even automates the loading process by triggering the retrieval of the Java applet's data from an onLoad event handler.

Listing 44-4
Java Applet Source Code

```
import java.net.*;
import java.io.*;

public class FileReader extends java.applet.Applet implements Runnable {

    Thread thread;
    URL url;
    String output;
    String fileName = "Bill of rights.txt";

    public void getFile(String fileName) throws IOException {
        String result, line;
        InputStream connection;
        DataInputStream dataStream;
        StringBuffer buffer = new StringBuffer();

        try {
            url = new URL(getDocumentBase(),fileName);
        }
        catch (MalformedURLException e) {
            output = "AppletError " + e;
        }

        try {
            connection = url.openStream();
            dataStream = new DataInputStream(new
BufferedInputStream(connection));

            while ((line = dataStream.readLine()) != null) {
                buffer.append(line + "\n");
            }
            result = buffer.toString();
        }
        catch (IOException e) {
            result = "AppletError: " + e;
        }
        output = result;
    }

    public String fetchText() {
        return output;
    }

    public void init() {
    }
```

```
    public void start() {
        if (thread == null) {
            thread = new Thread(this);
            thread.start();
        }
    }
    public void stop() {
        if (thread != null) {
        thread.stop();
            thread = null;
        }
    }

    public void run(){
        try {
            getFile(fileName);
        }
        catch (IOException e) {
            output = "AppletError: " + e;
        }
    }
}
```

All the work of actually retrieving the file is performed in the `getFile()` method (which runs immediately after the applet loads). Notice that the name of the file to be retrieved, `Bill of Rights.txt`, is stored as a variable near the top of the code, making it easy to change for a recompilation, if necessary. You can also modify the applet to accept the file name as an applet parameter, specified in the HTML code. Meanwhile, the only hook that JavaScript needs is the one public method called `fetchText()`, which merely returns the value of the output variable, which in turn holds the file's contents.

This Java source code must be compiled into a Java class file (already compiled and included on the CD-ROM as `FileReader.class`) and placed in the same directory as the HTML file that loads this applet. Also, no explicit pathname for the text file is supplied in the source code, so the text file is assumed to be in the same directory as the applet.

Listing 44-5
HTML Asking Applet to Read Text File

```
<HTML>
<HEAD>
<TITLE>Letting an Applet Do The Work</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
function getFile(form) {
    var output = document.readerApplet.fetchText()
    form.fileOutput.value = output
}
function autoFetch() {
    var output = document.readerApplet.fetchText()
    if (output != null) {
        document.forms[0].fileOutput.value = output
        return
    }
    var t = setTimeout("autoFetch()",1000)
}
</SCRIPT>
</HEAD>
<BODY onLoad="autoFetch()">

<H1>Text from a text file...</H1>
<FORM NAME="reader">
```

```
<INPUT TYPE="button" VALUE="Get File" onClick="getFile(this.form)">
<P>
<TEXTAREA NAME="fileOutput" ROWS=10 COLS=60 WRAP="hard"></TEXTAREA>
<P>
<INPUT TYPE="Reset" VALUE="Clear">
</FORM>
<APPLET CODE="FileReader.class" NAME="readerApplet" WIDTH=1 HEIGHT=1>
</APPLET>
</BODY>
</HTML>
```

Because an applet is usually the last detail to finish loading in a document, you can't use an applet to generate the page immediately. At best, an HTML document can display a pleasant welcome screen while the applet finishes loading itself and running whatever it does to prepare data for the page's form elements. In IE4+, the page can then be dynamically constructed out of the retrieved data; for NN4, you can create a new layer object, and use `document.write()` to install content into that layer. Notice in Listing 44-5 that the `onLoad` event handler calls a function that checks whether the applet has supplied the requested data. If not, then the same function is called repeatedly in a timer loop until the data is ready and the textarea can be set. The `<APPLET>` tag is located at the bottom of the Body, set to 1 pixel square — invisible to the user. No user interface exists for this applet, so you have no need to clutter up the page with any placeholder or bumper sticker.

Figure 44-3 shows the page generated by the HTML and applet working together. The Get File button is merely a manual demonstration of calling the same applet method that the onLoad event handler calls.

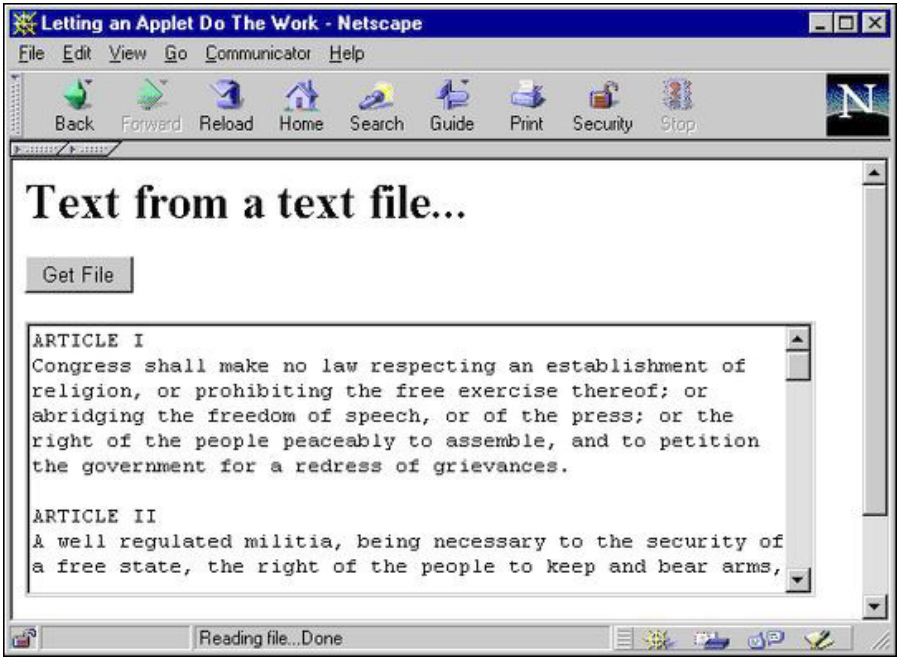

Figure 44-3
The page with text retrieved from a server file.

A faceless applet may be one way for Web authors to hide what may otherwise be JavaScript code that is open to any visitor's view. For example, if you want to deliver a small data collection lookup with a document, but don't want the array of data to be visible in the JavaScript code, you can create the array and lookup functionality inside a faceless applet. Then use form controls and JavaScript to act as query entry and output display devices (or dynamically generate a table in IE4+). Because the parameter values passed between JavaScript and Java applets must be string, numeric, or Boolean values, you won't be able to pass arrays without performing some amount of conversion either within the applet or the JavaScript code (JavaScript's *`string`*`.split()` and *`array`*`.join()` methods help a great deal here).

Data type conversions

The example in this chapter does not pass any parameters to the applet's methods, but you are free to do so. You need to pay attention to the way in which values are converted to Java data types. JavaScript strings and Boolean values are converted to Java String and Boolean objects. All JavaScript numbers, regardless of their subtype (that is, integer or floating-point number), are converted to Float objects. Therefore, if a method must accept a numeric parameter from a script, the parameter variable in the Java method must be defined as a Float type.

The distinction between JavaScript string values and string objects can impact data being passed to an applet. If an applet method requires a string object as a parameter, you may have to explicitly convert a JavaScript string value (for example, a string from a text field) to a string object via the `new String()` constructor (Chapter 34).

You can also pass references to objects, such as form control elements. Such objects get wrapped with a `JSObject` type (see discussion about this class later in the chapter). Therefore, parameter variables must be established as type `JSObject` (and the `netscape.javascript.JSObject` class must be imported into the applet).

Applet-to-Script Communication

The flip side of scripted applet control is having an applet control script and HTML content in the page. Before you undertake this avenue in page design, you must bear in mind that any calls made from the applet to the page are hard-wired for the specific scripts and HTML elements in the page. If this level of tight integration and dependence suits the application, the link up will be successful.

> **Note**
>
> The discussion of applet-to-script communication assumes you have experience writing Java applets. I use Java jargon quite freely in this discussion.

What your applet needs

NN3 and NN4 come with a zipped set of special class files tailored for use in LiveConnect. In NN3, the file is named `java_30` or `java_301`, the latter one being the latest version; in NN4, the file is named `java40.jar`. For NN6, the class files are located in an archive called `jaws.jar` (Windows) or `MRJPlugin.jar` (Mac).Use the file search facility of the OS to locate the relevant file on your system.Microsoft versions of these class files are also included in IE4+, buried in one of the large `.zip` files in the Windows\Java\Packages directory (the files you need are in one of the multi-megabyte `.zip` files, whose gibberish names change from version to version — open each with an unzip utility and look for the two packages mentioned next). The browser must see these class files (and have both Java and JavaScript enabled in the preferences screens) for LiveConnect to work.

These zipped class library files contain two vital classes in a `netscape` package (yes, even in IE):

```
netscape.javascript.JSObject
netscape.javascript.JSException
```

Both classes must be imported to your applet via the Java `import` compiler directive:

```
import netscape.javascript.*;
```

When the applet runs, the LiveConnect-aware browser knows how to find the two classes, so that the user doesn't have to do anything special as long as the supporting files are in their default locations.

Perhaps the biggest problem applet authors have with LiveConnect is importing these class libraries for applet compilation. Your Java compiler must be able to see these class libraries for compilation to be successful. The prescribed method is to include the path to the zipped class file (either the Netscape `.jar` archive or Microsoft `.zip` file) in the class path for the compiler.

Problems frequently occur when the Java compiler you use (perhaps inside an integrated development environment, such as Cafe) doesn't recognize either of the Netscape files as a legitimate zipped class file. You can make your compilation life simpler if you extract the `netscape` package from the `.jar` or `.zip` file, and place it in the same directory in which your compiler looks for the basic Java classes. For example, although the precise details may change in newer versions, Cafe stores the default Java class files inside zipped collections whose class paths (in Windows) are

```
C:\CAFE\BIN\..\JAVA\LIB\CLASSES.ZIP
C:\CAFE\BIN\..\JAVA\LIB\SYMCLASS.ZIP
```

These two class paths are inserted into new projects by default. Extract the two `netscape.javascript` class files and store them in the same LIB directory as CLASSES.ZIP and SYMCLASS.ZIP. In other words, in the LIB directory is a directory named `netscape`; inside the `netscape` directory is another directory named `javascript`; inside the `javascript` directory are the `JSObject.class` and

`JSException.class` files. Then I add the following class path to the project's class path setting:

```
C:\CAFE\BIN\..JAVA\LIB\
```

This path instructs Cafe to start looking for the `netscape` package (which contains the `javascript` package, which, in turn, contains the class files) in that directory.

Depending on the unzipping utility and operating system you use, you may have to force the utility to recognize `.jar` files as as zip archive files. If necessary, instruct the utility's file open dialog box to locate all file types in the directory. Both files will open as zipped archives. Sort the long list of files by name. Then select and extract only the two class files into the same directory as your compiler's Java class files. The utility should take care of creating the package directories for you.

What your HTML needs

As a security precaution, an <APPLET> tag requires one extra attribute to give the applet permission to access the HTML and scripting inside the document. That attribute is the single word MAYSCRIPT, and it can go anywhere inside the <APPLET> tag, as follows:

```
<APPLET CODE="myApplet.class" HEIGHT="200" WIDTH="300" MAYSCRIPT>
```

Permission is not required for JavaScript to access an applet's methods or properties, but if the applet initiates contact with the page, this attribute is required.

About JSObject.class

The portal between the applet and the HTML page that contains it is the `netscape.javascript.JSObject` class. This object's methods let the applet contact document objects and invoke JavaScript statements. Table 44-1 shows the object's methods and one static method.

Table 44-1
JSObject Class Methods

Method	Description
`call(String functionName, Object args[])`	Invokes JavaScript function, argument(s) passed as an array
`eval(String expression)`	Invokes a JavaScript statement
`getMember(String elementName)`	Retrieves a named object belonging to a container
`getSlot(Int index)`	Retrieves indexed object belonging to a container
`getWindow(Applet applet)`	Static method retrieves applet's containing

	window
`removeMember(String elementName)`	Removes a named object belonging to a container
`setMember(String elementName, Object value)`	Sets value of a named object belonging to a container
`setSlot(int index, Object value)`	Sets value of an indexed object belonging to a container
`toString()`	Returns string version of `JSObject`

Just as the `window` object is the top of the document object hierarchy for JavaScript references, the `window` object is the gateway between the applet code and the scripts and document objects. To open that gateway, use the `JSObject.getWindow()` method to retrieve a reference to the document window. Assign that object to a variable that you can use throughout your applet code. The following code fragment shows the start of an applet that assigns the window reference to a variable named `mainwin`:

```
import netscape.javascript.*;

public class myClass extends java.applet.Applet {
    private JSObject mainwin;

    public void init() {
    mainwin = JSObject.getWindow(this);
    }
}
```

If your applet will be making frequent trips to a particular object, you may want to create a variable holding a reference to that object. To accomplish this, the applet needs to make progressively deeper calls into the document object hierarchy with the `getMember()` method. For example, the following sequence assumes `mainwin` is a reference to the applet's document window. Eventually the statements set a form's field object to a variable for use elsewhere in the applet:

```
JSObject doc = (JSObject) mainwin.getMember("document");
JSObject form = (JSObject) doc.getMember("entryForm");
JSObject phonefld = (JSObject) form.getMember("phone");
```

Another option is to use the Java `eval()` method to execute an expression from the point of view of any object. For example, the following statement gets the same field object from the preceding fragment:

```
JSObject phonefld = mainwin.eval("document.entryForm.phone");
```

As soon as you have a reference to an object, you can access its properties via the `getMember()` method, as shown in the following example, which reads the `value` property of the text box, and casts the value into a Java `String` object:

```
String phoneNum = (String) phonefld.getMember("value");
```

Two `JSObject` class methods let your applet execute arbitrary JavaScript expressions and invoke object methods: the `eval()` and `call()` methods. Use these methods with any `JSObject`. If a value is to be returned from the executed statement, you must cast

the result into the desired object type. The parameter for the `eval()` method is a string of the expression to be evaluated by JavaScript. Scope of the expression depends on the object attached to the `eval()` method. If you use the `window` object, the expression would exist as if it were a statement in the document script (not defined inside a function).

Using the `call()` method is convenient for invoking JavaScript functions in the document, although it requires a little more preparation. The first parameter is a string of the function name. The second parameter is an array of arguments for the function. Parameters can be of mixed data types, in which case the array would be of type `Object`. If you don't need to pass a parameter to the function call, you can define an array of a single empty string value (for example, `String arg[] = {""}`) and pass that array as the second parameter.

Data type conversions

The strongly typed Java language is a mismatch for loosely typed JavaScript. As a result, with the exception of Boolean and string objects (which are converted to their respective JavaScript objects), you should be aware of the way LiveConnect adapts data types to JavaScript.

Any Java object that contains numeric data is converted to a JavaScript number value. Because JavaScript numbers are IEEE doubles, they can accommodate just about everything Java can throw its way.

If the applet extracts an object from the document and then passes that `JSObject` type back to JavaScript, that passed object is converted to its original JavaScript object type. But objects of other classes are passed as their native objects wrapped in JavaScript "clothing." JavaScript can access the applet object's methods and properties as if the object were a JavaScript object. Finally, Java arrays are converted to the same kind of JavaScript array created via the `new Array()` constructor. Elements can be accessed by integer index values (not named index values). All other JavaScript array properties and methods apply to this object as well.

Example applet-to-script application

To demonstrate several techniques for communicating from an applet to both JavaScript scripts and document objects, I present an applet that displays two simple buttons (see Figure 44-4). One button generates a new window, spawned from the main window, filling the window with dynamically generated content from the applet. The second button communicates from the applet to that second window by invoking a JavaScript function in the document. One last part of the demonstration shows the applet changing the value of a text box when the applet starts up.

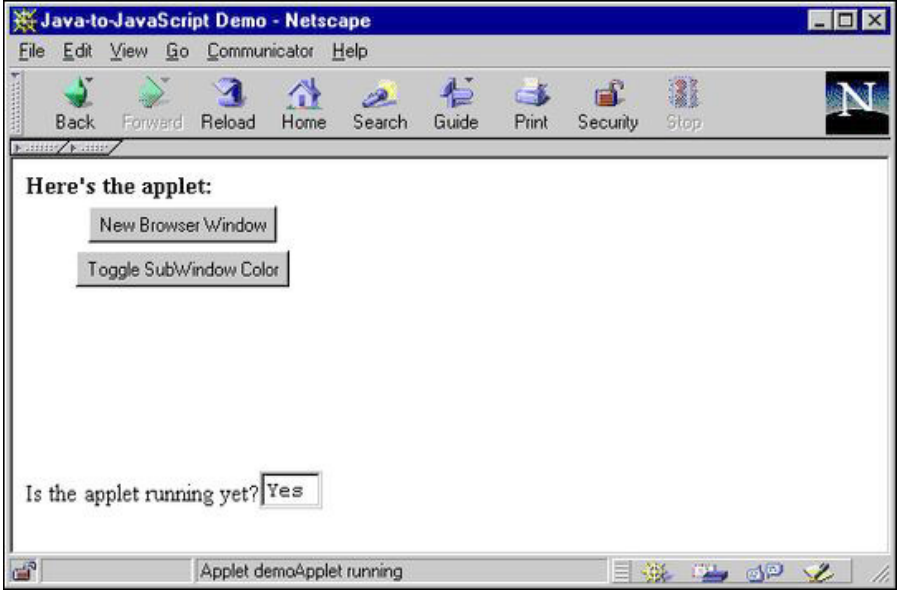

Figure 44-4
The applet displays two buttons seamlessly on the page.

Listing 44-6 shows the source code for the Java applet. For backward compatibility, it uses the JDK 1.02 event handling model.

Because the applet generates two buttons, the code begins by importing the `AWT` interface builder classes. I also import the `netscape.javascript` package to get the `JSObject` class. The name of this sample class is `JtoJSDemo`. I declare four global variables: two for the windows, two for the applet button objects.

Listing 44-6
Java Applet Source Code

```
import java.awt.*;
import netscape.javascript.*;

public class JtoJSDemo extends java.applet.Applet {
    private JSObject mainwin, subwin;
    private Button newWinButton, toggleButton;
```

The applet's `init()` method establishes the user interface elements for this simple applet. A white background is matched in the HTML with a white document background color, making the applet appear to blend in with the page. I use this opportunity to set the `mainwin` variable to the browser window that contains the applet.

```
    public void init() {
        setBackground(Color.white);
        newWinButton = new Button("New Browser Window");
        toggleButton = new Button("Toggle SubWindow Color");
        this.add(newWinButton);
        this.add(toggleButton);
        mainwin = JSObject.getWindow(this);
    }
```

As soon as the applet starts, it changes the `value` property of a text box in the HTML form. Because this is a one-time access to the field, I elected to use the `eval()` method from the point of view of the main window, rather than build successive object references through the object hierarchy with the `getMember()` method.

```
    public void start() {
        mainwin.eval("document.indicator.running.value = 'Yes'");
    }
```

Event handling is quite simple in this application. A click of the first button invokes `doNewWindow()`; a click of the second invokes `toggleColor()`. Both methods are defined later in the applet.

```
    public boolean action(Event evt, Object arg) {
        if (evt.target instanceof Button) {
            if (evt.target == newWinButton) {
                doNewWindow();
            } else if (evt.target == toggleButton) {
                toggleColor();
            }
        }
        return true;
    }
```

One of the applet's buttons calls the `doNewWindow()` method defined here. I use the `eval()` method to invoke the JavaScript `window.open()` method. The string parameter of the `eval()` method is exactly like the statement that appears in the page's JavaScript to open a new window. The `window.open()` method returns a reference to that subwindow, so that the statement here captures the returned value, casting it as a `JSObject` type for the `subwin` variable. That `subwin` variable can then be used as a reference for another `eval()` method that writes to that second window. Notice that the object to the left of the `eval()` method governs the recipient of the `eval()` method's expression. The same is true for closing the writing stream to the subwindow.

> **Note**
>
> Unfortunately, the IE4+ implementation of `JSObject` does not provide a suitable reference to the external window after it is created. Therefore, the window does not receive its content or respond to color changes in this example. Due to other anomalies with subwindows, I advise against using LiveConnect powers with multiple windows in IE4+.

Listing 44-6 (continued)

Java Applet Source Code

```
    void doNewWindow() {
        subwin = (JSObject)
mainwin.eval("window.open('','fromApplet','HEIGHT=200,WIDTH=200')");
        subwin.eval("document.write('<HTML><BODY BGCOLOR=white>Howdy from the
applet!</BODY></HTML>')");
        subwin.eval("document.close()");
    }
```

The second button in the applet calls the `toggleColor()` method. In the HTML document, a JavaScript function named `toggleSubWindowColor()` takes a `window` object reference as an argument. Therefore, I first assemble a one-element array of type `JSObject` consisting of the `subwin` object. That array is the second parameter of the `call()` method, following a string version of the JavaScript function name being called.

```
    void toggleColor() {
        if (subwin != null) {
            JSObject arg[] = {subwin};
            mainwin.call("toggleSubWindowColor", arg);
        }
    }
}
```

Now onto the HTML that loads the above applet class and is the recipient of its calls. The document is shown in Listing 44-7. One function is called by the applet. A text box in the form is initially set to "No," but gets changed to "Yes" by the applet after it has finished its initialization. The only other item of note is that the `<APPLET>` tag includes a `MAYSCRIPT` attribute to allow the applet to communicate with the page.

Listing 44-7

HTML Document Called by Applet

```
<HTML>
<HEAD><TITLE>Java-to-JavaScript Demo</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function toggleSubWindowColor(wind) {
    if (wind.closed) {
        alert("The subwindow is closed.  Can't change it's color.")
    } else {
        wind.document.bgColor = (wind.document.bgColor == "#ffffff") ? "red" :
"white"
    }
}
</SCRIPT>
</HEAD>

<BODY BGCOLOR="#FFFFFF">
<B>Here's the applet:</B><BR>
<APPLET CODE="JtoJSDemo.class" NAME="demoApplet" HEIGHT=150 WIDTH=200
MAYSCRIPT>
</APPLET>

<FORM NAME="indicator">
Is the applet running yet?<INPUT TYPE="text" NAME="running" SIZE="4"
VALUE="No">
</FORM>
</BODY>
</HTML>
```

Scripting Plug-ins

Controlling a plug-in (or Windows ActiveX control in IE) from JavaScript is much like controlling a Java applet. But you have more browser-specific concerns to worry about, even at the HTML level. Not all plug-ins are scriptable, of course, nor do all browsers permit such scripting, as described at the start of this chapter. Yet even when you have found the right combination of browser version(s) and plug-in(s), you must also learn

what the properties and/or methods of the plug-in are so that your scripts can control them. For common plug-in duties, such as playing audio, the likelihood that all users will have the same audio playback plug-in installed in a particular browser brand and operating system is perhaps too small to entrust your programming to a single plug-in. If, on the other hand, you are using a plug-in that works only with a special data type, then your page need check only that the plug-in is installed (and that it is the desired minimum version).

In this section of the chapter, you'll begin to understand the HTML issues and then examine two separate audio playback examples. One example lets users change tunes being played back; the other arrives with five sounds, each of which is controlled by a different onscreen interface element. Both of these audio playback examples employ a library that has been designed to provide basic audio playback interfaces to the three most popular scriptable audio playback plug-ins:

* Windows Media Player 6.4
* Apple QuickTime 4.1 or later
* Netscape LiveAudio (for NN3 and NN4)

The main goal of the library is to act as an API (Application Programming Interface) between your scripts and the three plug-ins. Your scripts issue one command, and the library figures out which plug-in is installed and how that particular command must be communicated to the installed plug-in. Additional verification takes place in the initialization routine to verify that a valid plug-in is installed in the user's browser.

The HTML side

Depending on the browser and operating system that you're using, one of two tags can be used to put the plug-in's powers into the page. With the plug-in embedded within the page (even if you don't see it), the plug-in becomes part of the document's object model, which means that your scripts can address it.

Using EMBED

The preferred way to embed such content into a page for NN (all OSes) and IE/Mac is to use the `<EMBED>` tag. Even though the W3C HTML standard does not recognize the EMBED element, it has been a part of browser implementations since the first embeddable media. The element is also a bit of a chameleon, because beyond a common set of recognized attributes, such as the `SRC` attribute that points to the content file to be loaded into the plug-in, its attributes are extensible to include items that apply only to a given plug-in. Uncovering the precise lists of attributes and values for a plug-in is not always easy, and frequently requires digging deeply into the developer documentation of the plug-in's producer. It is not unusual for a page author to anticipate that multiple plug-ins could play a particular kind of data (as is the case in the audio examples later in this chapter).

Therefore, a single EMBED element may include attributes that apply to more than one plug-in. You have to hope that the plug-ins' developers chose unique names for their attributes or that like-named attributes mean the same thing in multiple plug-ins. Any attributes that a plug-in doesn't recognize are ignored.

Typical behavior for a plug-in is to display some kind of controller or other panel in a rectangle associated with the media. You definitely need to specify the `HEIGHT` and `WIDTH` attribute values of such an EMBED element if it is to display visual media (some video plug-ins let you hide the controls, while still showing the viewing area). For audio, however, you can specify a one-pixel value for both dimensions, and leave the controls to your HTML content. Browsers that recognize style sheets can also set EMBED elements to be invisible.

As an example of what an EMBED element may look like, the following is adapted from Listing 44-9. The example includes attributes that apply to QuickTime and LiveAudio and is formatted here for ease of readability.

```
<EMBED NAME="jukebox"
    HEIGHT=1
    WIDTH=1
    SRC="Beethoven.aif"
    HIDDEN=TRUE
    AUTOSTART=FALSE
    AUTOPLAYT=FALSE
    ENABLEJAVASCRIPT=TRUE
    MASTERSOUND>
</EMBED>
```

After the page loads and encounters this tag, the browser reaches out to the server and loads the sound file into the plug-in, where it sits quietly until the plug-in is instructed to play it.

IE/Windows OBJECT

In the IE/Windows camp, the preferred way to get external media into the document is to load the plug-in (ActiveX control) as an object via the `<OBJECT>` tag. The OBJECT element is endorsed by the W3C HTML standard. In many ways the `<OBJECT>` tag works like the `<APPLET>` tag in that aside from specifying attributes that load the plug-in, additional nested PARAM elements let you make numerous settings to the plug-in while it loads, including the name of the file to pre-load. As with a plug-in's attributes, an object's parameters are unique to the object and are documented (somewhere) for every object intended to be put into an HTML page.

IE/Windows has a special (that is, far from intuitive) way it refers to the plug-in program: through its class ID (also known as a GUID). You must know this long string of numbers and letters in order to embed the object into your page. If you are having difficulty getting this information from a vendor, see Chapter 32 for tips on how to hunt for the information yourself. There, you also discover how to find out what parameters apply to an object.

The following example is an OBJECT element that loads the Windows Media Player 6.*x* plug-in (ActiveX control) into a page. The example is adapted from Listing 44-9.

```
<OBJECT ID="jukebox" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="FileName" VALUE="Beethoven.aif">
<PARAM NAME="AutoStart" VALUE="false">
</OBJECT>
```

When you compare the EMBED and OBJECT approaches, you can see many similar properties and values, which are just expressed differently (for example, attributes versus PARAM elements).

Using EMBED and OBJECT together

Because a public Web page must usually appeal to a broad range of browsers, you should design such a page to work with as many browsers as possible. For the convenience of your scripting (and especially if you use the audio playback API described later in this chapter), referring to a plug-in object by the same identifier is helpful, whether it is loaded via an EMBED or OBJECT element.

To the rescue comes a handy behavior of the OBJECT element. It is designed in such a way that you can nest the associated EMBED element inside the OBJECT element's tag set. If the browser doesn't know about the OBJECT element, that element is ignored, but the EMBED element is picked up. Similarly, if the browser that knows about the OBJECT element fails to load the plug-in identified in its attributes, the nested EMBED elements also get picked up. Therefore, you can combine the OBJECT and EMBED elements as shown in the following example, which combines the two previous examples:

```
<OBJECT ID="jukebox" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="FileName" VALUE="Beethoven.aif">
<PARAM NAME="AutoStart" VALUE="false">
    <EMBED NAME="jukebox"
        HEIGHT=1
        WIDTH=1
        SRC="Beethoven.aif"
        HIDDEN=TRUE
        AUTOSTART=FALSE
        AUTOPLAYT=FALSE
        ENABLEJAVASCRIPT=TRUE
        MASTERSOUND>
    </EMBED>
</OBJECT>
```

Notice that the identifier assigned to the `ID` of the OBJECT element and to the `NAME` of the EMBED element are the same. Because only one of these two elements will be valid in the document, you have no conflict of like-named elements.

Validating the plug-in

As described at length in Chapter 32, you may need to validate the installation of a particular plug-in before the external media will play. This validation is even more vital if

you want to control the plug-in from scripts, because you must have the right controlling vocabulary for each scriptable plug-in.

The coordination of plug-in and data type is not a big issue in IE/Windows, because your OBJECT element explicitly loads a known plug-in, even if the computer is equipped to play the same data type through a half-dozen different ActiveX controls. But in NN (and IE/Mac, although plug-ins are not scriptable there at least through Version 5), the association of a plug-in with a particular MIME type (data type of the incoming media) is perhaps a bit too automatic. It is not uncommon for plug-in installation programs to gobble up the associations of numerous MIME types. Knowledgeable users, who can fathom the nether worlds of browser preferences, can manually change these associations, but your scripts cannot direct a browser to use a specific plug-in to play your media unless the plug-in is already enabled for your media's MIME type. The more common and open your media's MIME type is (particularly audio and video), the more of a potential problem this presents to you. *Caveat scriptor*.

With these warnings in mind, review the approaches to checking the presence of a plug-in and its enabled status by way of the `mimeTypes` and `plugIns` objects described in Chapter 32. You see some of the routines from that chapter put to use in a moment.

The API approach

In this section, you see one version of an API that can be used to accomplish simple audio playback activities in a page through three different plug-in technologies (Windows Media Player 6, Apple QuickTime, and Netscape LiveAudio). Your scripts issue one command (for example, `play(1)`), and the API sends the precise command to the plug-in being used in the user's browser. At the same time, the API has its own initialization routine, which it uses not only to validate the plug-in being used, but alerts users of ill-equipped browsers with a relevant message about why their browser can't get the most out of the page.

This API is far from the be-all, end-all library, although you will see that it does quite a bit as-is. The code is offered as a starting point for your further development. Such development may take the shape of adding more operations to the API or adding capabilities for additional scriptable plug-ins. For example, while the API as shown supports Windows Media Player 6, Microsoft continues to upgrade the Player to new versions (with new GUIDs for your OBJECT tags) that have new command vocabularies. There is no reason that the API cannot be extended for new generations of Windows Media Player, while maintaining backward compatibility for the Version 6 generation.

You can find the complete API code on the CD-ROM within the folder of example listings for this chapter. The API file is named `DGAudioAPI.js`. Check out the following high points of this library.

Loading the library

Adding the library to your page is no different from any external `.js` library file. Include the following tag in the HEAD of your page:

```
<SCRIPT LANGUAGE="JavaScript" SRC="DGAudioAPI.js"></SCRIPT>
```

Except for two global variable initializations, no immediate code runs from the library. All of its activity is invoked from event handlers or other script statements in the main page.

Initializing the library

The first job for the library is to validate that your sounds have one of the three known plug-in technologies available. Before the library can do this, all loading of the OBJECT or EMBED elements must be concluded so that the objects exist for the initialization routine to examine. Therefore, use the `onLoad` event handler in the BODY to invoke the `initAudioAPI()` function. Parameters to be passed to this function are vital pieces of information.

Parameter values consist of one or more two-element arrays. The first value is a string of the identifier, which is assigned to the OBJECT and EMBED elements (recall that they are the same identifiers); the second value is a string of the MIME type. Getting the desired value may take some trial and error if you aren't familiar with MIME type terminology. Use the Edit/Preferences/Applications dialog box window listings in NN as a guide in finding the name of a MIME type based on the file name extension of the media file.

The following is an excerpt from Listing 44-9, which shows how the `jukebox` player object is initialized for the `audio/x-aiff` MIME type (all sound files for examples in this chapter have the `.aif` file name extension):

```
onLoad="initAudioAPI(['jukebox', 'audio/x-aiff'])"
```

Notice how the square bracket literal array syntax is used both to create the array of two values while passing them as parameters to the function. NN uses the MIME type to make sure that the plug-in that fired up as a result of the EMBED element is enabled for the MIME type.

As you see in Listing 44-10, the `initAudioAPI()` function lets you initialize multiple player objects, each one with its own MIME type, if necessary. Each object and MIME type pair are passed as their own array. For example, the following initializes the library for two different embedded plug-in objects, although both have the same MIME type:

```
onLoad="initAudioAPI(['cNatural','audio/x-aiff'],['cSharp','audio/x-aiff'])"
```

When the function receives multiple arrays, it loops through them, performing the initializations in sequence. The `initAudioAPI()` function follows:

```
function initAudioAPI() {
    var args = initAudioAPI.arguments
    var id, mime
    for (var i = 0; i < args.length; i++) {
        // don't init any more if browser lacks scriptable sound
        if (OKToTest) {
            id = args[i][0]
```

```
            mime = args[i][1]
            players[id] = new API(id, mime)
            players[id].type = setType(id, mime)
        }
    }
}
```

Notice that parameter variables are not explicitly declared for the function, but are, instead, retrieved via the `arguments` property of the function. The global `OKToTest` flag, initialized to true when the library loads, is set to false if the validation of a plug-in fails. The conditional construction here prevents multiple alerts from appearing when multiple plug-in and MIME type parameters are passed to the initialization function.

Sound player API objects

One of the jobs of the initialization routine is to create a player object for each plug-in identifier. The object's constructor is as follows:

```
// AudioAPI object constructor
function API(id, mime) {
    this.id = id
    this.type = ""  // values can be "isLA","isMP","isQT"
    this.mimeType = mime
    this.play = API_play
    this.stop = API_stop
    this.pause = API_pause
    this.rewind = API_rewind
    this.load = API_load
    this.getVolume = API_getVolume
    this.setVolume = API_setVolume
}
```

The object becomes a convenient place to preserve properties for each sound controller, including which type of plug-in it uses (described in a moment). But the bulk of the object is reserved for assigning methods — the methods that your main page's scripts invoke to play and stop the player, adjust its volume, and so on. The method names to the left of the assignment statements in the object constructor are the names your scripts use; the functions in the library (for example, `API_play()`) are the ones that send the right command to the right plug-in.

Each of these objects (even if there is only one for the page) is maintained in a hash table-like array (named `players[]`) in the library. The plug-in object's identifier is the string index for the array entry. This provides the gateway to your page's scripts. For example, if you initialize the library with a single identifier, `jukebox`, you access the methods of the library's jukebox-related player object through the array and the identifier:

```
players["jukebox"].rewind()
```

Plug-in checking

One more part of the initialization routine inside the library is a call to the `setType()` function, which ultimately assigns a value to the `players[]` object `type` property. For a valid plug-in, the value of the `type` property can be `isLA` (LiveAudio), `isMP` (Windows Media Player), `isQT` (QuickTime), or an empty string. Listing 44-8 shows code for the `setType()` function and some supporting functions.

Listing 44-8

setType() and Supporting Functions from DGAudioAPI.js

```
function setType(id, mime) {
    var type = ""
    var errMsg = "This browser is not equipped for scripted sound.\n\n"
    var OS = getOS()
    var brand = getBrand()
    var ver = getVersion(brand)
    if (brand == "IE") {
        if (ver > 4) {
            if (document.all(id) && document.all(id).HasError) {
                errMsg = document.all(id).ErrorDescription
            } else {
                if (OS == "Win") {
                    if (document.all(id) && document.all(id).CreationDate !=
"") {
                        return "isMP"
                    } else {
                        errMsg += "Expecting Windows Media Player Version 6.4."
                    }
                } else {
                    errMsg += "Only Internet Explorer for Windows is
supported."
                }
            }
        } else {
            errMsg += "Only Internet Explorer 4 or later for Windows is
supported."
        }
    } else if (brand == "NN") {
        if ((ver >= 3 && ver < 4.6) || (ver >= 4.7 && ver < 6)) {
            if (mimeAndPluginReady(mime, "LiveAudio")) {
                return "isLA"
            }
            if (mimeAndPluginReady(mime, "QuickTime")) {
                qtVer = parseFloat(document.embeds[id].GetPluginVersion(), 10)
                if (qtVer >= 4.1) {
                    return "isQT"
                } else {
                    errMsg += "QuickTime Plugin 4.1 or later is required."
                }
            } else {
                errMsg += "Sound control requires QuickTime Plugin 4.1 "
                errMsg += "(or later) or LiveAudio "
                errMsg += "enabled for MIME type: \'" + mime + "\'."
            }
        } else {
            errMsg += "Requires Navigator 3.x, 4.0-4.5, or 4.7-4.9."
        }
    } else {
        errMsg += "This page is certified only for versions of Internet
Explorer "
        errMsg == "and Netscape Navigator."
    }
    alert(errMsg)
    OKToTest = false
    return type
}

function getOS() {
    var ua = navigator.userAgent
    if (ua.indexOf("Win") != -1) {
        return "Win"
    }
    if (ua.indexOf("Mac") != -1) {
```

```
        return "Mac"
    }
    return "Other"
}

function getBrand() {
    var name = navigator.appName
    if (name == "Netscape") {
        return "NN"
    }
    if (name.indexOf("Internet Explorer") != -1) {
        return "IE"
    }
    return "Other"
}

function getVersion(brand) {
    var ver = navigator.appVersion
    var ua = navigator.userAgent
    if (brand == "NN") {
        if (parseInt(ver, 10) < 5) {
            return parseFloat(ver, 10)
        } else {
            // get full version for NN6+
            return parseFloat(ua.substring(ua.lastIndexOf("/")+1))
        }
    }
    if (brand == "IE") {
        var IEOffset = ua.indexOf("MSIE ")
        return parseFloat(ua.substring(IEOffset + 5, ua.indexOf(";",
IEOffset)))
    }
    return 0
}
```

The `setType()` function is an extensive decision tree that uses clues from the `navigator.userAgent` and `navigator.appVersion` properties to determine what environment is currently running. For each environment, plug-in detection takes place to verify that either the desired Windows ActiveX object is installed in IE or that one of the acceptable plug-ins is running in NN. All of the detection code is taken from Chapter 32. One of the advantages of such a detailed decision tree is that if a decision branch fails, it is for a reasonably specific reason — enough detail to advise the user intelligently about why the current browser can't do what the page author wants it to do.

Invoking methods

Establishing the `players[]` object type is a critical operation of this library, because all subsequent operation depends on the type being set. For example, to perform the action of rewinding the sound to the beginning, your script invokes the following statement:

```
players["jukebox"].rewind()
```

This, in turn invokes the library's `API_rewind()` function:

```
function API_rewind() {
    switch (this.type) {
        case "isLA" :
            document.embeds[this.id].stop()
            document.embeds[this.id].start_at_beginning()
            break
        case "isQT" :
            document.embeds[this.id].Stop()
```

```
            document.embeds[this.id].Rewind()
            break
        case "isMP" :
            if (document.embeds[this.id]) {
                document.embeds[this.id].Stop()
                document.embeds[this.id].CurrentPosition = 0
            } else {
                document.all(this.id).Stop()
                document.all(this.id).CurrentPosition = 0
            }
            break
        default:
    }
}
```

Each of the three plug-ins covered in this API has an entirely different way to perform (or simulate) a rewinding of the current sound to the beginning. The `type` property of the `players[]` object invoked by your script determines which branch of the `switch` statement to follow. For each plug-in type, the appropriate document object model reference and the plug-in-specific property or method is accessed. The identifier passed as a parameter to the initialization routine continues to play a role, providing the identifier to the actual DOM object that is the plug-in controller (for example, an index to the `document.embeds[]` array).

The library contains a function just as the one you just saw for each of the seven methods assigned to `players[]` objects. They remain invisible to the user and to you as well, because you work only with the simpler `players[]` object method calls, regardless of plug-in.

> **Note**
>
> If the Windows Media Player detects a problem with the audio hardware, it doesn't always reflect the error in the object until after all `onLoad` event handler functions finish executing. This weirdness prevents the error checking from being performed where it should be, in the `setType()` function. Therefore, error checking for this possibility is performed in the API branch that commands the Media Player to play the currently loaded sound.

Extending the library

Adding more plug-in types to the library requires modification in two areas. The first is to the `setType()` function's decision tree. You have to determine where in the tree the plug-in is best detected. For another Windows Media Player, for instance, it would be along the same branch that looks for the Version 6 player.

You then need to locate the properties and methods of the new plug-in for basic operations covered in the library (play, stop, and so on). For each of the action functions, you add another case for your newly defined type. Your main Web page scripts should not require any modification (although your OBJECT and/or EMBED tag attributes may change to accommodate the new plug-in).

Building a jukebox

The first example that utilizes the `DGAudioAPI.js` library is a jukebox that provides an interface (admittedly not pretty — that's for you to whip up) for selecting and controlling multiple sound files with a single plug-in tag set. The assumption for this application is that only one sound at a time need be handy for immediate playing.

Listing 44-9 shows the code for the jukebox. All sound files specified in the example are in the same folder as the listing on the companion CD-ROM (the AIFF-format files sound better in some plug-ins than others, so don't worry about the audio quality of these demo sounds).

Listing 44-9
A Scripted Jukebox

```
<HTML>
<HEAD>
<TITLE>Oldies but Goody's</TITLE>
<SCRIPT LANGUAGE="JavaScript" SRC="DGAudioAPI.js"></SCRIPT>
<SCRIPT>
// make sure currently selected tune is preloaded
function loadFirst(id) {
    var choice = document.forms[0].musicChoice
    var sndFile = choice.options[choice.selectedIndex].value
    players[id].load(sndFile)
}
// swap tunes
function changeTune(id, choice) {
    players[id].load(choice.options[choice.selectedIndex].value)
}
// control and display volume setting
function raiseVol(id) {
    var currLevel = players[id].getVolume()
    currLevel += Math.ceil(Math.abs(currLevel)/10)
    players[id].setVolume(currLevel)
    displayVol(id)
}
function lowerVol(id) {
    var currLevel = players[id].getVolume()
    currLevel -= Math.floor(Math.abs(currLevel)/10)
    players[id].setVolume(currLevel)
    displayVol(id)
}
function displayVol(id) {
    document.forms[0].volume.value = players[id].getVolume()
}
</SCRIPT>
</HEAD>

<BODY onLoad="initAudioAPI(['jukebox', 'audio/x-aiff']); loadFirst('jukebox');
displayVol('jukebox')">
<FORM>
<TABLE BORDER=2 ALIGN="center">
<CAPTION ALIGN=top><FONT SIZE=+3>Classical Piano Jukebox</FONT></CAPTION>
<TR><TD COLSPAN=2 ALIGN=center>
<SELECT NAME="musicChoice" onChange="changeTune('jukebox', this)">
    <OPTION VALUE="Beethoven.aif" SELECTED>Beethoven's Fifth Symphony (Opening)
    <OPTION VALUE="Chopin.aif">Chopin Ballade #1 (Opening)
    <OPTION VALUE="Scriabin.aif">Scriabin Etude in D-sharp minor (Finale)
</SELECT></TD></TR>
<TR><TH ROWSPAN=4>Action:</TH>
```

```
<TD>
    <INPUT TYPE="button" VALUE="Play"
onClick="players['jukebox'].play(parseInt(this.form.frequency[
this.form.frequency.selectedIndex].value))">
<SELECT NAME="frequency">
<OPTION VALUE=1 SELECTED>Once
<OPTION VALUE=2>Twice
<OPTION VALUE=3>Three times
<OPTION VALUE=TRUE>Continually
</SELECT></TD></TR>
<TR><TD>
    <INPUT TYPE="button" VALUE="Stop" onClick="players['jukebox'].stop()">
</TD></TR>
<TR><TD>
    <INPUT TYPE="button" VALUE="Pause" onClick="players['jukebox'].pause()">
</TD></TR
<TR><TD>
    <INPUT TYPE="button" VALUE="Rewind" onClick="players['jukebox'].rewind()">
</TD></TR>
<TR><TH ROWSPAN=3>Volume:</TH>
<TD>Current Setting:<INPUT TYPE="text" SIZE=10 NAME="volume"
onFocus="this.blur()"></TD></TR>
<TR><TD>
    <INPUT TYPE="button" VALUE="Higher" onClick="raiseVol('jukebox')">
</TD></TR>
<TR><TD>
    <INPUT TYPE="button" VALUE="Lower" onClick="lowerVol('jukebox')">
</TD></TR>
</TABLE>
</FORM>

<OBJECT ID="jukebox" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="AutoStart" VALUE="false">
    <EMBED NAME="jukebox" HEIGHT=2 WIDTH=2 SRC="Beethoven.aif"
              HIDDEN=TRUE AUTOSTART=FALSE AUTOPLAY=FALSE
               ENABLEJAVASCRIPT=TRUE MASTERSOUND>
    </EMBED>
</OBJECT>

</BODY>
</HTML>
```

You can see the user interface in Figure 44-5. One SELECT element contains a list of three possible choices. Most of the interface, however, consists of buttons that ultimately invoke methods of the current plug-in.

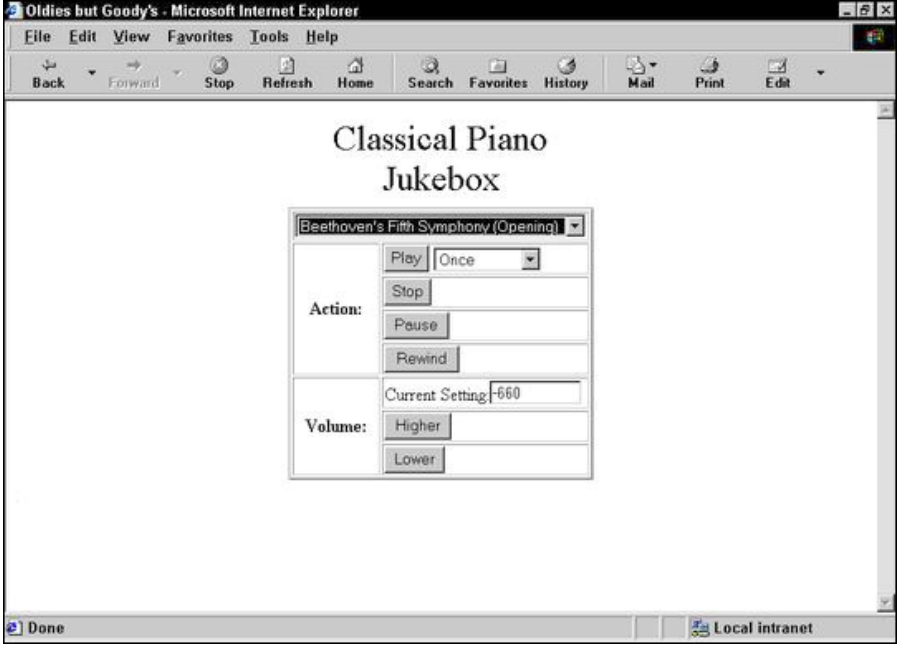

Figure 44-5
The jukebox page

Two functions are invoked by the `onLoad` event handler besides the initialization routine of the library. The `loadFirst()` function finds out which of the items in the SELECT element is chosen when the page loads, and it makes sure that the file is pre-loaded into the plug-in. This functionality is provided in case the user makes a choice and should use the Back button or history to return to the page. In some browsers, the SELECT element will be set to its most recent setting, so the `loadFirst()` function simply gets everything ready.

The second `onLoad` function call is to `displayVol()`. This function works its way through the library to read the volume setting of the plug-in and displays the resulting value in a text box in the form. Not all plug-ins use the same scale or numbering system for their volume controls. Windows Media Player 6, for instance, uses very large negative numbers, while QuickTime and LiveAudio are on different, positive scales. The other volume-related functions simply increase or decrease the current setting by 10 percent in response to clicking the associated buttons in the interface.

All functions defined for this page are designed to be as generalizable as possible. Thus, the identifier of the plug-in is passed as a parameter to each. If another plug-in were added to this page, the same functions could be used without modification, provided calls to the functions passed the identifier of the other plug-in.

All of the button controls are pretty straightforward except the Play button's `onClick` event handler. It invokes the `players[id].play()` method, but that method requires a parameter of how many times the sound should be played. In this user interface, a SELECT element controls that information. Getting the value of the selected item creates

a lengthy reference, but that's what is taking up so much space in the parameter slot of the `play()` method call.

Embedding multiple sounds

The final example of embedded media serves as a base on which you can build a page that needs to play multiple sounds without the user explicitly loading them. For example, you may have buttons generate different sounds after users click them (I'm not recommending this interface, but that won't necessarily stop you). Figure 44-6 shows you the simple five-key piano keyboard. The page loads five different sounds into the page, one for each note (actual piano sounds in this case). Each sound was recorded for about four seconds, so that you can get the action of attack and delay, just like a real piano. If you mouse down on a key, the sound plays for up to four seconds (getting softer all the time) or until you mouse up on the key (the attack time on the sample sounds on the CD-ROM is not instantaneous, so you may have to hold a key down for a fraction of a second to start the sound). The colors of the keys also change slightly to provide further user feedback to the action.

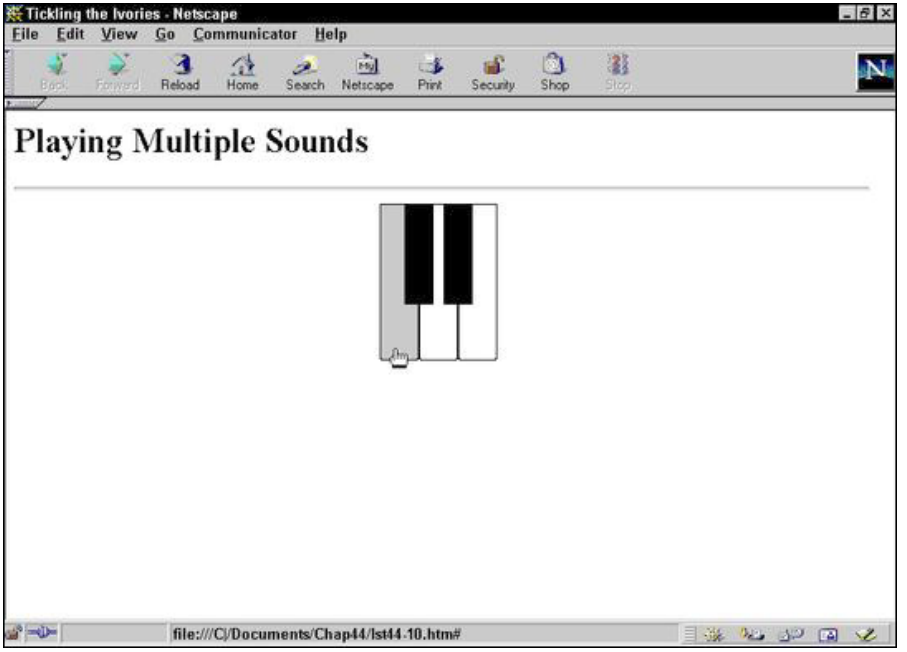

Figure 44-6
Controller for five sounds

Thanks to the `DGAudioAPI.js` library, very little code in this page is associated with the sounds. Far more is involved with the image swaps and the loading of the five plug-ins. Listing 44-10 shows the code for the page.

Listing 44-10
Scripting Multiple Sounds

```
<HTML>
```

```
<HEAD>
<TITLE>Tickling the Ivories</TITLE>
<STYLE TYPE="text/css">
OBJECT {visibility:hidden}
</STYLE>
<SCRIPT LANGUAGE="JavaScript" SRC="DGAudioAPI.js"></SCRIPT>
<SCRIPT>
// pre-cache 10 images
var onImages = new Array()
onImages["c"] = new Image(35, 140)
onImages["c"].src = "whiteDown.gif"
onImages["d"] = new Image(35, 140)
onImages["d"].src = "whiteDown.gif"
onImages["e"] = new Image(35, 140)
onImages["e"].src = "whiteDown.gif"
onImages["cHalf"] = new Image(26, 90)
onImages["cHalf"].src = "blackDown.gif"
onImages["dHalf"] = new Image(26, 90)
onImages["dHalf"].src = "blackDown.gif"

var offImages = new Array()
offImages["c"] = new Image(35, 140)
offImages["c"].src = "whiteUp.gif"
offImages["d"] = new Image(35, 140)
offImages["d"].src = "whiteUp.gif"
offImages["e"] = new Image(35, 140)
offImages["e"].src = "whiteUp.gif"
offImages["cHalf"] = new Image(26, 90)
offImages["cHalf"].src = "blackUp.gif"
offImages["dHalf"] = new Image(26, 90)
offImages["dHalf"].src = "blackUp.gif"

// swap images (on)
function imgOn(img) {
    if (document.images) {
        // handle NN4 layers that hold images
        if (document.layers) {
            if (img.length == 1) {
                document.ivories.document.images[img].src = onImages[img].src
            } else {
                document.ivories.document.layers["ivory" +
img].document.images[img].src
                    = onImages[img].src
            }
        } else {
            document.images[img].src = onImages[img].src
        }
    }
}
// swap images (off)
function imgOff(img) {
    if (document.images) {
        // handle NN4 layers that hold images
        if (document.layers) {
            if (img.length == 1) {
                    document.ivories.document.images[img].src =
offImages[img].src
            } else {
                document.ivories.document.layers["ivory" +
img].document.images[img].src
                    = offImages[img].src
            }
        } else {
            document.images[img].src = offImages[img].src
        }
    }
}
```

```
// play a note (mousedown)
function playNote(id) {
    players[id].rewind()
    players[id].play(1)
}
// stop playing (mouseup)
function stopNote(id) {
    players[id].stop()
    players[id].rewind()
}
</SCRIPT>
</HEAD>

<BODY onLoad="initAudioAPI(['cNatural','audio/x-aiff'],['cSharp','audio/x-
aiff'],['dNatural','audio/x-aiff'],['dSharp','audio/x-
aiff'],['eNatural','audio/x-aiff'])">
<H1>Playing Multiple Sounds</H1>
<HR>
<TABLE ALIGN="center">
<TR><TD>
<DIV ID="ivories" STYLE="position:relative">
<A HREF="#" onMouseDown="playNote('cNatural');imgOn('c');return false"
            onMouseUp="imgOff('c');stopNote('cNatural')"><IMG
            NAME="c" SRC="whiteUp.gif"
            HEIGHT="140" WIDTH="35" BORDER=0></A><A HREF="#"
            onMouseDown="playNote('dNatural');imgOn('d');return false"
            onMouseUp="imgOff('d');stopNote('dNatural')"><IMG
            NAME="d" SRC="whiteUp.gif"
            HEIGHT="140" WIDTH="35" BORDER=0></A><A HREF="#"
            onMouseDown="playNote('eNatural');imgOn('e');return false"
            onMouseUp="imgOff('e');stopNote('eNatural')"><IMG
            NAME="e" SRC="whiteUp.gif"
            HEIGHT="140" WIDTH="35" BORDER=0></A>
<SPAN ID="ivorycHalf" STYLE="position:absolute; left:22px">
<A HREF="#" onMouseDown="playNote('cSharp');imgOn('cHalf');return false"
            onMouseUp="imgOff('cHalf');stopNote('cSharp')"><IMG
            NAME="cHalf" SRC="blackUp.gif"
            HEIGHT="90" WIDTH="26" BORDER=0></A></SPAN>
<SPAN ID="ivorydHalf" STYLE="position:absolute; left:57px">
<A HREF="#" onMouseDown="playNote('dSharp');imgOn('dHalf');return false"
            onMouseUp="imgOff('dHalf');stopNote('dSharp')"><IMG
            NAME="dHalf" SRC="blackUp.gif"
            HEIGHT="90" WIDTH="26" BORDER=0></A></SPAN>
</DIV>
</TD>
</TR>
</TABLE>
<OBJECT ID="cNatural" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="FileName" VALUE="c.aif">
<PARAM NAME="AutoStart" VALUE="false">
<PARAM NAME="BufferingTime" VALUE="30">
    <EMBED NAME="cNatural" HEIGHT=2 WIDTH=2 SRC="c.aif"
            HIDDEN=TRUE AUTOSTART=FALSE AUTOPLAY=FALSE
              ENABLEJAVASCRIPT=TRUE MASTERSOUND>
    </EMBED>
</OBJECT>

<OBJECT ID="cSharp" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="FileName" VALUE="cSharp.aif">
<PARAM NAME="AutoStart" VALUE="false">
<PARAM NAME="BufferingTime" VALUE="30">
    <EMBED NAME="cSharp" HEIGHT=2 WIDTH=2 SRC="cSharp.aif"
```

```
            HIDDEN=TRUE AUTOSTART=FALSE AUTOPLAY=FALSE
              ENABLEJAVASCRIPT=TRUE MASTERSOUND>
    </EMBED>
</OBJECT>

<OBJECT ID="dNatural" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="FileName" VALUE="d.aif">
<PARAM NAME="AutoStart" VALUE="false">
<PARAM NAME="BufferingTime" VALUE="30">
    <EMBED NAME="dNatural" HEIGHT=2 WIDTH=2 SRC="d.aif"
            HIDDEN=TRUE AUTOSTART=FALSE AUTOPLAY=FALSE
              ENABLEJAVASCRIPT=TRUE MASTERSOUND>
    </EMBED>
</OBJECT>

<OBJECT ID="dSharp" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="FileName" VALUE="dSharp.aif">
<PARAM NAME="AutoStart" VALUE="false">
<PARAM NAME="BufferingTime" VALUE="30">
    <EMBED NAME="dSharp" HEIGHT=2 WIDTH=2 SRC="dSharp.aif"
            HIDDEN=TRUE AUTOSTART=FALSE AUTOPLAY=FALSE
              ENABLEJAVASCRIPT=TRUE MASTERSOUND>
    </EMBED>
</OBJECT>

<OBJECT ID="eNatural" WIDTH="1" HEIGHT="1"
CLASSID="CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95"
CODEBASE="#Version=6,0,0,0">
<PARAM NAME="FileName" VALUE="e.aif">
<PARAM NAME="AutoStart" VALUE="false">
<PARAM NAME="BufferingTime" VALUE="30">
    <EMBED NAME="eNatural" HEIGHT=2 WIDTH=2 SRC="e.aif"
            HIDDEN=TRUE AUTOSTART=FALSE AUTOPLAY=FALSE
              ENABLEJAVASCRIPT=TRUE MASTERSOUND>
    </EMBED>
</OBJECT>
</BODY>
</HTML>
```

Perhaps the trickiest part of this entire demonstration lies in the way the keyboard art and user interface are created. Because the white keys are not rectangular, the black key art is dropped atop the white keys by way of positioned elements (which become layer objects in NN4). The visual reward is worth the extra pain of managing references to the images within NN4 layers.

When you use the page, you may notice a slight delay in getting the sound to be heard after pressing down on a key. On older, slower machines, this delay is even more noticeable. Take this behavior into account when designing interactive sound.

Scripting Java Classes Directly

LiveConnect, as implemented in NN3 and NN4, allows scripts to access Java classes as if they were part of the JavaScript environment. Because you need to know your way around Java before programming Java classes directly from JavaScript, I won’t get into too much detail in this book. Fortunately, the designers of JavaScript have done a good

job of creating JavaScript equivalents for the most common Java language functionality, so there is not a strong need to access Java classes on a daily basis.

To script Java classes, it helps to have a good reference guide to the classes built into Java. Though intended for experienced Java programmers, *Java in a Nutshell* (O'Reilly & Associates, Inc.) offers a condensed view of the classes, their constructors, and their methods.

Java's built-in classes are divided into major groups (called *packages*) to help programmers find the right class and method for any need. Each package focuses on one particular aspect of programming, such as classes for user interface design in application and applet windows, network access, and basic language constructs, such as strings, arrays, and numbers. References to each class (object) defined in Java are "dot" references, just as in JavaScript. Each item following a dot helps zero-in on the desired item. As an example, consider one class that is part of the base language class. The base language class is referred to as

```
java.lang
```

One of the objects defined in `java.lang` is the `String` object, whose full reference is

```
java.lang.String
```

To access one of its methods, you use an invocation syntax with which you are already familiar:

```
java.lang.String.methodName([parameters])
```

To demonstrate accessing Java from JavaScript, I call upon one of Java's `String` object methods, `java.lang.String.equalsIgnoreCase()`, to compare two strings. Equivalent ways are available for accomplishing the same task in JavaScript (for example, comparing both strings in their `toUpperCase()` or `toLowerCase()` versions), so don't look to this Java demonstration for some great new powers along these lines.

Before you can work with data in Java, you have to construct a new object. Of the many ways to construct a new `String` object in Java, you use the one that accepts the actual string as the parameter to the constructor:

```
var mainString = new java.lang.String("TV Guide")
```

At this point, your JavaScript variable, `mainString`, contains a reference to the Java object. From here, you can call this object's Java methods directly:

```
var result = mainString.equalsIgnoreCase("tv Guide")
```

Even from JavaScript, you can use Java classes to create objects that are Java arrays and access them via the same kind of array references (with square brackets) as JavaScript arrays. In a few cases, you can use Java classes to obtain additional information about the user environment, such as the user's IP address (but not e-mail address). The process involves a couple of Java class calls, as follows:

```
var localHost = java.net.InetAddress.getLocalHost()
var IP = localhost.getHostAddress()
```

The more you work with these two languages, the more you see how much Java and JavaScript have in common.