Chapter 1

Multithreading

You are probably familiar with *multitasking:* the ability to have more than one program working at what seems like the same time. For example, you can print while editing or sending a fax. Of course, unless you have a multiple-processor machine, what is really going on is that the operating system is doling out resources to each program, giving the impression of parallel activity. This resource distribution is possible because while you may think you are keeping the computer busy by, for example, entering data, most of the CPU's time will be idle. (A fast typist takes around 1/20 of a second per character typed, after all, which is a huge time interval for a computer.)

Multitasking can be done in two ways, depending on whether the operating system interrupts programs without consulting with them first, or whether programs are only interrupted when they are willing to yield control. The former is called *preemptive multitasking;* the latter is called *cooperative* (or, simply, nonpreemptive) *multitasking.* Windows 3.1 and Mac OS 9 are cooperative multitasking systems, and UNIX/Linux, Windows NT (and Windows 95 for 32-bit programs),

and OS X are preemptive. (Although harder to implement, preemptive multitasking is much more effective. With cooperative multitasking, a badly behaved program can hog everything.)

Multithreaded programs extend the idea of multitasking by taking it one level lower: individual programs will appear to do multiple tasks at the same time. Each task is usually called a *thread*—which is short for thread of control. Programs that can run more than one thread at once are said to be *multithreaded.* Think of each thread as running in a separate context: contexts make it seem as though each thread has its own CPU—with registers, memory, and its own code.

So, what is the difference between multiple *processes* and multiple *threads?* The essential difference is that while each process has a complete set of its own variables, threads share the same data. This sounds somewhat risky, and indeed it can be, as you will see later in this chapter. But it takes much less overhead to create and destroy individual threads than it does to launch new processes, which is why all modern operating systems support multithreading. Moreover, inter-process communication is much slower and more restrictive than communication between threads.

Multithreading is extremely useful in practice. For example, a browser should be able to simultaneously download multiple images. An email program should let you read your email while it is downloading new messages. The Java programming language itself uses a thread to do garbage collection in the background—thus saving you the trouble of managing memory! Graphical user interface (GUI) programs have a separate thread for gathering user interface events from the host operating environment. This chapter shows you how to add multithreading capability to your Java applications and applets.

Fair warning: multithreading can get very complex. In this chapter, we present all of the tools that the Java programming language provides for thread programming. We explain their use and limitations and give some simple but typical examples. However, for more intricate situations, we suggest that you turn to a more advanced reference, such as *Concurrent Programming in Java* by Doug Lea [Addison-Wesley 1999].

NOTE: In many programming languages, you have to use an external thread package to do multithreaded programming. The Java programming language builds in multithreading, which makes your job much easier.

What Are Threads?

Let us start by looking at a program that does not use multiple threads and that, as a consequence, makes it difficult for the user to perform several tasks with that program. After we dissect it, we will then show you how easy it is

to have this program run separate threads. This program animates a bouncing ball by continually moving the ball, finding out if it bounces against a wall, and then redrawing it. (See Figure 1–1.)

As soon as you click on the "Start" button, the program launches a ball from the upper-left corner of the screen and the ball begins bouncing. The handler of the "Start" button calls the `addBall` method:

```
public void addBall()
{
  try
   {
      Ball b = new Ball(canvas);
      canvas.add(b);
      for (int i = 1; i <= 1000; i++)
      {
         b.move();
         Thread.sleep(5);
      }
   }
   catch (InterruptedException exception)
   {
   }
}
```

That method contains a loop running through 1,000 moves. Each call to `move` moves the ball by a small amount, adjusts the direction if it bounces against a wall, and then redraws the canvas. The static `sleep` method of the `Thread` class pauses for 5 milliseconds.

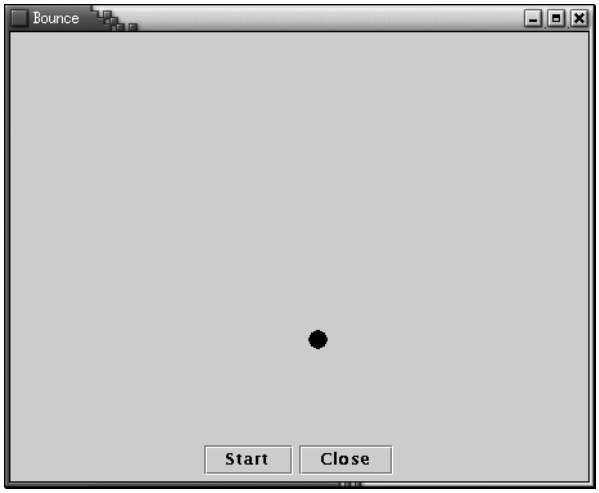

Figure 1–1: Using a thread to animate a bouncing ball

The call to `Thread.sleep` does not create a new thread—`sleep` is a static method of the `Thread` class that temporarily stops the activity of the current thread.

The `sleep` method can throw an `InterruptedException`. We will discuss this exception and its proper handling later. For now, we simply terminate the bouncing if this exception occurs.

If you run the program, the ball bounces around nicely, but it completely takes over the application. If you become tired of the bouncing ball before it has finished its 1,000 bounces and click on the "Close" button, the ball continues bouncing anyway. You cannot interact with the program until the ball has finished bouncing.

NOTE: If you carefully look over the code at the end of this section, you will notice the call

```
canvas.paint(canvas.getGraphics())
```

inside the `move` method of the `Ball` class. That is pretty strange—normally, you'd call `repaint` and let the AWT worry about getting the graphics context and doing the painting. But if you try to call `canvas.repaint()` in this program, you'll find out that the canvas is never repainted since the `addBall` method has completely taken over all processing. In the next program, where we use a separate thread to compute the ball position, we'll again use the familiar `repaint`.

Obviously, the behavior of this program is rather poor. You would not want the programs that you use behaving in this way when you ask them to do a time-consuming job. After all, when you are reading data over a network connection, it is all too common to be stuck in a task that you would *really* like to interrupt. For example, suppose you download a large image and decide, after seeing a piece of it, that you do not need or want to see the rest; you certainly would like to be able to click on a "Stop" or "Back" button to interrupt the loading process. In the next section, we will show you how to keep the user in control by running crucial parts of the code in a separate *thread.*

Example 1–1 is the entire code for the program.

Example 1–1: Bounce.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.    Shows an animated bouncing ball.
9. */
```


```
10. public class Bounce
11. {
12.    public static void main(String[] args)
13.    {
14.       JFrame frame = new BounceFrame();
15.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16.       frame.show();
17.    }
18. }
19.
20. /**
21.    The frame with canvas and buttons.
22. */
23. class BounceFrame extends JFrame
24. {
25.    /**
26.       Constructs the frame with the canvas for showing the
27.       bouncing ball and Start and Close buttons
28.    */
29.    public BounceFrame()
30.    {
31.       setSize(WIDTH, HEIGHT);
32.       setTitle("Bounce");
33.
34.       Container contentPane = getContentPane();
35.       canvas = new BallCanvas();
36.       contentPane.add(canvas, BorderLayout.CENTER);
37.       JPanel buttonPanel = new JPanel();
38.       addButton(buttonPanel, "Start",
39.          new ActionListener()
40.             {
41.                public void actionPerformed(ActionEvent evt)
42.                {
43.                   addBall();
44.                }
45.             });
46.
47.       addButton(buttonPanel, "Close",
48.          new ActionListener()
49.             {
50.                public void actionPerformed(ActionEvent evt)
51.                {
52.                   System.exit(0);
53.                }
54.             });
55.       contentPane.add(buttonPanel, BorderLayout.SOUTH);
56.    }
57.
58.    /**
59.       Adds a button to a container.
60.       @param c the container
```

```
61.       @param title the button title
62.       @param listener the action listener for the button
63.    */
64.    public void addButton(Container c, String title,
65.       ActionListener listener)
66.    {
67.       JButton button = new JButton(title);
68.       c.add(button);
69.       button.addActionListener(listener);
70.    }
71.
72.    /**
73.       Adds a bouncing ball to the canvas and makes
74.       it bounce 1,000 times.
75.    */
76.    public void addBall()
77.    {
78.       try
79.       {
80.          Ball b = new Ball(canvas);
81.          canvas.add(b);
82.
83.          for (int i = 1; i <= 1000; i++)
84.          {
85.             b.move();
86.             Thread.sleep(5);
87.          }
88.       }
89.       catch (InterruptedException exception)
90.       {
91.       }
92.    }
93.
94.    private BallCanvas canvas;
95.    public static final int WIDTH = 450;
96.    public static final int HEIGHT = 350;
97. }
98.
99. /**
100.    The canvas that draws the balls.
101. */
102. class BallCanvas extends JPanel
103. {
104.    /**
105.       Add a ball to the canvas.
106.       @param b the ball to add
107.    */
108.    public void add(Ball b)
109.    {
110.       balls.add(b);
111.    }
112.
```

```
113.    public void paintComponent(Graphics g)
114.    {
115.       super.paintComponent(g);
116.       Graphics2D g2 = (Graphics2D)g;
117.       for (int i = 0; i < balls.size(); i++)
118.       {
119.          Ball b = (Ball)balls.get(i);
120.          b.draw(g2);
121.       }
122.    }
123.
124.    private ArrayList balls = new ArrayList();
125. }
126.
127. /**
128.    A ball that moves and bounces off the edges of a
129.    component
130. */
131. class Ball
132. {
133.    /**
134.       Constructs a ball in the upper left corner
135.       @c the component in which the ball bounces
136.    */
137.    public Ball(Component c) { canvas = c; }
138.
139.    /**
140.       Draws the ball at its current position
141.       @param g2 the graphics context
142.    */
143.    public void draw(Graphics2D g2)
144.    {
145.       g2.fill(new Ellipse2D.Double(x, y, XSIZE, YSIZE));
146.    }
147.
148.    /**
149.       Moves the ball to the next position, reversing direction
150.       if it hits one of the edges
151.    */
152.    public void move()
153.    {
154.       x += dx;
155.       y += dy;
156.       if (x < 0)
157.       {
158.          x = 0;
159.          dx = -dx;
160.       }
161.       if (x + XSIZE >= canvas.getWidth())
162.       {
163.          x = canvas.getWidth() - XSIZE;
164.          dx = -dx;
```

```
165.       }
166.       if (y < 0)
167.       {
168.          y = 0;
169.          dy = -dy;
170.       }
171.       if (y + YSIZE >= canvas.getHeight())
172.       {
173.          y = canvas.getHeight() - YSIZE;
174.          dy = -dy;
175.       }
176.
177.       canvas.paint(canvas.getGraphics());
178.    }
179.
180.    private Component canvas;
181.    private static final int XSIZE = 15;
182.    private static final int YSIZE = 15;
183.    private int x = 0;
184.    private int y = 0;
185.    private int dx = 2;
186.    private int dy = 2;
187. }
```

java.lang.Thread

- `static void sleep(long millis)`
  sleeps for the given number of milliseconds

  Parameters: `millis` the number of milliseconds to sleep

In the previous sections, you learned what is required to split a program into multiple concurrent tasks. Each task needs to be placed into a `run` method of a class that extends `Thread`. But what if we want to add the `run` method to a class that already extends another class? This occurs most often when we want to add multithreading to an applet. An applet class already inherits from `JApplet`, and we cannot inherit from two parent classes, so we need to use an interface. The necessary interface is built into the Java platform. It is called `Runnable`. We take up this important interface next.

Using Threads to Give Other Tasks a Chance

We will make our bouncing-ball program more responsive by running the code that moves the ball in a separate thread.

NOTE: Since most computers do not have multiple processors, the Java virtual machine (JVM) uses a mechanism in which each thread gets a chance to run for a little while, then activates another thread. The virtual machine generally relies on the host operating system to provide the thread scheduling package.

Our next program uses *two* threads: one for the bouncing ball and another for the *event dispatch thread* that takes care of user interface events. Because each thread gets a chance to run, the main thread has the opportunity to notice when you click on the "Close" button while the ball is bouncing. It can then process the "close" action.

There is a simple procedure for running code in a separate thread: place the code into the `run` method of a class derived from `Thread`.

To make our bouncing-ball program into a separate thread, we need only derive a class `BallThread` from `Thread` and place the code for the animation inside the `run` method, as in the following code:

```
class BallThread extends Thread
{
   . . .
   public void run()
   {
      try
      {
         for (int i = 1; i <= 1000; i++)
         {
            b.move();
            sleep(5);
         }
      }
      catch (InterruptedException exception)
      {
      }
   }
   . . .
}
```

You may have noticed that we are catching an exception called `InterruptedException`. Methods such as `sleep` and `wait` throw this exception when your thread is interrupted because another thread has called the `interrupt` method. Interrupting a thread is a very drastic way of getting the thread's attention, even when it is not active. Typically, a thread is interrupted to terminate it. Accordingly, our `run` method exits when an `InterruptedException` occurs.

Running and Starting Threads

When you construct an object derived from `Thread`, the `run` method is not automatically called.

```
BallThread thread = new BallThread(. . .); // won't run yet
```

You must call the `start` method in your object to actually start a thread.

```
thread.start();
```

CAUTION: Do *not* call the `run` method directly—`start` will call it when the thread is set up and ready to go. Calling the `run` method directly merely executes its contents in the *same* thread—no new thread is started.

Beginners are sometimes misled into believing that every method of a `Thread` object automatically runs in a new thread. As you have seen, that is not true. The methods of any object (whether a `Thread` object or not) run in whatever thread they are called. A new thread is *only* started by the `start` method. That new thread then executes the `run` method.

In the Java programming language, a thread needs to tell the other threads when it is idle, so the other threads can grab the chance to execute the code in their `run` procedures. (See Figure 1–2.) The usual way to do this is through the static `sleep` method. The `run` method of the `BallThread` class uses the call to `sleep(5)` to indicate that the thread will be idle for the next five milliseconds. After five milliseconds, it will start up again, but in the meantime, other threads have a chance to get work done.

TIP: There are a number of static methods in the `Thread` class. They all operate on the *current thread,* that is, the thread that executes the method. For example, the static `sleep` method idles the thread that is calling `sleep`.

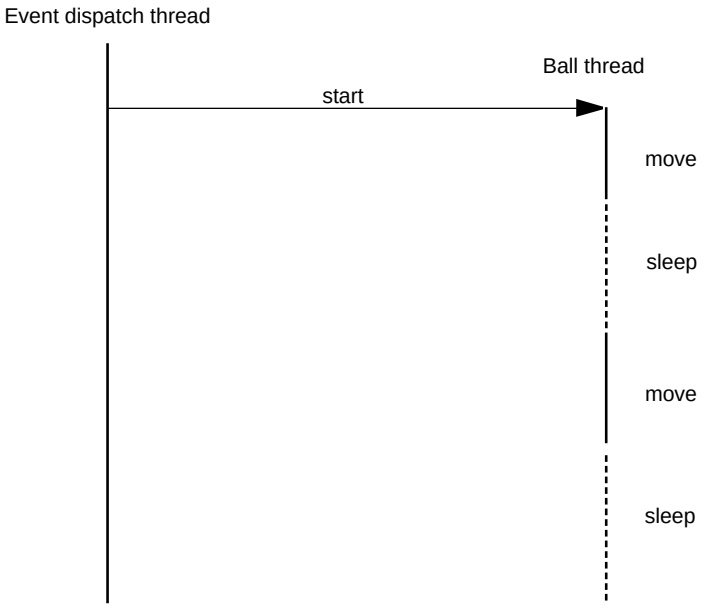

Figure 1–2: The Event Dispatch and Ball Threads

The complete code is shown in Example 1–2.

Example 1–2: BounceThread.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.    Shows an animated bouncing ball running in a separate thread
9. */
10. public class BounceThread
11. {
12.    public static void main(String[] args)
13.    {
14.       JFrame frame = new BounceFrame();
15.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16.       frame.show();
17.    }
18. }
19.
20. /**
21.    The frame with canvas and buttons.
22. */
23. class BounceFrame extends JFrame
24. {
25.    /**
26.       Constructs the frame with the canvas for showing the
27.       bouncing ball and Start and Close buttons
28.    */
29.    public BounceFrame()
30.    {
31.       setSize(WIDTH, HEIGHT);
32.       setTitle("BounceThread");
33.
34.       Container contentPane = getContentPane();
35.       canvas = new BallCanvas();
36.       contentPane.add(canvas, BorderLayout.CENTER);
37.       JPanel buttonPanel = new JPanel();
38.       addButton(buttonPanel, "Start",
39.          new ActionListener()
40.             {
41.                public void actionPerformed(ActionEvent evt)
42.                {
43.                   addBall();
```

```
44.                }
45.             });
46.
47.       addButton(buttonPanel, "Close",
48.          new ActionListener()
49.             {
50.                public void actionPerformed(ActionEvent evt)
51.                {
52.                   System.exit(0);
53.                }
54.             });
55.       contentPane.add(buttonPanel, BorderLayout.SOUTH);
56.    }
57.
58.    /**
59.       Adds a button to a container.
60.       @param c the container
61.       @param title the button title
62.       @param listener the action listener for the button
63.    */
64.    public void addButton(Container c, String title,
65.       ActionListener listener)
66.    {
67.       JButton button = new JButton(title);
68.       c.add(button);
69.       button.addActionListener(listener);
70.    }
71.
72.    /**
73.       Adds a bouncing ball to the canvas and starts a thread
74.       to make it bounce
75.    */
76.    public void addBall()
77.    {
78.       Ball b = new Ball(canvas);
79.       canvas.add(b);
80.       BallThread thread = new BallThread(b);
81.       thread.start();
82.    }
83.
84.    private BallCanvas canvas;
85.    public static final int WIDTH = 450;
86.    public static final int HEIGHT = 350;
87. }
88.
89. /**
90.    A thread that animates a bouncing ball.
```


```
*/
class BallThread extends Thread
{
   /**
      Constructs the thread.
      @aBall the ball to bounce
   */
   public BallThread(Ball aBall) { b = aBall; }

   public void run()
   {
      try
      {
         for (int i = 1; i <= 1000; i++)
         {
            b.move();
            sleep(5);
         }
      }
      catch (InterruptedException exception)
      {
      }
   }

   private Ball b;
}

/**
   The canvas that draws the balls.
*/
class BallCanvas extends JPanel
{
   /**
      Add a ball to the canvas.
      @param b the ball to add
   */
   public void add(Ball b)
   {
      balls.add(b);
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D)g;
      for (int i = 0; i < balls.size(); i++)
      {
```

```
138.          Ball b = (Ball)balls.get(i);
139.          b.draw(g2);
140.       }
141.    }
142.
143.    private ArrayList balls = new ArrayList();
144. }
145.
146. /**
147.    A ball that moves and bounces off the edges of a
148.    component
149. */
150. class Ball
151. {
152.    /**
153.       Constructs a ball in the upper left corner
154.       @c the component in which the ball bounces
155.    */
156.    public Ball(Component c) { canvas = c; }
157.
158.    /**
159.       Draws the ball at its current position
160.       @param g2 the graphics context
161.    */
162.    public void draw(Graphics2D g2)
163.    {
164.       g2.fill(new Ellipse2D.Double(x, y, XSIZE, YSIZE));
165.    }
166.
167.    /**
168.       Moves the ball to the next position, reversing direction
169.       if it hits one of the edges
170.    */
171.    public void move()
172.    {
173.       x += dx;
174.       y += dy;
175.       if (x < 0)
176.       {
177.          x = 0;
178.          dx = -dx;
179.       }
180.       if (x + XSIZE >= canvas.getWidth())
181.       {
182.          x = canvas.getWidth() - XSIZE;
183.          dx = -dx;
184.       }
```


```
185.       if (y < 0)
186.       {
187.          y = 0;
188.          dy = -dy;
189.       }
190.       if (y + YSIZE >= canvas.getHeight())
191.       {
192.          y = canvas.getHeight() - YSIZE;
193.          dy = -dy;
194.       }
195.
196.       canvas.repaint();
197.    }
198.
199.    private Component canvas;
200.    private static final int XSIZE = 15;
201.    private static final int YSIZE = 15;
202.    private int x = 0;
203.    private int y = 0;
204.    private int dx = 2;
205.    private int dy = 2;
206. }
207.
```

java.lang.Thread

- `Thread()`
  constructs a new thread. You must `start` the thread to activate its `run` method.
- `void run()`
  You must override this function and add the code that you want to have executed in the thread.
- `void start()`
  starts this thread, causing the `run()` method to be called. This method will return immediately. The new thread runs concurrently.

Running Multiple Threads

Run the program in the preceding section. Now, click on the “Start” button again while a ball is running. Click on it a few more times. You will see a whole bunch of balls bouncing away, as captured in Figure 1–3. Each ball will move 1,000 times until it comes to its final resting place.

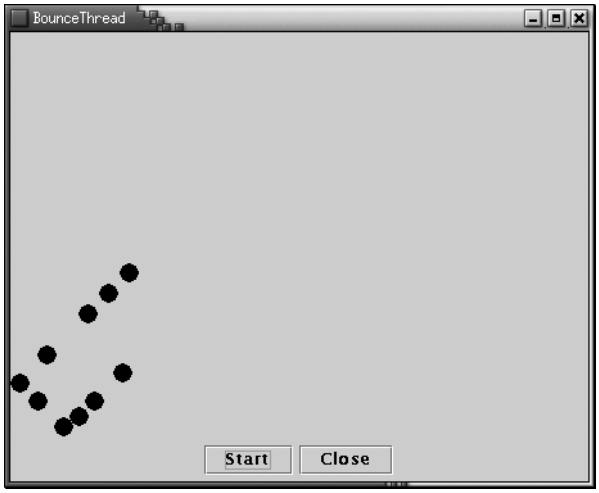

Figure 1–3: Multiple threads

This example demonstrates a great advantage of the thread architecture in the Java programming language. It is very easy to create any number of autonomous objects that appear to run in parallel.

Occasionally, you may want to enumerate the currently running threads—see the API note in the "Thread Groups" section for details.

The `Runnable` *Interface*

We could have saved ourselves a class by having the `Ball` class extend the `Thread` class. As an added advantage of that approach, the `run` method has access to the private fields of the `Ball` class:

```
class Ball extends Thread
{
   public void run()
   {
      try
      {
         for (int i = 1; i <= 1000; i++)
         {
            x += dx;
            y += dy;
            . . .
            canvas.repaint();
            sleep(5);
         }
      }
      catch (InterruptedException exception)
```


```
      {
      }
   }
   . . .
   private Component canvas;
   private int x = 0;
   private int y = 0;
   private int dx = 2;
   private int dy = 2;
}
```

Conceptually, of course, this is dubious. A ball isn't a thread, so inheritance isn't really appropriate. Nevertheless, programmers sometimes follow this approach when the `run` method of a thread needs to access private fields of another class. In the preceding section, we've avoided that issue altogether by having the `run` method call only public methods of the `Ball` class, but it isn't always so easy to do that.

Suppose the `run` method needs access to private fields, but the class into which you want to put the `run` method already has another superclass. Then it can't extend the `Thread` class, but you can make the class implement the `Runnable` interface. As though you had derived from `Thread`, put the code that needs to run in the `run` method. For example,

```
class Animation extends JApplet
   implements Runnable
{
   . . .
   public void run()
   {
      // thread action goes here
   }
}
```

You still need to make a thread object to launch the thread. Give that thread a reference to the `Runnable` object in its constructor. The thread then calls the `run` method of that object.

```
class Animation extends JApplet
   implements Runnable
{
   . . .
   public void start()
   {
      runner = new Thread(this);
      runner.start();
   }
   . . .
   private Thread runner;
}
```

In this case, the `this` argument to the `Thread` constructor specifies that the object whose `run` method should be called when the thread executes is an instance of the `Animation` object.

Some people even claim that you should always follow this approach and never subclass the `Thread` class. That advice made sense for Java 1.0, before inner classes were invented, but it is now outdated. If the `run` method of a thread needs private access to another class, you can often use an inner class, like this:

```
class Animation extends JApplet
{
   . . .
   public void start()
   {
      runner = new Thread()
      {
         public void run()
         {
            // thread action goes here
         }
      };
      runner.start();
   }
   . . .
   private Thread runner;
}
```

A plausible use for the `Runnable` interface would be a thread pool in which pre-spawned threads are kept around for running. Thread pools are sometimes used in environments that execute huge numbers of threads, to reduce the cost of creating and garbage collecting thread objects.

java.lang.Thread

- `Thread(Runnable target)`
  constructs a new thread that calls the `run()` method of the specified target.

java.lang.Runnable

- `void run()`
  You must override this method and place in the thread the code that you want to have executed.

Interrupting Threads

A thread terminates when its `run` method returns. (In the first version of the Java programming environment, there also was a `stop` method that another thread could call to terminate a thread. However, that method is now deprecated. We will discuss the reason later in this chapter.)

There is no longer a way to force a thread to terminate. However, the `interrupt` method can be used to *request* termination of a thread. That means that the `run` method of a thread ought to check once in a while whether it should exit.

```
public void run()
{
   . . .
   while (no request to terminate && more work to do)
   {
      do more work
   }
   // exit run method and terminate thread
}
```

However, as you have learned, a thread should not work continuously, but it should go to sleep or wait once in a while, to give other threads a chance to do their work. But when a thread is sleeping, it can't actively check whether it should terminate. This is where the `InterruptedException` comes in. When the `interrupt` method is called on a thread object that is currently blocked, the blocking call (such as `sleep` or `wait`) is terminated by an `InterruptedException`.

There is no language requirement that a thread that is interrupted should terminate. Interrupting a thread simply grabs its attention. The interrupted thread can decide how to react to the interruption by placing appropriate actions into the `catch` clause that deals with the `InterruptedException`. Some threads are so important that they should simply ignore their interruption by catching the exception and continuing. But quite commonly, a thread will simply want to interpret an interruption as a request for termination. The `run` method of such a thread has the following form:

```
public void run()
{
   try
   {
      . . .
      while (more work to do)
      {
         do more work
      }
   }
   catch(InterruptedException exception)
   {
      // thread was interrupted during sleep or wait
   }
   finally
```

```
   {
      cleanup, if required
   }
   // exit run method and terminate thread
}
```

However, there is a problem with this code skeleton. If the `interrupt` method was called while the thread was not sleeping or waiting, then no `InterruptedException` was generated. The thread needs to call the `interrupted` method to find out if it was recently interrupted.

```
while (!interrupted() && more work to do)
{
   do more work
}
```

In particular, if a thread was blocked while waiting for input/output, the input/output operations are *not* terminated by the call to `interrupt`. When the blocking operation has returned, you need to call the `interrupted` method to find out if the current thread has been interrupted.

NOTE: Curiously, there are two very similar methods, `interrupted` and `isInterrupted`. The `interrupted` method is a static method that checks whether the *current* thread has been interrupted. (Recall that a thread is interrupted because another thread has called its `interrupt` method.) Furthermore, calling the `interrupted` method resets the "interrupted" status of the thread. On the other hand, the `isInterrupted` method is an instance method that you can use to check whether any thread has been interrupted. Calling it does not change the "interrupted" status of its argument.

It is a bit tedious that there are two distinct ways of dealing with thread interruption—testing the "interrupted" flag and catching the `InterruptedException`.

It would have been nice if methods such as `sleep` had been defined to simply return with the "interrupted" flag set when an interruption occurs—then one wouldn't have to deal with the `InterruptedException` at all. Of course, you can manually set the "interrupted" flag when an `InterruptedException` is caught:

```
try
{
   sleep(delay);
}
catch (InterruptedException exception)
{
```


```
    Thread.currentThread().interrupt();
}
```

You need to use this approach if the `sleep` method is called from a method that can't throw any exceptions.

NOTE: You'll find lots of published code where the `InterruptedException` is squelched, like this:

```
try { sleep(delay); }
catch (InterruptedException exception) {} // DON'T!
```

Don't do that! Either set the "interrupted" flag of the current thread, or propagate the exception to the calling method (and ultimately to the `run` method).

If you don't want to clutter up lots of nested methods with `isInterrupted` tests, you can turn the "interrupted" flag into an exception.

```
if (isInterrupted()) throw new InterruptedException();
```

Assuming that your code is already prepared to terminate the `run` method when an `InterruptedException` is thrown, this is a painless way of immediately terminating the thread when an interruption is detected. The principal disadvantage is that you have to tag your methods with

```
throws InterruptedException
```

since, alas, the `InterruptedException` is a checked exception.

`java.lang.Thread`

- `void interrupt()`
  sends an interrupt request to a thread. The "interrupted" status of the thread is set to `true`. If the thread is currently blocked by a call to `sleep` or `wait`, an `InterruptedException` is thrown.
- `static boolean interrupted()`
  tests whether or not the *current* thread (that is, the thread that is executing this instruction) has been interrupted. Note that this is a static method. The call has a side effect—it resets the "interrupted" status of the current thread to `false`.
- `boolean isInterrupted()`
  tests whether or not a thread has been interrupted. Unlike the `static interrupted` method, this call does not change the "interrupted" status of the thread.
- `static Thread currentThread()`
  returns the `Thread` object representing the currently executing thread.

Thread Properties

Thread States

Threads can be in one of four states:

- new
- runnable
- blocked
- dead

Each of these states is explained in the sections that follow.

New threads

When you create a thread with the `new` operator—for example, `new Ball()`—the thread is not yet running. This means that it is in the *new* state. When a thread is in the new state, the program has not started executing code inside of it. A certain amount of bookkeeping needs to be done before a thread can run.

Runnable threads

Once you invoke the `start` method, the thread is *runnable.* A runnable thread may not yet be running. It is up to the operating system to give the thread time to run. When the code inside the thread begins executing, the thread is *running.* (The Java platform documentation does not call this a separate state, though. A running thread is still in the runnable state.)

NOTE: The runnable state has nothing to do with the `Runnable` interface.

Once a thread is running, it doesn't necessarily keep running. In fact, it is desirable if running threads are occasionally interrupted so that other threads have a chance to run. The details of thread scheduling depend on the services that the operating system provides. If the operating system doesn't cooperate, then the Java thread implementation does the minimum to make multithreading work. An example is the so-called *green threads* package that is used by the Java 1.x platform on Solaris. It keeps a running thread active until a higher-priority thread awakes and takes control. Other thread systems (such as Windows 95 and Windows NT) give each runnable thread a slice of time to perform its task. When that slice of time is exhausted, the operating system gives another thread an opportunity to work. This approach is called *time slicing*. Time slicing has an important advantage: an uncooperative thread can't prevent other threads from running. Current releases of the Java platform on Solaris can be configured to allow use of the native Solaris threads, which also perform time-slicing.

Always keep in mind that a runnable thread may or may not be running at any given time. (This is why the state is called "runnable" and not "running.") See Figure 1–4.

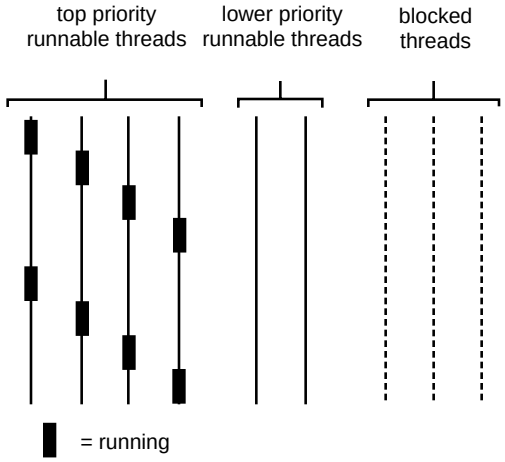

Figure 1–4: Time-slicing on a single CPU

Blocked threads

A thread enters the *blocked* state when one of the following actions occurs:

1. Someone calls the `sleep()` method of the thread.
2. The thread calls an operation that is *blocking on input/output*, that is, an operation that will not return to its caller until input and output operations are complete.
3. The thread calls the `wait()` method.
4. The thread tries to lock an object that is currently locked by another thread. We will discuss object locks later in this chapter.
5. Someone calls the `suspend()` method of the thread. However, this method is deprecated, and you should not call it in your code. We will explain the reason later in this chapter.

Figure 1–5 shows the states that a thread can have and the possible transitions from one state to another. When a thread is blocked (or, of course, when it dies), another thread is scheduled to run. When a blocked thread is reactivated (for example, because it has slept the required number of milliseconds or because the I/O it waited for is complete), the scheduler checks to see if it has a higher priority than the currently running thread. If so, it *preempts* the current thread and picks a new thread to run. (On a machine with multiple processors, each processor can run a thread, and you can have multiple threads run in parallel. On such a

machine, a thread is only preempted if a higher priority thread becomes runnable and there is no available processor to run it.)

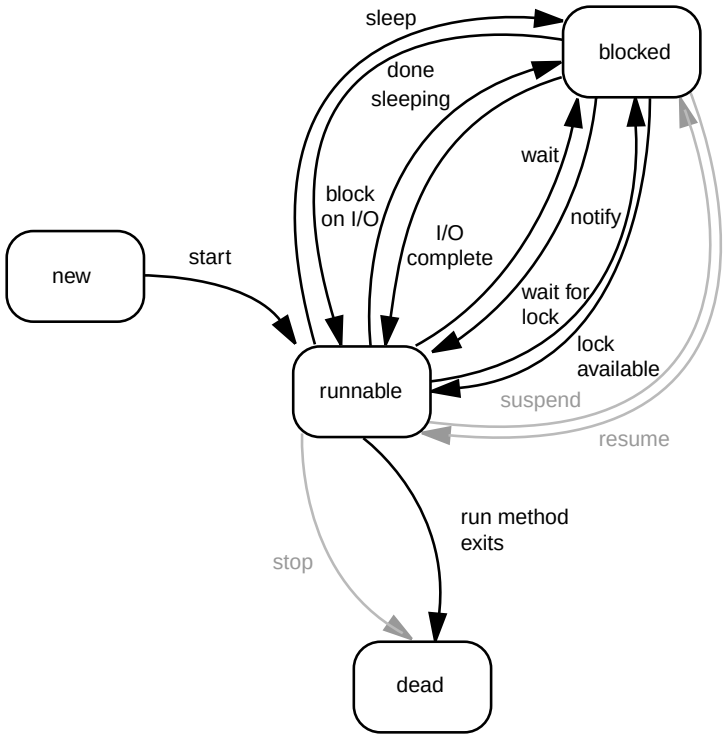

Figure 1–5: Thread states

For example, the `run` method of the `BallThread` blocks itself after it has completed a move, by calling the `sleep` method.

This gives other threads (in our case, other balls and the main thread) the chance to run.

If the computer has multiple processors, then more than one thread has a chance to run at the same time.

Moving Out of a Blocked State

The thread must move out of the blocked state and back into the runnable state, using the opposite of the route that put it into the blocked state.

1. If a thread has been put to sleep, the specified number of milliseconds must expire.
2. If a thread is waiting for the completion of an input or output operation, then the operation must have finished.

3. If a thread called `wait`, then another thread must call `notifyAll` or `notify`. (We cover the `wait` and `notifyAll/notify` methods later in this chapter.)
4. If a thread is waiting for an object lock that was owned by another thread, then the other thread must relinquish ownership of the lock. (You will see the details later in this chapter.)
5. If a thread has been suspended, then someone must call its `resume` method. However, since the `suspend` method has been deprecated, the `resume` method has been deprecated as well, and you should not call it in your own code.

NOTE: A blocked thread can only reenter the runnable state through the same route that blocked it in the first place. For example, if a thread is blocked on input, you cannot call its `resume` method to unblock it.

If you invoke a method on a thread that is incompatible with its state, then the virtual machine throws an `IllegalThreadStateException`. For example, this happens when you call `sleep` on a thread that is currently blocked.

Dead Threads

A thread is dead for one of two reasons:

- It dies a natural death because the `run` method exits normally.
- It dies abruptly because an uncaught exception terminates the `run` method.

In particular, it is possible to kill a thread by invoking its `stop` method. That method throws a `ThreadDeath` error object which kills the thread. However, the `stop` method is deprecated, and you should not call it in your own code. We will explain later in this chapter why the `stop` method is inherently dangerous.

To find out whether a thread is currently alive (that is, either runnable or blocked), use the `isAlive` method. This method returns `true` if the thread is runnable or blocked, `false` if the thread is still new and not yet runnable or if the thread is dead.

NOTE: You cannot find out if an alive thread is runnable or blocked, or if a runnable thread is actually running. In addition, you cannot differentiate between a thread that has not yet become runnable and one that has already died.

Daemon Threads

A thread can be turned into a *daemon thread* by calling

```
t.setDaemon(true);
```

There is nothing demonic about such a thread. A daemon is simply a thread that has no other role in life than to serve others. Examples are timer threads that send regular "timer ticks" to other threads. When only daemon threads remain, then the program exits. There is no point in keeping the program running if all remaining threads are daemons.

Thread Groups

Some programs contain quite a few threads. It then becomes useful to categorize them by functionality. For example, consider an Internet browser. If many threads are trying to acquire images from a server and the user clicks on a "Stop" button to interrupt the loading of the current page, then it is handy to have a way of interrupting all of these threads simultaneously. The Java programming language lets you construct what it calls a *thread group* so you can simultaneously work with a group of threads.

You construct a thread group with the constructor:

```
String groupName = . . .;
ThreadGroup g = new ThreadGroup(groupName)
```

The string argument of the `ThreadGroup` constructor identifies the group and must be unique. You then add threads to the thread group by specifying the thread group in the thread constructor.

```
Thread t = new Thread(g, threadName);
```

To find out whether any threads of a particular group are still runnable, use the `activeCount` method.

```
if (g.activeCount() == 0)
{
   // all threads in the group g have stopped
}
```

To interrupt all threads in a thread group, simply call `interrupt` on the group object.

```
g.interrupt(); // interrupt all threads in group g
```

We'll discuss interrupting threads in greater detail in the next section.

Thread groups can have child subgroups. By default, a newly created thread group becomes a child of the current thread group. But you can also explicitly name the parent group in the constructor (see the API notes). Methods such as `activeCount` and `interrupt` refer to all threads in their group and all child groups.

One nice feature of thread groups is that you can get a notification if a thread died because of an exception. You need to subclass the `ThreadGroup` class and override the `uncaughtException` method. You can then replace the default action (which prints a stack trace to the standard error stream) with something more sophisticated, such as logging the error in a file or displaying a dialog.

java.lang.Thread

- `Thread(ThreadGroup g, String name)`
  creates a new `Thread` that belongs to a given `ThreadGroup`.

  Parameters: `g` the thread group to which the new thread belongs
  `name` the name of the new thread

- `ThreadGroup getThreadGroup()`
  returns the thread group of this thread.
- `boolean isAlive()`
  returns `true` if the thread has started and has not yet terminated.
- `void suspend()`
  suspends this thread's execution. This method is deprecated.
- `void resume()`
  resumes this thread. This method is only valid after `suspend()` has been invoked. This method is deprecated.
- `void setDaemon(boolean on)`
  marks this thread as a daemon thread or a user thread. When there are only daemon threads left running in the system, the program exits. This method must be called before the thread is started.
- `void stop()`
  stops the thread. This method is deprecated.

java.lang.ThreadGroup

- `ThreadGroup(String name)`
  creates a new `ThreadGroup`. Its parent will be the thread group of the current thread.

  Parameters: `name` the name of the new thread group

- `ThreadGroup(ThreadGroup parent, String name)`
  creates a new `ThreadGroup`.

  Parameters: `parent` the parent thread group of the new thread group
  `name` the name of the new thread group

- `int activeCount()`
  returns an upper bound for the number of active threads in the thread group.
- `int enumerate(Thread[] list)`
  gets references to every active thread in this thread group. You can use the `activeCount` method to get an upper bound for the array; this method

returns the number of threads put into the array. If the array is too short (presumably because more threads were spawned after the call to `activeCount`), then as many threads as fit are inserted.

Parameters: `list` an array to be filled with the thread references

- `ThreadGroup getParent()`
  gets the parent of this thread group.
- `void interrupt()`
  interrupts all threads in this thread group and all of its child groups.
- `void uncaughtException(Thread t, Throwable e)`
  Override this method to react to exceptions that terminate any threads in this thread group. The default implementation calls this method of the parent thread group if there is a parent, or otherwise prints a stack trace to the standard error stream. (However, if `e` is a `ThreadDeath` object, then the stack trace is suppressed. `ThreadDeath` objects are generated by the deprecated `stop` method.).

  Parameters: `t` the thread that was terminated due to an uncaught exception

  `e` the uncaught exception object

Thread Priorities

In the Java programming language, every thread has a *priority*. By default, a thread inherits the priority of its parent thread. You can increase or decrease the priority of any thread with the `setPriority` method. You can set the priority to any value between `MIN_PRIORITY` (defined as 1 in the `Thread` class) and `MAX_PRIORITY` (defined as 10). `NORM_PRIORITY` is defined as 5.

Whenever the thread-scheduler has a chance to pick a new thread, it *generally* picks the *highest-priority thread that is currently runnable*.

CAUTION: We need to say right at the outset that the rules for thread priorities are highly system-dependent. When the virtual machine relies on the thread implementation of the host platform, the thread scheduling is at the mercy of that thread implementation. The virtual machine maps the thread priorities to the priority levels of the host platform (which may have more or fewer thread levels). What we describe in this section is an ideal situation that every virtual machine implementation tries to approximate to some degree.

The highest-priority runnable thread keeps running until:

- It yields by calling the `yield` method, or

- It ceases to be runnable (either by dying or by entering the blocked state), or
- A higher-priority thread has become runnable (because the higher-priority thread has slept long enough, or its I/O operation is complete, or someone unblocked it by calling the `notifyAll/notify` method on the object that the thread was waiting for).

Then, the scheduler selects a new thread to run. The highest-priority remaining thread is picked among those that are runnable.

What happens if there is more than one runnable thread with the same (highest) priority? One of the highest priority threads gets picked. It is completely up to the thread scheduler how to arbitrate between threads of the same priority. The Java programming language gives no guarantee that all of the threads get treated *fairly*. Of course, it would be desirable if all threads of the same priority are served in turn, to guarantee that each of them has a chance to make progress. But it is at least theoretically possible that on some platforms a thread scheduler picks a random thread or keeps picking the first available thread. This is a weakness of the Java programming language, and it is difficult to write multithreaded programs that are guaranteed to work identically on all virtual machines.

CAUTION: Some platforms (such as Windows NT) have fewer priority levels than the 10 levels that the Java platform specifies. On those platforms, no matter what mapping of priority levels is chosen, some of the 10 JVM levels will be mapped to the same platform levels. In the Sun JVM for Linux, thread priorities are ignored altogether, so you will not be able to see the "express threads" in action when you run the sample program at the end of this section.

Whenever the host platform uses fewer priority levels than the Java platform, a thread can be preempted by another thread with a seemingly lower priority. That plainly means that you cannot rely on priority levels in your multithreaded programs.

Consider, for example, a call to `yield`. It may have no effect on some implementations. The levels of all runnable threads might map to the same host thread level. The host scheduler might make no effort towards fairness and might simply keep reactivating the yielding thread, even though other threads would like to get their turn. It is a good idea to call `sleep` instead—at least you know that the current thread won't be picked again right away.

Most virtual machines have several (although not necessarily 10) priorities, and thread schedulers make some effort to rotate among equal priority threads. For example, if you watch a number of ball threads in the preceding example

program, all balls progressed to the end and they appeared to get executed at approximately the same rate. Of course, that is not a guarantee. If you need to ensure a fair scheduling policy, you must implement it yourself.

Consider the following example program, which modifies the previous program to run threads of one kind of balls (displayed in red) with a higher priority than the other threads. (The bold line in the code below shows how to increase the priority of the thread.)

If you click on the "Start" button, a thread is launched at the normal priority, animating a black ball. If you click on the "Express" button, then you launch a red ball whose thread runs at a higher priority than the regular ball threads.

```
public class BounceFrame
{
   public BounceFrame()
   {
       . . .
      addButton(buttonPanel, "Start",
         new ActionListener()
            {
               public void actionPerformed(ActionEvent evt)
               {
                  addBall(Thread.NORM_PRIORITY, Color.black);
               }
            });

      addButton(buttonPanel, "Express",
         new ActionListener()
            {
               public void actionPerformed(ActionEvent evt)
               {
                  addBall(Thread.NORM_PRIORITY + 2, Color.red);
               }
            });
     . . .
   }

   public void addBall(int priority, Color color)
   {
      Ball b = new Ball(canvas, color);
      canvas.add(b);
      BallThread thread = new BallThread(b);
      thread.setPriority(priority);
      thread.start();
   }
   . . .
}
```

Try it out. Launch a set of regular balls and a set of express balls. You will notice that the express balls seem to run faster. This is solely a result of their higher priority, *not* because the red balls run at a higher speed. The code to move the express balls is the same as that of the regular balls.

Here is why this demonstration works: five milliseconds after an express thread goes to sleep, the scheduler wakes it. Now:

- The scheduler again evaluates the priorities of all the runnable threads;
- It finds that the express threads have the highest priority.

One of the express threads gets another turn right away. This can be the one that just woke up, or perhaps it is another express thread—you have no way of knowing. The express threads take turns, and only when they are all asleep does the scheduler give the lower-priority threads a chance to run. See Figure 1–6 and Example 1–3.

Note that the lower-priority threads would have *no* chance to run if the express threads had called `yield` instead of `sleep`.

Once again, we caution you that this program works fine on Windows NT and Solaris, but the Java programming language specification gives no guarantee that it works identically on other implementations.

TIP: If you find yourself tinkering with priorities to make your code work, you are on the wrong track. Instead, read the remainder of this chapter, or delve into one of the cited references, to learn more about reliable mechanisms for controlling multithreaded programs.

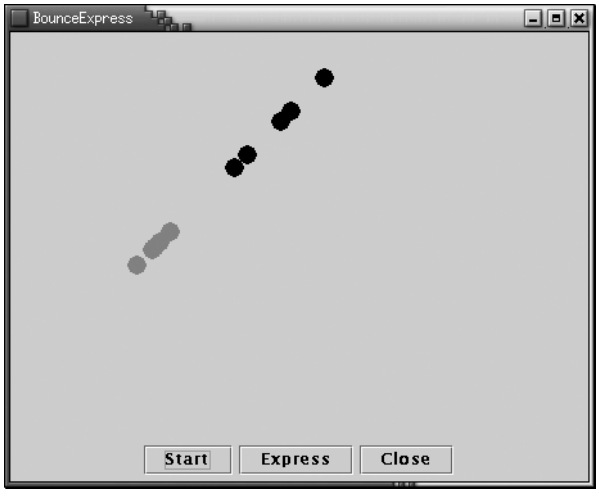

Figure 1–6: Threads with different priorities

Example 1–3: BounceExpress.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.    Shows animated bouncing balls, some running in higher priority
9.    threads
10. */
11. public class BounceExpress
12. {
13.    public static void main(String[] args)
14.    {
15.       JFrame frame = new BounceFrame();
16.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17.       frame.show();
18.    }
19. }
20.
21. /**
22.    The frame with canvas and buttons.
23. */
24. class BounceFrame extends JFrame
25. {
26.    /**
27.       Constructs the frame with the canvas for showing the
28.       bouncing ball and Start and Close buttons
29.    */
30.    public BounceFrame()
31.    {
32.       setSize(WIDTH, HEIGHT);
33.       setTitle("BounceExpress");
34.
35.       Container contentPane = getContentPane();
36.       canvas = new BallCanvas();
37.       contentPane.add(canvas, BorderLayout.CENTER);
38.       JPanel buttonPanel = new JPanel();
39.       addButton(buttonPanel, "Start",
40.          new ActionListener()
41.             {
42.                public void actionPerformed(ActionEvent evt)
43.                {
44.                   addBall(Thread.NORM_PRIORITY, Color.black);
45.                }
46.             });
47.
```


```
48.       addButton(buttonPanel, "Express",
49.          new ActionListener()
50.             {
51.                public void actionPerformed(ActionEvent evt)
52.                {
53.                   addBall(Thread.NORM_PRIORITY + 2, Color.red);
54.                }
55.             });
56.
57.       addButton(buttonPanel, "Close",
58.          new ActionListener()
59.             {
60.                public void actionPerformed(ActionEvent evt)
61.                {
62.                   System.exit(0);
63.                }
64.             });
65.       contentPane.add(buttonPanel, BorderLayout.SOUTH);
66.    }
67.
68.    /**
69.       Adds a button to a container.
70.       @param c the container
71.       @param title the button title
72.       @param listener the action listener for the button
73.    */
74.    public void addButton(Container c, String title,
75.       ActionListener listener)
76.    {
77.       JButton button = new JButton(title);
78.       c.add(button);
79.       button.addActionListener(listener);
80.    }
81.
82.    /**
83.       Adds a bouncing ball to the canvas and starts a thread
84.       to make it bounce
85.       @param priority the priority for the threads
86.       @color the color for the balls
87.    */
88.    public void addBall(int priority, Color color)
89.    {
90.       Ball b = new Ball(canvas, color);
91.       canvas.add(b);
92.       BallThread thread = new BallThread(b);
93.       thread.setPriority(priority);
94.       thread.start();
95.    }
```

```
96.
97.    private BallCanvas canvas;
98.    public static final int WIDTH = 450;
99.    public static final int HEIGHT = 350;
100. }
101.
102. /**
103.    A thread that animates a bouncing ball.
104. */
105. class BallThread extends Thread
106. {
107.    /**
108.       Constructs the thread.
109.       @aBall the ball to bounce
110.    */
111.    public BallThread(Ball aBall) { b = aBall; }
112.
113.    public void run()
114.    {
115.       try
116.       {
117.          for (int i = 1; i <= 1000; i++)
118.          {
119.             b.move();
120.             sleep(5);
121.          }
122.       }
123.       catch (InterruptedException exception)
124.       {
125.       }
126.    }
127.
128.    private Ball b;
129. }
130.
131. /**
132.    The canvas that draws the balls.
133. */
134. class BallCanvas extends JPanel
135. {
136.    /**
137.       Add a ball to the canvas.
138.       @param b the ball to add
139.    */
140.    public void add(Ball b)
141.    {
142.       balls.add(b);
143.    }
```


```
144.
145.    public void paintComponent(Graphics g)
146.    {
147.       super.paintComponent(g);
148.       Graphics2D g2 = (Graphics2D)g;
149.       for (int i = 0; i < balls.size(); i++)
150.       {
151.          Ball b = (Ball)balls.get(i);
152.          b.draw(g2);
153.       }
154.    }
155.
156.    private ArrayList balls = new ArrayList();
157. }
158.
159. /**
160.    A ball that moves and bounces off the edges of a
161.    component
162. */
163. class Ball
164. {
165.    /**
166.       Constructs a ball in the upper left corner
167.       @c the component in which the ball bounces
168.       @aColor the color of the ball
169.    */
170.    public Ball(Component c, Color aColor)
171.    {
172.       canvas = c;
173.       color = aColor;
174.    }
175.
176.    /**
177.       Draws the ball at its current position
178.       @param g2 the graphics context
179.    */
180.    public void draw(Graphics2D g2)
181.    {
182.       g2.setColor(color);
183.       g2.fill(new Ellipse2D.Double(x, y, XSIZE, YSIZE));
184.    }
185.
186.    /**
187.       Moves the ball to the next position, reversing direction
188.       if it hits one of the edges
189.    */
190.    public void move()
191.    {
192.       x += dx;
```

```
193.       y += dy;
194.       if (x < 0)
195.       {
196.          x = 0;
197.          dx = -dx;
198.       }
199.       if (x + XSIZE >= canvas.getWidth())
200.       {
201.          x = canvas.getWidth() - XSIZE;
202.          dx = -dx;
203.       }
204.       if (y < 0)
205.       {
206.          y = 0;
207.          dy = -dy;
208.       }
209.       if (y + YSIZE >= canvas.getHeight())
210.       {
211.          y = canvas.getHeight() - YSIZE;
212.          dy = -dy;
213.       }
214.
215.       canvas.repaint();
216.    }
217.
218.    private Component canvas;
219.    private Color color;
220.    private static final int XSIZE = 15;
221.    private static final int YSIZE = 15;
222.    private int x = 0;
223.    private int y = 0;
224.    private int dx = 2;
225.    private int dy = 2;
226. }
```

java.lang.Thread

- `void setPriority(int newPriority)`
  sets the priority of this thread. The priority must be between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`. Use `Thread.NORM_PRIORITY` for normal priority.
- `static int MIN_PRIORITY`
  is the minimum priority that a `Thread` can have. The minimum priority value is 1.
- `static int NORM_PRIORITY`
  is the default priority of a `Thread`. The default priority is 5.

- `static int MAX_PRIORITY`
  is the maximum priority that a `Thread` can have. The maximum priority value is 10.
- `static void yield()`
  causes the currently executing thread to yield. If there are other runnable threads whose priority is at least as high as the priority of this thread, they will be scheduled next. Note that this is a static method.

Selfish Threads

Our ball threads were well-behaved and gave each other a chance to run. They did this by calling the `sleep` method to wait their turns. The `sleep` method blocks the thread and gives the other threads an opportunity to be scheduled. Even if a thread does not want to put itself to sleep for any amount of time, it can call `yield()` whenever it does not mind being interrupted. A thread should always call `sleep` or `yield` when it is executing a long loop, to ensure that it is not monopolizing the system. A thread that does not follow this rule is called *selfish*.

The following program shows what happens when a thread contains a *tight loop*, a loop in which it carries out a lot of work without giving other threads a chance. When you click on the "Selfish" button, a blue ball is launched whose `run` method contains a tight loop.

```
class BallThread extends Thread
{
   . . .
   public void run()
   {
      try
      {
         for (int i = 1; i <= 1000; i++)
         {
            b.move();
            if (selfish)
            {
               // busy wait for 5 milliseconds
                long t = System.currentTimeMillis();
               while (System.currentTimeMillis() < t + 5)
                  ;
            }
            else
               sleep(5);
         }
      }
      catch (InterruptedException exception)
```

```
        {
        }
    }
    . . .
    private boolean selfish;
}
```

When the `selfish` flag is set, the `run` method will last about five seconds before it returns, ending the thread. In the meantime, it never calls `sleep` or `yield`.

What actually happens when you run this program depends on your operating system and choice of thread implementation. For example, when you run this program under Solaris or Linux with the "green threads" implementation as opposed to the "native threads" implementation, you will find that the selfish ball indeed hogs the whole application. Try closing the program or launching another ball; you will have a hard time getting even a mouse-click into the application. However, when you run the same program under Windows or the native threads implementation, nothing untoward happens. The blue balls can run in parallel with other balls.

The reason for this behavioral difference is that the underlying thread package of the operating system performs *time-slicing*. It periodically interrupts threads in midstream, even if they are not cooperating. When a thread (even a selfish thread) is interrupted, the scheduler activates another thread—picked among the top-priority-level runnable threads. The green threads implementation on Solaris and Linux does not perform time-slicing, but the native thread package does. (Why doesn't everyone simply use the native threads? Until recently, X11 and Motif were not thread safe, and using native threads could lock up the window manager.) Also, as Java gets ported to new platforms, the green threads implementation tends to get implemented first because it is easier to port.

If you *know* that your program will execute on a machine whose thread system performs time-slicing, then you do not need to worry about making your threads polite. But the point of Internet computing is that you generally *do not know* the environments of the people who will use your program. You should, therefore, plan for the worst and put calls to `sleep` or `yield` in every loop.

CAUTION: When programming with threads, expect platform-dependent behavior variations. You can't assume that your threads will get pre-empted by other threads, so you must plan for the case that they run forever unless you yield control. But you can't rely on your threads running without interruption—on most platforms, they won't. You must be prepared that they can lose control at any time.

See Example 1–4 for the complete source code of the "selfish ball" program.

Example 1–4: BounceSelfish.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.    Shows animated bouncing balls, some running in selfish
9.    threads
10. */
11. public class BounceSelfish
12. {
13.    public static void main(String[] args)
14.    {
15.       JFrame frame = new BounceFrame();
16.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17.       frame.show();
18.    }
19. }
20.
21. /**
22.    The frame with canvas and buttons.
23. */
24. class BounceFrame extends JFrame
25. {
26.    /**
27.       Constructs the frame with the canvas for showing the
28.       bouncing ball and Start and Close buttons
29.    */
30.    public BounceFrame()
31.    {
32.       setSize(WIDTH, HEIGHT);
33.       setTitle("BounceSelfish");
34.
35.       Container contentPane = getContentPane();
36.       canvas = new BallCanvas();
37.       contentPane.add(canvas, BorderLayout.CENTER);
38.       JPanel buttonPanel = new JPanel();
39.       addButton(buttonPanel, "Start",
40.          new ActionListener()
41.             {
42.                public void actionPerformed(ActionEvent evt)
43.                {
44.                   addBall(false, Color.black);
45.                }
46.             });
47.
```

```
48.       addButton(buttonPanel, "Selfish",
49.          new ActionListener()
50.             {
51.                public void actionPerformed(ActionEvent evt)
52.                {
53.                   addBall(true, Color.blue);
54.                }
55.             });
56.
57.       addButton(buttonPanel, "Close",
58.          new ActionListener()
59.             {
60.                public void actionPerformed(ActionEvent evt)
61.                {
62.                   System.exit(0);
63.                }
64.             });
65.       contentPane.add(buttonPanel, BorderLayout.SOUTH);
66.    }
67.
68.    /**
69.       Adds a button to a container.
70.       @param c the container
71.       @param title the button title
72.       @param listener the action listener for the button
73.    */
74.    public void addButton(Container c, String title,
75.       ActionListener listener)
76.    {
77.       JButton button = new JButton(title);
78.       c.add(button);
79.       button.addActionListener(listener);
80.    }
81.
82.    /**
83.       Adds a bouncing ball to the canvas and starts a thread
84.       to make it bounce
85.       @param priority the priority for the threads
86.       @color the color for the balls
87.    */
88.    public void addBall(boolean selfish, Color color)
89.    {
90.       Ball b = new Ball(canvas, color);
91.       canvas.add(b);
92.       BallThread thread = new BallThread(b, selfish);
93.       thread.start();
94.    }
95.
```


```
96.    private BallCanvas canvas;
97.    public static final int WIDTH = 450;
98.    public static final int HEIGHT = 350;
99. }
100.
101. /**
102.    A thread that animates a bouncing ball.
103. */
104. class BallThread extends Thread
105. {
106.    /**
107.       Constructs the thread.
108.       @aBall the ball to bounce
109.       @boolean selfishFlag true if the thread is selfish, using
110.       a busy wait instead of calling sleep
111.    */
112.    public BallThread(Ball aBall, boolean selfishFlag)
113.    {
114.       b = aBall;
115.       selfish = selfishFlag;
116.    }
117.
118.    public void run()
119.    {
120.       try
121.       {
122.          for (int i = 1; i <= 1000; i++)
123.          {
124.             b.move();
125.             if (selfish)
126.             {
127.                // busy wait for 5 milliseconds
128.                long t = System.currentTimeMillis();
129.                while (System.currentTimeMillis() < t + 5)
130.                   ;
131.             }
132.             else
133.                sleep(5);
134.          }
135.       }
136.       catch (InterruptedException exception)
137.       {
138.       }
139.    }
140.
141.    private Ball b;
142.    boolean selfish;
143. }
```

```
144.
145. /**
146.    The canvas that draws the balls.
147. */
148. class BallCanvas extends JPanel
149. {
150.    /**
151.       Add a ball to the canvas.
152.       @param b the ball to add
153.    */
154.    public void add(Ball b)
155.    {
156.       balls.add(b);
157.    }
158.
159.    public void paintComponent(Graphics g)
160.    {
161.       super.paintComponent(g);
162.       Graphics2D g2 = (Graphics2D)g;
163.       for (int i = 0; i < balls.size(); i++)
164.       {
165.          Ball b = (Ball)balls.get(i);
166.          b.draw(g2);
167.       }
168.    }
169.
170.    private ArrayList balls = new ArrayList();
171. }
172.
173. /**
174.    A ball that moves and bounces off the edges of a
175.    component
176. */
177. class Ball
178. {
179.    /**
180.       Constructs a ball in the upper left corner
181.       @c the component in which the ball bounces
182.       @aColor the color of the ball
183.    */
184.    public Ball(Component c, Color aColor)
185.    {
186.       canvas = c;
187.       color = aColor;
188.    }
189.
190.    /**
191.       Draws the ball at its current position
192.       @param g2 the graphics context
```


```
193.    */
194.    public void draw(Graphics2D g2)
195.    {
196.       g2.setColor(color);
197.       g2.fill(new Ellipse2D.Double(x, y, XSIZE, YSIZE));
198.    }
199.
200.    /**
201.       Moves the ball to the next position, reversing direction
202.       if it hits one of the edges
203.    */
204.    public void move()
205.    {
206.       x += dx;
207.       y += dy;
208.       if (x < 0)
209.       {
210.          x = 0;
211.          dx = -dx;
212.       }
213.       if (x + XSIZE >= canvas.getWidth())
214.       {
215.          x = canvas.getWidth() - XSIZE;
216.          dx = -dx;
217.       }
218.       if (y < 0)
219.       {
220.          y = 0;
221.          dy = -dy;
222.       }
223.       if (y + YSIZE >= canvas.getHeight())
224.       {
225.          y = canvas.getHeight() - YSIZE;
226.          dy = -dy;
227.       }
228.
229.       canvas.repaint();
230.    }
231.
232.    private Component canvas;
233.    private Color color;
234.    private static final int XSIZE = 15;
235.    private static final int YSIZE = 15;
236.    private int x = 0;
237.    private int y = 0;
238.    private int dx = 2;
239.    private int dy = 2;
240. }
```

Synchronization

In most practical multithreaded applications, two or more threads need to share access to the same objects. What happens if two threads have access to the same object and each calls a method that modifies the state of the object? As you might imagine, the threads step on each other's toes. Depending on the order in which the data was accessed, corrupted objects can result. Such a situation is often called a *race condition*.

Thread Communication Without Synchronization

To avoid simultaneous access of a shared object by multiple threads, you must learn how to *synchronize* the access. In this section, you'll see what happens if you do not use synchronization. In the next section, you'll see how to synchronize object access.

In the next test program, we simulate a bank with 10 accounts. We randomly generate transactions that move money between these accounts. There are 10 threads, one for each account. Each transaction moves a random amount of money from the account serviced by the thread to another random account.

The simulation code is straightforward. We have the class `Bank` with the method `transfer`. This method transfers some amount of money from one account to another. If the source account does not have enough money in it, then the call simply returns. Here is the code for the `transfer` method of the `Bank` class.

```
public void transfer(int from, int to, double amount)
   // CAUTION: unsafe when called from multiple threads
{
   if (accounts[from] < amount) return;
   accounts[from] -= amount;
   accounts[to] += amount;
   ntransacts++;
   if (ntransacts % NTEST == 0) test();
}
```

Here is the code for the `TransferThread` class. Its `run` method keeps moving money out of a fixed bank account. In each iteration, the `run` method picks a random target account and a random amount, calls `transfer` on the bank object, and then sleeps.

```
class TransferThread extends Thread
{
   public TransferThread(Bank b, int from, int max)
   {
      bank = b;
      fromAccount = from;
      maxAmount = max;
```


```
      }

      public void run()
      {
         try
         {
            while (!interrupted())
            {
               int toAccount = (int)(bank.size() * Math.random());
               int amount = (int)(maxAmount * Math.random());
               bank.transfer(fromAccount, toAccount, amount);
               sleep(1);
            }
         }
         catch(InterruptedException e) {}
      }

      private Bank bank;
      private int fromAccount;
      private int maxAmount;
   }
```

When this simulation runs, we do not know how much money is in any one bank account at any time. But we do know that the total amount of money in all the accounts should remain unchanged since all we do is move money from one account to another.

Every 10,000 transactions, the `transfer` method calls a `test` method that recomputes the total and prints it out.

This program never finishes. Just press CTRL+C to kill the program.

Here is a typical printout:

```
Transactions:10000 Sum: 100000
Transactions:20000 Sum: 100000
Transactions:30000 Sum: 100000
Transactions:40000 Sum: 100000
Transactions:50000 Sum: 100000
Transactions:60000 Sum: 100000
Transactions:70000 Sum: 100000
Transactions:80000 Sum: 100000
Transactions:90000 Sum: 100000
Transactions:100000 Sum: 100000
Transactions:110000 Sum: 100000
Transactions:120000 Sum: 100000
Transactions:130000 Sum: 94792
Transactions:140000 Sum: 94792
Transactions:150000 Sum: 94792
. . .
```

As you can see, something is very wrong. For quite a few transactions, the bank balance remains at $100,000, which is the correct total for 10 accounts of $10,000 each. But after some time, the balance changes slightly. When you run this program, you may find that errors happen quickly, or it may take a very long time for the balance to become corrupted. This situation does not inspire confidence, and you would probably not want to deposit your hard-earned money into this bank.

Example 1–5 provides the complete source code. See if you can spot the problem with the code. We will unravel the mystery in the next section.

Example 1–5: UnsynchBankTest.java

```
1. public class UnsynchBankTest
2. {
3.    public static void main(String[] args)
4.    {
5.       Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
6.       int i;
7.       for (i = 0; i < NACCOUNTS; i++)
8.       {
9.          TransferThread t = new TransferThread(b, i,
10.            INITIAL_BALANCE);
11.         t.setPriority(Thread.NORM_PRIORITY + i % 2);
12.         t.start();
13.      }
14.   }
15.
16.   public static final int NACCOUNTS = 10;
17.   public static final int INITIAL_BALANCE = 10000;
18. }
19.
20. /**
21.    A bank with a number of bank accounts.
22. */
23. class Bank
24. {
25.    /**
26.       Constructs the bank.
27.       @param n the number of accounts
28.       @param initialBalance the initial balance
29.       for each account
30.    */
31.    public Bank(int n, int initialBalance)
32.    {
33.       accounts = new int[n];
34.       int i;
35.       for (i = 0; i < accounts.length; i++)
```


```
36.          accounts[i] = initialBalance;
37.       ntransacts = 0;
38.    }
39.
40.    /**
41.       Transfers money from one account to another.
42.       @param from the account to transfer from
43.       @param to the account to transfer to
44.       @param amount the amount to transfer
45.    */
46.    public void transfer(int from, int to, int amount)
47.       throws InterruptedException
48.    {
49.       accounts[from] -= amount;
50.       accounts[to] += amount;
51.       ntransacts++;
52.       if (ntransacts % NTEST == 0) test();
53.    }
54.
55.    /**
56.       Prints a test message to check the integrity
57.       of this bank object.
58.    */
59.    public void test()
60.    {
61.       int sum = 0;
62.
63.       for (int i = 0; i < accounts.length; i++)
64.          sum += accounts[i];
65.
66.       System.out.println("Transactions:" + ntransacts
67.          + " Sum: " + sum);
68.    }
69.
70.    /**
71.       Gets the number of accounts in the bank.
72.       @return the number of accounts
73.    */
74.    public int size()
75.    {
76.       return accounts.length;
77.    }
78.
79.    public static final int NTEST = 10000;
80.    private final int[] accounts;
81.    private long ntransacts = 0;
82. }
83.
```

```
/**
   A thread that transfers money from an account to other
   accounts in a bank.
*/
class TransferThread extends Thread
{
   /**
      Constructs a transfer thread.
      @param b the bank between whose account money is transferred
      @param from the account to transfer money from
      @param max the maximum amount of money in each transfer
   */
   public TransferThread(Bank b, int from, int max)
   {
      bank = b;
      fromAccount = from;
      maxAmount = max;
   }

   public void run()
   {
      try
      {
         while (!interrupted())
         {
            for (int i = 0; i < REPS; i++)
            {
               int toAccount = (int)(bank.size() * Math.random());
               int amount = (int)(maxAmount * Math.random() / REPS);
               bank.transfer(fromAccount, toAccount, amount);
               sleep(1);
            }
         }
      }
      catch(InterruptedException e) {}
   }

   private Bank bank;
   private int fromAccount;
   private int maxAmount;
   private static final int REPS = 1000;
}
```

Synchronizing Access to Shared Resources

In the previous section, we ran a program in which several threads updated bank account balances. After a while, errors crept in and some amount of

money was either lost or spontaneously created. This problem occurs when two threads are simultaneously trying to update an account. Suppose two threads simultaneously carry out the instruction:

```
accounts[to] += amount;
```

The problem is that these are not *atomic* operations. The instruction might be processed as follows:

1. Load `accounts[to]` into a register.
2. Add `amount`.
3. Move the result back to `accounts[to]`.

Now, suppose the first thread executes Steps 1 and 2, and then it is interrupted. Suppose the second thread awakens and updates the same entry in the `account` array. Then, the first thread awakens and completes its Step 3.

That action wipes out the modification of the other thread. As a result, the total is no longer correct. (See Figure 1–7.)

Our test program detects this corruption. (Of course, there is a slight chance of false alarms if the thread is interrupted as it is performing the tests!)

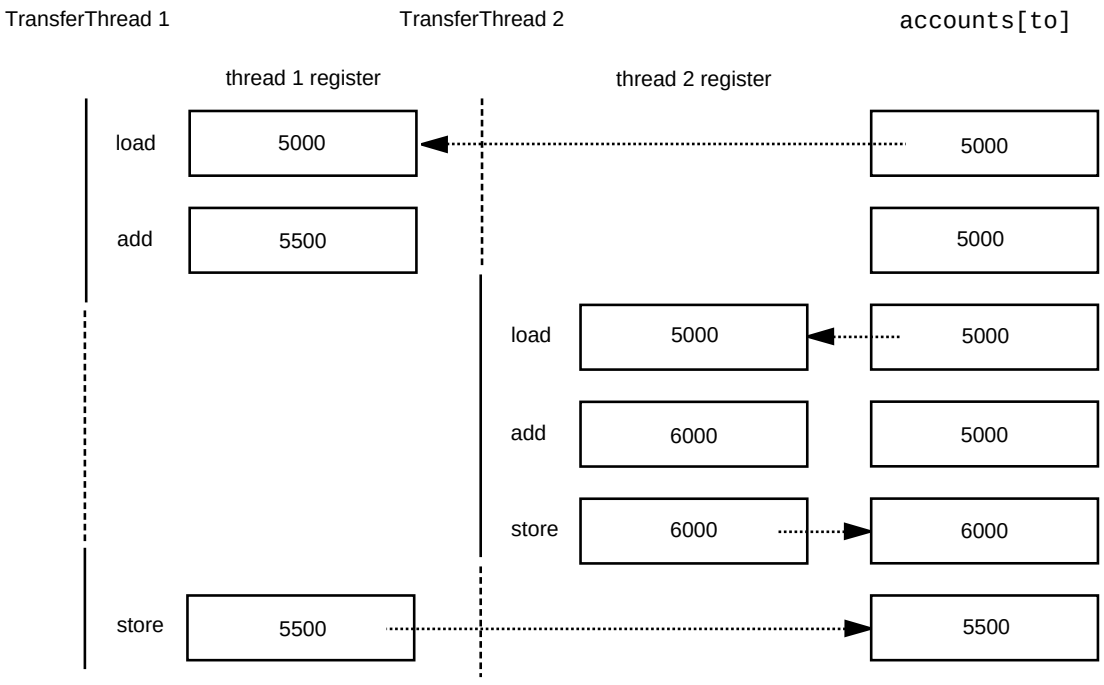

Figure 1–7: Simultaneous access by two threads

NOTE: You can actually peek at the virtual machine bytecodes that execute each statement in our class. Run the command

```
javap -c -v Bank
```

to decompile the `Bank.class` file. For example, the line

```
accounts[to] += amount;
```

is translated into the following bytecodes.

```
aload_0
getfield #16 <Field Bank.accounts [J>
iload_1
dup2
laload
iload_3
i2l
lsub
lastore
```

What these codes mean does not matter. The point is that the increment command is made up of several instructions, and the thread executing them can be interrupted at the point of any instruction.

What is the chance of this corruption occurring? It is quite low, because each thread does so little work before going to sleep again that it is unlikely that the scheduler will preempt it. We found by experimentation that we could boost the probability of corruption by various measures, depending on the target platform. On Windows 98, it helps to assign half of the transfer threads a higher priority than the other half.

```
for (i = 0; i < NACCOUNTS; i++)
{
   TransferThread t = new TransferThread(b, i,
      INITIAL_BALANCE);
   t.setPriority(Thread.NORM_PRIORITY + i % 2);
   t.start();
}
```

When a higher-priority transfer thread wakes up from its sleep, it will preempt a lower-priority transfer thread.

NOTE: This is exactly the kind of tinkering with priority levels that we tell you not to do in your own programs. We were in a bind—we wanted to show you a program that can demonstrate data corruption. In your own programs, you presumably will not want to increase the chance of corruption, so you should not imitate this approach.

On Linux, thread priorities are ignored, and instead, a slight change in the `run` method did the trick—to repeat the transfer multiple times before sleeping.

```
final int REPS = 1000;
for (int i = 0; i < REPS; i++)
{
   int toAccount = (int)(bank.size() * Math.random());
   int amount = (int)(maxAmount * Math.random() / REPS);
   bank.transfer(fromAccount, toAccount, amount);
   sleep(1);
}
```

On all platforms, it helps if you load your machine heavily, by running a few bloatware programs in parallel with the test.

The real problem is that the work of the `transfer` method can be interrupted in the middle. If we could ensure that the method runs to completion before the thread loses control, then the state of the bank account object would not be corrupted.

Many thread libraries force the programmer to fuss with so-called semaphores and critical sections to gain uninterrupted access to a resource. This is sufficient for procedural programming, but it is hardly object-oriented. The Java programming language has a nicer mechanism, inspired by the *monitors* invented by Tony Hoare.

You simply tag any operation that should not be interrupted as `synchronized`, like this:

```
public synchronized void transfer(int from, int to,
   int amount)
{
   if (accounts[from] < amount) return;
   accounts[from] -= amount;
   accounts[to] += amount;
   ntransacts++;
   if (ntransacts % NTEST == 0) test();
}
```

When one thread calls a synchronized method, it is guaranteed that the method will finish before another thread can execute any synchronized method on the same object (see Figure 1–8). When one thread calls `transfer` and then another thread also calls `transfer`, the second thread cannot continue. Instead, it is deactivated and must wait for the first thread to finish executing the `transfer` method.

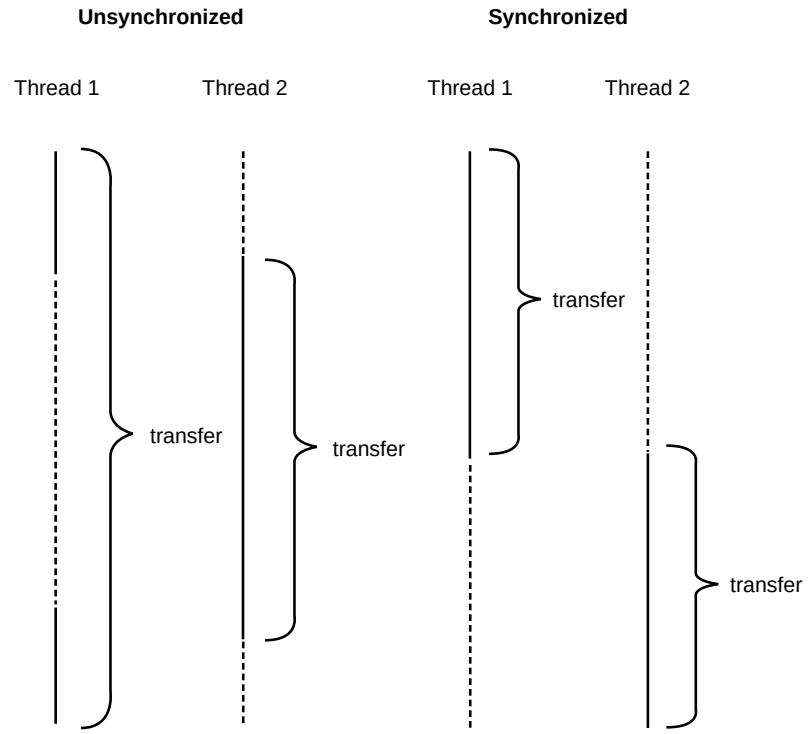

Figure 1–8: Comparison of unsynchronized and synchronized threads

Try it out. Tag the `transfer` method as `synchronized` and run the program again. You can run it forever, and the bank balance will not get corrupted.

In general, you will want to tag those methods as `synchronized` that require multiple operations to update a data structure, as well as those that retrieve a value from a data structure. You are then assured that these operations run to completion before another thread can use the same object.

Of course, the synchronization mechanism isn't free. As you'll see in the next section, some bookkeeping code is executed every time a synchronized method is called. Thus, you will not want to synchronize every method of every class. If objects are not shared among threads, then there is no need to use synchronization. If a method always returns the same value for a given object, then you don't need to synchronize that method. For example, the `size` method of our `Bank` class need not be synchronized—the size of a bank object is fixed after the object is constructed.

Because synchronization is too expensive to use for every class, it is usually a good idea to custom-build classes for thread communication. For example, suppose a browser loads multiple images in parallel and wants to know when

all images are loaded. You can define a `ProgressTracker` class with synchronized methods to update and query the loading progress.

Some programmers who have experience with other threading models complain about Java synchronization, finding it cumbersome and inefficient. For system level programming, these may be valid complaints. But for application programmers, the Java model works quite nicely. Just remember to use supporting classes for thread communication—don't try to hack existing code by sprinkling a few `synchronized` keywords over it.

NOTE: In some cases, programmers try to avoid the cost of synchronization in code that performs simple independent load or store operations. However, that can be dangerous, for two reasons. First, a load or store of a 64-bit value is *not* guaranteed to be atomic. That is, if you make an assignment to a `double` or `long` field, half of the assignment could happen, then the thread might be preempted. The next thread then sees the field in an inconsistent state. Moreover, in a multiprocessor machine, each processor can work on a separate cache of data from the main memory. The `synchronized` keyword ensures that local caches are made consistent with the main memory, but unsynchronized methods have no such guarantee. It can then happen that one thread doesn't see a modification to a shared variable made by another thread.

The `volatile` keyword is designed to address these situations. Loads and stores of a 64-bit variable that is declared as `volatile` are guaranteed to be atomic. In a multiprocessor machine, loads and stores of volatile variables should work correctly even for data in processor caches.

In some situations, it might be possible to avoid synchronization and use volatile variables instead. However, not only is this issue fraught with complexity, there also have been reports of virtual machine implementations that don't handle volatile variables correctly. Our recommendation is to use synchronization, not volatile variables, to guarantee thread safety.

Object Locks

When a thread calls a synchronized method, the *object* becomes "locked." Think of each object as having a door with a lock on the inside. It is quite common among Java programmers to visualize object locks by using a "rest room" analogy. The object corresponds to a rest room stall that can hold only one person at a time. In the interest of good taste, we will use a "telephone booth" analogy instead. Imagine a traditional, enclosed telephone booth and suppose it has a latch on the inside. When a thread enters the synchronized method, it closes the door and locks it. When another thread tries to call a synchronized method on the same object, it can't open the door, so it stops running. Eventually, the first thread exits its synchronized method and unlocks the door.

Periodically, the thread scheduler activates the threads that are waiting for the lock to open, using its normal activation rules that we already discussed. Whenever one of the threads that wants to call a synchronized method on the object runs again, it checks to see if the object is still locked. If the object is now unlocked, the thread gets to be the next one to gain exclusive access to the object.

However, other threads are still free to call unsynchronized methods on a locked object. For example, the `size` method of the `Bank` class is not synchronized and it can be called on a locked object.

When a thread leaves a synchronized method by throwing an exception, it still relinquishes the object lock. That is a good thing—you wouldn't want a thread to hog the object after it has exited the synchronized method.

If a thread owns the lock of an object and it calls another synchronized method of the same object, then that method is automatically granted access. The thread only relinquishes the lock when it exits the last synchronized method.

NOTE: Technically, each object has a *lock count* that counts how many synchronized methods the lock owner has called. Each time a new synchronized method is called, the lock count is increased. Each time a synchronized method terminates (either because of a normal return or because of an uncaught exception), the lock count is decremented. When the lock count reaches zero, the thread gives up the lock.

Note that you can have two different objects of the same class, each of which is locked by a different thread. These threads may even execute the same synchronized method. It's the object that's locked, not the method. In the telephone booth analogy, you can have two people in two separate booths. Each of them may execute the same synchronized method, or they may execute different methods—it doesn't matter.

Of course, an object's lock can only be owned by one thread at any given point in time. But a thread can own the locks of multiple objects at the same time, simply by calling a synchronized method on an object while executing a synchronized method of another object. (Here, admittedly, the telephone booth analogy breaks down.)

The `wait` *and* `notify` *methods*

Let us refine our simulation of the bank. We do not want to transfer money out of an account that does not have the funds to cover the transfer. Note that we cannot use code like:

```
if (bank.getBalance(from) >= amount)
   bank.transfer(from, to, amount);
```

It is entirely possible that the current thread will be deactivated between the successful outcome of the test and the call to `transfer`.

```
if (bank.getBalance(from) >= amount)
      // thread might be deactivated at this point
   bank.transfer(from, to, amount);
```

By the time the thread is running again, the account balance may have fallen below the withdrawal amount. You must make sure that the thread cannot be interrupted between the test and the insertion. You do so by putting both the test and the transfer action inside the same synchronized method:

```
public synchronized void transfer(int from, int to,
   int amount)
{
   while (accounts[from] < amount)
   {
      // wait
      . . .
   }
   // transfer funds
   . . .
}
```

Now, what do we do when there is not enough money in the account? We wait until some other thread has added funds. But the `transfer` method is `synchronized`. This thread has just gained exclusive access to the bank object, so no other thread has a chance to make a deposit. A second feature of synchronized methods takes care of this situation. You use the `wait` method of the `Object` class if you need to wait inside a synchronized method.

When `wait` is called inside a synchronized method, the current thread is blocked and gives up the object lock. This lets in another thread that can, we hope, increase the account balance.

The `wait` method can throw an `InterruptedException` when the thread is interrupted while it is waiting. In that case, you can either turn on the "interrupted" flag or propagate the exception—after all, the calling thread, not the bank object, should decide what to do with an interruption. In our case, we simply propagate the exception and add a `throws` specifier to the `transfer` method.

```
public synchronized void transfer(int from, int to,
   int amount) throws InterruptedException
{
   while (accounts[from] < amount)
      wait();
   // transfer funds
   . . .
}
```

Note that the `wait` method is a method of the class `Object`, not of the class `Thread`. When calling `wait`, the bank object unlocks itself and blocks the current thread. (Here, the `wait` method was called on the `this` reference.)

CAUTION: If a thread holds the locks to multiple objects, then the call to `wait` unlocks *only* the object on which `wait` was called. That means that the blocked thread can hold locks to other objects, which won't get unlocked until the thread is unblocked again. That's a dangerous situation that you should avoid.

There is an essential difference between a thread that is waiting to get inside a synchronized method and a thread that has called `wait`. Once a thread calls the `wait` method, it enters a *wait list* for that object. The thread is now blocked. Until the thread is removed from the wait list, the scheduler ignores it and it does not have a chance to continue running.

To remove the thread from the wait list, some other thread must call the `notifyAll` or `notify` method on the *same object*. The `notifyAll` method removes all threads from the object's wait list. The `notify` method removes just one arbitrarily chosen thread. When the threads are removed from the wait list, then they are again runnable, and the scheduler will eventually activate them again. At that time, they will attempt to reenter the object. As soon as the object lock is available, one of them will lock the object and continue where it left off after the call to `wait`.

To understand the `wait` and `notifyAll`/`notify` calls, let's trot out the telephone booth analogy once again. Suppose a thread locks an object and then finds it can't proceed, say because the phone is broken. First of all, it would be pointless for the thread to fall asleep while locked inside the booth. Then a maintenance engineerwould be prevented from entering and fixing the equipment. By calling `wait`, the thread unlocks the object and waits outside. Eventually, a maintenance engineer enters the booth, locks it from inside, and does something. After the engineer leaves the booth, the waiting threads won't know that the situation has improved—maybe the engineer just emptied the coin reservoir. The waiting threads just keep waiting, still without hope. By calling `notifyAll`, the engineer tells all waiting threads that the object state may have changed to their advantage. By calling `notify`, the engineer picks one of the waiting threads at random and only tells that one, leaving the others in their despondent waiting state.

It is crucially important that *some* other thread calls the `notify` or `notifyAll` method periodically. When a thread calls `wait`, it has no way of unblocking itself. It puts its faith in the other threads. If none of them bother to unblock the waiting thread, it will never run again. This can lead to unpleasant *deadlock* situations. If all other threads are blocked and the last active thread calls `wait` without unblocking one of the others, then it also blocks. There is no thread left to unblock the others, and the program hangs. The waiting threads are *not* automatically

reactivated when no other thread is working on the object. We will discuss deadlocks later in this chapter.

As a practical matter, it is dangerous to call `notify` because you have no control over which thread gets unblocked. If the wrong thread gets unblocked, that thread may not be able to proceed. We simply recommend that you use the `notifyAll` method and that all threads be unblocked.

When should you call `notifyAll`? The rule of thumb is to call `notifyAll` whenever the state of an object changes in a way that might be advantageous to waiting threads. For example, whenever an account balance changes, the waiting threads should be given another chance to inspect the balance. In our example, we will call `notifyAll` when we have finished with the funds transfer.

```
public synchronized void transfer(int from, int to,
   int amount)
{
   . . .
   accounts[from] -= amount;
   accounts[to] += amount;
   ntransacts++;
   notifyAll();
   . . .
}
```

This notification gives the waiting threads the chance to run again. A thread that was waiting for a higher balance then gets a chance to check the balance again. If the balance is sufficient, the thread performs the transfer. If not, it calls `wait` again.

Note that the call to `notifyAll` does not immediately activate a waiting thread. It only unblocks the waiting threads so that they can compete for entry into the object after the current thread has exited the synchronized method.

TIP: If your multithreaded program gets stuck, double-check that every `wait` is matched by a `notifyAll`.

If you run the sample program with the synchronized version of the `transfer` method, you will notice that nothing ever goes wrong. The total balance stays at $100,000 forever. (Again, you need to press CTRL+C to terminate the program.)

You will also notice that the program in Example 1–6 runs a bit slower—this is the price you pay for the added bookkeeping involved in the synchronization mechanism.

Example 1–6: SynchBankTest.java

```
1. public class SynchBankTest
2. {
3.    public static void main(String[] args)
```

```
4.    {
5.       Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
6.       int i;
7.       for (i = 0; i < NACCOUNTS; i++)
8.       {
9.          TransferThread t = new TransferThread(b, i,
10.            INITIAL_BALANCE);
11.         t.setPriority(Thread.NORM_PRIORITY + i % 2);
12.         t.start();
13.      }
14.   }
15.
16.   public static final int NACCOUNTS = 10;
17.   public static final int INITIAL_BALANCE = 10000;
18. }
19.
20. /**
21.    A bank with a number of bank accounts.
22. */
23. class Bank
24. {
25.    /**
26.       Constructs the bank.
27.       @param n the number of accounts
28.       @param initialBalance the initial balance
29.       for each account
30.    */
31.    public Bank(int n, int initialBalance)
32.    {
33.       accounts = new int[n];
34.       int i;
35.       for (i = 0; i < accounts.length; i++)
36.          accounts[i] = initialBalance;
37.       ntransacts = 0;
38.    }
39.
40.    /**
41.       Transfers money from one account to another.
42.       @param from the account to transfer from
43.       @param to the account to transfer to
44.       @param amount the amount to transfer
45.    */
46.    public synchronized void transfer(int from, int to, int amount)
47.       throws InterruptedException
48.    {
49.       while (accounts[from] < amount)
50.          wait();
51.       accounts[from] -= amount;
52.       accounts[to] += amount;
53.       ntransacts++;
54.       notifyAll();
55.       if (ntransacts % NTEST == 0) test();
```


```
56.    }
57.
58.    /**
59.       Prints a test message to check the integrity
60.       of this bank object.
61.    */
62.    public synchronized void test()
63.    {
64.       int sum = 0;
65.
66.       for (int i = 0; i < accounts.length; i++)
67.          sum += accounts[i];
68.
69.       System.out.println("Transactions:" + ntransacts
70.          + " Sum: " + sum);
71.    }
72.
73.    /**
74.       Gets the number of accounts in the bank.
75.       @return the number of accounts
76.    */
77.    public int size()
78.    {
79.       return accounts.length;
80.    }
81.
82.    public static final int NTEST = 10000;
83.    private final int[] accounts;
84.    private long ntransacts = 0;
85. }
86.
87. /**
88.    A thread that transfers money from an account to other
89.    accounts in a bank.
90. */
91. class TransferThread extends Thread
92. {
93.    /**
94.       Constructs a transfer thread.
95.       @param b the bank between whose account money is transferred
96.       @param from the account to transfer money from
97.       @param max the maximum amount of money in each transfer
98.    */
99.    public TransferThread(Bank b, int from, int max)
100.   {
101.      bank = b;
102.      fromAccount = from;
103.      maxAmount = max;
104.   }
105.
106.   public void run()
107.   {
```

```
108.       try
109.       {
110.          while (!interrupted())
111.          {
112.             int toAccount = (int)(bank.size() * Math.random());
113.             int amount = (int)(maxAmount * Math.random());
114.             bank.transfer(fromAccount, toAccount, amount);
115.             sleep(1);
116.          }
117.       }
118.       catch(InterruptedException e) {}
119.    }
120.
121.    private Bank bank;
122.    private int fromAccount;
123.    private int maxAmount;
124. }
```

Here is a summary of how the synchronization mechanism works.

1. To call a synchronized method, the implicit parameter must not be locked. Calling the method locks the object. Returning from the call unlocks the implicit parameter object. Thus, only one thread at a time can execute synchronized methods on a particular object.
2. When a thread executes a call to `wait`, it surrenders the object lock and enters a wait list for that object.
3. To remove a thread from the wait list, some other thread must make a call to `notifyAll` or `notify`, on the same object.

The scheduling rules are undeniably complex, but it is actually quite simple to put them into practice. Just follow these five rules:

1. If two or more threads modify an object, declare the methods that carry out the modifications as synchronized. Read-only methods that are affected by object modifications must also be synchronized.
2. If a thread must wait for the state of an object to change, it should wait inside the object, not outside, by entering a synchronized method and calling `wait`.
3. Don’t spend any significant amount of time in a synchronized method. Most operations simply update a data structure and quickly return. If you can’t complete a synchronized method immediately, call `wait` so that you give up the object lock while waiting.
4. Whenever a method changes the state of an object, it should call `notifyAll`. That gives the waiting threads a chance to see if circumstances have changed.

5. Remember that `wait` and `notifyAll/notify` are methods of the `Object` class, not the `Thread` class. Double-check that your calls to `wait` are matched up by a notification *on the same object.*

Synchronized Blocks

Occasionally, it is useful to lock an object and obtain exclusive access to it for just a few instructions without writing a new synchronized method. You use *synchronized blocks* to achieve this access. A synchronized block consists of a sequence of statements, enclosed in `{ . . . }` and prefixed with `synchronized (obj)`, where *`obj`* is the object to be locked. Here is an example of the syntax:

```
public void run()
{
   . . .
   synchronized (bank) // lock the bank object
   {
      if (bank.getBalance(from) >= amount)
         bank.transfer(from, to, amount);
   }
   . . .
}
```

In this sample code segment, the synchronized block will run to completion before any other thread can call a synchronized method on the `bank` object.

Application programmers tend to avoid this feature. It is usually a better idea to take a step back, think about the mechanism on a higher level, come up with a class that describes it, and use synchronized methods of that class. System programmers who consider additional classes "high overhead" are more likely to use synchronized blocks.

Synchronized Static Methods

A *singleton* is a class with just one object. Singletons are commonly used for management objects that need to be globally unique, such as print spoolers, database connection managers, and so on.

Consider the typical implementation of a singleton.

```
public class Singleton
{
   public static Singleton getInstance()
   {
      if (instance == null)
         instance = new Singleton(. . .);
      return instance;
   }

   private Singleton(. . .) { . . . }
   . . .
```

```
   private static Singleton instance;
}
```

However, the `getInstance` method is not threadsafe. Suppose one thread calls `getInstance` and is preempted in the middle of the constructor, before the `instance` field has been set. Then another thread gains control and it calls `getInstance`. Since `instance` is still `null`, that thread constructs a second object. That's just what a singleton is supposed to prevent.

The remedy is simple: make the `getInstance` method synchronized:

```
public static synchronized Singleton getInstance()
{
   if (instance == null)
      instance = new Singleton(. . .);
   return instance;
}
```

Now, the method runs to completion before another thread can call it.

If you paid close attention to the preceding sections, you may wonder how this works. When a thread calls a synchronized method, it acquires the lock of an object. But this method is static—what object does a thread lock when calling `Singleton.getInstance()`?

Calling a static method locks the *class object* `Singleton.class`. (Recall from Volume 1, Chapter 5, that there is a unique object of type `Class` that describes each class that the virtual machine has loaded.) Therefore, if one thread calls a synchronized static method of a class, all synchronized static methods of the class are blocked until the first call returns.

`java.lang.Object`

- `void notifyAll()`
  unblocks the threads that called `wait` on this object. This method can only be called from within a synchronized method or block. The method throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.
- `void notify()`
  unblocks one randomly selected thread among the threads that called `wait` on this object. This method can only be called from within a synchronized method or block. The method throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.
- `void wait()`
  causes a thread to wait until it is notified. This method can only be called from within a synchronized method. It throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.

Deadlocks

The synchronization feature in the Java programming language is convenient and powerful, but it cannot solve all problems that might arise in multithreading. Consider the following situation:

Account 1: $2,000

Account 2: $3,000

Thread 1: Transfer $3,000 from Account 1 to Account 2

Thread 2: Transfer $4,000 from Account 2 to Account 1

As Figure 1–9 indicates, Threads 1 and 2 are clearly blocked. Neither can proceed because the balances in Accounts 1 and 2 are insufficient.

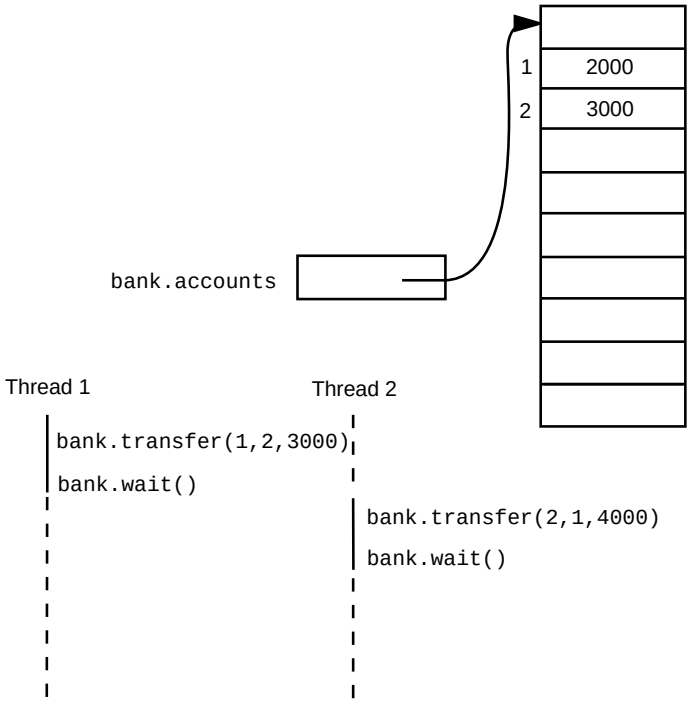

Figure 1–9: A deadlock situation

Is it possible that all 10 threads are blocked because each is waiting for more money? Such a situation is called a *deadlock*.

In our program, a deadlock cannot occur for a simple reason. Each transfer amount is for, at most, $10,000. Since there are 10 accounts and a total of $100,000 in them, at least one of the accounts must have more than $10,000 at any time. The thread moving money out of that account can therefore proceed.

But if you change the `run` method of the threads to remove the $10,000 transaction limit, deadlocks can occur quickly. Try it out. Construct each `TransferThread` with a `maxAmount` of 14000 and run the program. The program will run for a while and then hang.

Another way to create a deadlock is to make the `i`'th thread responsible for putting money into the `i`'th account, rather than for taking it out of the `i`'th account. In this case, there is a chance that all threads will gang up on one account, each trying to remove more money from it than it contains. Try it out. In the `SynchBankTest` program, turn to the `run` method of the `TransferThread` class. In the call to `transfer`, flip `fromAccount` and `toAccount`. Run the program and see how it deadlocks almost immediately.

Here is another situation in which a deadlock can occur easily: Change the `notifyAll` method to `notify` in the `SynchBankTest` program. You will find that the program hangs quickly. Unlike `notifyAll`, which notifies all threads that are waiting for added funds, the `notify` method unblocks only one thread. If that thread can't proceed, all threads can be blocked. Consider the following sample scenario of a developing deadlock.

Account 1: $19,000

All other accounts: $9,000 each

Thread 1: Transfer $9,500 from Account 1 to Account 2

All other threads: Transfer $9,100 from their account to another account

Clearly, all threads but Thread 1 are blocked, because there isn't enough money in their accounts.

Thread 1 proceeds. Afterward, we have the following situation:

Account 1: $9,500

Account 2: $18,500

All other accounts: $9,000 each

Then, Thread 1 calls `notify`. The `notify` method picks a thread at random to unblock. Suppose it picks Thread 3. That thread is awakened, finds that there isn't enough money in its account, and calls `wait` again. But Thread 1 is still running. A new random transaction is generated, say,

Thread 1: Transfer $9,600 to from Account 1 to Account 2

Now, Thread 1 also calls `wait`, and *all* threads are blocked. The system has deadlocked.

The culprit here is the call to `notify`. It only unblocks one thread, and it may not pick the thread that is essential to make progress. (In our scenario, Thread 2 must proceed to take money out of Account 2.) In contrast, `notifyAll` unblocks all threads.

Unfortunately, there is nothing in the Java programming language to avoid or break these deadlocks. You must design your threads to ensure that a deadlock situation cannot occur. You need to analyze your program and ensure that every blocked thread will eventually be notified, and that at least one of them can always proceed.

CAUTION: You should avoid *blocking* calls inside a synchronized method, for example a call to an I/O operation. If the thread blocks while holding the object lock, every other thread calling a synchronized method on the same object also blocks. If eventually all other threads call a synchronized method on that object, then all threads are blocked and deadlock results. This is called a "black hole." (Think of someone staying in a phone booth for a very long time, and everyone else waiting outside instead of doing useful work.)

Some programmers find Java thread synchronization overly deadlock-prone because they are accustomed to different mechanisms that don't translate well to Java. Simply trying to turn semaphores into a nested mess of synchronized blocks can indeed be a recipe for disaster. Our advice, if you get stuck with a deadlock problem, is to step back and ask yourself what communication pattern between threads you want to achieve. Then create another class for that purpose. That's the Object-Oriented way, and it often helps disentangle the logic of multithreaded programs.

Why the `stop` *and* `suspend` *Methods Are Deprecated*

The Java 1.0 platform defined a `stop` method that simply terminates a thread, and a `suspend` method that blocks a thread until another thread calls `resume`. Both of these methods have been deprecated in the Java 2 platform. The `stop` method is inherently unsafe, and experience has shown that the `suspend` method frequently leads to deadlocks. In this section, you will see why these methods are problematic and what you can do to avoid problems.

Let us turn to the `stop` method first. When a thread is stopped, it immediately gives up the locks on all objects that it has locked. This can leave objects in an inconsistent state. For example, suppose a `TransferThread` is stopped in the middle of moving money from one account to another, after the withdrawal and before the deposit. Now the bank object is *damaged*. That damage is observable from the other threads that have not been stopped.

CAUTION: Technically speaking, the `stop` method causes the thread to be stopped to throw an exception object of type `ThreadDeath`. This exception terminates all pending methods, including the `run` method.

For the same reason, *any* uncaught exception in a synchronized method can cause that method to terminate prematurely and lead to damaged objects.

When a thread wants to stop another thread, it has no way of knowing when the `stop` method is safe and when it leads to damaged objects. Therefore, the method has been deprecated.

NOTE: Some authors claim that the `stop` method has been deprecated because it can cause objects to be permanently locked by a stopped thread. However, that is not true. A stopped thread exits all synchronized methods it has called (through the processing of the `ThreadDeath` exception). As a consequence, the thread relinquishes the object locks that it holds.

If you need to stop a thread safely, you can have the thread periodically check a variable that indicates whether a stop has been requested.

```
public class MyThread extends Thread
{
   public void run()
   {
      while (!stopRequested && more work to do)
      {
          do more work
      }
   }

   public void requestStop()
   {
      stopRequested = true;
   }

   private boolean stopRequested;
}
```

This code leaves the `run` method to control when to finish, and it is up to the `run` method to ensure that no objects are left in a damaged state.

Testing the `stopRequested` variable in the main loop of a thread work is fine, except if the thread is currently blocked. In that case, the thread will only terminate after it is unblocked. You can force a thread out of a blocked state by interrupting it. Thus, you should define the `requestStop` method to call `interrupt`:

```
public void requestStop()
{
   stopRequested = true;
   interrupt();
}
```

You can test the `stopRequested` variable in the `catch` clause for the `InterruptedException`. For example,

```
try
{
   wait();
}
```


```
catch (InterruptedException e)
{
   if (stopRequested)
      return; // exit the run method
}
```

Actually, many programmers take the attitude that the only reason to interrupt a thread is to stop it. Then, you don't need to test the `stopRequested` variable—simply exit the `run` method whenever you sense an interrupt.

By the way, the `interrupt` method does *not* generate an `InterruptedException` when a thread is interrupted while it is working. Instead, it simply sets the "interrupted" flag. That way, interrupting a thread cannot corrupt object data. It is up to the thread to check the "interrupted" flag after all critical calculations have been completed.

Next, let us see what is wrong with the `suspend` method. Unlike `stop`, `suspend` won't damage objects. However, if you suspend a thread that owns a lock to an object, then the object is unavailable until the thread is resumed. If the thread that calls the `suspend` method tries to acquire the lock for the same object before calling `resume`, then the program deadlocks: the suspended thread waits to be resumed, and the suspending thread waits for the object to be unlocked.

This situation occurs frequently in graphical user interfaces. Suppose we have a graphical simulation of our bank. We have a button labeled "Pause" that suspends the transfer threads, and a button labeled "Resume" that resumes them.

```
pauseButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         for (int i = 0; i < threads.length; i++)
            threads[i].suspend(); // Don't do this
      }
   });
resumeButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         for (int i = 0; i < threads.length; i++)
            threads[i].resume(); // Don't do this
      }
   });
```

Suppose a `paintComponent` method paints a chart of each account, calling the `bank.getBalance` method, and that method is synchronized.

As you will see in the next section, both the button actions and the repainting occur in the same thread, the *event dispatch thread.*

Now consider the following scenario:

1. One of the transfer threads acquires the lock on the `bank` object.
2. The user clicks the "Pause" button.
3. All transfer threads are suspended; one of them still holds the lock on the `bank` object.
4. For some reason, the account chart needs to be repainted.
5. The `paintComponent` method calls the synchronized method `bank.getBalance`.

Now the program is frozen.

The event dispatch thread can't proceed because the `bank` object is locked by one of the suspended threads. Thus, the user can't click the "Resume" button, and the threads won't ever resume.

If you want to safely suspend the thread, you should introduce a variable `suspendRequested` and test it in a safe place of your `run` method—somewhere, where your thread doesn't lock objects that other threads need. When your thread finds that the `suspendRequested` variable has been set, keep waiting until it becomes available again.

For greater clarity, wrap the variable in a class `SuspendRequestor`, like this:

```
class SuspendRequestor
{
   public synchronized void set(boolean b)
   {
      suspendRequested = b;
      notifyAll();
   }

   public synchronized void waitForResume()
      throws InterruptedException
   {
      while (suspendRequested)
         wait();
   }

   private boolean suspendRequested;
}

class MyThread extends Thread
{
   public void requestSuspend()
   {
      suspender.set(true);
   }
```


```
   public void requestResume()
   {
      suspender.set(false);
   }

   public void run()
   {
      try
      {
         while (more work to do)
         {
            suspender.waitForResume();
            do more work
         }
      }
      catch (InterruptedException exception)
      {
      }
   }

   private SuspendRequestor suspender
      = new SuspendRequestor();
}
```

Now the call to `suspender.waitForResume()` blocks when suspension has been requested. To unblock, some other thread has to request resumption.

NOTE: Some programmers don't want to come up with a new class for such a simple mechanism. But they still need some object on which to synchronize, because the waiting thread needs to be added to the wait list of some object. It is possible to use a separate dummy variable, like this:

```
class MyThread extends Thread
{

   public void requestSuspend()
   {
      suspendRequested = true;
   }
   public void requestResume()
   {
      suspendRequested = false;
      synchronized (dummy)
      {
         dummy.notifyAll();
         // unblock the thread waiting on dummy
      }
   }

   private void waitForResume()
      throws InterruptedException
   {
```

```
         synchronized (dummy)
           // synchronized necessary for calling wait
         {
            while (suspendRequested)
               dummy.wait(); // block this thread
         }
      }

      . . .
      private boolean suspendRequested;
      private Integer dummy = new Integer(1);
         // any non-null object will work
   }
```

We don't like this coding style. It is just as easy to supply an additional class. Then the lock is naturally associated with that class, and you avoid the confusion that arises when you hijack an object just for its lock and wait list.

Of course, avoiding the `Thread.suspend` method does not automatically avoid deadlocks. If a thread calls `wait`, it might not wake up either. But there is an essential difference. A thread can control when it calls `wait`. But the `Thread.suspend` method can be invoked *externally* on a thread, at any time, without the thread's consent. The same is true for the `Thread.stop` method. For that reason, these two methods have been deprecated.

NOTE: In this section, we defined methods `requestStop`, `requestSuspend`, and `requestResume`. These methods provide functionality that is similar to the deprecated `stop`, `suspend`, and `resume` methods, while avoiding the risks of those deprecated methods. You will find that many programmers implement similar methods, but instead of giving them different names, they simply *override* the `stop`, `suspend`, and `resume` methods. It is entirely legal to override a deprecated method with another method that is not deprecated. If you see a call to `stop`, `suspend`, or `resume` in a program, you should not automatically assume that the program is wrong. First check whether the programmer overrode the deprecated methods with safer versions.

Timeouts

When you make a blocking call, such as a call to the `wait` method or to I/O, your thread loses control and is at the mercy of another thread or, in the case of I/O, external circumstances. You can limit that risk by using timeouts.

There are two `wait` methods with timeout parameters:

```
void wait(long millis)
void wait(long millis, int nanos)
```

that wait until the thread is awakened by a call to `notifyAll`/`notify` or for the given number of milliseconds, or milliseconds and nanoseconds—there are 1,000,000 nanoseconds in a millisecond.

However, when the `wait` method returns, you don't know whether the cause was a timeout or a notification. You probably want to know—there is no point in reevaluating the condition for which you were waiting if there was no notification.

You can compute the difference of the system time before and after the call:

```
long before = System.currentTimeMillis();
wait(delay);
long after = System.currentTimeMills();
if (after - before > delay)
   . . . // timeout
```

Alternatively, you can make the notifying thread set a flag.

A thread can also block indefinitely when calling an I/O operation that doesn't have a timeout. For example, as you will see in Chapter 3, to open a network socket, you need to call the socket constructor, and it doesn't have a provision for a timeout. You can always force a timeout by putting the blocking operation into a second thread, and then use the `join` method. The call

```
t.join(millis);
```

blocks the current thread until the thread `t` has completed or the given number of milliseconds has elapsed, whichever occurs first. Use this outline:

```
Thread t = new
   Thread()
   {
      public void run()
      {
         blocking operation
      }
   };
t.start();
t.join(millis);
```

The blocking operation either succeeds within the given number of milliseconds, or the `join` method returns control to the current thread. Of course, then the thread `t` is still alive. What to do about that depends on the nature of the blocking operation. If you know that the operation can be interrupted, you can call `t.interrupt()` to stop it. In the case of an I/O operation, you may know that there is an operating-system-dependent timeout. In that case, just leave `t` alive until the timeout occurs and the blocking operation returns. See the `SocketOpener` in Chapter 3 for a typical example.

`java.lang.Thread`

- `void wait(long millis)`
- `void wait(long millis, int nanos)`
  causes a thread to wait until it is notified or until the specified amount of time has passed. This method can only be called from within a synchronized

method. It throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.

Parameters:	`millis`	the number of milliseconds
	`nanos`	the number of nanoseconds, < 1,000,000

- `void join()`
  waits for the specified thread to cease to be alive.
- `void join(long millis)`
  waits for the specified thread to cease to be alive or for the specified number of milliseconds to pass.

Parameters:	`millis`	the number of milliseconds

User Interface Programming with Threads

In the following sections, we discuss threading issues that are of particular interest to user interface programming.

Threads and Swing

As we mentioned in the introduction, one of the reasons to use threads in your programs is to make your programs more responsive. When your program needs to do something time-consuming, then you should fire up another worker thread instead of blocking the user interface.

However, you have to be careful what you do in a worker thread because, perhaps surprisingly, Swing is *not thread safe.* That is, the majority of methods of Swing classes are not synchronized. If you try to manipulate user interface elements from multiple threads, then your user interface will become corrupted.

For example, run the test program whose code you will find at the end of this section. When you click on the "Bad" button, a new thread is started that edits a combo box, randomly adding and removing values.

```
class BadWorkerThread extends Thread
{
   public BadWorkerThread(JComboBox aCombo)
   {
      combo = aCombo;
      generator = new Random();
   }

   public void run()
   {
      try
      {
         while (!interrupted())
         {
```


```
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0)
               combo.insertItemAt(new Integer(i), 0);
            else if (combo.getItemCount() > 0)
               combo.removeItemAt(i % combo.getItemCount());

            sleep(1);
         }
      }
      catch (InterruptedException exception) {}
   }

   private JComboBox combo;
   private Random generator;
}
```

Try it out. Click on the "Bad" button. If you start the program from a console window, you will see an occasional exception report in the console (see Figure 1–10).

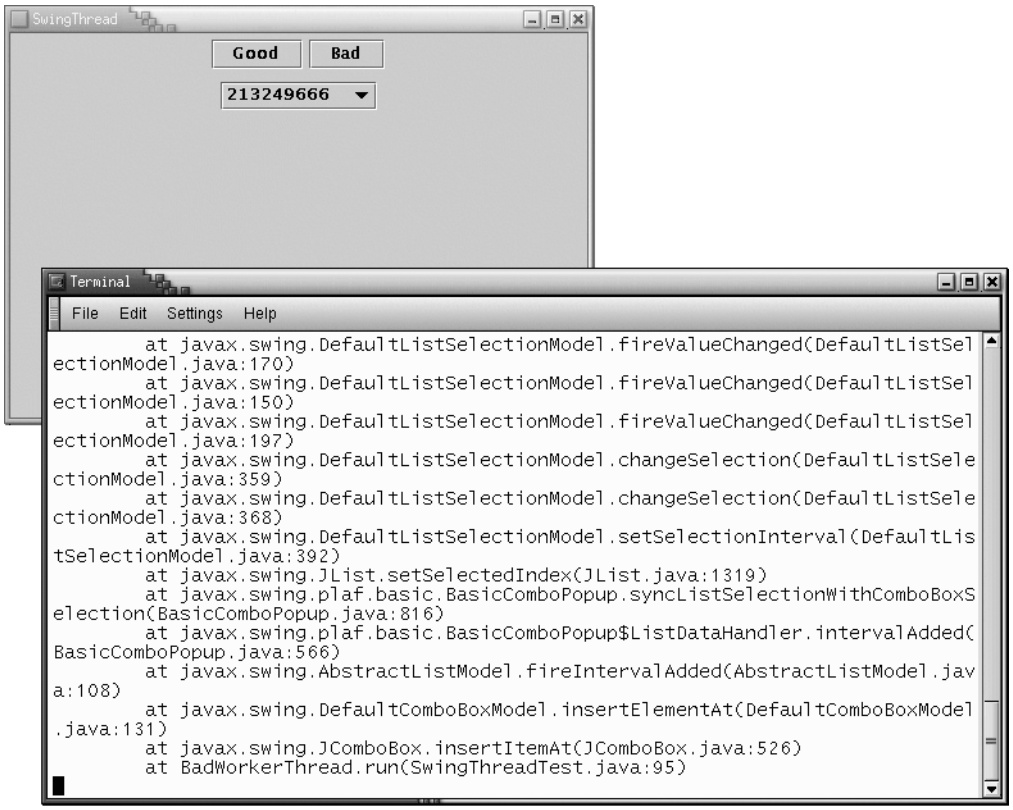

Figure 1–10: Exception reports in the console

What is going on? When an element is inserted into the combo box, the combo box fires an event to update the display. Then, the display code springs into action, reading the current size of the combo box and preparing to display the values. But the worker thread keeps on going, which can occasionally result in a reduction of the count of the values in the combo box. The display code then thinks that there are more values in the model than there actually are, asks for nonexistent values, and triggers an `ArrayIndexOutOfBounds` exception.

This situation could have been avoided by locking the combo box object while displaying it. However, the designers of Swing decided not to expend any effort to make Swing thread safe, for two reasons. First, synchronization takes time, and nobody wanted to slow down Swing any further. More importantly, the Swing team checked out what experience other teams had with thread-safe user interface toolkits. What they found was not encouraging. When building a user interface toolkit, you want it to be extensible so that other programmers can add their own user interface components. But user interface programmers using thread-safe toolkits turned out to be confused by the demands for synchronization and tended to create components that were prone to deadlocks.

Therefore, when you use threads together with Swing, you have to follow a few simple rules. First, however, let's see what threads are present in a Swing program.

Every Java application starts with a `main` method that runs in the *main thread.* In a Swing program, the `main` method typically does the following:

- First it calls a constructor that lays out components in a frame window;
- then it invokes the `show` or `setVisible` method on the window.

When the first window is shown, a second thread is created, the *event dispatch thread.* All event notifications, such as calls to `actionPerformed` or `paintComponent`, run in the event dispatch thread. The main thread keeps running until the `main` method exits. Usually, of course, the `main` method exits immediately after displaying the frame window (see Figure 1–11). Other threads are running behind the scenes, such as the thread that posts events into the event queue, but those threads are invisible to the application programmer.

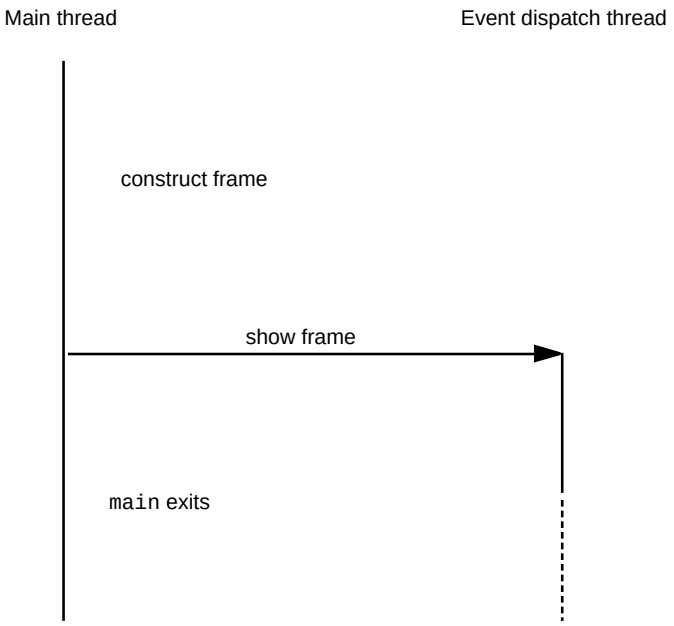

Figure 1–11: Threads in a Swing program

In a Swing application, essentially all code is contained in event handlers to respond to user interface and repaint requests. All that code runs on the event dispatch thread. Here are the rules that you need to follow.

1. If an action takes a long time, fire up a new thread to do the work. If you take a long time in the event dispatch thread, the application seems "dead" since it cannot respond to any events.
2. If an action can block on input or output, fire up a new thread to do the work. You don't want to freeze the user interface for the potentially indefinite time that a network connection is unresponsive.
3. If you need to wait for a specific amount of time, don't sleep in the event dispatch thread. Instead, use a timer.
4. The work that you do in your threads cannot touch the user interface. Read any information from the UI before you launch your threads, launch them, and then update the user interface from the event dispatching thread once the threads have completed.

The last rule is often called the *single thread rule* for Swing programming. There are a few exceptions to the single thread rule.

1. A few Swing methods are thread safe. They are specially marked in the API documentation with the sentence *"This method is thread safe, although most Swing methods are not."* The most useful among these thread-safe methods are:
   ```
   JTextComponent.setText
   JTextArea.insert
   JTextArea.append
   JTextArea.replaceRange
   ```
2. The following methods of the `JComponent` class can be called from any thread:
   ```
   repaint
   revalidate
   ```
   The `repaint` method schedules a repaint event. You use the `revalidate` method if the contents of a component have changed and the size and position of the component must be updated. The `revalidate` method marks the component's layout as invalid and schedules a layout event. (Just like paint events, layout events are *coalesced*. If there are multiple layout events in the event queue, the layout is only recomputed once.)

NOTE: We have used the `repaint` method many times in volume 1 of this book, but the `revalidate` method is less common. Its purpose is to force a layout of a component after the contents has changed. The traditional AWT has `invalidate` and `validate` methods to mark a component's layout as invalid and to force the layout of a component. For Swing components, you should simply call `revalidate` instead. (However, to force the layout of a `JFrame`, you still need to call `validate`—a `JFrame` is a `Component` but not a `JComponent`.)

3. You can safely add and remove event listeners in any thread. Of course, the `listener` methods will be invoked in the event dispatching thread.
4. You can construct components, set their properties, and add them into containers, as long as none of the components have been *realized*. A component has been realized if it can receive paint or validation events. This is the case as soon as the `show`, `setVisible(true)`, or `pack` methods have been invoked on the component, or if the component has been added to a container that has been realized. Once a component has been realized, you can no longer manipulate it from another thread.

   In particular, you can create the GUI of an application in the `main` method before calling `show`, and you can create the GUI of an applet in the applet constructor or the `init` method.

These rules look complex, but they aren't actually difficult to follow. It is an easy matter to start a new thread to start a time-consuming process. Upon a user

request, gather all the necessary information from the GUI, pass them to a thread, and start the thread.

```
public void actionPerformed(ActionEvent e)
{
   // gather data needed by thread
   MyThread t = new MyThread(data);
   t.start();
}
```

The difficult part is to update the GUI to indicate progress within your thread and to present the result of the work when your thread is finished. Remember that you can't touch any Swing components from your thread. For example, if you want to update a progress bar or a label text, then you can't simply set its value from your thread.

To solve this problem, there are two convenient utility methods that you can use in any thread to add arbitrary actions to the event queue. For example, suppose you want to periodically update a label "*x*% `complete`" in a thread, to indicate progress. You can't call `label.setText` from your thread, but you can use the `invokeLater` and `invokeAndWait` methods of the `EventQueue` class to have that call executed in the event dispatching thread.

NOTE: These methods are also available in the `javax.swing.SwingUtilities` class. If you use Swing with Java Development Kit (JDK) 1.1, you need to use that class—the methods were added to `EventQueue` in JDK 1.2.

Here is what you need to do. You place the Swing code into the `run` method of a class that implements the `Runnable` interface. Then, you create an object of that class and pass it to the static `invokeLater` or `invokeAndWait` method. For example, here is how you can update a label text. First, create the class with the `run` method.

```
public class LabelUpdater implements Runnable
{
   public LabelUpdater(JLabel aLabel, int aPercentage)
   {
      label = aLabel;
      percentage = aPercentage;
   }

   public void run()
   {
      label.setText(percentage + "% complete");
   }
}
```

Then, create an object and pass it to the `invokeLater` method.

```
Runnable updater = new LabelUpdater(label, percentage);
EventQueue.invokeLater(updater);
```

The `invokeLater` method returns immediately when the event is posted to the event queue. The `run` method is executed asynchronously. The `invokeAndWait` method waits until the `run` method has actually been executed. The `EventQueue` class handles the details of the synchronization. In the situation of updating a progress label, the `invokeLater` method is more appropriate. Users would rather have the worker thread make more progress than insist on the most precise display of the completed percentage.

To invoke code in the event dispatch thread, anonymous inner classes offer a useful shortcut. For example, the sample code given above can be simplified to the following cryptic, but shorter, command:

```
EventQueue.invokeLater(new
   Runnable()
   {
      public void run()
      {
         label.setText(percentage + "% complete");
      }
   });
```

NOTE: The `invokeLater` and `invokeAndWait` methods use objects that implement the `Runnable` interface. You already saw how to construct new threads out of `Runnable` objects. However, in this case, the code of the `run` method executes in the event dispatching thread, not a new thread.

Example 1–7 demonstrates how to use the `invokeLater` method to safely modify the contents of a combo box. If you click on the "Good" button, a thread inserts and removes numbers. However, the actual modification takes place in the event dispatching thread.

Example 1–7: SwingThreadTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5.
6. /**
7.    This program demonstrates that a thread that
8.    runs in parallel with the event dispatch thread
9.    can cause errors in Swing components.
10. */
11. public class SwingThreadTest
```


```
12. {
13.    public static void main(String[] args)
14.    {
15.       SwingThreadFrame frame = new SwingThreadFrame();
16.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17.       frame.show();
18.    }
19. }
20.
21. /**
22.    This frame has two buttons to fill a combo box from a
23.    separate thread. The "Good" button uses the event queue,
24.    the "Bad" button modifies the combo box directly.
25. */
26. class SwingThreadFrame extends JFrame
27. {
28.    public SwingThreadFrame()
29.    {
30.       setTitle("SwingThread");
31.       setSize(WIDTH, HEIGHT);
32.
33.       final JComboBox combo = new JComboBox();
34.
35.       JPanel p = new JPanel();
36.       p.add(combo);
37.       getContentPane().add(p, BorderLayout.CENTER);
38.
39.       JButton b = new JButton("Good");
40.       b.addActionListener(new ActionListener()
41.          {
42.             public void actionPerformed(ActionEvent event)
43.             {
44.                combo.showPopup();
45.                new GoodWorkerThread(combo).start();
46.             }
47.          });
48.       p = new JPanel();
49.       p.add(b);
50.       b = new JButton("Bad");
51.       b.addActionListener(new ActionListener()
52.          {
53.             public void actionPerformed(ActionEvent event)
54.             {
55.                combo.showPopup();
56.                new BadWorkerThread(combo).start();
57.             }
58.          });
59.       p.add(b);
```

```
60.
61.       getContentPane().add(p, BorderLayout.NORTH);
62.    }
63.
64.    public static final int WIDTH = 450;
65.    public static final int HEIGHT = 300;
66. }
67.
68. /**
69.    This thread modifies a combo box by randomly adding
70.    and removing numbers. This can result in errors because
71.    the combo box is not synchronized and the event dispatch
72.    thread accesses the combo box to repaint it.
73. */
74. class BadWorkerThread extends Thread
75. {
76.    public BadWorkerThread(JComboBox aCombo)
77.    {
78.       combo = aCombo;
79.       generator = new Random();
80.    }
81.
82.    public void run()
83.    {
84.       try
85.       {
86.          while (!interrupted())
87.          {
88.             int i = Math.abs(generator.nextInt());
89.             if (i % 2 == 0)
90.                combo.insertItemAt(new Integer(i), 0);
91.             else if (combo.getItemCount() > 0)
92.                combo.removeItemAt(i % combo.getItemCount());
93.
94.             sleep(1);
95.          }
96.       }
97.       catch (InterruptedException exception) {}
98.    }
99.
100.   private JComboBox combo;
101.   private Random generator;
102. }
103.
104. /**
105.    This thread modifies a combo box by randomly adding
106.    and removing numbers. In order to ensure that the
107.    combo box is not corrupted, the editing operations are
108.    forwarded to the event dispatch thread.
```


```
*/
class GoodWorkerThread extends Thread
{
   public GoodWorkerThread(JComboBox aCombo)
   {
      combo = aCombo;
      generator = new Random();
   }

   public void run()
   {
      try
      {
         while (!interrupted())
         {
            EventQueue.invokeLater(new
               Runnable()
               {
                  public void run()
                  {
                     int i = Math.abs(generator.nextInt());

                     if (i % 2 == 0)
                        combo.insertItemAt(new Integer(i), 0);
                     else if (combo.getItemCount() > 0)
                       combo.removeItemAt(i % combo.getItemCount());
                  }
               });
            Thread.sleep(1);
         }
      }
      catch (InterruptedException exception) {}
   }

   private JComboBox combo;
   private Random generator;
}
```

java.awt.EventQueue

- `static void invokeLater(Runnable runnable)`
  Causes the `run` method of the `runnable` object to be executed in the event dispatch thread, after pending events have been processed.
- `static void invokeAndWait(Runnable runnable)`
  Causes the `run` method of the `runnable` object to be executed in the event dispatch thread, after pending events have been processed. This call blocks until the `run` method has terminated.

Animation

In this section, we dissect one of the most common uses for threads in applets: animation. An animation sequence displays images, giving the viewer the illusion of motion. Each of the images in the sequence is called a *frame*. If the frames are complex, they should be rendered ahead of time—the computer running the applet may not have the horsepower to compute images fast enough for real-time animation.

Figure 1–12: This file has 36 images

You can put each frame in a separate image file or put all frames into one file. We do the latter. It makes the process of loading the image much easier. In our example, we use a file with 36 images of a rotating globe, courtesy of Silviu Marghescu of the University of Maryland. Figure 1–12 shows the first few frames.

The animation applet must first acquire all the frames. Then, it shows each of them in turn for a fixed time. To draw the `i`'th frame, we make a method call as follows:

```
g.drawImage(image, 0, - i * imageHeight
   / imageCount, null);
```

Figure 1–13 shows the *negative* offset of the *y*-coordinate.

This offset causes the first frame to be well above the origin of the canvas. The top of the `i`'th frame becomes the top of the canvas. After a delay, we increment `i` and draw the next frame.

We use a `MediaTracker` object to load the image. Behind the scenes and transparent to the programmer, the `addImage` method fires up a new thread to acquire the image. Loading an image can be very slow, especially if the image has many frames or is located across the network. The `waitForID` call blocks until the image is fully loaded.

NOTE: Readers who are familiar with the SDK 1.4 image I/O package may wonder why we don't use `ImageIO.read` to read the image. That method creates a temporary file—an operation that is not legal for an applet in the sandbox.

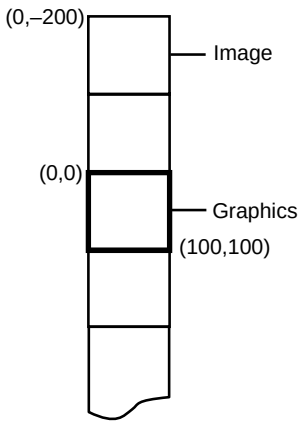

Figure 1–13: Picking a frame from a strip of frames

Once the image is loaded, we render one frame at a time. Our applet starts a single thread.

```
class Animation extends JApplet
{
   . . .
   private Thread runner = null;
}
```

You will see such a thread variable in many applets. Often, it is called `kicker`, and we once saw `killer` as the variable name. We think `runner` makes more sense, though.

First and foremost, we will use this thread to:

- Start the animation when the user is watching the applet;
- Stop the animation when the user has switched to a different page in the browser.

We do these tasks by creating the thread in the `start` method of the applet and by interrupting it in the `stop` method.

```
class Animation extends JApplet
{
   public void start()
   {
      if (runner == null)
      {
         runner = new
            Thread()
            {
               public void run()
               {
                  . . .
```

```
            }
         };
      runner.start();
   }
}

public void stop()
{
   runner.interrupt();
   runner = null;
}
. . .
}
```

Here is the `run` method. It simply loops, painting the screen, advancing the frame counter, and sleeping when it can.

```
public void run()
{
   try
   {
      while (!Thread.interrupted())
      {
         repaint();
         current = (current + 1) % imageCount;
         Thread.sleep(200);
      }
   }
   catch(InterruptedException e) {}
}
```

Finally, we implement another mechanism for stopping and restarting the animation. When you click the mouse on the applet window, the animation stops. When you click again, it restarts. Note that we use the thread variable, `runner`, as an indication whether the thread is currently running or not. Whenever the thread is terminated, the variable is set to `null`. This is a common idiom that you will find in many multithreaded applets.

```
public void init()
{
   addMouseListener(new
      MouseAdapter()
      {
         public void mousePressed(MouseEvent evt)
         {
            if (runner == null)
               start();
            else
```


```
                stop();
            }
         });

      . . .
   }
```

The applet reads the name of the image and the number of frames in the strip from the `param` section in the HTML file.

```
<applet code=Animation.class width=100 height=100>
<param name=imagename value="globe.gif">
<param name=imagecount value="36">
</applet>
```

Example 1–8 is the code of the applet. Note that the `start` and `stop` methods start and stop the applet—they are *not* methods of the thread that is generated.

This animation applet is simplified to show you what goes on behind the scenes, and to help you understand other applets with a similar structure. If you are interested only in how to put a moving image on your web page, look instead at the `Animator` applet in the demo section of the JDK. That applet has many more options than ours, and it enables you to add sound.

Furthermore, as you will see in the next section, you really don't need to implement your own threads for a simple animation—you can just use a Swing timer.

Example 1–8: Animation.java

```
1. import java.awt.*;
2. import java.awt.image.*;
3. import java.awt.event.*;
4. import javax.swing.*;
5. import java.net.*;
6.
7. /**
8.    An applet that shows a rotating globe.
9. */
10. public class Animation extends JApplet
11. {
12.    public void init()
13.    {
14.       addMouseListener(new MouseAdapter()
15.          {
16.             public void mousePressed(MouseEvent evt)
17.             {
18.                if (runner == null)
19.                   start();
20.                else
```

```
21.                stop();
22.             }
23.          });
24.
25.       try
26.       {
27.          String imageName = getParameter("imagename");
28.          imageCount = 1;
29.          String param = getParameter("imagecount");
30.          if (param != null)
31.             imageCount = Integer.parseInt(param);
32.          current = 0;
33.          image = null;
34.          loadImage(new URL(getDocumentBase(), imageName));
35.       }
36.       catch (Exception e)
37.       {
38.          showStatus("Error: " + e);
39.       }
40.    }
41.
42.    /**
43.       Loads an image.
44.       @param url the URL of the image file
45.    */
46.    public void loadImage(URL url)
47.       throws InterruptedException
48.          // thrown by MediaTracker.waitFor
49.    {
50.       image = getImage(url);
51.       MediaTracker tracker = new MediaTracker(this);
52.       tracker.addImage(image, 0);
53.       tracker.waitForID(0);
54.       imageWidth = image.getWidth(null);
55.       imageHeight = image.getHeight(null);
56.       resize(imageWidth, imageHeight / imageCount);
57.    }
58.
59.    public void paint(Graphics g)
60.    {
61.       if (image == null) return;
62.       g.drawImage(image, 0, - (imageHeight / imageCount)
63.          * current, null);
64.    }
65.
66.    public void start()
67.    {
```


```
68.       runner = new
69.          Thread()
70.          {
71.             public void run()
72.             {
73.                try
74.                {
75.                   while (!Thread.interrupted())
76.                   {
77.                      repaint();
78.                      current = (current + 1) % imageCount;
79.                      Thread.sleep(200);
80.                   }
81.                }
82.                catch(InterruptedException e) {}
83.             }
84.          };
85.       runner.start();
86.       showStatus("Click to stop");
87.    }
88.
89.    public void stop()
90.    {
91.       runner.interrupt();
92.       runner = null;
93.       showStatus("Click to restart");
94.    }
95.
96.    private Image image;
97.    private int current;
98.    private int imageCount;
99.    private int imageWidth;
100.   private int imageHeight;
101.   private Thread runner;
102.}
```

Timers

In many programming environments, you can set up timers. A timer alerts your program elements at regular intervals. For example, to display a clock in a window, the clock object must be notified once every second.

Swing has a built-in timer class that is easy to use. You construct a timer by supplying an object of a class that implements the `ActionListener` interface and the delay between timer alerts, in milliseconds.

```
Timer t = new Timer(1000, listener);
```

Call

```
t.start();
```

to start the timer. Then, the `actionPerformed` method of the listener class is called whenever a timer interval has elapsed. The `actionPerformed` method is automatically called on the event dispatch thread, not the timer thread, so that you can freely invoke `Swing` methods in the callback.

To stop the timer, call

```
t.stop();
```

Then the timer stops sending action events until you restart it.

NOTE: SDK 1.3 has an unrelated `java.util.Timer` class to schedule a `TimerTask` for later execution. The `TimerTask` class implements the `Runnable` interface and also supplies a `cancel` method to cancel the task. However, the `java.util.Timer` class has no provision for a periodic callback.

The example program at the end of this section puts the Swing timer to work. Figure 1–4 shows six different clocks.

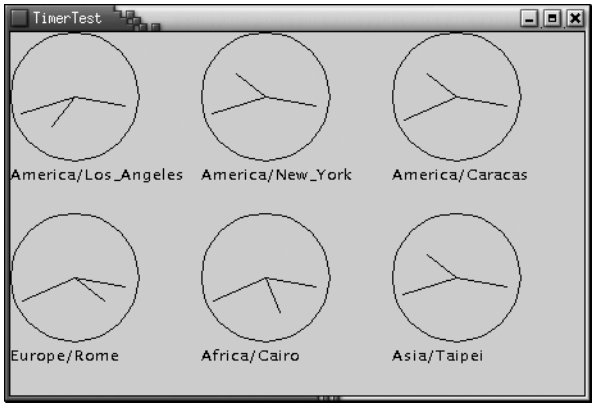

Figure 1–14: Clock threads

Each clock is an instance of the `ClockCanvas` class. The constructor sets up the timer:

```
public ClockCanvas(String tz)
{
   calendar = new GregorianCalendar(TimeZone.getTimeZone(tz));
   Timer t = new Timer(1000, new
      ActionListener()
```


```
         {
            public void actionPerformed(ActionEvent event)
            {
               calendar.setTime(new Date());
               repaint();
            }
         });
      t.start();
      . . .
   }
```

The `actionPerformed` method of the timer's anonymous action listener gets called approximately once per second. It calls `new Date()` to get the current time and repaints the clock.

As you can see, no thread programming is required at all in this case. The Swing timer takes care of the thread details. We could have used a timer for the animation of the preceding section as well.

You will find the complete code in Example 1–9.

Example 1–9: TimerTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. import java.util.*;
5. import javax.swing.Timer;
6.
7. /**
8.    This class shows a frame with several clocks that
9.    are updated by a timer thread.
10. */
11. public class TimerTest
12. {
13.    public static void main(String[] args)
14.    {
15.       TimerTestFrame frame = new TimerTestFrame();
16.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17.       frame.show();
18.    }
19. }
20.
21. /**
22.    The frame holding the clocks.
23. */
24. class TimerTestFrame extends JFrame
25. {
26.    public TimerTestFrame()
```

```
27.    {
28.       setTitle("TimerTest");
29.       setSize(WIDTH, HEIGHT);
30.
31.       Container c = getContentPane();
32.       c.setLayout(new GridLayout(2, 3));
33.       c.add(new ClockCanvas("America/Los_Angeles"));
34.       c.add(new ClockCanvas("America/New_York"));
35.       c.add(new ClockCanvas("America/Caracas"));
36.       c.add(new ClockCanvas("Europe/Rome"));
37.       c.add(new ClockCanvas("Africa/Cairo"));
38.       c.add(new ClockCanvas("Asia/Taipei"));
39.    }
40.
41.    public static final int WIDTH = 450;
42.    public static final int HEIGHT = 300;
43. }
44.
45. /**
46.    The canvas to display a clock that is updated by a timer.
47. */
48. class ClockCanvas extends JPanel
49. {
50.    /**
51.       Constructs a clock canvas.
52.       @param tz the time zone string
53.    */
54.    public ClockCanvas(String tz)
55.    {
56.       zone = tz;
57.       calendar = new GregorianCalendar(TimeZone.getTimeZone(tz));
58.       Timer t = new Timer(1000, new
59.          ActionListener()
60.          {
61.             public void actionPerformed(ActionEvent event)
62.             {
63.                calendar.setTime(new Date());
64.                repaint();
65.             }
66.          });
67.       t.start();
68.       setSize(WIDTH, HEIGHT);
69.    }
70.
71.    public void paintComponent(Graphics g)
72.    {
73.       super.paintComponent(g);
74.       g.drawOval(0, 0, 100, 100);
```


```
75.
76.       int seconds = calendar.get(Calendar.HOUR) * 60 * 60
77.          + calendar.get(Calendar.MINUTE) * 60
78.          + calendar.get(Calendar.SECOND);
79.       double hourAngle = 2 * Math.PI
80.          * (seconds - 3 * 60 * 60) / (12 * 60 * 60);
81.       double minuteAngle = 2 * Math.PI
82.          * (seconds - 15 * 60) / (60 * 60);
83.       double secondAngle = 2 * Math.PI
84.          * (seconds - 15) / 60;
85.       g.drawLine(50, 50, 50 + (int)(30
86.          * Math.cos(hourAngle)),
87.          50 + (int)(30 * Math.sin(hourAngle)));
88.       g.drawLine(50, 50, 50 + (int)(40
89.          * Math.cos(minuteAngle)),
90.          50 + (int)(40 * Math.sin(minuteAngle)));
91.       g.drawLine(50, 50, 50 + (int)(45
92.          * Math.cos(secondAngle)),
93.          50 + (int)(45 * Math.sin(secondAngle)));
94.       g.drawString(zone, 0, 115);
95.    }
96.
97.    private String zone;
98.    private GregorianCalendar calendar;
99.
100.   public static final int WIDTH = 125;
101.   public static final int HEIGHT = 125;
102. }
```

javax.swing.Timer

- `Timer(int delay, ActionListener listener)`
  Creates a new timer that sends events to a listener.

  Parameters: `delay` the delay, in milliseconds, between event notifications

  `listener` the action listener to be notified when the delay has elapsed

- `void start()`
  Start the timer. After this call, the timer starts sending events to its action listener.

- `void stop()`
  Stop the timer. After this call, the timer stops sending events to its action listener.

Progress Bars

A *progress bar* is a simple component—just a rectangle that is partially filled with color to indicate the progress of an operation. By default, progress is indicated by a string "*n %*". You can see a progress bar in the bottom right of Figure 1–15.

Figure 1–15: A progress bar

You construct a progress bar much as you construct a slider, by supplying the minimum and maximum value and an optional orientation:

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

You can also set the minimum and maximum with the `setMinimum` and `setMaximum` methods.

Unlike a slider, the progress bar cannot be adjusted by the user. Your program needs to call `setValue` to update it.

If you call

```
progressBar.setStringPainted(true);
```

the progress bar computes the completion percentage and displays a string "*n %*". If you want to show a different string, you can supply it with the `setString` method:

```
if (progressBar.getValue() > 900)
   progressBar.setString("Almost Done");
```

The program in Example 1–10 shows a progress bar that monitors a simulated time-consuming activity.

The `SimulatedActivity` class implements a thread that increments a value `current` ten times per second. When it reaches a target value, the thread finishes. If you want to terminate the thread before it has reached its target, you should interrupt it.

```
class SimulatedActivity extends Thread
{  . . .
   public void run()
   {
      try
```


```
      {
         while (current < target && !interrupted())
         {
            sleep(100);
            current++;
         }
      }
      catch(InterruptedException e)
      {
      }
   }

   int current;
   int target;
}
```

When you click on the "Start" button, a new `SimulatedActivity` thread is started. To update the progress bar, it would appear to be an easy matter for the simulated activity thread to make calls to the `setValue` method. But that is not thread safe. Recall that you should call Swing methods only from the event dispatch thread. In practice, it is also unrealistic. In general, a worker thread is not aware of the existence of the progress bar. Instead, the example program shows how to launch a timer that periodically polls the thread for a progress status and updates the progress bar.

CAUTION: If a worker thread is aware of a progress bar that monitors its progress, remember that it cannot set the progress bar value directly. To set the value in the event dispatch thread, the worker thread can use the `SwingUtilities.invokeLater` method.

Recall that a Swing timer calls the `actionPerformed` method of its listeners and that these calls occur in the event dispatch thread. That means it is safe to update Swing components in the timer callback. Here is the timer callback from the example program. The current value of the simulated activity is displayed both in the text area and the progress bar. If the end of the simulation has been reached, the timer is stopped and the "Start" button is reenabled.

```
public void actionPerformed(ActionEvent event)
{
   int current = activity.getCurrent();
   // show progress

   textArea.append(current + "\n");
   progressBar.setValue(current);

   // check if task is completed
   if (current == activity.getTarget())
   {
```

```
      activityMonitor.stop();
      startButton.setEnabled(true);
   }
}
```

Example 1–10 shows the full program code.

NOTE: SDK 1.4 adds support for an *indeterminate* progress bar that shows an animation indicating some kind of progress, without giving an indication of the percentage of completion. That is the kind of progress bar that you see in your browser—it indicates that the browser is waiting for the server and has no idea how long the wait may be. To display the "indeterminate wait" animation, call the `setIndeterminate` method.

Example 1–10: ProgressBarTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.event.*;
6. import javax.swing.Timer;
7.
8. /**
9.    This program demonstrates the use of a progress bar
10.   to monitor the progress of a thread.
11. */
12. public class ProgressBarTest
13. {
14.    public static void main(String[] args)
15.    {
16.       ProgressBarFrame frame = new ProgressBarFrame();
17.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.       frame.show();
19.    }
20. }
21.
22. /**
23.    A frame that contains a button to launch a simulated activity,
24.    a progress bar, and a text area for the activity output.
25. */
26. class ProgressBarFrame extends JFrame
27. {
28.    public ProgressBarFrame()
29.    {
30.       setTitle("ProgressBarTest");
31.       setSize(WIDTH, HEIGHT);
```


```
32.
33.       Container contentPane = getContentPane();
34.
35.       // this text area holds the activity output
36.       textArea = new JTextArea();
37.
38.       // set up panel with button and progress bar
39.
40.       JPanel panel = new JPanel();
41.       startButton = new JButton("Start");
42.       progressBar = new JProgressBar();
43.       progressBar.setStringPainted(true);
44.       panel.add(startButton);
45.       panel.add(progressBar);
46.       contentPane.add(new JScrollPane(textArea),
47.          BorderLayout.CENTER);
48.       contentPane.add(panel, BorderLayout.SOUTH);
49.
50.       // set up the button action
51.
52.       startButton.addActionListener(new
53.          ActionListener()
54.          {
55.             public void actionPerformed(ActionEvent event)
56.             {
57.                progressBar.setMaximum(1000);
58.                activity = new SimulatedActivity(1000);
59.                activity.start();
60.                activityMonitor.start();
61.                startButton.setEnabled(false);
62.             }
63.          });
64.
65.
66.       // set up the timer action
67.
68.       activityMonitor = new Timer(500, new
69.          ActionListener()
70.          {
71.             public void actionPerformed(ActionEvent event)
72.             {
73.                int current = activity.getCurrent();
74.
75.                // show progress
76.                textArea.append(current + "\n");
77.                progressBar.setValue(current);
78.
79.                // check if task is completed
```

```
80.                if (current == activity.getTarget())
81.                {
82.                   activityMonitor.stop();
83.                   startButton.setEnabled(true);
84.                }
85.             }
86.          });
87.    }
88.
89.    private Timer activityMonitor;
90.    private JButton startButton;
91.    private JProgressBar progressBar;
92.    private JTextArea textArea;
93.    private SimulatedActivity activity;
94.
95.    public static final int WIDTH = 300;
96.    public static final int HEIGHT = 200;
97. }
98.
99. /**
100.    A simulated activity thread.
101. */
102. class SimulatedActivity extends Thread
103. {
104.    /**
105.       Constructs the simulated activity thread object. The
106.       thread increments a counter from 0 to a given target.
107.       @param t the target value of the counter.
108.    */
109.    public SimulatedActivity(int t)
110.    {
111.       current = 0;
112.       target = t;
113.    }
114.
115.    public int getTarget()
116.    {
117.       return target;
118.    }
119.
120.    public int getCurrent()
121.    {
122.       return current;
123.    }
124.
125.    public void run()
126.    {
127.       try
```


```
128.       {
129.          while (current < target && !interrupted())
130.          {
131.             sleep(100);
132.             current++;
133.          }
134.       }
135.       catch(InterruptedException e)
136.       {
137.       }
138.    }
139.
140.    private int current;
141.    private int target;
142. }
```

Progress Monitors

A progress bar is a very simple component that can be placed inside a window. In contrast, a `ProgressMonitor` is a complete dialog box that contains a progress bar (see Figure 1–16). The dialog contains "OK" and "Cancel" buttons. If you click either, the monitor dialog is closed. In addition, your program can query whether the user has canceled the dialog and terminate the monitored action. (Note that the class name does not start with a "`J`".)

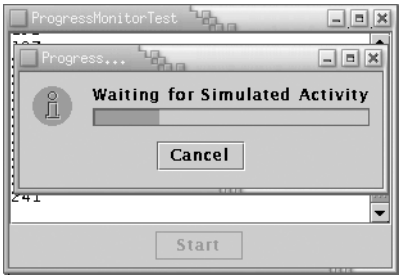

Figure 1–16: A progress monitor dialog

You construct a progress monitor by supplying the following:

- The parent component over which the dialog should pop up;
- An object (which should be a string, icon, or component) that is displayed on the dialog;
- An optional note to display below the object;
- The minimum and maximum values.

However, the progress monitor cannot measure progress or cancel an activity by itself.

You still need to periodically set the progress value by calling the `setProgress` method. (This is the equivalent of the `setValue` method of the `JProgressBar` class.) As you update the progress value, you should also call the `isCanceled` method to see if the program user has clicked on the "Cancel" button.

When the monitored activity has concluded, you should call the `close` method to dismiss the dialog. You can reuse the same dialog by calling `start` again.

The example program looks very similar to that of the preceding section. We still need to launch a timer to watch over the progress of the simulated activity and update the progress monitor. Here is the timer callback.

```
public void actionPerformed(ActionEvent event)
{
   int current = activity.getCurrent();

   // show progress
   textArea.append(current + "\n");
   progressDialog.setProgress(current);

   // check if task is completed or canceled
   if (current == activity.getTarget()
      || progressDialog.isCanceled())
   {
      activityMonitor.stop();
      progressDialog.close();
      activity.interrupt();
      startButton.setEnabled(true);
   }
}
```

Note that there are two conditions for termination. The activity might have completed, or the user might have canceled it. In each of these cases, we close down:

- the timer that monitored the activity;
- the progress dialog;
- the activity itself (by interrupting the thread).

If you run the program in Example 1–11, you can observe an interesting feature of the progress monitor dialog. The dialog doesn't come up immediately. Instead, it waits a for a short interval to see if the activity has already been completed or is likely to complete in less time than it would take for the dialog to appear. You control the timing as follows. Use the `setMillisToDecidePopup` method to set the number of milliseconds to wait between the construction of the dialog object and the decision whether to show the pop-up at all. The default value is 500 milliseconds. The `setMillisToPopup` is the time that you estimate that the dialog needs to pop up. The Swing designers set this value to a default of 2 seconds.

Clearly they were mindful of the fact that Swing dialogs don't always come up as snappily as we all would like. You should probably not touch this value.

Example 1–11 shows the progress monitor in action, again measuring the progress of a simulated activity. As you can see, the progress monitor is convenient to use and only requires that you periodically query the thread that you want to monitor.

Example 1–11: ProgressMonitorTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.event.*;
6. import javax.swing.Timer;
7.
8. /**
9.    A program to test a progress monitor dialog.
10. */
11. public class ProgressMonitorTest
12. {
13.    public static void main(String[] args)
14.    {
15.       JFrame frame = new ProgressMonitorFrame();
16.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17.       frame.show();
18.    }
19. }
20.
21. /**
22.    A frame that contains a button to launch a simulated activity
23.    and a text area for the activity output.
24. */
25. class ProgressMonitorFrame extends JFrame
26. {
27.    public ProgressMonitorFrame()
28.    {
29.       setTitle("ProgressMonitorTest");
30.       setSize(WIDTH, HEIGHT);
31.
32.       Container contentPane = getContentPane();
33.
34.       // this text area holds the activity output
35.       textArea = new JTextArea();
36.
37.       // set up a button panel
38.       JPanel panel = new JPanel();
39.       startButton = new JButton("Start");
40.       panel.add(startButton);
```

```
41.
42.       contentPane.add(new JScrollPane(textArea),
43.          BorderLayout.CENTER);
44.       contentPane.add(panel, BorderLayout.SOUTH);
45.
46.       // set up the button action
47.
48.       startButton.addActionListener(new
49.          ActionListener()
50.          {
51.             public void actionPerformed(ActionEvent event)
52.             {
53.                // start activity
54.                activity = new SimulatedActivity(1000);
55.                activity.start();
56.
57.                // launch progress dialog
58.                progressDialog = new ProgressMonitor(
59.                   ProgressMonitorFrame.this,
60.                   "Waiting for Simulated Activity",
61.                   null, 0, activity.getTarget());
62.
63.                // start timer
64.                activityMonitor.start();
65.
66.                startButton.setEnabled(false);
67.             }
68.          });
69.
70.       // set up the timer action
71.
72.       activityMonitor = new Timer(500, new
73.          ActionListener()
74.          {
75.             public void actionPerformed(ActionEvent event)
76.             {
77.                int current = activity.getCurrent();
78.
79.                // show progress
80.                textArea.append(current + "\n");
81.                progressDialog.setProgress(current);
82.
83.                // check if task is completed or canceled
84.                if (current == activity.getTarget()
85.                   || progressDialog.isCanceled())
86.                {
87.                   activityMonitor.stop();
88.                   progressDialog.close();
89.                   activity.interrupt();
```


```
90.                   startButton.setEnabled(true);
91.                }
92.             }
93.          });
94.    }
95.
96.    private Timer activityMonitor;
97.    private JButton startButton;
98.    private ProgressMonitor progressDialog;
99.    private JTextArea textArea;
100.   private SimulatedActivity activity;
101.
102.   public static final int WIDTH = 300;
103.   public static final int HEIGHT = 200;
104. }
105.
106. /**
107.    A simulated activity thread.
108. */
109. class SimulatedActivity extends Thread
110. {
111.    /**
112.       Constructs the simulated activity thread object. The
113.       thread increments a counter from 0 to a given target.
114.       @param t the target value of the counter.
115.    */
116.    public SimulatedActivity(int t)
117.    {
118.       current = 0;
119.       target = t;
120.    }
121.
122.    public int getTarget()
123.    {
124.       return target;
125.    }
126.
127.    public int getCurrent()
128.    {
129.       return current;
130.    }
131.
132.    public void run()
133.    {
134.       try
135.       {
136.          while (current < target && !interrupted())
137.          {
138.             sleep(100);
139.             current++;
```

```
140.          }
141.       }
142.       catch(InterruptedException e)
143.       {
144.       }
145.    }
146.
147.    private int current;
148.    private int target;
149. }
```

Monitoring the Progress of Input Streams

The Swing package contains a useful stream filter, `ProgressMonitorInputStream`, that automatically pops up a dialog that monitors how much of the stream has been read.

This filter is extremely easy to use. You sandwich in a `ProgressMonitorInputStream` between your usual sequence of filtered streams. (See Chapter 12 of Volume 1 for more information on streams.)

For example, suppose you read text from a file. You start out with a `FileInputStream`:

```
FileInputStream in = new FileInputStream(f);
```

Normally, you would convert `fileIn` to an `InputStreamReader`.

```
InputStreamReader reader = new InputStreamReader(in);
```

However, to monitor the stream, first turn the file input stream into a stream with a progress monitor:

```
ProgressMonitorInputStream progressIn
   = new ProgressMonitorInputStream(parent, caption, in);
```

You need to supply the parent component, a caption, and, of course, the stream to monitor. The `read` method of the progress monitor stream simply passes along the bytes and updates the progress dialog.

You now go on building your filter sequence:

```
InputStreamReader reader = new InputStreamReader(progressIn);
```

That's all there is to it. When the file is read, the progress monitor automatically pops up. This is a very nice application of stream filtering.

CAUTION: The progress monitor stream uses the `available` method of the `InputStream` class to determine the total number of bytes in the stream. However, the `available` method only reports the number of bytes in the stream that are available *without blocking*. Progress monitors work well for files and HTTP URLs because their length is known in advance, but they don't work with all streams.

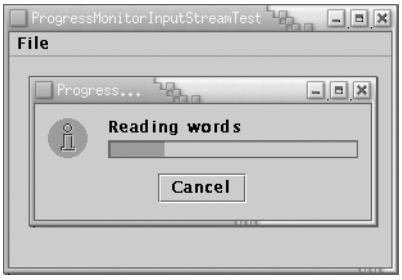

Figure 1–17: A progress monitor for an input stream

The program in Example 1–12 counts the lines in a file. If you read in a large file (such as "The Count of Monte Cristo" on the CD), then the progress dialog pops up.

Note that the program doesn't use a very efficient way of filling up the text area. It would be faster to first read in the file into a `StringBuffer` and then set the text of the text area to the string buffer contents. But in this example program, we actually like this slow approach—it gives you more time to admire the progress dialog.

To avoid flicker, the text area is not displayed while it is filled up.

Example 1–12: ProgressMonitorInputStreamTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.*;
5. import javax.swing.*;
6. import javax.swing.event.*;
7. import javax.swing.Timer;
8.
9. /**
10.    A program to test a progress monitor input stream.
11. */
12. public class ProgressMonitorInputStreamTest
13. {
14.    public static void main(String[] args)
15.    {
16.       JFrame frame = new TextFrame();
17.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.       frame.show();
19.    }
20. }
21.
22. /**
23.    A frame with a menu to load a text file and a text area
```

```
24.    to display its contents. The text area is constructed
25.    when the file is loaded and set as the content pane of
26.    the frame when the loading is complete. That avoids flicker
27.    during loading.
28. */
29. class TextFrame extends JFrame
30. {
31.    public TextFrame()
32.    {
33.       setTitle("ProgressMonitorInputStreamTest");
34.       setSize(WIDTH, HEIGHT);
35.
36.       // set up menu
37.
38.       JMenuBar menuBar = new JMenuBar();
39.       setJMenuBar(menuBar);
40.       JMenu fileMenu = new JMenu("File");
41.       menuBar.add(fileMenu);
42.       openItem = new JMenuItem("Open");
43.       openItem.addActionListener(new
44.          ActionListener()
45.          {
46.             public void actionPerformed(ActionEvent event)
47.             {
48.                try
49.                {
50.                   openFile();
51.                }
52.                catch(IOException exception)
53.                {
54.                   exception.printStackTrace();
55.                }
56.             }
57.          });
58.
59.       fileMenu.add(openItem);
60.       exitItem = new JMenuItem("Exit");
61.       exitItem.addActionListener(new
62.          ActionListener()
63.          {
64.             public void actionPerformed(ActionEvent event)
65.             {
66.                System.exit(0);
67.             }
68.          });
69.       fileMenu.add(exitItem);
70.    }
71.
```


```
72.    /**
73.       Prompts the user to select a file, loads the file into
74.       a text area, and sets it as the content pane of the frame.
75.    */
76.    public void openFile() throws IOException
77.    {
78.       JFileChooser chooser = new JFileChooser();
79.       chooser.setCurrentDirectory(new File("."));
80.       chooser.setFileFilter(
81.          new javax.swing.filechooser.FileFilter()
82.             {
83.                public boolean accept(File f)
84.                {
85.                   String fname = f.getName().toLowerCase();
86.                   return fname.endsWith(".txt")
87.                      || f.isDirectory();
88.                }
89.                public String getDescription()
90.                {
91.                   return "Text Files";
92.                }
93.             });
94.
95.       int r = chooser.showOpenDialog(this);
96.       if (r != JFileChooser.APPROVE_OPTION) return;
97.       final File f = chooser.getSelectedFile();
98.
99.       // set up stream and reader filter sequence
100.
101.      FileInputStream fileIn = new FileInputStream(f);
102.      ProgressMonitorInputStream progressIn
103.         = new ProgressMonitorInputStream(this,
104.            "Reading " + f.getName(), fileIn);
105.      InputStreamReader inReader
106.         = new InputStreamReader(progressIn);
107.      final BufferedReader in = new BufferedReader(inReader);
108.
109.      // the monitored activity must be in a new thread.
110.
111.      Thread readThread = new Thread()
112.      {
113.         public void run()
114.         {
115.            try
116.            {
117.               final JTextArea textArea = new JTextArea();
118.
119.               String line;
```

```
120.                while ((line = in.readLine()) != null)
121.                {
122.                   textArea.append(line);
123.                   textArea.append("\n");
124.                }
125.                in.close();
126.
127.                // set content pane in the event dispatch thread
128.                EventQueue.invokeLater(new
129.                   Runnable()
130.                   {
131.                      public void run()
132.                      {
133.                         setContentPane(new JScrollPane(textArea));
134.                         validate();
135.                      }
136.                   });
137.
138.             }
139.             catch(IOException exception)
140.             {
141.                exception.printStackTrace();
142.             }
143.          }
144.       };
145.
146.       readThread.start();
147.    }
148.
149.    private JMenuItem openItem;
150.    private JMenuItem exitItem;
151.
152.    public static final int WIDTH = 300;
153.    public static final int HEIGHT = 200;
154. }
```

javax.swing.JProgressBar

- `JProgressBar()`
- `JProgressBar(int direction)`
- `JProgressBar(int min, int max)`
- `JProgressBar(int direction, int min, int max)`

construct a horizontal slider with the given direction, minimum and maximum.

Parameters: `direction` one of `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL`. The default is horizontal.

`min`, `max` the minimum and maximum for the progress bar values. Defaults are 0 and 100.

- `int getMinimum()`
- `int getMaximum()`
- `void setMinimum(int value)`
- `void setMaximum(int value)`

  get and set the minimum and maximum values.

- `int getValue()`
- `void setValue(int value)`

  get and set the current value.

- `String getString()`
- `void setString(String s)`

  get and set the string to be displayed in the progress bar. If the string is `null`, then a default string "*n %*" is displayed.

- `boolean isStringPainted()`
- `void setStringPainted(boolean b)`

  get and set the "string painted" property. If this property is `true`, then a string is painted on top of the progress bar. The default is `false`; no string is painted.

- `boolean isIndeterminate()`
- `void setIndeterminate(boolean b)`

  get and set the "indeterminate" property (SDK 1.4). If this property is `true`, then the progress bar becomes a block that moves backwards and forwards, indicating a wait of unknown duration. The default is `false`.

`javax.swing.ProgressMonitor`

- `ProgressMonitor(Component parent, Object message, String note, int min, int max)`

  constructs a progress monitor dialog.

  Parameters: `parent` the parent component over which this dialog pops up.

  `message` the message object to display in the dialog.

  `note` the optional string to display under the message. If this value is `null`, then no space is set aside for the note, and a later call to `setNote` has no effect.

	`min`, `max`	the minimum and maximum values of the progress bar.

- `void setNote(String note)`
  changes the note text.
- `void setProgress(int value)`
  sets the progress bar value to the given value.
- `void close()`
  closes this dialog.
- `boolean isCanceled()`
  returns `true` if the user canceled this dialog.

`javax.swing.ProgressMonitorInputStream`

- `ProgressMonitorInputStream(Component parent, Object message, InputStream in)`
  constructs an input stream filter with an associated progress monitor dialog.

Parameters:		
	`parent`	the parent component over which this dialog pops up
	`message`	the message object to display in the dialog
	`in`	the input stream that is being monitored

Using Pipes for Communication Between Threads

Sometimes, the communication pattern between threads is very simple. A *producer* thread generates a stream of bytes. A *consumer* thread reads and processes that byte stream. If no bytes are available for reading, the consumer thread blocks. If the producer generates data much more quickly than the consumer can handle it, then the write operation of the producer thread blocks. The Java programming language has a convenient set of classes, `PipedInputStream` and `PipedOutputStream`, to implement this communication pattern. (There is another pair of classes, `PipedReader` and `PipedWriter`, if the producer thread generates a stream of Unicode characters instead of bytes.)

The principal reason to use pipes is to keep each thread simple. The producer thread simply sends its results to a stream and forgets about them. The consumer simply reads the data from a stream, without having to care where it comes from. By using pipes, you can connect multiple threads with each other without worrying about thread synchronization.

Example 1–13 is a program that shows off piped streams. We have a producer thread that emits random numbers at random times, a filter thread that reads the input numbers and continuously computes the average of the data, and a consumer thread that

prints out the answers. (You'll need to use CTRL+C to stop this program.) Figure 1–18 shows the threads and the pipes that connect them. UNIX users will recognize these pipe streams as the equivalent of pipes connecting processes in UNIX.

Piped streams are only appropriate if the communication between the threads is on a low level, such as a sequence of numbers as in this example. In other situations, you can use queues. The producing thread inserts objects into the queue, and the consuming thread removes them.

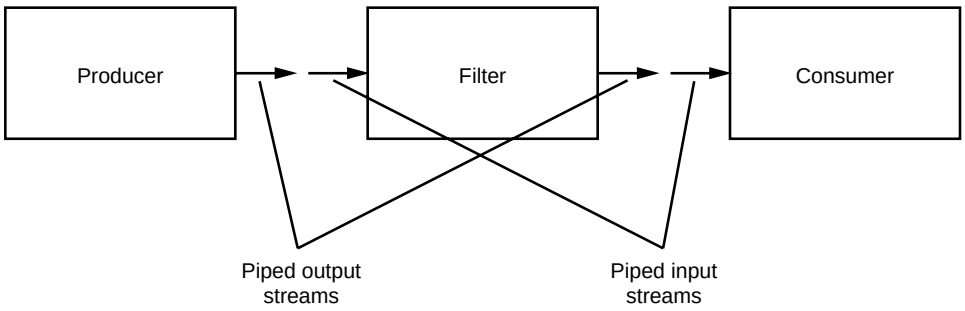

Figure 1–18: A sequence of pipes

Example 1–13: PipeTest.java

```
1. import java.util.*;
2. import java.io.*;
3.
4. /**
5.    This program demonstrates how multiple threads communicate
6.    through pipes.
7. */
8. public class PipeTest
9. {
10.    public static void main(String args[])
11.    {
12.       try
13.       {
14.          /* set up pipes */
15.          PipedOutputStream pout1 = new PipedOutputStream();
16.          PipedInputStream pin1 = new PipedInputStream(pout1);
17.
18.          PipedOutputStream pout2 = new PipedOutputStream();
19.          PipedInputStream pin2 = new PipedInputStream(pout2);
20.
21.          /* construct threads */
22.
23.          Producer prod = new Producer(pout1);
24.          Filter filt = new Filter(pin1, pout2);
```

```
25.          Consumer cons = new Consumer(pin2);
26.
27.          /* start threads */
28.
29.          prod.start();
30.          filt.start();
31.          cons.start();
32.       }
33.       catch (IOException e){}
34.    }
35. }
36.
37. /**
38.    A thread that writes random numbers to an output stream.
39. */
40. class Producer extends Thread
41. {
42.    /**
43.       Constructs a producer thread.
44.       @param os the output stream
45.    */
46.    public Producer(OutputStream os)
47.    {
48.       out = new DataOutputStream(os);
49.    }
50.
51.    public void run()
52.    {
53.       while (true)
54.       {
55.          try
56.          {
57.             double num = rand.nextDouble();
58.             out.writeDouble(num);
59.             out.flush();
60.             sleep(Math.abs(rand.nextInt() % 1000));
61.          }
62.          catch(Exception e)
63.          {
64.             System.out.println("Error: " + e);
65.          }
66.       }
67.    }
68.
69.    private DataOutputStream out;
70.    private Random rand = new Random();
71. }
72.
```


```
/**
   A thread that reads numbers from a stream and writes their
   average to an output stream.
*/
class Filter extends Thread
{
   /**
      Constructs a filter thread.
      @param is the output stream
      @param os the output stream
   */
   public Filter(InputStream is, OutputStream os)
   {
      in = new DataInputStream(is);
      out = new DataOutputStream(os);
   }

   public void run()
   {
      for (;;)
      {
         try
         {
            double x = in.readDouble();
            total += x;
            count++;
            if (count != 0) out.writeDouble(total / count);
         }
         catch(IOException e)
         {
            System.out.println("Error: " + e);
         }
      }
   }

   private DataInputStream in;
   private DataOutputStream out;
   private double total = 0;
   private int count = 0;
}

/**
   A thread that reads numbers from a stream and
   prints out those that deviate from previous inputs
   by a threshold value.
*/
class Consumer extends Thread
{
```

```
121.    /**
122.       Constructs a consumer thread.
123.       @param is the input stream
124.    */
125.    public Consumer(InputStream is)
126.    {
127.       in = new DataInputStream(is);
128.    }
129.
130.    public void run()
131.    {
132.       for(;;)
133.       {
134.          try
135.          {
136.             double x = in.readDouble();
137.             if (Math.abs(x - oldx) > THRESHOLD)
138.             {
139.                System.out.println(x);
140.                oldx = x;
141.             }
142.          }
143.          catch(IOException e)
144.          {
145.             System.out.println("Error: " + e);
146.          }
147.       }
148.    }
149.
150.    private double oldx = 0;
151.    private DataInputStream in;
152.    private static final double THRESHOLD = 0.01;
153. }
```

java.io.PipedInputStream

- `PipedInputStream()`
  creates a new piped input stream that is not yet connected to a piped output stream.
- `PipedInputStream(PipedOutputStream out)`
  creates a new piped input stream that reads its data from a piped output stream.

  Parameters: `out` the source of the data
- `void connect(PipedOutputStream out)`
  attaches a piped output stream from which the data will be read.

  Parameters: `out` the source of the data

java.io.PipedOutputStream

- `PipedOutputStream()`
  creates a new piped output stream that is not yet connected to a piped input stream.
- `PipedOutputStream(PipedInputStream in)`
  creates a new piped output stream that writes its data to a piped input stream.

  Parameters: `in` the destination of the data
- `void connect(PipedInputStream in)`
  attaches a piped input stream to which the data will be written.

  Parameters: `in` the destination of the data