

LiveConnect: Scripting Java Applets and Plug-ins

Netscape groups all the features that enable JavaScript scripts, Java applets, and plug-ins to communicate with each other under one technology umbrella, called LiveConnect. Having three avenues of access to LiveConnect makes it easy to become confused about how LiveConnect works and how to incorporate these powers into your Web site presentations. In this chapter, I focus 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 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

The backbone of the LiveConnect facilities in Navigator is the Java *virtual machine* (VM) you see loading (in the splash screen) during a Navigator 3 launch (in Navigator 4, the VM doesn't load until it is needed, sometimes causing a brief delay in initial execution). This virtual machine (which is entirely software-based) makes your computer look like every other computer that has a Java virtual machine running on it — hence the capability of Java applets (and applications) to run on Wintel, Macintosh, and UNIX computers without requiring modification from platform to platform. The Java virtual machine is not embedded in absolutely every

38

CHAPTER



In This Chapter

Communicating
with Java applets
from scripts

Accessing scripts
and objects from
Java applets

Controlling scriptable
plug-ins



platform, however. Windows 3.1 users are the last to get Java capabilities for their browsers; also, some LiveConnect communication with plug-ins is not available for non-PowerPC Macintoshes.

For the most part, the underlying Java engine is invisible to the scripter (you) and certainly to the visitors of your sites. At most, visitors see status bar messages about applets loading and running. To applet and plug-in makers, however, Java is the common denominator that enables applets to work with other applets or plug-ins to communicate with applets. In these early stages of LiveConnect history, very little of this cross-applet or applet-to-plug-in communication occurs; but I have little doubt that the situation will change, mostly for controlled intranet installations.

From the point of view of a JavaScript author, LiveConnect extends the 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.

Because LiveConnect is a proprietary Netscape technology, not all facilities are available in Internet Explorer. Later versions of Internet Explorer 3 and all versions of Internet Explorer 4 allow scripts to communicate with applets. Internet Explorer 4 has partial support for applet-to-script communication. Neither Internet Explorer 3 nor Internet Explorer 4 supports communication from JavaScript scripts to plug-ins or communication directly accessing Java classes.

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 on-screen 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 starts to go 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 things, 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 like 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 38-1 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 38-1: Stopping and Starting an Applet

```
<HTML>
<HEAD>
<TITLE>A Script That Could Stop a Clock</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
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 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:

```
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 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 38-1 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 wouldn'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 38-1: 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 38-2 from the CD-ROM. A portion of that program listing is shown in the following example.

Listing 38-2: 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 38-2 were 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. It 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 38-3, which takes advantage of that scriptability by providing several HTML form elements as user controls of the clock. The results are shown in Figure 38-2.

Listing 38-3: 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)
}

```

(continued)

Listing 38-3 (*continued*)

```

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. When 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.

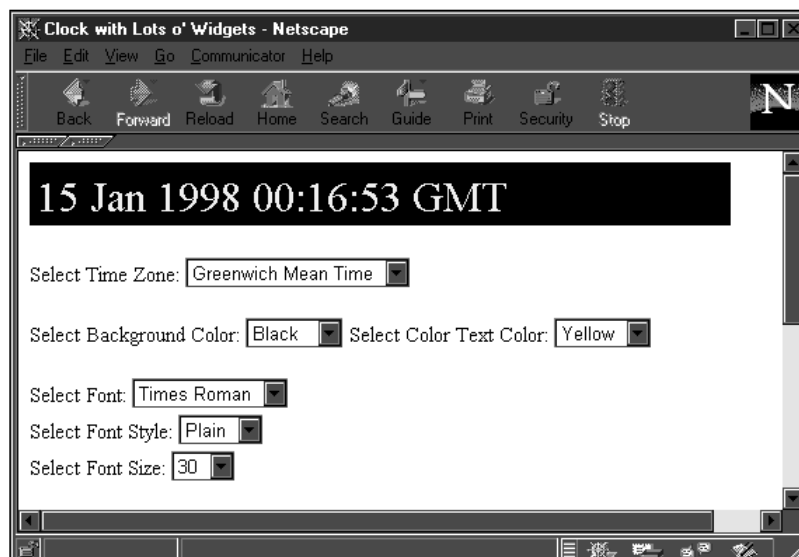


Figure 38-2: Scripting more of the ScriptableClock applet

Applet limitations

Because of concerns about security breaches via LiveConnect, Netscape currently 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 when it starts up and then script the access of that value from JavaScript (as described in the following section).

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 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 38-4 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 38-5. The HTML even automates the loading process by triggering the retrieval of the Java applet's data from an `onLoad=` event handler.

Listing 38-4: 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 could 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 38-5: 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 thing to finish loading in a document, you can't use an applet to generate the page immediately. At best, an HTML document could 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 (or for an entirely new page via `document.write()`). Notice, for instance, that in Listing 38-5, 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. Finally, 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 38-3 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.

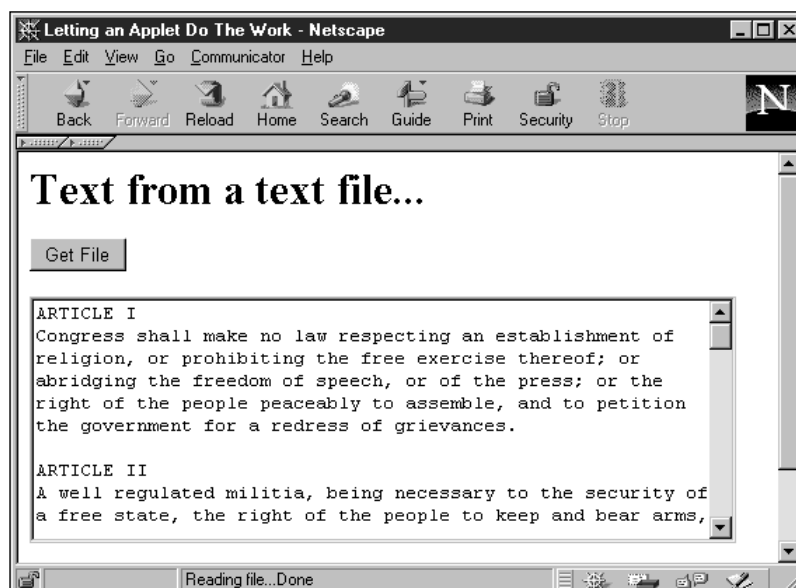


Figure 38-3: The page with text retrieved from a server file

A faceless applet may be one way for Web authors to hide what might 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 visible in the JavaScript code, you can create the array and lookup functionality inside a faceless applet. Then use form elements and JavaScript to act as query entry and output display devices. 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.

You can also pass objects, such as form 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 a page design, you must bear in mind that any calls made from the applet to the page will be 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. 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

Navigators since Versions 3 come with a zipped set of special class files tailored for use in LiveConnect. In Navigator 3, the file is named `java_30` or `java_301`, the latter one being the latest version; in Navigator 4, the file is named `java40.jar`. Microsoft versions of these class files are also included in Internet Explorer 4 (in the `H3rfb7jn.zip` file). The exact location of these files varies with operating system, but for Netscape users they are usually found inside a Java directory within the overall Navigator/Communicator program directory. 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:

```
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 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 java_301 or java40.jar file, depending on which Navigator browser version you, as author, have installed in your system) in the class path for the compiler (if you are using Internet Explorer 4 only, add the H3rfb7jn.zip file to your class path).

Problems frequently occur when the Java compiler you use (perhaps inside an integrated development environment, such as Symantec Cafe) doesn't recognize either of the Netscape files as a legitimate zipped class file. You can make your compilation life simpler if you unzip the class file archive and place the Netscape package in the same directory in which your compiler looks for the basic Java classes. For example, Symantec 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. I then extract the two Netscape.java script 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 JSEException.class files. Then I add the following class path to the project's class path setting:

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

This 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 java_301 and java40.jar 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.java script.JSObject class. This object's methods let the applet contact

document objects and invoke JavaScript statements. Table 38-1 shows the object's methods and one static method.

Table 38-1
JSObject class Methods

<i>Method</i>	<i>Description</i>
<code>call (String functionName, Object args[])</code>	Invokes JavaScript function, argument(s) passed as an array
<code>eval (String expression)</code>	Invokes a JavaScript statement
<code>getMember (String elementName)</code>	Retrieves a named object belonging to a container
<code>getSlot(Int index)</code>	Retrieves indexed object belonging to a container
<code>getWindow (Applet applet)</code>	Static method retrieves applet's containing window
<code>removeMember (String elementName)</code>	Removes a named object belonging to a container
<code>setMember (String elementName, Object value)</code>	Sets value of a named object belonging to a container
<code>setSlot(int index, Object value)</code>	Sets value of an indexed object belonging to a container
<code>toString()</code>	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 phoneId = (JSObject) form.getMember("phone");
```

Another option is to use the `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 phoneId = mainwin.eval("document.entryForm.phone");
```

Once you have a reference to an object, you can access its properties via the `getMember()` method, as follows:

```
String phoneNum = (String) phoneId.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. Since 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 38-4). 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.

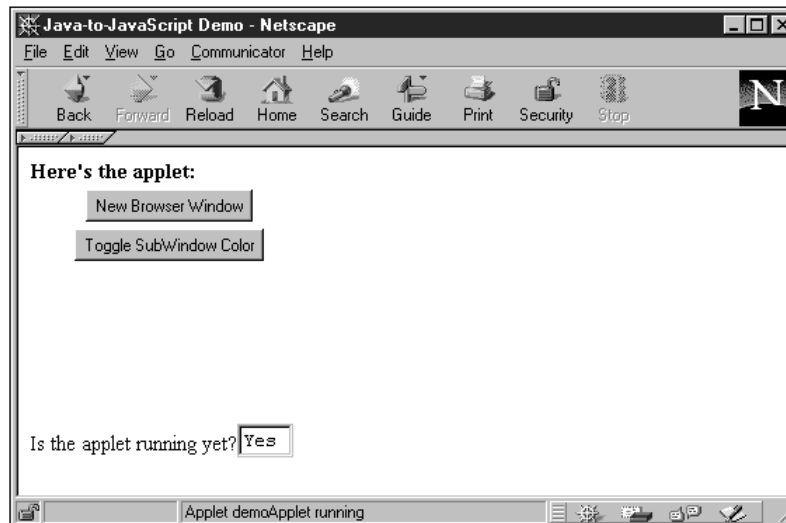


Figure 38-4: The applet displays two buttons seamlessly on the page.

Listing 38-6 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 38-6: 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 will be 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 would appear in the page's JavaScript to open a new window. The `window.open()` method returns a reference to that subwindow, so 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. Unfortunately, Internet Explorer 4's 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.

(continued)

Listing 38-6 (*continued*)

```

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 38-7. 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 38-7: 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 Navigator Plug-ins

Controlling a Navigator plug-in from JavaScript is much like controlling a Java applet. In fact, behind the scenes, the Java virtual machine provides the wires connecting JavaScript to plug-ins (although scripting plug-ins is unreliable on the Windows 3.1 or Macintosh 68K platforms). The primary difference between an applet and a plug-in is that a plug-in is initialized from an HTML document via the `<EMBED>` tag, rather than the `<APPLET>` tag.

Before your page loads the foreign MIME type that gets the plug-in functioning, your scripts should make sure that the desired plug-in is installed and set up to accommodate the specific MIME type (if the plug-in handles multiple MIME types). Chapter 25 provides a few examples of how to use JavaScript to verify that a particular MIME type is supported by the desired plug-in on a client's computer.

Once the plug-in (and usually one of its files) is loaded into memory, your scripts can take over, provided that the plug-in is written to be scripted. LiveAudio, for example, is the multiple audio format plug-in that Netscape ships with Navigator 3 and later. LiveAudio is scriptable. Even so, you must know the commands available for a plug-in before you can script it. Third-party scriptable plug-ins should have electronic documentation somewhere to help you script it.

Scripting LiveAudio

Because the LiveAudio plug-in ships with most platforms of Navigator, I want to spend a little time demonstrating how you script it. For up-to-date scripting and relevant `<EMBED>` tag attribute information, consult Netscape's online JavaScript documentation at <http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html>.

Table 38-2 shows the most important LiveAudio methods that you can access from your scripts. The last four methods in the table return values regarding the current state of the plug-in.

Table 38-2
Popular LiveAudio Plug-in Commands

<i>Method</i>	<i>Description</i>
<code>play('TRUE FALSE LoopCount'[, 'URL '])</code>	Plays the currently loaded sound, unless the URL parameter is provided
<code>stop()</code>	Stops the currently playing sound
<code>pause()</code>	Pauses or restarts the currently playing sound
<code>start_time(seconds)</code>	Sets starting time within sound
<code>end_time(seconds)</code>	Sets ending time within sound
<code>start_at_beginning()</code>	Overrides existing <code>start_time()</code> setting
<code>stop_at_end()</code>	Overrides existing <code>end_time()</code> setting
<code>setvol(percent)</code>	Sets volume percentage (0-100)
<code>fade_to(percent)</code>	Gradually adjusts volume-to-volume level
<code>fade_from_to(startPercent, endPercent)</code>	Gradually adjusts volume between two fixed levels
<code>IsReady()</code>	Returns Boolean of plug-in sound state
<code>IsPlaying()</code>	Returns Boolean of playing state
<code>IsPaused()</code>	Returns Boolean of paused state
<code>GetVolume()</code>	Returns integer (0-100) of volume level percentage

To get a plug-in loaded and ready for action, specify an initial sound file for the SRC attribute of the `<EMBED>` tag. By default, a LiveAudio sound does not automatically play when it loads (although you can override that with the `AUTOSTART` attribute inside the tag). You can use scripting to send a different sound file to the loaded plug-in anytime while the page is loaded in the browser.

LiveAudio at work

Listing 38-8 is a sample application showing how to create a kind of jukebox for selecting and controlling multiple sound files with a single `<EMBED>` tag and scripted calls to the LiveAudio plug-in. Audio files for the example are on the CD-ROM.

Listing 38-8: A Scripted Jukebox

```
<HTML>
<HEAD>
<TITLE>Oldies but Goody's</TITLE>
<SCRIPT>
```

```

function showEmbedProps() {
    var obj = document.theme
    var msg = ""
    for (var i in obj) {
        msg += "Embed." + i + "=" + obj[i] + "\n"
    }
    alert(msg)
}
function playIt() {
    var args = playIt.arguments
    var sndFile = ""
    var snd = document.jukebox
    if (args.length == 1) {
        sndFile = args[0][args[0].selectedIndex].value
    }
    var howMany =
document.forms[0].frequency[document.forms[0].frequency.selectedIndex].
value
    howMany = (howMany != "TRUE") ? parseInt(howMany) : true
    if (!snd.IsReady()) {
        alert ("Sorry, still loading the sound. Try again in a few
seconds.")
        return
    }
    if (sndFile != "") {
        snd.play(howMany,sndFile)
        return
    }
    snd.play(howMany)
}
function stopIt() {
    document.jukebox.stop()
}
function pauseIt(){
    document.jukebox.pause()
}
function rewindIt() {
    document.jukebox.stop()
    document.jukebox.start_at_beginning()
    playIt()
}
function raiseVol() {
    var newLevel = Math.min(currLevel()+10,100)
    setVolumeLevel(newLevel)
}
function lowerVol() {
    var newLevel = Math.max(currLevel()-10,0)
    setVolumeLevel(newLevel)
}
function setVolume(entry) {
    var newLevel = parseInt(entry.value)
    setVolumeLevel(newLevel)
}

```

(continued)

Listing 38-8 (continued)

```

function setVolumeLevel(newLevel) {
    var snd = document.jukebox
    snd.setvol(newLevel)
    displayVol()
}
function currLevel() {
    return document.jukebox.GetVolume()
}
function displayVol() {
    document.forms[0].volume.value = currLevel()
}
</SCRIPT>
</HEAD>

<BODY onLoad="displayVol()">
<FORM>
<TABLE BORDER=2><CAPTION ALIGN=top><FONT SIZE=+3>Classical Piano
Jukebox
</FONT></CAPTION>
<TR><TD COLSPAN=2 ALIGN=center>
<SELECT NAME="musicChoice" onChange="playIt(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="playIt()">
<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="stopIt()"></TD></TR>
<TR><TD><INPUT TYPE="button" VALUE="Pause"
onClick="pauseIt()"></TD></TR>
<TR><TD><INPUT TYPE="button" VALUE="Rewind" onClick="rewindIt()"></TD>
</TR>
<TR><TH ROWSPAN=3>Volume:</TH>
<TD>Current Setting:<INPUT TYPE="text" SIZE=3 NAME="volume"
onChange="setVolume(this)"></TD></TR>
<TR><TD><INPUT TYPE="button" VALUE="Higher" onClick="raiseVol()"></TD>
</TR>
<TR><TD><INPUT TYPE="button" VALUE="Lower"
onClick="lowerVol()"></TD></TR>
</TABLE>
</FORM>
<EMBED NAME="jukebox" SRC="Beethoven.aif" HIDDEN=TRUE AUTOSTART=FALSE
MASTERSOUND>
</BODY>
</HTML>

```

For this demonstration (shown in Figure 38-5), the `<EMBED>` tag initially loads one of three audio files. Be sure to include the `MASTERSOUND` attribute if you want to script the plug-in.

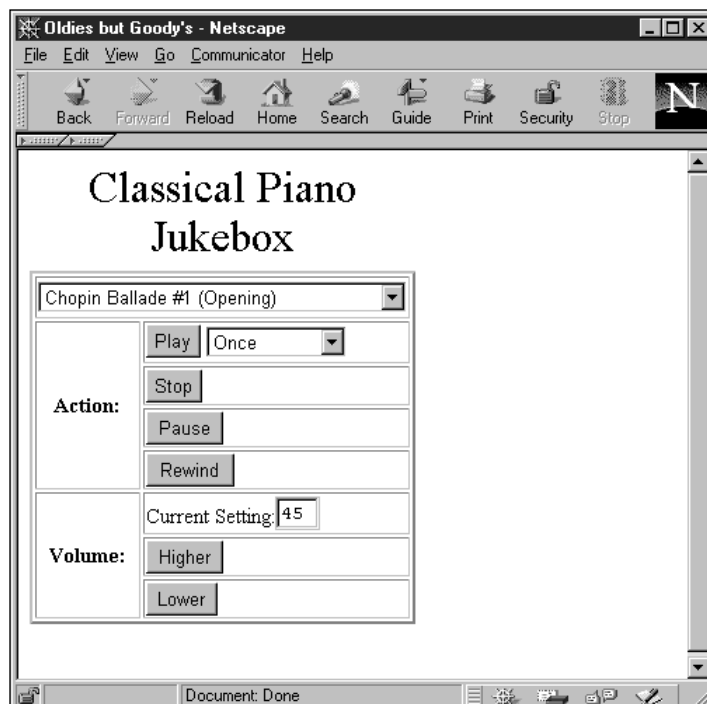


Figure 38-5: The jukebox page

As the user chooses an item from the pop-up listing at the top of the table, the corresponding sound file is loaded and played (there may be a delay if the sound is not cached in the browser). Although the example uses local source files, any valid URL that points to an audio file whose MIME type is readable by LiveAudio will work. Additional choices in this application enable the user to set how many times the selection should play (once, twice, three times, or in a continuous loop). Another group of settings displays and controls the volume level for the plug-in. The scripting is designed to adjust the volume in increments of 10 percent, but on some machines, the default displays may not be so conveniently rendered. Even so, you can manually set the volume to any value between 0 and 100 in the field.

If you decide to run Listing 38-8, you may experience some bugs in LiveConnect's capability to work with LiveAudio on some operating system platforms. I've experienced difficulties with Navigator 3 for Macintosh, especially on 68K Macs (such as those with 68030 and 68040 microprocessors).

Notice that the first function in Listing 38-8 is a property browser for the plug-in object. Although this function is not connected to a button in the listing, feel free to add a button that calls the `showEmbedProps()` function. The results are similar to the kind of listing you saw earlier in this chapter for Java applets. You see a

listing of all methods available to the scripter, plus some Java-level methods that are not usually scripted. As with this kind of exposure to Java methods, you cannot tell from such listings what kind of parameters are required to script the methods. You still need outside documentation.

Scripting Java Classes Directly

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 like 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 will 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. The more you work with these two languages, the more you will see they have in common.

