Chapter 1

Multithreading

You are probably familiar with *multitasking* in your operating system: 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, the operating system is really doling out CPU time to each program, giving the impression of parallel activity. This resource distribution is possible because although you may think you are keeping the computer busy by, for example, entering data, much of the CPU's time will be idle.

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*. Older operating systems such as Windows 3.x and Mac OS 9 are cooperative multitasking systems, as are the operating systems on simple devices such as cell phones. UNIX/Linux, Windows NT/XP (and Windows 9x 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*.

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. However, shared variables make communication between threads more efficient and easier to program than interprocess communication. Moreover, on some operating systems, threads are more "lightweight" than processes—it takes less overhead to create and destroy individual threads than it does to launch new processes.

Multithreading is extremely useful in practice. For example, a browser should be able to simultaneously download multiple images. A web server needs to be able to serve concurrent requests. 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.

Multithreading changed dramatically in JDK 5.0, with the addition of a large number of classes and interfaces that provide high-quality implementations of the mechanisms that most application programmers will need. In this chapter, we explain the new features of JDK 5.0 as well as the classic synchronization mechanisms, and help you choose between them.

Fair warning: multithreading can get very complex. In this chapter, we cover all the tools that an application programmer is likely to need. However, for more intricate system-level programming, we suggest that you turn to a more advanced reference, such as *Concurrent Programming in Java* by Doug Lea [Addison-Wesley 1999].

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 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 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. 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 panel.

```
Ball ball = new Ball();
panel.add(ball);

for (int i = 1; i <= STEPS; i++)
{
   ball.move(panel.getBounds());
   panel.paint(panel.getGraphics());
   Thread.sleep(DELAY);
}
```

The static `sleep` method of the `Thread` class pauses for the given number of milliseconds.

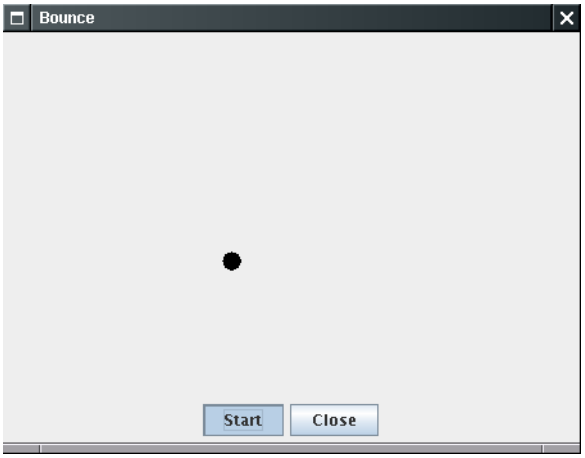

Figure 1–1: Using threads to animate bouncing balls

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 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 moves and click 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

```
panel.paint(panel.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 `panel.repaint()` in this program, you'll find that the panel is never repainted because the `addBall` method has completely taken over all processing. In the next program, in which 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 a Stop or Back button to interrupt the loading process. In the next section, we show you how to keep the user in control by running crucial parts of the code in a separate *thread*.

Example 1–1 shows the 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.setVisible(true);
17.    }
18. }
19.
20. /**
21.    A ball that moves and bounces off the edges of a
22.    rectangle
23. */
24. class Ball
25. {
26.    /**
27.       Moves the ball to the next position, reversing direction
28.       if it hits one of the edges
29.    */
30.    public void move(Rectangle2D bounds)
31.    {
32.       x += dx;
33.       y += dy;
34.       if (x < bounds.getMinX())
35.       {
36.          x = bounds.getMinX();
37.          dx = -dx;
38.       }
39.       if (x + XSIZE >= bounds.getMaxX())
40.       {
41.          x = bounds.getMaxX() - XSIZE;
42.          dx = -dx;
43.       }
44.       if (y < bounds.getMinY())
45.       {
46.          y = bounds.getMinY();
47.          dy = -dy;
48.       }
49.       if (y + YSIZE >= bounds.getMaxY())
50.       {
51.          y = bounds.getMaxY() - YSIZE;
52.          dy = -dy;
53.       }
54.    }
55.
56.    /**
57.       Gets the shape of the ball at its current position.
58.    */
59.    public Ellipse2D getShape()
```


```
60.    {
61.       return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
62.    }
63.
64.    private static final int XSIZE = 15;
65.    private static final int YSIZE = 15;
66.    private double x = 0;
67.    private double y = 0;
68.    private double dx = 1;
69.    private double dy = 1;
70. }
71.
72. /**
73.    The panel that draws the balls.
74. */
75. class BallPanel extends JPanel
76. {
77.    /**
78.       Add a ball to the panel.
79.       @param b the ball to add
80.    */
81.    public void add(Ball b)
82.    {
83.       balls.add(b);
84.    }
85.
86.    public void paintComponent(Graphics g)
87.    {
88.       super.paintComponent(g);
89.       Graphics2D g2 = (Graphics2D) g;
90.       for (Ball b : balls)
91.       {
92.          g2.fill(b.getShape());
93.       }
94.    }
95.
96.    private ArrayList<Ball> balls = new ArrayList<Ball>();
97. }
98.
99. /**
100.    The frame with panel and buttons.
101. */
102. class BounceFrame extends JFrame
103. {
104.    /**
105.       Constructs the frame with the panel for showing the
106.       bouncing ball and Start and Close buttons
107.    */
108.    public BounceFrame()
109.    {
110.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
111.       setTitle("Bounce");
112.
113.       panel = new BallPanel();
114.       add(panel, BorderLayout.CENTER);
```

```
115.    JPanel buttonPanel = new JPanel();
116.    addButton(buttonPanel, "Start",
117.       new ActionListener()
118.       {
119.          public void actionPerformed(ActionEvent event)
120.          {
121.             addBall();
122.          }
123.       });
124.
125.    addButton(buttonPanel, "Close",
126.       new ActionListener()
127.       {
128.          public void actionPerformed(ActionEvent event)
129.          {
130.             System.exit(0);
131.          }
132.       });
133.    add(buttonPanel, BorderLayout.SOUTH);
134. }
135.
136. /**
137.    Adds a button to a container.
138.    @param c the container
139.    @param title the button title
140.    @param listener the action listener for the button
141. */
142. public void addButton(Container c, String title, ActionListener listener)
143. {
144.    JButton button = new JButton(title);
145.    c.add(button);
146.    button.addActionListener(listener);
147. }
148.
149. /**
150.    Adds a bouncing ball to the panel and makes
151.    it bounce 1,000 times.
152. */
153. public void addBall()
154. {
155.    try
156.    {
157.       Ball ball = new Ball();
158.       panel.add(ball);
159.
160.       for (int i = 1; i <= STEPS; i++)
161.       {
162.          ball.move(panel.getBounds());
163.          panel.paint(panel.getGraphics());
164.          Thread.sleep(DELAY);
165.       }
166.    }
167.    catch (InterruptedException e)
168.    {
169.    }
```

```
170.    }
171.
172.    private BallPanel panel;
173.    public static final int DEFAULT_WIDTH = 450;
174.    public static final int DEFAULT_HEIGHT = 350;
175.    public static final int STEPS = 1000;
176.    public static final int DELAY = 3;
177. }
```

java.lang.Thread 1.0

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

  Parameters: `millis` The number of milliseconds to sleep

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. In fact, you will be able to launch multiple balls. Each of them is moved by its own thread. In addition, the AWT *event dispatch thread* continues running in parallel, taking care of user interface events. Because each thread gets a chance to run, the main thread has the opportunity to notice when a user clicks the Close button while the balls are bouncing. The thread can then process the "close" action.

Here is a simple procedure for running a task in a separate thread:

1. Place the code for the task into the `run` method of a class that implements the `Runnable` interface. That interface is very simple, with a single method:

   ```
   public interface Runnable
   {
      void run();
   }
   ```

   You simply implement a class, like this:

   ```
   class MyRunnable implements Runnable
   {
      public void run()
      {
         task code
      }
   }
   ```

2. Construct an object of your class:

   ```
   Runnable r = new MyRunnable();
   ```

3. Construct a `Thread` object from the `Runnable`:

   ```
   Thread t = new Thread(r);
   ```

4. Start the thread.

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

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

```
class BallRunnable implements Runnable
{
   . . .
```

```
   public void run()
   {
      try
      {
         for (int i = 1; i <= STEPS; i++)
         {
            ball.move(component.getBounds());
            component.repaint();
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException exception)
      {
      }
   }
   . . .
}
```

Again, we need to catch an `InterruptedException` that the `sleep` method threatens to throw. We discuss this exception in the next section. Typically, a thread is terminated by being interrupted. Accordingly, our `run` method exits when an `InterruptedException` occurs.

Whenever the Start button is clicked, the `addBall` method launches a new thread (see Figure 1–2):

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start();
```

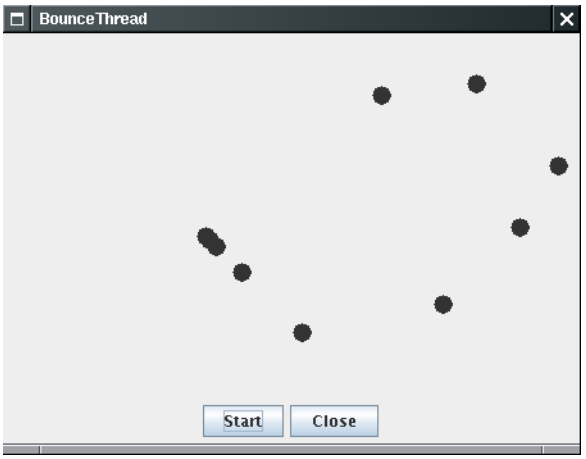

Figure 1–2: Running multiple threads

That's all there is to it! You now know how to run tasks in parallel. The remainder of this chapter tells you how to control the interaction between threads.

The complete code is shown in Example 1–2.

NOTE: You can also define a thread by forming a subclass of the `Thread` class, like this:

```
class MyThread extends Thread
{
   public void run()
   {
      task code
   }
}
```

Then you construct an object of the subclass and call its `start` method. However, this approach is no longer recommended. You should decouple the *task* that is to be run in parallel from the *mechanism* of running it. If you have many tasks, it is too expensive to create a separate thread for each one of them. Instead, you can use a thread pool—see page 59.

CAUTION: Do *not* call the `run` method of the `Thread` class or the `Runnable` object. Calling the `run` method directly merely executes the task in the *same* thread—no new thread is started. Instead, call the `Thread.start` method. It will create a new thread that executes the `run` method.

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.
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.setVisible(true);
17.    }
18. }
19.
20. /**
21.    A runnable that animates a bouncing ball.
22. */
23. class BallRunnable implements Runnable
24. {
25.    /**
26.       Constructs the runnable.
27.       @aBall the ball to bounce
28.       @aPanel the component in which the ball bounces
29.    */
30.    public BallRunnable(Ball aBall, Component aComponent)
31.    {
32.       ball = aBall;
33.       component = aComponent;
```

```
34.    }
35.
36.    public void run()
37.    {
38.       try
39.       {
40.          for (int i = 1; i <= STEPS; i++)
41.          {
42.             ball.move(component.getBounds());
43.             component.repaint();
44.             Thread.sleep(DELAY);
45.          }
46.       }
47.       catch (InterruptedException e)
48.       {
49.       }
50.    }
51.
52.    private Ball ball;
53.    private Component component;
54.    public static final int STEPS = 1000;
55.    public static final int DELAY = 5;
56. }
57.
58. /**
59.    A ball that moves and bounces off the edges of a
60.    rectangle
61. */
62. class Ball
63. {
64.    /**
65.       Moves the ball to the next position, reversing direction
66.       if it hits one of the edges
67.    */
68.    public void move(Rectangle2D bounds)
69.    {
70.       x += dx;
71.       y += dy;
72.       if (x < bounds.getMinX())
73.       {
74.          x = bounds.getMinX();
75.          dx = -dx;
76.       }
77.       if (x + XSIZE >= bounds.getMaxX())
78.       {
79.          x = bounds.getMaxX() - XSIZE;
80.          dx = -dx;
81.       }
82.       if (y < bounds.getMinY())
83.       {
84.          y = bounds.getMinY();
85.          dy = -dy;
86.       }
87.       if (y + YSIZE >= bounds.getMaxY())
88.       {
89.          y = bounds.getMaxY() - YSIZE;
```


```
90.            dy = -dy;
91.         }
92.      }
93.
94.      /**
95.         Gets the shape of the ball at its current position.
96.      */
97.      public Ellipse2D getShape()
98.      {
99.         return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
100.     }
101.
102.     private static final int XSIZE = 15;
103.     private static final int YSIZE = 15;
104.     private double x = 0;
105.     private double y = 0;
106.     private double dx = 1;
107.     private double dy = 1;
108. }
109.
110. /**
111.    The panel that draws the balls.
112. */
113. class BallPanel extends JPanel
114. {
115.    /**
116.       Add a ball to the panel.
117.       @param b the ball to add
118.    */
119.    public void add(Ball b)
120.    {
121.       balls.add(b);
122.    }
123.
124.    public void paintComponent(Graphics g)
125.    {
126.       super.paintComponent(g);
127.       Graphics2D g2 = (Graphics2D) g;
128.       for (Ball b : balls)
129.       {
130.          g2.fill(b.getShape());
131.       }
132.    }
133.
134.    private ArrayList<Ball> balls = new ArrayList<Ball>();
135. }
136.
137. /**
138.    The frame with panel and buttons.
139. */
140. class BounceFrame extends JFrame
141. {
142.    /**
143.       Constructs the frame with the panel for showing the
144.       bouncing ball and Start and Close buttons
145.    */
```

```
146.  public BounceFrame()
147.  {
148.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
149.     setTitle("BounceThread");
150.
151.     panel = new BallPanel();
152.     add(panel, BorderLayout.CENTER);
153.     JPanel buttonPanel = new JPanel();
154.     addButton(buttonPanel, "Start",
155.        new ActionListener()
156.        {
157.           public void actionPerformed(ActionEvent event)
158.           {
159.              addBall();
160.           }
161.        });
162.
163.     addButton(buttonPanel, "Close",
164.        new ActionListener()
165.        {
166.           public void actionPerformed(ActionEvent event)
167.           {
168.              System.exit(0);
169.           }
170.        });
171.     add(buttonPanel, BorderLayout.SOUTH);
172.  }
173.
174.  /**
175.     Adds a button to a container.
176.     @param c the container
177.     @param title the button title
178.     @param listener the action listener for the button
179.  */
180.  public void addButton(Container c, String title, ActionListener listener)
181.  {
182.     JButton button = new JButton(title);
183.     c.add(button);
184.     button.addActionListener(listener);
185.  }
186.
187.  /**
188.     Adds a bouncing ball to the canvas and starts a thread
189.     to make it bounce
190.  */
191.  public void addBall()
192.  {
193.     Ball b = new Ball();
194.     panel.add(b);
195.     Runnable r = new BallRunnable(b, panel);
196.     Thread t = new Thread(r);
197.     t.start();
198.  }
199.
200.  private BallPanel panel;
201.  public static final int DEFAULT_WIDTH = 450;
```


```
202.    public static final int DEFAULT_HEIGHT = 350;
203.    public static final int STEPS = 1000;
204.    public static final int DELAY = 3;
205. }
```

java.lang.Thread 1.0

- `Thread(Runnable target)`
  constructs a new thread that calls the `run()` method of the specified target.
- `void start()`
  starts this thread, causing the `run()` method to be called. This method will return immediately. The new thread runs concurrently.
- `void run()`
  calls the `run` method of the associated `Runnable`.

java.lang.Runnable 1.0

- `void run()`
  You must override this method and supply the instructions for the task that you want to have executed.

Interrupting Threads

A thread terminates when its `run` method returns. In JDK 1.0, there also was a `stop` method that another thread could call to terminate a thread. However, that method is now deprecated. We discuss the reason on page 46.

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.

When the `interrupt` method is called on a thread, the *interrupted status* of the thread is set. This is a Boolean flag that is present in every thread. Each thread should occasionally check whether it has been interrupted.

To find out whether the interrupted status was set, first call the static `Thread.currentThread` method to get the current thread and then call the `isInterrupted` method:

```
while (!Thread.currentThread().isInterrupted() && more work to do)
{
   do more work
}
```

However, if a thread is blocked, it cannot check the interrupted status. This is where the `InterruptedException` comes in. When the `interrupt` method is called on a blocked thread, 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. Some threads are so important that they should handle the exception and continue. 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 (!Thread.currentThread().isInterrupted() && more work to do)
      {
         do more work
      }
   }
   catch(InterruptedException e)
   {
      // thread was interrupted during sleep or wait
   }
   finally
   {
      cleanup, if required
   }
   // exiting the run method terminates the thread
}
```

The `isInterrupted` check is not necessary if you call the `sleep` method after every work iteration. The `sleep` method throws an `InterruptedException` if you call it when the interrupted status is set. Therefore, if your loop calls `sleep`, don't bother checking the interrupted status and simply catch the `InterruptedException`. Then your `run` method has the form

```
public void run()
{
   try
   {
      . . .
      while (more work to do)
      {
         do more work
         Thread.sleep(delay);
      }
   }
   catch(InterruptedException e)
   {
      // thread was interrupted during sleep or wait
   }
   finally
   {
      cleanup, if required
   }
   // exiting the run method terminates the thread
}
```

CAUTION: When the `sleep` method throws an `InterruptedException`, it also *clears* the interrupted status.

NOTE: 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. Furthermore, calling the `interrupted` method *clears* 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.

You'll find lots of published code in which the InterruptedException is squelched at a low level, like this:

```
void mySubTask()
{
   . . .
   try { sleep(delay); }
   catch (InterruptedException e) {} // DON'T IGNORE!
   . . .
}
```

Don't do that! If you can't think of anything good to do in the catch clause, you still have two reasonable choices:

- In the catch clause, call Thread.currentThread().interrupt() to set the interrupted status. Then the caller can test it.

```
void mySubTask()
{
   . . .
   try { sleep(delay); }
   catch (InterruptedException e) { Thread().currentThread().interrupt(); }
   . . .
}
```

- Or, even better, tag your method with throws InterruptedException and drop the try block. Then the caller (or, ultimately, the run method) can catch it.

```
void mySubTask() throws InterruptedException
{
   . . .
   sleep(delay);
   . . .
}
```

java.lang.Thread 1.0

- 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, then an InterruptedException is thrown.
- static boolean interrupted()
  tests whether 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 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 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 Thread(r)`—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 or may not actually be running. It is up to the operating system to give the thread time to run. (The Java specification 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 occasionally pause so that other threads have a chance to run. The details of thread scheduling depend on the services that the operating system provides. Preemptive scheduling systems give each runnable thread a slice of time to perform its task. When that slice of time is exhausted, the operating system *preempts* the thread and gives another thread an opportunity to work (see Figure 1–4 on page 27). When selecting the next thread, the operating system takes into account the thread *priorities*—see page 19 for more information on priorities.

All modern desktop and server operating systems use preemptive scheduling. However, small devices such as cell phones may use cooperative scheduling. In such a device, a thread loses control only when it calls a method such as `sleep` or `yield`.

On a machine with multiple processors, each processor can run a thread, and you can have multiple threads run in parallel. Of course, if there are more threads than processors, the scheduler still has to do 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.")

Blocked Threads

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

- The thread goes to sleep by calling the `sleep` method.
- 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.
- The thread tries to acquire a lock that is currently held by another thread. We discuss locks on page 27.
- The thread waits for a condition—see page 30.
- Someone calls the `suspend` method of the thread. However, this method is deprecated, and you should not call it in your code.

Figure 1–3 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 can

be 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.

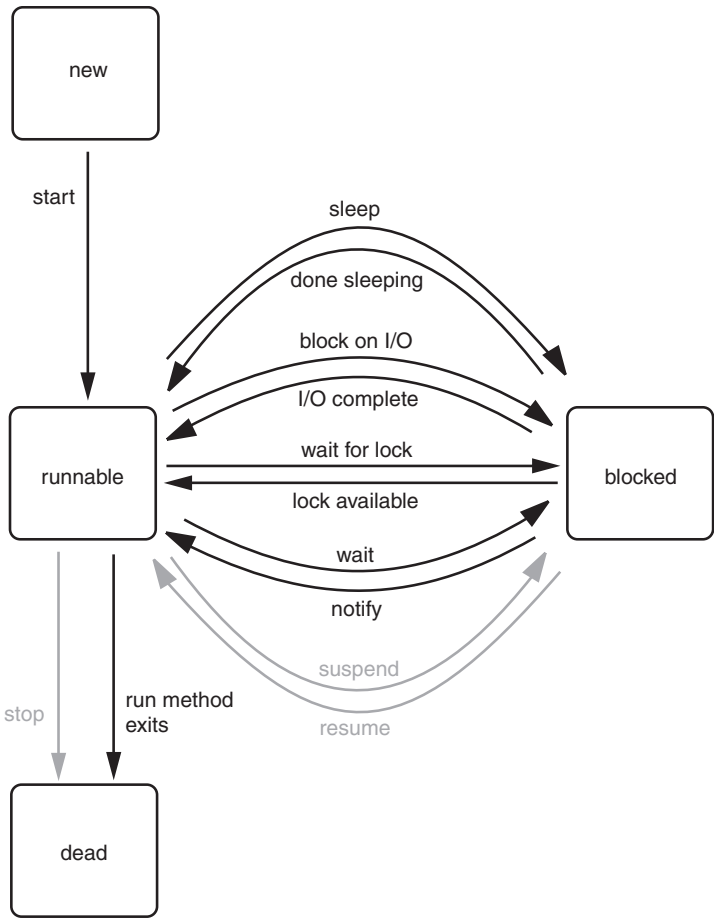

Figure 1–3: Thread states

A thread moves out of the blocked state and back into the runnable state by one of the following pathways.

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 is waiting for a lock that was owned by another thread, then the other thread must relinquish ownership of the lock. (It is also possible to wait with a timeout. Then the thread unblocks when the timeout elapses.)
4. If a thread waits for a condition, then another thread must signal that the condition may have changed. (If the thread waits with a timeout, then the thread is unblocked when the timeout elapses.)

5. If a thread has been suspended, then someone must call its `resume` method. However, because the `suspend` method has been deprecated, the `resume` method has been deprecated as well, and you should not call it in your own code.

A blocked thread can only reenter the runnable state through the same route that blocked it in the first place. In particular, you cannot simply call the `resume` method to unblock a blocking thread.

TIP: If you need to unblock an I/O operation, you should use the *channel* mechanism from the "new I/O" library. When another thread closes the channel, the blocked thread becomes runnable again, and the blocking operation throws a `ClosedChannelException`.

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, you can kill a thread by invoking its `stop` method. That method throws a `ThreadDeath` error object that kills the thread. However, the `stop` method is deprecated, and you should not call it in your own code.

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.

`java.lang.Thread` 1.0

- `boolean isAlive()`
  returns `true` if the thread has started and has not yet terminated.
- `void stop()`
  stops the thread. This method is deprecated.
- `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 join()`
  waits for the specified thread to die.
- `void join(long millis)`
  waits for the specified thread to die or for the specified number of milliseconds to pass.

Thread Properties

In the following sections, we discuss miscellaneous properties of threads: thread priorities, daemon threads, thread groups, and handlers for uncaught exceptions.

Thread Priorities

In the Java programming language, every thread has a *priority.* By default, a thread inherits the priority of its parent thread, that is, the thread that started it. 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 prefers threads with higher priority. However, thread priorities are *highly system dependent*. When the virtual machine relies on the thread implementation of the host platform, the Java thread priorities are mapped to the priority levels of the host platform, which may have more or fewer thread priority levels.

For example, Windows NT/XP has seven priority levels. Some of the Java priorities will map to the same operating system level. In the Sun JVM for Linux, thread priorities are ignored altogether—all threads have the same priority.

Thus, it is best to treat thread priorities only as hints to the scheduler. You should never structure your programs so that their correct functioning depends on priority levels.

CAUTION: If you do use priorities, you should be aware of a common beginner's error. If you have several threads with a high priority that rarely block, the lower-priority threads may *never* execute. Whenever the scheduler decides to run a new thread, it will choose among the highest-priority threads first, even though that may starve the lower-priority threads completely.

`java.lang.Thread` 1.0

- `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 with a priority at least as high as the priority of this thread, they will be scheduled next. Note that this is a static method.

Daemon Threads

You can turn a thread 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, the virtual machine exits. There is no point in keeping the program running if all remaining threads are daemons.

java.lang.Thread 1.0

- `void setDaemon(boolean isDaemon)`
  marks this thread as a daemon thread or a user thread. This method must be called before the thread is started.

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 these threads simultaneously. The Java programming language lets you construct what it calls a *thread group* so that 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
```

However, executors let you achieve the same task without requiring the use of thread groups—see page 63.

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.

java.lang.Thread 1.0

- `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.

java.lang.ThreadGroup 1.0

- `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.

Handlers for Uncaught Exceptions

The `run` method of a thread cannot throw any checked exceptions, but it can be terminated by an unchecked exception. In that case, the thread dies.

However, there is no `catch` clause to which the exception can be propagated. Instead, just before the thread dies, the exception is passed to a handler for uncaught exceptions.

The handler must belong to a class that implements the `Thread.UncaughtExceptionHandler` interface. That interface has a single method,

```
void uncaughtException(Thread t, Throwable e)
```

As of JDK 5.0, you can install a handler into any thread with the `setUncaughtExceptionHandler` method. You can also install a default handler for all threads with the static method `setDefaultUncaughtExceptionHandler` of the `Thread` class. A replacement handler might use the logging API to send reports of uncaught exceptions into a log file.

If you don't install a default handler, the default handler is `null`. However, if you don't install a handler for an individual thread, the handler is the thread's `ThreadGroup` object.

The `ThreadGroup` class implements the `Thread.UncaughtExceptionHandler` interface. Its `uncaughtException` method takes the following action:

1. If the thread group has a parent, then the `uncaughtException` method of the parent group is called.
2. Otherwise, if the `Thread.getDefaultExceptionHandler` method returns a non-`null` handler, it is called.
3. Otherwise, if the `Throwable` is an instance of `ThreadDeath`, nothing happens.
4. Otherwise, the name of the thread and the stack trace of the `Throwable` are printed on `System.err`.

That is the stack trace that you have undoubtedly seen many times in your programs.

NOTE: Prior to JDK 5.0, you could not install a handler for uncaught exceptions into each thread, nor could you specify a default handler. To install a handler, you needed to subclass the `ThreadGroup` class and override the `uncaughtException` method.

java.lang.Thread 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` **5.0**
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` **5.0**

  set or get the default handler for uncaught exceptions.
- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` **5.0**
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` **5.0**

  set or get the handler for uncaught exceptions. If no handler is installed, the thread group object is the handler.

java.lang.Thread.UncaughtExceptionHandler 5.0

- `void uncaughtException(Thread t, Throwable e)`

  Define this method to log a custom report when a thread is terminated with an uncaught exception.

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

  `e` The uncaught exception object

java.lang.ThreadGroup 1.0

- `void uncaughtException(Thread t, Throwable e)`

  calls this method of the parent thread group if there is a parent, or calls the default handler of the `Thread` class if there is a default handler, 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.)

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 can step on each other's toes. Depending on the order in which the data were accessed, corrupted objects can result. Such a situation is often called a *race condition*.

An Example of a Race Condition

To avoid corruption of shared data 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 data access.

In the next test program, we simulate a bank with a number of accounts. We randomly generate transactions that move money between these accounts. Each account has one thread. 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
{
   System.out.print(Thread.currentThread());
   accounts[from] -= amount;
```


```
      System.out.printf(" %10.2f from %d to %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
   }
```

Here is the code for the `TransferRunnable` 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 TransferRunnable implements Runnable
{
   . . .
   public void run()
   {
      try
      {
         int toAccount = (int) (bank.size() * Math.random());
         double amount = maxAmount * Math.random();
         bank.transfer(fromAccount, toAccount, amount);
         Thread.sleep((int) (DELAY * Math.random()));
      }
      catch(InterruptedException e) {}
   }
}
```

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 because all we do is move money from one account to another.

At the end of each transaction, the `transfer` method recomputes the total and prints it.

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

Here is a typical printout:

```
. . .
Thread[Thread-11,5,main]     588.48 from 11 to 44 Total Balance:  100000.00
Thread[Thread-12,5,main]     976.11 from 12 to 22 Total Balance:  100000.00
Thread[Thread-14,5,main]     521.51 from 14 to 22 Total Balance:  100000.00
Thread[Thread-13,5,main]     359.89 from 13 to 81 Total Balance:  100000.00
. . .
Thread[Thread-36,5,main]     401.71 from 36 to 73 Total Balance:   99291.06
Thread[Thread-35,5,main]     691.46 from 35 to 77 Total Balance:   99291.06
Thread[Thread-37,5,main]      78.64 from 37 to 3 Total Balance:   99291.06
Thread[Thread-34,5,main]     197.11 from 34 to 69 Total Balance:   99291.06
Thread[Thread-36,5,main]      85.96 from 36 to 4 Total Balance:   99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main]       7.31 from 31 to 32 Total Balance:   99979.24
     627.50 from 4 to 5 Total Balance:   99979.24
. . .
```

As you can see, something is very wrong. For a few transactions, the bank balance remains at $100,000, which is the correct total for 100 accounts of $1,000 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 in this bank.

Example 1–3 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 1–3: UnsynchBankTest.java

```
1. /**
2.    This program shows data corruption when multiple threads access a data structure.
3. */
4. public class UnsynchBankTest
5. {
6.    public static void main(String[] args)
7.    {
8.       Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
9.       int i;
10.      for (i = 0; i < NACCOUNTS; i++)
11.      {
12.         TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
13.         Thread t = new Thread(r);
14.         t.start();
15.      }
16.   }
17.
18.   public static final int NACCOUNTS = 100;
19.   public static final double INITIAL_BALANCE = 1000;
20. }
21.
22. /**
23.    A bank with a number of bank accounts.
24. */
25. class Bank
26. {
27.    /**
28.       Constructs the bank.
29.       @param n the number of accounts
30.       @param initialBalance the initial balance
31.       for each account
32.    */
33.    public Bank(int n, double initialBalance)
34.    {
35.       accounts = new double[n];
36.       for (int i = 0; i < accounts.length; i++)
37.          accounts[i] = initialBalance;
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, double amount)
47.    {
48.       if (accounts[from] < amount) return;
49.       System.out.print(Thread.currentThread());
50.       accounts[from] -= amount;
51.       System.out.printf(" %10.2f from %d to %d", amount, from, to);
52.       accounts[to] += amount;
53.       System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
54.    }
```


```
55.
56.    /**
57.       Gets the sum of all account balances.
58.       @return the total balance
59.    */
60.    public double getTotalBalance()
61.    {
62.       double sum = 0;
63.
64.       for (double a : accounts)
65.          sum += a;
66.
67.       return 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.    private final double[] accounts;
80. }
81.
82. /**
83.    A runnable that transfers money from an account to other
84.    accounts in a bank.
85. */
86. class TransferRunnable implements Runnable
87. {
88.    /**
89.       Constructs a transfer runnable.
90.       @param b the bank between whose account money is transferred
91.       @param from the account to transfer money from
92.       @param max the maximum amount of money in each transfer
93.    */
94.    public TransferRunnable(Bank b, int from, double max)
95.    {
96.       bank = b;
97.       fromAccount = from;
98.       maxAmount = max;
99.    }
100.
101.   public void run()
102.   {
103.      try
104.      {
105.         while (true)
106.         {
107.            int toAccount = (int) (bank.size() * Math.random());
108.            double amount = maxAmount * Math.random();
109.            bank.transfer(fromAccount, toAccount, amount);
110.            Thread.sleep((int) (DELAY * Math.random()));
```

```
111.          }
112.       }
113.       catch (InterruptedException e) {}
114.    }
115.
116.    private Bank bank;
117.    private int fromAccount;
118.    private double maxAmount;
119.    private int DELAY = 10;
120. }
```

The Race Condition Explained

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 1–4.)

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!)

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        #2; //Field accounts:[D
iload_2
dup2
daload
dload_3
dadd
dastore
```

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.

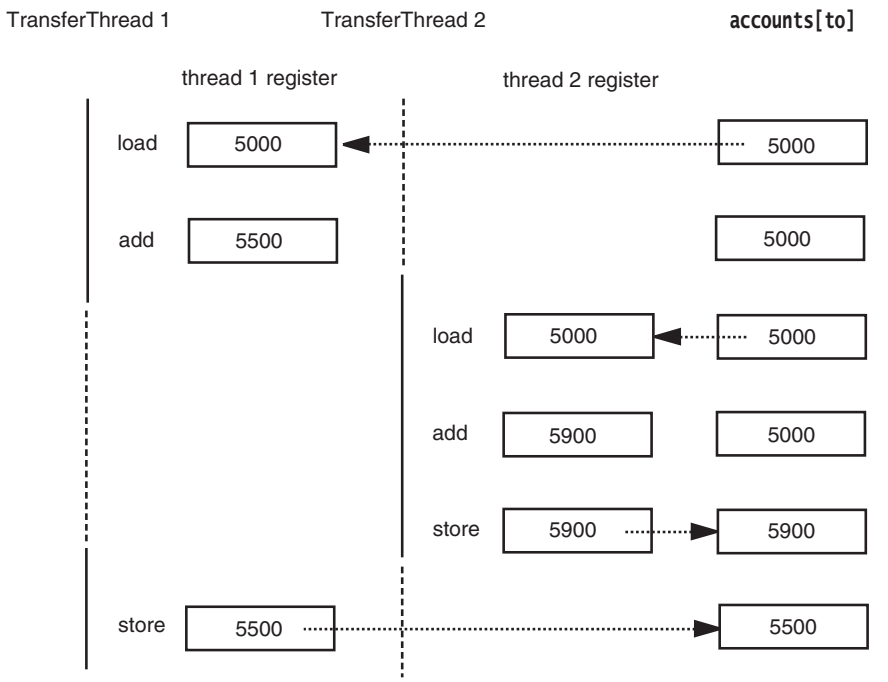

Figure 1–4: Simultaneous access by two threads

What is the chance of this corruption occurring? We boosted the chance of observing the problem by interleaving the print statements with the statements that update the balance.

If you omit the print statements, the risk of corruption is quite a bit lower because each thread does so little work before going to sleep again, and it is unlikely that the scheduler will preempt it in the middle of the computation. However, the risk of corruption does not completely go away. If you run lots of threads on a heavily loaded machine, then the program will still fail even after you have eliminated the print statements. The failure may take a few minutes or hours or days to occur. Frankly, there are few things worse in the life of a programmer than an error that only manifests itself once every few days.

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 never be corrupted.

Lock Objects

Starting with JDK 5.0, there are two mechanisms for protecting a code block from concurrent access. Earlier versions of Java used the `synchronized` keyword for this purpose, and JDK 5.0 introduces the `ReentrantLock` class. The `synchronized` keyword automatically provides a lock as well as an associated "condition." We believe that it is easier to understand the `synchronized` keyword after you have seen locks and conditions in isolation. JDK 5.0 provides separate classes for these fundamental mechanisms, which we explain here and on page 30. We will discuss the `synchronized` keyword on page 35.

The basic outline for protecting a code block with a `ReentrantLock` is:

```
myLock.lock(); // a ReentrantLock object
try
{
   critical section
}
finally
{
   myLock.unlock(); // make sure the lock is unlocked even if an exception is thrown
}
```

This construct guarantees that only one thread at a time can enter the critical section. As soon as one thread locks the lock object, no other thread can get past the `lock` statement. When other threads call `lock`, they are blocked until the first thread unlocks the lock object.

Let us use a lock to protect the `transfer` method of the `Bank` class.

```
public class Bank
{
   public void transfer(int from, int to, int amount)
   {
      bankLock.lock();
      try
      {
         if (accounts[from] < amount) return;
         System.out.print(Thread.currentThread());
         accounts[from] -= amount;
         System.out.printf(" %10.2f from %d to %d", amount, from, to);
         accounts[to] += amount;
         System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
      }
      finally
      {
         bankLock.unlock();
      }
   }
   . . .
   private Lock bankLock = new ReentrantLock(); // ReentrantLock implements the Lock interface
}
```

Suppose one thread calls `transfer` and gets preempted before it is done. Suppose a second thread also calls `transfer`. The second thread cannot acquire the lock and is blocked in the call to the `lock` method. It is deactivated and must wait for the first thread to finish executing the `transfer` method. When the first thread unlocks the lock, then the second thread can proceed (see Figure 1–5).

Try it out. Add the locking code to the `transfer` method and run the program again. You can run it forever, and the bank balance will not become corrupted.

Note that each `Bank` object has its own `ReentrantLock` object. If two threads try to access the same `Bank` object, then the lock serves to serialize the access. However, if two threads access different `Bank` objects, then each thread acquires a different lock and neither thread is blocked. This is as it should be, because the threads cannot interfere with another when they manipulate different `Bank` instances.

The lock is called *reentrant* because a thread can repeatedly acquire a lock that it already owns. The lock keeps a *hold count* that keeps track of the nested calls to the `lock` method. The thread has to call `unlock` for every call to `lock` in order to relinquish the lock. Because of this feature, code that is protected by a lock can call another method that uses the same locks.

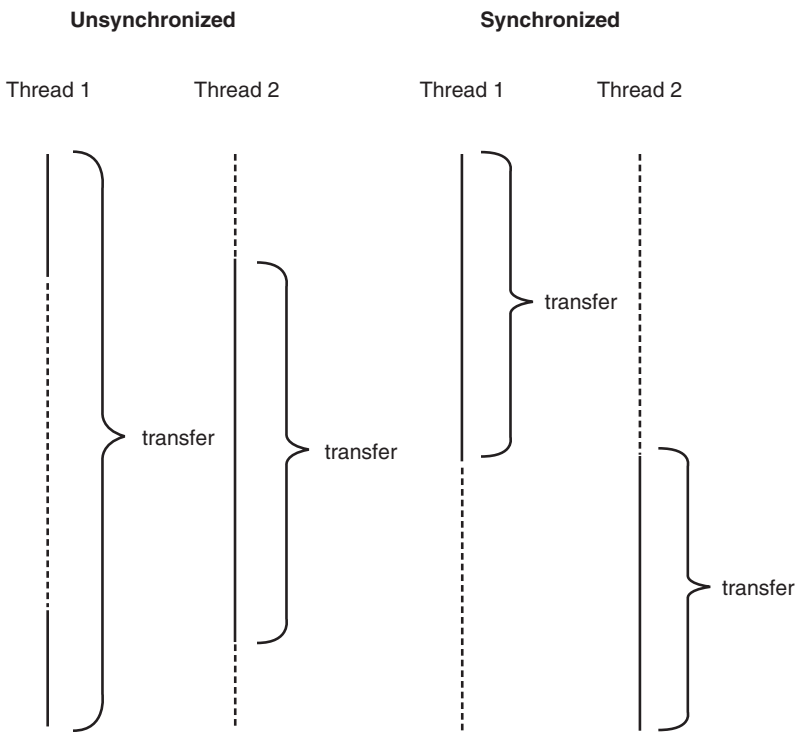

Figure 1–5: Comparison of unsynchronized and synchronized threads

For example, the `transfer` method calls the `getTotalBalance` method, which also locks the `bankLock` object, which now has a hold count of 2. When the `getTotalBalance` method exits, the hold count is back to 1. When the `transfer` method exits, the hold count is 0, and the thread relinquishes the lock.

In general, you will want to protect blocks of code that require multiple operations to update or inspect a data structure. You are then assured that these operations run to completion before another thread can use the same object.

CAUTION: You need to be careful that code in a critical section is not bypassed through the throwing of an exception. If an exception is thrown before the end of the section, then the `finally` clause will relinquish the lock but the object may be in a damaged state.

***java.util.concurrent.locks.Lock* 5.0**

- `void lock()`
  acquires this lock; blocks if the lock is currently owned by another thread.
- `void unlock()`
  releases this lock.

java.util.concurrent.locks.ReentrantLock 5.0

- `ReentrantLock()`
  constructs a reentrant lock that can be used to protect a critical section.

Condition Objects

Often, a thread enters a critical section, only to discover that it can't proceed until a condition is fulfilled. You use a *condition object* to manage threads that have acquired a lock but cannot do useful work. In this section, we introduce the implementation of condition objects in the Java library. (For historical reasons, condition objects are often called *condition variables*.)

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 protecting both the test and the transfer action with a lock:

```
public void transfer(int from, int to, int amount)
{
   bankLock.lock();
   try
   {
      while (accounts[from] < amount)
      {
         // wait
         . . .
      }
      // transfer funds
      . . .
   }
   finally
   {
      bankLock.unlock();
   }
}
```

Now, what do we do when there is not enough money in the account? We wait until some other thread has added funds. But this thread has just gained exclusive access to the `bankLock`, so no other thread has a chance to make a deposit. This is where condition objects come in.

A lock object can have one or more associated condition objects. You obtain a condition object with the `newCondition` method. It is customary to give each condition object a name that evokes the condition that it represents. For example, here we set up a condition object to represent the "sufficient funds" condition.

```
class Bank
{
   public Bank()
   {
```


```
      . . .
      sufficientFunds = bankLock.newCondition();
   }
   . . .
   private Condition sufficientFunds;
}
```

If the `transfer` method finds that sufficient funds are not available, it calls

```
sufficientFunds.await();
```

The current thread is now blocked and gives up the lock. This lets in another thread that can, we hope, increase the account balance.

There is an essential difference between a thread that is waiting to acquire a lock and a thread that has called `await`. Once a thread calls the `await` method, it enters a *wait set* for that condition. The thread is *not* unblocked when the lock is available. Instead, it stays blocked until another thread has called the `signalAll` method on the same condition.

When another thread transfers money, then it should call

```
sufficientFunds.signalAll();
```

This call unblocks all threads that are waiting for the condition. When the threads are removed from the wait set, 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 lock is available, one of them will acquire the lock *and continue where it left off*, returning from the call to `await`.

At this time, the thread should test the condition again. There is no guarantee that the condition is now fulfilled—the `signalAll` method merely signals to the waiting threads that it *may be* fulfilled at this time and that it is worth checking for the condition again.

NOTE: In general, a call to `await` should always be inside a loop of the form

```
while (!(ok to proceed))
   condition.await();
```

It is crucially important that *some* other thread calls the `signalAll` method eventually. When a thread calls `await`, 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 `await` without unblocking one of the others, then it also blocks. No thread is left to unblock the others, and the program hangs.

When should you call `signalAll`? The rule of thumb is to call `signalAll` 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 call `signalAll` when we have finished the funds transfer.

```
public void transfer(int from, int to, int amount)
{
   bankLock.lock();
   try
   {
      while (accounts[from] < amount)
         sufficientFunds.await();
      // transfer funds
      . . .
```

```
            sufficientFunds.signalAll();
        }
        finally
        {
            bankLock.unlock();
        }
    }
```

Note that the call to `signalAll` 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.

Another method, `signal`, unblocks only a single thread from the wait set, chosen at random. That is more efficient than unblocking all threads, but there is a danger. If the randomly chosen thread finds that it still cannot proceed, then it becomes blocked again. If no other thread calls `signal` again, then the system deadlocks.

CAUTION: A thread can only call `await`, `signalAll`, or `signal` on a condition when it owns the lock of the condition.

If you run the sample program in Example 1–4, you will notice that nothing ever goes wrong. The total balance stays at $100,000 forever. No account ever has a negative balance. (Again, you need to press CTRL+C to terminate the program.) You may also notice that the program runs a bit slower—this is the price you pay for the added bookkeeping involved in the synchronization mechanism.

Example 1–4: SynchBankTest.java

```
1.  import java.util.concurrent.locks.*;
2.
3.  /**
4.     This program shows how multiple threads can safely access a data structure.
5.  */
6.  public class SynchBankTest
7.  {
8.     public static void main(String[] args)
9.     {
10.       Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
11.       int i;
12.       for (i = 0; i < NACCOUNTS; i++)
13.       {
14.          TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
15.          Thread t = new Thread(r);
16.          t.start();
17.       }
18.    }
19.
20.    public static final int NACCOUNTS = 100;
21.    public static final double INITIAL_BALANCE = 1000;
22. }
23.
24. /**
25.    A bank with a number of bank accounts.
26. */
27. class Bank
28. {
```


```
29.    /**
30.       Constructs the bank.
31.       @param n the number of accounts
32.       @param initialBalance the initial balance
33.       for each account
34.    */
35.    public Bank(int n, double initialBalance)
36.    {
37.       accounts = new double[n];
38.       for (int i = 0; i < accounts.length; i++)
39.          accounts[i] = initialBalance;
40.       bankLock = new ReentrantLock();
41.       sufficientFunds = bankLock.newCondition();
42.    }
43.
44.    /**
45.       Transfers money from one account to another.
46.       @param from the account to transfer from
47.       @param to the account to transfer to
48.       @param amount the amount to transfer
49.    */
50.    public void transfer(int from, int to, double amount)
51.       throws InterruptedException
52.    {
53.       bankLock.lock();
54.       try
55.       {
56.          while (accounts[from] < amount)
57.             sufficientFunds.await();
58.          System.out.print(Thread.currentThread());
59.          accounts[from] -= amount;
60.          System.out.printf(" %10.2f from %d to %d", amount, from, to);
61.          accounts[to] += amount;
62.          System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
63.          sufficientFunds.signalAll();
64.       }
65.       finally
66.       {
67.          bankLock.unlock();
68.       }
69.    }
70.
71.    /**
72.       Gets the sum of all account balances.
73.       @return the total balance
74.    */
75.    public double getTotalBalance()
76.    {
77.       bankLock.lock();
78.       try
79.       {
80.          double sum = 0;
81.
82.          for (double a : accounts)
83.             sum += a;
```

```
84.
85.          return sum;
86.       }
87.       finally
88.       {
89.          bankLock.unlock();
90.       }
91.    }
92.
93.    /**
94.       Gets the number of accounts in the bank.
95.       @return the number of accounts
96.    */
97.    public int size()
98.    {
99.       return accounts.length;
100.   }
101.
102.   private final double[] accounts;
103.   private Lock bankLock;
104.   private Condition sufficientFunds;
105. }
106.
107. /**
108.    A runnable that transfers money from an account to other
109.    accounts in a bank.
110. */
111. class TransferRunnable implements Runnable
112. {
113.    /**
114.       Constructs a transfer runnable.
115.       @param b the bank between whose account money is transferred
116.       @param from the account to transfer money from
117.       @param max the maximum amount of money in each transfer
118.    */
119.    public TransferRunnable(Bank b, int from, double max)
120.    {
121.       bank = b;
122.       fromAccount = from;
123.       maxAmount = max;
124.    }
125.
126.    public void run()
127.    {
128.       try
129.       {
130.          while (true)
131.          {
132.             int toAccount = (int) (bank.size() * Math.random());
133.             double amount = maxAmount * Math.random();
134.             bank.transfer(fromAccount, toAccount, amount);
135.             Thread.sleep((int) (DELAY * Math.random()));
136.          }
137.       }
138.       catch (InterruptedException e) {}
```


```
139.    }
140.
141.    private Bank bank;
142.    private int fromAccount;
143.    private double maxAmount;
144.    private int repetitions;
145.    private int DELAY = 10;
146. }
```

java.util.concurrent.locks.Lock 5.0

- `Condition newCondition()`
  returns a condition object that is associated with this lock.

java.util.concurrent.locks.Condition 5.0

- `void await()`
  puts this thread on the wait set for this condition.
- `void signalAll()`
  unblocks all threads in the wait set for this condition.
- `void signal()`
  unblocks one randomly selected thread in the wait set for this condition.

The synchronized Keyword

In the preceding sections, you saw how to use `Lock` and `Condition` objects. Before going any further, let us summarize the key points about locks and conditions:

- A lock protects sections of code, allowing only one thread to execute the code at a time.
- A lock manages threads that are trying to enter a protected code segment.
- A lock can have one or more associated condition objects.
- Each condition object manages threads that have entered a protected code section but that cannot proceed.

Before the `Lock` and `Condition` interfaces were added to JDK 5.0, the Java language used a different concurrency mechanism. Ever since version 1.0, *every object* in Java has an implicit lock. If a method is declared with the `synchronized` keyword, then the object's lock protects the entire method. That is, to call the method, a thread must acquire the object lock.

In other words,

```
public synchronized void method()
{
   method body
}
```

is the equivalent of

```
public void method()
{
   implicitLock.lock();
   try
   {
      method body
   }
   finally { implicitLock.unlock(); }
}
```

For example, instead of using an explicit lock, we can simply declare the `transfer` method of the `Bank` class as `synchronized`.

The implicit object lock has a single associated condition. The `wait` method adds a thread to the wait set, and the `notifyAll`/`notify` methods unblock waiting threads. In other words, calling `wait` or `notifyAll` is the equivalent of

implicitCondition`.await();`
implicitCondition`.signalAll();`

NOTE: The `wait` and `signal` methods belong to the `Object` class. The `Condition` methods had to be named `await` and `signalAll` so that they don't conflict with the final methods `wait` and `notifyAll` methods of the `Object` class.

For example, you can implement the `Bank` class in Java like this:

```
class Bank
{
   public synchronized void transfer(int from, int to, int amount) throws InterruptedException
   {
      while (accounts[from] < amount)
         wait(); // wait on object lock's single condition
      accounts[from] -= amount;
      accounts[to] += amount;
      notifyAll(); // notify all threads waiting on the condition
   }
   public synchronized double getTotalBalance() { . . . }
   private double accounts[];
}
```

As you can see, using the `synchronized` keyword yields code that is much more concise. Of course, to understand this code, you have to know that each object has an implicit lock, and that the lock has an implicit condition. The lock manages the threads that try to enter a `synchronized` method. The condition manages the threads that have called `wait`.

However, the implicit locks and conditions have some limitations. Among them are:

- You cannot interrupt a thread that is trying to acquire a lock.
- You cannot specify a timeout when trying to acquire a lock.
- Having a single condition per lock can be inefficient.
- The virtual machine locking primitives do not map well to the most efficient locking mechanisms available in hardware.

What should you use in your code—`Lock` and `Condition` objects or synchronized methods? Here is our recommendation:

1. It is best to use neither `Lock`/`Condition` nor the `synchronized` keyword. In many situations, you can use one of the mechanisms of the `java.util.concurrent` package that do all the locking for you. For example, on page 48, you will see how to use a blocking queue to synchronize threads that work on a common task.
2. If the `synchronized` keyword works for your situation, by all means, use it. You write less code and have less room for error. Example 1–5 shows the bank example, implemented with synchronized methods.
3. Use `Lock`/`Condition` if you specifically need the additional power that these constructs give you.

NOTE: At least for now, using the `synchronized` keyword has an added benefit. Tools that monitor the virtual machine can report on the implicit locks and conditions, which is helpful for debugging deadlock problems. It will take some time for these tools to be extended to the `java.util.concurrent` mechanisms.

Example 1–5: SynchBankTest2.java

```
1. /**
2.    This program shows how multiple threads can safely access a data structure, using
3.    synchronized methods.
4. */
5. public class SynchBankTest2
6. {
7.    public static void main(String[] args)
8.    {
9.       Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
10.      int i;
11.      for (i = 0; i < NACCOUNTS; i++)
12.      {
13.         TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
14.         Thread t = new Thread(r);
15.         t.start();
16.      }
17.   }
18.
19.   public static final int NACCOUNTS = 100;
20.   public static final double INITIAL_BALANCE = 1000;
21. }
22.
23. /**
24.    A bank with a number of bank accounts.
25. */
26. class Bank
27. {
28.    /**
29.       Constructs the bank.
30.       @param n the number of accounts
31.       @param initialBalance the initial balance
32.       for each account
33.    */
34.    public Bank(int n, double initialBalance)
35.    {
36.       accounts = new double[n];
37.       for (int i = 0; i < accounts.length; i++)
38.          accounts[i] = initialBalance;
39.    }
40.
41.    /**
42.       Transfers money from one account to another.
43.       @param from the account to transfer from
44.       @param to the account to transfer to
45.       @param amount the amount to transfer
46.    */
```

```
47.    public synchronized void transfer(int from, int to, double amount)
48.       throws InterruptedException
49.    {
50.       while (accounts[from] < amount)
51.          wait();
52.       System.out.print(Thread.currentThread());
53.       accounts[from] -= amount;
54.       System.out.printf(" %10.2f from %d to %d", amount, from, to);
55.       accounts[to] += amount;
56.       System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
57.       notifyAll();
58.    }
59.
60.    /**
61.       Gets the sum of all account balances.
62.       @return the total balance
63.    */
64.    public synchronized double getTotalBalance()
65.    {
66.       double sum = 0;
67.
68.       for (double a : accounts)
69.          sum += a;
70.
71.       return sum;
72.    }
73.
74.    /**
75.       Gets the number of accounts in the bank.
76.       @return the number of accounts
77.    */
78.    public int size()
79.    {
80.       return accounts.length;
81.    }
82.
83.    private final double[] accounts;
84. }
85.
86. /**
87.    A runnable that transfers money from an account to other
88.    accounts in a bank.
89. */
90. class TransferRunnable implements Runnable
91. {
92.    /**
93.       Constructs a transfer runnable.
94.       @param b the bank between whose account money is transferred
95.       @param from the account to transfer money from
96.       @param max the maximum amount of money in each transfer
97.    */
98.    public TransferRunnable(Bank b, int from, double max)
99.    {
100.      bank = b;
```


```
101.       fromAccount = from;
102.       maxAmount = max;
103.    }
104.
105.    public void run()
106.    {
107.       try
108.       {
109.          while (true)
110.          {
111.             int toAccount = (int) (bank.size() * Math.random());
112.             double amount = maxAmount * Math.random();
113.             bank.transfer(fromAccount, toAccount, amount);
114.             Thread.sleep((int) (DELAY * Math.random()));
115.          }
116.       }
117.       catch (InterruptedException e) {}
118.    }
119.
120.    private Bank bank;
121.    private int fromAccount;
122.    private double maxAmount;
123.    private int repetitions;
124.    private int DELAY = 10;
125. }
```

java.lang.Object 1.0

- `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.
- `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. These methods can only be called from within a synchronized method. They throw 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

Monitors

The locks and conditions are powerful tools for thread synchronization, but they are not very object oriented. For many years, researchers have looked for ways to make multithreading safe without forcing programmers to think about explicit locks. One of the most successful solutions is the *monitor* concept that was pioneered by Per Brinch Hansen and Tony Hoare in the 1970s. In the terminology of Java, a monitor has these properties:

- A monitor is a class with only private fields.
- Each object of that class has an associated lock.
- All methods are locked by that lock. In other words, if a client calls `obj.method()`, then the lock for `obj` is automatically acquired at the beginning of the method call and relinquished when the method returns. Because all fields are private, this arrangement ensures that no thread can access the fields while another thread manipulates them.
- The lock can have any number of associated conditions.

Earlier versions of monitors had a single condition, with a rather elegant syntax. You can simply call `await accounts[from] >= balance` without using an explicit condition variable. However, research showed that indiscriminate retesting of conditions can be inefficient. This problem is solved with explicit condition variables, each managing a separate set of threads.

The Java designers loosely adapted the monitor concept. *Every object* in Java has an implicit lock and an implicit condition. If a method is declared with the `synchronized` keyword, then it acts like a monitor method. The condition variable is accessed by calling `wait/notify/notifyAll`.

However, a Java object differs from a monitor in two important ways, compromising its security:

- Fields are not required to be `private`.
- Methods are not required to be `synchronized`.

This disrespect for security enraged Per Brinch Hansen. In a scathing review of the multithreading primitives in Java, he wrote: "It is astounding to me that Java's insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit." [Java's Insecure Parallelism, *ACM SIGPLAN Notices* 34:38–45, April 1999]

Synchronized Blocks

However, a Java object differs from a monitor in three ways:

- Fields are not required to be `private`.
- Methods are not required to be `synchronized`.
- The lock has only one condition.

If you deal with legacy code, you need to know something about the built-in synchronization primitives. Recall that each object has a lock. A thread can acquire the lock in one of two ways, by calling a synchronized method or by entering a *synchronized block*. If the thread calls `obj.method()`, it acquires the lock for `obj`. Similarly, if a thread enters a block of the form

```
synchronized (obj) // this is the syntax for a synchronized block
{
   critical section
}
```

then the thread acquires the lock for `obj`. The lock is reentrant. If a thread has acquired the lock, it can acquire it again, incrementing the hold count. In particular, a synchronized method can call other synchronized methods with the same implicit parameter without having to wait for the lock.

You will often find "ad hoc" locks in legacy code, such as

```
class Bank
{
   public void transfer(int from, int to, int amount)
   {
      synchronized (lock) // an ad-hoc lock
      {
         accounts[from] -= amount;
         accounts[to] += amount;
      }
      System.out.println(. . .);
   }
   . . .
   private double accounts[];
   private Object lock = new Object(); }
```

Here, the `lock` object is created only to use the lock that every Java object possesses.

It is legal to declare static methods as synchronized. If such a method is called, it acquires the lock of the associated class object. For example, if the `Bank` class has a static synchronized method, then the lock of the `Bank.class` object is locked when it is called.

Volatile Fields

Sometimes, it seems excessive to pay the cost of synchronization just to read or write an instance field or two. After all, what can go wrong? Unfortunately, with modern processors and compilers, there is plenty of room for error:

- Computers with multiple processors can temporarily hold memory values in registers or local memory caches. As a consequence, threads running in different processors may see different values for the same memory location!
- Compilers can reorder instructions for maximum throughput. Compilers won't choose an ordering that changes the meaning of the code, but they make the assumption that memory values are only changed when there are explicit instructions in the code. However, a memory value can be changed by another thread!

If you use locks to protect code that can be accessed by multiple threads, then you won't have these problems. Compilers are required to respect locks by flushing local caches as necessary and not inappropriately reordering instructions. The details are explained in the Java Memory Model and Thread Specification developed by JSR 133 (see `http://www.jcp.org/en/jsr/detail?id=133`). Much of the specification is highly complex and technical, but the document also contains a number of clearly explained examples. A more accessible overview article by Brian Goetz is available at `http://www-106.ibm.com/developerworks/java/library/j-jtp02244.html`.

NOTE: Brian Goetz coined the following "synchronization motto": "If you write a variable which may next be read by another thread, or you read a variable which may have last been written by another thread, you must use synchronization."

The `volatile` keyword offers a lock-free mechanism for synchronizing access to an instance field. If you declare a field as `volatile`, then the compiler and the virtual machine take into account that the field may be concurrently updated by another thread.

For example, suppose an object has a `boolean` flag `done` that is set by one thread and queried by another thread. You have two choices:

1. Use a lock, for example:

```
public synchronized boolean isDone() { return done; }
private boolean done;
```

   (This approach has a potential drawback: the `isDone` method can block if another thread has locked the object.)

2. Declare the field as `volatile`:

```
public boolean isDone() { return done; }
private volatile boolean done;
```

Of course, accessing a volatile variable will be slower than accessing a regular variable—that is the price to pay for thread safety.

NOTE: Prior to JDK 5.0, the semantics of `volatile` were rather permissive. The language designers attempted to give implementors leeway in optimizing the performance of code that uses volatile fields. However, the old specification was so complex that implementors didn't always follow it, and it allowed confusing and undesirable behavior, such as immutable objects that weren't truly immutable.

In summary, concurrent access to a field is safe in these three conditions:

- The field is `volatile`.
- The field is `final`, and it is accessed after the constructor has completed.
- The field access is protected by a lock.

Deadlocks

Locks and conditions cannot solve all problems that might arise in multithreading. Consider the following situation:

Account 1: $1,200

Account 2: $1,300

Thread 1: Transfer $300 from Account 1 to Account 2

Thread 2: Transfer $400 from Account 2 to Account 1

As Figure 1–6 indicates, Threads 1 and 2 are clearly blocked. Neither can proceed because the balances in Accounts 1 and 2 are insufficient.

Is it possible that all 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, $1,000. Because there are 100 accounts and a total of $100,000 in them, at least one of the accounts must have more than $1,000 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 $1,000 transaction limit, deadlocks can occur quickly. Try it out. Set `NACCOUNTS` to 10. Construct each transfer thread with a `maxAmount` of `2000` and run the program. The program will run for a while and then hang.

TIP: When the program hangs, type CTRL+\. You will get a thread dump that lists all threads. Each thread has a stack trace, telling you where it is currently blocked.

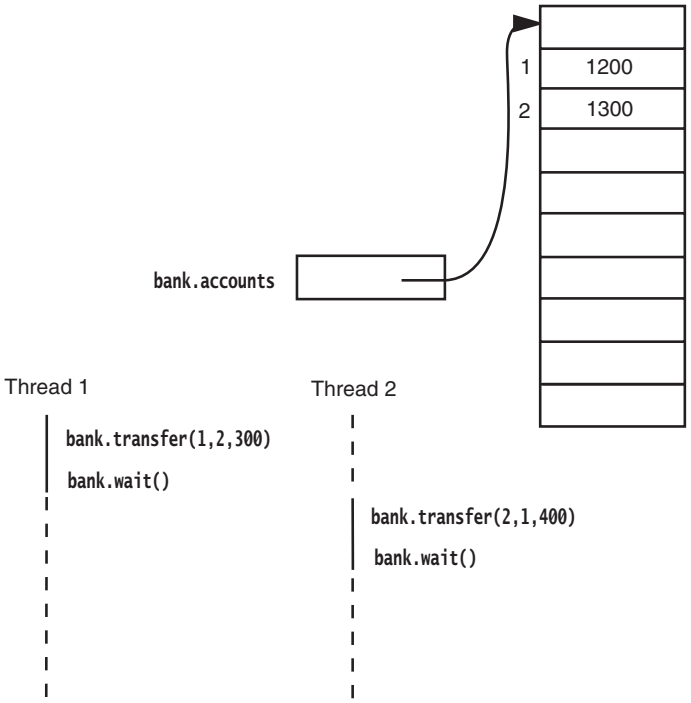

Figure 1–6: A deadlock situation

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 `TransferRunnable` 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 `signalAll` method to `signal` in the `SynchBankTest` program. You will find that the program hangs eventually. (Again, it is best to set `NACCOUNTS` to 10 to observe the effect more quickly.) Unlike `signalAll`, which notifies all threads that are waiting for added funds, the `signal` 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: $1,990

All other accounts: $990 each

Thread 1: Transfer $995 from Account 1 to Account 2

All other threads: Transfer $995 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: $995

Account 2: $1,985

All other accounts: $990 each

Then, Thread 1 calls `signal`. The `signal` 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 `await` again. But Thread 1 is still running. A new random transaction is generated, say,

Thread 1: Transfer $997 to from Account 1 to Account 2

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

The culprit here is the call to `signal`. 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.)

Unfortunately, there is nothing in the Java programming language to avoid or break these deadlocks. You must design your program to ensure that a deadlock situation cannot occur.

Fairness

When you construct a `ReentrantLock`, you can specify that you want a *fair locking policy*:

```
Lock fairLock = new ReentrantLock(true);
```

A fair lock favors the thread that has been waiting for the longest time. However, this fairness guarantee can be a significant drag on performance. Therefore, by default, locks are not required to be fair.

Even if you use a fair lock, you have no guarantee that the thread scheduler is fair. If the thread scheduler chooses to neglect a thread that has been waiting a long time for the lock, then it doesn't get the chance to be treated fairly by the lock.

CAUTION: It sounds nicer to be fair, but fair locks are *a lot slower* than regular locks. You should only enable fair locking if you have a specific reason why fairness is essential for your problem. This is definitely an advanced technique.

java.util.concurrent.locks.ReentrantLock 5.0

- `ReentrantLock(boolean fair)`
  constructs a lock with the given fairness policy.

Lock Testing and Timeouts

A thread blocks indefinitely when it calls the `lock` method to acquire a lock that is owned by another thread. You can be more cautious about acquiring a lock. The `tryLock` method tries to acquire a lock and returns `true` if it was successful. Otherwise, it immediately returns `false`, and the thread can go off and do something else.

```
if (myLock.tryLock())
   // now the thread owns the lock
   try  { . . .  }
   finally  { myLock.unlock(); }
else
  // do something else
```

You can call `tryLock` with a timeout parameter, like this:

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

`TimeUnit` is an enumeration with values `SECONDS`, `MILLISECONDS`, `MICROSECONDS`, and `NANOSECONDS`.

These methods deal with fairness and thread interruption in subtly different ways.

The tryLock method with a timeout parameter respects fairness in the same way as the lock method. But the tryLock method without timeout can *barge in*—if the lock is available when the call is made, the current thread gets it, even if another thread has been waiting to lock it. If you don't want that behavior, you can always call

```
if (myLock.tryLock(0, TimeUnit.SECONDS)) . . .
```

The lock method cannot be interrupted. If a thread is interrupted while it is waiting to acquire a lock, the interrupted thread continues to be blocked until the lock is available. If a deadlock occurs, then the lock method can never terminate.

However, if you call tryLock with a timeout, then an InterruptedException is thrown if the thread is interrupted while it is waiting. This is clearly a useful feature because it allows a program to break up deadlocks.

You can also call the lockInterruptibly method. It has the same meaning as tryLock with an infinite timeout.

When you wait on a condition, you can also supply a timeout:

```
myCondition.await(100, TimeUnit.MILLISECONDS))
```

The await method returns if another thread has activated this thread by calling signalAll or signal, or if the timeout has elapsed, or if the thread was interrupted.

The await methods throw an InterruptedException if the waiting thread is interrupted. In the (perhaps unlikely) case that you'd rather continue waiting, use the awaitUninterruptibly method instead.

***java.util.concurrent.locks.Lock* 5.0**

- `boolean tryLock()`
  tries to acquire the lock without blocking; returns `true` if it was successful. This method grabs the lock if it is available even if it has a fair locking policy and other threads have been waiting.
- `boolean tryLock(long time, TimeUnit unit)`
  tries to acquire the lock, blocking no longer than the given time; returns `true` if it was successful.
- `void lockInterruptibly()`
  acquires the lock, blocking indefinitely. If the thread is interrupted, throws an `InterruptedException`.

***java.util.concurrent.locks.Condition* 5.0**

- `boolean await(long time, TimeUnit unit)`
  enters the wait set for this condition, blocking until the thread is removed from the wait set or the given time has elapsed. Returns `false` if the method returned because the time elapsed, `true` otherwise.
- `void awaitUninterruptibly()`
  enters the wait set for this condition, blocking until the thread is removed from the wait set. If the thread is interrupted, this method does not throw an `InterruptedException`.

Read/Write Locks

The java.util.concurrent.locks package defines two lock classes, the ReentrantLock that we already discussed and the ReentrantReadWriteLock class. The latter is useful when there are many threads that read from a data structure and fewer threads that modify it. In that

situation, it makes sense to allow shared access for the readers. Of course, a writer must still have exclusive access.

Here are the steps that are necessary to use read/write locks:

1. Construct a `ReentrantReadWriteLock` object:
```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```
2. Extract read and write locks:
```
private Lock readLock = rwl.readLock();
private Lock writeLock = rwl.writeLock();
```
3. Use the read lock in all accessors:
```
public double getTotalBalance()
{
   readLock.lock();
   try { . . . }
   finally { readLock.unlock(); }
}
```
4. Use the write lock in all mutators:
```
public void transfer(. . .)
{
   writeLock.lock();
   try { . . . }
   finally { writeLock.unlock(); }
}
```

java.util.concurrent.locks.ReentrantReadWriteLock 5.0

- `Lock readLock()`
  gets a read lock that can be acquired by multiple readers, excluding all writers.
- `Lock writeLock()`
  gets a write lock that excludes all other readers and writers.

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

JDK 1.0 defined a `stop` method that simply terminates a thread, and a `suspend` method that blocks a thread until another thread calls `resume`. The `stop` and `suspend` methods have something in common: Both attempt to control the behavior of a given thread without the thread's cooperation.

Both of these methods have been deprecated since JDK 1.2. 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. This method terminates all pending methods, including the `run` method. 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.* Since the lock has been relinquished, the damage is observable from the other threads that have not been stopped.

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. You should interrupt a thread when you want it to stop. The interrupted thread can then stop when it is safe to do so.

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 claim is not valid. A stopped thread exits all synchronized methods it has called—technically, by throwing a `ThreadDeath` exception. As a consequence, the thread relinquishes the object locks that it holds.

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, then the lock is unavailable until the thread is resumed. If the thread that calls the `suspend` method tries to acquire the same lock, then the program deadlocks: The suspended thread waits to be resumed, and the suspending thread waits for the lock.

This situation occurs frequently in graphical user interfaces. Suppose we have a graphical simulation of our bank. A button labeled Pause suspends the transfer threads, and a button labeled Resume 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(. . .); // calls resume on all transfer threads
```

Suppose a `paintComponent` method paints a chart of each account, calling a `getBalances` method to get an array of balances.

As you will see on page 72, both the button actions and the repainting occur in the same thread, the *event dispatch thread*. Consider the following scenario:

1. One of the transfer threads acquires the lock of 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 `getBalances` method.
6. That method tries to acquire the lock of the `bank` object.

Now the program is frozen.

The event dispatch thread can't proceed because the lock is owned 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 a thread, introduce a variable `suspendRequested` and test it in a safe place of your `run` method—somewhere 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.

The following code framework implements that design:

```
public void run()
{
   while (. . .)
   {
```

```
      . . .
      if (suspendRequested)
      {
         suspendLock.lock();
         try { while (suspendRequested) suspendCondition.await(); }
         finally { suspendLock.unlock(); }
      }
   }
}
public void requestSuspend() { suspendRequested = true; }
public void requestResume()
{
   suspendRequested = false;
   suspendLock.lock();
   try { suspendCondition.signalAll(); }
   finally { suspendLock.unlock(); }
}
private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();
```

Blocking Queues

A *queue* is a data structure with two fundamental operations: to add an element to the *tail* of the queue and to remove an element from the *head*. That is, the queue manages the data in a first-in/first-out discipline. A *blocking queue* causes a thread to block when you try to add an element when the queue is currently full or to remove an element when the queue is empty. Blocking queues are a useful tool for coordinating the work of multiple threads. Worker threads can periodically deposit intermediate results in a blocking queue. Other worker threads remove the intermediate results and modify them further. The queue automatically balances the workload. If the first set of threads runs slower than the second, the second set blocks while waiting for the results. If the first set of threads runs faster, the queue fills up until the second set catches up. Table 1–1 shows the operations for blocking queues.

Table 1–1: Blocking Queue Operations

Method	Normal Action	Failure Action
`add`	Adds an element	Throws an `IllegalStateException` if the queue is full
`remove`	Removes and returns the head element	Throws a `NoSuchElementException` if the queue is empty
`element`	Returns the head element	Throws a `NoSuchElementException` if the queue is empty
`offer`	Adds an element and returns `true`	Returns `false` if the queue is full
`poll`	Removes and returns the head element	Returns `null` if the queue was empty
`peek`	Returns the head element	Returns `null` if the queue was empty
`put`	Adds an element	Blocks if the queue is full
`take`	Removes and returns the head element	Blocks if the queue is empty

The blocking queue operations fall into three categories, depending on their response. The `add`, `remove`, and `element` operations throw an exception when you try to add to a full queue or get the head of an empty queue. Of course, in a multithreaded program, the queue might become full or empty at any time, so you will instead want to use the `offer`, `poll`, and `peek` methods. These methods simply return with a failure indicator instead of throwing an exception if they cannot carry out their tasks.

NOTE: The `poll` and `peek` methods return `null` to indicate failure. Therefore, it is illegal to insert `null` values into these queues.

There are also variants of the `offer` and `poll` methods with a timeout. For example, the call

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

tries for 100 milliseconds to insert an element to the tail of the queue. If it succeeds, it immediately returns `true`; otherwise, it returns `false` when it times out. Similarly, the call

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

returns `true` for 100 milliseconds to remove the head of the queue. If it succeeds, it immediately returns the head; otherwise, it returns `null` when it times out.

Finally, we have blocking operations `put` and `take`. The `put` method blocks if the queue is full, and the `take` method blocks if the queue is empty. These are the equivalents of `offer` and `poll` with no timeout.

The `java.util.concurrent` package supplies four variations of blocking queues. By default, the `LinkedBlockingQueue` has no upper bound on its capacity, but a maximum capacity can be optionally specified. The `ArrayBlockingQueue` is constructed with a given capacity and an optional parameter to require fairness. If fairness is specified, then the longest-waiting threads are given preferential treatment. As always, fairness exacts a significant performance penalty, and you should only use it if your problem specifically requires it.

The `PriorityBlockingQueue` is a priority queue, not a first-in/first-out queue. Elements are removed in order of their priority. The queue has unbounded capacity, but retrieval will block if the queue is empty. (We discuss priority queues in greater detail in Chapter 2.)

Finally, a `DelayQueue` contains objects that implement the `Delayed` interface:

```
interface Delayed extends Comparable<Delayed>
{
   long getDelay(TimeUnit unit);
}
```

The `getDelay` method returns the remaining delay of the object. A negative value indicates that the delay has elapsed. Elements can only be removed from a `DelayQueue` if their delay has elapsed. You also need to implement the `compareTo` method. The `DelayQueue` uses that method to sort the entries.

The program in Example 1–6 shows how to use a blocking queue to control a set of threads. The program searches through all files in a directory and its subdirectories, printing lines that contain a given keyword.

A producer thread enumerates all files in all subdirectories and places them in a blocking queue. This operation is fast, and the queue would quickly fill up with all files in the file system if it was not bounded.

We also start a large number of search threads. Each search thread takes a file from the queue, opens it, prints all lines containing the keyword, and then takes the next file. We use a trick to terminate the application when no further work is required. In order to signal

completion, the enumeration thread places a dummy object into the queue. (This is similar to a dummy suitcase with a label "last bag" in a baggage claim belt.) When a search thread takes the dummy, it puts it back and terminates.

Note that no explicit thread synchronization is required. In this application, we use the queue data structure as a synchronization mechanism.

Example 1–6: BlockingQueueTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. public class BlockingQueueTest
6. {
7.    public static void main(String[] args)
8.    {
9.       Scanner in = new Scanner(System.in);
10.      System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
11.      String directory = in.nextLine();
12.      System.out.print("Enter keyword (e.g. volatile): ");
13.      String keyword = in.nextLine();
14.
15.      final int FILE_QUEUE_SIZE = 10;
16.      final int SEARCH_THREADS = 100;
17.
18.      BlockingQueue<File> queue = new ArrayBlockingQueue<File>(FILE_QUEUE_SIZE);
19.
20.      FileEnumerationTask enumerator = new FileEnumerationTask(queue, new File(directory));
21.      new Thread(enumerator).start();
22.      for (int i = 1; i <= SEARCH_THREADS; i++)
23.         new Thread(new SearchTask(queue, keyword)).start();
24.   }
25. }
26.
27. /**
28.    This task enumerates all files in a directory and its subdirectories.
29. */
30. class FileEnumerationTask implements Runnable
31. {
32.    /**
33.       Constructs a FileEnumerationTask.
34.       @param queue the blocking queue to which the enumerated files are added
35.       @param startingDirectory the directory in which to start the enumeration
36.    */
37.    public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory)
38.    {
39.       this.queue = queue;
40.       this.startingDirectory = startingDirectory;
41.    }
42.
43.    public void run()
44.    {
45.       try
46.       {
47.          enumerate(startingDirectory);
48.          queue.put(DUMMY);
```


```
49.       }
50.       catch (InterruptedException e) {}
51.    }
52.
53.    /**
54.       Recursively enumerates all files in a given directory and its subdirectories
55.       @param directory the directory in which to start
56.    */
57.    public void enumerate(File directory) throws InterruptedException
58.    {
59.       File[] files = directory.listFiles();
60.       for (File file : files)      {
61.          if (file.isDirectory()) enumerate(file);
62.          else queue.put(file);
63.       }
64.    }
65.
66.    public static File DUMMY = new File("");
67.
68.    private BlockingQueue<File> queue;
69.    private File startingDirectory;
70. }
71.
72. /**
73.    This task searches files for a given keyword.
74. */
75. class SearchTask implements Runnable
76. {
77.    /**
78.       Constructs a SearchTask.
79.       @param queue the queue from which to take files
80.       @param keyword the keyword to look for
81.    */
82.    public SearchTask(BlockingQueue<File> queue, String keyword)
83.    {
84.       this.queue = queue;
85.       this.keyword = keyword;
86.    }
87.
88.    public void run()
89.    {
90.       try
91.       {
92.          boolean done = false;
93.          while (!done)
94.          {
95.             File file = queue.take();
96.             if (file == FileEnumerationTask.DUMMY) { queue.put(file); done = true; }
97.             else search(file);
98.          }
99.       }
100.      catch (IOException e) { e.printStackTrace(); }
101.      catch (InterruptedException e) {}
102.   }
103.
104.   /**
```

```
105.       Searches a file for a given keyword and prints all matching lines.
106.       @param file the file to search
107.    */
108.    public void search(File file) throws IOException
109.    {
110.       Scanner in = new Scanner(new FileInputStream(file));
111.       int lineNumber = 0;
112.       while (in.hasNextLine())
113.       {
114.          lineNumber++;
115.          String line = in.nextLine();
116.          if (line.contains(keyword))
117.             System.out.printf("%s:%d:%s%n", file.getPath(), lineNumber, line);
118.       }
119.       in.close();
120.    }
121.
122.    private BlockingQueue<File> queue;
123.    private String keyword;
124. }
```

java.util.concurrent.ArrayBlockingQueue<E> 5.0

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

construct a blocking queue with the given capacity and fairness settings. The queue is implemented as a circular array.

java.util.concurrent.LinkedBlockingQueue<E> 5.0

- `LinkedBlockingQueue()`

constructs an unbounded blocking queue, implemented as a linked list.

- `LinkedBlockingQueue(int capacity)`

constructs a bounded blocking queue with the given capacity, implemented as a linked list.

java.util.concurrent.DelayQueue<E extends Delayed> 5.0

- `DelayQueue()`

constructs an unbounded bounded blocking queue of `Delayed` elements. Only elements whose delay has expired can be removed from the queue.

java.util.concurrent.Delayed 5.0

- `long getDelay(TimeUnit unit)`

gets the delay for this object, measured in the given time unit.

java.util.concurrent.PriorityBlockingQueue<E> 5.0

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`

constructs an unbounded blocking priority queue implemented as a heap.

Parameters	`initialCapacity`	The initial capacity of the priority queue. Default is 11
	`comparator`	The comparator used to compare elements. If not specified, the elements must implement the `Comparable` interface

`java.util.concurrent.BlockingQueue<E>` 5.0

- `void put(E element)`
  adds the element, blocking if necessary.
- `boolean offer(E element)`
- `boolean offer(E element, long time, TimeUnit unit)`
  adds the given element and returns `true` if successful, or returns without adding the element and returns `false` if the queue is full. The second method blocks if necessary, until the element has been added or the time has elapsed.
- `E take()`
  removes and returns the head element, blocking if necessary.
- `E poll(long time, TimeUnit unit)`
  removes and returns the head element, blocking if necessary until an element is available or the time has elapsed. Returns `null` upon failure.

`java.util.Queue<E>` 5.0

- `E poll()`
  removes and returns the head element or `null` if the queue is empty.
- `E peek()`
  returns the head element or `null` if the queue is empty.

Thread-Safe Collections

If multiple threads concurrently modify a data structure such as a hash table, then it is easily possible to damage the data structure. (We discuss hash tables in greater detail in Chapter 2.) For example, one thread may begin to insert a new element. Suppose it is preempted while it is in the middle of rerouting the links between the hash table's buckets. If another thread starts traversing the same list, it may follow invalid links and create havoc, perhaps throwing exceptions or being trapped in an infinite loop.

You can protect a shared data structure by supplying a lock, but it is usually easier to choose a thread-safe implementation instead. The blocking queues that we discussed in the preceding section are, of course, thread-safe collections. In the following sections, we discuss the other thread-safe collections that the Java library provides.

Efficient Queues and Hash Tables

The `java.util.concurrent` package supplies efficient implementations for a queue and a hash table, `ConcurrentLinkedQueue` and `ConcurrentHashMap`. The concurrent hash map can efficiently support a large number of readers and a fixed number of writers. By default, it is assumed that there are up to 16 *simultaneous* writer threads. There can be many more writer threads, but if more than 16 write at the same time, the others are temporarily blocked. You can specify a higher number in the constructor, but it is unlikely that you will need to.

These collections use sophisticated algorithms that never lock the entire table and that minimize contention by allowing simultaneous access to different parts of the data structure.

The collections return *weakly consistent* iterators. That means that the iterators may or may not reflect all modifications that are made after they were constructed, but they will not return a value twice and they will not throw any exceptions.

NOTE: In contrast, an iterator of a collection in the `java.util` package throws a `ConcurrentModificationException` when the collection has been modified after construction of the iterator.

The `ConcurrentHashMap` has useful methods for atomic insertion and removal of associations. The `putIfAbsent` method adds a new association provided there wasn't one before. This is useful for a cache that is accessed by multiple threads, to ensure that only one thread adds an item into the cache:

```
cache.putIfAbsent(key, value);
```

The opposite operation is `remove` (which perhaps should have been called `removeIfPresent`). The call

```
cache.remove(key, value)
```

atomically removes the key and value if they are present in the map. Finally,

```
cache.replace(key, oldValue, newValue)
```

atomically replaces the old value with the new one, provided the old value was associated with the given key.

java.util.concurrent.ConcurrentLinkedQueue<E> 5.0

- `ConcurrentLinkedQueue<E>()`
  constructs an unbounded, nonblocking queue that can be safely accessed by multiple threads.

java.util.concurrent.ConcurrentHashMap<K, V> 5.0

- `ConcurrentHashMap<K, V>()`
- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`
  construct a hash map that can be safely accessed by multiple threads.

Parameters		
	`initialCapacity`	The initial capacity for this collection. Default is 16
	`loadFactor`	Controls resizing: If the average load per bucket exceeds this factor, the table is resized. Default is 0.75
	`concurrencyLevel`	The estimated number of concurrent writer threads

- `V putIfAbsent(K key, V value)`
  if the key is not yet present in the map, associates the given value with the given key and returns `null`. Otherwise returns the existing value associated with the key.
- `boolean remove(K key, V value)`
  if the given key is currently associated with this value, removes the given key and value and returns `true`. Otherwise returns `false`.
- `boolean replace(K key, V oldValue, V newValue)`
  if the given key is currently associated with `oldValue`, associates it with `newValue`. Otherwise, returns `false`.

Copy on Write Arrays

The `CopyOnWriteArrayList` and `CopyOnWriteArraySet` are thread-safe collections in which all mutators make a copy of the underlying array. This arrangement is useful if the number of threads that iterate over the collection greatly outnumbers the threads that mutate it. When you construct an iterator, it contains a reference to the current array. If the array is later mutated, the iterator still has the old array, but the collection's array is replaced. As a consequence, the older iterator has a consistent (but potentially outdated) view that it can access without any synchronization expense.

Older Thread-Safe Collections

Ever since JDK 1.0, the `Vector` and `Hashtable` classes provided thread-safe implementations of a dynamic array and a hash table. In JDK 1.2, these classes were declared obsolete and replaced by the `ArrayList` and `HashMap` classes. Those classes are not thread-safe. Instead, a different mechanism is supplied in the collections library. Any collection class can be made thread-safe by means of a *synchronization wrapper*:

```
List synchArrayList = Collections.synchronizedList(new ArrayList());
Map synchHashMap = Collections.synchronizedMap(new HashMap());
```

The methods of the resulting collections are protected by a lock, providing thread-safe access. However, if you want to *iterate* over the collection, you need to use a synchronized block:

```
synchronized (synchHashMap)
{
   Iterator iter = synchHashMap.keySet().iterator();
   while (iter.hasNext()) . . .;
}
```

We discuss these wrappers in greater detail in Chapter 2.

Callables and Futures

A `Runnable` encapsulates a task that runs asynchronously; you can think of it as an asynchronous method with no parameters and no return value. A `Callable` is similar to a `Runnable`, but it returns a value. The `Callable` interface is a parameterized type, with a single method `call`.

```
public interface Callable<V>
{
   V call() throws Exception;
}
```

The type parameter is the type of the returned value. For example, a `Callable<Integer>` represents an asynchronous computation that eventually returns an `Integer` object.

A `Future` holds the *result* of an asynchronous computation. You use a `Future` object so that you can start a computation, give the result to someone, and forget about it. The owner of the `Future` object can obtain the result when it is ready.

The `Future` interface has the following methods:

```
public interface Future<V>
{
   V get() throws . . .;
   V get(long timeout, TimeUnit unit) throws . . .;
   void cancel(boolean mayInterrupt);
   boolean isCancelled();
   boolean isDone();
}
```

A call to the first `get` method blocks until the computation is finished. The second method throws a `TimeoutException` if the call timed out before the computation finished. If the thread running the computation is interrupted, both methods throw an `InterruptedException`. If the computation has already finished, then `get` returns immediately.

The `isDone` method returns `false` if the computation is still in progress, `true` if it is finished.

You can cancel the computation with the `cancel` method. If the computation has not yet started, it is canceled and will never start. If the computation is currently in progress, then it is interrupted if the `mayInterrupt` parameter is `true`.

The `FutureTask` wrapper is a convenient mechanism for turning a `Callable` into both a `Future` and a `Runnable`—it implements both interfaces. For example,

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // it's a Runnable
t.start();
. . .
Integer result = task.get(); // it's a Future
```

The program in Example 1–7 puts these concepts to work. This program is similar to the preceding example that found files containing a given keyword. However, now we will merely count the number of matching files. Thus, we have a long-running task that yields an integer value—an example of a `Callable<Integer>`.

```
class MatchCounter implements Callable<Integer>
{
   public MatchCounter(File directory, String keyword) { . . . }
   public Integer call() { . . . } // returns the number of matching files
}
```

Then we construct a `FutureTask` object from the `MatchCounter` and use it to start a thread.

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

Finally, we print the result.

```
System.out.println(task.get() + " matching files.");
```

Of course, the call to `get` blocks until the result is actually available.

Inside the `call` method, we use the same mechanism recursively. For each subdirectory, we produce a new `MatchCounter` and launch a thread for it. We also stash the `FutureTask` objects away in an `ArrayList<Future<Integer>>`. At the end, we add up all results:

```
for (Future<Integer> result : results)
   count += result.get();
```

Each call to `get` blocks until the result is available. Of course, the threads run in parallel, so there is a good chance that the results will all be available at about the same time.

Example 1–7: FutureTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. public class FutureTest
6. {
7.    public static void main(String[] args)
8.    {
9.       Scanner in = new Scanner(System.in);
```


```
10.       System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
11.       String directory = in.nextLine();
12.       System.out.print("Enter keyword (e.g. volatile): ");
13.       String keyword = in.nextLine();
14.
15.       MatchCounter counter = new MatchCounter(new File(directory), keyword);
16.       FutureTask<Integer> task = new FutureTask<Integer>(counter);
17.       Thread t = new Thread(task);
18.       t.start();
19.       try
20.       {
21.          System.out.println(task.get() + " matching files.");
22.       }
23.       catch (ExecutionException e)
24.       {
25.          e.printStackTrace();
26.       }
27.       catch (InterruptedException e) {}
28.    }
29. }
30.
31. /**
32.    This task counts the files in a directory and its subdirectories that contain a given keyword.
33. */
34. class MatchCounter implements Callable<Integer>
35. {
36.    /**
37.       Constructs a MatchCounter.
38.       @param directory the directory in which to start the search
39.       @param keyword the keyword to look for
40.    */
41.    public MatchCounter(File directory, String keyword)
42.    {
43.       this.directory = directory;
44.       this.keyword = keyword;
45.    }
46.
47.    public Integer call()
48.    {
49.       count = 0;
50.       try
51.       {
52.          File[] files = directory.listFiles();
53.          ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
54.
55.          for (File file : files)
56.             if (file.isDirectory())
57.             {
58.                MatchCounter counter = new MatchCounter(file, keyword);
59.                FutureTask<Integer> task = new FutureTask<Integer>(counter);
60.                results.add(task);
61.                Thread t = new Thread(task);
62.                t.start();
63.             }
64.             else
65.             {
66.                if (search(file)) count++;
```

```
            }

         for (Future<Integer> result : results)
            try
            {
               count += result.get();
            }
            catch (ExecutionException e)
            {
               e.printStackTrace();
            }
      }
      catch (InterruptedException e) {}
      return count;
   }

   /**
      Searches a file for a given keyword.
      @param file the file to search
      @return true if the keyword is contained in the file
   */
   public boolean search(File file)
   {
      try
      {
         Scanner in = new Scanner(new FileInputStream(file));
         boolean found = false;
         while (!found && in.hasNextLine())
         {
            String line = in.nextLine();
            if (line.contains(keyword)) found = true;
         }
         in.close();
         return found;
      }
      catch (IOException e)
      {
         return false;
      }
   }

   private File directory;
   private String keyword;
   private int count;
}
```

java.util.concurrent.Callable<V> 5.0

- `V call()`
  runs a task that yields a result.

java.util.concurrent.Future<V> 5.0

- `V get()`
- `V get(long time, TimeUnit unit)`
  gets the result, blocking until it is available or the given time has elapsed. The second method throws a `TimeoutException` if it was unsuccessful.

- `boolean cancel(boolean mayInterrupt)`
  attempts to cancel the execution of this task. If the task has already started and the `mayInterrupt` parameter is `true`, it is interrupted. Returns `true` if the cancellation was successful.
- `boolean isCancelled()`
  returns `true` if the task was canceled before it completed.
- `boolean isDone()`
  returns `true` if the task completed, through normal completion, cancellation, or an exception.

`java.util.concurrent.FutureTask<V>` 5.0

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`
  constructs an object that is both a `Future<V>` and a `Runnable`.

Executors

Constructing a new thread is somewhat expensive because it involves interaction with the operating system. If your program creates a large number of short-lived threads, then it should instead use a *thread pool*. A thread pool contains a number of idle threads that are ready to run. You give a `Runnable` to the pool, and one of the threads calls the `run` method. When the `run` method exits, the thread doesn't die but stays around to serve the next request.

Another reason to use a thread pool is to throttle the number of concurrent threads. Creating a huge number of threads can greatly degrade performance and even crash the virtual machine. If you have an algorithm that creates lots of threads, then you should use a "fixed" thread pool that bounds the total number of concurrent threads.

The `Executors` class has a number of static factory methods for constructing thread pools; see Table 1–2 for a summary.

Table 1–2: Executors Factory Methods

Method	Description
`newCachedThreadPool`	New threads are created as needed; idle threads are kept for 60 seconds.
`newFixedThreadPool`	The pool contains a fixed set of threads; idle threads are kept indefinitely.
`newSingleThreadExecutor`	A "pool" with a single thread that executes the submitted tasks sequentially.
`newScheduledThreadPool`	A fixed-thread pool for scheduled execution.
`newSingleThreadScheduledExecutor`	A single-thread "pool" for scheduled execution.

Thread Pools

Let us look at the first three methods in Table 1–2. We discuss the remaining methods on page 63. The `newCachedThreadPool` method constructs a thread pool that executes each task immediately, using an existing idle thread when available and creating a new thread otherwise. The `newFixedThreadPool` method constructs a thread pool with a fixed size. If more tasks are submitted than there are idle threads, then the unserved tasks are placed on a queue. They are run when other tasks have completed. The `newSingleThreadExecutor` is a degenerate

pool of size 1: A single thread executes the submitted tasks, one after another. These three methods return an object of the ThreadPoolExecutor class that implements the ExecutorService interface.

You can submit a Runnable or Callable to an ExecutorService with one of the following methods:

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

The pool will run the submitted task at its earliest convenience. When you call submit, you get back a Future object that you can use to query the state of the task.

The first submit method returns an odd-looking Future<?>. You can use such an object to call isDone, cancel, or isCancelled. But the get method simply returns null upon completion.

The second version of submit also submits a Runnable, and the get method of the Future returns the given result object upon completion.

The third version submits a Callable, and the returned Future gets the result of the computation when it is ready.

When you are done with a connection pool, call shutdown. This method initiates the shutdown sequence for the pool. An executor that is shut down accepts no new tasks. When all tasks are finished, the threads in the pool die. Alternatively, you can call shutdownNow. The pool then cancels all tasks that have not yet begun and attempts to interrupt the running threads.

Here, in summary, is what you do to use a connection pool:

1. Call the static newCachedThreadPool or newFixedThreadPool method of the Executors class.
2. Call submit to submit Runnable or Callable objects.
3. If you want to be able to cancel a task or if you submit Callable objects, hang on to the returned Future objects.
4. Call shutdown when you no longer want to submit any tasks.

For example, the preceding example program produced a large number of short-lived threads, one per directory. The program in Example 1–8 uses a thread pool to launch the tasks instead.

For informational purposes, this program prints out the largest pool size during execution. This information is not available through the ExecutorService interface. For that reason, we had to cast the pool object to the ThreadPoolExecutor class.

Example 1–8: ThreadPoolTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. public class ThreadPoolTest
6. {
7.    public static void main(String[] args) throws Exception
8.    {
9.       Scanner in = new Scanner(System.in);
10.      System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
11.      String directory = in.nextLine();
12.      System.out.print("Enter keyword (e.g. volatile): ");
13.      String keyword = in.nextLine();
14.
15.      ExecutorService pool = Executors.newCachedThreadPool();
```


```
16.
17.       MatchCounter counter = new MatchCounter(new File(directory), keyword, pool);
18.       Future<Integer> result = pool.submit(counter);
19.
20.       try
21.       {
22.          System.out.println(result.get() + " matching files.");
23.       }
24.       catch (ExecutionException e)
25.       {
26.          e.printStackTrace();
27.       }
28.       catch (InterruptedException e) {}
29.       pool.shutdown();
30.
31.       int largestPoolSize = ((ThreadPoolExecutor) pool).getLargestPoolSize();
32.       System.out.println("largest pool size=" + largestPoolSize);
33.    }
34. }
35.
36. /**
37.    This task counts the files in a directory and its subdirectories that contain a given keyword.
38. */
39. class MatchCounter implements Callable<Integer>
40. {
41.    /**
42.       Constructs a MatchCounter.
43.       @param directory the directory in which to start the search
44.       @param keyword the keyword to look for
45.       @param pool the thread pool for submitting subtasks
46.    */
47.    public MatchCounter(File directory, String keyword, ExecutorService pool)
48.    {
49.       this.directory = directory;
50.       this.keyword = keyword;
51.       this.pool = pool;
52.    }
53.
54.    public Integer call()
55.    {
56.       count = 0;
57.       try
58.       {
59.          File[] files = directory.listFiles();
60.          ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
61.
62.          for (File file : files)
63.             if (file.isDirectory())
64.             {
65.                MatchCounter counter = new MatchCounter(file, keyword, pool);
66.                Future<Integer> result = pool.submit(counter);
67.                results.add(result);
68.             }
69.             else
70.             {
71.                if (search(file)) count++;
```

```
72.             }
73.
74.          for (Future<Integer> result : results)
75.             try
76.             {
77.                count += result.get();
78.             }
79.             catch (ExecutionException e)
80.             {
81.                e.printStackTrace();
82.             }
83.       }
84.       catch (InterruptedException e) {}
85.       return count;
86.    }
87.
88.    /**
89.       Searches a file for a given keyword.
90.       @param file the file to search
91.       @return true if the keyword is contained in the file
92.    */
93.    public boolean search(File file)
94.    {
95.       try
96.       {
97.          Scanner in = new Scanner(new FileInputStream(file));
98.          boolean found = false;
99.          while (!found && in.hasNextLine())
100.         {
101.            String line = in.nextLine();
102.            if (line.contains(keyword)) found = true;
103.         }
104.         in.close();
105.         return found;
106.      }
107.      catch (IOException e)
108.      {
109.         return false;
110.      }
111.   }
112.
113.   private File directory;
114.   private String keyword;
115.   private ExecutorService pool;
116.   private int count;
117. }
```

java.util.concurrent.Executors 5.0

- `ExecutorService newCachedThreadPool()`
  returns a cached thread pool that creates threads as needed and terminates threads that have been idle for 60 seconds.
- `ExecutorService newFixedThreadPool(int threads)`
  returns a thread pool that uses the given number of threads to execute tasks.

- `ExecutorService newSingleThreadExecutor()`
  returns an executor that executes tasks sequentially in a single thread.

java.util.concurrent.ExecutorService 5.0

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
- `Future<?> submit(Runnable task)`
  submits the given task for execution.
- `void shutdown()`
  shuts down the service, completing the already submitted tasks but not accepting new submissions.

java.util.concurrent.ThreadPoolExecutor 5.0

- `int getLargestPoolSize()`
  returns the largest size of the thread pool during the life of this executor.

Scheduled Execution

The `ScheduledExecutorService` interface has methods for scheduled or repeated execution of tasks. It is a generalization of `java.util.Timer` that allows for thread pooling. The `newScheduledThreadPool` and `newSingleThreadScheduledExecutor` methods of the `Executors` class return objects that implement the `ScheduledExecutorService` interface.

You can schedule a `Runnable` or `Callable` to run once, after an initial delay. You can also schedule a `Runnable` to run periodically. See the API notes for details.

java.util.concurrent.Executors 5.0

- `ScheduledExecutorService newScheduledThreadPool(int threads)`
  returns a thread pool that uses the given number of threads to schedule tasks.
- `ScheduledExecutorService newSingleThreadScheduledExecutor()`
  returns an executor that schedules tasks in a single thread.

java.util.concurrent.ScheduledExecutorService 5.0

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`
  schedules the given task after the given time has elapsed.
- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`
  schedules the given task to run periodically, every `period` units, after the initial delay has elapsed.
- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`
  schedules the given task to run periodically, with `delay` units between completion of one invocation and the start of the next, after the initial delay has elapsed.

Controlling Groups of Threads

You have seen how to use an executor service as a thread pool to increase the efficiency of task execution. Sometimes, an executor is used for a more tactical reason, simply to control a group of related tasks. For example, you can cancel all tasks in an executor with the `shutdownNow` method.

The invokeAny method submits all objects in a collection of Callable objects and returns the result of a completed task. You don't know which task that is—presumably, it was the one that finished most quickly. You would use this method for a search problem in which you are willing to accept any solution. For example, suppose that you need to factor a large integer—a computation that is required for breaking the RSA cipher. You could submit a number of tasks, each of which attempts a factorization by using numbers in a different range. As soon as one of these tasks has an answer, your computation can stop.

The invokeAll method submits all objects in a collection of Callable objects and returns a list of Future objects that represent the solutions to all tasks. You can combine the results of the computation when they are available, like this:

```
ArrayList<Callable<Integer>> tasks = . . .;
List<Future<Integer>> results = executor.invokeAll(tasks);
for (Future<Integer> result : results)
   count += result.get();
```

A disadvantage with this approach is that you may wait needlessly if the first task happens to take a long time. It would make more sense to obtain the results in the order in which they are available. This can be arranged with the ExecutorCompletionService.

Start with an executor, obtained in the usual way. Then construct an ExecutorCompletionService. Submit tasks to the completion service. The service manages a blocking queue of Future objects, containing the results of the submitted tasks as they become available. Thus, a more efficient organization for the preceding computation is the following:

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for (Callable<Integer> task : tasks) service.submit(task);
for (int i = 0; i < taks.size(); i++)
   count += service.take().get();
```

java.util.concurrent.ExecutorService 5.0

- T invokeAny(Collection<Callable<T>> tasks)
- T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)
  executes the given tasks and returns the result of one of them. The second method throws a TimeoutException if a timeout occurs.
- List<Future<T>> invokeAll(Collection<Callable<T>> tasks)
- List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)
  executes the given tasks and returns the results of all of them. The second method throws a TimeoutException if a timeout occurs.

java.util.concurrent.ExecutorCompletionService 5.0

- ExecutorCompletionService(Executor e)
  constructs an executor completion service that collects the results of the given executor.
- Future<T> submit(Callable<T> task)
- Future<T> submit(Runnable task, T result)
  submits a task to the underlying executor.
- Future<T> take()
  removes the next completed result, blocking if no completed results are available.
- Future<T> poll()
- Future<T> poll(long time, TimeUnit unit)
  removes the next completed result or null if no completed results are available. The second method waits for the given time.

Synchronizers

The java.util.concurrent package contains several classes that help manage a set of collaborating threads—see Table 1–3. These mechanisms have "canned functionality" for common rendezvous patterns between threads. If you have a set of collaborating threads that follows one of these behavior patterns, you should simply reuse the appropriate library class instead of trying to come up with a handcrafted collection of locks.

Table 1–3: Synchronizers

Class	What It Does	When To Use
CyclicBarrier	Allows a set of threads to wait until a predefined count of them has reached a common barrier, and then optionally executes a barrier action.	When a number of threads need to complete before their results can be used.
CountDownLatch	Allows a set of threads to wait until a count has been decremented to 0.	When one or more threads need to wait until a specified number of results are available.
Exchanger	Allows two threads to exchange objects when both are ready for the exchange.	When two threads work on two instances of the same data structure, one by filling an instance and the other by emptying the other.
SynchronousQueue	Allows a thread to hand off an object to another thread.	To send an object from one thread to another when both are ready, without explicit synchronization.
Semaphore	Allows a set of threads to wait until permits are available for proceeding.	To restrict the total number of threads that can access a resource. If permit count is one, use to block threads until another thread gives permission.

Barriers

The CyclicBarrier class implements a rendezvous called a *barrier*. Consider a number of threads that are working on parts of a computation. When all parts are ready, the results need to be combined. When a thread is done with its part, we let it run against the barrier. Once all threads have reached the barrier, the barrier gives way and the threads can proceed.

Here are the details. First, construct a barrier, giving the number of participating threads:

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

Each thread does some work and calls await on the barrier upon completion:

```
public void run()
{
   doWork();
   barrier.await();
   . . .
}
```

The await method takes an optional timeout parameter:

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

If any of the threads waiting for the barrier leaves the barrier, then the barrier *breaks*. (A thread can leave because it called `await` with a timeout or because it was interrupted.) In that case, the `await` method for all other threads throws a `BrokenBarrierException`. Threads that are already waiting have their `await` call terminated immediately.

You can supply an optional *barrier action* that is executed when all threads have reached the barrier:

```
Runnable barrierAction = . . .;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

The action can harvest the result of the individual threads.

The barrier is called *cyclic* because it can be reused after all waiting threads have been released.

Countdown Latches

A `CountDownLatch` lets a set of threads wait until a count has reached zero. It differs from a barrier in these respects:

- Not all threads need to wait for the latch until it can be opened.
- The latch can be counted down by external events.
- The countdown latch is one-time only. Once the count has reached 0, you cannot reuse it.

A useful special case is a latch with a count of 1. This implements a one-time *gate*. Threads are held at the gate until another thread sets the count to 0.

Imagine, for example, a set of threads that need some initial data to do their work. The worker threads are started and wait at the gate. Another thread prepares the data. When it is ready, it calls `countDown`, and all worker threads proceed.

Exchangers

An `Exchanger` is used when two threads are working on two instances of the same data buffer. Typically, one thread fills the buffer, and the other consumes its contents. When both are done, they exchange their buffers.

Synchronous Queues

A synchronous queue is a mechanism that pairs up producer and consumer threads. When a thread calls `put` on a `SynchronousQueue`, it blocks until another thread calls `take`, and vice versa. Unlike the case with an `Exchanger`, data are only transferred in one direction, from the producer to the consumer.

Even though the `SynchronousQueue` class implements the `BlockingQueue` interface, it is not conceptually a queue. It does not contain any elements—its `size` method always returns 0.

Semaphores

Conceptually, a semaphore manages a number of *permits*. To proceed past the semaphore, a thread requests a permit by calling `acquire`. Only a fixed number of permits are available, limiting the number of threads that are allowed to pass. Other threads may issue permits by calling `release`. There are no actual permit objects. The semaphore simply keeps a count. Moreover, a permit doesn't have to be released by the thread that acquires it. In fact, any thread can issue any number of permits. If it issues more than the maximum available, the semaphore is simply set to the maximum count. This generality makes semaphores both very flexible and potentially confusing.

Semaphores were invented by Edsger Dijkstra in 1968, for use as a *synchronization primitive*. Dijkstra showed that semaphores can be efficiently implemented and that they are powerful

enough to solve many common thread synchronization problems. In just about any operating systems textbook, you will find implementations of bounded queues using semaphores. Of course, application programmers shouldn't reinvent bounded queues. We suggest that you only use semaphores when their behavior maps well onto your synchronization problem, without your going through mental contortions.

For example, a semaphore with a permit count of 1 is useful as a gate that can be opened and closed by another thread. Imagine a program that does some work, then waits for a user to study the result and press a button to continue, then does the next unit of work. The worker thread calls `acquire` whenever it is ready to pause. The GUI thread calls `release` whenever the user clicks the Continue button.

What happens if the user clicks the button multiple times while the worker thread is ready? Because only one permit is available, the permit count stays at 1.

The program in Example 1–9 puts this idea to work. The program animates a sorting algorithm. A worker thread sorts an array, stopping periodically and waiting for the user to give permission to proceed. The user can admire a painting of the current state of the algorithm and press the Continue button to allow the worker thread to go to the next step.

We didn't want to bore you with the code for a sorting algorithm, so we simply call `Arrays.sort`. To pause the algorithm, we supply a `Comparator` object that waits for the semaphore. Thus, the animation is paused whenever the algorithm compares two elements. We paint the current values of the array and highlight the elements that are being compared (see Figure 1–7).

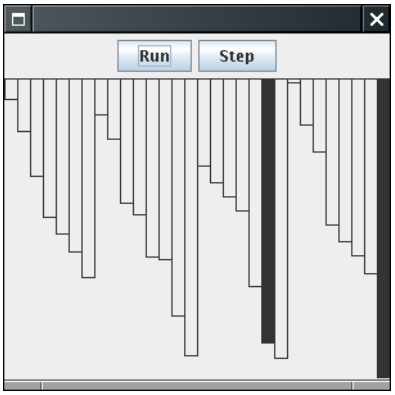

Figure 1–7: Animating a sort algorithm

Example 1–9: AlgorithmAnimation.java

```
1. import java.awt.*;
2. import java.awt.geom.*;
3. import java.awt.event.*;
4. import java.util.*;
5. import java.util.concurrent.*;
6. import javax.swing.*;
7.
8. /**
9.    This program animates a sort algorithm.
10. */
11. public class AlgorithmAnimation
```

```
{
   public static void main(String[] args)
   {
      JFrame frame = new AnimationFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   This frame shows the array as it is sorted, together with buttons to single-step the animation
   or to run it without interruption.
*/
class AnimationFrame extends JFrame
{
   public AnimationFrame()
   {
      ArrayPanel panel = new ArrayPanel();
      add(panel, BorderLayout.CENTER);

      Double[] values = new Double[VALUES_LENGTH];
      final Sorter sorter = new Sorter(values, panel);

      JButton runButton = new JButton("Run");
      runButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               sorter.setRun();
            }
         });

      JButton stepButton = new JButton("Step");
      stepButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               sorter.setStep();
            }
         });

      JPanel buttons = new JPanel();
      buttons.add(runButton);
      buttons.add(stepButton);
      add(buttons, BorderLayout.NORTH);
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      for (int i = 0; i < values.length; i++)
         values[i] = new Double(Math.random());

      Thread t = new Thread(sorter);
      t.start();
   }
```


```
67.
68.    private static final int DEFAULT_WIDTH = 300;
69.    private static final int DEFAULT_HEIGHT = 300;
70.    private static final int VALUES_LENGTH = 30;
71. }
72.
73. /**
74.    This runnable executes a sort algorithm.
75.    When two elements are compared, the algorithm
76.    pauses and updates a panel.
77. */
78. class Sorter implements Runnable
79. {
80.    /**
81.       Constructs a Sorter.
82.       @param values the array to be sorted
83.       @param panel the panel on which to display the sorting progress
84.    */
85.    public Sorter(Double[] values, ArrayPanel panel)
86.    {
87.       this.values = values;
88.       this.panel = panel;
89.       this.gate = new Semaphore(1);
90.       this.run = false;
91.    }
92.
93.    /**
94.       Sets the sorter to "run" mode.
95.    */
96.    public void setRun()
97.    {
98.       run = true;
99.       gate.release();
100.   }
101.
102.   /**
103.      Sets the sorter to "step" mode.
104.   */
105.   public void setStep()
106.   {
107.      run = false;
108.      gate.release();
109.   }
110.
111.   public void run()
112.   {
113.      Comparator<Double> comp = new
114.         Comparator<Double>()
115.         {
116.            public int compare(Double i1, Double i2)
117.            {
118.               panel.setValues(values, i1, i2);
119.               try
120.               {
121.                  if (run)
```

```
122.                      Thread.sleep(DELAY);
123.                   else
124.                      gate.acquire();
125.                }
126.                catch (InterruptedException exception)
127.                {
128.                   Thread.currentThread().interrupt();
129.                }
130.                return i1.compareTo(i2);
131.             }
132.          };
133.       Arrays.sort(values, comp);
134.       panel.setValues(values, null, null);
135.    }
136.
137.    private Double[] values;
138.    private ArrayPanel panel;
139.    private Semaphore gate;
140.    private static final int DELAY = 100;
141.    private boolean run;
142. }
143.
144. /**
145.    This panel draws an array and marks two elements in the
146.    array.
147. */
148. class ArrayPanel extends JPanel
149. {
150.
151.    public void paintComponent(Graphics g)
152.    {
153.       if (values == null) return;
154.       super.paintComponent(g);
155.       Graphics2D g2 = (Graphics2D) g;
156.       int width = getWidth() / values.length;
157.       for (int i = 0; i < values.length; i++)
158.       {
159.          double height = values[i] * getHeight();
160.          Rectangle2D bar = new Rectangle2D.Double(width * i, 0, width, height);
161.          if (values[i] == marked1 || values[i] == marked2)
162.             g2.fill(bar);
163.          else
164.             g2.draw(bar);
165.       }
166.    }
167.
168.    /**
169.       Sets the values to be painted.
170.       @param values the array of values to display
171.       @param marked1 the first marked element
172.       @param marked2 the second marked element
173.    */
174.    public void setValues(Double[] values, Double marked1, Double marked2)
175.    {
176.       this.values = values;
177.       this.marked1 = marked1;
```


```
178.       this.marked2 = marked2;
179.       repaint();
180.    }
181.
182.    private Double marked1;
183.    private Double marked2;
184.    private Double[] values;
185. }
```

java.util.concurrent.CyclicBarrier 5.0

- `CyclicBarrier(int parties)`
- `CyclicBarrier(int parties, Runnable barrierAction)`
  constructs a cyclic barrier for the given number of parties. The `barrierAction` is executed when all parties have called `await` on the barrier.
- `int await()`
- `int await(long time, TimeUnit unit)`
  waits until all parties have called `await` on the barrier or until the timeout has been reached, in which case a `TimeoutException` is thrown. Upon success, returns the arrival index of this party. The first party has index `parties – 1`, and the last party has index 0.

java.util.concurrent.CountDownLatch 5.0

- `CountdownLatch(int count)`
  constructs a countdown latch with the given count.
- `void await()`
  waits for this latch to count down to 0.
- `boolean await(long time, TimeUnit unit)`
  waits for this latch to count down to 0 or for the timeout to elapse. Returns `true` if the count is 0, `false` if the timeout elapsed.
- `public void countDown()`
  counts down the counter of this latch.

java.util.concurrent.Exchanger<V> 5.0

- `V exchange(V item)`
- `V exchange(V item, long time, TimeUnit unit)`
  blocks until another thread calls this method, and then exchanges the item with the other thread and returns the other thread's item. The second method throws a `TimeoutException` after the timeout has elapsed.

java.util.concurrent.SynchronousQueue<V> 5.0

- `SynchronousQueue()`
- `SynchronousQueue(boolean fair)`
  constructs a synchronous queue that allows threads to hand off items. If `fair` is `true`, the queue favors the longest-waiting threads.
- `void put(V item)`
  blocks until another thread calls `take` to take this item.
- `V take()`
  blocks until another thread calls `put`. Returns the item that the other thread provided.

java.util.concurrent.Semaphore 5.0

- `Semaphore(int permits)`
- `Semaphore(int permits, boolean fair)`
  construct a semaphore with the given maximum number of permits. If `fair` is `true`, the queue favors the longest-waiting threads.
- `void acquire()`
  waits to acquire a permit.
- `boolean tryAcquire()`
  tries to acquire a permit; returns `false` if none is available.
- `boolean tryAcquire(long time, TimeUnit unit)`
  tries to acquire a permit within the given time; returns `false` if none is available.
- `void release()`
  releases a permit.

Threads and Swing

As we mentioned in the introduction to this chapter, 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*. If you try to manipulate user interface elements from multiple threads, then your user interface can become corrupted.

To see the problem, run the test program in Example 1–10. When you click the Bad button, a new thread is started whose `run` method tortures a combo box, randomly adding and removing values.

```
public void run()
{
   try
   {
      while (true)
      {
         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 e) {}
   }
}
```

Try it out. Click the Bad button. Click the combo box a few times. Move the scroll bar. Move the window. Click the Bad button again. Keep clicking the combo box. Eventually, you should see an exception report (see Figure 1–8).

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 going—occasionally resulting in a reduction of the count of the values in the combo box. The display code then thinks that there are more values in

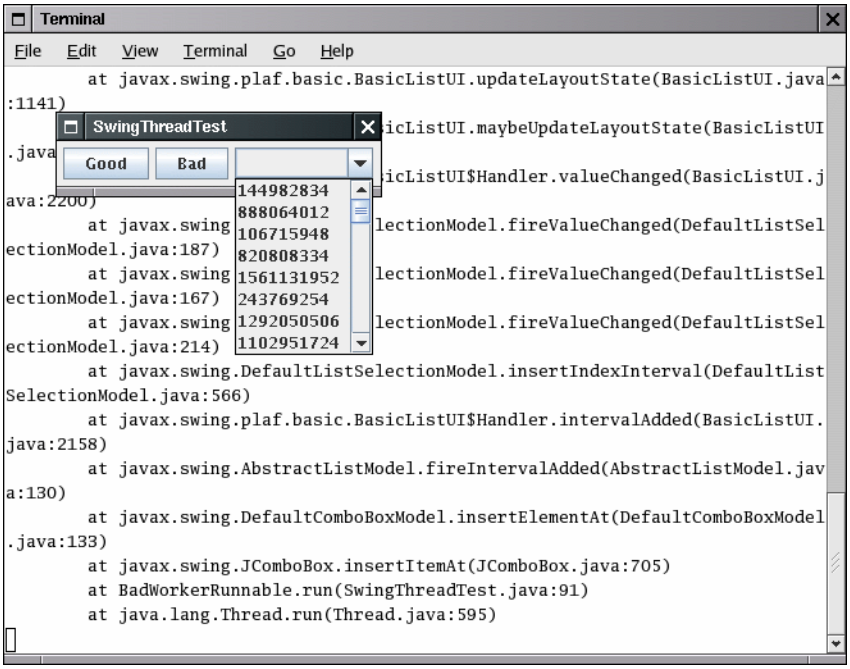

Figure 1–8: Exception reports in the console

the model than there actually are, asks for nonexistent values, and triggers an `ArrayIndexOutOfBounds` exception.

This situation could have been avoided by enabling programmers to lock 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 important, the Swing team checked out the experience other teams had with thread-safe user interface toolkits. What they found was not encouraging. Builders of a user interface toolkit want it to be extensible so that other programmers can add their own user interface components. However, 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.

The "Single Thread" Rule

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 `setVisible` method on the frame 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.

Other threads, such as the thread that posts events into the event queue, are running behind the scenes, but those threads are invisible to the application programmer.

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" because 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 timer events.
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 multiple layout events are in the event queue, the layout is only recomputed once.)

NOTE: We 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 have changed. The traditional AWT has a `validate` method 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 `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 `setVisible(true)`, and you can create the GUI of an applet in the applet constructor or the `init` method.

Now suppose you fire up a separate thread to run a time-consuming task. You may want to update the user interface to indicate progress while your thread is working. When your task is finished, you want to update the GUI again. But you can't touch 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, you can use two convenient utility methods in any thread to add arbitrary actions to the event queue. For example, suppose you want to periodically update a label in a thread to indicate progress. You can't call `label.setText` from your thread. Instead, use the `invokeLater` and `invokeAndWait` methods of the `EventQueue` class to have that call executed in the event dispatching thread.

Here is what you 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 to update a label text.

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

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.

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 have the most precise progress indicator.

Both methods execute the `run` method in the event dispatch thread. No new thread is created.

Example 1–10 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 1–10: 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.setVisible(true);
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("SwingThreadTest");
31.
32.       final JComboBox combo = new JComboBox();
33.       combo.insertItemAt(new Integer(Integer.MAX_VALUE), 0);
34.       combo.setPrototypeDisplayValue(combo.getItemAt(0));
35.       combo.setSelectedIndex(0);
36.
37.       JPanel panel = new JPanel();
38.
39.       JButton goodButton = new JButton("Good");
40.       goodButton.addActionListener(new ActionListener()
41.          {
42.             public void actionPerformed(ActionEvent event)
43.             {
44.                new Thread(new GoodWorkerRunnable(combo)).start();
45.             }
46.          });
47.       panel.add(goodButton);
48.       JButton badButton = new JButton("Bad");
49.       badButton.addActionListener(new ActionListener()
50.          {
51.             public void actionPerformed(ActionEvent event)
52.             {
53.                new Thread(new BadWorkerRunnable(combo)).start();
54.             }
55.          });
56.       panel.add(badButton);
57.       panel.add(combo);
58.       add(panel);
59.       pack();
60.    }
61. }
62.
63. /**
64.    This runnable modifies a combo box by randomly adding
65.    and removing numbers. This can result in errors because
66.    the combo box methods are not synchronized and both the worker
67.    thread and the event dispatch thread access the combo box.
68. */
69. class BadWorkerRunnable implements Runnable
70. {
```


```
71.    public BadWorkerRunnable(JComboBox aCombo)
72.    {
73.       combo = aCombo;
74.       generator = new Random();
75.    }
76.
77.    public void run()
78.    {
79.       try
80.       {
81.          while (true)
82.          {
83.             combo.showPopup();
84.             int i = Math.abs(generator.nextInt());
85.             if (i % 2 == 0)
86.                combo.insertItemAt(new Integer(i), 0);
87.             else if (combo.getItemCount() > 0)
88.                combo.removeItemAt(i % combo.getItemCount());
89.             Thread.sleep(1);
90.          }
91.       }
92.       catch (InterruptedException e) {}
93.    }
94.
95.    private JComboBox combo;
96.    private Random generator;
97. }
98.
99. /**
100.    This runnable modifies a combo box by randomly adding
101.    and removing numbers. In order to ensure that the
102.    combo box is not corrupted, the editing operations are
103.    forwarded to the event dispatch thread.
104. */
105. class GoodWorkerRunnable implements Runnable
106. {
107.    public GoodWorkerRunnable(JComboBox aCombo)
108.    {
109.       combo = aCombo;
110.       generator = new Random();
111.    }
112.
113.    public void run()
114.    {
115.       try
116.       {
117.          while (true)
118.          {
119.             EventQueue.invokeLater(new
120.                Runnable()
121.                {
122.                   public void run()
123.                   {
124.                      combo.showPopup();
125.                      int i = Math.abs(generator.nextInt());
```

```
126.                     if (i % 2 == 0)
127.                        combo.insertItemAt(new Integer(i), 0);
128.                     else if (combo.getItemCount() > 0)
129.                        combo.removeItemAt(i % combo.getItemCount());
130.                  }
131.               });
132.            Thread.sleep(1);
133.         }
134.      }
135.      catch (InterruptedException e) {}
136.   }
137.
138.   private JComboBox combo;
139.   private Random generator;
140. }
```

java.awt.EventQueue 1.0

- `static void invokeLater(Runnable runnable)` **1.2**
  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)` **1.2**
  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.

A Swing Worker

When a user issues a command for which processing takes a long time, you will want to fire up a new thread to do the work. As you saw in the preceding section, that thread should use the `EventQueue.invokeLater` method to update the user interface.

Several authors have produced convenience classes to ease this task. A well-known example is Hans Muller's `SwingWorker` class, described in `http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html`. Here, we present a slightly different class that makes it easier for a thread to update the user interface after each unit of work.

The program in Example 1–11 has commands for loading a text file and for canceling the file loading process. You should try the program with a long file, such as the full text of *The Count of Monte Cristo*, supplied in the `gutenberg` directory of the book's companion code. The file is loaded in a separate thread. While the file is read, the Open menu item is disabled and the Cancel item is enabled (see Figure 1–9). After each line is read, a line counter in the status bar is updated. After the reading process is complete, the Open menu item is reenabled, the Cancel item is disabled, and the status line text is set to Done.

This example shows the typical UI activities of a worker thread:

- Make an initial update to the UI before starting the work.
- After each work unit, update the UI to show progress.
- After the work is finished, make a final change to the UI.

The `SwingWorkerTask` class in Example 1–11 makes it easy to implement such a task. You extend the class and override the `init`, `update`, and `finish` methods and implement the logic for the UI updates. The superclass contains convenience methods `doInit`, `doUpdate`, and `doFinish` that supply the unwieldy code for running these methods in the event dispatch thread. For example,


```
private void doInit()
{
   EventQueue.invokeLater(new
      Runnable()
      {
         public void run() { init(); }
      });
}
```

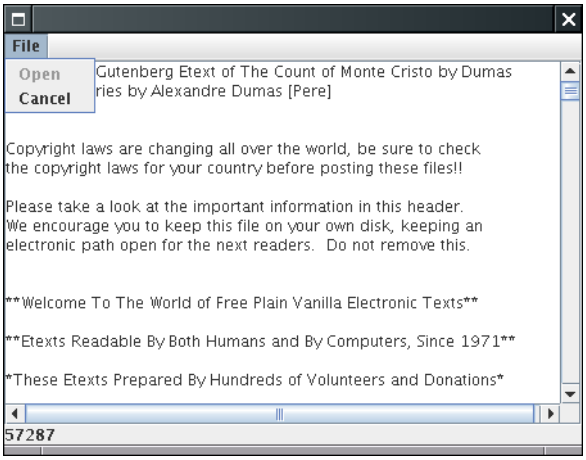

Figure 1–9: Loading a file in a separate thread

Then supply a `work` method that contains the work of the task. In the `work` method, you need to call `doUpdate` (not `update`) after every unit of work. For example, the file reading task has this `work` method:

```
public void work() // exception handling not shown
{
   Scanner in = new Scanner(new FileInputStream(file));
   textArea.setText("");
   while (!Thread.currentThread().isInterrupted() && in.hasNextLine())
   {
      lineNumber++;
      line = in.nextLine();
      textArea.append(line);
      textArea.append("\n");
      doUpdate();
   }
}
```

The `SwingWorkerTask` class implements the `Runnable` interface. The `run` method is straightforward:

```
public final void run()
{
   doInit();
   try
   {
      done = false;
      work();
   }
   catch (InterruptedException e)
```

```
      {
      }
      finally
      {
         done = true;
         doFinish();
      }
   }
```

You simply start a thread with an object of your `SwingWorkerTask object` or submit it to an `Executor`. The sample program does that in the handler for the Open menu item. The handler returns immediately, allowing the user to select other user interface elements.

This simple technique allows you to execute time-consuming tasks while keeping the user interface responsive.

Example 1–11: SwingWorkerTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.*;
5. import java.util.concurrent.*;
6. import javax.swing.*;
7.
8. /**
9.    This program demonstrates a worker thread that runs
10.   a potentially time-consuming task.
11. */
12. public class SwingWorkerTest
13. {
14.    public static void main(String[] args) throws Exception
15.    {
16.       JFrame frame = new SwingWorkerFrame();
17.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.       frame.setVisible(true);
19.    }
20. }
21.
22. /**
23.    This frame has a text area to show the contents of a text file,
24.    a menu to open a file and cancel the opening process, and
25.    a status line to show the file loading progress.
26. */
27. class SwingWorkerFrame extends JFrame
28. {
29.    public SwingWorkerFrame()
30.    {
31.       chooser = new JFileChooser();
32.       chooser.setCurrentDirectory(new File("."));
33.
34.       textArea = new JTextArea();
35.       add(new JScrollPane(textArea));
36.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.
38.       statusLine = new JLabel();
39.       add(statusLine, BorderLayout.SOUTH);
40.
```


```
41.       JMenuBar menuBar = new JMenuBar();
42.       setJMenuBar(menuBar);
43.
44.       JMenu menu = new JMenu("File");
45.       menuBar.add(menu);
46.
47.       openItem = new JMenuItem("Open");
48.       menu.add(openItem);
49.       openItem.addActionListener(new
50.          ActionListener()
51.          {
52.             public void actionPerformed(ActionEvent event)
53.             {
54.                // show file chooser dialog
55.                int result = chooser.showOpenDialog(null);
56.
57.                // if file selected, set it as icon of the label
58.                if (result == JFileChooser.APPROVE_OPTION)
59.                {
60.                   readFile(chooser.getSelectedFile());
61.                }
62.             }
63.          });
64.
65.       cancelItem = new JMenuItem("Cancel");
66.       menu.add(cancelItem);
67.       cancelItem.setEnabled(false);
68.       cancelItem.addActionListener(new
69.          ActionListener()
70.          {
71.             public void actionPerformed(ActionEvent event)
72.             {
73.                if (workerThread != null) workerThread.interrupt();
74.             }
75.          });
76.    }
77.
78.    /**
79.       Reads a file asynchronously, updating the UI during the reading process.
80.       @param file the file to read
81.    */
82.    public void readFile(final File file)
83.    {
84.       Runnable task = new
85.          SwingWorkerTask()
86.          {
87.             public void init()
88.             {
89.                lineNumber = 0;
90.                openItem.setEnabled(false);
91.                cancelItem.setEnabled(true);
92.             }
93.
94.             public void update()
95.             {
96.                statusLine.setText("" + lineNumber);
```

```
97.             }
98.
99.             public void finish()
100.            {
101.               workerThread = null;
102.               openItem.setEnabled(true);
103.               cancelItem.setEnabled(false);
104.               statusLine.setText("Done");
105.            }
106.
107.            public void work()
108.            {
109.               try
110.               {
111.                  Scanner in = new Scanner(new FileInputStream(file));
112.                  textArea.setText("");
113.                  while (!Thread.currentThread().isInterrupted() && in.hasNextLine())
114.                  {
115.                     lineNumber++;
116.                     line = in.nextLine();
117.                     textArea.append(line);
118.                     textArea.append("\n");
119.                     doUpdate();
120.                  }
121.               }
122.               catch (IOException e)
123.               {
124.                  JOptionPane.showMessageDialog(null, "" + e);
125.               }
126.            }
127.
128.            private String line;
129.            private int lineNumber;
130.         };
131.
132.      workerThread = new Thread(task);
133.      workerThread.start();
134.   }
135.
136.   private JFileChooser chooser;
137.   private JTextArea textArea;
138.   private JLabel statusLine;
139.   private JMenuItem openItem;
140.   private JMenuItem cancelItem;
141.   private Thread workerThread;
142.
143.   public static final int DEFAULT_WIDTH = 450;
144.   public static final int DEFAULT_HEIGHT = 350;
145. }
146.
147. /**
148.    Extend this class to define an asynchronous task
149.    that updates a Swing UI.
150. */
151. abstract class SwingWorkerTask implements Runnable
152. {
```


```
153.    /**
154.       Place your task in this method. Be sure to call doUpdate(), not update(), to show the
155.       update after each unit of work.
156.    */
157.    public abstract void work() throws InterruptedException;
158.
159.    /**
160.       Override this method for UI operations before work commences.
161.    */
162.    public void init() {}
163.    /**
164.       Override this method for UI operations after each unit of work.
165.    */
166.    public void update() {}
167.    /**
168.       Override this method for UI operations after work is completed.
169.    */
170.    public void finish() {}
171.
172.    private void doInit()
173.    {
174.       EventQueue.invokeLater(new
175.          Runnable()
176.          {
177.             public void run() { init(); }
178.          });
179.    }
180.
181.    /**
182.       Call this method from work() to show the update after each unit of work.
183.    */
184.    protected final void doUpdate()
185.    {
186.       if (done) return;
187.       EventQueue.invokeLater(new
188.          Runnable()
189.          {
190.             public void run() { update(); }
191.          });
192.    }
193.
194.    private void doFinish()
195.    {
196.       EventQueue.invokeLater(new
197.          Runnable()
198.          {
199.             public void run() { finish(); }
200.          });
201.    }
202.
203.    public final void run()
204.    {
205.       doInit();
206.       try
207.       {
208.          done = false;
```

```
209.            work();
210.         }
211.         catch (InterruptedException ex)
212.         {
213.         }
214.         finally
215.         {
216.            done = true;
217.            doFinish();
218.         }
219.      }
220.
221.      private boolean done;
222. }
```