

as move them across the screen. The moving part is a simple for loop between the start and end arguments, setting the x position to the current loop value. Swapping the images means merely testing to see which one is active at any turn of the loop and assigning the other one to the current image. Finally, at each new frame, you'll call `repaint()` and `sleep()` for a bit.

Actually, given that during this animation there will be a lot of sleeping of various intervals, it makes sense to create a method that does the sleeping for the appropriate time interval. Call it `pause`—here's its definition:

```
void pause(int time) {
    try { Thread.sleep(time); }
    catch (InterruptedException e) { }
}
```

Back to the `nekorun()` method. To summarize, `nekorun()` iterates from the start position to the end position. For each turn of the loop, it sets the current x position, sets `currentimg` to the right animation frame, calls `repaint()`, and pauses. Got it? Here's the definition of `nekorun`:

```
void nekorun(int start, int end) {
    for (int i = start; i < end; i+=10) {
        xpos = i;
        // swap images
        if (currentimg == nekopics[0])
            currentimg = nekopics[1];
        else if (currentimg == nekopics[1])
            currentimg = nekopics[0];
        else currentimg = nekopics[0];
        repaint();
        pause(150);
    }
}
```

Note that in that second line you increment the loop by ten pixels. Why ten pixels, and not, say, five or eight? The answer is determined mostly through trial and error to see what looks right. Ten seems to work best for the animation. When you write your own animations, you have to play with both the distances and the sleep times until you get an animation you like.

Speaking of `repaint`, let's cover the `paint()` method, which paints each frame. Here the `paint()` method is trivially simple; all `paint()` is responsible for is painting the current image at the current x and y positions. All that information is stored in instance variables, so the `paint()` method has only a single line in it:

```
public void paint(Graphics g) {
    g.drawImage(currentimg, xpos, ypos, this);
}
```

Now let's back up to the `run()` method, where the main processing of this animation is happening. You've created the `nekorun()` method; in `run` you'll call that method with the appropriate values to make Neko run from the left edge of the screen to the center:

```
// run from one side of the screen to the middle
nekorun(0, size().width / 2);
```

The second major thing Neko does in this animation is stop and yawn. You have a single frame for each of these things (in positions 2 and 3 in the array), so you don't really need a separate method for them. All you need to do is set the appropriate image, call `repaint()`, and pause for the right amount of time. This example pauses for a second each time for both stopping and yawning—again, using trial and error. Here's the code:

```
// stop and pause
currentimg = nekopics[2];
repaint();
pause(1000);

// yawn
currentimg = nekopics[3];
repaint();
pause(1000);
```

Let's move on to the third part of the animation: scratching. There's no horizontal for this part of the animation. You alternate between the two scratching images (stored in positions 4 and 5 of the image array). Because scratching is a distinct action, however, let's create a separate method for it.

The `nekoscratch()` method takes a single argument: the number of times to scratch. With that argument, you can iterate, and then, inside the loop, alternate between the two scratching images and repaint each time:

```
void nekoscratch(int numtimes) {
    for (int i = numtimes; i > 0; i--) {
        currentimg = nekopics[4];
        repaint();
        pause(150);
        currentimg = nekopics[5];
        repaint();
        pause(150);
    }
}
```

Inside the `run` method, you can then call `nekoscratch()` with an argument of (4):

```
// scratch four times
nekoscratch(4);
```

Onward! After scratching, Neko sleeps. Again, you have two images for sleeping (in positions 6 and 7 of the array), which you'll alternate a certain number of times. Here's the `nekosleep()` method, which takes a single number argument, and animates for that many "turns":

```
void nekosleep(int numtimes) {
    for (int i = numtimes; i > 0; i--) {
        currentimg = nekopics[6];
```

```

        repaint();
        pause(250);
        currentimg = nekopics[7];
        repaint();
        pause(250);
    }
}

```

Call `nekosleep` in the `run()` method like this:

```

// sleep for 5 "turns"
nekosleep(5);

```

Finally, to finish off the applet, Neko wakes up and runs off to the right side of the screen. `awake` up is your last image in the array (position eight), and you can reuse the `nekorun` method to finish:

```

// wake up and run off
currentimg = nekopics[8];
repaint();
pause(500);
nekorun(xpos, size().width + 10);

```

There's one more thing left to do to finish the applet. The images for the animation all have white backgrounds. Drawing those images on the default applet background (a medium grey) means an unsightly white box around each image. To get around the problem, merely set the applet's background to white at the start of the `run()` method:

```

setBackground(Color.white);

```

Got all that? There's a lot of code in this applet, and a lot of individual methods to accomplish a rather simple animation, but it's not all that complicated. The heart of it, as in the heart of all Java animations, is to set up the frame and then call `repaint()` to enable the screen to be drawn.

Note that you don't do anything to reduce the amount of flicker in this applet. It turns out that the images are small enough, and the drawing area also small enough, that flicker is not a problem for this applet. It's always a good idea to write your animations to do the simplest thing first, and then add behavior to make them run cleaner.

To finish up this section, Listing 11.3 shows the complete code for the Neko applet.

Type

Listing 11.3. The final Neko applet.

```

1: import java.awt.Graphics;
2: import java.awt.Image;
3: import java.awt.Color;
4:
5: public class Neko extends java.applet.Applet
6:     implements Runnable {
7:
8:     Image nekopics[] = new Image[9];
9:     Image currentimg;
10:    Thread runner;
11:    int xpos;

```

```

12:     int ypos = 50;
13:
14:     public void init() {
15:         String nekosrc[] = { "right1.gif", "right2.gif",
16:             "stop.gif", "yawn.gif", "scratch1.gif",
17:             "scratch2.gif", "sleep1.gif", "sleep2.gif",
18:             "awake.gif" };
19:
20:         for (int i=0; i < nekopics.length; i++) {
21:             nekopics[i] = getImage(getCodeBase(),
22:                 "images/" + nekosrc[i]);
23:         }
24:     }
25:     public void start() {
26:         if (runner == null) {
27:             runner = new Thread(this);
28:             runner.start();
29:         }
30:     }
31:
32:     public void stop() {
33:         if (runner != null) {
34:             runner.stop();
35:             runner = null;
36:         }
37:     }
38:
39:     public void run() {
40:
41:         setBackground(Color.white);
42:
43:         // run from one side of the screen to the middle
44:         nekorun(0, size().width / 2);
45:
46:         // stop and pause
47:         currentimg = nekopics[2];
48:         repaint();
49:         pause(1000);
50:
51:         // yawn
52:         currentimg = nekopics[3];
53:         repaint();
54:         pause(1000);
55:
56:         // scratch four times
57:         nekoscratch(4);
58:
59:         // sleep for 5 "turns"
60:         nekosleep(5);
61:
62:         // wake up and run off
63:         currentimg = nekopics[8];
64:         repaint();
65:         pause(500);
66:         nekorun(xpos, size().width + 10);

```



continues

Listing 11.3. continued

```

67:     }
68:
69:     void nekorun(int start, int end) {
70:         for (int i = start; i < end; i += 10) {
71:             xpos = i;
72:             // swap images
73:             if (currentimg == nekopics[0])
74:                 currentimg = nekopics[1];
75:             else if (currentimg == nekopics[1])
76:                 currentimg = nekopics[0];
77:             else currentimg = nekopics[0];
78:             repaint();
79:             pause(150);
80:         }
81:     }
82:
83:     void nekoscratch(int numtimes) {
84:         for (int i = numtimes; i > 0; i--) {
85:             currentimg = nekopics[4];
86:             repaint();
87:             pause(150);
88:             currentimg = nekopics[5];
89:             repaint();
90:             pause(150);
91:         }
92:     }
93:
94:     void nekosleep(int numtimes) {
95:         for (int i = numtimes; i > 0; i--) {
96:             currentimg = nekopics[6];
97:             repaint();
98:             pause(250);
99:             currentimg = nekopics[7];
100:            repaint();
101:            pause(250);
102:        }
103:
104:        void pause(int time) {
105:            try { Thread.sleep(time); }
106:            catch (InterruptedException e) { }
107:        }
108:
109:        public void paint(Graphics g) {
110:            g.drawImage(currentimg, xpos, ypos, this);
111:        }
112:    }

```

Retrieving and Using Sounds

Java has built-in support for playing sounds in conjunction with running animations or for sounds on their own. In fact, support for sound, like support for images, is built into the Applet and awt classes, so using sound in your Java applets is as easy as loading and using images.

Currently, the only sound format that Java supports is Sun's AU format, sometimes called μ -law format. AU files tend to be smaller than sound files in other formats, but the sound quality is not very good. If you're especially concerned with sound quality, you may want your sound clips to be references in the traditional HTML way (as links to external files) rather than included in a Java applet.

The simplest way to retrieve and play a sound is through the `play()` method, part of the Applet class and therefore available to you in your applets. The `play()` method is similar to the `getImage` method in that it takes one of two forms:

- ☐ `play()` with one argument, a URL object, loads and plays the given audio clip at that URL.
- ☐ `play()` with two arguments, one a base URL and one a pathname, loads and plays that audio file. The first argument can most usefully be either a call to `getDocumentBase()` or `getCodeBase()`.



For example, the following line of code retrieves and plays the sound `meow.au`, which is contained in the audio directory. The audio directory, in turn, is located in the same directory as this applet:

```
play(getCodeBase(), "audio/meow.au");
```

The `play()` method retrieves and plays the given sound as soon as possible after it is called. If it can't find the sound, you won't get an error; you just won't get any audio when you expect it.

If you want to play a sound repeatedly, start and stop the sound clip, or run the clip as a loop (play it over and over), things are slightly more complicated—but not much more so. In this case, you use the applet method `getAudioClip()` to load the sound clip into an instance of the class `AudioClip` (part of `java.applet`—don't forget to import it) and then operate directly on that `AudioClip` object.

Suppose, for example, that you have a sound loop that you want to play in the background of your applet. In your initialization code, you can use this line to get the audio clip:

```
AudioClip clip = getAudioClip(getCodeBase(),
    "audio/loop.au");
```

Then, to play the clip once, use the `play()` method:

```
clip.play();
```

To stop a currently playing sound clip, use the `stop()` method:

```
clip.stop();
```

To loop the clip (play it repeatedly), use the `loop()` method:

```
clip.loop();
```

If the `getAudioClip` method can't find the sound you indicate, or can't load it for any reason, it returns `null`. It's a good idea to test for this case in your code before trying to play the audio clip, because trying to call the `play()`, `stop()`, and `loop()` methods on a `null` object will result in an error (actually, an exception).

In your applet, you can play as many audio clips as you need; all the sounds you use will mix together properly as they are played by your applet.

Note that if you use a background sound—a sound clip that loops repeatedly—that sound clip will not stop playing automatically when you suspend the applet's thread. This means that even if your reader moves to another page, the first applet's sounds will continue to play. You can fix this problem by stopping the applet's background sound in your `stop()` method:

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

Listing 11.4 shows a simple framework for an applet that plays two sounds: the first, a background sound called `loop.au`, plays repeatedly. The second, a horn honking (`beep.au`) plays every five seconds. (I won't bother giving you a picture of this applet, because it doesn't actually display anything other than a simple string to the screen).

Type

Listing 11.4. The AudioLoop applet.

```
1: import java.awt.Graphics;
2: import java.applet.AudioClip;
3:
4: public class AudioLoop extends java.applet.Applet
5:     implements Runnable {
6:
7:     AudioClip bgsound;
8:     AudioClip beep;
9:     Thread runner;
10:
11:     public void start() {
12:         if (runner == null) {
13:             runner = new Thread(this);
14:             runner.start();
```

```

15:     }
16: }
17:
18: public void stop() {
19:     if (runner != null) {
20:         if (bg sound != null) bg sound = null;} bg sound.stop();
21:         runner.stop();
22:         runner = null;
23:     }
24: }
25:
26: public void init() {
27:     bg sound = getAudioClip(getCodeBase(),"audio/loop.au");
28:     beep = getAudioClip(getCodeBase(), "audio/beep.au");
29: }
30:
31: public void run() {
32:     if (bg sound != null) bg sound.loop();
33:     while (runner != null) {
34:         try { Thread.sleep(5000); }
35:         catch (InterruptedException e) { }
36:         if (bg sound != null) beep.play();
37:     }
38: }
39:
40: public void paint(Graphics g) {
41:     g.drawString("Playing Sounds....", 10, 10);
42: }
43: }

```



Sun's Animator Applet

Because most Java animations have a lot of code in common, being able to reuse all that code as much as possible makes creating animations with images and sounds much easier, particular for Java developers who aren't as good at the programming side of Java. For just this reason, Sun provides an Animator class as part of the standard Java release.

The Animator applet provides a simple, general-purpose animation interface. You compile the code and create an HTML file with the appropriate parameters for the animation. Using the Animator applet, you can do the following:

- ☐ Create an animation loop, that is, an animation that plays repeatedly.
- ☐ Add a soundtrack to the applet.
- ☐ Add sounds to be played at individual frames.
- ☐ Indicate the speed at which the animation is to occur.
- ☐ Specify the order of the frames in the animation—which means that you can reuse frames that repeat during the course of the animation.

Even if you don't intend to use Sun's Animator code, it's a great example of how animations work in Java and the sorts of clever tricks you can use in a Java applet.

The Animator class is part of the Java distribution (in the demo directory), or you can find out more information about it at the Java home page, <http://java.sun.com>.

More About Flicker: Double-Buffering

Yesterday, you learned two simple ways to reduce flickering in animations. Although you learned specifically about animations using drawing, flicker can also result from animations using images. In addition to the two flicker-reducing methods described yesterday, there is one other way to reduce flicker in an application: double-buffering.

NEW TERM With *double-buffering*, you create a second surface (offscreen, so to speak), do all your painting to that offscreen surface, and then draw the whole surface at once onto the actual applet (and onto the screen) at the end—rather than drawing to the applet's actual graphics surface. Because all the work actually goes on behind the scenes, there's no opportunity for interim parts of the drawing process to appear accidentally and disrupt the smoothness of the animation.

Double-buffering isn't always the best solution. If your applet is suffering from flicker, try overriding `update()` and drawing only portions of the screen first; that may solve your problem. Double-buffering is less efficient than regular buffering, and also takes up more memory and space, so if you can avoid it, make an effort to do so. In terms of nearly eliminating animation flicker, however, double-buffering works exceptionally well.

Creating Applets with Double-Buffering

To create an applet that uses double-buffering, you need two things: an offscreen image to draw on and a graphics context for that image. Those two together mimic the effect of the applet's drawing surface: the graphics context (an instance of `Graphics`) to provide the drawing methods, such as `drawImage` and `drawString`, and the `Image` to hold the dots that get drawn.

There are four major steps to adding double-buffering to your applet. First, your offscreen image and graphics context need to be stored in instance variables so that you can pass them to the `paint()` method. Declare the following instance variables in your class definition:

```
Image offscreenImage;
Graphics offscreenGraphics;
```

Second, during the initialization of the applet, you'll create an `Image` and a `Graphics` object and assign them to these variables (you have to wait until initialization so you know how big they're going to be). The `createImage()` method gives you an instance of `Image`, which you can then send the `getGraphics()` method in order to get a new graphics context for that image:

```
offscreenImage = createImage(size().width,
    size().height);
offscreenGraphics = offscreenImage.getGraphics();
```

Now, whenever you have to draw to the screen (usually in your `paint()` method), rather than drawing to `paint`'s graphics, draw to the offscreen graphics. For example, to draw an image called `img` at position `10,10`, use this line:

```
offscreenGraphics.drawImage(img, 10, 10, this);
```

Finally, at the end of your `paint` method, after all the drawing to the offscreen image is done, add the following line to place the offscreen buffer on to the real screen:

```
g.drawImage(offscreenImage, 0, 0, this);
```

Of course, you most likely will want to override `update()` so that it doesn't clear the screen between paintings:

```
public void update(Graphics g) {
    paint(g);
}
```

Let's review those four steps:

- ☐ Add instance variables to hold the image and graphics contexts for the offscreen buffer.
- ☐ Create an image and a graphics context when your applet is initialized.
- ☐ Do all your applet painting to the offscreen buffer, not the applet's drawing surface.
- ☐ At the end of your `paint` method, draw the offscreen buffer to the real screen.



An Example: Checkers Revisited

Yesterday's example featured the animated moving red oval to demonstrate animation flicker and how to reduce it. Even with the operations you did yesterday, however, the Checkers applet still flashed occasionally. Let's revise that applet to include double-buffering.

First, add the instance variables for the offscreen image and its graphics context:

```
Image offscreenImg;
Graphics offscreenG;
```

Second, add an `init` method to initialize the offscreen buffer:

```
public void init() {
    offscreenImg = createImage(size().width,
        size().height);
    offscreenG = offscreenImg.getGraphics();
}
```

Third, modify the `paint()` method to draw to the offscreen buffer instead of to the main graphics buffer:

```
public void paint(Graphics g) {
    // Draw background
    offscreenG.setColor(Color.black);
    offscreenG.fillRect(0, 0, 100, 100);
    offscreenG.setColor(Color.white);
    offscreenG.fillRect(100, 0, 100, 100);

    // Draw checker
    offscreenG.setColor(Color.red);
    offscreenG.fillOval(xpos, 5, 90, 90);

    g.drawImage(offscreenImg, 0, 0, this);
}
```

Note that you're still clipping the main graphics rectangle in the `update()` method, as you did yesterday; you don't have to change that part. The only part that is relevant is that final `paint` method wherein everything is drawn offscreen before finally being displayed.

Summary

Three major topics were the focus of today's lesson. First, you learned about using images in your applets—locating them, loading them, and using the `drawImage()` method to display them, either at their normal size or scaled to different sizes. You also learned how to create animations using images.

Secondly, you learned how to use sounds, which can be included in your applets any time you need them—at specific moments, or as background sounds that can be repeated while the applet executes. You learned how to locate, load, and play sounds both using the `play()` and the `getAudioClip()` methods.

Finally, you learned about double-buffering, a technique that enables you virtually to eliminate flicker in animations, at some expense of animation efficiency and speed. Using images and graphics contexts, you can create an offscreen buffer to draw to, the result of which is then displayed to the screen at the last possible moment.

Q&A

Q In the Neko program, you put the image loading into the `init()` method. It seems to me that it might take Java a long time to load all those images, and because `init()` isn't in the main thread of the applet, there's going to be a distinct pause there. Why not put the image loading at the beginning of the `run()` method instead?

A There are sneaky things going on behind the scenes. The `getImage()` method doesn't actually load the image; in fact, it returns an `Image` object almost instantaneously, so it isn't taking up a large amount of processing time during initialization. The image data that `getImage()` points to isn't actually loaded until the image is needed. This way, Java doesn't have to keep enormous images around in memory if the program is going to use only a small piece. Instead, it can just keep a reference to that data and retrieve what it needs later.

Q I wrote an applet to do a background sound using the `getAudioClip()` and `loop()` methods. The sounds works great, but it won't stop. I've tried suspending the current thread and killing, but the sound goes on.

A I mentioned this as a small note in the section on sounds; background sounds don't run in the main thread of the applet, so if you stop the thread, the sound keeps going. The solution is easy—in the same method where you stop the thread, also stop the sound, like this:

```
runner.stop() //stop the thread
bgsound.stop() //also stop the sound
```

Q If I use double-buffering, do I still have to clip to a small region of the screen? Because double-buffering eliminates flicker, it seems easier to draw the whole frame every time.

A Easier, yes, but less efficient. Drawing only part of the screen not only reduces flicker, it often also limits the amount of work your applet has to do in the `paint()` method. The faster the `paint()` method works, the faster and smoother your animation will run. Using clip regions and drawing only what is necessary is a good practice to follow in general—not just if you have a problem with flicker.





13

WEEK
2

The Java Abstract Windowing Toolkit

by Laura Lemay

For the past five days you've concentrated on creating applets that do very simple things: display text, play an animation or a sound, or enable very basic interactions with the user. Once you get past that point, however, you may want to start creating more complex applets that behave like real applications, embedded in a Web page—applets that start to look like real GUI applications with buttons, menus, text fields and other elements of a real application.

It's this sort of real work in Java applets and applications that Java's Abstract Windowing Toolkit, or AWT, was designed for. You've actually been using the AWT all along, as you might have guessed from the classes you've been importing. The `Applet` class and most of the classes you've been using this week are all integral parts of the AWT. In fact, the HotJava browser is also written in Java and uses the AWT as well.

The AWT provides the following:

- ☐ A full set of UI widgets and other components, including windows, menus, buttons, checkboxes, text fields, scrollbars, and scrolling lists
- ☐ Support for UI "containers," which can contain other embedded containers or UI widgets
- ☐ An event system for managing system and user events between and among parts of the AWT
- ☐ Mechanisms for laying out components in a way that enables platform-independent UI design

Today, you'll learn about how to use all these things in your Java applets. Tomorrow, you'll learn about creating windows, menus, and dialogs, which enable you to pop up separate windows from the browser window. In addition, you can use the AWT in stand-alone applications, so everything you've learned so far this week can still be used. If you find the framework of the Web browser too limiting, you can take your AWT background and start writing full-fledged Java applications.

Today, however, you'll continue focusing on applets.



Note: This is by far the most complex lesson so far. There's a lot to cover and a lot of code to go through today, so if it starts becoming overwhelming, you might want to take two days (or more) for this one.

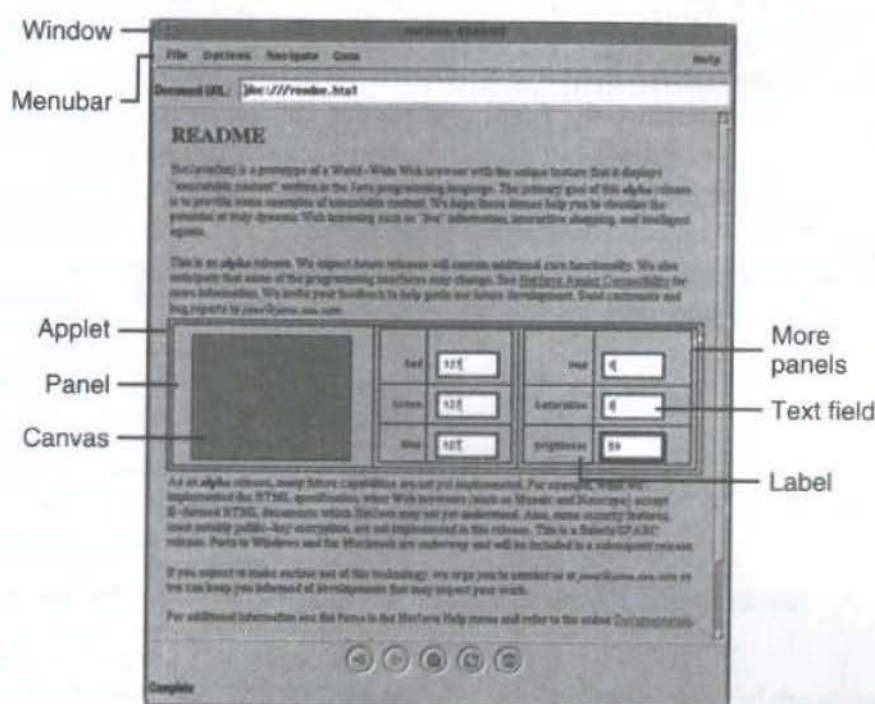
An AWT Overview

The basic idea behind the AWT is that a Java window is a set of nested components, starting from the outermost window all the way down to the smallest UI component. Components can

include things you can actually see on the screen, such as windows, menubars, buttons, and text fields, and they can also include containers, which in turn can contain other components. Figure 13.1 shows how a sample page in a Java browser might include several different components, all of which are managed through the AWT.

This nesting of components within containers within other components creates a hierarchy of components, from the smallest checkbox inside an applet to the overall window on the screen. The hierarchy of components determines the arrangement of items on the screen and inside other items, the order in which they are painted, and how events are passed from one component to another.

Figure 13.1.
AWT components.



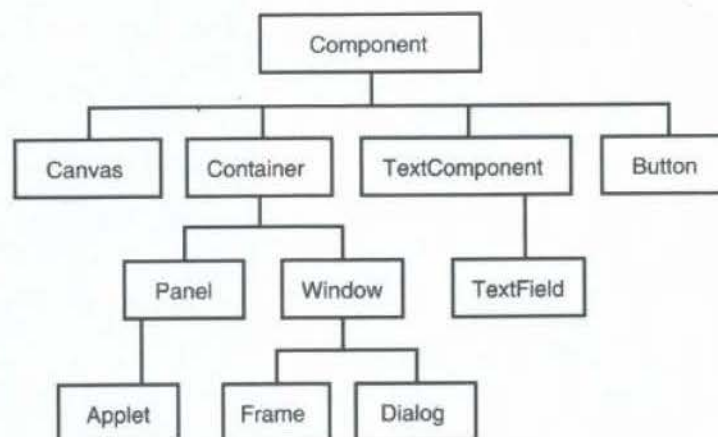
These are the major components you can work with in the AWT:

- ☐ **Containers.** Containers are generic AWT components that can contain other components, including other containers. The most common form of container is the panel, which represents a container that can be displayed on screen. Applets are a form of panel (in fact, the `Applet` class is a subclass of the `Panel` class).
- ☐ **Canvases.** A canvas is a simple drawing surface. Although you can draw on panels (as you've been doing all along), canvases are good for painting images or other graphics operations.
- ☐ **UI components.** These can include buttons, lists, simple popup menus, checkboxes, test fields, and other typical elements of a user interface.

- *Window construction components.* These include windows, frames, menubars, and dialogs. These are listed separately from the other UI components because you'll use these less often—particularly in applets. In applets, the browser provides the main window and menubar, so you don't have to use these. Your applet may create a new window, however, or you may want to write your own Java application that uses these components.

The classes inside the `java.awt` package are written and organized to mirror the abstract structure of containers, components, and individual UI components. Figure 13.2 shows some of the class hierarchy that makes up the main classes in the AWT. The root of most of the AWT components is the class `Component`, which provides basic display and event handling features. The classes `Container`, `Canvas`, `TextComponent`, and many of the other UI components inherit from `Component`. Inheriting from the `Container` class are objects that can contain other AWT components—the `Panel` and `Window` classes, in particular. Note that the `java.applet.Applet` class, even though it lives in its own package, inherits from `Panel`, so your applets are an integral part of the hierarchy of components in the AWT system.

Figure 13.2.
A partial AWT class hierarchy.



A graphical user interface-based application that you write by using the AWT can be as complex as you like, with dozens of nested containers and components inside each other. AWT was designed so that each component can play its part in the overall AWT system without needing to duplicate or keep track of the behavior of other parts in the system.

The Basic User Interface Components

The simplest form of AWT component is the basic UI component. You can create and add these to your applet without needing to know anything about creating containers or panels—your applet, even before you start painting and drawing and handling events, is already an AWT container. Because an applet is a container, you can put other AWT components—such as UI components or other containers—into it.

In this section, you'll learn about the basic UI components: labels, buttons, checkboxes, choice menus, and text fields. In each case, the procedure for creating the component is the same—you first create the component, and then add it to the panel that holds it, at which point it is displayed on the screen. To add a component to a panel (such as your applet, for example), use the `add()` method:

```
public void init() {
    Button b = new Button("OK");
    add(b);
}
```

Note that where the component appears in the panel depends on the layout that panel is defined to have. The default layout for panels such as applets is `FlowLayout`, with a centered alignment, which means that components are added from left to right in rows, and then row by row as they fit, with each row centered. This explains why some of the examples in this section look a little funny. You'll learn more about panels and layouts in the next section.

Note also that each of these components has an action associated with it—that is, something that component does when it's activated. Actions generally trigger events or other activities in your applet (often called callbacks in other window toolkits). In this section, you'll focus on creating the components themselves; you'll learn about adding actions to them later in today's lesson.

On to the components!

Labels

The simplest form of UI component is the label.

NEW TERM *Labels* are, effectively, text strings that you can use to label other UI components.

The advantages that a label has over an ordinary text string is that it follows the layout of the given panel, and you don't have to worry about repainting it every time the panel is redrawn. Labels also can be easily aligned within a panel, enabling you to attach labels to other UI components without knowing exact pixel positions.

To create a label, use one of the following constructors:

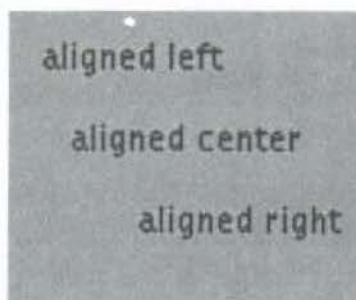
- ☐ `Label()` creates an empty label, with its text aligned left.
- ☐ `Label(String)` creates a label with the given text string, also aligned left.
- ☐ `Label(String, int)` creates a label with the given text string and the given alignment. The available alignments are stored in class variables in `Label`, making them easier to remember: `Label.RIGHT`, `Label.LEFT`, and `Label.CENTER`.

The label's font is determined by the overall font for the component (as set by the `setFont()` method).

Here's some simple code to create a few labels. Figure 13.3 shows how this looks on screen:

```
add(new Label("aligned left "));
add(new Label("aligned center", Label.CENTER));
add(new Label(" aligned right", Label.RIGHT));
```

Figure 13.3.
Labels.



Once you have a label object, you can use methods defined in the `Label` class to get and set the values of the text as shown in Table 13.1.

Table 13.1. Label methods.

Method	Action
<code>getText()</code>	Returns a string containing this label's text
<code>setText(String)</code>	Changes the text of this label
<code>getAlignment()</code>	Returns an integer representing the alignment of this label: 0 is <code>Label.LEFT</code> , 1 is <code>Label.CENTER</code> , 2 is <code>Label.RIGHT</code>
<code>setAlignment(int)</code>	Changes the alignment of this label to the given integer—use the class variables above.

Buttons

The second user interface component to explore is the button.

NEW TERM *Buttons* are simple UI components that trigger some action in your interface when they are pressed. For example, a calculator applet might have buttons for each number and operator, or a dialog box might have buttons for “OK” and “Cancel.”

To create a button, use one of the following constructors:

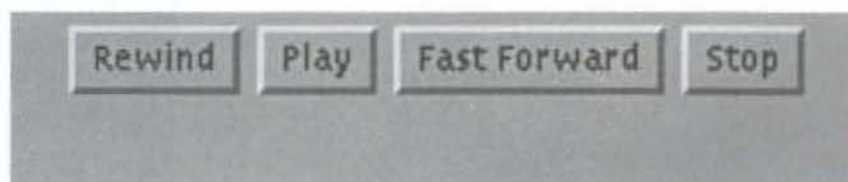
- ☐ `Button()` creates an empty button with no label.
- ☐ `Button(String)` creates a button with the given string object as a label.

Once you have a button object, you can get the value of the button's label by using the `getLabel()` method and set the label using the `setLabel(String)` methods.

Figure 13.4 shows some simple buttons, created using the following code:

```
add(new Button("Rewind"));
add(new Button("Play"));
add(new Button("Fast Forward"));
add(new Button("Stop"));
```

Figure 13.4.
Buttons.



Checkboxes

Checkboxes can be selected or deselected to provide options.

NEW TERM *Checkboxes* are user interface components that have two states: on and off (or checked and unchecked, selected and unselected, true and false, and so on). Unlike buttons, checkboxes usually don't trigger direct actions in a UI but, instead, are used to indicate optional features of some other action.

Checkboxes can be used in two ways:

- ☐ Nonexclusive, meaning that given a series of checkboxes, any of them can be selected.
- ☐ Exclusive, meaning that within one series, only one checkbox can be selected at a time.

The latter kind of checkboxes are called radio buttons or checkbox groups, and are described in the next section.

Nonexclusive checkboxes can be created by using the `Checkbox` class. You can create a checkbox by using one of the following constructors:

- ☐ `Checkbox()` creates an empty checkbox, unselected.
- ☐ `Checkbox(String)` creates a checkbox with the given string as a label.
- ☐ `Checkbox(String, null, boolean)` creates a checkbox that is either selected or unselected based on whether the boolean argument is true or false, respectively. (The `null` is used as a placeholder for a group argument. Only radio buttons have groups, as you'll learn in the next section).

Table 13.2 lists the checkbox methods; Figure 13.5 shows a few simple checkboxes (only Underwear is selected), generated using the following code:

```
add(new Checkbox("Shoes"));
add(new Checkbox("Socks"));
add(new Checkbox("Pants"));
add(new Checkbox("Underwear", null, true));
add(new Checkbox("Shirt"));
```

Figure 13.5.
Checkboxes.

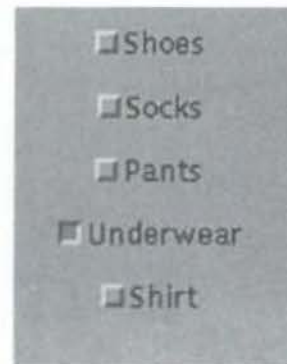


Table 13.2. Checkbox methods.

Method	Action
<code>getLabel()</code>	Returns a string containing this checkbox's label
<code>setLabel(String)</code>	Changes the text of the checkbox's label
<code>getState()</code>	Returns true or false, based on whether the checkbox is selected or not
<code>setState(boolean)</code>	Changes the checkbox's state to selected (true) or unselected (false)

Radio Buttons

Radio buttons are a variation on the checkbox.

NEW TERM *Radio buttons* have the same appearance as checkboxes, but only one in a series can be selected at a time.

To create a series of radio buttons, first create an instance of `CheckboxGroup`:

```
CheckboxGroup cbg = new CheckboxGroup();
```

Then create and add the individual checkboxes, using the group as the second argument, and whether or not that checkbox is selected (only one in the series can be selected):

```
add(new Checkbox("Yes", cbg, true);
add(new Checkbox("no", cbg, false);
```

Here's a simple example (the results of which are shown in Figure 13.6):

```
CheckboxGroup cbg = new CheckboxGroup();

add(new Checkbox("Red", cbg, false));
add(new Checkbox("Blue", cbg, false));
add(new Checkbox("Yellow", cbg, false));
add(new Checkbox("Green", cbg, false));
add(new Checkbox("Orange", cbg, true));
add(new Checkbox("Purple", cbg, false));
```

Figure 13.6.
Radio buttons.



All the checkbox methods defined in the previous section can be used with the checkboxes in the group. In addition, you can use the `getCheckboxGroup()` and `setCheckboxGroup()` methods to access and change the group of any given checkbox.

Finally, the `getCurrent()` and `setCurrent(Checkbox)` methods, defined in the checkbox group, can be used to get or set the currently selected checkbox.

13

Choice Menus

The choice menu is a more complex UI component than labels, buttons, or checkboxes.

NEW  *Choice menus* are popup (or pulldown) menus that enable you to select an item from that menu. The menu then displays that choice on the screen.

To create a choice menu, create an instance of the `Choice` class, and then use the `addItem()` method to add individual items to it in the order in which they should appear:

```
Choice c = new Choice();

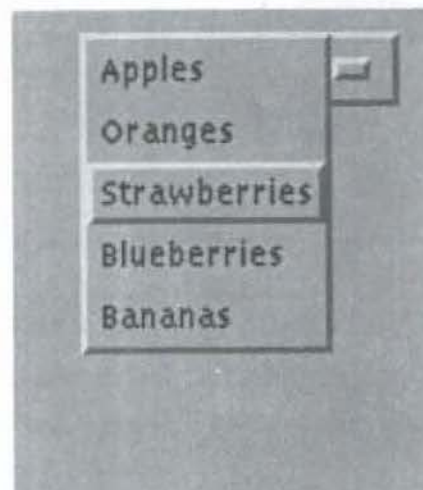
c.addItem("Apples");
c.addItem("Oranges");
c.addItem("Strawberries");
c.addItem("Blueberries");
c.addItem("Bananas");
```

Finally, add the entire choice menu to the panel in the usual way:

```
add(c);
```

Figure 13.7 shows a simple choice menu generated from code in the previous example:

Figure 13.7.
Choice menus.



Tip: Choice menus allow only one selection per menu. If you want to select multiple items, use a scrolling list instead.

Once your choice menu is created, regardless of whether it's added to a panel, you can continue to add items to that menu by using the `addItem()` method. Table 13.3 shows some other methods that may be useful in working with choice menus.

Table 13.3. Choice menu methods.

Method	Action
<code>getItem(int)</code>	Returns the string item at the given position (items inside a choice begin at 0, same as arrays)
<code>countItems()</code>	Returns the number of items in the menu
<code>getSelectedIndex()</code>	Returns the index position of the item that's selected

Method	Action
<code>getSelectedItem()</code>	Returns the currently selected item as a string
<code>select(int)</code>	Selects the item at the given position
<code>select(String)</code>	Selects the item with the given string

Text Fields

Unlike the UI components up to this point, which enable you to select only among several options to perform an action, text fields allow you to enter any values.

NEW TERM *Text fields* enable your reader to enter text.

To create a text field, use one of the following constructors:

- ☐ `TextField()` creates an empty `TextField` 0 characters wide.
- ☐ `TextField(int)` creates an empty text field with the given width in characters.
- ☐ `TextField(String)` creates a text field 0 characters wide, initialized with the given string.
- ☐ `TextField(String, int)` creates a text field with the given width in characters and containing the given string. If the string is longer than the width, you can select and drag portions of the text within the field and the box will scroll left or right.

For example, the following line creates a text field 30 characters wide with the string "Enter Your Name" as its initial contents.

```
TextField tf = new TextField("Enter Your Name", 30);
add(tf);
```



Tip: Text fields include only the editable field itself. You usually need to include a label with a text field to indicate what belongs in that text field.



Note: Text fields are different from text areas; text fields are limited in size and are best used for one-line items, whereas text areas have scrollbars and are better for larger text windows. Both can be edited and enable selections with the mouse. You'll learn about text areas later today.

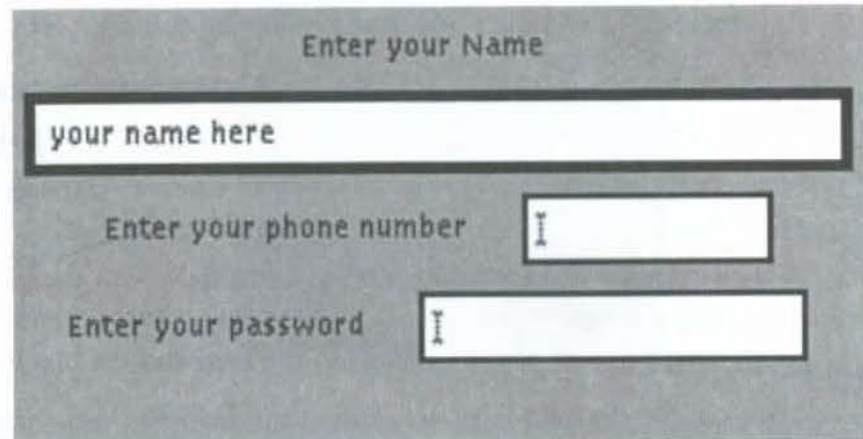
You can also create a text field that obscures the characters typed into it—for example, for password fields. To do this, first create the text field itself, and then use the `setEchoCharacter()` method to set the character that is echoed on the screen. Here is an example:

```
TextField tf = new TextField(30);
tf.setEchoCharacter('*');
```

Figure 13.8 shows three text boxes (and labels) that were created by using the following code:

```
add(new Label("Enter your Name"));
add(new TextField("your name here", 45));
add(new Label("Enter your phone number"));
add(new TextField(12));
add(new Label("Enter your password"));
TextField t = new TextField(20);
t.setEchoCharacter('*');
add(t);
```

Figure 13.8.
Text fields.



Text fields inherit from the class `TextComponent` and have a whole suite of methods, both inherited from that class and defined in its own class, that may be useful to you in your Java programs. Table 13.4 shows a selection of those methods.

Table 13.4. Text field methods.

Method	Action
<code>getText()</code>	Returns the text this text field contains (as a string)
<code>setText(String)</code>	Puts the given text string into the field
<code>getColumns()</code>	Returns the width of this text field
<code>select(int, int)</code>	Selects the text between the two integer positions (positions start from 0)
<code>selectAll()</code>	Selects all the text in the field

Method	Action
<code>isEditable()</code>	Returns <code>true</code> or <code>false</code> based on whether the text is editable or not
<code>setEditable(boolean)</code>	<code>true</code> (the default) enables text to be edited; <code>false</code> freezes the text
<code>getEchoChar()</code>	Returns the character used for masking input
<code>echoCharIsSet()</code>	Returns <code>true</code> or <code>false</code> whether the field has a masking character or not

Panels and Layout

You know at this point that an AWT panel can contain UI components or other panels. The question now is how those components are actually arranged and displayed on the screen.

In other windowing systems, UI components are often arranged using hard-coded pixel measurements—put text field `tf` at 10,30, for example—the same way you used the graphics operations to paint squares and ovals on the screen. In the AWT, the window may be displayed on many different windowing systems on many different screens and with many different kinds of fonts with different font metrics. Therefore, you need a more flexible method of arranging components on the screen so that a layout that looks nice on one platform isn't a jumbled unusable mess on another.

For just this purpose, Java has layout managers, insets, and hints that each component can provide for helping lay out the screen.

Note that the nice thing about AWT components and user interface items is that you don't have to paint them—the AWT system manages all that for you. If you have graphical components or images, or you want to create animations inside panels, you still have to do that by hand, but for most of the basic components, all you have to do is put them on the screen and Java will handle the rest.

13

Layout Managers

The actual appearance of the AWT components on the screen is determined by two things: the order in which they are added to the panel that holds them, and the layout manager that panel is currently using to lay out the screen. The layout manager determines how portions of the screen will be sectioned and how components within that panel will be placed.

Note that each panel on the screen can have its own layout manager. By nesting panels within panels, and using the appropriate layout manager for each one, you can often arrange your UI to group and arrange components in a way that is both functionally useful and also looks good

on a variety of platforms and windowing systems. You'll learn about nesting panels in a later section.

The AWT provides five basic layout managers: `FlowLayout`, `GridLayout`, `GridBagLayout`, `BorderLayout`, and `CardLayout`. To create a layout manager for a given panel, use the `setLayout()` method for that panel:

```
public void init() {
    setLayout(new FlowLayout());
}
```

Setting the default layout manager, like creating the user interface components, is best done during the applet's initialization, which is why it's included here.

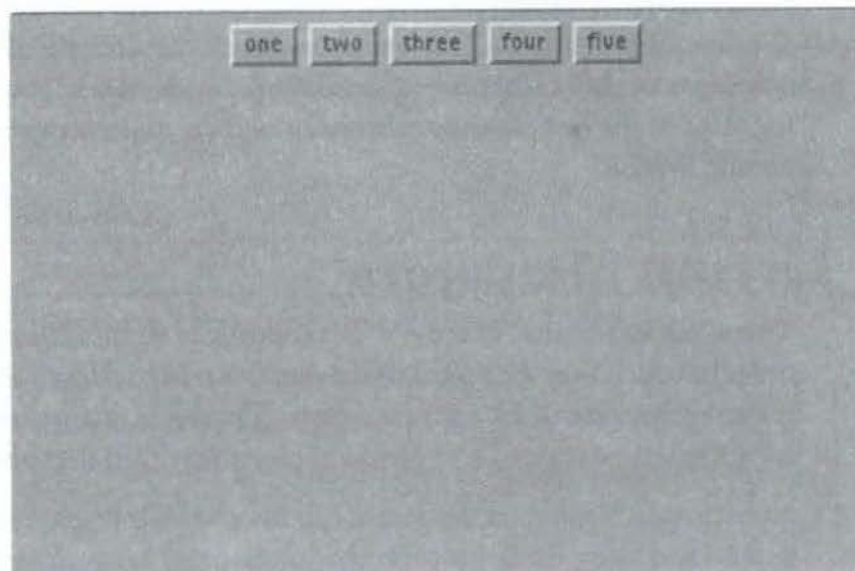
Once the layout manager is set, you can start adding components to the panel. The order in which components are added is often significant, depending on which layout manager is currently active. Read on for information about the specific layout managers and how they present components within the panel to which they apply.

The following sections describe the five basic Java AWT layout managers.

The *FlowLayout* Class

The `FlowLayout` class is the most basic of layouts. Using the flow layout, components are added to the panel one at a time, row by row. If a component doesn't fit onto a row, it's wrapped onto the next row. The flow layout also has an alignment, which determines the alignment of each row. By default, each row is aligned centered. Figure 13.9 shows a flow layout at its best—a simple row of buttons, centered on a line.

Figure 13.9.
Flow layout.



To create a basic flow layout with a centered alignment, use the following line of code in your panel's initialization (because this is the default pane layout, you don't need to include this line if that is your intent):

```
setLayout(new FlowLayout());
```

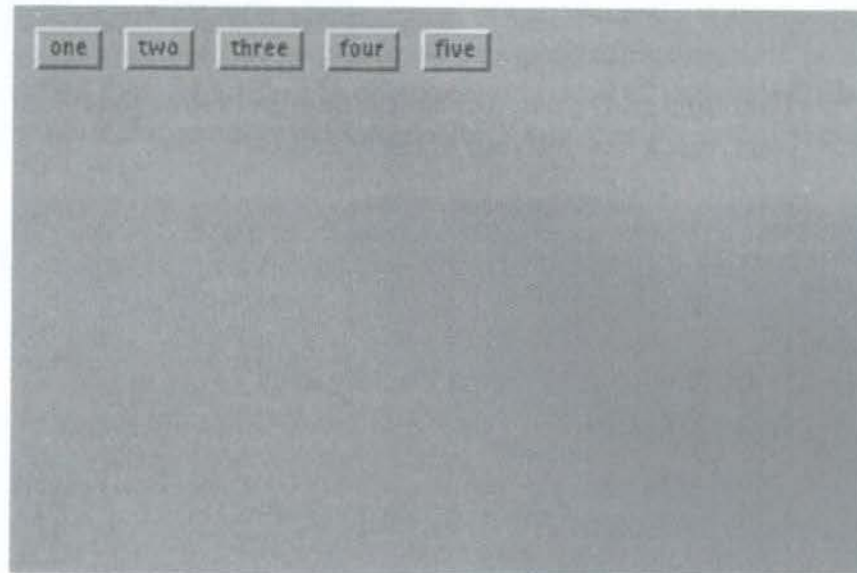
To create a flow layout with an alignment other than centered, add the `FlowLayout.RIGHT` or `FlowLayout.LEFT` class variable as an argument:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

You can also set horizontal and vertical gap values by using flow layouts. The gap is the number of pixels between components in a panel; by default, the horizontal and vertical gap values are three pixels, which can be very close indeed. Horizontal gap spreads out components to the left and to the right, vertical gap to the top and bottom of each component. Add integer arguments to the flow layout constructor to increase the gap (a layout gap of 10 points in both the horizontal and vertical directions is shown in Figure 13.10):

```
setLayout(new FlowLayout(FlowLayout.LEFT), 10, 10);
```

Figure 13.10.
*Flow layout with a gap of
10 points.*



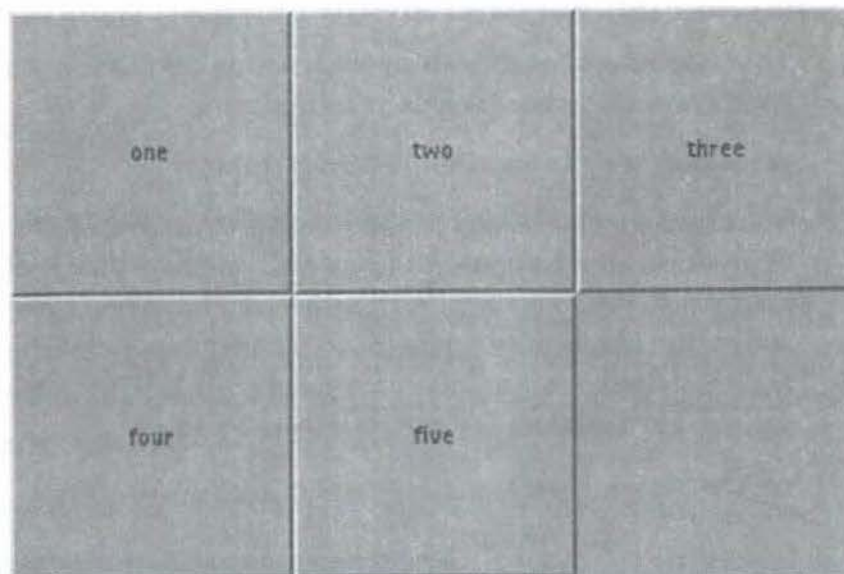
13

Grid and Grid Bag Layouts

Grid layouts use a layout that offers more control over the placement of components inside a panel. Using a grid layout, you portion off the area of the panel into rows and columns. Each component you then add to the panel is placed in a "cell" of the grid, starting from the top row and progressing through each row from left to right (here's where the order of calls to the `add()` method are very relevant to how the screen is laid out). By using grid layouts and nested grids,

you can often approximate the use of hard-coded pixel values to place your UI components precisely where you want them. Figure 13.11 shows a grid layout with three columns and three rows.

Figure 13.11.
Grid layout.



To create a grid layout, indicate the number of rows and columns you want the grid to have when you create a new instance of the `GridLayout` class:

```
setLayout(new GridLayout(3, 3));
```

Grid layouts can also have a horizontal and vertical gap between components; to create gaps, add those pixel values:

```
setLayout(new GridLayout(3, 3, 10, 15));
```

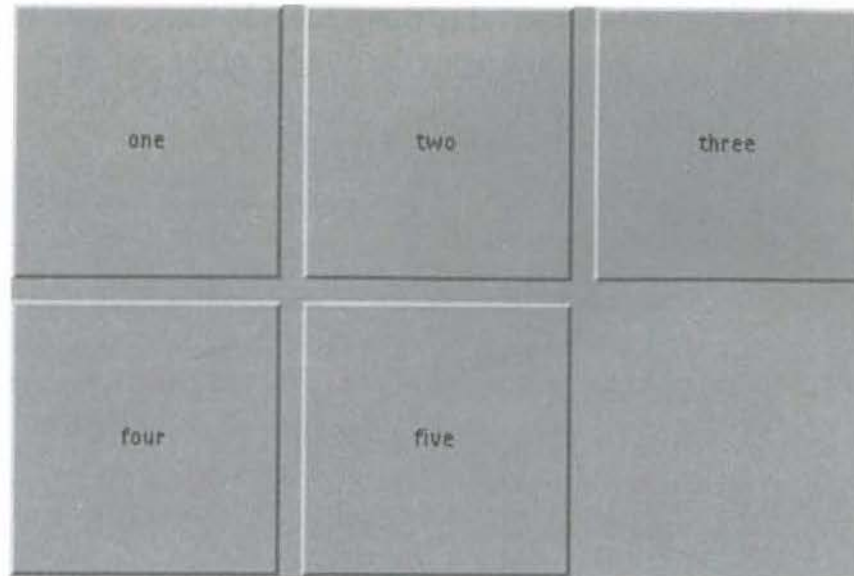
Figure 13.12 shows a grid layout with a 10-pixel horizontal gap and a 15-pixel vertical gap.

Grid bag layouts, as implemented by the `GridBagLayout` class, are variations on grid layouts. Grid bag layouts also enable you to lay out your user interface elements in a rectangular grid, but with grid bag layouts you have much more control over the presentation of each element in the grid. Grid bag layouts use a helper class, `GridBagConstraints`, to indicate how each cell in the grid is to be formatted.



Note: The `GridBagLayout` and `GridBagConstraints` classes were added to the Java Developer's Kit just before this book went to press. For a much better description of grid bag layouts, see the API documentation for those classes that comes with the JDK.

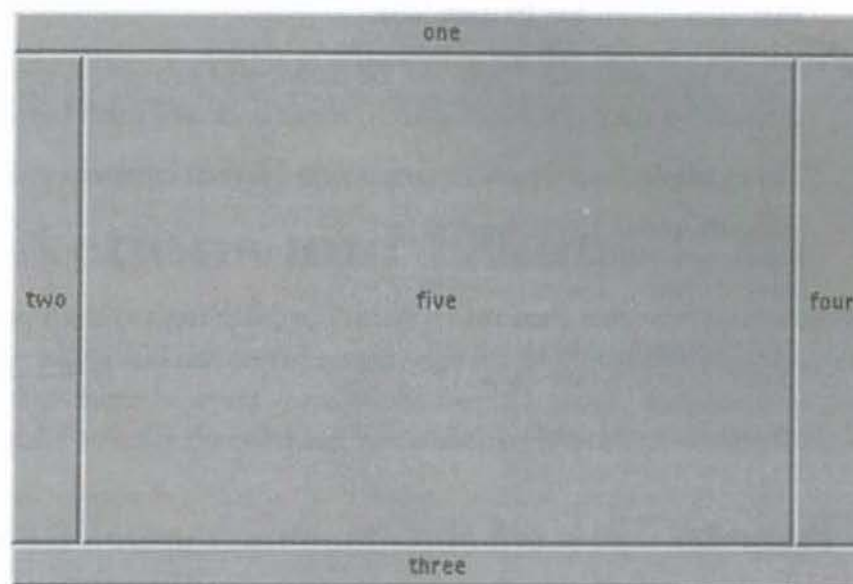
Figure 13.12.
*Grid layouts with horizontal
and vertical gap.*



Border Layouts

Border layouts behave differently from flow and grid layouts. When you add a component to a panel that uses a border layout, you indicate its placement as a geographic direction: north, south, east, west, and center (see Figure 13.13). The components around all the edges are laid out with as much size as they need; the component in the center, if any, gets any space left over.

Figure 13.13.
Border layout.



13

To use a border layout, you create it as you do the other layouts:

```
setLayout(new BorderLayout());
```

Then you add the individual components by using a special `add()` method: the first argument to `add()` is a string indicating the position of the component within the layout:

```
add("North", new TextField("Title", 50));
add("South", new TextField("Status", 50));
```

You can also use this form of `add()` for the other layout managers; the string argument will just be ignored if it's not needed.

Border layouts can also have horizontal and vertical gaps. Note that the north and south components extend all the way to the edge of the panel, so the gap will result in less space for the east, right, and center components. To add gaps to a border layout, include those pixel values as before:

```
setLayout(new BorderLayout(10, 10));
```

Card Layouts

Card layouts are different from the other layouts. Unlike with the other three layouts, when you add components to a card layout, they are not all displayed on the screen at once. Card layouts are used to produce slide shows of components, one at a time. If you've ever used the HyperCard program on the Macintosh, you've worked with the same basic idea.

Generally when you create a card layout, the components you add to it will be other container components—usually panels. You can then use different layouts for those individual “cards” so that each screen has its own look.

When you add each “card” to the panel, you can give it a name. Then you can use methods defined on the `CardLayout` class to move back and forth between different cards in the layout.

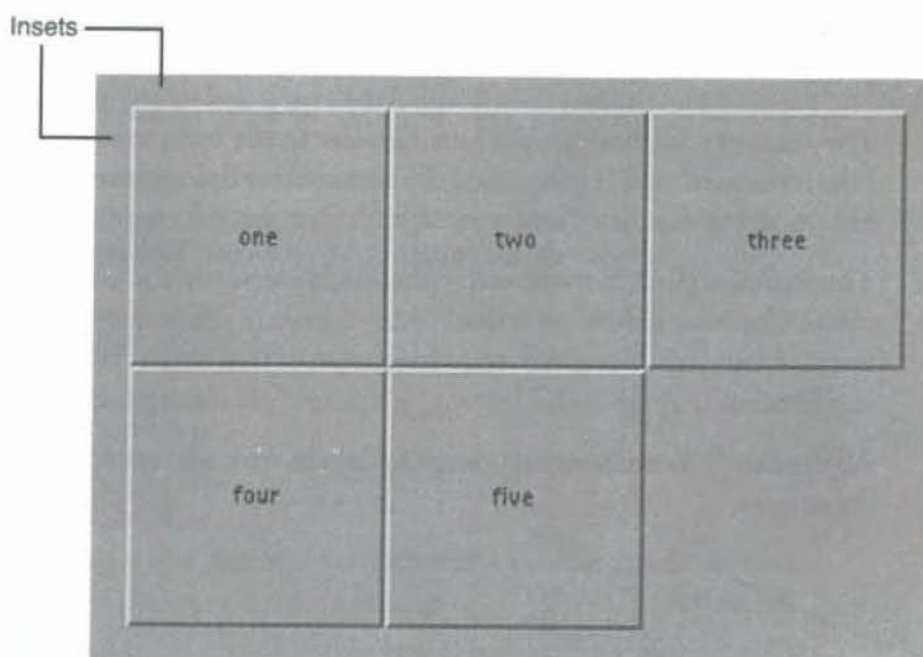
For example, here's how to create a card layout containing three cards:

```
setLayout(new CardLayout());
Panel one = new Panel()
add("first", one);
Panel two = new Panel()
add("second", two);
Panel three = new Panel()
add("third", three);
show(this, "second");
```

Insets

Whereas horizontal gap and vertical gap are used to determine the amount of space between components in a panel, insets are used to determine the amount of space around the panel itself. The `Insets` class provides values for the top, bottom, left, and right insets, which are then used when the panel itself is drawn. Figure 13.14 shows an inset in a `GridLayout`.

Figure 13.14.
Insets.



To include an inset, override the `insets()` method in your class (your `Applet` class or other class that serves as a panel):

```
public Insets insets() {
    return new Insets(10, 10, 10, 10);
}
```

The arguments to the `Insets` constructor provide pixel insets for the top, bottom, left, and right edges of the panel. This particular example provides an inset of 10 pixels on all four sides of the panel.

Handling UI Actions and Events

If you stopped reading today's lesson right now, you could go out and create an applet that had lots of little UI components, nicely laid out on the screen with the proper layout manager, gap, and insets. If you did stop right here, however, your applet would be really dull, because none of your UI components would actually do anything when they were pressed or typed into or selected.

For your UI components to do something when they are activated, you need to hook up the UI's action with an operation.

Testing for an action by a UI component is a form of event management—the things you learned yesterday about events will come in handy here. In particular, UI components produce the special kind of event called an action. To intercept an action by any UI component, you define an `action()` method in your applet or class:

```
public boolean action(Event evt, Object arg) {
    ...
}
```

The `action()` method should look familiar to the basic mouse and keyboard event methods. Like those methods, it gets passed the event object that represents this event. It also gets an extra object, which can be of any type. What's that second argument for?

The second argument to the action method depends on the UI component that's generating the event. The basic definition is that it's any arbitrary argument—when a component generates an event, it can pass along any extra information that might later be needed. Because that extra information may be useful for you, it's passed on through the `action()` method.

All the basic UI components (except for labels, which have no action) have different actions and arguments:

- ☐ Buttons create actions when they are selected, and a button's argument is the label of the button.
- ☐ Checkboxes, both exclusive and nonexclusive, generate actions when a box is checked. The argument is always `true`.
- ☐ Choice menus generate an action when a menu item is selected, and the argument is that item.
- ☐ Text fields create actions when the user presses Return inside that text field. Note that if the user tabs to a different text field or uses the mouse to change the input focus, an action is *not* generated. Only a Return triggers the action.

Note that with actions, unlike with ordinary events, you can have many different kinds of objects generating the event, as opposed to a single event such as a `mouseDown`. To deal with those different UI components and the actions they generate, you have to test for the type of object that sent/created the event in the first place inside the body of your `action()` method. That object is stored in the event's `target` instance variable, and you can use the `instanceof` operator to find out what kind of UI component sent it:

```
public boolean action(Event evt, Object arg) {
    if (evt.target instanceof TextField)
        handleText(evt.target);
    else if (evt.target instanceof Choice)
        handleChoice(arg);
    ...
}
```

Although you can handle UI actions in the body of the `action()` method, it's much more common simply to define a handler method and call that method from `action()` instead. Here, there are two handler methods: one to handle the action on the text field (`handleText()`) and one to handle the action on the choice menu (`handleChoice()`). Depending on the action you want to handle, you may also want to pass on the argument from the action, the UI component that sent it, or any other information that the event might contain.

Here's a simple applet that has five buttons labeled with colors. The `action()` method tests for a button action and then passes off the word to a method called `changeColor()`, which changes the background color of the applet based on which button was pressed (see Figure 13.15 to see the applet in action):

```
import java.awt.*;

public class ButtonActionsTest extends java.applet.Applet {

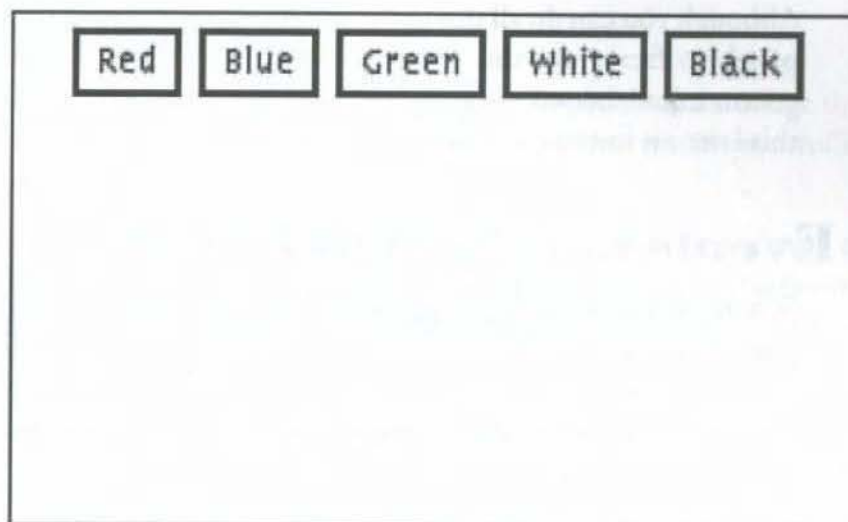
    public void init() {
        setBackground(Color.white);

        add(new Button("Red"));
        add(new Button("Blue"));
        add(new Button("Green"));
        add(new Button("White"));
        add(new Button("Black"));
    }

    public boolean action(Event evt, Object arg) {
        if (evt.target instanceof Button)
            changeColor((String)arg);
        return true;
    }

    void changeColor(String bname) {
        if (bname.equals("Red")) setBackground(Color.red);
        else if (bname.equals("Blue")) setBackground(Color.blue);
        else if (bname.equals("Green")) setBackground(Color.green);
        else if (bname.equals("White")) setBackground(Color.white);
        else setBackground(Color.black);
    }
}
```

Figure 13.15.
The ButtonAction applet.



Nesting Panels and Components

Adding UI components to individual applets is fun, but applets begin to turn into lots of fun when you begin working with nested panels. By nesting different panels inside your applet, and panels inside those panels, you can create different layouts for different parts of the overall applet area, isolate background and foreground colors and fonts to individual parts of an applet, and manage the design of your UI components much more cleanly and simply. The more complex the layout of your applet, the more likely you're going to want to use nested panels.

Nested Panels

Panels, as you've already learned, are components that can be actually displayed on screen; `Panel`'s superclass `Container` provides the generic behavior for holding other components inside it. The `Applet` class, which your applets all inherit from, is a subclass of `Panel`. To nest other panels inside an applet, you merely create a new panel and add it to the applet, just as you would add any other UI component:

```
setLayout(new GridLayout(1, 2, 10, 10));
Panel panel1 = new Panel();
Panel panel2 = new Panel();
add(panel1);
add(panel2);
```

You can then set up an independent layout for those subpanels and add AWT components to them (including still more subpanels) by calling the `add()` method in the appropriate panel:

```
panel1.setLayout(new FlowLayout());
panel1.add(new Button("Up"));
panel1.add(new Button("Down"));
```

Although you can do all this in a single class, it's common in applets that make heavy use of the panels to factor out the layout and behavior of the subpanels into separate classes, and to communicate between the panels by using method calls. You'll look at an extensive example of this later on in today's lesson.

Events and Nested Panels

When you create applets with nested panels, those panels form a hierarchy from the outermost panel (the applet, usually), to the innermost UI component. This hierarchy is important to how each component in an applet interacts with the other components in the applet or with the browser that contains that applet; in particular, the component hierarchy determines the order in which components are painted to the screen.

More importantly, the hierarchy also affects event handling, particularly for user input events such as mouse and keyboard events.

Events are received by the innermost component in the component hierarchy and passed up the chain to the root. Suppose, for example, that you have an applet with a subpanel that can handle mouse events (using the `mouseDown()` and `mouseUp()` methods) and that panel contains a button. Clicking on the button means that the button receives the event before the panel does; if the button isn't interested in that `mouseDown()`, the event gets passed to the panel, which can then process it or pass it further up the hierarchy.

Remember the discussion about the basic event methods yesterday? You learned that the basic event methods all return boolean values. Those boolean values become important when you're talking about handling events or passing them on.

An event handling method, whether it is the set of basic event methods or the more generic `handleEvent()`, can do one of three things, given any random event:

- ☐ Not be interested in the event (this is usually true only for `handleEvent()`, which receives all the events generated by the system). If this is the case, the event is passed on up the hierarchy until a component processes it (or it is ignored altogether). In this case, the event handling method should return `false`.
- ☐ Intercept the event, process it, and return `true`. In this case, the event stops with that event method. Recall that this is the case with the basic `mouseDown()` and `keyDown()` methods that you learned about yesterday.
- ☐ Intercept the method, process it, and pass it on to the next event handler. This is a more unusual case, but you may create a user interface by using nested components that will want to do this. In this case, the event method should return `false` to pass the event on to the next handler in the chain.

More UI Components

Once you master the basic UI components and how to add them to panels and manage their events, you can add more UI components. In this section, you'll learn about text areas, scrolling lists, scrollbars, and canvases.

Note that the UI components in this section do not produce actions, so you can't use the `action()` method to handle their behavior. Instead, you have to use a generic `handleEvent()` method to test for specific events that these UI components generate. You'll learn more about this in the next section.

Text Areas

Text areas are like text fields, except they have more functionality for handling large amounts of text. Because text fields are limited in size and don't scroll, they are better for one-line responses and text entry; text areas can be any given width and height and have scrollbars by default, so you can deal with larger amounts of text more easily.

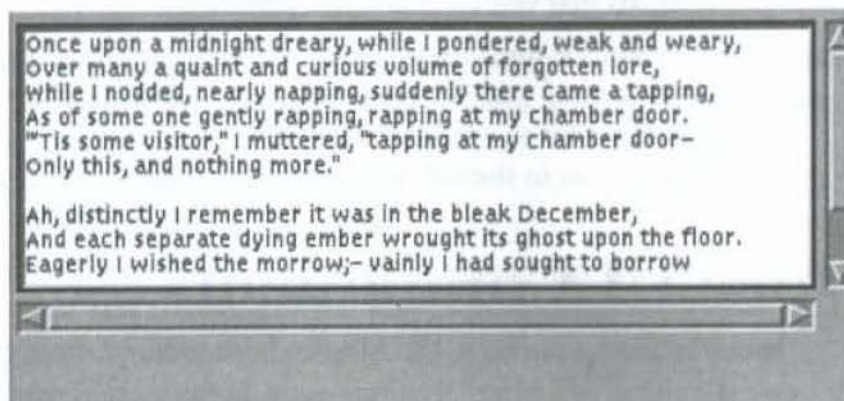
To create a text area, use one of the following constructors:

- ☐ `TextArea()` creates an empty text area 0 rows long and 0 characters wide. Given that a text area with no dimensions can't be displayed, you should make sure you change the dimensions of this new text area before adding it to a panel (or just use the next constructor instead).
- ☐ `TextArea(int, int)` creates an empty text area with the given number of rows and columns (characters).
- ☐ `TextArea(String)` creates a text area displaying the given string, 0 rows by 0 columns.
- ☐ `TextArea(String, int, int)` creates a text area displaying the given string and with the given dimensions.

Figure 13.16 shows a simple text area generated from the following code:

```
String str = "Once upon a midnight dreary, while I pondered, weak and weary,\n" +
    "Over many a quaint and curious volume of forgotten lore,\n" +
    "While I nodded, nearly napping, suddenly there came a tapping,\n" +
    "As of some one gently rapping, rapping at my chamber door.\n" +
    "'Tis some visitor," I muttered, "tapping at my chamber door-\n" +
    "... ;\n";
add(new TextArea(str,10,60));
```

Figure 13.16.
A text area.



Both text areas and text fields inherit from the `TextComponent` class, so a lot of the behavior for text fields (particularly getting and setting text and selections) is usable on text areas as well (refer to Table 13.4). Text areas also have a number of their own methods that you may find useful. Table 13.5 shows a sampling of those methods.

Table 13.5. Text area methods.

Method	Action
<code>getColumns()</code>	Returns the width of the text area, in characters or columns
<code>getRows()</code>	Returns the number of rows in the text area (not the number of rows of text that the text area contains)

Method	Action
<code>insertText(String, int)</code>	Inserts the string at the given position in the text (text positions start at 0)
<code>replaceText(String, int, int)</code>	Replace the text between the given integer positions with the new string
<code>setLineIncrement(int inc)</code>	Change the increment for how far to scroll when the endpoints of the scroll bar are selected. The default is 1.
<code>getLineIncrement()</code>	Returns the increment for how far to scroll when the endpoints of the scroll bar are selected.
<code>setPageIncrement(int inc)</code>	Change the increment for how far to scroll when the inside range of the scroll bar is selected. The default is 10.
<code>getPageIncrement()</code>	Returns the increment for how far to scroll when the inside range of the scroll bar is selected.

Scrolling Lists

Remember the choice menu, which enables you to choose one of several different options? A scrolling list is functionally similar to a choice menu in that it lets you pick several options from a list. Scrolling lists differ in two significant ways:

- ☐ Scrolling lists are not popup menus. They're lists of items in which you can choose one or more items from a list. If the number of items is larger than the list box, a scrollbar is automatically provided so that you can see the other items.
- ☐ A scrolling list can be defined to accept only one item at a time (exclusive), or multiple items (nonexclusive).

To create a scrolling list, create an instance of the `List` class and then add individual items to that list. The `List` class has two constructors:

- ☐ `List()` creates an empty scrolling list that enables only one selection at a time.
- ☐ `List(int, boolean)` creates a scrolling list with the given number of visible lines on the screen (you're unlimited as to the number of actual items you can add to the list). The boolean argument indicates whether this list enables multiple selections (`true`) or not (`false`).

After creating a `List` object, add items to it using the `addItem()` method and then add the list itself to the panel that contains it. Here's an example, the result of which is shown in Figure 13.17:

```
List lst = new List(5, true);
lst.addItem("Hamlet");
```

```
lst.addItem("Claudius");
lst.addItem("Gertrude");
lst.addItem("Polonius");
lst.addItem("Horatio");
lst.addItem("Laertes");
lst.addItem("Ophelia");

add(lst);
```

Figure 13.17.
A scrolling list.

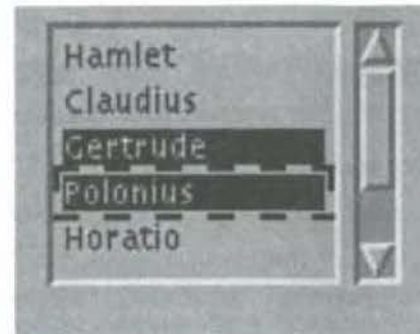


Table 13.6 shows some of the methods available to scrolling lists. See the API documentation for a complete set.

Table 13.6. Scrolling list methods.

Method	Action
<code>getItem(int)</code>	Returns the string item at the given position
<code>countItems()</code>	Returns the number of items in the menu
<code>getSelectedIndex()</code>	Returns the index position of the item that's selected (used for lists that enable only single selections)
<code>getSelectedIndexes()</code>	Returns an array of index positions (used for lists that enable multiple selections)
<code>getSelectedItem()</code>	Returns the currently selected item as a string
<code>getSelectedItems()</code>	Returns an array of strings containing all the selected items
<code>select(int)</code>	Selects the item at the given position
<code>select(String)</code>	Selects the item with that string

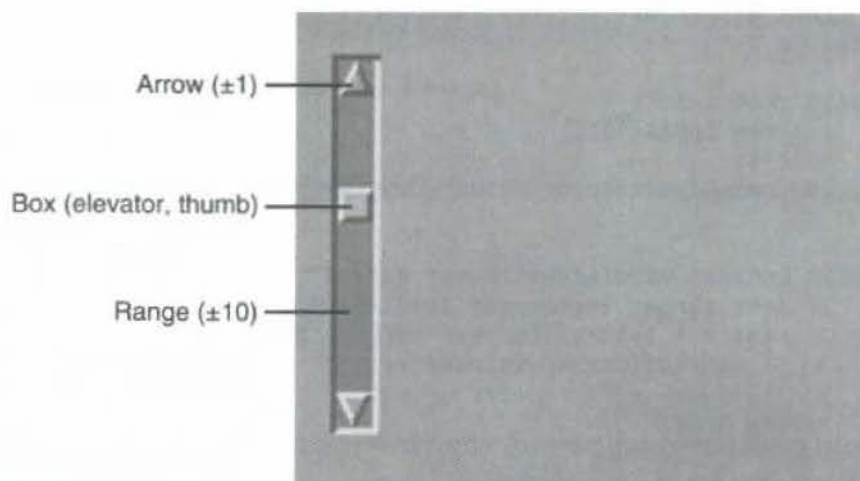
Scrollbars and Sliders

Text areas and scrolling lists come with their own scrollbars, which are built into those UI components and enable you to manage both the body of the area or the list and its scrollbar as a single unit. You can also create individual scrollbars, or sliders, to manipulate a range of values.

Scrollbars are used to select a value between a maximum and a minimum value. To change the current value of that scrollbar, you can use three different parts of the scrollbar (see Figure 13.18):

- ☐ Arrows on either end, which increment or decrement the values by some small unit (1 by default).
- ☐ A range in the middle, which increments or decrements the value by a larger amount (10 by default).
- ☐ A box in the middle, often called an *elevator* or *thumb*, whose position shows where in the range of values the current value is located. Moving this box with the mouse causes an absolute change in the value, based on the position of the box within the scrollbar.

Figure 13.18.
Scrollbar parts.



Choosing any of these visual elements causes a change in the scrollbar's value; you don't have to update anything or handle any events. All you have to do is give the scrollbar a maximum and minimum, and Java will handle the rest.

To create a scrollbar, you can use one of three constructors:

- ☐ `Scrollbar()` creates a scrollbar with 0, 0 as its initial maximum and initial minimum values, in a vertical orientation.
- ☐ `Scrollbar(int)` creates a scrollbar with 0, 0 as its initial maximum and initial minimum values. The argument represents an orientation, for which you can use the class variables `Scrollbar.HORIZONTAL` and `Scrollbar.VERTICAL`.
- ☐ `Scrollbar(int, int, int, int, int)` creates a scrollbar with the following arguments (each one is an integer, and must be presented in this order):
The first argument is the orientation of the scrollbar: `Scrollbar.HORIZONTAL` and `Scrollbar.VERTICAL`.

The second argument is the initial value of the scrollbar, which should be a value between the scrollbar's maximum and minimum values.

The third argument is the overall width (or height, depending on the orientation) of the scrollbar's box. In user interface design, a larger box implies that a larger amount of the total range is currently showing (applies best to things such as windows and text areas).

The fourth and fifth arguments are the minimum and maximum values for the scrollbar.

Here's a simple example of a scrollbar that increments a single value (see Figure 13.19). The label to the left of the scrollbar is updated each time the scrollbar's value changes:

```
import java.awt.*;

public class SliderTest extends java.applet.Applet {
    Label l;

    public void init() {
        l = new Label("0");
        add(l);
        add(new Scrollbar(Scrollbar.HORIZONTAL, 1, 0, 1, 100));
    }

    public boolean handleEvent(Event evt) {
        if (evt.target instanceof Scrollbar) {
            int v = ((Scrollbar)evt.target).getValue();
            l.setText(String.valueOf(v));
        }
        return true;
    }
}
```

Figure 13.19.
A scrollbar.



The `Scrollbar` class provides several methods for managing the values within scrollbars (see Table 13.7).

Table 13.7. Scrollbar methods.

Method	Action
<code>getMaximum()</code>	Returns the maximum value
<code>getMinimum()</code>	Returns the minimum value

Method	Action
<code>getOrientation()</code>	Returns the orientation of this scrollbar: 0 is <code>Scrollbar.HORIZONTAL</code> , 1 is <code>Scrollbar.VERTICAL</code>
<code>getValue()</code>	Returns the scrollbar's current value
<code>setValue(int)</code>	Sets the current value of the scrollbar

Canvases

Although you can draw on most AWT components, such as panels, canvases do little *except* let you draw on them. They can't contain other components, but they can accept events, and you can create animations and display images on them. Canvases, in other words, could have been used for much of the stuff you learned about earlier this week.

NEW TERM A *canvas* is a component that you can draw on.

To create a canvas, use the `Canvas` class and add it to a panel as you would any other component:

```
Canvas can = new Canvas();
add(can);
```

More UI Events

Yesterday, you learned about some basic event types that are generated from user input to the mouse or the keyboard. These event types are stored in the `Event` object as the event ID, and can be tested for in the body of a `handleEvent()` method by using class variables defined in `Event`. For many basic events, such as `mouseDown()` and `keyDown()`, you can define methods for those events to handle the event directly. You learned a similar mechanism today for UI actions where creating an `action()` method handled a specific action generated by a UI component.

The most general way of managing events, however, continues to be the `handleEvent()` method. For events relating to scrollbars and scrolling lists, the only way to intercept these events is to override `handleEvent()`.

To intercept a specific event, test for that event's ID. The available IDs are defined as class variables in the `Event` class, so you can test them by name. You learned about some of the basic events yesterday; Table 13.8 shows additional events that may be useful to you for the components you've learned about today (or that you might find useful in general).

Table 13.8. Additional events.

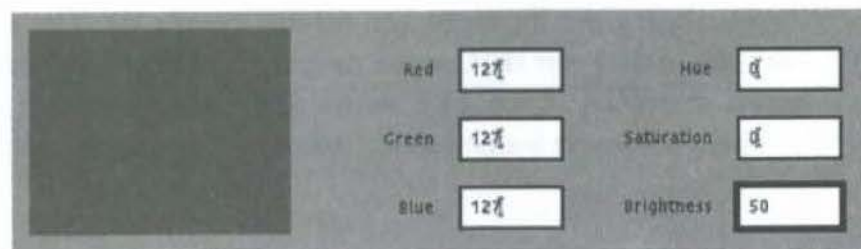
Event ID	What It Represents
ACTION_EVENT	Generated when a UI component action occurs
KEY_ACTION	Generated when text field action occurs
LIST_DESELECT	Generated when an item in a scrolling list is deselected
LIST_SELECT	Generated when an item in a scrolling list is selected
SCROLL_ABSOLUTE	Generated when a scrollbar's box has been moved
SCROLL_LINE_DOWN	Generated when a scrollbar's bottom or left endpoint (button) is selected
SCROLL_LINE_UP	Generated when a scrollbar's top or right endpoint (button) is selected
SCROLL_PAGE_DOWN	Generated when the scrollbar's field below (or to the left of) the box is selected
SCROLL_PAGE_UP	Generated when the scrollbar's field above (or to the right of) the box is selected

A Complete Example: RGB to HSB Converter

Let's take a break here from theory and smaller examples to create a larger, more complex example that puts together much of what you've learned so far. The following applet example demonstrates layouts, nesting panels, creating user interface components, and catching and handling actions, as well as using multiple classes to put together a single applet. In short, it's the most complex applet you'll create so far.

Figure 13.20 shows the applet you'll be creating in this example. The ColorTest applet enables you to pick colors based on RGB (red, green, and blue) and HSB (hue, saturation, and brightness) values.

Figure 13.20.
The ColorTest applet.



The ColorTest applet has three main parts: a colored box on the left side and two groups of text fields on the right. The first group indicates RGB values, the right, HSB. By changing any of the values in any of the text boxes, the colored box is updated to the new color, as are the values in the other group of text boxes.

This applet uses two classes:

- ☐ ColorTest, which inherits from Applet. This is the controlling class for the applet itself.
- ☐ ColorControls, which inherits from Panel. You'll create this class to represent a group of three text fields and to handle actions from those text fields. Two instances of this class, one for the RGB values and one for the HSB ones, will be created and added to the applet.

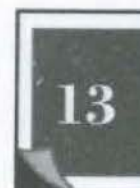
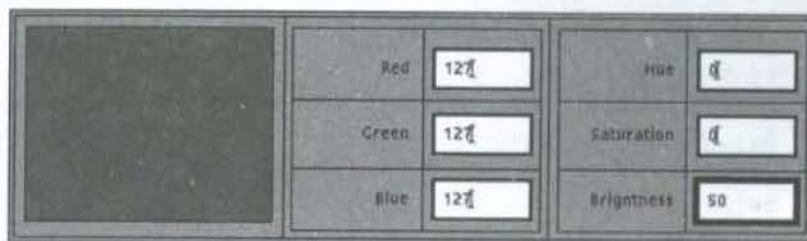
Let's work through this step by step, because it's very complicated and can get confusing. All the code for this applet will be shown at the end of this section.

Create the Applet Layout

The best way to start creating an applet that uses AWT components is to worry about the layout first and then worry about the functionality. When dealing with the layout, you also should start with the outermost panel first and work inward.

Making a sketch of your UI design can help you figure out how to organize the panels inside your applet or window to best take advantage of layout and space. Figure 13.21 shows the ColorTest applet with a grid drawn over it so that you can get an idea of how the panels and embedded panels work.

Figure 13.21.
The ColorTest applet panels and components.



Create the Panel Layout

Let's start with the outermost panel—the applet itself. This panel has three parts: the color box on the left, the RGB text fields in the middle, and the HSB fields on the right.

Because this is the applet, your ColorTest class will be the applet class and inherit from Applet. You'll also import the AWT classes here (note that because you use so many of them in this program, it's easiest to just import the entire package):

```
import java.awt.*;

public class ColorTest extends java.applet.Applet {
    ...
}
```

Let's start with the `init()` method, where all the basic initialization and layout takes place. There are four major steps:

1. Set the layout for the big parts of the panel. Although a flow layout would work, a grid layout with one row and three columns is a much better idea.
2. Create the three components of this applet: a canvas for the color box and two subpanels for the text fields.
3. Add those components to the applet.
4. Finally, initialize the default color and update all the panels to reflect that default color.

Before you do any of that, let's set up instance variables to hold the three major components of this applet. You need to keep hold of these objects so you can update things when a value changes.

The color box is easy—it's just a canvas. Call it `swatch`.

```
Canvas swatch;
```

Now onto the subpanels. There are two of them, and although they have different labels and values, they're essentially the same panel. You could just create code for each one here, but you'd end up duplicating a lot of the same code. This is a perfect opportunity, therefore, to create another class to represent the subpanels with the text fields on them. Call them `ColorControls` (you'll get around to creating the class later) and define two variables, `rgbControls` and `hsbControls`, to hold them:

```
ColorControls rgbControls, hsbControls;
```

Back to the `init()` method. Step one is the layout. Let's use a grid layout and a gap of ten points to separate each of the components:

```
setLayout(new GridLayout(1, 3, 10, 10));
```

Step two is creating the components, the canvas first. You have an instance variable to hold that one:

```
swatch = new Canvas();
```

You need to create two instances of your as-of-yet nonexistent `ColorControls` panels here as well, but you don't know exactly what you need to create them yet, so let's put in some basic constructors and fill in the details later:

```
rgbControls = new ColorControls()
hsbControls = new ColorControls();
```

Step three is adding them to the panel.

```
add(swatch);
add(rgbControls);
add(hsbControls);
```

While you're working on layout, add an inset just for fun—ten points along all the edges:

```
public Insets insets() {
    return new Insets(10, 10, 10, 10);
}
```

Got it so far? Now you have a skeleton `init()` method and an `insets()` method in your `ColorTest` class. Let's move on now to creating the subpanel layout—to creating that `ColorControls` class.

Define the Subpanels

The `ColorControls` class will have behavior for laying out and handling the subpanels that represent the RGB and HSB values for the color. `ColorControls` doesn't need to be a subclass of `Applet` because it isn't actually an applet, it's just a panel. Define it to inherit from `Panel`:

```
class ColorControls extends Panel {
    ...
}
```

Note: You can put the `ColorControls` class in the same file as the `ColorTest` class. You haven't been doing this so far because the applets and applications you've been creating had only one class. If you remember way back to Day 1, however, you learned that you can have multiple class definitions in a single file as long as only one of those definitions is declared `public`. In this case, the `ColorTest` class is `public` (it's an applet, so it has to be), but the `ColorControls` class doesn't need to be, so everything works out fine. Normally, you'd put `ColorControls` in a separate file, anyway.

13

You need a couple of instance variables in this class. The first thing you need is a hook back up to the applet class that contains this panel. Why? The applet class is the class that oversees how the subcomponents work, so it's going to be the class that updates everything. Eventually, you're going to have to call a method in that class to indicate that something in this panel has changed. Without an actual reference to that outer class, there's no way to do this. So, instance variable number one is a reference to the class `ColorTest`:

```
ColorTest outerparent;
```

If you figure that the applet class is the one that's going to be updating everything, that class is going to need a way to get hold of the pieces inside this class. In particular, it's going to be

interested in the individual text fields, so you're going to need instance variables to hold those. This creates three of them:

```
TextField f1, f2, f3;
```

Now for the constructor for this class. Again, this isn't an applet, so you don't use `init()`; all you need is a constructor method.

What do you need inside that constructor? You need to set the layout for the subpanel, create the text fields, and add them to the panel. The goal here is to make the `ColorControls` class generic enough so that you can use it for both the RGB fields and the HSB fields.

The two different panels differ in two respects: the labels for the text fields, and the initial values for the text fields. That's six values to get before you can create the object. You can pass those six values in through the constructors in `ColorTest`. You also need one more. Because you need that hook back to the applet class, you should also pass in a reference to that object as part of the constructor.

You now have seven arguments to the basic constructor for the `ColorControls` class. Here's the signature for that constructor:

```
ColorControls(ColorTest target,  
             String l1, String l2, String l3,  
             int v1, int v2, int v3) {  
}
```

Given those arguments, you can assign the right values to your instance variables:

```
outerparent = target;  
  
f1 = new TextField(String.valueOf(v1), 10);  
f2 = new TextField(String.valueOf(v2), 10);  
f3 = new TextField(String.valueOf(v3), 10);
```

Note that because the first argument to the `TextField` constructor is a string, and the values that you passed in were integers, you have to use the `valueOf()` class method (defined in `String`) to convert the integer to a string before creating each text field.

Next, you create the layout for this panel. You also use a grid layout for these subpanels, as you did for the applet panel, but this time the grid will have three rows (one for each of the text field and label pairs) and two columns (one for the labels and one for the fields).

Given the 3-by-2 grid, you can now add the text fields and labels to that panel. Note that by separating the labels and the text fields into separate cells in the grid, you can align the labels, creating a nice aligned layout.

```
add(new Label(l1, Label.RIGHT));  
add(f1);  
add(new Label(l2, Label.RIGHT));  
add(f2);
```

```
add(new Label(13, Label.RIGHT));
add(f3);
```

Finally (because I like insets), you'll inset the contents of the subpanel a bit—only on the top and bottom edges—by including an `insets()` method:

```
public Insets insets() {
    return new Insets(10, 10, 0, 0);
}
```

You're almost there. You have 98 percent of the layout in place and ready to go, but you're missing two things: creating the `ColorControls` objects in `ColorTest`, and initializing everything so that all the components have the right values.

For both, you need to go back to the `ColorTest` class and the `init()` method you defined there. Let's start with the initialization part, because that's easy. The default color is black. Set up a local variable to hold that color object:

```
Color theColor = new Color(0, 0, 0);
```

To set the initial color of the color box, all you need to do is set its background:

```
swatch.setBackground(theColor);
```

Now, let's finally tackle initializing those subpanels. The constructor for `ColorControls` has seven arguments: the `ColorTest` object, three labels (strings), and three initial values for the text fields (integers). Let's do the RGB controls first, because you can easily extract the initial red, green, and blue values out of the `Color` object:

```
rgbControls = new ColorControls(this, "Red", "Green", "Blue",
    theColor.getRed(), theColor.getGreen(),
    theColor.getBlue());
```

Things get complicated on the HSB side of the panel. The `Color` class provides you with a method to get the HSB values out of a `Color` object, but there are two problems:

- ☐ The `RGBtoHSB()` method is a single class method that insists on returning an array of the three values.
- ☐ The HSB values are measured in floating-point values. I prefer to think of HSB as integers, wherein the hue is a degree value around a color wheel (0 through 360), and saturation and brightness are percentages from 0 to 100. Having HSB as integer values also enables you to have a generic subpanel, as was the intent.

Initializing the HSB subpanel is going to be a little difficult.

First, let's extract those HSB values. Given that the method takes three RGB arguments—an array of three floats—and returns an array of three floats, you have to go through this process to get those values:

```
float[] HSB = Color.RGBtoHSB(theColor.getRed(),
    theColor.getGreen(), theColor.getBlue(), (new float[3]));
```

Now you have an array of floats, where `hsb[0]` is the hue, `hsb[1]` is the saturation, and `hsb[2]` is the brightness. You can now (finally!) initialize the HSB side of the applet, making sure that when you pass those HSB values into the subpanel, you multiply them by the right values (360 for the hues, 100 for the saturation and the brightness) and convert them to integers:

```
hsbcontrols = new ColorControls(this,
    "Hue", "Saturation", "Brightness",
    (int)(hsb[0] * 360), (int)(hsb[1] * 100),
    (int)(hsb[2] * 100));
```

Ready to give up? Fear not—you've done the hard part. From here, it's (mostly) easy. Once you have your layout working, you can compile your Java program and see how it looks. None of your UI components actually does anything, but perfecting the layout is half the battle.

Handle the Actions

After creating the layout, you set up actions with the UI components so that when the user interacts with the applet, the applet can respond.

The action of this applet occurs when the user changes a value in any of the text fields. By causing an action in a text field, the color changes, the color box updates to the new color, and the values of the fields in the opposite subpanel change to reflect the new color.

The `ColorTest` class is responsible for actually doing the updating, because it keeps track of all the subpanels. You should be tracking and intercepting events in the subpanel in which they occur, however. Because the action of the applet is an actual text action, you can use an `action()` method to intercept it:

```
public boolean action(Event evt, Object arg) {
    if (evt.target instanceof TextField) {
        outerparent.update(this);
        return true;
    }
    else return false;
}
```

In the `action()` method, you test to make sure the action was indeed generated by a text field (because there are only text fields available, that's the only action you'll get, but it's a good idea to test for it anyhow). If so, call the `update()` method, defined in `ColorTest`, to update the applet to reflect all the new values. Because the outer applet is responsible for doing all the updating, this is precisely why you need that hook back to the applet—so you can call the right method at the right time.

Update the Result

The only part left now is to update all the values and the color swatch if one of the values changes. For this, you define the `update()` method in the `ColorTest` class. This `update()` method takes

a single argument—the `ColorControls` instance that contains the changed value (you get that argument from the `action()` method in the subpanel).



Note: Won't this `update()` method interfere with the system's `update()` method? Nope. Remember, methods can have the same names, but different signatures and definitions. Because this `update()` has a single argument of type `ColorControls`, it doesn't interfere with the other version of `update()`. Normally, all methods called `update()` should mean basically the same thing; it's not true here, but it's only an example.

The `update()` method is responsible for updating all the panels in the applet. To know which panel to update, you need to know which panel changed. You can find out by testing to see whether the argument you got passed is the same as the subpanels you have stored in the `RGBcontrols` and `HSBcontrols` instance variables:

```
void update(ColorControls in) {
    if (in == rgbControls) { // the change was in rgb
        ...
    }
    else { // change was in hsb
    }
}
```

This test is the heart of the `update()` method. Let's start with that first case—a number has been changed in the RGB text fields. So now, based on those new RGB values, you have to generate a new color object and update the values on the HSB panel. To reduce some typing, you create a few local variables to hold some basic values. In particular, the values of the text fields are strings, and you get into them by accessing the text field instance variables for the `ColorControls` panel (`f1`, `f2`, `f3`) and then using the `getText()` method to extract the actual values. Extract those values and store them in string variables so that you don't have to keep typing:

```
String v1 = in.f1.getText();
String v2 = in.f2.getText();
String v3 = in.f3.getText();
```

Given those string values for RGB, you now create a color object by converting those strings to integers:

```
Color c;
c = new Color(Integer.parseInt(v1), Integer.parseInt(v2), Integer.parseInt(v3));
```



Note: This part of the example isn't very robust; it assumes that the user has indeed entered real numbers into the text fields. A better version of this would test to make sure that no parsing errors had occurred (I was trying to keep this example small).

When you have a color object, you can update the color swatch:

```
swatch.setBackground(c);
```

The next step is to update the HSB panel to the new HSB values. Doing this in the `init()` method is no fun at all, and it's even less fun here. To do this, you call `RGBtoHSB` to get the floating-point values, convert them to integers with the right values, convert them to strings, and then put them back into the text fields for the HSB subpanel. Got all that? Here's the code:

```
float[] HSB = Color.RGBtoHSB(c.getRed(), c.getGreen(),
    c.getBlue(), (new float[3]));
hsb[0] *= 360;
hsb[1] *= 100;
hsb[2] *= 100;
hsbControls.f1.setText(String.valueOf((int)hsb[0]));
hsbControls.f2.setText(String.valueOf((int)hsb[1]));
hsbControls.f3.setText(String.valueOf((int)hsb[2]));
```

The second part of the `update()` method is called when a value on the HSB side of the panel is changed. This is the "else" in the if-else that determines what to update, given a change.

Believe it or not, it's easier to update RGB values given HSB than it is to do it the other way around. First, convert the string values from the HSB text fields to integers by using these lines:

```
int f1 = Integer.parseInt(v1);
int f2 = Integer.parseInt(v2);
int f3 = Integer.parseInt(v3);
```

There's a class method in the `Color` class that creates a new color object when given three HSB values. The catch is that those values are floats, and they're not the values you currently have. To call `getHSBColor()` (that's the name of the method), convert the integers to floats and divide by the right amounts:

```
c = Color.getHSBColor((float)f1 / 360, (float)f2 / 100, (float)f3/100);
```

Now that you have a color object, the rest is easy. Set the color swatch:

```
swatch.setBackground(c);
```

Then update the RGB text fields with the new RGB values from the color object:

```
rgbControls.f1.setText(String.valueOf(c.getRed()));
rgbControls.f2.setText(String.valueOf(c.getGreen()));
rgbControls.f3.setText(String.valueOf(c.getBlue()));
```

The Complete Source Code

Listing 13.1 shows the complete source code; often it's easier to figure out what's going on in this applet when it's all in one place and you can follow the method calls and how values are passed back and forth. Start with the `init()` method in applet, and go from there.



Listing 13.1. The ColorTest applet.

```

1: import java.awt.*;
2:
3: public class ColorTest extends java.applet.Applet {
4:     ColorControls rgbControls, hsbControls;
5:     Canvas swatch;
6:
7:     public void init() {
8:         Color theColor = new Color(0, 0, 0);
9:         float[] hsb = Color.RGBtoHSB(theColor.getRed(),
10:            theColor.getGreen(), theColor.getBlue(),
11:            (new float[3]));
12:
13:         setLayout(new GridLayout(1, 3, 10, 10));
14:
15:         // The color swatch
16:         swatch = new Canvas();
17:         swatch.setBackground(theColor);
18:
19:         // the control panels
20:         rgbControls = new ColorControls(this,
21:            "Red", "Green", "Blue",
22:            theColor.getRed(), theColor.getGreen(),
23:            theColor.getBlue());
24:
25:         hsbControls = new ColorControls(this,
26:            "Hue", "Saturation", "Brightness",
27:            (int)(hsb[0] * 360), (int)(hsb[1] * 100),
28:            (int)(hsb[2] * 100));
29:
30:         add(swatch);
31:         add(rgbControls);
32:         add(hsbControls);
33:
34:     }
35:
36:     public Insets insets() {
37:         return new Insets(10, 10, 10, 10);
38:     }
39:
40:     void update(ColorControls in) {
41:         Color c;
42:         String v1 = in.f1.getText();
43:         String v2 = in.f2.getText();
44:         String v3 = in.f3.getText();
45:
46:         if (in == rgbControls) { // change to RGB
47:             c = new Color(Integer.parseInt(v1),
48:                Integer.parseInt(v2),
49:                Integer.parseInt(v3));
50:             swatch.setBackground(c);

```

Listing 13.1. continued

```

51:         float[] HSB = Color.RGBtoHSB(c.getRed(),c.getGreen(),
52:             c.getBlue(), (new float[3]));
53:         hsb[0] *= 360;
54:         hsb[1] *= 100;
55:         hsb[2] *= 100;
56:         hsbControls.f1.setText(String.valueOf((int)HSB[0]));
57:         hsbControls.f2.setText(String.valueOf((int)HSB[1]));
58:         hsbControls.f3.setText(String.valueOf((int)HSB[2]));
59:     }
60:     else { // change to HSB
61:         int f1 = Integer.parseInt(v1);
62:         int f2 = Integer.parseInt(v2);
63:         int f3 = Integer.parseInt(v3);
64:         c = Color.getHSBColor((float)f1 / 360,
65:             (float)f2 / 100, (float)f3/100);
66:         swatch.setBackground(c);
67:         RGBcontrols.f1.setText(String.valueOf(c.getRed()));
68:         RGBcontrols.f2.setText(String.valueOf(
69:             c.getGreen()));
70:         RGBcontrols.f3.setText(String.valueOf(c.getBlue()));
71:     }
72: }
73: }
74:
75:
76: class ColorControls extends Panel {
77:     TextField f1, f2, f3;
78:     ColorTest outerparent;
79:
80:     ColorControls(ColorTest target,
81:         String l1, String l2, String l3,
82:         int v1, int v2, int v3) {
83:
84:         this.outerparent = target;
85:         setLayout(new GridLayout(3,4,10,10));
86:
87:         f1 = new TextField(String.valueOf(v1),10);
88:         f2 = new TextField(String.valueOf(v2),10);
89:         f3 = new TextField(String.valueOf(v3),10);
90:
91:         add(new Label(l1, Label.RIGHT));
92:         add(f1);
93:         add(new Label(l2, Label.RIGHT));
94:         add(f2);
95:         add(new Label(l3, Label.RIGHT));
96:         add(f3);
97:     }
98:
99:     public Insets insets() {
100:         return new Insets(10,10,0,0);
101:     }
102:
103:     public boolean action(Event evt, Object arg) {
104:         if (evt.target instanceof TextField) {

```

```

105:         outerparent.update(this);
106:         retruue true;
107:     }
108:     else return false;
109: }
110: }

```

Summary

The Java AWT, or Abstract Windowing Toolkit, is a package of Java classes and interfaces for creating full-fledged access to a window-based graphical user interface system, with mechanisms for graphics display, event management, text and graphics primitives, user interface components, and cross-platform layout. The AWT is used by the HotJava browser itself for all its functionality. Applets are also an integral part of the AWT toolkit.

Today has been a big day; the lesson has brought together everything you've learned up to this point about simple applet management and added a lot more about creating applets, panels, and user interface components and managing the interactions between all of them. With the information you got today and the few bits that you'll learn tomorrow, you can create cross-platform Java applications that do just about anything you want.

Q&A

Q You've mentioned a lot about the `Component` and `Container` classes, but it looks like the only `Container` objects that ever get created are `Panels`. What do the `Component` and `Container` classes give me?

A Those classes factor out the behavior for components (generic AWT components) and containers (components that can contain other components). Although you don't necessarily create direct instances of these classes, you can create subclasses of them if you want to add behavior to the AWT that the default classes do not provide. As with most of the Java classes, any time you need a superclass's behavior, don't hesitate to extend that class by using your own subclass.

Q Can I put a UI component at a specific x and y position on the screen?

A By using the existing layout managers supplied with the AWT toolkit, no. This is actually a good thing because you don't know what kind of display environment your applet will be run under, what kind of fonts are installed, or what kind of fonts are being currently used. By using the layout managers provided with the AWT, you can be reasonably sure that every portion of your window will be viewable and readable and usable (fonts may cause you trouble). You can't guarantee anything like that with hard-coded layouts.

Q I was exploring the AWT package, and I saw this subpackage called `peer`. There's also references to the peer classes sprinkled throughout the API documentation. What do peers do?

A Peers are responsible for the platform-specific parts of the AWT. For example, when you create a Java AWT window, you have an instance of the `Window` class that provides generic `Window` behavior, and then you have an instance of a class implementing `WindowPeer` that creates the very specific window for that platform—a motif window under X Window, a Macintosh-style window under the Macintosh, or a Windows 95 window under Windows 95. These “peers” classes also handle communication between the window system and the Java window itself. By separating the generic component behavior (the AWT classes) from the actual system implementation and appearance (the peer classes), you can focus on providing behavior in your Java application and let the Java implementation deal with the platform-specific details.

Q There's a whole lot of functionality in the AWT that you haven't talked about here. Why?

A Given that even a basic introduction took this long, I figured that if I put in even more detail than I already have that this book would turn into *Teach Yourself Java in 21 Days Plus a Few Extra for the AWT Stuff*.

As it is, I've left windows, menus, and dialogs until tomorrow, so you'll have to wait for those. But you can find out about a lot of the other features of AWT merely by exploring the API documentation. Start with the `Applet` class and examine the sorts of methods you can call. Then look at `Panel`, from which `Applet` inherits—you have all that class's functionality as well. The superclass of `Panel` is `Container`, which provides still more interesting detail. `Component` comes next. Explore the API and see what you can do with it. You might find something interesting.



18

WEEK
3

Multithreading

by Charles L. Perkins

Today, you'll learn more about the threads mentioned briefly in Week 2:

- ☐ How to "think multithreaded"
- ☐ How to protect your methods and variables from unintended thread conflicts
- ☐ How to create, start, and stop threads and threaded classes
- ☐ How the scheduler works in Java

First, let's begin by motivating the need for threads.

Threads are a relatively recent invention in the computer science world. Although processes, their larger parent, have been around for decades, threads have only recently been accepted into the mainstream. What's odd about this is that they are extremely valuable, and programs written with them are noticeably better, even to the casual user. In fact, some of the best individual, Herculean efforts over the years have involved implementing a threads-like facility by hand to give a program a more friendly feel to its users.

Imagine that you're using your favorite text editor on a large file. When it starts up, does it need to examine the entire file before it lets you edit? Does it need to make a copy of the file? If the file is huge, this can be a nightmare. Wouldn't it be nicer for it to show you the first page, enabling you to begin editing, and somehow (in the background) complete the slower tasks necessary for initialization? Threads allow exactly this kind of within-the-program parallelism.

Perhaps the best example of threading (or lack of it) is a WWW browser. Can your browser download an indefinite number of files and Web pages at once while still enabling you to continue browsing? While these pages are downloading, can your browser download all the pictures, sounds, and so forth in parallel, interleaving the fast and slow download times of multiple Internet servers? HotJava can do all of these things—and more—by using the built-in threading of the Java language.

The Problem with Parallelism

If threading is so wonderful, why doesn't every system have it? Many modern operating systems have the basic primitives needed to create and run threads, but they are missing a key ingredient. The rest of their environment is not *thread-safe*. Imagine that you are in a thread, one of many, and each of you is sharing some important data managed by the system. If you were managing that data, you could take steps to protect it (as you'll see later today), but the system is managing it. Now visualize a piece of code in the system that reads some crucial value, thinks about it for a while, and then adds 1 to the value:

```
if (crucialValue > 0) {
    . . . // think about what to do
    crucialValue += 1;
}
```

Remember that any number of threads may be calling upon this part of the system at once. The disaster occurs when two threads have both executed the `if` test before either has incremented the `crucialValue`. In that case, the value is clobbered by them both with the same `crucialValue + 1`, and one of the increments has been lost. This may not seem so bad to you, but imagine instead that the crucial value affects the state of the screen as it is being displayed. Now, unfortunate ordering of the threads can cause the screen to be updated incorrectly. In the same way, mouse or keyboard events can be lost, databases can be inaccurately updated, and so forth.

This disaster is inescapable if any significant part of the system has not been written with threads in mind. Therein lies the barrier to a mainstream threaded environment—the large effort required to rewrite existing libraries for thread safety. Luckily, Java was written from scratch with this in mind, and every Java class in its library is thread-safe. Thus, you now have to worry only about your own synchronization and thread-ordering problems, because you can assume that the Java system will do the right thing.

NEW TERM *Atomic operations* are operations that appear to happen “all at once”—exactly at the same time—to other threads.



Note: Some readers may wonder what the fundamental problem really is. Can't you just make the ... area in the example smaller and smaller to reduce or eliminate the problem? Without atomic operations, the answer is no. Even if the ... took zero time, you must first look at the value of some variable to make any decision and then change something to reflect that decision. These two steps can never be made to happen at the same time without an atomic operation. Unless you're given one by the system, it's literally impossible to create your own.

Even the one line `crucialValue += 1` involves three steps: get the current value, add one to it, and store it back. (Using `++crucialValue` doesn't help either.) All three steps need to happen “all at once” (atomically) to be safe. Special Java primitives, at the lowest levels of the language, provide you with the basic atomic operations you need to build safe, threaded programs.

18

Thinking Multithreaded

Getting used to threads takes a little while and a new way of thinking. Rather than imagining that you always know exactly what's happening when you look at a method you've written, you have to ask yourself some additional questions. What will happen if more than one thread calls

into this method at the same time? Do you need to protect it in some way? What about your class as a whole? Are you assuming that only one of its methods is running at the same time?

Often you make such assumptions, and a local instance variable will be messed up as a result. Let's make a few mistakes and then try to correct them. First, the simplest case:

```
public class ThreadCounter {
    int crucialValue;

    public void countMe() {
        crucialValue += 1;
    }

    public int howMany() {
        return crucialValue;
    }
}
```

This code suffers from the most pure form of the "synchronization problem:" the += takes more than one step, and you may miscount the number of threads as a result. (Don't worry about how threads are created yet, just imagine that a whole bunch of them are able to call countMe(), at once, at slightly different times.) Java allows you to fix this:

```
public class SafeThreadCounter {
    int crucialValue;

    public synchronized void countMe() {
        crucialValue += 1;
    }

    public int howMany() {
        return crucialValue;
    }
}
```

The synchronized keyword tells Java to make the block of code in the method thread safe. Only one thread will be allowed inside this method at once, and others have to wait until the currently running thread is finished with it before they can begin running it. This implies that synchronizing a large, long-running method is almost always a bad idea. All your threads would end up stuck at this bottleneck, waiting single file to get their turn at this one slow method.

It's even worse than you might think for unsynchronized variables. Because the compiler can keep them around in registers during computations, and a thread's registers can't be seen by other threads (especially if they're on another processor in a true multiprocessor computer), a variable can be updated in such a way that *no possible order* of thread updates could have produced the result. This is completely incomprehensible to the programmer. To avoid this bizarre case, you can label a variable `volatile`, meaning that you know it will be updated asynchronously by multiprocessor-like threads. Java then loads and stores it each time it's needed and does not use registers.

Note: In earlier releases, variables that were safe from these bizarre effects were labeled `threadsafe`. Because most variables are safe to use, however, they are now assumed to be thread-safe unless you mark them `volatile`. Using `volatile` is an extremely rare event. In fact, in the 1.0 release, the Java library does not use `volatile` anywhere.

Points About *Points*

The method `howMany()` in the last example doesn't need to be synchronized, because it simply returns the current value of an instance variable. Someone higher in the call chain may need to be synchronized, though—someone who uses the value returned from the method. Here's an example:

```
public class Point {           //redefines class Point from package java.awt
    private float x, y;        //OK since we're in a different package here

    public float x() {         // needs no synchronization
        return x;
    }

    public float y() {         // ditto
        return y;
    }
    . . . // methods to set and change x and y
}

public class UnsafePointPrinter {
    public void print(Point p) {
        System.out.println("The point's x is " + p.x()
                           + " and y is " + p.y() + ".");
    }
}
```

18

The analogous methods to `howMany()` are `x()` and `y()`. They need no synchronization, because they just return the values of instance variables. It is the responsibility of the caller of `x()` and `y()` to decide whether it needs to synchronize itself—and in this case, it does. Although the method `print()` simply reads values and prints them out, it reads *two* values. This means that there is a chance that some other thread, running between the call to `p.x()` and the call to `p.y()`, could have changed the value of `x` and `y` stored inside the `Point p`. Remember, you don't know how many other threads have a way to reach and call methods in this `Point` object! "Thinking multithreaded" comes down to being careful any time you make an assumption that something has *not* happened between two parts of your program (even two parts of the same line, or the same expression, such as the string + expression in this example).

TryAgainPointPrinter

You could try to make a safe version of `print()` by simply adding the `synchronized` keyword modifier to it, but instead, let's try a slightly different approach:

```
public class TryAgainPointPrinter {
    public void print(Point p) {
        float safeX, safeY;

        synchronized(this) {
            safeX = p.x();    // these two lines now
            safeY = p.y();    // happen atomically
        }
        System.out.print("The point's x is " + safeX
                        + " y is " + safeY);
    }
}
```

The `synchronized` statement takes an argument that says what object you would like to lock to prevent more than one thread from executing the enclosed block of code at the same time. Here, you use `this` (the instance itself), which is exactly the object that would have been locked by the `synchronized` method as a whole if you had changed `print()` to be like your `safe countMe()` method. You have an added bonus with this new form of synchronization: you can specify exactly what part of a method needs to be safe, and the rest can be left unsafe.

Notice how you took advantage of this freedom to make the protected part of the method as small as possible, while leaving the `String` creations, concatenations, and printing (which together take a small but nonzero amount of time) outside the "protected" area. This is both good style (as a guide to the reader of your code) and more efficient, because fewer threads get stuck waiting to get into protected areas.

SafePointPrinter

The astute reader, though, may still be worried by the last example. It seems as if you made sure that no one executes *your* calls to `x()` and `y()` out of order, but have you prevented the `Point p` from changing out from under you? The answer is no, you still have not solved the problem. You really do need the full power of the `synchronized` statement:

```
public class SafePointPrinter {
    public void print(Point p) {
        float safeX, safeY;

        synchronized(p) {    // no one can change p
            safeX = p.x();    // while these two lines
            safeY = p.y();    // are happening atomically
        }
        System.out.print("The point's x is " + safeX
                        + " y is " + safeY);
    }
}
```

Now you've got it. You actually needed to protect the `Point p` from changes, so you lock it by giving it as the argument to your synchronized statement. Now when `x()` and `y()` happen together, they can be sure to get the current `x` and `y` of the `Point p`, without any other thread being able to call a modifying method between. You're still assuming, however, that the `Point p` has properly protected *itself*. (You can always assume this about system classes—but *you* wrote this `Point` class.) You can make sure by writing the only method that can change `x` and `y` inside `p` yourself:

```
public class Point {
    private float x, y;

    . . .          // the x() and y() methods

    public synchronized void setXAndY(float newX, float newY) {
        x = newX;
        y = newY;
    }
}
```

By making the only “set” method in `Point` synchronized, you guarantee that any other thread trying to grab the `Point p` and change it out from under you has to wait: you've locked the `Point p` with your `synchronized(p)` statement, and any other thread has to try to lock the same `Point p` via the implicit `synchronized(this)` statement `p` now executes when entering `setXAndY()`. Thus, at last, you are thread-safe.

18

Note: By the way, if Java had some way of returning more than one value at once, you could write a synchronized `getXAndY()` method for `Points` that returns both values safely. In the current Java language, such a method could return a new, unique `Point` to guarantee to its callers that no one else has a copy that might be changed. This sort of trick can be used to minimize the parts of the system that need to worry about synchronization.

ReallySafePoint

An added benefit of the use of the `synchronized` modifier on methods (or of `synchronized(this)` { . . . }) is that only one of these methods (or blocks of code) can run at once. You can use that knowledge to guarantee that only one of several crucial methods in a class will run at once:

```
public class ReallySafePoint {
    private float x, y;

    public synchronized Point getUniquePoint() {
        return new Point(x, y);    // can be a less safe Point
    }                             // because only the caller has it
}
```

```

public synchronized void setXAndY(float newX, float newY) {
    x = newX;
    y = newY;
}

public synchronized void scale(float scaleX, float scaleY) {
    x *= scaleX;
    y *= scaleY;
}

public synchronized void add(ReallySafePoint aRSP) {
    Point p = aRSP.getUniquePoint();

    x += p.x();
    y += p.y();
} // Point p is soon thrown away by GC; no one else ever saw it
}

```

This example combines several of the ideas mentioned previously. To avoid a caller's having to `synchronize(p)` whenever getting your `x` and `y`, you give them a synchronized way to get a unique `Point` (like returning multiple values). Each method that modifies the object's instance variables is also synchronized to prevent it from running between the `x` and `y` references in `getUniquePoint()` and from stepping on one another as they modify the local `x` and `y`. Note that `add()` itself uses `getUniquePoint()` to avoid having to say `synchronized(aRSP)`.

Classes that are this safe are a little unusual; it is more often your responsibility to protect yourself from other threads' use of commonly held objects (such as `Points`). Only when you know for certain that you're the only one that knows about an object, can you fully relax. Of course, if you created the object yourself and gave it to no one else, you *can* be that certain.

Protecting a Class Variable

Finally, suppose you want a class variable to collect some information across all a class's instances:

```

public class StaticCounter {
    private static int crucialValue;

    public synchronized void countMe() {
        crucialValue += 1;
    }
}

```

Is this safe? If `crucialValue` were an instance variable, it would be. Because it's a class variable, however, and there is only one copy of it for all instances, you can still have multiple threads modifying it by using different *instances* of the class. (Remember, the `synchronized` modifier locks the object this—an instance.) Luckily, you already know the tools you need to solve this:

```

public class StaticCounter {
    private static int crucialValue;
}

```

```

public void countMe() {
    synchronized(getClass()) { // can't directly name StaticCounter
        crucialValue += 1;      // the (shared) class is now locked
    }
}

```

The trick is to “lock” on a different object—not on an instance of the class, but on the class itself. Because a class variable is “inside” a class, just as an instance variable is inside an instance, this shouldn’t be all that unexpected. In a similar way, classes can provide global resources that any instance (or other class) can access directly by using the class name, and lock by using that same class name. In the last example, `crucialValue` was used from within an instance of `StaticCounter`, but if `crucialValue` were declared `public` instead, from anywhere in the program, it would be safe to say the following:

```

synchronized(Class.forName("StaticCounter")) {
    StaticCounter.crucialValue += 1;
}

```

Note: The direct use of another classes (object’s) variable is really bad style—it’s used here simply to demonstrate a point quickly. `StaticCounter` would normally provide a `countMe()`-like class method of its own to do this sort of dirty work.

18

You can now begin to appreciate how much work the Java team has done for you by thinking all these hard thoughts for each and every class (and method!) in the Java class library.

Creating and Using Threads

Now that you understand the power (and the dangers) of having many threads running at once, how are those threads actually created?

Caution: The system itself always has a few so-called *daemon* threads running, one of which is constantly doing the tedious task of garbage collection for you in the background. There is also a main user thread that listens for events from your mouse and keyboard. If you’re not careful, you can sometimes lock up this main thread. If you do, no events are sent to your program and it appears to be dead. A good rule of thumb is that whenever you’re doing something that *can* be done in a separate thread, it probably *should* be. Threads in Java are relatively cheap to create, run, and destroy, so don’t use them too sparingly.

Because there is a class `java.lang.Thread`, you might guess that you could create a thread of your own by subclassing it—and you are right:

```
public class MyFirstThread extends Thread { // a.k.a., java.lang.Thread
    public void run() {
        . . . // do something useful
    }
}
```

You now have a new type of `Thread` called `MyFirstThread`, which does something useful (unspecified) when its `run()` method is called. Of course, no one has created this thread or called its `run()` method, so it does absolutely nothing at the moment. To actually create and run an instance of your new thread class, you write the following:

```
MyFirstThread aMFT = new MyFirstThread();

aMFT.start(); // calls our run() method
```

What could be simpler? You create a new instance of your thread class and then ask it to start running. Whenever you want to stop the thread, you use this:

```
aMFT.stop();
```

Besides responding to `start()` and `stop()`, a thread can also be temporarily suspended and later resumed:

```
Thread t = new Thread();

t.suspend();
. . . // do something special while t isn't running
t.resume();
```

A thread will automatically `suspend()` and then `resume()` when it's first blocked at a synchronized point and then later unblocked (when it's that thread's "turn" to run).

The *Runnable* Interface

This is all well and good if every time you want to create a `Thread` you have the luxury of being able to place it under the `Thread` class in the single-inheritance class tree. What if it more naturally belongs under some other class, from which it needs to get most of its implementation? The interfaces of Day 16 come to the rescue:

```
public class MySecondThread extends ImportantClass implements Runnable {
    public void run() {
        . . . // do something useful
    }
}
```

By implementing the interface `Runnable`, you declare your intention to run in a separate thread. In fact, the class `Thread` itself implements this interface, as you might expect from the design discussions on Day 16. As you also might guess from the example, the interface `Runnable`

specifies only one method: `run()`. As in `MyFirstThread`, you expect someone to create an instance of a thread and somehow call your `run()` method. Here's how this is accomplished:

```
MySecondThread  aMST = new MySecondThread();
Thread          aThread = new Thread(aMST);

aThread.start(); // calls our run() method, indirectly
```

First, you create an instance of `MySecondThread`. Then, by passing this instance to the constructor making the new `Thread`, you make it the target of that `Thread`. Whenever that new `Thread` starts up, its `run()` method calls the `run()` method of the target it was given (assumed by the `Thread` to be an object that implements the `Runnable` interface). When `start()` is called, `aThread` (indirectly) calls your `run()` method. You can stop `aThread` with `stop()`. If you don't need to talk to the `Thread` explicitly or to the instance of `MySecondThread`, here's a one line shortcut:

```
new Thread(new MySecondThread()).start();
```



Note: As you can see, the class name, `MySecondThread`, is a bit of a misnomer—it does not descend from `Thread`, nor is it actually the thread that you `start()` and `stop()`. It probably should have been called `MySecondThreadedClass` or `ImportantRunnableClass`.

18

ThreadTester

Here's a longer example:

```
public class SimpleRunnable implements Runnable {
    public void run() {
        System.out.println("in thread named '"
            + Thread.currentThread().getName() + "'");
    } // any other methods run() calls are in current thread as well
}

public class ThreadTester {
    public static void main(String argv[]) {
        SimpleRunnable aSR = new SimpleRunnable();

        while (true) {
            Thread t = new Thread(aSR);

            System.out.println("new Thread() " + (t == null ?
                "fail" : "succeed") + "ed.");
            t.start();
            try { t.join(); } catch (InterruptedException ignored) { }
            // waits for thread to finish its run() method
        }
    }
}
```



Note: You may be worried that only one instance of the class `SimpleRunnable` is created, but many new `Threads` are using it. Don't they get confused? Remember to separate in your mind the `asr` instance (and the methods it understands) from the various threads of execution that can pass through it. `asr`'s methods provide a template for execution, and the multiple threads created are sharing that template. Each remembers where it is executing and whatever else it needs to make it distinct from the other running threads. They all share the same instance and the same methods. That's why you need to be so careful, when adding synchronization, to imagine numerous threads running rampant over each of your methods.

The class method `currentThread()` can be called to get the thread in which a method is currently executing. If the `SimpleRunnable` class were a subclass of `Thread`, its methods would know the answer already (*it is the thread running*). Because `SimpleRunnable` simply implements the interface `Runnable`, however, and counts on someone else (`ThreadTester`'s `main()`) to create the thread, its `run()` method needs another way to get its hands on that thread. Often, you'll be deep inside methods called by your `run()` method when suddenly you need to get the current thread. The class method shown in the example works, no matter where you are.



Warning: You can do some reasonably disastrous things with your knowledge of threads. For example, if you're running in the main thread of the system and, because you think you are in a different thread, you accidentally say the following:

```
Thread.currentThread().stop();  
it has unfortunate consequences for your (soon-to-be-dead) program!
```

The example then calls `getName()` on the current thread to get the thread's name (usually something helpful, such as `Thread-23`) so it can tell the world in which thread `run()` is running. The final thing to note is the use of the method `join()`, which, when sent to a thread, means "I'm planning to wait forever for you to finish your `run()` method." You don't want to do this lightly: if you have anything else important you need to get done in your thread any time soon, you can't count on how long the `join()`ed thread may take to finish. In the example, its `run()` method is short and finishes quickly, so each loop can safely wait for the previous thread to die before creating the next one. (Of course, in this example, you didn't have anything else you wanted to do while waiting for `join()` anyway.) Here's the output produced:

```
new Thread() succeeded.
```

```
in thread named 'Thread-1'
new Thread() succeeded.
in thread named 'Thread-2'
new Thread() succeeded.
in thread named 'Thread-3'
^C
```

Ctrl+C was pressed to interrupt the program, because it otherwise would continue forever.

NamedThreadTester

If you want your threads to have particular names, you can assign them yourself by using a two-argument form of Thread's constructor:

```
public class NamedThreadTester {
    public static void main(String argv[]) {
        SimpleRunnable aSR = new SimpleRunnable();

        for (int i = 1; true; ++i) {
            Thread t = new Thread(aSR, "" + (100 - i)
                                + " threads on the wall...");

            System.out.println("new Thread() " + (t == null ?
                                "fail" : "succeed") + "ed.");
            t.start();
            try { t.join(); } catch (InterruptedException ignored) { }
        }
    }
}
```

18

which takes a target object, as before, and a String, which names the new thread. Here's the output:

```
new Thread() succeeded.
in thread named '99 threads on the wall...'
new Thread() succeeded.
in thread named '98 threads on the wall...'
new Thread() succeeded.
in thread named '97 threads on the wall...'
^C
```

Naming a thread is one easy way to pass it some information. This information flows from the parent thread to its new child. It's also useful, for debugging purposes, to give threads meaningful names (such as network input) so that when they appear during an error—in a stack trace, for example—you can easily identify which thread caused the problem. You might also think of using names to help group or organize your threads, but Java actually provides you with a ThreadGroup class to perform this function. A ThreadGroup allows you to group threads, to control them all as a unit, and to keep them from being able to affect other threads (useful for security).

Knowing When a Thread has Stopped

Let's imagine a different version of the last example, one that creates a thread and then hands the thread off to other parts of the program. Suppose it would then like to know when that thread dies so that it can perform some cleanup operation. If `SimpleRunnable` were a subclass of `Thread`, you might try to catch `stop()` whenever it's sent—but look at `Thread`'s declaration of the `stop()` method:

```
public final void stop() { . . . }
```

The `final` here means that you can't override this method in a subclass. In any case, `SimpleRunnable` is *not* a subclass of `Thread`, so how can this imagined example possibly catch the death of its thread? The answer is to use the following magic:

```
public class SingleThreadTester {
    public static void main(String argv[]) {
        Thread t = new Thread(new SimpleRunnable());

        try {
            t.start();
            someMethodThatMightStopTheThread(t);
        } catch (ThreadDeath aTD) {
            . . . // do some required cleanup
            throw aTD; // re-throw the error
        }
    }
}
```

You understand most of this magic from yesterday's lesson. All you need to know is that if the thread created in the example dies, it throws an error of class `ThreadDeath`. The code catches that error and performs the required cleanup. It then rethrows the error, allowing the thread to die. The cleanup code is not called if the thread exits normally (its `run()` method completes), but that's fine; you posited that the cleanup was needed only when `stop()` was used on the thread.

Note: Threads can die in other ways—for example, by throwing exceptions that no one catches. In these cases, `stop()` is never called, and the previous code is not sufficient. (If the cleanup always has to occur, even at the normal end of a thread's life, you can put it in a `finally` clause.) Because unexpected exceptions can come out of nowhere to kill a thread, multithreaded programs that carefully catch and handle all their exceptions are more predictable, robust, and easier to debug.

Thread Scheduling

You might wonder exactly what order your threads will be run in, and how you can control that order. Unfortunately, the current implementations of the Java system cannot precisely answer the former, though with a lot of work, you can always do the latter.

NEW The part of the system that decides the real-time ordering of threads is called the **TERM** *scheduler*.

Preemptive Versus Nonpreemptive

Normally, any scheduler has two fundamentally different ways of looking at its job: non-preemptive scheduling and preemptive time-slicing.

NEW With *non-preemptive scheduling*, the scheduler runs the current thread forever, **TERM** requiring that thread explicitly to tell it when it is safe to start a different thread. With *preemptive time-slicing*, the scheduler runs the current thread until it has used up a certain tiny fraction of a second, and then “preempts” it, `suspend()`s it, and `resume()`s another thread for the next tiny fraction of a second.

Non-preemptive scheduling is very courtly, always asking for permission to schedule, and is quite valuable in extremely time-critical, real-time applications where being interrupted at the wrong moment, or for too long, could mean crashing an airplane.

Most modern schedulers use preemptive time-slicing, because, except for a few time-critical cases, it has turned out to make writing multithreaded programs much easier. For one thing, it does not force each thread to decide exactly when it should “yield” control to another thread. Instead, every thread can just run blindly on, knowing that the scheduler will be fair about giving all the other threads their chance to run.

It turns out that this approach is still not the ideal way to schedule threads. You’ve given a little too much control to the scheduler. The final touch many modern schedulers add is to allow you to assign each thread a priority. This creates a total ordering of all threads, making some threads more “important” than others. Being higher priority often means that a thread gets run more often (or gets more total running time), but it always means that it can interrupt other, lower-priority threads, even before their “time-slice” has expired.

The current Java release does not precisely specify the behavior of its scheduler. Threads can be assigned priorities, and when a choice is made between several threads that all want to run, the highest-priority thread wins. However, among threads that are all the same priority, the behavior is not well-defined. In fact, the different platforms on which Java currently runs have different behaviors—some behaving more like a preemptive scheduler, and some more like a non-preemptive scheduler.



Note: This incomplete specification of the scheduler is terribly annoying and, presumably, will be corrected in later releases. Not knowing the fine details of how scheduling occurs is perfectly all right, but not knowing whether equal priority threads must explicitly yield or face running forever, is not all right. For example, all the threads you have created so far are equal priority threads, so you don't know their basic scheduling behavior!

Testing Your Scheduler

To find out what kind of scheduler you have on your system, try the following:

```
public class RunnablePotato implements Runnable {
    public void run() {
        while (true)
            System.out.println(Thread.currentThread().getName());
    }
}

public class PotatoThreadTester {
    public static void main(String argv[]) {
        RunnablePotato aRP = new RunnablePotato();

        new Thread(aRP, "one potato").start();
        new Thread(aRP, "two potato").start();
    }
}
```

For a non-preemptive scheduler, this prints the following:

```
one potato
one potato
one potato
...
```

forever, until you interrupt the program. For a preemptive scheduler that time-slices, it repeats the line one potato a few times, followed by the same number of two potato lines, over and over:

```
one potato
one potato
...
one potato
two potato
two potato
...
two potato
...
```

until you interrupt the program. What if you want to be sure the two threads will take turns, no matter what the system scheduler wants to do? You rewrite `RunnablePotato` as follows:

```
public class RunnablePotato implements Runnable {
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
            Thread.yield(); // let another thread run for a while
        }
    }
}
```

 **Tip:** Normally you would have to say `Thread.currentThread().yield()` to get your hands on the current thread, and then call `yield()`. Because this pattern is so common, however, the `Thread` class provides a shortcut.

The `yield()` method explicitly gives any other threads that want to run a chance to begin running. (If there are no threads waiting to run, the thread that made the `yield()` simply continues.) In our example, there's another thread that's just *dying* to run, so when you now execute the class `ThreadTester`, it should output the following:

```
one potato
two potato
one potato
two potato
one potato
two potato
...
```

even if your system scheduler is non-preemptive, and would never normally run the second thread.

PriorityThreadTester

To see whether priorities are working on your system, try this:

```
public class PriorityThreadTester {
    public static void main(String argv[]) {
        RunnablePotato aRP = new RunnablePotato();
        Thread t1 = new Thread(aRP, "one potato");
        Thread t2 = new Thread(aRP, "two potato");

        t2.setPriority(t1.getPriority() + 1);
        t1.start();
        t2.start(); // at priority Thread.NORM_PRIORITY + 1
    }
}
```



Tip: The values representing the lowest, normal, and highest priorities that threads can be assigned are stored in class variables of the Thread class: Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, and Thread.MAX_PRIORITY. The system assigns new threads, by default, the priority Thread.NORM_PRIORITY. Priorities in Java are currently defined in a range from 1 to 10, with 5 being normal, but you shouldn't depend on these values; use the class variables, or tricks like the one shown in this example.

If one potato is the first line of output, your system does not preempt using priorities.

Why? Imagine that the first thread (t1) has just begun to run. Even before it has a chance to print anything, along comes a higher-priority thread (t2) that wants to run right away. That higher-priority thread should preempt (interrupt) the first, and get a chance to print two potato before t1 finishes printing anything. In fact, if you use the RunnablePotato class that never yield(), t2 stays in control forever, printing two potato lines, because it's a higher priority than t1 and it never yields control. If you use the latest RunnablePotato class (with yield()), the output is alternating lines of one potato and two potato as before, but starting with two potato.

Here's a good, illustrative example of how complex threads behave:

```
public class ComplexThread extends Thread {
    private int delay;

    ComplexThread(String name, float seconds) {
        super(name);
        delay = (int) seconds * 1000; // delays are in milliseconds
        start(); // start up ourself!
    }

    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                return;
            }
        }
    }

    public static void main(String argv[]) {
        new ComplexThread("one potato", 1.1F);
        new ComplexThread("two potato", 1.3F);
        new ComplexThread("three potato", 0.5F);
        new ComplexThread("four", 0.7F);
    }
}
```

This example combines the thread and its tester into a single class. Its constructor takes care of naming (itself) and of starting (itself), because it is now a `Thread`. The `main()` method creates new instances of its own class, because that class is a subclass of `Thread`. `run()` is also more complicated, because it now uses, for the first time, a method that can throw an unexpected exception.

The `Thread.sleep()` method forces the current thread to `yield()` and then waits for at least the specified amount of time to elapse before allowing the thread to run again. It might be interrupted, however, while sleeping by another thread. In such a case, it throws an `InterruptedException`. Now, because `run()` is not defined as throwing this exception, you must “hide” the fact by catching and handling it yourself. Because interruptions are usually requests to stop, you should exit the thread, which you can do by simply returning from the `run()` method.

This program should output a repeating but complex pattern of four different lines, where every once in a great while you see the following:

```
...
one potato
two potato
three potato
four
...
```



You should study the pattern output to prove to yourself that true parallelism is going on inside Java programs. You may also begin to appreciate that, if even this simple set of four threads can produce such complex behavior, many more threads must be capable of producing near chaos if not carefully controlled. Luckily, Java provides the synchronization and thread-safe libraries you need to control that chaos.

Summary

Today, you learned that parallelism is desirable and powerful, but introduces many new problems—methods and variables now need to be *protected* from thread conflicts—that can lead to chaos if not carefully controlled.

By “thinking multithreaded,” you can detect the places in your programs that require synchronized statements (or modifiers) to make them thread-safe. A series of `Point` examples demonstrated the various levels of safety you can achieve, while `ThreadTesters` showed how subclasses of `Thread`, or classes that implement the `Runnable` interface, are created and `run()` to generate multithreaded programs.

You also learned how to `yield()`, how to `start()`, `stop()`, `suspend()`, and `resume()` your threads, and how to catch `ThreadDeath` whenever it happens.

Finally, you learned about preemptive and non-preemptive scheduling, both with and without priorities, and how to test your Java system to see which of them your scheduler is using.

This wraps up the description of threads. You now know enough to write the most complex of programs: multithreaded ones. As you get more comfortable with threads, you may begin to use the `ThreadGroup` class or to use the enumeration methods of `Thread` to get your hands on all the threads in the system and manipulate them. Don't be afraid to experiment; you can't permanently break anything, and you only learn by trying.

Q&A

Q If they're so important to Java, why haven't threads appeared throughout the entire book?

A Actually, they have. Every stand-alone program written so far has "created" at least one thread, the one in which it is running. (Of course the system created that `Thread` for it automatically.)

Q How exactly do these threads get created and run? What about applets?

A When a simple, stand-alone Java program starts up, the system creates a main thread, and its `run()` method calls your `main()` method to start your program—you do nothing to get that `Thread`. Likewise, when a simple applet loads into a Java-capable browser, a `Thread` has already been created by the browser, and its `run()` method calls your `init()` and `start()` methods to start your program. In either case, a new `Thread()` of some kind was done somewhere, by the Java environment itself.

Q The `ThreadTester` class has an infinite loop that creates `Threads` and then `join()`s with them. Is it really infinite?

A In theory, yes. In actuality, how far the loop runs determines the resource limits of (and tests the stability of) the threads package and garbage collector in your Java release. Over time, all Java releases will converge on making the loop truly infinite.

Q I know Java releases are still a little fuzzy about the scheduler's behavior, but what's the current story?

A Here are the gruesome details, relayed by Arthur van Hoff at Sun: the way Java schedules threads "...depends on the platform. It is usually preemptive, but not always time-sliced. Priorities are not always observed, depending on the underlying implementation." This final clause gives you a hint that all this confusion is an implementation problem, and that in some future release, the design and implementation will both be clear about scheduling behavior.

Q Does Java support more complex multithreaded concepts, such as semaphores?

A The class `Object` in Java provides methods that can be used to build up condition variables, semaphores, and any higher-level parallel construct you might need. The method `wait()` (and its two variants with a timeout) causes the current thread to wait until some condition has been satisfied. The method `notify()` (or `notifyAll()`), which must be called from within a `synchronized` method or block, tells the thread (or *all* threads) to wake up and check that condition again, because something has changed. By careful combinations of these two primitive methods, any data structure can be manipulated safely by a set of threads, and all the classical parallel primitives needed to implement published parallel algorithms can be built.

Q My parallel friends tell me I should worry about something called “deadlock.” Should I?

A Not for simple multithreaded programs. However, in more complicated programs, one of the biggest worries does become one of avoiding a situation in which one thread has locked an object and is waiting for another thread to finish, while that other thread is waiting for the first thread to release that same object before it can finish. That’s a deadlock—both threads will be stuck forever. Mutual dependencies like this involving more than two threads can be quite intricate, convoluted, and difficult to locate, much less rectify. They are one of the main challenges in writing complex multithreaded programs.



21

WEEK
3

Under the Hood

by Charles L. Perkins

On today, your final day, the inner workings of the Java system will be revealed.

You'll find out all about Java's vision, Java's virtual machine, those bytecodes you've heard so much about, that mysterious garbage collector, and why you might worry about security but don't have to.

Let's begin, however, with the big picture.

The Big Picture

The Java team is very ambitious. Their ultimate goal is nothing less than to revolutionize the way software is written and distributed. They've started with the Internet, where they believe much of the interesting software of the future will live.

To achieve such an ambitious goal, a large fraction of the Internet programming community itself must be marshalled behind a similar goal and given the tools to help achieve it. The Java language, with its four S's (small, simple, safe, secure), and its flexible, Net-oriented environment, hopes to become the focal point for the rallying of this new legion of programmers.

To this end, Sun Microsystems has done something rather gutsy. What was originally a secret, tens-of-millions-of-dollars research and development project, and 100 percent proprietary, has become a free, open, and relatively unencumbered technology standard upon which anyone can build. They are literally *giving it away* and reserving only the rights they need to maintain and grow the standard.



Note: Actually, as Sun's lawyers have had more and more time to think, the original intentions of the Java team get further obscured by legal details. It is still *relatively* unencumbered, but its earlier releases were completely unencumbered. Let's hope that this is not a pattern that will continue.

Any truly open standard must be supported by at least one excellent, freely available "demonstration" implementation. Sun has already shipped an alpha, and now a beta, of one to the Internet and plans on a final release soon. In parallel, several universities, companies, and individuals have already expressed their intention to duplicate the Java environment, based on the open API that Sun has created.

Several other languages are even contemplating compiling down to Java bytecodes, to help support them in becoming a more robust and widespread standard for moving executable content around on the Net.

Why It's a Powerful Vision

One of the reasons this brilliant move on Sun's part has a real chance of success is the pent-up frustration of literally a whole generation of programmers who desperately want to share their code with one another. Right now, the computer science world is balkanized into factions at universities and companies all over the world, with hundreds of languages, dozens of them widely used, dividing and separating us all. It's the worst sort of Tower of Babel. Java hopes to build some bridges and help tear down that tower. Because it is so simple, because it's so useful for programming over the Internet, and because the Internet is so "hot" right now—this confluence of forces should help propel Java onto centerstage.

It deserves to be there. It is the natural outgrowth of ideas that, since the early 1970s inside the Smalltalk group at Xerox PARC, have lain relatively dormant in the mainstream. Smalltalk, in fact, invented the first object-oriented bytecode interpreter and pioneered many of the deep ideas that Java builds on today. Those efforts were not embraced over the intervening decades as a solution to the general problems of software, however. Today, with those problems becoming so much more obvious, and with the Net crying out for a new kind of programming, the soil is fertile to grow something stronger from those old roots, something that just might spread like wildfire. (Is it a coincidence that Java's previous internal names were Green and OAK?)

This new vision of software is one in which the Net becomes an ocean of objects, classes, and the open APIs between them. Traditional applications have vanished, replaced by skeletal frameworks like the Eiffel tower, into which can be fitted any parts from this ocean, on demand, to suit any purpose. User interfaces will be mixed and matched, built in pieces and constructed to taste, whenever the need arises, *by their own users*. Menus of choices will be filled by dynamic lists of *all* the choices available for that function, at that exact moment, across the entire ocean (of the Net).

In such a world, software distribution is no longer an issue. Software will be *everywhere* and will be paid for via a plethora of new micro-accounting models, which charge tiny fractions of cents for the parts as they are assembled and used. Frameworks will come into existence to support entertainment, business, and the social (cyber-)spaces of the near future.

This is a dream that many of us have waited *all our lives* to be a part of. There are tremendous challenges to making it all come true, but the powerful winds of change we all feel must stir us into action, because, at last, there is a base on which to build that dream—Java.

21

The Java Virtual Machine

To make visions like this possible, Java must be ubiquitous. It must be able to run on any computer and any operating system—now, and in the future. In order to achieve this level of portability, Java must be very precise not only about the language itself, but about the

environment in which the language lives. You can see, from earlier in the book and Appendix B, that the Java environment includes a generally useful set of packages of classes and a freely available implementation of them. This takes care of a part of what is needed, but it is crucial also to specify exactly how the run-time environment of Java behaves.

This final requirement is what has stymied many attempts at ubiquity in the past. If you base your system on any assumptions about what is “beneath” the run-time system, you lose. If you depend in any way on the computer or operating system below, you lose. Java solves this problem by *inventing* an abstract computer of its own and running on that.

This “virtual” machine runs a special set of “instructions” called bytecodes that are simply a stream of formatted bytes, each of which has a precise specification of exactly what each bytecode does to this virtual machine. The virtual machine is also responsible for certain fundamental capabilities of Java, such as object creation and garbage collection.

Finally, in order to be able to move bytecodes safely across the Internet, you need a bulletproof model of security—and how to maintain it—and a precise format for how this stream of bytecodes can be sent from one virtual machine to another.

Each of these requirements is addressed in today’s lesson.



Note: This discussion blurs the distinction between the run-time and the virtual machine of Java. This is intentional but a little unconventional. Think of the virtual machine as providing *all* the capabilities, even those that are conventionally assigned to the run-time. This book uses the words “run-time” and “virtual machine” interchangeably. Equating the two highlights the single *environment* that must be created to support Java.

Much of the following description is based closely on the latest “Virtual Machine Specifications” documents (and the 1.0 bytecodes), so if you delve more deeply into the details online, you should cover some familiar ground.

An Overview

It is worth quoting the introduction to the Java virtual machine documentation here, because it is so relevant to the vision outlined earlier:

The Java virtual machine specification has a purpose that is both like and unlike equivalent documents for other languages and abstract machines. It is intended to present an abstract, logical machine design free from the distraction of inconsequential

details of any implementation. It does not anticipate an implementation technology, or an implementation host. At the same time it gives a reader sufficient information to allow implementation of the abstract design in a range of technologies.

However, the intent of the [...] Java project is to create a language [...] that will allow the interchange over the Internet of “executable content,” which will be embodied by compiled Java code. The project specifically does not want Java to be a proprietary language and does not want to be the sole purveyor of Java language implementations. Rather, we hope to make documents like this one, and source code for our implementation, freely available for people to use as they choose.

This vision [...] can be achieved only if the executable content can be reliably shared between different Java implementations. These intentions prohibit the definition of the Java virtual machine from being fully abstract. Rather, relevant logical elements of the design have to be made sufficiently concrete to allow the interchange of compiled Java code. This does not collapse the Java virtual machine specification to a description of a Java implementation; elements of the design that do not play a part in the interchange of executable content remain abstract. But it does force us to specify, in addition to the abstract machine design, a concrete interchange format for compiled Java code.

The Java virtual machine specification consists of the following:

- ☐ The bytecode syntax, including opcode and operand sizes, values, and types, and their alignment and endian-ness
- ☐ The values of any identifiers (for example, type identifiers) in bytecodes or in supporting structures
- ☐ The layout of the supporting structures that appear in compiled Java code (for example, the constant pool)
- ☐ The Java .class file format

Each of these is covered today.

Despite this degree of specificity, there are still several elements of the design that remain (purposely) abstract, including the following:

- ☐ The layout and management of the run-time data areas
- ☐ The particular garbage-collection algorithms, strategies, and constraints used
- ☐ The compiler, development environment, and run-time extensions (apart from the need to generate and read valid Java bytecodes)
- ☐ Any optimizations performed, once valid bytecodes are received

These places are where the creativity of a virtual machine implementor has full rein.



The Fundamental Parts

The Java virtual machine can be divided into five fundamental pieces:

- ☐ A bytecode instruction set
- ☐ A set of registers
- ☐ A stack
- ☐ A garbage-collected heap
- ☐ An area for storing methods

Some of these might be implemented by using an interpreter, a native binary code compiler, or even a hardware chip—but all these logical, abstract components of the virtual machine must be supplied in *some* form in every Java system.



Note: The memory areas used by the Java virtual machine are not required to be at any particular place in memory, to be in any particular order, or even to use contiguous memory. However, all but the method area must be able to represent align 32-bit values (for example, the Java stack is 32 bits wide).

The virtual machine, and its supporting code, is often referred to as the run-time environment, and when this book refers to something being done at run-time, the virtual machine is what's doing it.

Java Bytecodes

The Java virtual machine instruction set is optimized to be small and compact. It is designed to travel across the Net, and so has traded off speed-of-interpretation for space. (Given that both Net bandwidth and mass storage speeds increase less rapidly than CPU speed, this seems like an appropriate trade-off.)

As mentioned, Java source code is “compiled” into bytecodes and stored in a .class file. On Sun's Java system, this is performed using the `javac` tool. It is not exactly a traditional “compiler,” because `javac` translates source code into bytecodes, a lower-level format that cannot be run directly, but must be further interpreted by each computer. Of course, it is exactly this level of “indirection” that buys you the power, flexibility, and extreme portability of Java code.

Note: Quotation marks are used around the word “compiler” when talking about `javac` because later today you will also learn about the “just-in-time” compiler, which acts more like the back end of a traditional compiler. The use of the same word “compiler” for these two different pieces of Java technology is unfortunate, but somewhat reasonable, because each is really one-half (either the front or the back end) of a more traditional compiler.

A bytecode instruction consists of a one-byte opcode that serves to identify the instruction involved and zero or more operands, each of which may be more than one byte long, that encode the parameters the opcode requires.

Note: When operands are more than one byte long, they are stored in big-endian order, high-order byte first. These operands must be assembled from the byte stream at run-time. For example, a 16-bit parameter appears in the stream as two bytes so that its value is `first_byte * 256 + second_byte`. The bytecode instruction stream is only byte-aligned, and alignment of any larger quantities is not guaranteed (except for “within” the special bytecodes `lookupswitch` and `tableswitch`, which have special alignment rules of their own).

Bytecodes interpret data in the run-time memory areas as belonging to a fixed set of types: the primitive types you’ve seen several times before, consisting of several signed integer types (8-bit `byte`, 16-bit `short`, 32-bit `int`, 64-bit `long`), one unsigned integer type (16-bit `char`), and two signed floating-point types (32-bit `float`, 64-bit `double`), plus the type “reference to an object” (a 32-bit pointer-like type). Some special bytecodes (for example, the `dup` instructions), treat run-time memory areas as raw data, without regard to type. This is the exception, however, not the rule.

These primitive types are distinguished and managed by the compiler, `javac`, not by the Java run-time environment. These types are not “tagged” in memory, and thus cannot be distinguished at run-time. Different bytecodes are designed to handle each of the various primitive types uniquely, and the compiler carefully chooses from this palette based on its knowledge of the actual types stored in the various memory areas. For example, when adding two integers, the compiler generates an `iadd` bytecode; for adding two floats, `fadd` is generated. (You’ll see all this in gruesome detail later.)

Registers

The registers of the Java virtual machine are just like the registers inside a “real” computer.

NEW TERM *Registers* hold the machine’s state, affect its operation, and are updated after each bytecode is executed.

The following are the Java registers:

- ☐ *pc*, the program counter, which indicates what bytecode is being executed
- ☐ *optop*, a pointer to the top of the operand stack, which is used to evaluate all arithmetic expressions
- ☐ *frame*, a pointer to the execution environment of the current method, which includes an activation record for this method call and any associated debugging information
- ☐ *vars*, a pointer to the first local variable of the currently executing method

The virtual machine defines these registers to be 32 bits wide.

Note: Because the virtual machine is primarily stack-based, it does not use any registers for passing or receiving arguments. This is a conscious choice skewed toward bytecode simplicity and compactness. It also aids efficient implementation on register-poor architectures, which most of today’s computers, unfortunately, are. Perhaps when the majority of CPUs out there are a little more sophisticated, this choice will be reexamined, though simplicity and compactness may still be reason enough!

By the way, the *pc* register is also used when the run-time handles exceptions; catch clauses are (ultimately) associated with ranges of the *pc* within a method’s bytecodes.

The Stack

The Java virtual machine is stack-based. A Java stack frame is similar to the stack frame of a conventional programming language—it holds the state for a single method call. Frames for nested method calls are stacked on top of this frame.

NEW TERM The *stack* is used to supply parameters to bytecodes and methods, and to receive results back from them.

Each stack frame contains three (possibly empty) sets of data: the local variables for the method call, its execution environment, and its operand stack. The sizes of these first two are fixed at the start of a method call, but the operand stack varies in size as bytecodes are executed in the method.

Local variables are stored in an array of 32-bit slots, indexed by the register `vars`. Most types take up one slot in the array, but the `long` and `double` types each take up two slots.

Note: `long` and `double` values, stored or referenced via an index `N`, take up the (32-bit) slots `N` and `N + 1`. These 64-bit values are thus not guaranteed to be 64-bit-aligned. Implementors are free to decide the appropriate way to divide these values among the two slots.

The execution environment in a stack frame helps to maintain the stack itself. It contains a pointer to the previous stack frame, a pointer to the local variables of the method call, and pointers to the stack's current "base" and "top." Additional debugging information can also be placed into the execution environment.

The operand stack, a 32-bit first-in-first-out (FIFO) stack, is used to store the parameters and return values of most bytecode instructions. For example, the `iadd` bytecode expects two integers to be stored on the top of the stack. It pops them, adds them together, and pushes the resulting sum back onto the stack.

Each primitive data type has unique instructions that know how to extract, operate, and push back operands of that type. For example, `long` and `double` operands take two "slots" on the stack, and the special bytecodes that handle these operands take this into account. It is illegal for the types on the stack and the instruction operating on them to be incompatible (javac outputs bytecodes that always obey this rule).

Note: The top of the operand stack and the top of the overall Java stack are almost always the same. Thus, "the stack," refers to both stacks, collectively.

21

The Heap

The heap is that part of memory from which newly created instances (objects) are allocated.

The heap is often assigned a large, fixed size when the Java run-time system is started, but on systems that support virtual memory, it can grow as needed, in a nearly unbounded fashion.

Because objects are automatically garbage-collected in Java, programmers do not have to

(and, in fact, *cannot*) manually free the memory allocated to an object when they are finished using it.

Java objects are referenced indirectly in the run-time, via handles, which are a kind of pointer into the heap.

Because objects are never referenced directly, parallel garbage collectors can be written that operate independently of your program, moving around objects in the heap at will. You'll learn more about garbage collection later.

The Method Area

Like the compiled code areas of conventional programming language environments, or the TEXT segment in a UNIX process, the method area stores the Java bytecodes that implement almost every method in the Java system. (Remember that some methods might be native, and thus implemented, for example, in C.) The method area also stores the symbol tables needed for dynamic linking, and any other additional information debuggers or development environments which might want to associate with each method's implementation.

Because bytecodes are stored as byte streams, the method area is aligned on byte boundaries. (The other areas are all aligned on 32-bit word boundaries.)

The Constant Pool

In the heap, each class has a constant pool "attached" to it. Usually created by `javac`, these constants encode all the names (of variables, methods, and so forth) used by any method in a class. The class contains a count of how many constants there are and an offset that specifies how far into the class description itself the array of constants begins. These constants are typed via specially coded bytes and have a precisely defined format when they appear in the `.class` file for a class. Later today, a little of this file format is covered, but everything is fully specified by the virtual machine specifications in your Java release.

Limitations

The virtual machine, as currently defined, places some restrictions on legal Java programs by virtue of the choices it has made (some were previously described, and more will be detailed later today).

These limitations and their implications are

- ☐ 32-bit pointers, which imply that the virtual machine can address only 4G of memory (this may be relaxed in later releases)

- ☐ Unsigned 16-bit indices into the exception, line number, and local variable tables, which limit the size of a method's bytecode implementation to 64K (this limitation may be eliminated in the final release)
- ☐ Unsigned 16-bit indices into the constant pool, which limits the number of constants in a class to 64K (a limit on the complexity of a class)

In addition, Sun's implementation of the virtual machine uses so-called `_quick` bytecodes, which further limit the system. Unsigned 8-bit offsets into objects may limit the number of methods in a class to 256 (this limit may not exist in the final release), and unsigned 8-bit argument counts limit the size of the argument list to 255 32-bit words. (Although this means that you can have up to 255 arguments of most types, you can have only 127 of them if they're all `long` or `double`.)

Bytecodes in More Detail

One of the main tasks of the virtual machine is the fast, efficient execution of the Java bytecodes in methods. Unlike in the discussion yesterday about generality versus efficiency, this is a case where speed is of the upmost importance. Every Java program suffers from a slow implementation here, so the run-time must use as many "tricks" as possible to make bytecodes run fast. The only other goal (or limitation) is that Java programmers must not be able to see these tricks in the behavior of their programs.

A Java run-time implementer must be extremely clever to satisfy both these goals.

The Bytecode Interpreter

A bytecode interpreter examines each opcode byte (bytecode) in a method's bytecode stream, in turn, and executes a unique action for that bytecode. This might consume further bytes for the operands of the bytecode and might affect which bytecode will be examined next. It operates like the hardware CPU in a computer, which examines memory for instructions to carry out in exactly the same manner. It is the software CPU of the Java virtual machine.

Your first, naive attempt to write such a bytecode interpreter will almost certainly be disastrously slow. The inner loop, which dispatches one bytecode each time through the loop, is notoriously difficult to optimize. In fact, smart people have been thinking about this problem, in one form or another, for more than 20 years. Luckily, they've gotten results, all of which can be applied to Java.

The final result is that the interpreter shipped in the current release of Java has an extremely fast inner loop. In fact, on even a relatively slow computer, this interpreter can perform more than 590,000 bytecodes per second! This is really quite good, because the CPU in that computer does only about 30 times better using *hardware*.

This interpreter is fast enough for most Java programs (and for those *requiring* more speed, they can always use native methods—see yesterday’s discussion)—but what if a smart implementor wants to do better?

The “Just-in-Time” Compiler

About a decade ago, a really clever trick was discovered by Peter Deutsch while trying to make Smalltalk run faster. He called it “dynamic translation” during interpretation. Sun calls it “just-in-time” compiling.

The trick is to notice that the really fast interpreter you’ve just written—in C, for example—already has a useful sequence of native binary code for each bytecode that it interprets: *the binary code that the interpreter itself is executing*. Because the interpreter has already been compiled from C into native binary code, for each bytecode that it interprets, it passes through a sequence of native code instructions for the hardware CPU on which it is running. By saving a copy of each binary instruction as it “goes by,” the interpreter can keep a running log of the binary code it *itself* has run to interpret a bytecode. It can just as easily keep a log of the set of bytecodes that it ran to interpret an entire method.

You take that log of instructions and “peephole-optimize” it, just as a smart compiler does. This eliminates redundant or unnecessary instructions from the log, and makes it look just like the optimized binary code that a good compiler might have produced.



Note: This is where the name compiler comes from, in “just-in-time” compiler, but it’s really only the back end of a traditional compiler—the part that does code generation. By the way, the front end here is `javac`.

Here’s where the trick comes in. The next time that method is run (in exactly the same way), the interpreter can now simply execute directly the stored log of binary native code. Because this optimizes out the inner-loop overhead of each bytecode, as well as any other redundancies between the bytecodes in a method, it can gain a factor of 10 or more in speed. In fact, an experimental version of this technology at Sun has shown that Java programs using it can run as fast as compiled C programs.

Note: The parenthetical in the last paragraph is needed because if anything is different about the input to the method, it takes a different path through the interpreter and must be relogged. (There are sophisticated versions of this technology that solve this, and other, difficulties.) The cache of native code for a method must be invalidated whenever the method has changed, and the interpreter must pay a small cost up front each time a method is run for the first time. However, these small bookkeeping costs are far outweighed by the amazing gains in speed possible.

The *java2c* Translator

Another, simpler, trick, which works well whenever you have a good, portable C compiler on each system that runs your program, is to translate the bytecodes into C and then compile the C into binary native code. If you wait until the first use of a method or class, and then perform this as an “invisible” optimization, it gains you an additional speedup over the approach outlined previously, without the Java programmer needing to know about it.

Of course, this does limit you to systems with a C compiler, but as you learned yesterday, there are extremely good, freely available C compilers. In theory, your Java code might be able to travel with its own C compiler, or know where to pull one from the Net as needed, for each new computer and operating system it faced. (Because this violates some of the rules of normal Java code movement over the Net, though, it should be used sparingly.)

If you’re using Java, for example, to write a server that lives only on *your* computer, it might be appropriate to use Java for its flexibility in writing and maintaining the server (and for its capability of dynamically linking new Java code on the fly), and then to run *java2c* by hand to translate the basic server itself entirely into native code. You’d link the Java run-time environment into that code so that your server remains a fully capable Java program, but it’s now an extremely fast one.

In fact, an experimental version of the *java2c* translator inside Sun shows that it can reach the speed of compiled and optimized C code. This is the best that you can hope to do!

21

Note: Unfortunately, as of the 1.0 release, there is still no publicly available *java2c* tool, and Sun’s virtual machine does not perform “just-in-time” compilation. Both of these have been promised in a later release (perhaps 1.1).

The Bytecodes Themselves

Let's look at a (progressively less and less) detailed description of each class of bytecodes.

For each bytecode, some brief text describes its function, and a textual "picture" of the stack, both before and after the bytecode has been executed, is shown. This text picture will look like the following:

```
..., value1, value2 => ..., value3
```

This means that the bytecode expects two operands—`value1` and `value2`—to be on the top of the stack, pops them both off the stack, operates on them to produce `value3`, and pushes `value3` back onto the top of the stack. You should read each stack from right to left, with the rightmost value being the top of the stack. The ... is read as "the rest of the stack below," which is irrelevant to the current bytecode. All operands on the stack are 32-bits wide.

Because most bytecodes take their arguments from the stack and place their results back there, the brief text descriptions that follow only say something about the source or destination of values if they are *not* on the stack. For example, the description "Load integer from local variable." means that the integer is loaded onto the stack, and "Integer add." intends its integers to be taken from—and the result returned to—the stack.

Bytecodes that don't affect control flow simply move the pc onto the next bytecode that follows in sequence. Those that do affect the pc say so explicitly. Whenever you see `byte1`, `byte2`, and so forth, it refers to the first byte, second byte, and so on, that follow the opcode byte itself. After such a bytecode is executed, the pc automatically advances over these operand bytes to start the next bytecode in sequence.



Note: The next few sections are in "reference manual style," presenting each bytecode separately in all its (often redundant) detail. Later sections begin to collapse and coalesce this verbose style into something shorter, and more readable. The verbose form is shown at first because the online reference manuals will look more like it, and because it drives home the point that each bytecode "function" comes in many, nearly identical bytecodes, one for each primitive type in Java.

Pushing Constants onto the Stack

```
bipush      ... => ..., value
```

Push one-byte signed integer. `byte1` is interpreted as a signed 8-bit value. This `value` is expanded to an `int` and pushed onto the operand stack.

`sipush` ... => ..., value

Push two-byte signed integer. `byte1` and `byte2` are assembled into a signed 16-bit value. This value is expanded to an `int` and pushed onto the operand stack.

`ldc1` ... => ..., item

Push `item` from constant pool. `byte1` is used as an unsigned 8-bit index into the constant pool of the current class. The `item` at that index is resolved and pushed onto the stack.

`ldc2` ... => ..., item

Push `item` from constant pool. `byte1` and `byte2` are used to construct an unsigned 16-bit index into the constant pool of the current class. The `item` at that index is resolved and pushed onto the stack.

`ldc2w` ... => ..., constant-word1, constant-word2

Push long or double from constant pool. `byte1` and `byte2` are used to construct an unsigned 16-bit index into the constant pool of the current class. The two-word constant at that index is resolved and pushed onto the stack.

`aconst_null` ... => ..., null

Push the `null` object reference onto the stack.

`iconst_m1` ... => ..., -1

Push the `int -1` onto the stack.

`iconst_<I>` ... => ..., <I>

Push the `int <I>` onto the stack. There are six of these bytecodes, one for each of the integers 0-5: `iconst_0`, `iconst_1`, `iconst_2`, `iconst_3`, `iconst_4`, and `iconst_5`.

`lconst_<L>` ... => ..., <L>-word1, <L>-word2

Push the long <L> onto the stack. There are two of these bytecodes, one for each of the integers 0 and 1: `lconst_0`, and `lconst_1`.

`fconst_<F>` ... => ..., <F>

Push the float <F> onto the stack. There are three of these bytecodes, one for each of the integers 0-2: `fconst_0`, `fconst_1`, and `fconst_2`.

`dconst_<D>` ... => ..., <D>-word1, <D>-word2

Push the double <D> onto the stack. There are two of these bytecodes, one for each of the integers 0 and 1: `dconst_0`, and `dconst_1`.

Loading Local Variables onto the Stack

`iload` ... => ..., value

Load `int` from local variable. Local variable `byte1` in the current Java frame must contain an `int`. The value of that variable is pushed onto the operand stack.

`iload_<I>` ... => ..., value

Load `int` from local variable. Local variable `<I>` in the current Java frame must contain an `int`. The value of that variable is pushed onto the operand stack. There are four of these bytecodes, one for each of the integers 0-3: `iload_0`, `iload_1`, `iload_2`, and `iload_3`.

`lload` ... => ..., value-word1, value-word2

Load `long` from local variable. Local variables `byte1` and `byte1 + 1` in the current Java frame must together contain a `long integer`. The values contained in those variables are pushed onto the operand stack.

`lload_<L>` ... => ..., value-word1, value-word2

Load `long` from local variable. Local variables `<L>` and `<L> + 1` in the current Java frame must together contain a `long integer`. The value contained in those variables is pushed onto the operand stack. There are four of these bytecodes, one for each of the integers 0-3: `lload_0`, `lload_1`, `lload_2`, and `lload_3`.

`fload` ... => ..., value

Load `float` from local variable. Local variable `byte1` in the current Java frame must contain a single precision floating-point number. The value of that variable is pushed onto the operand stack.

`fload_<F>` ... => ..., value

Load `float` from local variable. Local variable `<F>` in the current Java frame must contain a single precision floating-point number. The value of that variable is pushed onto the operand stack. There are four of these bytecodes, one for each of the integers 0-3: `fload_0`, `fload_1`, `fload_2`, and `fload_3`.

`dload` ... => ..., value-word1, value-word2

Load `double` from local variable. Local variables `byte1` and `byte1 + 1` in the current Java frame must together contain a double precision floating-point number. The value contained in those variables is pushed onto the operand stack.

`dload_<D> ... => ..., value-word1, value-word2`

Load double from local variable. Local variables `<D>` and `<D> + 1` in the current Java frame must together contain a double precision floating-point number. The value contained in those variables is pushed onto the operand stack. There are four of these bytecodes, one for each of the integers 0-3: `dload_0`, `dload_1`, `dload_2`, and `dload_3`.

`aload ... => ..., value`

Load object reference from local variable. Local variable `byte1` in the current Java frame must contain a return address or reference to an object or array. The value of that variable is pushed onto the operand stack.

`aload_<A> ... => ..., value`

Load object reference from local variable. Local variable `<A>` in the current Java frame must contain a return address or reference to an object. The value of that variable is pushed onto the operand stack. There are four of these bytecodes, one for each of the integers 0-3: `aload_0`, `aload_1`, `aload_2`, and `aload_3`.

Storing Stack Values into Local Variables

`istore ..., value => ...`

Store int into local variable. `value` must be an int. Local variable `byte1` in the current Java frame is set to `value`.

`istore_<I> ..., value => ...`

Store int into local variable. `value` must be an int. Local variable `<I>` in the current Java frame is set to `value`. There are four of these bytecodes, one for each of the integers 0-3: `istore_0`, `istore_1`, `istore_2`, and `istore_3`.

`lstore ..., value-word1, value-word2 => ...`

Store long into local variable. `value` must be a long integer. Local variables `byte1` and `byte1 + 1` in the current Java frame are set to `value`.

`lstore_<L> ..., value-word1, value-word2 => ...`

Store long into local variable. `value` must be a long integer. Local variables `<L>` and `<L> + 1` in the current Java frame are set to `value`. There are four of these bytecodes, one for each of the integers 0-3: `lstore_0`, `lstore_1`, `lstore_2`, and `lstore_3`.

`fstore ..., value => ...`

Store float into local variable. `value` must be a single precision floating-point number. Local variable `byte1` in the current Java frame is set to `value`.

`fstore_<F>` ..., value => ...

Store `float` into local variable. `value` must be a single precision floating-point number. Local variable `<F>` in the current Java frame is set to `value`. There are four of these bytecodes, one for each of the integers 0-3: `fstore_0`, `fstore_1`, `fstore_2`, and `fstore_3`.

`dstore` ..., value-word1, 'value-word2 => ...

Store `double` into local variable. `value` must be a double precision floating-point number. Local variables `byte1` and `byte1 + 1` in the current Java frame are set to `value`.

`dstore_<D>` ..., value-word1, value-word2 => ...

Store `double` into local variable. `value` must be a double precision floating-point number. Local variables `<D>` and `<D> + 1` in the current Java frame are set to `value`. There are four of these bytecodes, one for each of the integers 0-3: `dstore_0`, `dstore_1`, `dstore_2`, and `dstore_3`.

`astore` ..., handle => ...

Store object reference into local variable. `handle` must be a return address or a reference to an object. Local variable `byte1` in the current Java frame is set to `value`.

`astore_<A>` ..., handle => ...

Store object reference into local variable. `handle` must be a return address or a reference to an object. Local variable `<A>` in the current Java frame is set to `value`. There are four of these bytecodes, one for each of the integers 0-3: `astore_0`, `astore_1`, `astore_2`, and `astore_3`.

`iinc` -no change-

Increment local variable by constant. Local variable `byte1` in the current Java frame must contain an `int`. Its value is incremented by the value `byte2`, where `byte2` is treated as a signed 8-bit quantity.

Managing Arrays

`newarray` ..., size => result

Allocate new array. `size` must be an `int`. It represents the number of elements in the new array. `byte1` is an internal code that indicates the type of array to allocate. Possible values for `byte1` are as follows: `T_BOOLEAN` (4), `T_CHAR` (5), `T_FLOAT` (6), `T_DOUBLE` (7), `T_BYTE` (8), `T_SHORT` (9), `T_INT` (10), and `T_LONG` (11).

An attempt is made to allocate a new array of the indicated type, capable of holding `size` elements. This will be the `result`. If `size` is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown. All elements of the array are initialized to their default values.

```
anewarray ..., size => result
```

Allocate new array of objects. `size` must be an `int`. It represents the number of elements in the new array. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry must be a class.

An attempt is made to allocate a new array of the indicated class type, capable of holding `size` elements. This will be the `result`. If `size` is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown. All elements of the array are initialized to `null`.

Note: `anewarray` is used to create a single dimension of an array of objects. For example, the request `new Thread[7]` generates the following bytecodes:

```
bipush 7
anewarray <Class "java.lang.Thread">
```

`anewarray` can also be used to create the outermost dimension of a multidimensional array. For example, the array declaration `new int[6][]` generates this:

```
bipush 6
anewarray <Class "[I">
```

(See the section "Method Signatures" for more information on strings such as `[I`.)

```
multianewarray ..., size1 size2...sizeN => result
```

Allocate new multidimensional array. Each `size<I>` must be an `int`. Each represents the number of elements in a dimension of the array. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry must be an array class of one or more dimensions.

`byte3` is a positive integer representing the number of dimensions being created. It must be less than or equal to the number of dimensions of the array class. `byte3` is also the number of elements that are popped off the stack. All must be `ints` greater than or equal to zero. These are used as the sizes of the dimensions. An attempt is made to allocate a new array of the indicated class type, capable of holding `size<1> * size<2> * ... * <sizeN>` elements. This will be the `result`. If any of the `size<I>` arguments on the stack is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

Note: `new int[6][3][]` generates these bytecodes:

```
bipush 6
bipush 3
multianewarray <Class "[[I"> 2
```

It's more efficient to use `newarray` or `anewarray` when creating arrays of single dimension.

```
arraylength    ..., array => ..., length
```

Get length of array. `array` must be a reference to an array object. The length of the array is determined and replaces `array` on the top of the stack. If `array` is null, a `NullPointerException` is thrown.

```
iaload        ..., array, index => ..., value
laload        ..., array, index => ..., value-word1, value-word2
faload        ..., array, index => ..., value
daload        ..., array, index => ..., value-word1, value-word2
aaload        ..., array, index => ..., value
baload        ..., array, index => ..., value
caload        ..., array, index => ..., value
saload        ..., array, index => ..., value
```

Load `<type>` from array. `array` must be an array of `<type>`s. `index` must be an `int`. The `<type>` value at position number `index` in `array` is retrieved and pushed onto the top of the stack. If `array` is null, a `NullPointerException` is thrown. If `index` is not within the bounds of array, an `ArrayIndexOutOfBoundsException` is thrown. `<type>` is, in turn, `int`, `long`, `float`, `double`, `object` reference, `byte`, `char`, and `short`. `<type>`s `long` and `double` have two word values, as you've seen in previous load bytecodes.

```
iastore       ..., array, index, value => ...
lastore       ..., array, index, value-word1, value-word2 => ...
fastore       ..., array, index, value => ...
dastore       ..., array, index, value-word1, value-word2 => ...
aastore       ..., array, index, value => ...
bastore       ..., array, index, value => ...
castore       ..., array, index, value => ...
sastore       ..., array, index, value => ...
```

Store into `<type>` array. `array` must be an array of `<type>`s, `index` must be an `int`, and `value` a `<type>`. The `<type>` value is stored at position `index` in `array`. If `array` is null, a `NullPointerException` is thrown. If `index` is not within the bounds of array, an `ArrayIndexOutOfBoundsException` is thrown. `<type>` is, in turn, `int`, `long`, `float`, `double`, `object` reference, `byte`, `char`, and `short`. `<type>`s `long` and `double` have two word values, as you've seen in previous store bytecodes.

Stack Operations

nop -no change-

Do nothing.

pop ..., any => ...

Pop the top word from the stack.

pop2 ..., any2, any1 => ...

Pop the top two words from the stack.

dup ..., any => ..., any, any

Duplicate the top word on the stack.

dup2 ..., any2, any1 => ..., any2, any1, any2, any1

Duplicate the top two words on the stack.

dup_x1 ..., any2, any1 => ..., any1, any2, any1

Duplicate the top word on the stack and insert the copy two words down in the stack.

dup2_x1 ..., any3, any2, any1 => ..., any2, any1, any3, any2, any1

Duplicate the top two words on the stack and insert the copies two words down in the stack.

dup_x2 ..., any3, any2, any1 => ..., any1, any3, any2, any1

Duplicate the top word on the stack and insert the copy three words down in the stack.

dup2_x2 ..., any4, any3, any2, any1 => ..., any2, any1, any4, any3, any2, any1

Duplicate the top two words on the stack and insert the copies three words down in the stack.

swap ..., any2, any1 => ..., any1, any2

Swap the top two elements on the stack.

Arithmetic Operations

iadd ..., v1, v2 => ..., result

ladd ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2

fadd ..., v1, v2 => ..., result

dadd ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2

v1 and v2 must be <type>s. The vs are added and are replaced on the stack by their <type> sum.

<type> is, in turn, int, long, float, and double.

```
isub    ..., v1, v2 => ..., result
lsub    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
fsub    ..., v1, v2 => ..., result
dsb     ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
```

v1 and v2 must be <type>s. v2 is subtracted from v1, and both vs are replaced on the stack by their <type> difference. <type> is, in turn, int, long, float, and double.

```
imul    ..., v1, v2 => ..., result
lmul    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
fmul    ..., v1, v2 => ..., result
dmul    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
```

v1 and v2 must be <type>s. Both vs are replaced on the stack by their <type> product. <type> is, in turn, int, long, float, and double.

```
idiv    ..., v1, v2 => ..., result
ldiv    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
fddiv   ..., v1, v2 => ..., result
ddiv    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
```

v1 and v2 must be <type>s. v2 is divided by v1, and both vs are replaced on the stack by their <type> quotient. An attempt to divide by zero results in an `ArithmeticException` being thrown. <type> is, in turn, int, long, float, and double.

```
irem    ..., v1, v2 => ..., result
lrem    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
frem    ..., v1, v2 => ..., result
drem    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
```

v1 and v2 must be <type>s. v2 is divided by v1, and both vs are replaced on the stack by their <type> remainder. An attempt to divide by zero results in an `ArithmeticException` being thrown. <type> is, in turn, int, long, float, and double.

```
ineg    ..., value => ..., result
lneg    ..., value-word1, value-word2 => ..., result-word1, result-word2
fneg    ..., value => ..., result
dneg    ..., value-word1, value-word2 => ..., result-word1, result-word2
```

value must be a <type>. It is replaced on the stack by its arithmetic negation. <type> is, in turn, int, long, float, and double.



Note: Now that you're familiar with the look of the bytecodes, the summaries that follow will become shorter and shorter (for space reasons). You can always get any desired level of detail from the full virtual machine specification in the latest Java release.

Logical Operations

```
ishl    ..., v1, v2 => ..., result
lshl    ..., v1-word1, v1-word2, v2 => ..., r-word1, r-word2
ishr    ..., v1, v2 => ..., result
lshr    ..., v1-word1, v1-word2, v2 => ..., r-word1, r-word2
iushr   ..., v1, v2 => ..., result
lushr   ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
```

For types `int` and `long`: arithmetic shift-left, shift-right, and logical shift-right.

```
iand    ..., v1, v2 => ..., result
land    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
ior     ..., v1, v2 => ..., result
lor     ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
ixor    ..., v1, v2 => ..., result
lxor    ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., r-word1, r-word2
```

For types `int` and `long`: bitwise AND, OR, and XOR.

Conversion Operations

```
i2l     ..., value => ..., result-word1, result-word2
i2f     ..., value => ..., result
i2d     ..., value => ..., result-word1, result-word2
l2i     ..., value-word1, value-word2 => ..., result
l2f     ..., value-word1, value-word2 => ..., result
l2d     ..., value-word1, value-word2 => ..., result-word1, result-word2
f2i     ..., value => ..., result
f2l     ..., value => ..., result-word1, result-word2
f2d     ..., value => ..., result-word1, result-word2
d2i     ..., value-word1, value-word2 => ..., result
d2l     ..., value-word1, value-word2 => ..., result-word1, result-word2
d2f     ..., value-word1, value-word2 => ..., result

int2byte  ..., value => ..., result
int2char  ..., value => ..., result
int2short ..., value => ..., result
```

These bytecodes convert from a value of type `<lhs>` to a result of type `<rhs>`. `<lhs>` and `<rhs>` can be any of `i`, `l`, `f`, and `d`, which represent `int`, `long`, `float`, and `double`, respectively. The final three bytecodes convert types that are self-explanatory.

Transfer of Control

```
ifeq    ..., value => ...
ifne    ..., value => ...
iflt    ..., value => ...
ifgt    ..., value => ...
ifle    ..., value => ...
ifge    ..., value => ...

if_icmpeq ..., value1, value2 => ...
if_icmpne ..., value1, value2 => ...
if_icmplt ..., value1, value2 => ...
```

```
if_icmpgt    ..., value1, value2 => ...
if_icmple   ..., value1, value2 => ...
if_icmpge   ..., value1, value2 => ...

ifnull      ..., value => ...
ifnonnull   ..., value => ...
```

When value `<rel> 0` is true in the first set of bytecodes, `value1 <rel> value2` is true in the second set, or `value` is null (or not null) in the third, `byte1` and `byte2` are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise, execution proceeds at the bytecode following. `<rel>` is one of `eq`, `ne`, `lt`, `gt`, `le`, and `ge`, which represent equal, not equal, less than, greater than, less than or equal, and greater than or equal, respectively.

```
lcmp        ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., result

fcmpl       ..., v1, v2 => ..., result
dcmpl       ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., result

fcmpg       ..., v1, v2 => ..., result
dcmpg       ..., v1-word1, v1-word2, v2-word1, v2-word2 => ..., result
```

`v1` and `v2` must be long, float, or double. They are both popped from the stack and compared. If `v1` is greater than `v2`, the int value 1 is pushed onto the stack. If `v1` is equal to `v2`, 0 is pushed onto the stack. If `v1` is less than `v2`, -1 is pushed onto the stack. For floating-point, if either `v1` or `v2` is NaN, -1 is pushed onto the stack for the first pair of bytecodes, +1 for the second pair.

```
if_acmpeq   ..., value1, value2 => ...
if_acmpne   ..., value1, value2 => ...
```

Branch if object references are equal/not equal. `value1` and `value2` must be references to objects. They are both popped from the stack. If `value1` is equal/not equal to `value2`, `byte1` and `byte2` are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise, execution proceeds at the bytecode following.

```
goto        -no change-
goto_w      -no change-
```

Branch always. `byte1` and `byte2` (plus `byte3` and `byte4` for `goto_w`) are used to construct a signed 16-bit (32-bit) offset. Execution proceeds at that offset from the pc.

```
jsr         ... => ..., return-address
jsr-w       ... => ..., return-address
```

Jump subroutine. The address of the bytecode immediately following the `jsr` is pushed onto the stack. `byte1` and `byte2` (plus `byte3` and `byte4` for `goto_w`) are used to construct a signed 16-bit (32-bit) offset. Execution proceeds at that offset from the pc.

```
ret         -no change-
ret2_w      -no change-
```

Return from subroutine. Local variable `byte1` (plus `byte2` for `ret_w` are assembled into a 16-bit index) in the current Java frame must contain a return address. The contents of that local variable are written into the pc.



Note: `jsr` pushes the address onto the stack, and `ret` gets it out of a local variable. This asymmetry is intentional. The `jsr` and `ret` bytecodes are used in the implementation of Java's `finally` keyword.

Method Return

`return` ... => [empty]

Return (void) from method. All values on the operand stack are discarded. The interpreter then returns control to its caller.

```
ireturn    ..., value => [empty]
lreturn    ..., value-word1, value-word2 => [empty]
freturn    ..., value => [empty]
dreturn    ..., value-word1, value-word2 => [empty]
areturn    ..., value => [empty]
```

Return <type> from method. `value` must be a <type>. The `value` is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller. <type> is, in turn, `int`, `long`, `float`, `double`, and object reference.



Note: The stack behavior of the “return” bytecodes may be confusing to anyone expecting the Java operand stack to be just like the C stack. Java's operand stack actually consists of a number of discontinuous segments, each corresponding to a method call. A return bytecode empties the Java operand stack segment corresponding to the frame of the returning call, but does not affect the segment of any parent calls.

Table Jumping

`tableswitch` ..., index => ...

`tableswitch` is a variable-length bytecode. Immediately after the `tableswitch` opcode, zero to three 0 bytes are inserted as padding so that the next byte begins at an address that is a multiple of four. After the padding are a series of signed 4-byte quantities: `default-offset`, `low`, `high`, and

then $(\text{high} - \text{low} + 1)$ further signed 4-byte offsets. These offsets are treated as a 0-based jump table.

The index must be an int. If index is less than low or index is greater than high, default-offset is added to the pc. Otherwise, the $(\text{index} - \text{low})$ 'th element of the jump table is extracted and added to the pc.

lookupswitch ..., key => ...

lookupswitch is a variable-length bytecode. Immediately after the lookupswitch opcode, zero to three 0 bytes are inserted as padding so that the next byte begins at an address that is a multiple of four. Immediately after the padding is a series of pairs of signed 4-byte quantities. The first pair is special; it contains the default-offset and the number of pairs that follow. Each subsequent pair consists of a match and an offset.

The key on the stack must be an int. This key is compared to each of the matches. If it is equal to one of them, the corresponding offset is added to the pc. If the key does not match any of the matches, the default-offset is added to the pc.

Manipulating Object Fields

putfield ..., handle, value => ...
putfield ..., handle, value-word1, value-word2 => ...

Set field in object. byte1 and byte2 are used to construct an index into the constant pool of the current class. The constant pool item is a field reference to a class name and a field name. The item is resolved to a field block pointer containing the field's width and offset (both in bytes).

The field at that offset from the start of the instance pointed to by handle will be set to the value on the top of the stack. The first stack picture is for 32-bit, and the second for 64-bit wide fields. This bytecode handles both. If handle is null, a NullPointerException is thrown. If the specified field is a static field, an IncompatibleClassChangeError is thrown.

getfield ..., handle => ..., value
getfield ..., handle => ..., value-word1, value-word2

Fetch field from object. byte1 and byte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a class name and a field name. The item is resolved to a field block pointer containing the field's width and offset (both in bytes).

handle must be a reference to an object. The value at offset into the object referenced by handle replaces handle on the top of the stack. The first stack picture is for 32-bit, and the second for 64-bit wide fields. This bytecode handles both. If the specified field is a static field, an IncompatibleClassChangeError is thrown.

```
putstatic    ..., value => ...
putstatic    ..., value-word1, value-word2 => ...
```

Set static field in class. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field will be set to have the value on the top of the stack. The first stack picture is for 32-bit, and the second for 64-bit wide fields. This bytecode handles both. If the specified field is not a static field, an `IncompatibleClassChangeError` is thrown.

```
getstatic    ..., => ..., value_
getstatic    ..., => ..., value-word1, value-word2
```

Get static field from class. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The value of that field is placed on the top of the stack. The first stack picture is for 32-bit, and the second for 64-bit wide fields. This bytecode handles both. If the specified field is not a static field, an `IncompatibleClassChangeError` is thrown.

Method Invocation

```
invokevirtual    ..., handle, [arg1, [arg2, ...]], ... => ...
```

Invoke instance method based on run-time type. The operand stack must contain a reference to an object and some number of arguments. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object reference. The method signature is looked up in the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is an index into the method table of the named class that's used to look in the method table of the object's run-time type, where a pointer to the method block for the matched method is found. The method block indicates the type of method (native, synchronized, and so on) and the number of arguments (`nargs`) expected on the operand stack.

If the method is marked synchronized, the monitor associated with `handle` is entered.

The base of the local variables array for the new Java stack frame is set to point to `handle` on the stack, making `handle` and the supplied arguments (`arg1`, `arg2`, ...) the first `nargs` local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first bytecode of the matched method.

If `handle` is null, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

`invokenonvirtual ... , handle, [arg1, [arg2, ...]], ... => ...`

Invoke instance method based on compile-time type. The operand stack must contain a reference (`handle`) to an object and some number of arguments. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature and class. The method signature is looked up in the method table of the class indicated. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, and so on) and the number of arguments (`nargs`) expected on the operand stack. (The last three paragraphs are identical to the previous bytecode.)

`invokestatic ... , , [arg1, [arg2, ...]], ... => ...`

Invoke class (static) method. The operand stack must contain some number of arguments. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature and class. The method signature is looked up in the method table of the class indicated. The method signature is guaranteed to match one of the method signatures in the class's method table exactly.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, and so on) and the number of arguments (`nargs`) expected on the operand stack.

If the method is marked `synchronized`, the monitor associated with the class is entered. (The last two paragraphs are identical to those in `invokevirtual`, except that no `NullPointerException` can be thrown.)

`invokeinterface ... , handle, [arg1, [arg2, ...]], ... => ...`

Invoke interface method. The operand stack must contain a reference (`handle`) to an object and some number of arguments. `byte1` and `byte2` are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object reference. The method signature is looked up in the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, and so on) but, unlike the other "invoke" bytecodes, the number of available arguments (`nargs`) is taken from `byte3`; `byte4` is reserved for future use. (The last three paragraphs are identical to those in `invokevirtual`.)

Exception Handling

throw ... , handle => [undefined]

Throw exception. **handle** must be a handle to an exception object. That exception, which must be an instance of `Throwable` (or a subclass), is thrown. The current Java stack frame is searched for the most recent catch clause that handles the exception. If a matching "catch-list" entry is found, the pc is reset to the address indicated by the catch-list pointer, and execution continues there.

If no appropriate catch clause is found in the current stack frame, that frame is popped and the exception is rethrown, starting the process all over again in the parent frame. If **handle** is null, then a `NullPointerException` is thrown instead.

Miscellaneous Object Operations

new ... => ... , handle

Create new object. **byte1** and **byte2** are used to construct an index into the constant pool of the current class. The item at that index should be a class name that can be resolved to a class pointer, class. A new instance of that class is then created and a reference (**handle**) for the instance is placed on the top of the stack.

checkcast ... , handle => ... , [handle | ...]

Make sure object is of given type. **handle** must be a reference to an object. **byte1** and **byte2** are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name that can be resolved to a class pointer, class.

checkcast determines whether **handle** can be cast to a reference to an object of that class. (A null **handle** can be cast to any class.) If **handle** can be legally cast, execution proceeds at the next bytecode, and the **handle** remains on the stack. If not, a `ClassCastException` is thrown and the stack is emptied.

instanceof ... , handle => ... , result

Determine whether object is of given type. **handle** must be a reference to an object. **byte1** and **byte2** are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name that can be resolved to a class pointer, class.

If **handle** is null, the result is 0 (false). Otherwise, **instanceof** determines whether **handle** can be cast to a reference to an object of that class. The result is 1 (true) if it can, and 0 (false) otherwise.

Monitors

`monitorenter` ..., handle => ...

Enter monitored region of code. `handle` must be a reference to an object. The interpreter attempts to obtain exclusive access via a lock mechanism to `handle`. If another thread already has `handle` locked, the current thread waits until the `handle` is unlocked. If the current thread already has `handle` locked, execution continues normally. If `handle` has no lock on it, this bytecode obtains an exclusive lock. (A `null` in either bytecode throws `NullPointerException`.)

`monitorexit` ..., handle => ...

Exit monitored region of code. `handle` must be a reference to an object. The lock on `handle` is released. If this is the last lock that this thread has on that `handle` (one thread is allowed to have multiple locks on a single `handle`), other threads that are waiting for `handle` are allowed to proceed. (A `null` in either bytecode throws `NullPointerException`.)

Debugging

`breakpoint` -no change-

Call breakpoint handler. The breakpoint bytecode is used to overwrite a bytecode to force control temporarily back to the debugger prior to the effect of the overwritten bytecode. The original bytecode's operands (if any) are not overwritten, and the original bytecode is restored when the breakpoint bytecode is removed.

The *_quick* Bytecodes

The following discussion, straight out of the Java virtual machine documentation, shows you an example of the cleverness mentioned earlier that's needed to make a bytecode interpreter fast:

The following set of pseudo-bytecodes, suffixed by *_quick*, are all variants of standard Java bytecodes. They are used by the run-time to improve the execution speed of the bytecode interpreter. They aren't officially part of the virtual machine specification and are invisible outside a Java virtual machine implementation. However, inside that implementation they have proven to be an effective optimization.

First, you should know that `javac` still generates only non-*_quick* bytecodes. Second, all bytecodes that have a *_quick* variant reference the constant pool. When *_quick* optimization is turned on, each non-*_quick* bytecode (that has a *_quick* variant) resolves the specified item in the constant pool, signals an error if the item in the constant pool could not be resolved for some reason, turns itself into the *_quick* variant of itself, and then performs its intended operation.

This is identical to the actions of the non-`_quick` bytecode, except for the step of overwriting itself with its `_quick` variant. The `_quick` variant of a bytecode assumes that the item in the constant pool has already been resolved, and that this resolution did not produce any errors. It simply performs the intended operation on the resolved item.

Thus, as your bytecodes are being interpreted, they are automatically getting faster and faster! Here are all the `_quick` variants in the current Java run-time:

```
ldc1_quick
ldc2_quick
ldc2w_quick

anewarray_quick
multinewarray_quick

putfield_quick
putfield2_quick
getfield_quick
getfield2_quick
putstatic_quick
putstatic2_quick
getstatic_quick
getstatic2_quick

invokevirtual_quick
invokevirtualobject_quick
invokenonvirtual_quick
invokestatic_quick
invokeinterface_quick

new_quick
checkcast_quick
instanceof_quick
```

If you'd like to go back in today's lesson and look at what each of these does, you can find the name of the original bytecode on which a `_quick` variant is based by simply removing the `_quick` from its name. The bytecodes `putstatic`, `getstatic`, `putfield`, and `getfield` have two `_quick` variants each, one for each stack picture in their original descriptions. `invokevirtual` has two variants: one for objects and one for arrays (to do fast lookups in `java.lang.Object`).

Note: One last note on the `_quick` optimization, regarding the unusual handling of the constant pool (for detail fanatics only):

When a class is read in, an array `constant_pool[]` of size `nconstants` is created and assigned to a field in the class. `constant_pool[0]` is set to point to a dynamically allocated array that indicates which fields in the `constant_pool` have already been resolved. `constant_pool[1]` through `constant_pool[nconstants - 1]` are set to point at the "type" field that corresponds to this constant item.

When a bytecode is executed that references the constant pool, an index is generated, and `constant_pool[0]` is checked to see whether the index has already been resolved. If so, the value of `constant_pool[index]` is returned. If not, the value of `constant_pool[index]` is resolved to be the actual pointer or data, and overwrites whatever value was already in `constant_pool[index]`.

The .class File Format

You won't be given the entire .class file format here, only a taste of what it's like. (You can read all about it in the release documentation.) It's mentioned here because it is one of the parts of Java that needs to be specified carefully if all Java implementations are to be compatible with one another, and if Java byte codes are expected to travel across arbitrary networks—to and from arbitrary computers and operating systems—and yet arrive safely.

The rest of this section paraphrases, and extensively condenses, the latest release of the .class documentation.

.class files are used to hold the compiled versions of both Java classes and Java interfaces. Compliant Java interpreters must be capable of dealing with all .class files that conform to the following specification.

A Java .class file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two or four 8-bit bytes, respectively. The bytes are joined together in big-endian order. (Use `java.io.DataInput` and `java.io.DataOutput` to read and write class files.)

The class file format is presented below as a series of C-struct-like structures. However, unlike a C struct, there is no padding or alignment between pieces of the structure, each field of the structure may be of variable size, and an array may be of variable size (in this case, some field prior to the array gives the array's dimension). The types `u1`, `u2`, and `u4` represent an unsigned one-, two-, or four-byte quantity, respectively.

Attributes are used at several different places in the .class format. All attributes have the following format:

```
GenericAttribute_info {
    u2 attribute_name;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

The `attribute_name` is a 16-bit index into the class's constant pool; the value of `constant_pool[attribute_name]` is a string giving the name of the attribute. The field `attribute_length` gives the length of the subsequent information in bytes. This length does not

include the four bytes needed to store `attribute_name` and `attribute_length`. In the following text, whenever an attribute is required, names of all the attributes that are currently understood are listed. In the future, more attributes will be added. Class file readers are expected to skip over and ignore the information in any attributes that they do not understand.

The following pseudo-structure gives a top-level description of the format of a class file:

```
ClassFile {
    u4  magic;
    u2  minor_version
    u2  major_version
    u2  constant_pool_count;
    cp_info    constant_pool[constant_pool_count - 1];
    u2  access_flags;
    u2  this_class;
    u2  super_class;
    u2  interfaces_count;
    u2  interfaces[interfaces_count];
    u2  fields_count;
    field_info    fields[fields_count];
    u2  methods_count;
    method_info    methods[methods_count];
    u2  attributes_count;
    attribute_info  attributes[attribute_count];
}
```

Here's one of the smaller structures used:

```
method_info {
    u2  access_flags;
    u2  name_index;
    u2  signature_index;
    u2  attributes_count;
    attribute_info  attributes[attribute_count];
}
```

Finally, here's a sample of one of the later structures in the .class file description:

```
Code_attribute {
    u2  attribute_name_index;
    u2  attribute_length;
    u1  max_stack;
    u1  max_locals;
    u2  code_length;
    u1  code[code_length];
    u2  exception_table_length;
    { u2  start_pc;
      u2  end_pc;
      u2  handler_pc;
      u2  catch_type;
    } exception_table[exception_table_length];
    u2  attributes_count;
    attribute_info  attributes[attribute_count];
}
```

None of this is meant to be completely comprehensible (though you might be able to guess at what a lot of the structure members are for), but just suggestive of the sort of structures that live inside .class files. Because the compiler and run-time sources are available, you can always begin with them if you actually have to read or write .class files yourself. Thus, you don't need to have a deep understanding of the details, even in that case.

Method Signatures

Because method signatures are used in .class files, now is an appropriate time to explore them in the detail promised on earlier days—but they're probably most useful to you when writing the native methods of yesterday's lesson.

NEW TERM A *signature* is a string representing the type of a method, field, or array.

A field signature represents the value of an argument to a method or the value of a variable and is a series of bytes in the following grammar:

```
<field signature> := <field_type>
<field_type>      := <base_type> | <object_type> | <array_type>
<base_type>       := B | C | D | F | I | J | S | Z
<object_type>     := L <full.ClassName> ;
<array_type>      := [ <optional_size> <field_type>
<optional_size>   := [0-9]*
```

Here are the meanings of the base types: B (byte), C (char), D (double), F (float), I (int), J (long), S (short), and Z (boolean).

A return-type signature represents the return value from a method and is a series of bytes in the following grammar:

```
<return signature> := <field_type> | V
```

The character V (void) indicates that the method returns no value. Otherwise, the signature indicates the type of the return value. An argument signature represents an argument passed to a method:

```
<argument signature> := <field_type>
```

Finally, a method signature represents the arguments that the method expects, and the value that it returns:

```
<method_signature> := (<arguments signature>) <return signature>
<arguments signature> := <argument signature>*
```

Let's try out the new rules: a method called `complexMethod()` in the class `my.package.name.ComplexClass` takes three arguments—a long, a boolean, and a two-dimensional array of shorts—and returns this. Then, `(JZ[ (S) Lmy.package.name.ComplexClass;` is its method signature.

A method signature is often prefixed by the name of the method, or by its full package (using an underscore in the place of dots) and its class name followed by a slash / and the name of the method, to form a *complete method signature*. (You saw several of these generated in stub comments yesterday.) Now, at last, you have the full story! Thus, the following:

```
my_package_name_ComplexClass/complexMethod(JZ[ (S) Lmy.package.name.ComplexClass;
```

is the full, complete method signature of `complexMethod()`. (Phew!)

The Garbage Collector

Decades ago, programmers in both the Lisp and the Smalltalk community realized how extremely valuable it is to be able to ignore memory deallocation. They realized that, although allocation is fundamental, deallocation is forced on the programmer by the laziness of the system—it should be able to figure out what is no longer useful, and get rid of it. In relative obscurity, these pioneering programmers developed a whole series of garbage collectors to perform this job, each getting more sophisticated and efficient as the years went by. Finally, now that the mainstream programming community has begun to recognize the value of this automated technique, Java can become the first really widespread application of the technology those pioneers developed.

The Problem

Imagine that you're a programmer in a C-like language (probably not too difficult for you, because these languages are the dominant ones right now). Each time you create something, anything, *dynamically* in such a language, you are completely responsible for tracking the life of this object throughout your program and mentally deciding when it will be safe to deallocate it. This can be quite a difficult (sometimes impossible) task, because any of the other libraries or methods you've called might have "squirreled away" a pointer to the object, unbeknownst to you. When it becomes impossible to know, you simply choose *never* to deallocate the object, or at least to wait until every library and method call involved has completed, which could be nearly as long.

The uneasy feeling you get when writing such code is a natural, healthy response to what is inherently an unsafe and unreliable style of programming. If you have tremendous discipline—and so does everyone who writes every library and method you call—you can, in principle,

survive this responsibility without too many mishaps. But aren't you human? Aren't they? There must be some small slips in this perfect discipline due to error. What's worse, such errors are virtually undetectable, as anyone who's tried to hunt down a stray pointer problem in C will tell you. What about the thousands of programmers who don't have that sort of discipline?

Another way to ask this question is: Why should any programmers be forced to have this discipline, when it is entirely possible for the system to remove this heavy burden from their shoulders?

Software engineering estimates have recently shown that for every 55 lines of production C-like code in the world, there is one bug. This means that your electric razor has about 80 bugs, and your TV, 400. Soon they will have even more, because the size of this kind of embedded computer software is growing exponentially. When you begin to think of how much C-like code is in your car's engine, it should give you pause.

Many of these errors are due to the misuse of pointers, by misunderstanding or by accident, and to the early, incorrect freeing of allocated objects in memory. Java addresses both of these—the former, by eliminating explicit pointers from the Java language altogether and the latter, by including, in every Java system, a garbage collector that solves the problem.

The Solution

Imagine a run-time system that tracks each object you create, notices when the last reference to it has vanished, and frees the object for you. How could such a thing actually work?

One brute-force approach, tried early in the days of garbage collecting, is to attach a reference counter to every object. When the object is created, the counter is set to 1. Each time a new reference to the object is made, the counter is incremented, and each time such a reference disappears, the counter is decremented. Because all such references are controlled by the language—as variables and assignments, for example—the compiler can tell whenever an object reference might be created or destroyed, just as it does in handling the scoping of local variables, and thus it can assist with this task. The system itself “holds onto” a set of root objects that are considered too important to be freed. The class `Object` is one example of such a V.I.P. object. (V.I.O.?) Finally, all that's needed is to test, after each decrement, whether the counter has hit 0. If it has, the object is freed.

If you think carefully about this approach, you will soon convince yourself that it is definitely correct when it decides to free anything. It is so simple that you can immediately tell that it will work. The low-level hacker in you might also feel that if it's *that* simple, it's probably not fast enough to run at the lowest level of the system—and you'd be right.

Think about all the stack frames, local variables, method arguments, return values, and local variables created in the course of even a few hundred milliseconds of a program's life. For each

of these tiny, nano-steps in the program, an extra increment—at best—or decrement, test, and deallocation—at worst—will be added to the running time of the program. In fact, the first garbage collectors were slow enough that many predicted they could never be used at all!

Luckily, a whole generation of smart programmers has invented a big bag of tricks to solve these overhead problems. One trick is to introduce special “transient object” areas that don’t need to be reference counted. The best of these generational scavenging garbage collectors today can take less than 3 percent of the total time of your program—a remarkable feat if you realize that many other language features, such as loop overheads, can be as large or larger!

There are other problems with garbage collection. If you are constantly freeing and reclaiming space in a program, won’t the heap of objects soon become fragmented, with small holes everywhere and no room to create new, large objects? Because the programmer is now free from the chains of manual deallocation, won’t they create even more objects than usual?

What’s worse, there is another way that this simple reference counting scheme is inefficient, in space rather than time. If a long chain of object references eventually comes full circle, back to the starting object, each object’s reference count remains at least 1 *forever*. None of these objects will ever be freed!

Together, these problems imply that a good garbage collector must, every once in a while, step back to compact or to clean up wasted memory.

NEW TERM *Compaction* occurs when a garbage collector steps back and reorganizes memory, eliminating the holes created by fragmentation. Compacting memory is simply a matter of repositioning objects one-by-one into a new, compact grouping that places them all in a row, leaving all the free memory in the heap in one big piece.

Cleaning up the circular garbage still lying around after reference counting is called *marking and sweeping*. A mark-and-sweep of memory involves first marking every root object in the system and then following all the object references inside those objects to new objects to mark, and so on, recursively. Then, when you have no more references to follow, you “sweep away” all the unmarked objects, and compact memory as before.

The good news is that this solves the space problems you were having. The bad news is that when the garbage collector “steps back” and does these operations, a nontrivial amount of time passes during which your program is unable to run—all its objects are being marked, swept, rearranged, and so forth, in what seems like an uninterruptible procedure. Your first hint to a solution is the word “seems.”

Garbage collecting can actually be done a little at a time, between or in parallel with normal program execution, thus dividing up the large time needed to “step back” into numerous so-small-you-don’t-notice-them chunks of time that happen between the cracks. (Of course, years of smart thinking went into the abstruse algorithms that make all this possible!)

One final problem that might worry you a little has to do with these object references. Aren’t these “pointers” scattered throughout your program and not just buried in objects? Even if they’re only in objects, don’t they have to be changed whenever the object they point to is moved by these procedures? The answer to both of these questions is a resounding *yes*, and overcoming them is the final hurdle to making an efficient garbage collector.

There are really only two choices. The first, brute force, assumes that all the memory containing object references needs to be searched on a regular basis, and whenever the object references found by this search match objects that have moved, the old reference is changed. This assumes that there are “hard” pointers in the heap’s memory—ones that point directly to other objects. By introducing various kinds of “soft” pointers, including pointers that are like forwarding addresses, the algorithm improves greatly. Although these brute-force approaches sound slow, it turns out that modern computers can do them fast enough to be useful.



Note: You might wonder how the brute-force techniques identify object references. In early systems, references were specially tagged with a “pointer bit,” so they could be unambiguously located. Now, so-called conservative garbage collectors simply assume that if it looks like an object reference, it is—at least for the purposes of the mark and sweep. Later, when actually trying to update it, they can find out whether it really is an object reference or not.

The final approach to handling object references, and the one Java currently uses, is also one of the very first ones tried. It involves using 100 percent “soft” pointers. An object reference is actually a handle, sometimes call an “OOP,” to the real pointer, and a large object table exists to map these handles into the actual object reference. Although this does introduce extra overhead on almost every object reference (some of which can be eliminated by clever tricks, as you might guess), it’s not too high a price to pay for this incredibly valuable level of indirection.

This indirection allows the garbage collector, for example, to mark, sweep, move, or examine one object at a time. Each object can be independently moved “out from under” a running Java program by changing only the object table entries. This not only allows the “step back” phase to happen in the tiniest steps, but it makes a garbage collector that runs literally in parallel with your program much easier to write. This is what the Java garbage collector does.

Warning: You need to be very careful about garbage collection when you're doing critical, real-time programs (such as those mentioned yesterday that legitimately require native methods)—but how often will your Java code be flying a commercial airliner in real-time, anyway?

Java's Parallel Garbage Collector

Java applies almost all these advanced techniques to give you a fast, efficient, parallel garbage collector. Running in a separate thread, it cleans up the Java environment of almost all trash (it is conservative), silently and in the background, is efficient in both space and time, and never steps back for more than a small amount of time. You should never need to know it's there.

By the way, if you want to force a full mark-and-sweep garbage collection to happen soon, you can do so simply by calling the `System.gc()` method. You might want to do this if you just freed up a majority of the heap's memory in circular garbage, and want it all taken away quickly. You might also call this whenever you're idle, as a hint to the system about when it would be best to come and collect the garbage. This "meta knowledge" is rarely needed by the system, however.

Ideally, you'll never notice the garbage collector, and all those decades of programmers beating their brains out on your behalf will simply let you sleep better at night—and what's wrong with that?

The Security Story

Speaking of sleeping well at night, if you haven't stepped back yet and said, "My Goodness! You mean Java programs will be running rampant on the Internet!?" you better do so now, for it is a legitimate concern. In fact, it is one of the major technical stumbling blocks (the others being mostly social and economic) to achieving the dream of ubiquity and code sharing mentioned earlier in today's lesson.

Why You Should Worry

Any powerful, flexible technology can be abused. As the Net becomes mainstream and widespread, it, too, will be abused. Already, there have been many blips on the security radar screens of those of us who worry about such things, warning that (at least until today), not enough attention has been paid by the computer industry (or the media) to solving some of the

problems that this new world brings with it. One of the benefits of constructively solving security once and for all will be a flowering unseen before in the virtual communities of the Net; whole new economies based on people's attention and creativity will spring to life, rapidly transforming our world in new and positive ways.

The downside to all this new technology, is that we (or someone!) must worry long and hard about how to make the playgrounds of the future safe for our children, and for us. Fortunately, Java is a big part of the answer.

Why You Might Not Have To

What gives me any confidence that the Java language and environment will be *safe*, that it will solve the technically daunting and extremely thorny problems inherent in any good form of security, especially for networks?

One simple reason is the history of the people, and the company, that created Java. Many of them are the very smart programmers referred to throughout the book, who helped pioneer many of the ideas that make Java great and who have worked hard over the decades to make techniques such as garbage collection a mainstream reality. They are technically capable of tackling and solving the hard problems that need to be solved. In particular, from discussions with Chuck McManis, one of Java's security gurus, I have confidence that he has thought through these hard problems deeply, and that he knows what needs to be done.

Sun Microsystems, the company, has been pushing networks as the central theme of all its software for more than a decade. Sun has the engineers and the commitment needed to solve these hard problems, because these same problems are at the very center of both its future business and its vision of the future, in which networking is the center of everything—and global networks are nearly useless without good security. Just this year, Sun has advanced the state of the art in easy-to-use Internet security with its new SunScreen products, and it has assigned Whitfield Diffie to oversee them, who is the man who discovered the underlying ideas on which essentially *all* interesting forms of modern encryption are based.

Enough on “deep background.” What does the Java environment provide *right now* that helps me feel secure?

Java's Security Model

Java protects you against potential “nasty” Java code via a series of interlocking defenses that, together, form an imposing barrier to any and all such attacks.

Caution: Of course, no one can protect you from your own ignorance or carelessness. If you're the kind of person who blindly downloads binary executables from your Internet browser and runs them, you need read no further! You are already in more danger than Java will ever pose.

As a user of this powerful new medium, the Internet, you should educate yourself to the possible threats this new and exciting world entails. In particular, downloading "auto running macros" or reading e-mail with "executable attachments" is just as much a threat as downloading binaries from the Net and running them.

Java does not introduce any new dangers here, but by being the first mainstream use of executable and mobile code on the Net, it is responsible for making people suddenly aware of the dangers that have always been there. Java is already, as you will soon see, much less dangerous than any of these common activities on the Net, and can be made safer still over time. Most of these other (dangerous) activities can never be made safe. So please, do not do them!

A good rule of thumb on the Net is: Don't download anything that you plan to execute (or that will be automatically executed for you) except from someone (or some company) you know well and with whom you've had positive, personal experience. If you don't care about losing all the data on your hard drive, or about your privacy, you can do anything you like, but for most of us, this rule should be law.

Fortunately, Java allows you to relax that law. You can run Java applets from anyone, anywhere, in relative safety.

Java's powerful security mechanisms act at four different levels of the system architecture. First, the Java language itself was designed to be safe, and the Java compiler ensures that source code doesn't violate these safety rules. Second, all bytecodes executed by the run-time are screened to be sure that they also obey these rules. (This layer guards against having an altered compiler produce code that violates the safety rules.) Third, the class loader ensures that classes don't violate name space or access restrictions when they are loaded into the system. Finally, API-specific security prevents applets from doing destructive things. This final layer depends on the security and integrity guarantees from the other three layers.

Let's now examine each of these layers in turn.

The Language and the Compiler

The Java language and its compiler are the first line of defense. Java was designed to be a safe language.

Most other C-like languages have facilities to control access to “objects,” but also have ways to “forge” access to objects (or to parts of objects), usually by (mis-)using pointers. This introduces two fatal security flaws to any system built on these languages. One is that no object can protect itself from outside modification, duplication, or “spoofing” (others pretending to be that object). Another is that a language with powerful pointers is more likely to have serious bugs that compromise security. These pointer bugs, where a “runaway pointer” starts modifying some other object’s memory, were responsible for most of the public (and not-so-public) security problems on the Internet this past decade.

Java eliminates these threats in one stroke by eliminating pointers from the language altogether. There are still pointers of a kind—object references—but these are carefully controlled to be safe: they are unforgeable, and all casts are checked for legality before being allowed. In addition, powerful new array facilities in Java not only help to offset the loss of pointers, but add additional safety by strictly enforcing array bounds, catching more bugs for the programmer (bugs that, in other languages, might lead to unexpected and, thus, bad-guy-exploitable problems).

The language definition, and the compilers that enforce it, create a powerful barrier to any “nasty” Java programmer.

Because an overwhelming majority of the “net-savvy” software on the Internet may soon be Java, its safe language definition and compilers help to guarantee that most of this software has a solid, secure base. With fewer bugs, Net software will be more predictable—a property that thwarts attacks.

Verifying the Bytecodes

What if that “nasty” programmer gets a little more determined, and rewrites the Java compiler to suit his nefarious purposes? The Java run-time, getting the lion’s share of its bytecodes from the Net, can never tell whether those bytecodes were generated by a “trustworthy” compiler. Therefore, it must *verify* that they meet all the safety requirements.

Before running any bytecodes, the run-time subjects them to a rigorous series of tests that vary in complexity from simple format checks all the way to running a theorem prover, to make certain that they are playing by the rules. These tests verify that the bytecodes do not forge pointers, violate access restrictions, access objects as other than what they are (InputStreams are always used as InputStreams, and never as anything else), call methods with inappropriate argument values or types, nor overflow the stack.

Consider the following Java code sample:

```
public class VectorTest {
    public int array[];

    public int sum() {
        int[] localArray = array;
        int sum = 0;

        for (int i = localArray.length; --i >= 0; )
            sum += localArray[i];
        return sum;
    }
}
```

The bytecodes generated when this code is compiled look something like the following:

aload_0	Load this
getfield #10	Load this.array
astore_1	Store in localArray
iconst_0	Load 0
istore_2	Store in sum
aload_1	Load localArray
arraylength	Gets its length
istore_3	Store in i
A: iinc 3 -1	Subtract 1 from i
iload_3	Load i
iflt 8	Exit loop if < 0
iload_2	Load sum
aload_1	Load localArray
iload_3	Load i
iaload	Load localArray[i]
iadd	Add sum
istore_2	Store in sum
goto A	Do it again
B: iload_2	Load sum
ireturn	Return it



Note: The excellent examples and descriptions in this section of the book are paraphrased from the tremendously informative security paper in the (alpha) Java release. I'd encourage you to read whatever the latest version of this document is in newer releases, if you want to follow the ongoing Java security story.

Extra Type Information and Requirements

Java bytecodes encode more type information than is strictly necessary for the interpreter. Even though, for example, the `aload` and `iload` opcodes do exactly the same thing, `aload` is always used to load an object reference and `iload` used to load an integer. Some bytecodes (such as `getfield`) include a symbol table reference—and that symbol table has *even more* type information. This extra type information allows the run-time system to guarantee that Java objects and data aren't illegally manipulated.

Conceptually, before and after each bytecode is executed, every slot in the stack and every local variable has some type. This collection of type information—all the slots and local variables—is called the *type state* of the execution environment. An important requirement of the Java type state is that it must be determinable statically by induction—that is, before any program code is executed. As a result, as the run-time systems reads bytecodes, each is required to have the following inductive property: given only the type state before the execution of the bytecode, the type state afterward must be fully determined.

Given “straight-line” bytecodes (no branches), and starting with a known stack state, the state of each slot in the stack is therefore always known. For example, starting with an empty stack:

<code>iload_1</code>	Load integer variable. Stack type state is <code>I</code> .
<code>iconst 5</code>	Load integer constant. Stack type state is <code>II</code> .
<code>iadd</code>	Add two integers, producing an integer. Stack type state is <code>I</code> .



Note: Smalltalk and PostScript bytecodes do not have this restriction. Their more dynamic type behavior does create additional flexibility in those systems, but Java needs to provide a secure execution environment. It must therefore know all types *at all times*, in order to guarantee a certain level of security.

Another requirement made by the Java run-time is that when a set of bytecodes can take more than one path to arrive at the same point, all such paths must arrive there with exactly the same type state. This is a strict requirement, and implies, for example, that compilers cannot generate bytecodes that load all the elements of an array onto the stack. (Because each time through such a loop the stack's type state changes, the start of the loop—"the same point" in multiple paths—would have more than one type state, which is not allowed).

The Verifier

Bytecodes are checked for compliance with all these requirements, using the extra type information in a .class file, by a part of the run-time called the *verifier*. It examines each bytecode in turn, constructing the full type state as it goes, and verifies that all the types of parameters, arguments, and results are correct. Thus, the verifier acts as a gatekeeper to your run-time environment, letting in only those bytecodes that pass muster.

Warning: The verifier is *the crucial piece* of Java's security, and it depends on your having a correctly implemented (no bugs, intentional or otherwise) run-time system. As of this writing, only Sun is producing Java run-times, and theirs are secure. In the future, however, you should be careful when downloading or buying another company's (or individual's) version of the Java run-time environment. Eventually, Sun will implement validation suites for run-times, compilers, and so forth to be sure that they are safe and correct. In the meantime, *caveat emptor!* Your run-time is the base on which all the rest of Java's security is built, so make sure it is a good, solid, secure base.

When bytecodes have passed the verifier, they are guaranteed not to: cause any operand stack under- or overflows; use parameter, argument, or return types incorrectly; illegally convert data from one type to another (from an integer to a pointer, for example); nor access any object's fields illegally (that is, the verifier checks that the rules for public, private, package, and protected are obeyed).

As an added bonus, because the interpreter can now count on all these facts being true, it can run much faster than before. All the required checks for safety have been done up front, so it can run at full throttle. In addition, object references can now be treated as capabilities, because they are unforgeable—capabilities allow, for example, advanced security models for file I/O and authentication to be safely built on top of Java.



Note: Because you can now trust that a private variable really is private, and that no bytecode can perform some magic with casts to extract information from it (such as your credit card number), many of the security problems that might arise in other, less safe environments simply vanish! These guarantees also make erecting barriers against destructive applets possible, and easier. Because the Java system doesn't have to worry about "nasty" bytecodes, it can get on with creating the other levels of security it wants to provide to you.

The Class Loader

The class loader is another kind of gatekeeper, albeit a higher-level one. The verifier was the security of last resort. The class loader is the security of first resort.

When a new class is loaded into the system, it is placed into (lives in) one of several different "realms." In the current release, there are three possible realms: your local computer, the firewall-guarded local network on which your computer is located, and the Internet (the global Net). Each of these realms is treated differently by the class loader.



Note: Actually, there can be as many realms as your desired level of security (or paranoia) requires. This is because the class loader is under your control. As a programmer, you can make your own class loader that implements your own peculiar brand of security. (This is a radical step: you may have to give the users of your program a whole bunch of classes—and they give you a whole lot of trust—to accomplish this.)

As a user, you can tell your Java-capable browser, or Java system, what realm of security (of the three) you'd like it to implement for you right now, or from now on.

As a system administrator, Java has global security policies that you can set up to help guide your users to not "give away the store" (that is, set all their preferences to be unrestricted, promiscuous, "hurt me please!").

In particular, the class loader never allows a class from a "less protected" realm to replace a class from a more protected realm. The file system's I/O primitives, about which you should be *very* worried (and rightly so), are all defined in a local Java class, which means that they all live in the local-computer realm. Thus, no class from outside your computer (from either the supposedly trustworthy local network or from the Internet) can take the place of these classes and "spoof"

Java code into using “nasty” versions of these primitives. In addition, classes in one realm cannot call upon the methods of classes in other realms, unless those classes have explicitly declared those methods `public`. This implies that classes from other than your local computer cannot even *see* the file system I/O methods, much less call them, unless you or the system wants them to.

In addition, every new applet loaded from the network is placed into a separate package-like namespace. This means that applets are protected even from each other! No applet can access another’s methods (or variables) without its cooperation. Applets from inside the firewall can even be treated differently from those outside the firewall, if you like.



Note: Actually, it’s all a little more complex than this. In the current release, an applet is in a package “namespace” along with any other applets from that *source*. This source, or origin, is most often a host (domain name) on the Internet. This special “subrealm” is used extensively in the next section. Depending on where the source is located, outside the firewall (or inside), further restrictions may apply (or be removed entirely). This model is likely to be extended in future releases of Java, providing an even finer degree of control over which classes get to do what.

The class loader essentially partitions the world of Java classes into small, protected little groups, about which you can safely make assumptions that will *always* be true. This type of predictability is the key to well-behaved and secure programs.

You’ve now seen the full lifetime of a method. It starts as source code on some computer, is compiled into bytecodes on some (possibly different) computer, and can then travel (as a .class file) into any file system or network anywhere in the world. When you run an applet in a Java-capable browser (or download a class and run it by hand using `java`), the method’s bytecodes are extracted from its .class file and carefully looked over by the verifier. Once they are declared safe, the interpreter can execute them for you (or a code generator can generate native binary code for them using either the “just-in-time” compiler or `java2c`, and then run that native code directly).

At each stage, more and more security is added. The final level of that security is the Java class library itself, which has several carefully designed classes and APIs that add the final touches to the security of the system.

The Security Manager

`SecurityManager` is an abstract class that was recently added to the Java system to collect, in one place, all the security policy decisions that the system has to make as bytecodes run. You learned before that you can create your own class loader. In fact, you may not have to, because you can subclass `SecurityManager` to perform most of the same customizations.

An instance of some subclass of `SecurityManager` is always installed as the current security manager. It has complete control over which of a well-defined set of “dangerous” methods are allowed to be called by any given class. It takes the realms from the last section into account, the source (origin) of the class, and the type of the class (stand-alone, or loaded by an applet). Each of these can be separately configured to have the effect you (the programmer) like on your Java system. For nonprogrammers, the system provides several levels of default security policies from which you can choose.

What is this “well-defined set” of methods that are protected?

File I/O is a part of the set, for obvious reasons. Applets, by default, can open, read, or write files only with the express permission of the user—and even then, only in certain restricted directories. (Of course, users can always be stupid about this, but that’s what system administrators are for!)

Also in this protected set are the methods that create and use network connections, both incoming and outgoing.

The final members of the set are those methods that allow one thread to access, control, and manipulate other threads. (Of course, additional methods can be protected as well, by creating a new subclass of `SecurityManager` that handles them.)

For both file and network access, the user of a Java-capable browser can choose between three realms (and one subrealm) of protection:

- ☐ *unrestricted* (allows applets to do anything)
- ☐ *firewall* (allows applets within the firewall to do anything)
- ☐ *source* (allows applets to do things only with their origin Internet host, or with other applets from there)
- ☐ *local* (disallows all file and network access)

For file access, the *source* subrealm is not meaningful, so it really has only three realms of protection. (As a programmer, of course, you have full access to the security manager and can set up your own peculiar criteria for granting and revoking privileges to your heart’s content.)

For network access, you can imagine wanting many more realms. For example, you might specify different groups of trusted domains (companies), each of which is allowed added privileges when applets from that group are loaded. Some groups can be more trusted than others, and you might even allow groups to grow automatically by allowing existing members to recommend new members for admission. (The Java seal of approval?)

In any case, the possibilities are endless, as long as there is a secure way of recognizing the original creator of an applet.

You might think this problem has already been solved, because classes are tagged with their origin. In fact, the Java run-time goes far out of its way to be sure that that origin information is never lost—any executing method can be dynamically restricted by this information anywhere in the call chain. So why *isn't* this enough?

Because what you'd really like to be able to do is permanently "tag" an applet with its original creator (its true origin), and no matter where it has traveled, a browser could verify the integrity and authenticate the creator of that applet. Just because you don't know the company or individual that operates a particular server machine doesn't mean that you *want* to mistrust every applet stored on that machine. It's just that, currently, to be really safe, you *should* mistrust those applets.

If somehow those applets were irrevocably tagged with a digital signature by their creator, and that signature could also guarantee that the applet had not been tampered with, you'd be golden.

Note: Luckily, Sun is planning to do exactly that for Java, as soon as export restrictions can be resolved.

Here's a helpful hint of where the team would like to go, from the security documentation: "...a mechanism exists whereby public keys and cryptographic message digests can be securely attached to code fragments that not only identify who originated the code, but guarantee its integrity as well. This latter mechanism will be implemented in future releases."

Look for these sorts of features in every release of Java; they will be a key part of the future of the Internet!

One final note about security. Despite the best efforts of the Java team, there is always a trade-off between useful functionality and absolute security. For example, Java applets can create windows, an extremely useful capability, but a "nasty" applet could use this to spoof the user into typing private password information, by showing a familiar program (or operating system) window and then asking an expected, legitimate-looking question in it. (The beta release adds a special banner to applet-created windows to solve this problem.)

Flexibility and security can't both be maximized. Thus far on the Net, people have chosen maximum flexibility, and have lived with the minimal security the Net now provides. Let's hope that Java can help tip the scales a bit, enabling much better security, while sacrificing only a minimal amount of the flexibility that has drawn so many to the Net.

Summary

Today, you learned about the grand vision that some of us have for Java, and about the exciting future it promises.

Under the hood, the inner workings of the virtual machine, the bytecode interpreter (and all its bytecodes), the garbage collector, the class loader, the verifier, the security manager, and the powerful security features of Java were all revealed.

You now know *almost* enough to write a Java run-time environment of your own—but luckily, you don't have to. You can simply download the latest release of Java—or use a Java-capable browser to enjoy most of the benefits of Java right away.

I hope that Java ends up opening new roads in your mind, as it has in mine.

Q&A

Q I'm still a little unclear about why the Java language and compiler make the Net safer. Can't they just be "side-stepped" by nasty bytecodes?

A Yes, they can—but don't forget that the whole point of using a safe language and compiler was to make the Net *as a whole* safer as more Java code is written. An overwhelming majority of this Java code will be written by "honest" Java programmers, who will produce safe bytecodes. This makes the Net more predictable over time, and thus more secure.

Q I know you said that garbage collection is something I don't have to worry about, but what if I want (or need) to?

A So, you *are* planning to fly a plane with Java. Cool! For just such cases, there is a way to ask the Java run-time, during startup (`java -noasyncgc`), *not* to run garbage collection unless forced to, either by an explicit call (`System.gc()`) or by running out of memory. (This can be quite useful if you have multiple threads that are messing each other up and want to "get the gc thread out of the way" while testing them.) Don't forget that turning garbage collection off means that any object you create will live a *long, long time*. If you're real-time, you never want to "step back" for a full gc—so be sure to reuse objects often, and don't create too many of them!

Q I like the control above; is there anything else I can do to the garbage collector?

A You can also force the `finalize()` methods of any recently freed objects to be called immediately via `System.runFinalization()`. You might want to do this if you're about to ask for some resources that you suspect might still be tied up by objects that are "gone but not forgotten" (waiting for `finalize()`). This is even rarer than starting a gc by hand, but it's mentioned here for completeness.

Q What's the last word on Java?

A Java adds much more than it can ever take away. It has always done so for me, and now, I hope it will for you, as well.

The future of the Net is filled with as-yet-undreamt horizons, and the road is long and hard, but Java is a great traveling companion.