WEEK 2

DAY 14

Writing Java Applets and Java Web Start Applications

The first exposure of many people to the Java programming language is in the form of *applets*, small and secure Java programs that run as part of a World Wide Web page.

Though you can do similar things with Macromedia Flash and other technology, Java remains an effective choice for Web-based programming.

Java Web Start, a protocol for downloading and running Java programs, makes it possible to run applications from a Web browser as if they were applets.

Today, you will learn how to create both kinds of Web-based Java programming as you explore the following topics:

- How to create a simple applet and present it on a Web page
- How to send information from a Web page to an applet

- How to store an applet in a Java archive so that it can be downloaded more quickly by Web browsers
- How to create applets run by the Java Plug-in, a virtual machine that improves a Web browser's Java support
- How to install and run Java applications in a Web browser
- How to publish your application's files and run it

How Applets and Applications Are Different

The difference between Java applets and applications lies in how they are run.

Applications are usually run by loading the application's main class file with a Java interpreter, such as the `java` tool in the Java 2 SDK.

Applets, on the other hand, are run on any browser that can support Java, such as Mozilla, Internet Explorer, and Opera. Applets also can be tested by using the SDK's `appletviewer` tool.

For an applet to run, it must be included on a Web page using HTML tags. When a user with a Java-capable browser loads a Web page that includes an applet, the browser downloads the applet from a Web server and runs it on the Web user's own system using a Java interpreter.

For many years, browsers used their own built-in Java interpreters to run applets. Because these interpreters haven't been kept current with new versions of the language, Sun offers the Java Plug-in, an interpreter that can be configured to run applets with each of the popular browsers.

Like an application, a Java applet includes a class file and any other helper classes that are needed to run the program. The Java 2 class library is included automatically.

Applet Security Restrictions

Because Java applets are run on a Web user's system when loaded by a browser, they are subject to stringent security restrictions:

- They cannot read or write files on the user's file system.
- They cannot communicate with an Internet site other than the one that served the applet's Web page.
- They cannot load or run programs stored on the user's system, such as executable programs and shared libraries.

Java applications have none of these restrictions. They can take full advantage of Java's capabilities.

These restrictive security rules apply to all applets run by the Java Plug-in or a browser's built-in interpreter. There is one way to get around this with the plug-in—an applet that has been digitally signed to verify the identity of the author can be granted the same access as Java applications. If a Web user accepts a signed applet's security certificate, the applet can run without restriction.

CAUTION

Although Java's security model makes it difficult for a malicious applet to do harm to a user's system, it will never be 100 percent secure. Search Google or another Web search engine for "hostile applets", and you'll find discussion of security issues in different versions of Java and how they have been addressed.

Choosing a Java Version

A Java programmer who writes applets must decide which Java version to employ.

Some programmers have stuck with Java 1.1, the most up-to-date version of the language supported by the built-in Java interpreter available for Internet Explorer and some versions of Netscape Navigator.

However, this strategy has become less effective because Microsoft has stopped including the Java interpreter with Internet Explorer as part of an ongoing legal matter involving Sun Microsystems.

To provide a way for applet programmers to use current versions of Java, Sun offers the Java Plug-in, which must be downloaded and installed by browser users unless it was included with their operating system.

Java 2 has been designed so that a program using only Java 1.1 features can usually compile and run successfully on a Java 1.1 interpreter or 1.1-capable browser.

If an applet uses any feature introduced with Java 2, the program only will run successfully on a browser has been equipped with the Java Plug-in. The only test environment that always supports the most current version of Java is the latest `appletviewer` from the corresponding SDK.

This situation is a common source of errors for Java applet programmers. If you write a Java 2 applet and run it on a browser without the plug-in, you will get security errors, class-not-found errors, and other problems that prevent the applet from running.

NOTE

In this book, Java 2 techniques are used for all programs, even applets. Techniques to write applets for older versions are covered in Appendix E, "Writing Java 1.1 Applets."

Creating Applets

The Java programs you've created up to this point have been applications.

Applets differ significantly from applications. First, applets do not have a `main()` method that automatically is called to begin the program. Instead, several methods are called at different points in the execution of an applet.

Applets are subclasses of the `JApplet` class in the `javax.swing` package and are graphical user interface components similar to frames. They can contain other components and have layout managers applied to them.

By inheriting from `JApplet`, an applet runs as part of a Web browser and can respond to events such as the browser page being reloaded. It also can take input from users.

All applets are created by subclassing `JApplet`:

```
public class AppletName extends javax.swing.JApplet {
    // Applet code here
}
```

Applets must be declared `public`.

When a Java-equipped browser finds a Java applet on a Web page, the applet's class is loaded along with any other helper classes used by the applet. The browser automatically creates an instance of the applet's class and calls methods of the `JApplet` class when specific events take place.

Different applets that use the same class use different instances, so you could place more than one copy of the same type of applet on a page.

Applet Methods

Applets have methods called when specific things occur as the applet runs.

One example is the `paint()` method, which is called whenever the applet window needs to be displayed or redisplayed.

The `paint()` method is similar to the `paintComponent()` method you worked with yesterday on Day 13, "Using Color, Fonts, and Graphics." For text and graphics to be displayed on the applet window, the `paint()` method must be overridden with behavior to display something.

The following sections describe events that may occur as an applet runs: initialization, starting, stopping, destruction, and painting.

Initialization

Initialization occurs the first time an applet is loaded. It can be used to create objects the applet needs, load graphics or fonts, and the like.

To provide behavior for the initialization of an applet, you override the `init()` method in the following manner:

```
public void init() {
    // Code here
}
```

One useful thing to do when initializing an applet is to set the color of its background window using a `Color` object, as in the following example:

```
Color avocado = new Color(102, 153, 102);
setBackground(avocado)
```

The preceding statements changes the applet window to avocado green when placed in an `init()` method.

Starting

An applet is started after it is initialized and when the applet is restarted after being stopped. Starting can occur several times during an applet's life cycle, but initialization happens only once.

An applet restarts when a Web user returns to a page containing the applet. If a user clicks the Back button to return to the applet's page, it starts again.

To provide starting behavior, override the `start()` method as follows:

```
public void start() {
    // Code here
}
```

Functionality that you put in the `start()` method might include starting a thread to control the applet and calling methods of other objects that it uses.

Stopping

An applet stops when the user leaves a Web page that contains a running applet or when the `stop()` method is called.

By default, any threads an applet has started continue running even after a user leaves the applet's page. By overriding `stop()`, you can suspend execution of these threads and restart them if the applet is viewed again. A `stop()` method takes this form:

```
public void stop() {
    // Code here
}
```

Destruction

Destruction is the opposite of initialization. An applet's `destroy()` method enables it to clean up after itself before it is freed from memory or the browser exits.

You can use this method to kill any running threads or to release any other running objects. Generally, you won't want to override `destroy()` unless you have specific resources that need to be released, such as threads that the applet has created. To provide cleanup behavior for your applet, override the `destroy()` method as follows:

```
public void destroy() {
    // Code here
}
```

Because Java handles the removal of objects automatically when they are no longer needed, there's normally no need to use the `destroy()` method in an applet.

Painting

Painting is how an applet displays text and graphics in its window.

The `paint()` method is called automatically by the environment that contains the applet—normally a Web browser—whenever the applet window must be redrawn. It can occur hundreds of times: once at initialization, again if the browser window is brought out from behind another window or moved to a different position, and so on.

You must override the `paint()` method of your `JApplet` subclass to display anything. The method takes the following form:

```
public void paint(Graphics screen) {
    Graphics2D screen2D = (Graphics2D)screen;
    // Code here
}
```

Unlike other applet methods, `paint()` takes an argument: a `Graphics` object that represents the area in which the applet window is displayed.

Like yesterday with the `paintComponent()` method, a `Graphics2D` object is cast from this `Graphics` object and used for all text and graphics drawn in the applet window using Java2D.

The `Graphics` and `Graphics2D` classes are part of the `java.awt` package.

There are times in an applet when you do something that requires the window to be repainted. For example, if you change the applet's background color, it won't be shown until the applet window is redrawn.

To request that the window be redrawn in an applet, call the applet's `repaint()` method without any arguments:

```
repaint();
```

Writing an Applet

Today's first project is the `Watch` applet, which displays the current date and time and updates the information roughly once a second.

This project uses objects of several classes:

- `GregorianCalendar`—A class in the `java.util` package that represents date/time values in the Gregorian calendar system, which is in use throughout the Western world
- `Font`—A `java.awt` class that represents the size, style, and family of a display font
- `Color` and `Graphics2D`—Two `java.awt` classes described in the previous section

Listing 14.1 shows the source code for the applet.

LISTING 14.1 The Full Text of `Watch.java`

```
 1: import java.awt.*;
 2: import java.util.*;
 3:
 4: public class Watch extends javax.swing.JApplet {
 5:     private Color butterscotch = new Color(255, 204, 102);
 6:     private String lastTime = "";
 7:
 8:     public void init() {
 9:         setBackground(Color.black);
10:     }
11:
12:     public void paint(Graphics screen) {
13:         Graphics2D screen2D = (Graphics2D)screen;
14:         Font type = new Font("Monospaced", Font.BOLD, 20);
15:         screen2D.setFont(type);
16:         GregorianCalendar day = new GregorianCalendar();
17:         String time = day.getTime().toString();
18:         screen2D.setColor(Color.black);
19:         screen2D.drawString(lastTime, 5, 25);
20:         screen2D.setColor(butterscotch);
21:         screen2D.drawString(time, 5, 25);
22:         try {
23:             Thread.sleep(1000);
24:         } catch (InterruptedException e) {
25:             // do nothing
26:         }
```

LISTING 14.1 continued

```
27:         lastTime = time;
28:         repaint();
29:     }
30: }
```

After you create this program, you can compile it, but you won't be able to try it out yet. The applet overrides the `init()` method to set the background color of the applet window to black.

The `paint()` method is where this applet's real work occurs. The `Graphics` object passed into the `paint()` method holds the graphics state, which keeps track of the current attributes of the drawing surface. The state includes details about the current font and color to use for any drawing operation, for example. By using casting in line 13, a `Graphics2D` object is created that contains all this information.

Lines 14–15 set up the font for this graphics state. The `Font` object is held in the `type` instance variable and set up as a bold, monospaced, 20-point font. The call to `setFont()` establishes this font as the one that will be used for subsequent drawing operations.

Lines 16–17 create a new `GregorianCalendar` object that holds the current date and time. The `getTime()` method of this object returns the date and time as a `Date` object, another class of the `java.util` package. Calling `toString()` on this object returns the date and time as a string you can display.

Lines 18–19 set the color for drawing operations to black and then calls `drawString()` to display the string `lastTime` in the applet window at the x,y position 5, 25. Because the background is black, nothing appears. You'll see soon why this is done.

Lines 20–21 set the color using a `Color` object called `butterscotch` and then display the string `time` using this color.

Lines 22–26 use a class method of the `Thread` class to make the program do nothing for 1,000 milliseconds (one second). Because the `sleep()` method generates an `InterruptedException` error if anything occurs that should interrupt this delay, the call to `sleep()` must be enclosed in a `try-catch` block.

Lines 27–28 make the `lastTime` variable refer to the same string as the `time` variable and then call `repaint()` to request that the applet window be redrawn.

Calling `repaint()` causes the applet's `paint()` method to be called again. When this occurs, `lastTime` is displayed in black text, overwriting the last `time` string displayed. This clears the screen so that the new value of `time` can be shown.

CAUTION

Calling `repaint()` within an applet's `paint()` method is not the ideal way to handle animation; it's suitable here primarily because the applet is a simple one. A better technique is to use threads and devote a thread to the task of animation.

Including an Applet on a Web Page

After you create the class or classes that compose your applet and compile them into class files, you must create a Web page on which to place the applet.

Applets are placed on a page by using `APPLET`, an HTML markup tag used like other Web page elements. Numerous Web-page development tools, such as Microsoft FrontPage 2003 and Macromedia Dreamweaver, also can be used to add applets to a page without using HTML.

The `APPLET` tag places an applet on a Web page and controls how it looks in relation to other parts of the page.

Java-capable browsers use the information contained in the tag to find and execute the applet's class file.

NOTE

The following section assumes that you have at least a passing understanding of HTML or know how to use a Web development tool to create Web pages. If you need help in this area, one of the co-authors of this book, Laura Lemay, has written *Sams Teach Yourself Web Publishing with HTML and XHTML in 21 Days* with Rafe Colburn (ISBN 0-672-32519-5).

The `APPLET` Tag

In its simplest form, the `APPLET` tag uses `CODE`, `WIDTH`, and `HEIGHT` attributes to create a rectangle of the appropriate size and then loads and runs the applet in that space. The tag also includes several other attributes that can help you better integrate an applet into a Web page's overall design.

NOTE

The attributes available for the `APPLET` tag are almost identical to those for the `IMG` tag, which is used to display graphics on a Web page.

Listing 14.2 contains the HTML markup for a Web page that includes the `Watch` applet.

LISTING 14.2 The Full Text of `Watch.html`

```
 1: <html>
 2: <head>
 3: <title>Watch Applet</title>
 4: </head>
 5: <body>
 6: <applet code="Watch.class" height="50" width="345">
 7: This program requires a Java-enabled browser.
 8: </applet>
 9: </body>
10: </html>
```

HTML tags are not case-sensitive, so `<applet>` is the same as `<APPLET>`.

In Listing 14.2, the `APPLET` tag is contained in lines 6–8 and includes three attributes:

- `CODE`—The name of the applet's main class file
- `WIDTH`—The width of the applet window on the Web page
- `HEIGHT`—The height of the applet window

The class file indicated by the `CODE` attribute must be in the same folder as the Web page containing the applet unless you use a `CODEBASE` attribute to specify a different folder. You will learn how to do that later today.

`WIDTH` and `HEIGHT` are required attributes because the Web browser needs to know how much space to devote to the applet on the page.

The text in Line 7 of Listing 14.2 only will be displayed on a Web browser that doesn't support Java programs. Text, graphics, and other Web page elements can be included between the opening `<APPLET>` tag and the closing `</APPLET>` tag. If you don't specify anything between the tags, browsers that don't support Java display nothing in place of the applet.

TIP

The Java Plug-in can be downloaded and installed from the Java Web site at the address `http://www.java.com`. For the courtesy of people who view your Web page without a Java-enabled browser, you can include a link to this site within the `APPLET` tag using this HTML markup:

```
This applet requires the <a href="http://www.java.com">Java Plug-in</a>.
```

Other Attributes

The APPLET tag supports additional attributes that can be used to customize the presentation of the applet.

The ALIGN attribute defines how the applet will be laid out on a Web page in relation to other parts of the page. This attribute can have several different values. The most useful are "Left" to present the applet on the left of adjacent text and graphics, "Right" on the right, and "Top" to align it with the topmost edge of adjacent items.

If you are using a Web-development tool that enables you to place Java applets on a page, you should be able to set the ALIGN attribute by choosing "Left", "Right", or one of the other values from within the program.

The HSPACE and VSPACE attributes set the amount of space, in pixels, between an applet and its surrounding text. HSPACE controls the horizontal space to the left and right of the applet, and VSPACE controls the vertical space above and below the applet. For example, here's the HTML markup for an applet with vertical space of 50 and horizontal space of 10:

```
<applet code="ShowSmiley.class" width="45" height="42" align="Left"
vspace="50" hspace="10">
    This applet requires Java.
</applet>
```

The CODE and CODEBASE attributes indicate where the applet's main class file and other files can be found.

CODE indicates the filename of the applet's main class file. If CODE is used without an accompanying CODEBASE attribute, the class file will be loaded from the same folder as the Web page containing the applet.

You must specify the .class file extension with the CODE attribute. The following example loads an applet called Bix.class from the same folder as the Web page:

```
<applet code="Bix.class" height="40" width="400">
</applet>
```

The CODEBASE attribute indicates the folder where the applet's class is stored. The following markup loads a class called Bix.class from a folder called Torshire:

```
<applet code="Bix.class" codebase="Torshire"
height="40" width="400">
</applet>
```

Loading an Applet

After you have an applet's class file and a Web page that includes the applet, you can run the applet by loading the page with a Web browser.

Open the `Watch.html` page created from Listing 14.2 in a Web browser. One of three things may happen:

- If the browser is equipped with the Java Plug-in, the applet will be loaded and will begin running.
- If the browser does not offer any Java support, the following text will be displayed in place of the applet: "This program requires a Java-enabled browser."
- If the browser is not equipped with the Java Plug-in, but it does have its own built-in Java interpreter, the applet will not be loaded. An empty gray box will be displayed in its place.

If you installed the Java 2 SDK, it's likely you saw the applet running. The Java Plug-in can be installed along with the SDK and configured to replace the built-in Java interpreter in Internet Explorer and other Web browsers.

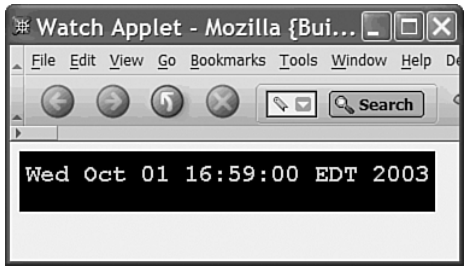
TIP

If you are using the SDK, you also can use the `appletviewer` tool to view applets. Unlike a browser, `appletviewer` displays only the applets included on a Web page. It does not display the Web page itself.

Figure 14.1 shows the `Watch.html` page loaded with a copy of Mozilla 1.1 that has been equipped with the Java Plug-in.

FIGURE 14.1

Running an applet on a Web page with the Java Plug-in.



Try to load this Web page with each of the browsers installed on your computer. To try it with the SDK's `appletviewer`, use the following command:

```
appletviewer Watch.html
```

If you can't get the applet to load in a Web browser, but you can load it with the SDK's `appletviewer` tool, the likeliest reason is because the browser isn't equipped with the Java 2 Plug-in yet.

This is a circumstance that will be faced by many of the people using your applet. They must download and install the Java Plug-in before they can view any Java 2 applets, such as `Watch`, in their browser.

Putting Applets on the Web

After you have an applet that works successfully when you test it on your computer, you can make the applet available on the Web.

If you know how to publish Web sites, you don't have to learn any new skills to publish Java applets on a site.

Java applets are presented by a Web server in the same way as Web pages, graphics, and other files. You store the applet in a folder accessible to the Web server—often the same folder that contains the Web page that features the applet.

When you upload an applet to a Web server, make sure to include each of the following files:

- The Web page where the applet is presented
- The applet's main class file
- Any other class files required by the applet—with the exception of the Java 2 class library
- Any graphics and other files used by the applet

The most common ways to publish on the Web are by sending files through FTP (File Transfer Protocol) or Web-design software that can publish sites.

Java Archives

The primary way to place a Java applet on a Web page is to use the `APPLET` tag to indicate the applet's class file. A Java-enabled browser downloads and runs the applet, loading any classes and any other files needed by the applet from the same Web server.

Every file an applet needs requires a separate connection from a Web browser to the server containing the file. Because a fair amount of time is needed just to make the connection, this can increase the amount of time it takes to download an applet and everything it needs to run.

The solution to this problem is to package the applet in a Java archive, which is also called a JAR file. A *Java archive* is a collection of Java classes and other files packaged into a single file.

When an archive is used, a Web browser makes only one connection to download the applet and its associated files and can start running more quickly.

The SDK includes a tool called `jar` that can add and remove files in Java archives. JAR files can be compressed using the Zip format or packed without using compression.

Run `jar` without any arguments to see a list of options that can be used with the program. The following command packs all of a folder's class and GIF graphics files into a single Java archive called `Animate.jar`:

```
jar cf Animate.jar *.class *.gif
```

The argument `cf` specifies two command-line options that can be used when running the `jar` program. The `c` option indicates that a Java archive file should be created, and `f` indicates that the name of the archive file will follow as one of the next arguments.

You also can add specific files to a Java archive with a command, such as

```
jar cf AudioLoop.jar AudioLoop.class beep.au loop.au
```

This creates an `AudioLoop.jar` archive containing three files: `AudioLoop.class`, `loop.au`, and `beep.au`.

In an `APPLET` tag, the `ARCHIVE` attribute shows where the archive can be found, as in the following example:

```
<applet code="AudioLoop.class" archive="AudioLoop.jar"
width="45" height="42">
</applet>
```

This tag specifies that an archive called `AudioLoop.jar` contains files used by the applet. Browsers and browsing tools that support JAR files will look inside the archive for files that are needed as the applet runs.

CAUTION

Although a Java archive can contain class files, the `ARCHIVE` attribute does not remove the need for the `CODE` attribute. A browser still needs to know the name of the applet's main class file to load it.

Passing Parameters to Applets

Java applications support the use of command-line arguments, which are stored in a String array when the `main()` method is called to begin execution of the class.

Applets offer a similar feature: They can read parameters that are set up with the HTML tag `PARAM`, which has `NAME` and `VALUE` attributes.

In the Web page that contains the applet, one or more parameters can be specified with this tag. Each one must be placed within the opening and closing APPLET tags, as in the following example:

```
<applet code="QueenMab.class" width="100" height="100">
    <param name="font" value="TimesRoman">
    <param name="size" value="24">
    This applet requires <a href="http://www.java.com">Java</a>.
</applet>
```

This example defines two parameters to the QueenMab applet: font with a value of "TimesRoman" and size with a value of "24".

Parameters are passed to an applet when it is loaded. In the init() method for your applet, you can retrieve a parameter by calling the getParameter(*String*) method with its name as the only argument. This method returns a string containing the value of that parameter or null if no parameter of that name exists.

For example, the following statement retrieves the value of the font parameter from a Web page:

```
String fontName = getParameter("font");
```

Parameter names are case-sensitive, so you must capitalize them exactly as they appear in an applet's PARAM tag.

Listing 14.3 contains a modified version of the Watch applet that enables the background color to be specified as a parameter called background.

LISTING 14.3 The Full Text of NewWatch.java

```
 1: import java.awt.*;
 2: import java.util.*;
 3:
 4: public class NewWatch extends javax.swing.JApplet {
 5:     private Color butterscotch = new Color(255, 204, 102);
 6:     private String lastTime = "";
 7:     Color back;
 8:
 9:     public void init() {
10:         String in = getParameter("background");
11:         back = Color.black;
12:         if (in != null) {
13:             try {
14:                 back = Color.decode(in);
15:             } catch (NumberFormatException e) {
16:                 showStatus("Bad parameter " + in);
17:             }
18:         }
```

LISTING 14.3 continued

```
19:         setBackground(back);
20:     }
21:
22:     public void paint(Graphics screen) {
23:         Graphics2D screen2D = (Graphics2D)screen;
24:         Font type = new Font("Monospaced", Font.BOLD, 20);
25:         screen2D.setFont(type);
26:         GregorianCalendar day = new GregorianCalendar();
27:         String time = day.getTime().toString();
28:         screen2D.setColor(back);
29:         screen2D.drawString(lastTime, 5, 25);
30:         screen2D.setColor(butterscotch);
31:         screen2D.drawString(time, 5, 25);
32:         try {
33:             Thread.sleep(1000);
34:         } catch (InterruptedException e) {
35:             // do nothing
36:         }
37:         lastTime = time;
38:         repaint();
39:     }
40: }
```

The `init()` method in lines 9–20 has been rewritten to work with a parameter named `"background"`.

This parameter should be specified as a hexadecimal string—a pound character (“#”) followed by three hexadecimal numbers that represent the red, green, and blue values of a color. Black is #000000, red is #FF0000, green is #00FF00, blue is #0000FF, white is #FFFFFF, and so on. If you are familiar with HTML, you have probably used hexadecimal strings before.

The `Color` class has a `decode(`*`String`*`)` class method that creates a `Color` object from a hexadecimal string. This is called in line 14; the `try-catch` block handles the `NumberFormatException` error that occurs if `in` does not contain a valid hexadecimal string.

If no `background` parameter was specified, the default is black.

Line 19 sets the applet window to the color represented by the `back` object. To try this program, create the HTML document in Listing 14.4.

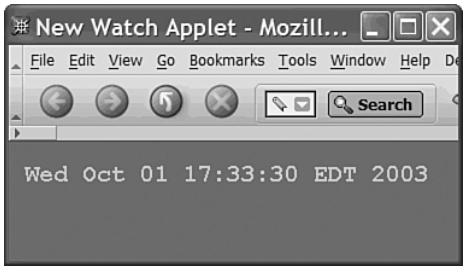
LISTING 14.4 The Full Text of `NewWatch.html`

```
 1: <html>
 2: <head>
 3: <title>New Watch Applet</title>
 4: </head>
 5: <body bgcolor="#996633">
 6: <applet code="NewWatch.class" height="50" width="345">
 7:   <param name="background" value="#996633">
 8: This program requires <a href="http://www.java.com">Java</a>.
 9: </applet>
10: </body>
11: </html>
```

On this page, the `"background"` parameter is specified on line 7 with the value `#996633`. This string value is the hexadecimal color value for a shade of brown. In Listing 14.4, line 5 of the applet sets the background color of the page using the same hexadecimal color value.

Loading this HTML file produces the result shown in Figure 14.2.

FIGURE 14.2
Viewing the `NewWatch.html` *page in a browser.*



Because the applet window and Web page have the same background color, the edges of the applet are not visible in Figure 14.2.

Sun's HTML Converter

At this point, you have learned how to use the `APPLET` tag to present applets. Two other tags also can be used by some Web browsers: `OBJECT` and `EMBED`.

Creating a Web page that supports all these options is difficult even for experienced Web developers.

To make the process easier, Sun offers a Java application called HTMLConverter that converts an existing Web page so that all its applets are run by the Java Plug-in.

This application is included with the Java 2 SDK and can be run at a command line.

To use the converter, first create a Web page that loads an applet using an `APPLET` tag. HTMLConverter can load this page and convert its HTML to use the Java Plug-in.

After you have created a page, run HTMLConverter with the name of the page as an argument. For example:

```
HTMLConverter Watch3.html
```

This command converts all applets contained in `Watch3.html` to be run by the Java Plug-in.

CAUTION

HTMLConverter overwrites the existing HTML markup on a Web page. If you also want the non-Java Plug-in version of the page, you should run HTMLConverter on a copy of the page instead of the original.

Java Web Start

One of the issues you must deal with as a Java programmer is how to make your software available to your users.

Java applications require a Java interpreter, so one must either be included with the application or previously installed on the computer. Lacking either of those, users must install an interpreter themselves. The easiest solution (for you) is to require that users download and install the Java Runtime Environment from Sun's Web site at `http://www.java.com`.

Regardless of how you deal with the requirement for an interpreter, you distribute an application like any other program, making it available on a CD, Web site, or some other means. A user must run an installation program to set it up, if one is available, or copy the files and folders manually.

Applets are easier to make available because they can be run by Web browsers. However, if your program is a Java 2 applet, users must be running browsers equipped with the Java Plug-in. This, too, can be downloaded from Sun as part of the Java Runtime Environment.

There are several drawbacks to offering applets instead of applications, as detailed earlier today. The biggest is the default security policy for applets, which makes it impossible for them to read and write data on a user's computer, among other restrictions.

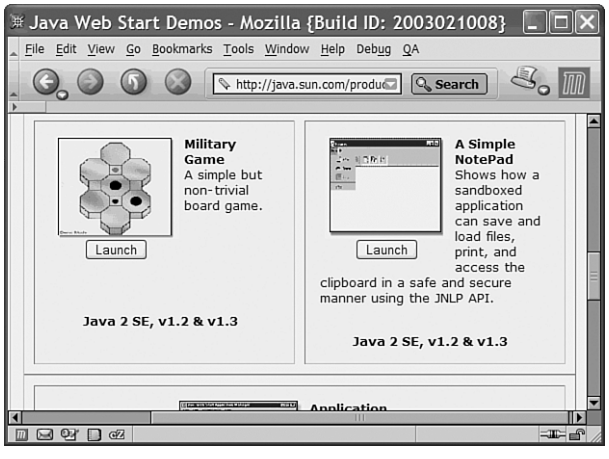
Java 2 eases the challenges of software deployment with Java Web Start, a means of running Java applications presented on a Web page and stored on a Web server. Here's how it works:

1. A programmer packages an application and all the files it needs into a JAR archive along with a file that uses the Java Network Launching Protocol (JNLP), part of Java Web Start.
2. The file is stored on a Web server with a Web page that links to that file.
3. A user loads the page with a browser and clicks the link.
4. If the user does not have the Java Runtime Environment, a dialog box opens asking whether it should be downloaded and installed. The full installation is more than 65M in size and could take 30–45 minutes to download on a 56k Internet connection (or 3–5 minutes on a high-speed connection).
5. The Java Runtime Environment installs and runs the program, opening new frames and other interface components like any other application. The program is saved in a cache, so it can be run again later without requiring installation.

To see it in action, visit Sun's Java Web Start site at `http://java.sun.com/products/javawebstart` and click the Code Samples & Apps link. The Web Start Demos page contains pictures of several Java applications, each with a Launch button you can use to run the application, as shown in Figure 14.3.

FIGURE 14.3

Presenting Web Start applications on a Web page.



Click the Launch button of one of the applications. If you don't have the Java Runtime Environment yet, a dialog box opens asking whether you want to download and install it.

The runtime environment includes the Java Plug-in, a Java interpreter that adds support for the current version of the language to browsers such as Internet Explorer and Mozilla. The environment can also be used to run applications, whether or not they use Java Web Start.

When an application is run using Java Web Start, a title screen displays on your computer briefly, and the application's graphical user interface appears.

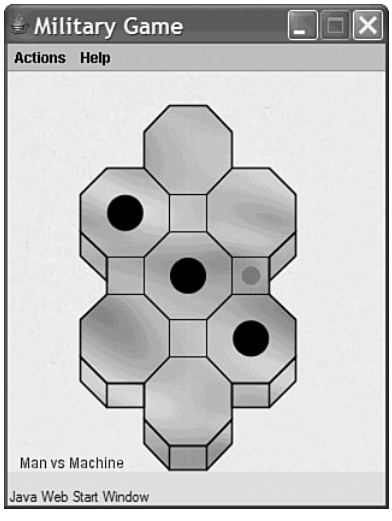
NOTE

If you have installed the Java 2 Software Development Kit, you are likely to have the Java Runtime Environment on your computer already.

Figure 14.4 shows one of the demo applications offered by Sun, a military strategy game in which three black dots attempt to keep a red dot from moving into their territory.

FIGURE 14.4

Running a Java Web Start application.



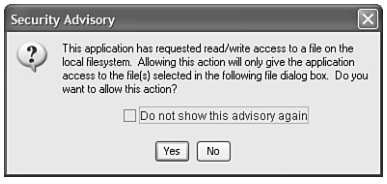
As you can see in Figure 14.4, the application looks no different from any other application. Unlike applets, which are presented in conjunction with a Web page, applications launched with Java Web Start run in their own windows, as if they were run from a command line.

One thing that's different about a Java Web Start application is the security that can be offered to users. When an application attempts to do something, such as read or write files, the user can be asked for permission.

For example, another one of the demo programs is a text editor. When you try to save a file for the first time with this application, the Security Advisory dialog box opens (see Figure 14.5).

FIGURE 14.5

Choosing an application's security privileges.



If the user does not permit something, the application cannot function fully. The kinds of things that trigger a security dialog are the same things not allowed by default in applets: reading and writing files, loading network resources from servers other than the one hosting the program, and the like.

After an application has been run by Java Web Start, it is stored on a user's computer in a cache, enabling it to be run again later without installation. The only exception is when a new version of the application becomes available. In this case, the new version is downloaded and installed automatically in place of the existing one.

A Java Web Start application viewer can be run directly to see the applications that have been cached, run them, and change some of their settings. The application is called `javaws.exe` and can be found in the same folder as `java` and the other command-line programs in the Java 2 SDK. There also should be a menu item for Java Web Start that was added during installation.

NOTE

Although you run a Java Web Start application for the first time using a Web browser, that's not a requirement. To see this, run the Java Web Start application viewer, select a program, and choose `Application, Install Shortcuts`. A shortcut to run the application is added to your desktop. You can use it to run the program without a browser.

The default security restrictions in place for a Java Web Start application can be overridden if it is stored in a digitally signed Java archive. The user is presented with the signed security certificate, which documents the author of the program and the certificate granting authority vouching for its identity, and asked whether to accept it or reject it. The application won't run unless the certificate has been accepted.

Using Java Web Start

Any Java application can be run using Java Web Start as long as the Web server that offers the application is configured to work with the technology and all the class files and other files it needs have been packaged together.

To prepare an application to use Java Web Start, you must save the application's files in a Java archive file, create a special Java Web Start configuration file for the application, and upload the files to the Web server.

The configuration file that must be created uses Java Network Launching Protocol (JNLP), an XML file format that specifies the application's main class file, its JAR archive, and other things about the program.

NOTE

XML, which is short for Extensible Markup Language, is introduced during Day 20, "Reading and Writing Data Using JDBC and XML." Because the format of JNLP files is relatively self-explanatory, you don't need to know much about XML to create a JNLP file.

The next project you will undertake is to use Java Web Start to launch and run PageData, an application that displays information about Web pages. The application requires the `PageData.class` file, which can be downloaded from the Day 14 page of the book's Web site at `http://www.java21days.com`. (You might also want `PageData.java` in case you decide to make any changes to the application.)

To get ready, put a copy of that project's class file in the folder you are using as a workspace for your Java programming.

Creating a JNLP File

The first thing you must do is package all of an application's class files into a Java archive file along with any other files it needs. If you are using the Software Development Kit, you can create the JAR file with the following command:

```
jar -cf PageData.jar PageData.class
```

A JAR file called `PageData.jar` is created that holds the class file.

Next you should create an icon graphic for the application, which will be displayed when it is loaded and used as its icon in menus and desktops. The icon can be in either GIF or JPEG format, and should be 64 pixels wide and 64 pixels tall.

For this project, if you don't want to create a new icon, you can download `pagedataicon.gif` from the book's Web site. Go to `http://www.java21days.com` and open the Day 14 page. Right-click the `pagedataicon.gif` link and save the file to the same folder as your `PageData.jar` file.

The final thing you must do is create the JNLP file that describes the application. Listing 14.5 contains a JNLP file used to distribute the PageData application. Open your word processor and enter the text of this listing; then save the file as `PageData.jnlp`.

LISTING 14.5 The Full Text of `PageData.jnlp`

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <!-- JNLP File for the PageData Application -->
 3: <jnlp
 4:   codebase="http://www.cadenhead.org/book/java-21-days/java"
 5:   href="PageData.jnlp">
 6:   <information>
 7:     <title>PageData Application</title>
 8:     <vendor>Rogers Cadenhead</vendor>
 9:     <homepage href="http://www.java21days.com"/>
10:     <icon href="pagedataicon.gif"/>
11:     <offline-allowed/>
12:   </information>
13:   <resources>
14:     <j2se version="1.5"/>
15:     <jar href="PageData.jar"/>
16:   </resources>
17:   <security>
18:     <all-permissions/>
19:   </security>
20:   <application-desc main-class="PageData"/>
21: </jnlp>
```

The structure of a JNLP file is similar to the HTML markup required to put a Java applet on a Web page. Everything within < and > marks is a tag, and tags are placed around the information the tag describes. There's an opening tag before the information and a closing tag after it.

For example, Line 7 of Listing 14.5 contains the following text:

```
<title>PageData Application</title>
```

In order from left to right, this line contains the opening tag `<title>`, the text `PageData Application`, and the closing tag `</title>`. The text between the tags, "PageData Application," is the title of the application. This title will be displayed by Java Web Start as the application is being loaded and used in menus and shortcuts.

The difference between opening tags and closing tags is that closing tags begin with a slash (/) character, and opening tags do not. In Line 8, <vendor> is the opening tag, </vendor> is the closing tag, and these tags surround the name of the vendor who created the application. I've used my name here. Delete it and replace it with your own name, taking care not to alter the <vendor> or </vendor> tags around it.

Some tags have an opening tag only, such as Line 11:

```
<offline-allowed/>
```

The `offline-allowed` tag indicates that the application can be run even if the user is not connected to the Internet. If it was omitted from the JNLP file, the opposite would be true, and the user would be forced to go online before running this application.

In XML, all tags that do not have a closing tag end with `/>` instead of `>`.

Tags can also have attributes, which are another way to define information in an XML file. An *attribute* is a name inside a tag that is followed by an equals sign and some text within quotes.

For example, consider Line 9 of Listing 14.5:

```
<homepage href="http://www.java21days.com"/>
```

This is the `homepage` tag, and it has one attribute, `href`. The text between the quotation marks is used to set the value of this attribute to `http://www.java21days.com`. This defines the home page of the application—the Web page that users should visit if they want to read more information about the program and how it works.

The PageData JNLP file defines a simple Java Web Start application that runs with no security restrictions, as defined in lines 17–19:

```
<security>
  <all-permissions/>
</security>
```

In addition to the tags that have already been described, Listing 14.5 defines other information required by Java Web Start.

Line 1 designates that the file uses XML and the UTF-8 character set. This same line can be used on any of the JNLP files you create for applications.

Line 2 is a comment. Like other comments in Java, it's placed in the file solely for the benefit of humans. Java Web Start ignores it.

The `jnlp` element, which begins on Line 3 and ends on Line 18, must surround all the other tags that configure Web Start.

This tag has two attributes, `codebase` and `href`, which indicate where the JNLP file for this application can be found. The `codebase` attribute is the URL of the folder that contains the JNLP file. The `href` attribute is the name of the file or a relative URL that includes a folder and the name (such as `"pub/PageData.jnlp"`).

In Listing 14.5, the attributes indicate that the application's JNLP file is at the following Web address:

```
http://www.cadenhead.org/book/java-21-days/java/PageData.jnlp
```

The `information` element (lines 6–12) defines information about the application. Elements can contain other elements in XML, and in Listing 14.5, the `information` element contains `title`, `vendor`, `homepage`, `icon`, and `offline-allowed` tags.

The `title`, `vendor`, `homepage`, and `offline-allowed` elements were described earlier.

The `icon` element (line 10) contains an `href` attribute that indicates the name (or folder location and name) of the program's icon. Like all file references in a JNLP file, this element uses the `codebase` attribute to determine the full URL of the resource. In this example, the `icon` element's `href` attribute is `pagedataicon.gif`, and the `codebase` is `http://www.cadenhead.org/book/java21days/java`, so the icon file is at the following Web address:

```
http://www.cadenhead.org/book/java21days/java/pagedataicon.gif
```

The `resources` element (lines 13–16) defines resources used by the application when it runs.

The `j2se` element has a `version` attribute that indicates the version of the Java interpreter that should run the application. This attribute can specify a general version (such as `"1.3"`, `"1.4"`, or `"1.5"`), a specific version (such as `"1.5.1-beta"`), or a reference to multiple versions—follow a general version number with a plus sign. The tag `<j2se version="1.3+">` sets up an application to be run by any Java interpreter from version 1.3 upward.

NOTE

When you're using the `j2se` element to specify multiple versions, Java Web Start will not use a beta version to run an application. The only way to run an application with a beta release is to indicate that release specifically.

The `jar` element has an `href` attribute that specifies the application's JAR file. This attribute can be a filename or a reference to a folder and filename, and it uses `codebase`. In the PageData example, the JAR file is in `http://www.cadenhead.org/book/java21days/java/PageData.jar`.

The `application-desc` element indicates the application's main class file and any arguments that should be used when that class is executed.

The `main-class` attribute identifies the name of the class file, which is specified without the `.class` file extension.

If the class should be run with one or more arguments, place `argument` elements within an opening `<application-desc>` tag and a closing `</application-desc>` tag.

The following XML specifies that the `PageData` class should be run with two arguments: `http://java.sun.com` and `yes`:

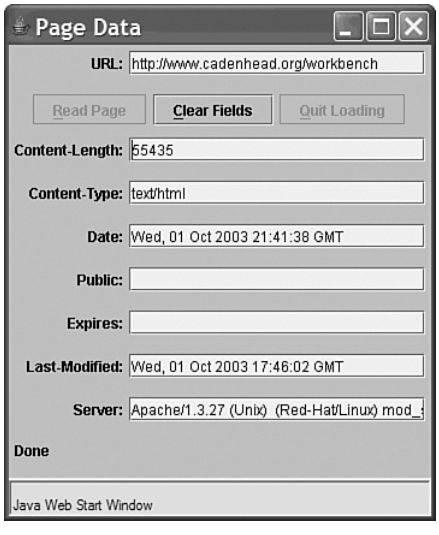
```
<application-desc main-class="PageData">
  <argument>http://java.sun.com</argument>
  <argument>yes</argument>
</application-desc>
```

After you have created the `PageData.jnlp` file, change Line 4 of Listing 14.5 so that it refers to the folder on a Web server where your application's JAR file, icon file, and JNLP file will be stored.

Upload all three of the project's files to this folder; then run your browser and load the JNLP file using its full Web address. If your Web server is configured to support Java Web Start, the application will be loaded and begin running, as in Figure 14.6.

FIGURE 14.6

Running PageData using Java Web Start.



For this application to be run without restriction, the `PageData.jar` file must be digitally signed. For real-world applications, this requires the services of a certificate-granting authority such as Thawte or Verisign and can cost $1,000 or more per year.

For testing purposes, the `keystore` and `jarsigner` tools in the Java 2 SDK can be used to create a key and use it to digitally sign a JAR file.

The first step is to use `keytool` to create a key and assign it an alias and password:

```
keytool -genkey -alias examplekey -keypass swordfish
```

The `-genkey` argument generates a new key—which in this example is named `examplekey` and has the password `swordfish`. If this is the first time `keytool` has been used, you'll be prompted for a password that protects access to the key database, which is called a *keystore*.

After a key is placed in the keystore, it can be used with the `jarsigner` tool to sign an archive file. This tool requires the keystore and key passwords and the alias of the key. Here's how the `PageData.jar` archive could be signed with the `examplekey` key:

```
jarsigner -storepass secret -keypass swordfish PageData.jar examplekey
```

The keystore password in this example is `secret`. The security certificate used to sign the archive lasts 90 days and is described as an "untrusted source" when the Java Web Start application is run.

NOTE

There's no easy way to avoid being described as "untrusted." The only way to establish your trustworthiness is to go through one of the professional certificate-granting companies.

Java developer Roedy Green offers a guide to Java security certification that lists several companies and their prices. Visit the Web page `http://mindprod.com/jgloss/certificate.html`.

Supporting Web Start on a Server

If your server does not support Java Web Start, which is more likely than not because it is a relatively new technology, you may see the text of your JNLP file loaded in a page, and the application will not open.

A Web server must be configured to recognize that JNLP files are a new type of data that should cause a Java application to run. This is usually accomplished by setting the MIME type associated with files of the extension JNLP.

MIME, which is an acronym for Multipurpose Internet Mail Extensions, is a protocol for defining Internet content such as email messages, attached files, and any file that can be delivered by a Web server.

On an Apache Web server, the server administrator can support JNLP by adding the following line to the server's `mime.types` (or `.mime.types`) file:

```
application/x-java-jnlp-file JNLP
```

If you can't get Java Web Start working on your server, you can test this project on the book's official site. Load the Web page `http://www.cadenhead.org/book/java-21-days/java/PageData.jnlp`, or visit the Web address `http://www.java21days.com` and open the Day 14 page.

CAUTION

Java Web Start applications should look exactly like the applications do when run by other means. However, there appear to be a few bugs in how much space is allocated to components on a graphical user interface. On a Windows system, you might need to add 50 pixels to the height of an application before employing it in Java Web Start. Otherwise, the text fields are not tall enough to display numbers.

Additional JNLP Elements

The JNLP format has other elements that can affect the performance of Java Web Start.

It can be used to change the title graphic that appears when the application is launched, run signed applications that have different security privileges, run an application using different versions of the Java interpreter, and other options.

Security

By default, all Java Web Start applications will not have access to some features of a user's computer unless the user has given permission. This is similar to how the functionality of applets is limited.

If your application's JAR file has been digitally signed to verify its authenticity, it can be run without these security restrictions by using the `security` element.

This element is placed inside the `jnlp` element, and it contains one element of its own: `all-permissions`. To remove security restrictions for an application, add this to a JNLP file:

```
<security>
  <all-permissions/>
</security>
```

Descriptions

If you want to provide more information about your application for users of Java Web Start, one or more `description` elements can be placed inside the `information` element.

Four kinds of descriptions can be provided using the `kind` attribute of the `description` element:

- `kind="one-line"`—A succinct one-line description, used in lists of Web Start applications
- `kind="short"`—A paragraph-long description, used when space is available
- `kind="tooltip"`—A tooltip description
- No `kind` attribute—A default description, used for any other descriptions not specified

All these are optional. Here's an example that provides descriptions for the PageData application:

```
<description>The PageData application.</description>
<description kind="one-line">An application to learn more about Web
servers and pages.</description>
<description kind="tooltip">Learn about Web servers and
pages.</description>
<description kind="short">PageData, a simple Java application that
takes a URL and displays information about that URL and the Web
server that delivered it.</description>
```

Icons

The PageData JNLP file included a 64×64 icon, `pagedataicon.gif`, used in two different ways:

- When the PageData application is being loaded by Java Web Start, the icon is displayed on a window next to the program's name and author.
- If a PageData icon is added to a user's desktop, the icon will be used at a different size: 32×32.

When an application is loading, you can use a second `icon` element to specify a graphic that will be displayed in place of the icon, title, and author. This graphic is called the application's "splash screen," and it is specified with the `kind="splash"` attribute, as in this example:

```
<icon kind="splash" href="pagedatasplash.gif"/ width="300" height="200">
```

The `width` and `height` attributes, which also can be used with the other kind of icon graphic, specify the image's display size in pixels.

This second `icon` element should be placed inside the `information` element.

Running Applets

Although all the text up to this point has covered applications, Java Web Start also can be used to run applets.

An applet is run differently by Web Start than it would be run by a browser. The applet uses the same default security as an application, asking users for permission to undertake some tasks, and it is not displayed in a Web page. The applet runs in its own window, making it appear to be an application.

To run an applet with Web Start, use the `applet-desc` element instead of the `application-desc` element.

The `applet-desc` element, which must be contained within the `resources` element, has five attributes:

- `name`—The name of the applet.
- `main-class`—The name of the applet's main class, which will be executed when the applet is run.
- `width`—The width of the applet window.
- `height`—The height of the applet window.
- `documentBase`—The base URL of the document. When the applet is run by a browser, the document base is a folder that holds the page containing the applet. Java Web Start does not run applets inside a page, so there is no way to automatically determine the document base. Because some applets call `getDocumentBase()` to use this URL, it must be explicitly specified as an attribute.

If an applet is run with parameters, these are specified with the `param` element, which is contained within `applet-desc`. This takes the same form in a JNLP file that it does on an HTML page:

```
<param name="somename" value="somevalue">
```

Today's final project is the conversion of the `NewWatch` applet to use Java Web Start.

To get ready for the project, copy the `NewWatch.class` file to a folder and add it to a new JAR archive using the following command:

```
jar -cf NewWatch NewWatch.class
```

A JAR archive called `NewWatch.jar` is created. This file is used by Java Web Start to run the applet. To accomplish this, enter the text of Listing 14.6, saving the JNLP file as `NewWatch.jnlp`.

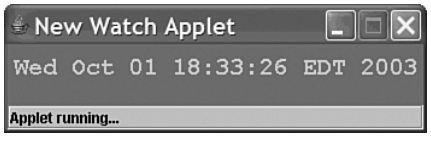
LISTING 14.6 The Full Text of `NewWatch.jnlp`

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <!-- JNLP File for the NewWatch Applet -->
 3: <jnlp
 4:   codebase="http://www.cadenhead.org/book/java-21-days/java"
 5:   href="NewWatch.jnlp">
 6:   <information>
 7:     <title>New Watch Applet</title>
 8:     <vendor>Rogers Cadenhead</vendor>
 9:     <homepage href="http://www.java21days.com"/>
10:     <description kind="one-line">An applet that displays
11: the current time.</description>
12:     <offline-allowed/>
13:   </information>
14:   <resources>
15:     <j2se version="1.2+"/>
16:     <jar href="NewWatch.jar"/>
17:   </resources>
18:   <applet-desc
19:     name="NewWatch applet"
20:     main-class="NewWatch"
21:     documentBase="http://www.cadenhead.org/book/java21days"
22:     width="345"
23:     height="50">
24:
25:     <param name="background" value="#996633">
26:   </applet-desc>
27: </jnlp>
```

Figure 14.7 shows the applet.

FIGURE 14.7

Running an applet with Java Web Start.



The `applet-desc` element runs the `NewWatch` applet in a window 345 pixels wide and 50 pixels high. One parameter is specified using a `param-name` element in line 25: the background color to display underneath the current date and time.

NOTE

For more information on using the technology with your own applications, visit Sun's Java Web Start site at the following address:

`http://java.sun.com/products/javawebstart`

Summary

Although applets are no longer the focus of Java development, they are still the element of Java technology that reaches the most people, appearing on thousands of World Wide Web sites.

Because they are executed and displayed within Web pages, applets can use the graphics, user interface, and event structure provided by the Web browser. This capability provides the applet programmer with a lot of functionality without a lot of extra toil.

Another Web-based way to present Java programs is to use Java Web Start, a new technology that blurs the distinction between applications and applets.

With Web Start, users no longer need to run an installation program to set up a Java application and the interpreter that executes the class. Web Start takes care of this automatically, after the user's browser has been equipped to use the Java 2 Runtime Environment.

Support for Web Start is offered through the Java Network Launching Protocol (JNLP), an XML file format used to define and set up Java Web Start.

Q&A

Q I have an applet that takes parameters and an HTML file that passes it those parameters, but when my applet runs, all I get are `null` values. What's going on here?

A Do the names of your parameters (in the `NAME` attribute) exactly match the names you're testing for in `getParameter()`? They must be exact, including case, for the match to be made. Also make sure that your `PARAM` tags are inside the opening and closing `APPLET` tags and that you haven't misspelled anything.

Q Because applets don't have a command line or a standard output stream, how can I do simple debugging output such as `System.out.println()` in an applet?

A Depending on your browser or other Java-enabled environment, you might have a console window in which debugging output (the result of `System.out.println()`) appears, or it might be saved to a log file. (Netscape has a Java Console under the Options menu; Internet Explorer uses a Java log file that you must enable by choosing Options, Advanced.)

You can continue to print messages using `System.out.println()` in your applets—just remember to remove them after you're finished so that they don't confuse your users.

Q An applet I am trying to run doesn't work—all I see is a gray box. Is there a place I can view any error messages generated by this applet?

A If you're using the Java Plug-in to run applets, you can view error messages and other information by opening the Java Console. To see it in Windows, double-click the Java cup icon in the System Tray. Current versions of Mozilla and Netscape Navigator make the Java output window available as a pull-down menu: Choose Tools, Web Development, Java Console.

Q I have written an applet that I want to make available using Java Web Start. Should I convert it to an application or go ahead and run it as-is?

A If you would be converting your program to an application simply to run it with Web Start, that's probably not necessary. The purpose of the `applet-desc` tag is to make it possible to run applets without modification in Java Web Start. The only reason to undertake the conversion is if there are other things you want to change about your program, such as the switch from `init()` to a constructor method.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which class should an applet inherit from if Swing features will be used in the program?
   a. `java.applet.Applet`
   b. `javax.swing.JApplet`
   c. Either one
2. Which XML element is used to identify the name, author, and other details about a Java Web Start-run application?
   a. `jnlp`
   b. `information`
   c. `resources`
3. What happens if you put a Java 2 applet on a Web page using the `APPLET` tag and it is loaded by a copy of Internet Explorer 6 that does not include the Java Plug-in?
   a. It runs correctly.
   b. It doesn't run, and an empty gray box is displayed.
   c. The user is offered a chance to download and install the Java Plug-in.

Answers

1. b. If you're going to use Swing's improved interface and event-handling capabilities, the applet must be a subclass of `JApplet`.
2. b. The application is described using elements contained within an opening `<information>` tag and a closing `<information>` tag.
3. b. The applet won't run because Internet Explorer doesn't offer a Java interpreter. The user won't be given a chance to download and install the Java Plug-in unless you use a Web page converted by HTMLConverter.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given the following,

```
import java.awt.*;
import javax.swing.*;

public class SliderFrame extends JFrame {
    public SliderFrame() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container pane = getContentPane();
        JSlider value = new JSlider(0, 255, 100);
        setContentPane(pane);
        setSize(325, 150);
        setVisible(true);
    }

    public static void main(String[] arguments) {
        new SliderFrame();
    }
}
```

What will happen when you attempt to compile and run this source code?

a. It compiles without error and runs correctly.

b. It compiles without error but does not display anything in the frame.

c. It does not compile because the content pane is empty.

d. It does not compile because of the `new AskFrame()` statement.

The answer is available on the book's Web site at `http://www.java21days.com`. Visit the Day 14 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Enhance the `NewWatch` applet so that you can set the color of the text with a parameter.
2. Create a new JNLP file that runs the PageData application using version 1.3 of the Java interpreter and force users to be connected to the Internet when it is run.

Where applicable, exercise solutions are offered on the book's Web site at `http://www.java21days.com`.