CONTENTS

Chapter 9. Applets and Basic Graphics

Topics in This Chapter

- Creating *applets:* Java programs that are embedded in Web pages
- The applet life cycle
- Customizing applets through parameters embedded in HTML
- The Java Plug-In
- Creating graphical *applications:* Java programs that run independently of a Web page or browser
- Basic drawing, color, font, and clipping area operations
- Loading and drawing images
- Preloading images
- Controlling image loading with MediaTracker

In this chapter we discuss the two basic types of graphical Java programs: applets and applications. For applets, we explain how to create them, how to associate them with Web pages, and how to supply customization parameters through the `PARAM` element. For applications, we discuss one common approach to creating windows, postponing alternatives until Chapter 13 (AWT Components). We cover the basic drawing operations that can be performed in applets and applications. However, for applications, Java2D graphics, presented in Chapter 10 (Java 2D: Graphics in Java 2), are preferred for basic drawing operations. Finally, we discuss methods for loading and displaying images in both applets and applications.

9.1 What Are Applets?

An applet is a particular type of Java program that is intended to be embedded in a Web page. When a user opens a Web page containing an applet, the applet runs *locally* (on the client machine that is running the Web browser), not *remotely* (on the system running the HTTP server). Consequently, security considerations are paramount, and applets are restricted from performing various operations that are allowed in general Java programs ("applications"). For instance, you might need to write a stand-alone program that deletes files, but you certainly don't want to let applets that come in over the Web delete your files. These restrictions are enforced by a `SecurityManager` object on the client

system. In version 1.1 and later of the Java Platform, classes can be digitally signed, and the user can ask the security manager to allow various restricted operations in classes signed by certain individuals or organizations. Technically, the precise restrictions placed upon applets depend on the `SecurityManager`. However, in Netscape and Internet Explorer, the default manager verifies that applets:

Do not read from the local (client) disk. That is, they cannot read arbitrary files. Applets can, however, instruct the browser to display pages that are generally accessible on the Web, which might include some local files.

Do not write to the local (client) disk. The browser may choose to cache certain files, including some loaded by applets, but this choice is not under direct control of the applet.

Do not open network connections other than to the server from which the applet was loaded. This restriction prevents applets from browsing behind network firewalls.

Do not call local programs. Ordinary Java applications can invoke locally installed programs (with the `exec` method of the `Runtime` class) as well as link to local C/C++ modules ("native" methods). These actions are prohibited in applets because there is no way to determine whether the operations these local programs perform are safe.

Cannot discover private information about the user. Applets should not be able to discover the username of the person running them or specific system information such as current users, directory names or listings, system software, and so forth. However, applets *can* determine the name of the host they are on; this information is already reported to the HTTP server that delivered the applet.

9.2 Creating an Applet

Creating an applet involves two steps: making the Java class and making the associated HTML document. The Java class defines the actual behavior of the applet, and the HTML document associates the applet with a particular rectangular region of the Web page.

Template for Applets

Listing 9.1 shows the typical organization of an applet. It contains a section for declaring instance variables, an `init` method, and a `paint` method. The `init` method is automatically called by the browser when the applet is first created. Then, when the applet is ready to be drawn, `paint` is called. The `paint` method is automatically called again whenever the image has been obscured and is reexposed or when graphical components are added, or the method can be invoked programmatically. The default implementations of `init` and `paint` don't do anything; they are just provided as placeholders for the programmer to override. Although there are additional placeholders for user code (see Section 9.4, "The Applet Life Cycle"), this basic structure (declarations, `init`, `paint`) is a good starting point for most applets. Although the Java programming language, unlike C++, allows direct initialization of instance variables when they are declared in the body of the class, this initialization is not recommended for applets. Prior to `init`, the applet has not set up everything set up that it needs in order to initialize certain types of graphical objects. So, rather than trying to remember which variables can be directly initialized (strings) and which must be done in `init` (images), follow the good rule of simply declaring them in the main body of the applet and initializing them in `init`. Of course, variables that are only needed in `init` can be local to `init` rather than instance variables available to the whole class.

Listing 9.1 Java applet template

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

public class AppletTemplate extends Applet {
```

```
  // Variable declarations.

  public void init() {
    // Variable initializations, image loading, etc.
  }

  public void paint(Graphics g) {
    // Drawing operations.
  }
}
```

Template for HTML

Once an applet has been created and compiled, the resultant class file must be associated with a Web page. The APPLET element is used for this, as shown in Listing 9.2. This element is discussed further in Section 9.6, but for now note that CODE designates the name of the Java class file and that the two attributes, WIDTH and HEIGHT, are required. In addition, you need to use CODEBASE if the applet is being loaded from somewhere other than the place the associated HTML document resides. After the applet has been compiled and associated with a Web page, loading the Web page in a Java-enabled browser executes the applet. You sometimes name the HTML file with the same prefix as the Java file (e.g., AppletTemplate.html to correspond to AppletTemplate.class), but the name is arbitrary, and in fact, a single HTML document can load multiple applets.

Although the compiled file (*file*.class) for your applet must be available on the Web, the source code (*file*.java) need not be. If you put the source and class files in different locations, remember that *all* nonsystem class files used by your applet need to be WWW accessible and should be either in the same directory as the applet or in subdirectories corresponding to their package. If you move class files from one system to another by using FTP, be sure to use binary (raw) mode, not text mode.

Finally, some Java systems have the concept of a CLASSPATH that tells the system where to look for classes. If your system uses this feature, note that CLASSPATH settings apply only to local programs; a remote user accessing your applet won't know anything about your CLASSPATH. Also, since both Netscape and Internet Explorer grant extra privileges to class files that are listed in your CLASSPATH, you want to be sure your CLASSPATH is *not* set in the process that starts your browser. This practice prevents people who know your CLASSPATH from using your own classes to attack you.

Core Security

Make sure your CLASSPATH is not set when you start your browser.

Listing 9.2 HTML applet template

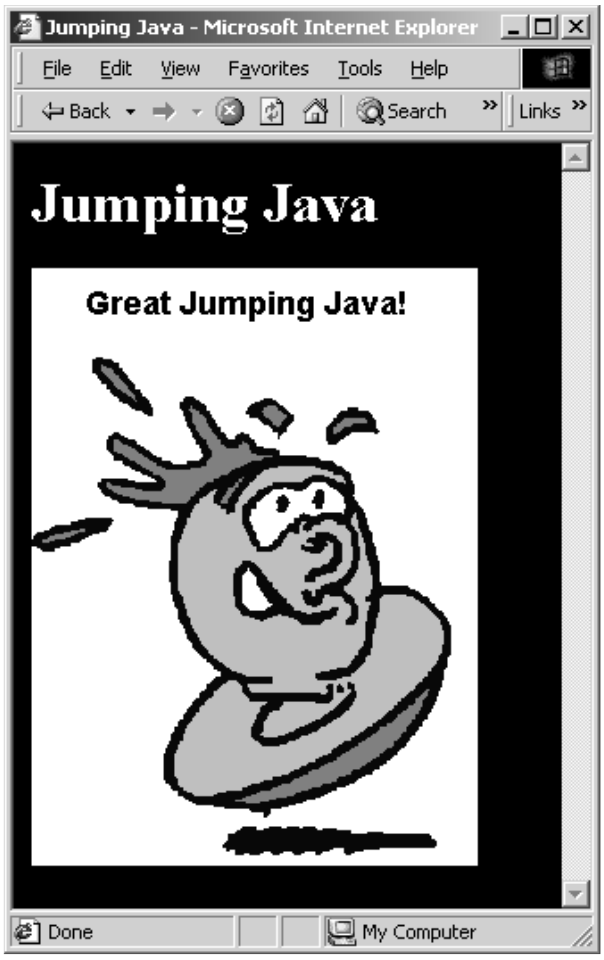
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>A Template for Loading Applets</TITLE>
</HEAD>

<BODY>
<H1>A Template for Loading Applets</H1>
<P>
<APPLET CODE="AppletTemplate.class" WIDTH=120 HEIGHT=60>
```

```
  <B>Error! You must use a Java-enabled browser.</B>
</APPLET>

</BODY>
</HTML>
```

9.3 An Example Applet

Listing 9.3 presents an applet that follows the pattern shown in the previous section. The details of the various steps the applet performs are covered in later sections, but the basic approach is what is important here. The variable declaration section declares a variable of type `Image`. The `init` method sets the default color and font, loads an image file from the network and assigns it to the `Image` declared earlier, adds a label to the applet, then performs an informational printout. The `paint` method draws the image, placing its top 50 pixels from the top of the applet. Listing 9.4 shows the associated HTML document; the result in Internet Explorer 5.0 (Windows 2000) is shown in Figure 9-1.

Figure 9-1. Result of applet shown in Listing 9.3 and Listing 9.4.



Listing 9.3 `JavaJump.java`

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

/** An applet that draws an image. */
```

```
public class JavaJump extends Applet {
  private Image jumpingJava; // Instance var declarations here

  public void init() {       // Initializations here
    setBackground(Color.white);
    setFont(new Font("SansSerif", Font.BOLD, 18));
    jumpingJava = getImage(getDocumentBase(),
                           "images/Jumping-Java.gif");
    add(new Label("Great Jumping Java!"));
    System.out.println("Yow! I'm jiving with Java.");
  }

  public void paint(Graphics g) {  // Drawing here
    g.drawImage(jumpingJava, 0, 50, this);
  }
}
```

Listing 9.4 `JavaJump.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Jumping Java</TITLE>
</HEAD>
<BODY BGCOLOR="BLACK" TEXT="WHITE">
<H1>Jumping Java</H1>
<P>
<APPLET CODE="JavaJump.class" WIDTH=250 HEIGHT=335>
  <B>Sorry, this example requires Java.</B>
</APPLET>
</BODY>
</HTML>
```

Redrawing Automatically

In most applets, the main drawing is done in `paint`. If something changes that requires the drawing to change, you can call the `repaint` method, which calls `update`, which normally clears the screen and then calls `paint`. We'll give more details on `repaint` and `update` later, but the point is that they are called whenever the *programmer* wants the screen to be redrawn. The *system* may also want to redraw the screen, as is typical when part of the screen has been covered up by some other window and then reexposed. In the vast majority of cases, you don't care why `paint` is called; you do the same thing either way. However, when the system calls `paint`, it sets the clipping region of the `Graphics` object to be the part that was obscured. This means that in the unlikely case that `paint` draws something different every time it is called but `repaint` isn't invoked, the screen won't be redrawn correctly unless you adjust the clipping region (see Section 9.11, "Graphics Operations").

Reloading Applets During Development

In Netscape and Internet Explorer, the applet is automatically cached in memory. As a result, you can change the applet code and not see the changes reflected in the browser because the browser is using the previously cached applet version. To force loading of the new applet, use *Shift-Reload* (holding down the shift key when clicking the Reload button) in Netscape and use *Control-Reload* in Internet Explorer.

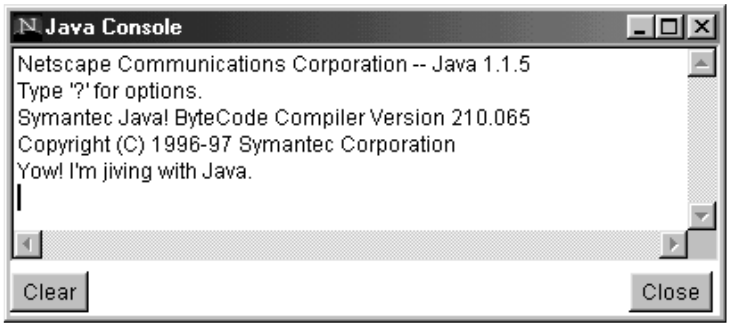
Getting Standard Output

For output in "production" applets, you normally create a text area or other GUI control to display results. We cover this approach at length in Chapter 13 (AWT Components). You can also send a single line of debugging results to the status line by using the applet's `showStatus` method. However, during development it is often convenient to print multiple lines of debugging output with `System.out.println`. When you print this way from an applet, where do you see the result? The answer depends on the browser being used.

Standard Output

In Netscape Navigator, the user can select Java Console from the Window menu, or in Netscape Communicator, the user can select Java Console from the Communicator menu, followed by the Tools menu, to get a pop-up window in which to see output. For example, Figure 9-2 shows the Java Console after the `JavaJump` example is run. You can enter the ? keystroke in the Java Console for a listing of available command options.

Figure 9-2. In Netscape, standard output is sent to a separate window: the Java Console.



In Internet Explorer 5, printing is a little more cumbersome to set up. First, from the Tools menu, select Internet Options, then choose the Advanced screen. There, under the Java VM category, select Java console enabled. Then, quit Internet Explorer, and restart. After restart, from the View menu, select Java Console to get a pop-up window for output. Entering the ? keystroke in the Java Console provides a listing of available command options. Note that the Java Console is enabled by default in Internet Explorer 4.

Standard Output in Appletviewer

In appletviewer on Windows and Unix platforms, the output is sent to the window that started appletviewer. On a Mac, a separate window pops up when output is sent to `System.out`.

9.4 The Applet Life Cycle

`Applet` is an unusual class since the browser automatically creates an instance of it and calls certain methods at certain times. The `main` method is never called by the browser. Instead, the following placeholder methods are called at various times. The methods are empty unless overridden by the author.

public void init()

This method is called after the applet instance is first created by the browser. In Netscape, `init` is not called again if the applet is stopped and restarted unless the applet has been "trimmed" (deleted from memory) by the browser. In Internet Explorer, `init` is called whenever the user returns to the page containing the applet.

public void start()

The `start` method is called after `init` is completed, but before the first invocation of `paint`. If the applet is stopped and restarted, `start` is called again each time. This

makes `start` a good place to restart animation that gets paused when the user leaves the page. Netscape calls `start` and `stop` when the browser window is resized; Internet Explorer, appletviewer, and HotJava don't.

public void paint(Graphics g)

This is where user-level drawing is placed. The method is invoked by the browser after `init` and `start` have completed, and again whenever the browser thinks the screen needs to be redrawn, typically when part of the screen has been obscured and then reexposed. From another method, you can call `repaint` with no arguments to tell the browser's graphics thread to call `paint` at its first opportunity.

public void stop()

The browser calls `stop` when the user leaves the page containing the applet. The method can be used to halt animation that will be restarted by `start`. In Netscape, this method is also called when the browser window is resized.

public void destroy()

The `destroy` method is called when the applet is about to be permanently destroyed (e.g., when the browser is shut down or when it "trims" the applet from memory to keep the browser image size from growing too large). The method can be used for bookkeeping or to free up shared resources but is not used frequently. In Internet Explorer, `destroy` is called whenever the user leaves the page containing the applet.

Core Alert

In Internet Explorer, unlike in Navigator, `init` is called each time the user returns (in the same session) to a page containing a previously loaded applet, and `destroy` is called whenever the user leaves the page containing the applet.

9.5 Other Applet Methods

The built-in versions of the methods described in Section 9.4 don't actually *do* anything; they are simply placeholders for the user to override. `Applet` also contains a number of methods that perform common, useful tasks; the most frequently used ones are summarized below. `Applet` inherits from `Panel`, `Container`, and `Component`; full details of these classes are given in Chapter 13 (AWT Components).

public void add(Component c)

public void add(Component c, Object constraints)

public void add(String location, Component c)

public void add(PopupMenu menu)

These methods insert graphical components into the applet window. They are discussed in Chapter 13 (AWT Components) and Chapter 12 (Layout Managers).

public boolean contains(int x, int y)

public boolean contains(Point p)

The `contains` method determines if the specified location is contained inside the applet. That is, it returns `true` if and only if the *x*-coordinate is less than or equal to the applet's

width and the *y*-coordinate is less than or equal to the applet's height.

public Image createImage(int width, int height)

public Image createImage(ImageProducer producer)

The `createImage` method is used to make an off-screen pixmap. Like `getImage`, `createImage` will fail if used prior to `init`. So, it should not be used to directly initialize instance variables.

public String getAppletInfo()

You can override this method to return a string describing the author, version, and other information about the applet.

public AudioClip getAudioClip(URL audioFile)

public AudioClip getAudioClip(URL base, String audioFilename)

These methods retrieve an audio or MIDI file from a remote location and assign it to an `AudioClip` object, which supports `play`, `loop`, and `stop` methods. The JDK 1.1 supports only `.au` file formats. In JDK 1.2, `.aiff`, and `.wav` audio file formats are also supported, along with MIDI Type 0, MIDI Type 1, and RMF song file formats.

public Color getBackground()

public void setBackground(Color bgColor)

These methods get and set the background color of the applet. Create colors by calling the `Color` constructor, as follows:

```
Color someColor = new Color(red, green, blue);
```

The red, green, and blue parameters should be integers from 0 to 255 or floats from 0.0 to 1.0. Colors can also be created with `Color.getHSBColor(hue, saturation, brightness)`, where the arguments are floats between 0.0 and 1.0. Alternatively, there are thirteen predefined colors: `Color.black`, `Color.blue`, `Color.cyan`, `Color.darkGray`, `Color.gray`, `Color.green`, `Color.lightGray`, `Color.magenta`, `Color.orange`, `Color.pink`, `Color.red`, `Color.white`, and `Color.yellow`.

A `SystemColor` class provides access to the desktop colors. This class lets you create applets that conform to the user's current color scheme. For instance, in `paint` you could call `g.setColor(SystemColor.windowText)` before doing `drawString`. Table 9.1 lists the options available.

Table 9.1. System Colors

Color (Static variables in SystemColor class)	***Meaning***
`activeCaption`	The background color for captions of active windows.
`activeCaptionBorder`	The border color for captions of active windows.
`control`	The background color for control objects ("widgets").
`controlDkShadow`	The dark shadow color used to give a 3D effect.
`controlHighlight`	The emphasis color.
`controlLtHighlight`	A lighter emphasis color.
`controlShadow`	The light shadow color used to give a 3D effect.

controlText	The text color.
desktop	The desktop background color.
inactiveCaption	The background color for captions of inactive windows.
inactiveCaptionBorder	The border color for captions of inactive windows.
inactiveCaptionText	The text color for captions of inactive windows.
info	The background color for help text ("tool tips").
infoText	The text color for help text ("tool tips").
menu	The background color of deselected menu items. Selected menu items should use textHighlight.
menuText	The text color of deselected menu items. Selected menu items should use textHighlightText.
scrollbar	The background color for scrollbars.
text	The background color for text components.
textHighlight	The background color for highlighted text such as selected text in a textfield, selected menu items, etc.
textHighlightText	The text color for highlighted text.
textInactiveText	The text color for inactive components.
textText	The text color for text components.
window	The background color for windows.
windowBorder	The border color for windows.
windowText	The text color for windows.

The SystemColor class allows you to directly determine the desktop colors for setting the look of the applet to match the windowing system. However, if you create applets strictly based on Swing components, then you can easily present a particular platform "look and feel" by calling methods from the UIManager class. Available look and feels include Motif, Windows, Mac, and Java (Metal). See Chapter 14 (Basic Swing) for details on setting the look and feel of a Swing applet.

public URL getCodeBase()

public URL getDocumentBase()

These methods return the locations of the applet (getCodeBase) and the HTML document using the applet (getDocumentBase).

public Component getComponentAt(int x, int y)

The getComponentAt method returns the topmost component at the specified location. This will be the applet itself if no other component is at this location; null is returned if x and y are outside the applet.

public Cursor getCursor()

public void setCursor(Cursor cursor)

These methods get and set the cursor.

public Font getFont()

public void setFont(Font defaultFont)

These methods get and set the default font for the applet. Unless overridden explicitly, the default font is used for labels, buttons, textfields, and other such components; when strings

are drawn, use the `drawString` method of `Graphics`. Fonts are created with the `Font` constructor, which takes a family, style, and size as follows:

```
String family = "Serif";
int style = Font.BOLD;
int size = 18;
Font font = new Font(family, style, size);
setFont(font);
```

In JDK 1.1, the font family can be `Serif`, `SansSerif`, `Monospaced`, `Dialog`, and `DialogInput`. The style should be one of `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`, or `Font.BOLD|Font.ITALIC`. The size must be an integer. In JDK 1.1, nonstandard fonts cannot be used even if they are locally installed. In JDK 1.2, any local font is available provided you call the `getAvailableFontFamilyNames` or `getFonts` method of `GraphicsEnvironment` first. See Section 10.5 (Using Local Fonts) for more details on font graphics.

public FontMetrics getFontMetrics(Font f)

This method returns an object that can be used to determine the size of strings (`stringWidth`) or individual characters (`charWidth`) in a given font.

public Color getForeground()

public Color setForeground(Color fgColor)

These methods get and set the default foreground color for the applet. See `getBackground` for a discussion of colors.

public Graphics getGraphics()

The `getGraphics` method returns the current graphics object for the applet. Use this method if you want to perform drawing operations from a method that is not directly called from `paint`. The `paint` method automatically has the graphics object.

public Image getImage(URL imageFile)

public Image getImage(URL base, String imageFilename)

These methods "register" a remote image file with an `Image` object. The Java platform does not actually load the image file until you try to draw it or ask the system to start loading it with `prepareImage` or with `MediaTracker`. For more details, see Section 9.12 (Drawing Images).

public Locale getLocale()

public void setLocale(Locale locale)

These methods get and retrieve the current locale for use in internationalized code.

public String getParameter(String parameterName)

This method retrieves the value of parameters set in the `PARAM` tag inside the `APPLET` element in the HTML document. For details, see Section 9.7 (Reading Applet Parameters).

public String[ ][ ] getParameterInfo()

This method supplies documentation on the parameters an applet recognizes. Each element in the top-level array should be an array containing the parameter name, its type, and a short description.

public Container getParent()

In general, `getParent` returns the enclosing window or `null` if there is none.

public Dimension getSize()

The `getSize` method retrieves a `Dimension` object describing the size of the applet. `Dimension` has `width` and `height` fields. Thus, to get the width of the applet, use `getSize().width`. Technically, applets also have a `setSize()` method. However, in practice this method is ignored by most browsers other than appletviewer, and the sizes specified in the `WIDTH` and `HEIGHT` attributes of the `APPLET` element are used as the *permanent* dimensions.

public boolean isActive()

This method determines whether the applet is "active." Applets are inactive before `start` is called and after `stop` is called; otherwise, they are active.

public void play(URL audioFile)

public void play(URL base, String audioFilename)

These methods retrieve and play an audio file in `.au` format. The JDK 1.2 also supports `.aiff`, and `.wav` format, as well as MIDI Type 0, MIDI Type 1, and RMF song formats. See `getAudioClip`.

public void repaint()

public void repaint(long millisecondDelay)

public void repaint(int x, int y, int width, int height)

public void repaint(long msDelay, int x, int y, int width, int height)

The `repaint` method asks the AWT update thread to call `update`, either immediately or after the specified number of milliseconds. In either case, control is returned immediately; the actual updating and painting are done in a separate thread. You can also ask the system to repaint only a portion of the screen. Doing so results in `update` and `paint` getting a `Graphics` object with the specified clipping region set.

public void showDocument(URL htmlDoc) [in class AppletContext]

public void showDocument(URL htmlDoc, String frameName) [in class AppletContext]

These methods ask the browser to retrieve and display a Web page. They are actually part of the `AppletContext` class, not `Applet`, but they are used from applets similarly to the other methods described here. To use them, you call `getAppletContext().showDocument(...)`, not just `showDocument(...)`. The `showDocument` method is ignored by appletviewer.

public void showStatus(String message)

The `showStatus` method displays a string in the status line at the bottom of the browser.

public void update(Graphics g)

This method is called by the AWT thread after `repaint` is called. The default implementation of `update` clears the screen, then calls `paint`. Animation and double buffering applications typically override `update` to simply call `paint`, omitting the screen-clearing step. This subject is discussed further in Chapter 16 (Concurrent Programming with Java Threads).

addComponentListener, addFocusListener, addKeyListener, andMouseListener, addMouseMotionListener

These public methods add listeners to the applet for handling of various events. Each `addXxxListener` method has a corresponding `removeXxxListener` method. They are discussed in Chapter 11 (Handling Mouse and Keyboard Events).

9.6 The HTML APPLET Element

HTML Element:	`<APPLET CODE="..." WIDTH=xxx HEIGHT=xxx ...>` `... </APPLET>`
Attributes:	`CODE`, `WIDTH` (required), `HEIGHT` (required), `CODEBASE`, `ALT`, `ALIGN`, `HSPACE`, `VSPACE`, `NAME`, `OBJECT`, `ARCHIVE` (nonstandard), `MAYSCRIPT` (nonstandard)

The `APPLET` element associates a class file with a Web page. The referenced class file must extend the `Applet` class. Either the `CODE` or `OBJECT` attribute, as well as the `WIDTH` and `HEIGHT` attributes are required in the `APPLET` tag.

CODE `CODE` designates the filename of the Java class file to load and is required unless the `OBJECT` attribute is present. This is not an absolute URL; it is interpreted with respect to the current document's base directory unless `CODEBASE` is supplied. Although the class file must be Web accessible, the Java source file need not be.

Core Note

`CODE` cannot be used to give an absolute URL. Use `CODEBASE` if you want to load applets from someplace other than the current document's location.

WIDTH and HEIGHT `WIDTH` and `HEIGHT` specify the area the applet will occupy. They can be specified in pixels or as a percentage of the browser window width. However, appletviewer cannot handle percentages because there is no preexisting window for the percentage to refer to. These attributes are required in all applets.

Core Alert

Sun's appletviewer cannot handle `WIDTH` and `HEIGHT` attributes as percentages.

CODEBASE This attribute designates the base URL. The entry in `CODE` is taken with respect to this directory. The default behavior is to use the directory from which the main HTML document originated.

ALT Java-enabled browsers ignore markup between `<APPLET ...>` and `</APPLET>`, so alternative text for these browsers is normally placed there. The `ALT` attribute was

intended for browsers that have Java disabled. The attribute is not widely supported. We recommend avoiding `ALT`.

ALIGN This attribute specifies alignment options and has the same possible values (`LEFT`, `RIGHT`, `TOP`, `BOTTOM`, `MIDDLE`) and interpretation of values as the `IMG` element (see Section 3.4, "Embedded Images").

HSPACE `HSPACE` specifies the empty space at the left and right of the applet (in pixels).

VSPACE `VSPACE` specifies the empty space at the top and bottom of the applet (in pixels).

NAME `NAME` gives a name to the applet. It is used in the Java programming language for interapplet communication and by JavaScript to reference an applet by name instead of using an index in the applet array. However, a bug in Netscape prevents recognition of applets that contain uppercase characters in their names. So, if you want two applets to talk to each other, use lowercase names.

Core Approach

If you supply a `NAME` for the applet, use all lowercase letters.

OBJECT The `OBJECT` attribute can be used to supply a serialized applet that was saved with Java's object serialization facility.

ARCHIVE `ARCHIVE` specifies an archive of class files to be preloaded. The archive should be in a Java ARchive (`.jar`) format, but Netscape 3.01 allows *uncompressed* Zip (`.zip`) archives. In addition to class files, you can also include images used by the applet in a JAR file. Often, the JAR filename is chosen to match the applet classname, but this approach is not required. For example, if the applet `JavaMan1.class` required the image `JavaMan.gif`, you could compress the two files into a JAR file named `JavaMan.jar` and specify the JAR file in the `APPLET` container, as in

```
<APPLET CODE="JavaMan1.class"
        ARCHIVE="JavaMan.jar"
        WIDTH=375 HEIGHT=370>
  <B>Sorry, you have a Java-challenged browser.</B>
</APPLET>
```

See Section 7.10 (Packages, Classpath, and JAR Archives) for information on creating JAR files.

MAYSCRIPT Netscape and Internet Explorer use this attribute to determine if JavaScript is permitted to control the applet.

9.7 Reading Applet Parameters

HTML Element:	**`<PARAM id="..." VALUE="...">`**
	(No End Tag)
Attributes:	`NAME` (required), `VALUE` (required)

An applet does not receive the `String[]` argument list that applications get in the `main` method. However, you can customize an applet by supplying information inside `PARAM` tags located between `<APPLET ...>` and `</APPLET>`. These parameters are declared as follows:

```
<PARAM id="Parameter Name" VALUE="Parameter Value">
```

The parameters are read from within an applet by `getParameter ("Parameter Name")`, which returns `"Parameter Value"` as a `String`, or `null` if the parameter is not found. Note that `getParameter` is case sensitive, but as with HTML in general, the `PARAM`, `NAME`, and `VALUE` element and attribute names themselves are case insensitive. Note also that strings should not be compared with `==`, because `==` simply checks whether the two strings are the same object. Use the `equals` (case sensitive) or `equalsIgnoreCase` (case insensitive) method of `String` for this.

Although the return value of `getParameter` is always a `String`, you can convert it into an `int` by using the static `parseInt` method of the `Integer` class. Section 9.10 (Graphical Applications) gives an example of `Integer.parseInt` and lists methods to convert strings to bytes, shorts, longs, floats, and doubles.

Finally, be aware that `PARAM` names of `WIDTH` or `HEIGHT` override the `WIDTH` or `HEIGHT` values supplied in the `APPLET` tag itself and should be avoided.

Core Alert

Never use `WIDTH` or `HEIGHT` as `PARAM` names.

Reading Applet Parameters:[?lb]An Example

Listing 9.5 gives a variation of the `HelloWWW` applet (Section 6.5) that allows the applet to be customized in the HTML document by a `PARAM` entry of the form

```
<PARAM id="BACKGROUND" VALUE="LIGHT">
```

or

```
<PARAM id="BACKGROUND" VALUE="DARK">
```

Note the check to see if the `backgroundType` is `null`, which would happen if the `PARAM` entry was missing or had a `NAME` other than `"BACKGROUND"` in all uppercase. If this test was not performed and the value was `null`, the `backgroundType.equals(...)` call would crash since `null` does not have an `equals` method (or any other method, for that matter). This could be avoided by

```
if ("LIGHT".equals(backgroundType))
```

instead of

```
if (backgroundType.equals("LIGHT"))
```

but many authors prefer to have an explicit test for `null`.

Core Approach

If you read applet parameters, be sure you handle the case when the parameter is not found

Listing 9.6 shows an HTML document that loads the same applet three different times with different

configuration parameters. Figure 9-3 shows the result in Netscape 4.08 on Windows 98.

Figure 9-3. The PARAM element can be used in HTML documents to pass customizations parameters to applets.

Listing 9.5 `HelloWWW2.java`

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

public class HelloWWW2 extends Applet {
  public void init() {
    setFont(new Font("SansSerif", Font.BOLD, 30));
    Color background = Color.gray;
    Color foreground = Color.darkGray;
    String backgroundType = getParameter("BACKGROUND");
    if (backgroundType != null) {
      if (backgroundType.equalsIgnoreCase("LIGHT")) {
        background = Color.white;
        foreground = Color.black;
      } else if (backgroundType.equalsIgnoreCase("DARK")) {
        background = Color.black;
        foreground = Color.white;
      }
    }
    setBackground(background);
    setForeground(foreground);
  }

  public void paint(Graphics g) {
    g.drawString("Hello, World Wide Web.", 5, 35);
  }
}
```

Listing 9.6 `HelloWWW2.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Customizable HelloWWW Applet</TITLE>
</HEAD>
<BODY>
<H1>Customizable HelloWWW Applet</H1>
<P>
<APPLET CODE="HelloWWW2.class" WIDTH=400 HEIGHT=40>
  <PARAM id="BACKGROUND" VALUE="LIGHT">
  <B>Error! You must use a Java-enabled browser.</B>
</APPLET>
<P>
<APPLET CODE="HelloWWW2.class" WIDTH=400 HEIGHT=40>
  <PARAM id="BACKGROUND" VALUE="DARK">
  <B>Error! You must use a Java-enabled browser.</B>
</APPLET>
<P>
<APPLET CODE="HelloWWW2.class" WIDTH=400 HEIGHT=40>
  <B>Error! You must use a Java-enabled browser.</B>
</APPLET>
</BODY>
</HTML>
```

9.8 HTML OBJECT Element

HTML Element:	**<OBJECT CLASSid="..." ... > ... </OBJECT>**
Attributes:	CLASSID, CODETYPE, CODEBASE, STANDBY, WIDTH, HEIGHT, NAME, ALIGN, HSPACE, VSPACE

Surprisingly, the HTML 4.0 specification deprecated the APPLET element in favor of the more universal element, OBJECT. Even though the OBJECT element can accommodate many types of objects, for example, ActiveX, Applet, QuickTime, many authors still prefer the simpler APPLET element in their HTML documents. Of course, the APPLET element forces a *Transitional* DOCTYPE declaration at the beginning of the HTML file, whereas the OBJECT element does not.

Technically, 32 attributes are defined for the OBJECT element in HTML 4.0, and browsers still support an additional 6 deprecated attributes for basic spacing and layout. Only those OBJECT attributes commonly used for an applet are presented here. The OBJECT element is fully described in Section 3.6 (Embedding Other Objects in Documents).

CLASSID This specifies the URL. For applets, CLASSID is of the form `java:Applet.class`.

CODETYPE CODETYPE defines the content type of the object to download. For applets, the form is `CODETYPE="application/java"`.

CODEBASE The CODEBASE specifies the base URL of the object (applet).

STANDBY This attribute provides a string to be displayed while the object is loading.

WIDTH, HEIGHT, NAME, ALIGN, HSPACE, and VSPACE These attributes are used exactly the same way as they are for the APPLET element. See Section 9.6 (The HTML

APPLET Element). Technically, ALIGN, HSPACE, and VSPACE are deprecated in HTML 4.0 in favor; see Chapter 5.

For specifying an applet with the OBJECT element, provide the CODETYPE to signify that the content type is Java and provide the CLASSID attribute to signify the URL of the class file. If the applet is located in a different directory from that of the HTML document, then add the CODEBASE attribute. An example applet using the OBJECT element is shown in Listing 9.7.

Listing 9.7 HelloWWWObject.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>
  <TITLE>A HelloWWW Object</TITLE>
</HEAD>

<BODY>
<H1>A HelloWWW Object</H1>

<OBJECT  CODETYPE="application/java"
         CLASSid="java:HelloWWW.class"
         CODEBASE="applets"
         WIDTH=400 HEIGHT=55>
  <PARAM id="codebase" value="applets">
  <PARAM id="code" value="HelloWWW.class">
  <PARAM id="BACKGROUND" VALUE="DARK">
  <B>Error! You must use a Java-enabled browser.</B>
</OBJECT>

</BODY>
</HTML>
```

Internet Explorer and appletviewer do not properly recognize the CODEBASE attribute if the class file is located in a different directory from that of the HTML file; thus, a common practice is to add a "codebase" PARAM element in the OBJECT declaration. In addition, appletviewer expects to see a CODE attribute to determine which class file to load. To resolve this minor discrepancy, simply add code as a named parameter in the OBJECT element whose value mirrors the Java class filename specified in the CLASSID attribute, for example,

```
<PARAM id="code" value="HelloWWW.class">
```

Core Approach

Add named code *and* codebase *parameter elements,* <PARAM id=...> *to the* OBJECT *container for proper loading of the applet by both Internet Explorer and appletviewer.*

9.9 The Java Plug-In

The Java Plug-In (at http://java.sun.com/products/plugin/), once installed, allows browsers to run the most current version of the Java Runtime Environment (JRE). This permits you to deliver applets designed with the latest version of the Java Platform that can run on either Netscape and Internet Explorer. Reliance on the vendors to include the latest Java release in their browser is no longer required. However, this capability comes at a price.

First, the Java 2 Plug-In is a hefty 5 Mbytes in size. Downloading the plug-in for your "state of the art" applet is not a viable solution for an *internet* client, especially one sitting on the other end of a V.90 (56.6K) or slower modem. However, in an *intranet* setting, the situation is different, and the Java Plug-In is a viable solution. Simply place the plug-in on a server located in the Local Area Network (LAN) for automatic downloading by *intranet* clients.

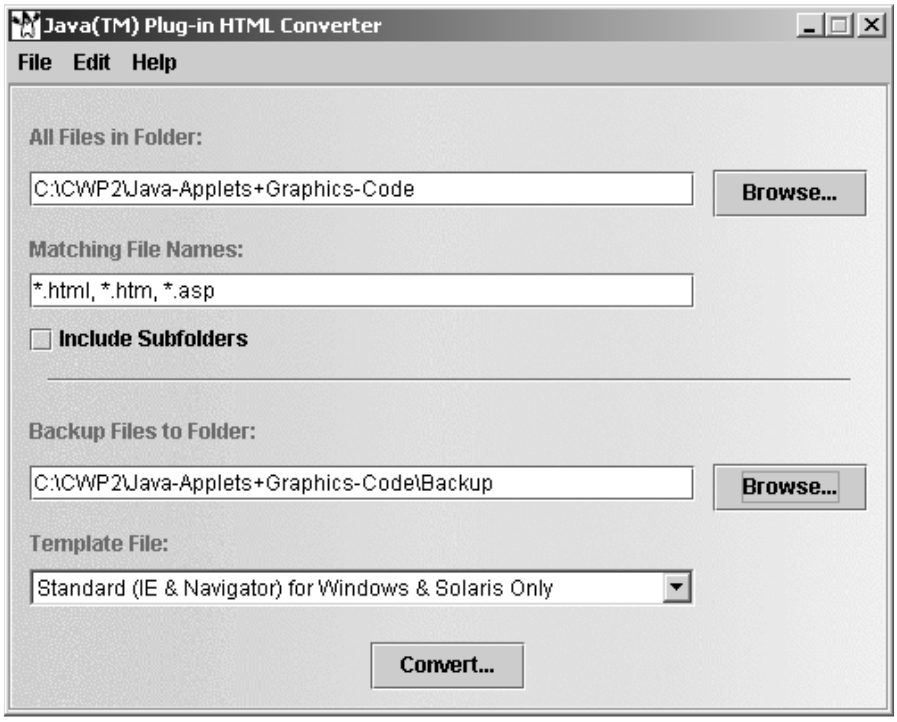
Core Approach

Avoid the Java Plug-In for Internet-based applets. The Java Plug-In is only suitable for intranet based-applets where the plug-in can easily be downloaded from a local server.

Second, the process to invoke the Java Plug-In is different on Internet Explorer than on Netscape. Internet Explorer requires the `APPLET` to be converted to an `OBJECT` element, whereas Netscape requires the `APPLET` to be converted to an `EMBED` element. Sun provides an HTML converter, located at http://java.sun.com/products/plugin/1.3/features.html, to automatically perform the HTML conversion of your `applet`.

The Java Plug-In HTML converter, shown in Figure 9-4, is simply a Java program, `HTMLConverter`, that provides multiple templates for targeting various platforms and browsers when converting applets in selected HTML files. Available conversion options in Version 1.3 of the Java Plug-In HTML Converter include:

Figure 9-4. Java Plug-In HTML Converter, Version 1.3.



- Standard (IE & Navigator) for Windows and Solaris only
- Extended (standard + all browsers/platforms)
- Internet Explorer for Windows and Solaris only
- Navigator for Windows only

Or, you can define your own template for converting the applet.

Based on the backup directory specified, the converter first creates a backup copy of the HTML file, then parses the document, converting each APPLET element to an OBJECT element, an EMBED element, or both, depending on the selected template and browser target. Listing 9.8 shows a typical APPLET prior to conversion by the Java Plug-In HTML Converter. The resultant APPLET container is shown in Listing 9.9 for a "Navigator for Windows Only" target, and in Listing 9.10 a "Internet Explorer for Windows & Solaris Only" target is shown.

Listing 9.8 Applet prior to conversion

```
<APPLET CODE="HelloWWW.class" CODEBASE="applets"
        WIDTH=400 HEIGHT=40>
   <PARAM id="BACKGROUND" VALUE="DARK">
   <B>Error! You must use a Java-enabled browser.</B>
</APPLET>
```

Listing 9.9 Applet conversion for "Navigator for Windows Only"

```
<EMBED type="application/x-java-applet;version=1.3"
   CODE = "HelloWWW.class" CODEBASE = "applets"
   WIDTH = 400 HEIGHT = 40
   BACKGROUND = "LIGHT"
   scriptable=false
   pluginspage="http://java.sun.com/products/plugin/1.3/
       plugin-install.html"
>
   <NOEMBED>
     <B>Error! You must use a Java-enabled browser.</B>
   </NOEMBED>
</EMBED>
```

Listing 9.10 Applet conversion for "Internet Explorer for Windows and Solaris Only"

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
   WIDTH = 400 HEIGHT = 40
   codebase="http://java.sun.com/products/plugin/1.3/
       jinstall-13-win32.cab#Version=1,3,0,0"
>
   <PARAM NAME = CODE VALUE = "HelloWWW.class" >
   <PARAM NAME = CODEBASE VALUE = "applets" >
   <PARAM id="type"
          VALUE="application/x-java-applet;version=1.3">
   <PARAM id="scriptable" VALUE="false">
   <PARAM NAME = "BACKGROUND" VALUE ="LIGHT">
   <B>Error! You must use a Java-enabled browser.</B>
</OBJECT>
```

Examination of the three listings shows that the HTML Converter adds a link pointing to the Java Plug-In for downloading if the plug-in is not already installed on the client browser. If the plug-in is not installed, the client is prompted to download the installation file. Typically, in an intranet setting, the required plug-in files are first downloaded from Sun and then placed on a server in the LAN where greater bandwidth is available. After the install point on the intranet is established for the plug-in, the link in the APPLET container is modified accordingly to point to the new location.

9.10 Graphical Applications

The previous examples used applets: Java programs that run within a Web browser. Local Java programs can use windows as well. Stand-alone graphical Java programs start with a Java `JFrame`, which is a heavyweight Swing component. Applications differ significantly from applets, in the sense that applets should be based on AWT components because most browsers do not fully support the new Swing components unless the Java Plug-In (covered in Section 9.9) is installed or Swing classes are supplied over the network.

On the other hand, Java applications run as a stand-alone instance of the Java Virtual Machine on the client workstation. Thus, you can assume that the client has a version of the JVM that does support Swing components, and in this regard, Swing components, which are truly platform-to-platform independent, should be used for all Java applications. Chapter 14 (Basic Swing) describes Swing components in much greater detail. For now, we only briefly discuss the topic. What is important throughout the remaining sections of this chapter are the graphical and image loading techniques presented.

The key steps in creating a `JFrame` for a Java application are to add a title through the class constructor, specify the width and height through `setSize`, and then pop up the frame by a call to `setVisible`. The following presents a basic template for creating an application frame,

```
public class MyFrame extends JFrame {
  JFrame frame;
  ...
  public static void main(String[] args) {
    frame = new MyFrame("title");
    ...
    frame.addWindowListener(new ExitListener());
    frame.setSize(width, height);
    frame.setVisible(true);
}
```

One of the surprising things about frames is that the users cannot quit the (parent) frame unless you explicitly put in code to let them do so (child frames are closable). That's the purpose of adding the `ExitListener` object attached to the frame. The `ExitListener` class simply calls `System.exit(0)` on a window closing event. For example,

```
public class ExitListener extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
```

Chapter 14 (Basic Swing) provides additional examples of creating Java applications. In Chapter 14, a utility class, `WindowUtilities`, provides numerous helper methods to simplify the task of creating windows. See Listing 14.1 and Listing 14.2 in Section 14.1 (Getting Started with Swing) for details.

9.11 Graphics Operations

In traditional applets and applications, the `paint` method is used to implement custom drawing. It takes a `Graphics` object as an argument; this object is used to draw onto the window. Methods outside `paint` can obtain the current `Graphics` object by calling `getGraphics`. Note that it is not reliable to simply call `getGraphics` once and store the `Graphics` object in an instance variable because subsequent invocations of `paint` get new versions. However, it *is* reliable to pass the `Graphics` object to other methods that draw and then return before `paint` returns.

Similarly, in newer *Swing* applets and applications, the `Graphics` object provides a means to perform *simple* drawing. In Swing, the `paintComponent` method, not `paint`, is used to perform

basic drawing. The `paintComponent` method is only available in lightweight Swing components, which `JApplet` and `JFrame` are not. Thus, to perform drawing in Swing, a lightweight component, most often a `JPanel`, is always added to the `JApplet` or `JFrame`, and the drawing is performed in the lightweight component. Chapter 10 (Java 2D: Graphics in Java 2) and Chapter 14 (Basic Swing) provide more in-depth coverage of graphics drawing in applets and applications.

The drawing methods discussed next can be used in windows such as `Panel`, `Canvas`, `Frame`, and so forth, in addition to `Applet`. These other components are discussed in Chapter 13 (AWT Components). In addition, these drawing methods also work in lightweight Swing components, for example, a `JPanel`, but for drawing in Java 2, the use of Java 2D graphics is the recommended approach.

Java does not supply any method to determine the absolute location of an applet in the browser window, although you can discover the location of a frame. In any case, all coordinates in the following methods are relative, not absolute, and are interpreted with respect to (0,0) being the top-left corner of the window, with x increasing to the right and y increasing downward. As with many graphical systems, Java coordinates are considered to be between the screen pixels. Operations that draw the outline of figures draw the pixels down and to the right of the coordinates, and operations that fill a figure fill the interior of the coordinates. This means that drawing the outline of a rectangle will take one extra pixel on the bottom and right sides compared to filling the same rectangle.

The AWT `Graphics` object only supports simple drawing and does not support pen widths (line thicknesses) or fill patterns. However, the newer Java 2 Platform added the Java 2D API, a greatly improved graphics package based on the `Graphics2D` object; Java 2D includes pen widths, stroke styles (dashed, dotted, etc.), fill patterns, antialiasing, much-improved font support, and much more. For additional information on Java 2D, see Chapter 10 or http://java.sun.com/products/java-media/2D/.

Core Approach

For simple drawing in applets, use the `Graphics` object. For advanced, professional-quality graphics in applets, use the Java 2D `Graphics2D` object (which requires the Java Plug-In).

The following subsections summarize the methods supported by `Graphics`.

Drawing Operations

public void clearRect(int left, int top, int width, int height)

The `clearRect` method draws a solid rectangle in the current background color.

public void copyArea(int left, int top, int width, int height, int deltaX, int deltaY)

This method copies all pixels from the rectangle defined by (left, top, width, height) to (left+deltaX, top+deltaY, width, height).

public Graphics create()

public Graphics create(int left, int top, int width, int height)

This method creates a new graphics context. If a rectangle is specified, the context is translated to the designated location and its clipping region is set to the specified width and height.

public void draw3DRect(int left, int top, int width, int height, boolean raised)

This method draws a 1-pixel-wide outline around the specified rectangle. If `raised` is

`true`, then the left and top edges will be lighter, giving the appearance of the rectangle being above the surface of the window. If `raised` is `false`, then the rectangle is drawn with the top and left edges darker, giving the appearance of an indented rectangle. In most cases, it is a good idea to set the foreground color to be the same as the background color before calling this method, so that the shading calculation is based on the background.

public void fill3DRect(int left, int top, int width, int height, boolean raised)

This method makes a solid rectangle with a 3D outline.

public void drawArc(int left, int top, int width, int height, int startAngle, int deltaAngle)

This method draws a curve taken from a portion of the outside of an oval. The first four parameters specify the bounding rectangle for an oval. The angles specify what part of the oval will be drawn; 0 means east (3 o'clock), and angles go counterclockwise. Unlike the trigonometric functions in the `Math` class, angles are in degrees, not radians.

public void fillArc(int left, int top, int width, int height, int startAngle, int deltaAngle)

This method draws a solid "pie wedge" from an oval pie. See `drawArc`.

public void drawImage(Image image, int left, int top, ImageObserver observer)

This method draws an image in its original size. Create the image with the `getImage` method of `Applet` or `Toolkit`, but note that `getImage` operates asynchronously, so calls to `drawImage` immediately after `getImage` may draw blank images. Pass the applet or window (with `this`) as the argument for observer. See Section 9.12 for details on using images.

public void drawImage(Image image, int left, int top, int width, int height, ImageObserver observer)

This method draws an image scaled to fit in the rectangle defined by (left, top, width, height).

public void drawImage(Image image, int left, int top, Color bgColor, ImageObserver observer)

public void drawImage(Image image, int left, int top, int width, int height, Color bgColor, ImageObserver observer)

These methods are variations of the two previous methods for transparent images. The specified background color is used for transparent pixels.

public void drawLine(int x1, int y1, int x2, int y2)

This method draws a 1-pixel-thick line.

public void drawOval(int left, int top, int width, int height)

This method draws the outline of an oval. Arguments describe the rectangle that contains the oval. For example, `drawOval(75, 75, 50, 50)` specifies a circle of radius 50 centered at (100, 100).

public void fillOval(int left, int top, int width, int height)

This method draws a solid oval bounded by the specified rectangle.

public void drawPolygon(int[ ] xArray, int[ ] yArray, int numPoints)

public void drawPolygon(Polygon polygon)

These methods draw the outline of a polygon defined by the arrays or `Polygon` (a class that stores a series of points). The polygon is not closed by default. To make a closed polygon, specify the same location for the first and last points.

public void fillPolygon(int[ ] xArray, int[ ] yArray, int numPoints)

public void fillPolygon(Polygon polygon)

This method draws a solid polygon. The polygon is closed by default; a connection is automatically made between the first and last points.

public void drawRect(int left, int top, int width, int height)

This method draws the outline of a rectangle (1-pixel border) in the current color. See `draw3DRect` and `drawRoundRect` for variations on the theme.

public void fillRect(int left, int top, int width, int height)

This method draws a solid rectangle in the current color. The current AWT has no provision for setting fill patterns or images, so that filling would have to be reproduced manually. See also `fill3DRect` and `fillRoundRect`.

public void drawRoundRect(int left, int top, int width, int height, int arcWidth, int arcHeight)

This method draws the outline of a rectangle with rounded corners. The `arcWidth` and `arcHeight` parameters specify the amount of curve (in degrees) on the top/bottom and left/right sides. If either is zero, square corners are used.

public void drawString(String string, int left, int bottom)

This method draws a string in the current font and color with the *bottom*-left corner at the specified location. This is one of the few methods where the y coordinate refers to the bottom, not the top. There are also `drawChars` and `drawBytes` methods that take arrays of `char` or `byte`.

Colors and Fonts

public Color getColor()

This method returns the current `Color`. For more information on using custom and built-in colors, see the discussion of `getBackground` and `setBackground` in Section 9.5 (Other Applet Methods).

public void setColor(Color color)

This method sets the foreground color. When the `Graphics` object is created, the default drawing color is the foreground color of the window. Color changes made by calling `setColor` on the `Graphics` object do not change the default, so the next time `paint` or `getGraphics` is called, the new `Graphics` is reinitialized with the window defaults. You record permanent changes by calling the applet's or frame's `setForeground` method, but this call only affects drawing done with `Graphics` objects created *after* the call to `setForeground`.

public Font getFont()

This method returns the current `Font`. See the discussion of `getFont` and `setFont` in Section 9.5 (Other Applet Methods) for more information on fonts. Both `Component` (and thus `Applet`, which inherits from it) and `Graphics` have a `getFontMetrics` method that takes a `Font` as an argument. This `FontMetrics` object can then be used to find out the size of characters (`charWidth`) and strings (`stringWidth`) in that font.

public void setFont(Font font)

This method sets the font to be used by the `drawString` method. The font changes specified by the `setFont` method of the `Graphics` object do not persist to the next invocation of `paint` or to the next time `getGraphics` is called. Permanent font changes can be specified with the `setFont` method of the applet or other associated component.

Drawing Modes

public void setXORMode(Color color)

This method specifies that subsequent drawing operations will use XOR: the color of each pixel in the result will be determined by bitwise XORing the specified color with the color of the pixel at the location being drawn. Thus, a line drawn in XOR mode over a multicolor background will be in multiple colors. The resultant color at each pixel is unpredictable, since the XOR is done on the bits as they appear in the internal representation, which may vary from machine to machine. But drawing something using XOR twice in a row will return it to the original condition. This is useful for rubberbanding or other short-term erasable drawing done on top of some more complex drawing.

You should avoid using `Color.black` as the specified color, since it will be represented internally by all zeros on many (but not all) platforms, so the XOR results in the original color and your drawing will be invisible. Set the drawing mode back to normal with `setPaintMode()`.

public void setPaintMode()

This method sets the drawing mode back to normal (vs. XOR) mode. That is, drawing will use the normal foreground color only.

Coordinates and Clipping Rectangles

public void clipRect(int left, int top, int width, int height)

This method shrinks the clipping region to the intersection of the current clipping region and the specified rectangle.

public Rectangle getClipBounds()

This method returns the current clipping rectangle, which may be `null`.

public Shape getClip()

This method returns a `Shape` object describing the clipping region.

pubic void setClip(Shape clippingRegion).

This method designates a new clipping region.

public void translate(int deltaX, int deltaY)

This method moves the origin by the specified amount.

9.12 Drawing Images

Applets and applications written in the Java programming language can load and display static images in GIF or JPEG format, as well as GIF89A images (a.k.a. "animated GIF").

Image drawing is done in two steps. First, a remote or local image is registered by means of the `getImage` method of `Applet` or `Toolkit`. Second, the image is drawn on the screen with the `drawImage` method of `Graphics`. You can draw the image at its regular size or supply an explicit width and height. The key point to remember is that calls to `getImage` don't actually initiate image loading. Instead, Java technology doesn't start loading the image until it is needed.

Actual loading of the image is done in a background thread, and the image can be drawn incrementally while loading. Instead of waiting until you try to draw the image, you can specify that the image be loaded in advance by calling `prepareImage` or using a `MediaTracker` object. The first approach (`prepareImage`) loads the image in the background, returning control to you immediately. This behavior is normally an advantage because processing can continue while the program might otherwise be waiting for a slow network connection. However, if `drawImage` is called before the image is done loading, it just draws the portion that has arrived (possibly none) without giving any error messages. The `paint` method (or the `paintComponent` method in the case of lightweight Swing components) will get called once the image is done, so assuming `drawImage` is being invoked from `paint`, the image will eventually be drawn in its entirety. In the meantime, however, partial images may be drawn and the width and height of the image may be incorrect. If you want to be sure the image is finished before you do any drawing, you can use the second approach: the `MediaTracker` class.

Loading Applet Images from Relative URLs

The `Applet` class contains a `getImage` method that takes two arguments: a URL corresponding to a directory and a string corresponding to a filename relative to that URL. For the relative URLs, supply `getCodeBase()` (the applet's home directory) or `getDocumentBase()` (the Web page's home directory) for the URL argument. The `getImage` will not succeed until the applet's context is set up. This means that you should call `getImage` in `init` rather than trying to directly initialize the `Image` instance variable by:

```
private Image myImage = getImage(...); // fails
```

Core Warning

Trying to declare and initialize `Image` instance variables in the body of an applet will fail. Initialize them in `init` or a method that runs after `init`.

To actually draw the image, use the `drawImage` of the `Graphics` class. If you're using a method other than `paint` (which is automatically passed the current `Graphics` context), you can obtain the window's `Graphics` context by calling `getGraphics`. There are two variations of `drawImage`:

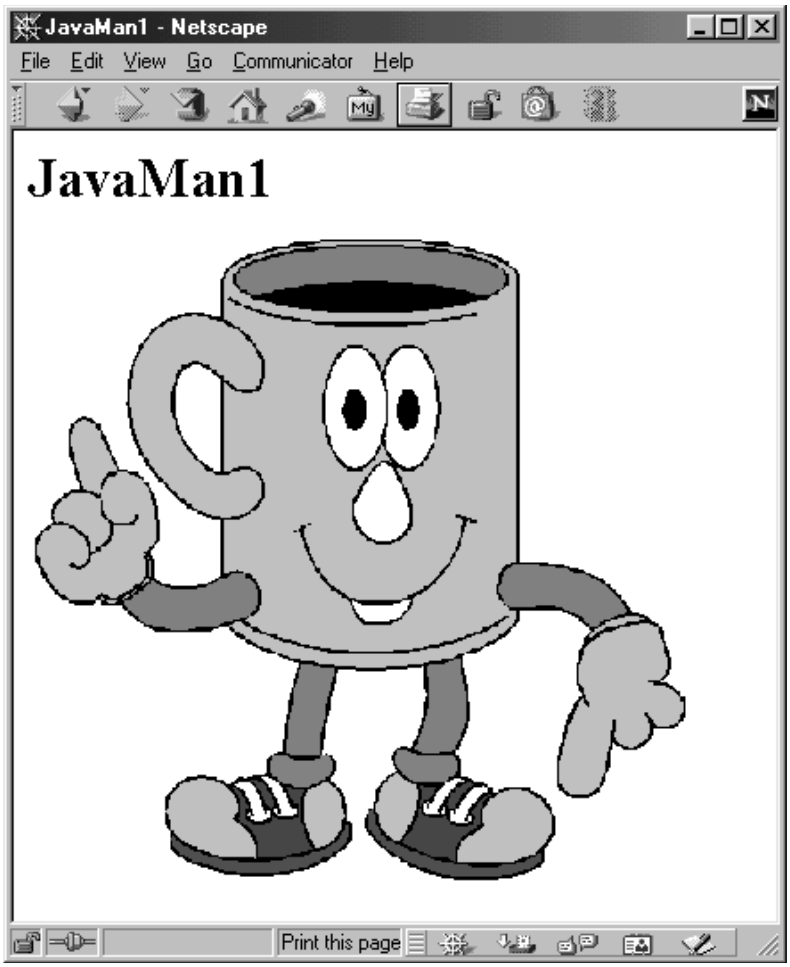
```
drawImage(image, left, top, window)
```

and

```
drawImage(image, left, top, width, height, window)
```

The first uses the image's normal size; the second stretches the image to fit in the specified area. Technically, the last argument is an `ImageObserver`; in ordinary cases, you just use the current window (the applet in this case). So, for image drawing being performed in the paint method, `this` is almost always used as the last argument to `drawImage`. Listings 9.11 and 9.12 show an applet that loads images from the `images` subdirectory of the applet's home directory. The result is shown in Figure 9-5.

Figure 9-5. The most common way to load images in applets is to use `getImage(getCodeBase(), path)`, or `getImage(getDocumentBase(), path)`.



Listing 9.11 `JavaMan1.java`

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

/** An applet that loads an image from a relative URL. */

public class JavaMan1 extends Applet {
  private Image javaMan;

  public void init() {
    javaMan = getImage(getCodeBase(),"images/Java-Man.gif");
  }

  public void paint(Graphics g) {
    g.drawImage(javaMan, 0, 0, this);
```

```
  }
}
```

Listing 9.12 `JavaMan1.html`

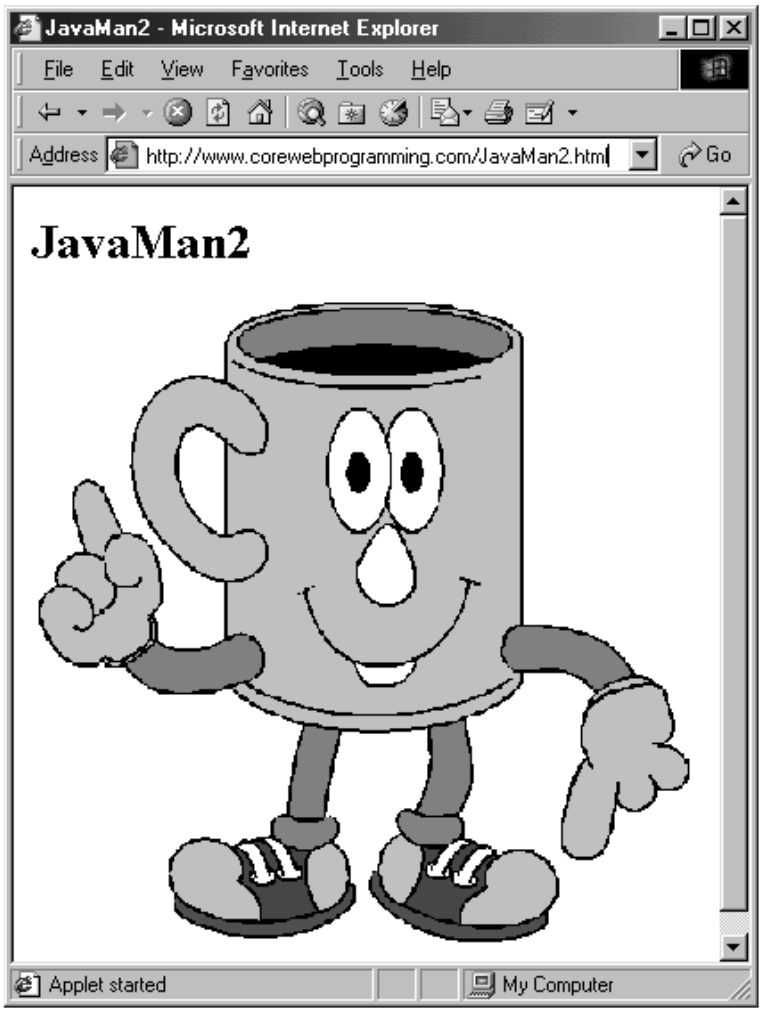
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>JavaMan1</TITLE>
</HEAD>
<BODY BGCOLOR="WHITE">
<H1>JavaMan1</H1>

<APPLET CODE="JavaMan1.class" WIDTH=370 HEIGHT=365>
  <B>Sorry, you have a Java-challenged browser.</B>
</APPLET>

</BODY>
</HTML>
```

Loading Applet Images from Absolute URLs

Using absolute URLs is a bit more cumbersome than using relative ones because making a `URL` object requires catching the exception that would result if the URL is in an illegal format. See Section 8.12 (Exceptions) for a review of handling exceptions. Furthermore, because the `SecurityManager` of most browsers only allows an applet to load images from the machine that served the applet, this approach is not particularly common. However, it is quite possible that you store images in a different location than that of the applets and their associated HTML documents, so using an absolute URL would be more convenient.

Listings 9.13 and 9.14 give an example applet and associated HTML document, with the result shown in Figure 9-6. Note the `try`/`catch` block around the URL constructor, and note, too, that the `java.net` package is imported: it contains the `URL` and `MalformedURLException` classes. Using this approach does not change the fact that the image loading is postponed until the image is needed, so the image might appear to flicker into view as progressively larger pieces are drawn. If this is a problem for your application, see the next sections on how to partially or completely preload the image.

Figure 9-6. Applets can load images from absolute URLs, but security restrictions apply.



Listing 9.13 JavaMan2.java

```
import java.applet.Applet;
import java.awt.*;
import java.net.*;

/** An applet that loads an image from an absolute
 *  URL on the same machine that the applet came from.
 */

public class JavaMan2 extends Applet {
  private Image javaMan;

  public void init() {
    try {
      URL imageFile = new URL("http://www.corewebprogramming.com" +
                              "/images/Java-Man.gif");
      javaMan = getImage(imageFile);
    } catch(MalformedURLException mue) {
      showStatus("Bogus image URL.");
      System.out.println("Bogus URL");
    }
  }
```

```
  public void paint(Graphics g) {
    g.drawImage(javaMan, 0, 0, this);
  }
}
```

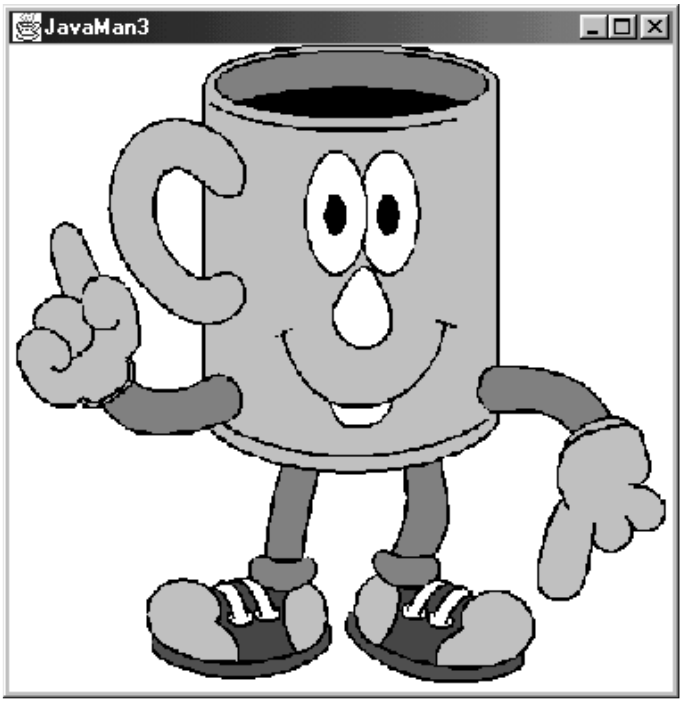
Listing 9.14 `JavaMan2.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>JavaMan2</TITLE>
</HEAD>
<BODY BGCOLOR="WHITE">
<H1>JavaMan2</H1>

<APPLET CODE="JavaMan2.class" WIDTH=370 HEIGHT=365>
  <B>Sorry, you have a Java-challenged browser.</B>
</APPLET>

</BODY>
</HTML>
```

Loading Images in Applications

Graphical applications can load images from absolute URLs or from local files by using the `getImage` method of the `Toolkit` class. The concept of a relative URL is not completely applicable because applications are not normally associated with a Web page. Nevertheless, even though applications do not have a two-argument version of `getImage` as applets do, a `URL` object can be created from an existing `URL` and a filename. You can use this technique if, for instance, you want to load multiple images from the same directory. You can obtain the current `Toolkit` from any graphical object by calling `getToolkit()` or from an arbitrary object by calling `Toolkit.getDefaultToolkit()`. For instance, Listing 9.15 creates a simple `JPanel` and then draws an `Image` in the `JPanel`. Note the use of `System.getProperty("user.dir")` to make the filename relative to the directory containing the application. This use makes it easier to move directories around or to move the application from machine to machine. Figure 9-7 shows the result. The source code for `WindowUtilities`, which simply encapsulates the panel in a `JFrame` and attaches a `WindowListener`, is provided in Chapter 14 (Basic Swing).

Figure 9-7. Images can be loaded in applications through `getToolkit().getImage(arg)` or `Toolkit.getDefaultToolkit().getImage(arg)`. The `arg` can be a URL or local filename.



Core Approach

Whenever possible, refer to local files by using pathnames relative to the application's directory.

Listing 9.15 JavaMan3.java

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

/** An application that loads an image from a local file.
 *  Applets are not permitted to do this.
 */

class JavaMan3 extends JPanel {
  private Image javaMan;

  public JavaMan3() {
    String imageFile = System.getProperty("user.dir") +
                       "/images/Java-Man.gif";
    javaMan = getToolkit().getImage(imageFile);
    setBackground(Color.white);
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(javaMan, 0, 0, this);
  }

  public static void main(String[] args) {
    JPanel panel = new JavaMan3();
    WindowUtilities.setNativeLookAndFeel();
```

```
    WindowUtilities.openInJFrame(panel, 380, 390);
  }
}
```

9.13 Preloading Images

In many cases, you'd like the system to start loading the images as soon as possible, rather than waiting until you try to draw them with `drawImage`. This is particularly true if the images will not be drawn until the user initiates some action such as clicking on a button or choosing a menu option. That way, if the user doesn't act right away, the image might arrive before action is taken. You can use the `prepareImage` method to start the image loading in a background process and immediately return control to you. There are two versions of `prepareImage`, one for each version of `drawImage`:

```
prepareImage(image, window)
```

and

```
prepareImage(image, width, height, window)
```

Each time you stretch the image, it counts as a new one, so be sure to call `prepareImage` once for each size at which you plan to draw. For example, Listing 9.16 shows an application that draws an image only when the user presses a button. The time from when the user presses the button to the time when the drawing is completed is printed in a textfield. If a `-preload` command-line argument is supplied, `prepareImage` is called. Figure 9-8 shows the result when `prepareImage` is not used, and Figure 9-9 shows what happens when the same image is loaded, `-preload` is specified, and the button is not clicked until several seconds have gone by. Of course, the time shown in Figure 9-8 could be much smaller or much larger, depending upon the speed of your network connection, but the fact remains that the time before the button is pressed is wasted unless you use `prepareImage`.

Figure 9-8. Results when no preload argument is supplied. If you use `getImage` and `drawImage` only, the image is not loaded over the network until the system tries to draw it.

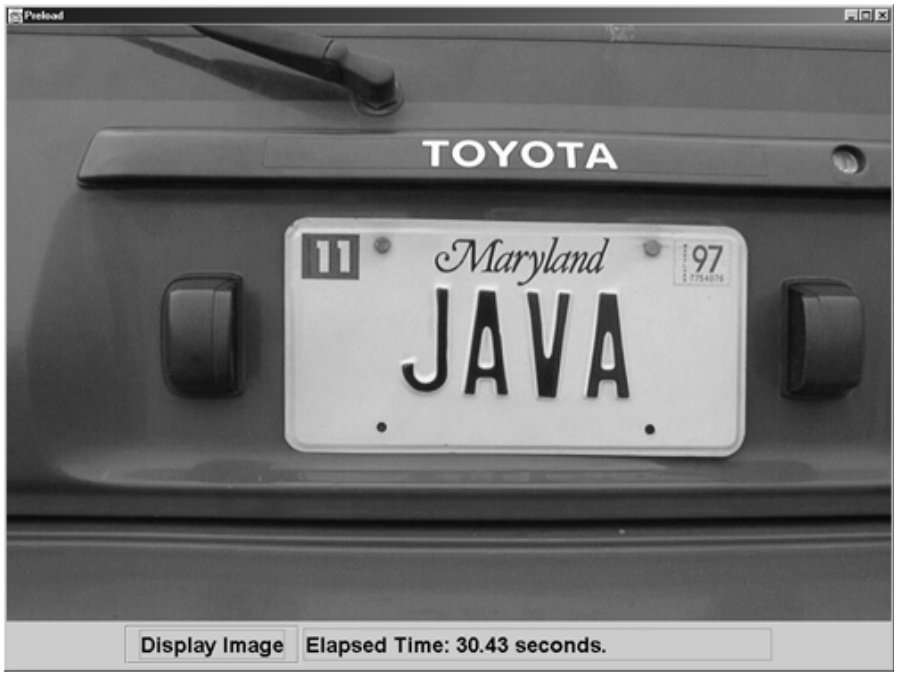
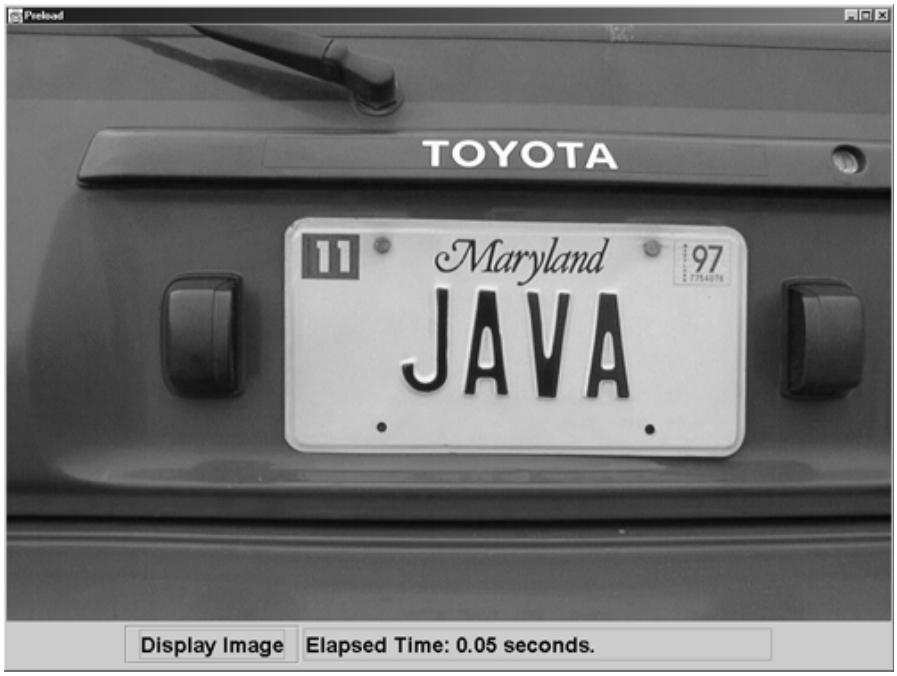


Figure 9-9. Results when a preload argument is supplied. If you use `prepareImage`, the system starts loading the image immediately.



For now, don't worry about the details of Listing 9.16; we'll cover user interfaces at length in the upcoming chapters. Instead, concentrate on what goes on in the `registerImage` method, which is called from the `Preload` constructor.

Listing 9.16 `Preload.java`

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

/** A class that compares the time to draw an image preloaded
 *  (getImage, prepareImage, and drawImage) vs. regularly
 *  (getImage and drawImage).
 *  <P>
 *  The answer you get the regular way is dependent on the
 *  network speed and the size of the image, but if you assume
 *  you load the applet "long" (compared to the time the image
 *  loading requires) before pressing the button, the drawing
 *  time in the preloaded version depends only on the speed of
 *  the local machine.
 */

public class Preload extends JPanel implements ActionListener {

  private JTextField timeField;
  private long start = 0;
  private boolean draw = false;
  private JButton button;
  private Image plate;

  public Preload(String imageFile, boolean preload) {
    setLayout(new BorderLayout());
    button = new JButton("Display Image");
```

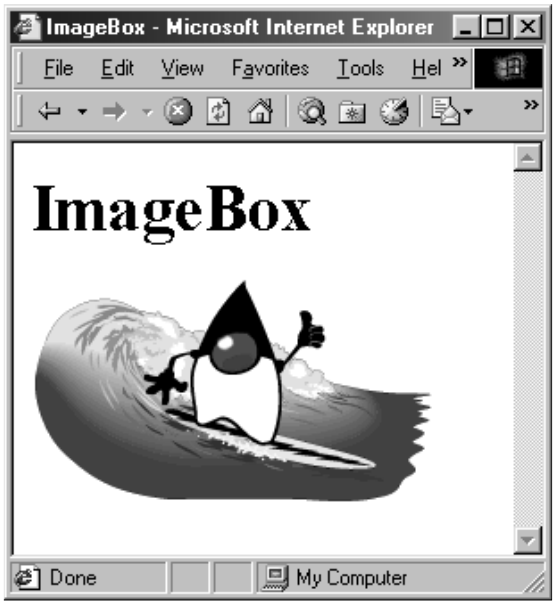
```
    button.setFont(new Font("SansSerif", Font.BOLD, 24));
    button.addActionListener(this);
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(button);
    timeField = new JTextField(25);
    timeField.setEditable(false);
    timeField.setFont(new Font("SansSerif", Font.BOLD, 24));
    buttonPanel.add(timeField);
    add(buttonPanel, BorderLayout.SOUTH);
    registerImage(imageFile, preload);
  }

  /** No need to check which object caused this,
   *  since the button is the only possibility.
   */

  public void actionPerformed(ActionEvent event) {
    draw = true;
    start = System.currentTimeMillis();
    repaint();
  }

  // Do getImage, optionally starting the loading.

  private void registerImage(String imageFile, boolean preload) {
    try {
      plate = getToolkit().getImage(new URL(imageFile));
      if (preload) {
        prepareImage(plate, this);
      }
    } catch(MalformedURLException mue) {
      System.out.println("Bad URL: " + mue);
    }
  }

  /** If button has been clicked, draw image and
   *  show elapsed time. Otherwise, do nothing.
   */

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (draw) {
      g.drawImage(plate, 0, 0, this);
      showTime();
    }
  }

  // Show elapsed time in textfield.
  private void showTime() {
    timeField.setText("Elapsed Time: " + elapsedTime() +
                      " seconds.");
  }
```

```
  // Time in seconds since button was clicked.
  private double elapsedTime() {
    double delta = (double)(System.currentTimeMillis() - start);
    return(delta/1000.0);
  }

  public static void main(String[] args) {
    JPanel preload;

    if (args.length == 0) {
      System.out.println("Must provide URL");
      System.exit(0);
    }
    if (args.length == 2 && args[1].equals("-preload")) {
      preload = new Preload(args[0], true);
    } else {
      preload = new Preload(args[0], false);
    }

    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(preload, 1000, 750);
  }
}
```

9.14 Controlling Image Loading: Waiting for Images and Checking Status

Even if you preload images, you often want to be sure that the images have finished loading before you perform certain tasks. For instance, because you cannot determine an image's width and height until the image has finished loading, programs that try to draw outlines around images must be careful how they go about it. As an example of a common but incorrect approach, consider Listings 9.17 and 9.18, which record the image's width and height in `init`, then draw a rectangle in `paint` based on these dimensions. Figure 9-10 shows the result in Internet Explorer 5 on a Windows 98 system; the rectangle is missing because the height is `-1`.

Figure 9-10. Trying to determine an image's size when you aren't sure it has finished loading can lead to bad results.



Listing 9.17 `ImageBox.java`

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

/** A class that incorrectly tries to load an image and draw an
 *  outline around it. Don't try this at home.
 */

public class ImageBox extends Applet {
  private int imageWidth, imageHeight;
  private Image image;

  public void init() {
    String imageName = getParameter("IMAGE");
    if (imageName != null) {
      image = getImage(getDocumentBase(), imageName);
    } else {
      image = getImage(getDocumentBase(), "error.gif");
    }
    setBackground(Color.white);

    // The following is wrong, since the image won't be done
    // loading, and -1 will be returned.
    imageWidth = image.getWidth(this);
    imageHeight = image.getHeight(this);
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, this);
    g.drawRect(0, 0, imageWidth, imageHeight);
  }
}
```

Listing 9.18 `ImageBox.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>ImageBox</TITLE>
</HEAD>
<BODY>
<H1>ImageBox</H1>
<APPLET CODE="ImageBox.class" WIDTH=235 HEIGHT=135>
  <PARAM id="IMAGE" VALUE="images/surfing.gif">
  Sorry, you need a <B>real</B>browser.
</APPLET>
</BODY>
</HTML>
```

The solution to this problem is to use the `MediaTracker` class. This class lets you start to load one or more images, register them with a `MediaTracker` through `addImage`, then at some point

explicitly wait by calling `waitForID` or `waitForAll` until all the images are loaded. `MediaTracker` also has various methods for checking whether the image file was not found or whether other errors occurred. These methods are summarized below.

MediaTracker

public void addImage(Image image, int id)

public void addImage(Image image, int id, int width, int height)

These methods register a normal or scaled image with a given ID. You can register one or more images with a particular ID, then either check the status of, or wait for, images with a given ID. You can also wait for all images; when you do so, the system tries to load the images with lower IDs first.

public boolean checkAll() public boolean checkAll(boolean startLoading)

These methods return `true` if all the images registered with the `MediaTracker` have finished loading. They return `false` otherwise. If you supply `true` for the `startLoading` argument, the system will begin loading the images if it wasn't doing so already. Note that you should not normally put `CheckAll` inside a loop to wait until images are loaded. Instead, use `waitForAll`, which accomplishes the same goal without consuming nearly as much of the CPU resources.

public boolean checkID(int id)

public boolean checkID(int id, boolean startLoading)

These methods are similar to `checkForAll`, but they only report the status of images registered under a particular ID.

public Object[ ] getErrorsAny()

public Object[ ] getErrorsID(int id)

These methods return an array of images that have encountered an error while loading.

public boolean isErrorAny()

public boolean isErrorID(int id)

These methods return `true` if any image encountered an error while loading; `false` otherwise.

public void removeImage(Image image)

public void removeImage(Image image, int id)

public void removeImage(Image image, int id, int width, int height)

These methods let you "unregister" an image.

public int statusAll() public int statusID(int id, boolean startLoading)

These methods return the bitwise inclusive OR of the status flags of all images being loaded. The status flag options are `MediaTracker.LOADING`, `MediaTracker.ABORTED`, `MediaTracker.ERRORED`, and `MediaTracker.COMPLETE`. Images that haven't started loading have zero for their

status. If you supply `true` for the `startLoading` argument, the system will begin loading the images if it wasn't doing so already.

public void waitForAll()

public boolean waitForAll(long milliseconds)

These methods start loading any images that are not already loading; the methods do not return until all the images are loaded or the specified time has elapsed. The system starts loading images with lower IDs before those with higher ones. The methods throw an `InterruptedException` when done; you are required to catch it.

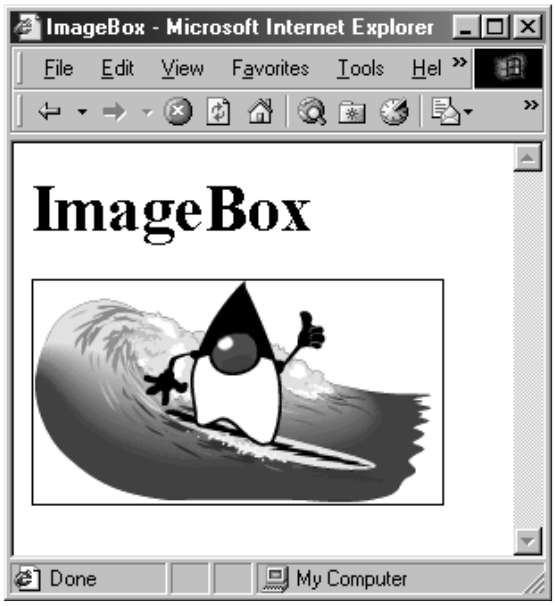
public void waitForID(int id)

public boolean waitForID(int id, long milliseconds)

These methods start loading any images registered under the specified ID that are not already loading; the methods do not return until all images are loaded or the specified time has elapsed. The methods throw an `InterruptedException` when done; you are required to catch it.

Listings 9.19 and 9.20 show a corrected version of the `ImageBox` applet that waits until the image is loaded before trying to determine its size. Figure 9-11 shows the result.

Figure 9-11. With `MediaTracker` you can wait until images are done loading.



Listing 9.19 `BetterImageBox.java`

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

/** This version fixes the problems associated with ImageBox by
 *  using a MediaTracker to be sure the image is loaded before
 *  you try to get its dimensions.
 */

public class BetterImageBox extends Applet {
  private int imageWidth, imageHeight;
```

```
  private Image image;

  public void init() {
    String imageName = getParameter("IMAGE");
    if (imageName != null) {
      image = getImage(getDocumentBase(), imageName);
    } else {
      image = getImage(getDocumentBase(), "error.gif");
    }
    setBackground(Color.white);
    MediaTracker tracker = new MediaTracker(this);
    tracker.addImage(image, 0);
    try {
      tracker.waitForAll();
    } catch(InterruptedException ie) {}
    if (tracker.isErrorAny()) {
      System.out.println("Error while loading image");
    }
    // This is safe: image is fully loaded
    imageWidth = image.getWidth(this);
    imageHeight = image.getHeight(this);
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, this);
    g.drawRect(0, 0, imageWidth, imageHeight);
  }
}
```

Listing 9.20 `BetterImageBox.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>ImageBox</TITLE>
</HEAD>
<BODY>
<H1>ImageBox</H1>
<APPLET CODE="BetterImageBox.class" WIDTH=235 HEIGHT=135>
  <PARAM id="IMAGE" VALUE="images/surfing.gif">
  Sorry, you need a <B>real</B>browser.
</APPLET>
</BODY>
</HTML>
```

Because waiting for images and checking for errors is the most common use of `MediaTracker`, it is convenient to combine these two tasks into a single method. Listing 9.21 defines a `TrackerUtil` class with two static methods: `waitForImage` and `waitForImages`. The `waitForImage` method can be used as follows:

```
someImage = getImage(...);
doSomeOtherStuff();
if (TrackerUtil.waitForImage(someImage, this))
  // someImage finished loading.
```

```
else
  // error loading someImage.
```

Similarly, the `waitForImages` method can be used as follows:

```
image1 = getImage(...);
image2 = getImage(...);
...
imageN = getImage(...);
doSomeOtherStuff();
Image[] images = { image1, image2, ... , imageN };
if (TrackerUtil.waitForImages(images, this))
  // all images finished loading.
else
  // error loading an image.
```

If you want more control over image loading than `MediaTracker` provides, you can override the `imageUpdate` method of the window. See the API for details.

Listing 9.21 `TrackerUtil.java`

```
import java.awt.*;

/** A utility class that lets you load and wait for an image or
 *  images in one fell swoop. If you are loading multiple
 *  images, only use multiple calls to waitForImage if you
 *  <B>need</B> loading to be done serially. Otherwise, use
 *  waitForImages, which loads concurrently, which can be
 *  much faster.
 */

public class TrackerUtil {
  public static boolean waitForImage(Image image, Component c) {
    MediaTracker tracker = new MediaTracker(c);
    tracker.addImage(image, 0);
    try {
      tracker.waitForAll();
    } catch(InterruptedException ie) {}
    if (tracker.isErrorAny()) {
      return(false);
    } else {
      return(true);
    }
  }

  public static boolean waitForImages(Image[] images,
                                      Component c) {
    MediaTracker tracker = new MediaTracker(c);
    for(int i=0; i<images.length; i++) {
      tracker.addImage(images[i], 0);
    }
    try {
      tracker.waitForAll();
    } catch(InterruptedException ie) {}
```

```
    if (tracker.isErrorAny()) {
      return(false);
    } else {
      return(true);
    }
  }
}
```

9.15 Summary

An applet is a type of graphical program that can be embedded in a Web page. Applets run on the client machine and consequently have various security restrictions. Applets are created by extending the `java.applet.Applet` class and are associated with a Web page through the `APPLET` element. Applets based on the newer Swing graphical components are possible; because browser support is lacking, Swing applets require installation of the Java Plug-In or require the Swing classes to be sent over the network, which is not a suitable approach for most Internet clients.

Graphical Java programs that will not be run in a Web browser are created by use of the Swing `JFrame` class. In the case of applets, drawing is typically performed in the `paint` method, whereas in the case of applications, drawing is performed in the `paintComponent` method of a lightweight Swing component, usually a `JPanel` is added to the `JFrame`. Both methods take a `Graphics` object as an argument; the `Graphics` class has a variety of basic drawing operations. But for professional-looking, advanced drawing techniques, including antialiasing, gradients, and textures, use the Java 2D API, which is the topic of the next chapter.

Lastly, another graphics operation of particular interest is `drawImage`, which can be used to draw GIF or JPEG images loaded earlier by the `getImage` method of `Applet` or `Toolkit`. Images loaded this way are loaded in a background thread and can be drawn in incremental pieces unless loading is explicitly controlled by use of a `MediaTracker`. For applets, any images referred to by an absolute or relative URL can only be loaded from the same server on which the class files are located.

CONTENTS