CHAPTER 14

Concurrency

In this chapter

You are probably familiar with *multitasking*—your operating system's ability to have more than one program working at what seems like the same time. For example, you can print while editing or downloading your email. Nowadays, you are likely to have a computer with more than one CPU, but the number of concurrently executing processes is not limited by the number of CPUs. The operating system assigns CPU time slices to each process, giving the impression of parallel activity.

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. 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.

Fair warning: Concurrent programming 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 *Java Concurrency in Practice* by Brian Goetz et al. (Addison-Wesley Professional, 2006).

14.1 What Are Threads?

Let us start by looking at a program that does not use multiple threads and that, as a consequence, makes it difficult for the user to perform several tasks with that program. After we dissect it, we will 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 14.1.)

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 redraws the panel.

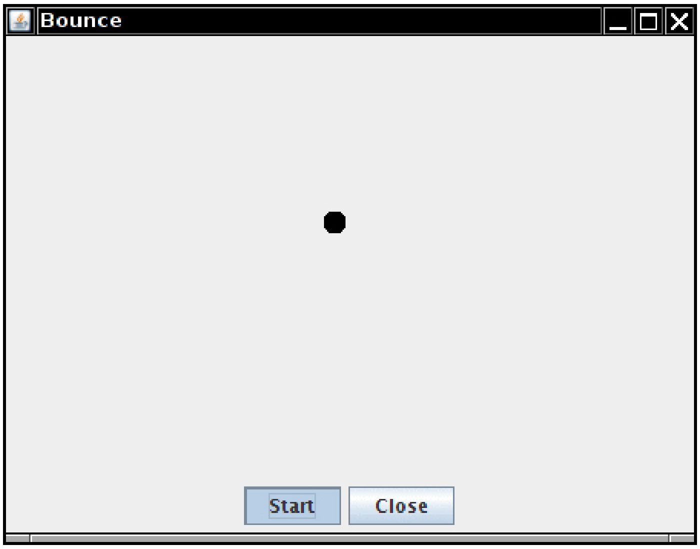


Figure 14.1 Using a thread to animate a bouncing ball

```
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 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 for the given number of milliseconds.

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

```
comp.paint(comp.getGraphics())
```

inside the `addBall` method of the `BounceFrame` 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 `comp.repaint()` in this program, you'll find that the panel is only repainted after the `addBall` method has returned. Also note that the ball component extends `JPanel`; this makes it easier to erase the background. In the next program, in which we use a separate thread to compute the ball position, we can go back to the familiar use of `repaint` and `JComponent`.

Obviously, the behavior of this program is rather poor. You would not want a program you use to behave in this way when you ask it 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 will show you how to keep the user in control by running crucial parts of the code in a separate *thread*.

Listings 14.1 through 14.3 show the code for the program.

Listing 14.1 `bounce/Bounce.java`

```
package bounce;

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

/**
 * Shows an animated bouncing ball.
 * @version 1.34 2015-06-21
 * @author Cay Horstmann
 */
public class Bounce
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() -> {
         JFrame frame = new BounceFrame();
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
            frame.setVisible(true);
         });
      }
}

/**
 * The frame with ball component and buttons.
 */
class BounceFrame extends JFrame
{
   private BallComponent comp;
   public static final int STEPS = 1000;
   public static final int DELAY = 3;

   /**
    * Constructs the frame with the component for showing the bouncing ball and
    * Start and Close buttons
    */
   public BounceFrame()
   {
      setTitle("Bounce");
      comp = new BallComponent();
      add(comp, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start", event -> addBall());
      addButton(buttonPanel, "Close", event -> System.exit(0));
      add(buttonPanel, BorderLayout.SOUTH);
      pack();
   }

   /**
    * Adds a button to a container.
    * @param c the container
    * @param title the button title
    * @param listener the action listener for the button
    */
   public void addButton(Container c, String title, ActionListener listener)
   {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }

   /**
    * Adds a bouncing ball to the panel and makes it bounce 1,000 times.
    */
   public void addBall()
   {
```

(Continues)

Listing 14.1 *(Continued)*

```
      try
      {
         Ball ball = new Ball();
         comp.add(ball);

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

Listing 14.2 `bounce/Ball.java`

```
package bounce;

import java.awt.geom.*;

/**
 * A ball that moves and bounces off the edges of a rectangle
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
 */
public class Ball
{
   private static final int XSIZE = 15;
   private static final int YSIZE = 15;
   private double x = 0;
   private double y = 0;
   private double dx = 1;
   private double dy = 1;

   /**
    * Moves the ball to the next position, reversing direction if it hits one of the edges
    */
   public void move(Rectangle2D bounds)
   {
      x += dx;
      y += dy;
      if (x < bounds.getMinX())
      {
         x = bounds.getMinX();
         dx = -dx;
      }
      if (x + XSIZE >= bounds.getMaxX())
      {
         x = bounds.getMaxX() - XSIZE;
         dx = -dx;
      }
      if (y < bounds.getMinY())
      {
         y = bounds.getMinY();
         dy = -dy;
      }
      if (y + YSIZE >= bounds.getMaxY())
      {
         y = bounds.getMaxY() - YSIZE;
         dy = -dy;
      }
   }

   /**
    * Gets the shape of the ball at its current position.
    */
   public Ellipse2D getShape()
   {
      return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
   }
}
```

Listing 14.3 `bounce/BallComponent.java`

```
package bounce;

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

/**
 * The component that draws the balls.
 * @version 1.34 2012-01-26
 * @author Cay Horstmann
 */
public class BallComponent extends JPanel
{
   private static final int DEFAULT_WIDTH = 450;
   private static final int DEFAULT_HEIGHT = 350;
```

(Continues)

Listing 14.3 *(Continued)*

```java

   private java.util.List<Ball> balls = new ArrayList<>();

   /**
    * Add a ball to the component.
    * @param b the ball to add
    */
   public void add(Ball b)
   {
      balls.add(b);
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g); // erase background
      Graphics2D g2 = (Graphics2D) g;
      for (Ball b : balls)
      {
         g2.fill(b.getShape());
      }
   }

   public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
}
```

`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

14.1.1 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 moved by its own thread. In addition, the AWT *event dispatch thread* will continue running in parallel, taking care of user interface events. Since each thread gets a chance to run, the event dispatch thread has the opportunity to notice that the user clicks the Close button while the balls are bouncing. The thread can then process the "close" action.

We use ball-bouncing code as an example to give you a visual impression of the need for concurrency. In general, you need to be wary of any long-running computation. Your computation is likely to be a part of some bigger framework, such as a GUI or web framework. Whenever the framework calls one of your methods, there is usually an expectation of a quick return. If you need to do any task that takes a long time, your task should run concurrently.

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();
   }
   ```

   Since `Runnable` is a functional interface, you can make an instance with a lambda expression:

   ```
   Runnable r = () -> { task code };
   ```

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

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

3. Start the thread:

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

To make our bouncing ball program into a separate thread, we need only place the code for the animation inside the `run` method of a `Runnable`, and then start a thread:

```
Runnable r = () -> {
   try
   {
      for (int i = 1; i <= STEPS; i++)
      {
         ball.move(comp.getBounds());
         comp.repaint();
         Thread.sleep(DELAY);
      }
   }
   catch (InterruptedException e)
   {
   }
};
Thread t = new Thread(r);
t.start();
```

Again, we need to catch an `InterruptedException` that the `sleep` method threatens to throw. We will discuss this exception in the next section. Typically, interruption is used to request that a thread terminates. Accordingly, our `run` method exits when an `InterruptedException` occurs.

Whenever the Start button is clicked, the ball is moved in a new thread (see Figure 14.2).

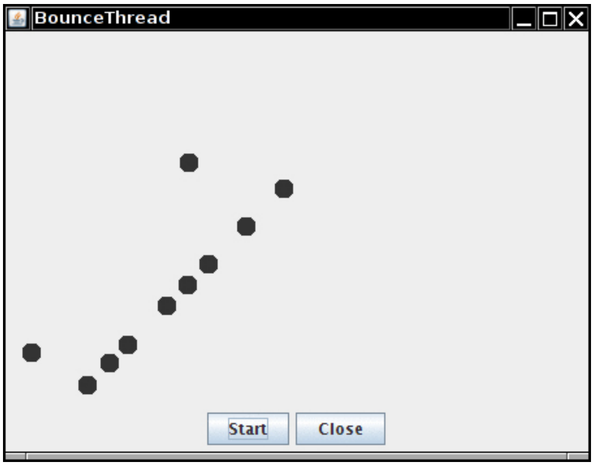


Figure 14.2 Running multiple threads

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 Listing 14.4.

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 of them. Instead, you can use a thread pool—see Section 14.9, "Executors," on p. 920.

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 creates a new thread that executes the `run` method.

Listing 14.4 `bounceThread/BounceThread.java`

```java
package bounceThread;

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

/**
 * Shows animated bouncing balls.
 * @version 1.34 2015-06-21
 * @author Cay Horstmann
 */
public class BounceThread
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() -> {
         JFrame frame = new BounceFrame();
         frame.setTitle("BounceThread");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * The frame with panel and buttons.
 */
class BounceFrame extends JFrame
{
   private BallComponent comp;
   public static final int STEPS = 1000;
   public static final int DELAY = 5;

   /**
    * Constructs the frame with the component for showing the bouncing ball and
    * Start and Close buttons
    */
```

(Continues)

Listing 14.4 *(Continued)*

```java
   public BounceFrame()
   {
      comp = new BallComponent();
      add(comp, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start", event -> addBall());
      addButton(buttonPanel, "Close", event -> System.exit(0));
      add(buttonPanel, BorderLayout.SOUTH);
      pack();
   }

   /**
    * Adds a button to a container.
    * @param c the container
    * @param title the button title
    * @param listener the action listener for the button
    */
   public void addButton(Container c, String title, ActionListener listener)
   {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }

   /**
    * Adds a bouncing ball to the canvas and starts a thread to make it bounce
    */
   public void addBall()
   {
      Ball ball = new Ball();
      comp.add(ball);
      Runnable r = () -> {
         try
         {
            for (int i = 1; i <= STEPS; i++)
            {
               ball.move(comp.getBounds());
               comp.repaint();
               Thread.sleep(DELAY);
            }
         }
         catch (InterruptedException e)
         {
         }
      };
      Thread t = new Thread(r);
      t.start();
   }
}
```

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()`

  must be overridden and supplied with instructions for the task that you want to have executed.

14.2 Interrupting Threads

A thread terminates when its `run` method returns—by executing a `return` statement, after executing the last statement in the method body, or if an exception occurs that is not caught in the method. In the initial release of Java, 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 in Section 14.5.15, "Why the `stop` and `suspend` Methods Are Deprecated," on p. 896.

Other than with the deprecated `stop` method, there is no 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 thread that blocks on a call such as `sleep` or `wait`, the blocking call is terminated by an `InterruptedException`. (There are blocking I/O calls that cannot be interrupted; you should consider interruptible alternatives. See Chapters 1 and 3 of Volume II for details.)

There is no language requirement that a thread which 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:

```
Runnable r = () -> {
   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 neither necessary nor useful if you call the `sleep` method (or another interruptible method) after every work iteration. If you call the `sleep` method when the interrupted status is set, it doesn't sleep. Instead, it clears the status (!) and throws an `InterruptedException`. Therefore, if your loop calls `sleep`, don't check the interrupted status. Instead, catch the `InterruptedException`, like this:

```
Runnable r = () -> {
   try
   {
      . . .
      while (more work to do)
      {
         do more work
         Thread.sleep(delay);
      }
   }
   catch(InterruptedException e)
   {
      // thread was interrupted during sleep
   }
   finally
   {
      cleanup, if required
   }
   // exiting the run method terminates the thread
};
```

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.

14.3 Thread States

Threads can be in one of six states:

- New
- Runnable
- Blocked
- Waiting
- Timed waiting
- Terminated

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

To determine the current state of a thread, simply call the `getState` method.

14.3.1 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.

14.3.2 Runnable Threads

Once you invoke the `start` method, the thread is in the *runnable* state. 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.)

Once a thread is running, it doesn't necessarily keep running. In fact, it is desirable that 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 14.4). When selecting the next thread, the operating system takes into account the thread *priorities*—see Section 14.4.1, "Thread Priorities," on p. 858 for more information.

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 the `yield` method, or when it is blocked or waiting.

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.")

14.3.3 Blocked and Waiting Threads

When a thread is blocked or waiting, it is temporarily inactive. It doesn't execute any code and consumes minimal resources. It is up to the thread scheduler to reactivate it. The details depend on how the inactive state was reached.

- When the thread tries to acquire an intrinsic object lock (but not a `Lock` in the `java.util.concurrent` library) that is currently held by another thread, it becomes *blocked*. (We discuss `java.util.concurrent` locks in Section 14.5.3, "Lock Objects," on p. 868 and intrinsic object locks in Section 14.5.5, "The `synchronized` Keyword," on p. 878.) The thread becomes unblocked when all other threads have relinquished the lock and the thread scheduler has allowed this thread to hold it.
- When the thread waits for another thread to notify the scheduler of a condition, it enters the *waiting* state. We discuss conditions in Section 14.5.4, "Condition Objects," on p. 872. This happens by calling the `Object.wait` or `Thread.join` method, or by waiting for a `Lock` or `Condition` in the `java.util.concurrent` library. In practice, the difference between the blocked and waiting state is not significant.
- Several methods have a timeout parameter. Calling them causes the thread to enter the *timed waiting* state. This state persists either until the timeout expires or the appropriate notification has been received. Methods with timeout include `Thread.sleep` and the timed versions of `Object.wait`, `Thread.join`, `Lock.tryLock`, and `Condition.await`.

Figure 14.3 shows the states that a thread can have and the possible transitions from one state to another. When a thread is blocked or waiting (or, of course, when it terminates), another thread will be scheduled to run. When a thread is reactivated (for example, because its timeout has expired or it has succeeded in acquiring a lock), the scheduler checks to see if it has a higher priority than the currently running threads. If so, it preempts one of the current threads and picks a new thread to run.

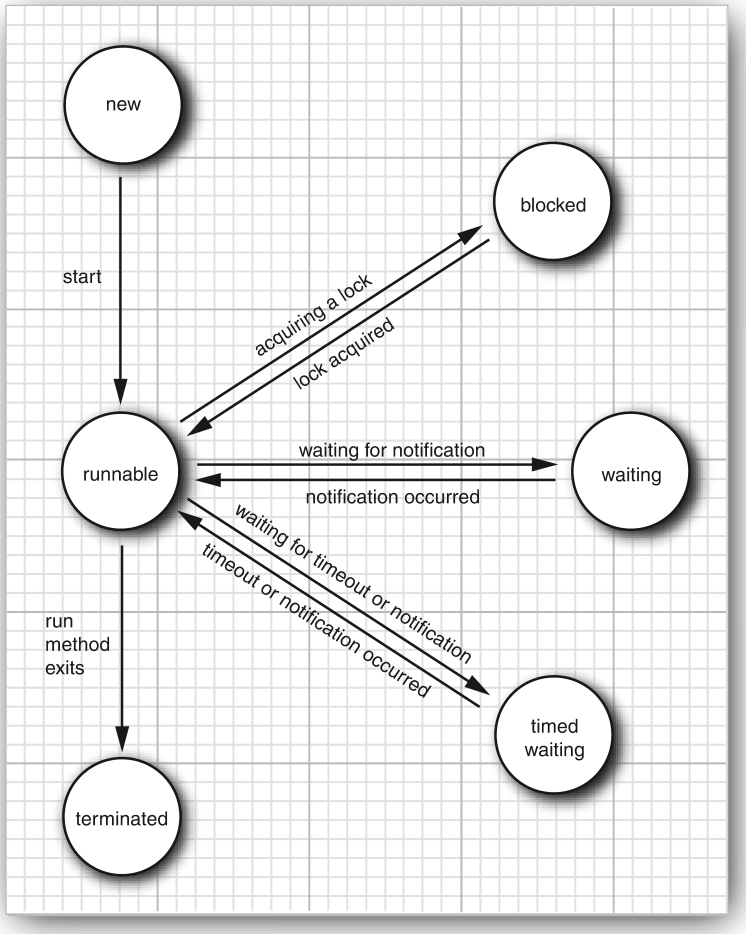


Figure 14.3 Thread states

14.3.4 Terminated Threads

A thread is terminated 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 never call it in your own code.

java.lang.Thread 1.0

- `void join()`

  waits for the specified thread to terminate.
- `void join(long millis)`

  waits for the specified thread to die or for the specified number of milliseconds to pass.
- `Thread.State getState()` 5.0

  gets the state of this thread: one of `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, or `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.

14.4 Thread Properties

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

14.4.1 Thread Priorities

In the Java programming language, every thread has a *priority.* By default, a thread inherits the priority of the thread that constructed 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 has seven priority levels. Some of the Java priorities will map to the same operating system level. In the Oracle JVM for Linux, thread priorities are ignored altogether—all threads have the same priority.

Beginning programmers sometimes overuse thread priorities. There are few reasons ever to tweak priorities. You should certainly 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 don't become inactive, 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.

14.4.2 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 or threads that clean up stale cache entries.

When only daemon threads remain, the virtual machine exits. There is no point in keeping the program running if all remaining threads are daemons.

Daemon threads are sometimes mistakenly used by beginners who don't want to think about shutdown actions. However, this can be dangerous. A daemon thread should never access a persistent resource such as a file or database since it can terminate at any time, even in the middle of an operation.

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.

14.4.3 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)
```

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.

NOTE: A thread group is a collection of threads that can be managed together. By default, all threads that you create belong to the same thread group, but it is possible to establish other groupings. Since there are now better features for operating on collections of threads, we recommend that you do not use thread groups in your programs.

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.getDefaultUncaughtExceptionHandler` 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.

`java.lang.Thread` 1.0

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

  sets or gets the default handler for uncaught exceptions.

- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5.0
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` 5.0

  sets or gets 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)`

  defined 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, the stack trace is suppressed. `ThreadDeath` objects are generated by the deprecated `stop` method.)

14.5 Synchronization

In most practical multithreaded applications, two or more threads need to share access to the same data. 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.*

14.5.1 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. (We don't yet worry about negative account balances.) 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 Runnable instances. The run method keeps moving money out of a given 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.

```
Runnable r = () -> {
   try
   {
      while (true)
      {
         int toAccount = (int) (bank.size() * Math.random());
         double amount = MAX_AMOUNT * 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, errors may 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 such a bank.

The program in Listings 14.5 and 14.6 provides the complete source code. See if you can spot the problems with the code. We will unravel the mystery in the next section.

Listing 14.5 `unsynch/UnsynchBankTest.java`

```java
package unsynch;

/**
 * This program shows data corruption when multiple threads access a data structure.
 * @version 1.31 2015-06-21
 * @author Cay Horstmann
 */
public class UnsynchBankTest
{
   public static final int NACCOUNTS = 100;
   public static final double INITIAL_BALANCE = 1000;
   public static final double MAX_AMOUNT = 1000;
   public static final int DELAY = 10;

   public static void main(String[] args)
   {
      Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
      for (int i = 0; i < NACCOUNTS; i++)
      {
         int fromAccount = i;
         Runnable r = () -> {
            try
            {
               while (true)
               {
                  int toAccount = (int) (bank.size() * Math.random());
                  double amount = MAX_AMOUNT * Math.random();
                  bank.transfer(fromAccount, toAccount, amount);
                  Thread.sleep((int) (DELAY * Math.random()));
               }
            }
            catch (InterruptedException e)
            {
            }
         };
         Thread t = new Thread(r);
         t.start();
      }
   }
}
```

Listing 14.6 `unsynch/Bank.java`

```
package unsynch;

import java.util.*;

/**
 * A bank with a number of bank accounts.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
   private final double[] accounts;

   /**
    * Constructs the bank.
    * @param n the number of accounts
    * @param initialBalance the initial balance for each account
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      Arrays.fill(accounts, initialBalance);
   }

   /**
    * Transfers money from one account to another.
    * @param from the account to transfer from
    * @param to the account to transfer to
    * @param amount the amount to transfer
    */
   public void transfer(int from, int to, double amount)
   {
      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());
   }

```

(Continues)

Listing 14.6 *(Continued)*

```
   /**
    * Gets the sum of all account balances.
    * @return the total balance
    */
   public double getTotalBalance()
   {
      double sum = 0;

      for (double a : accounts)
         sum += a;

      return sum;
   }

   /**
    * Gets the number of accounts in the bank.
    * @return the number of accounts
    */
   public int size()
   {
      return accounts.length;
   }
}
```

14.5.2 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 preempted. 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 14.4).

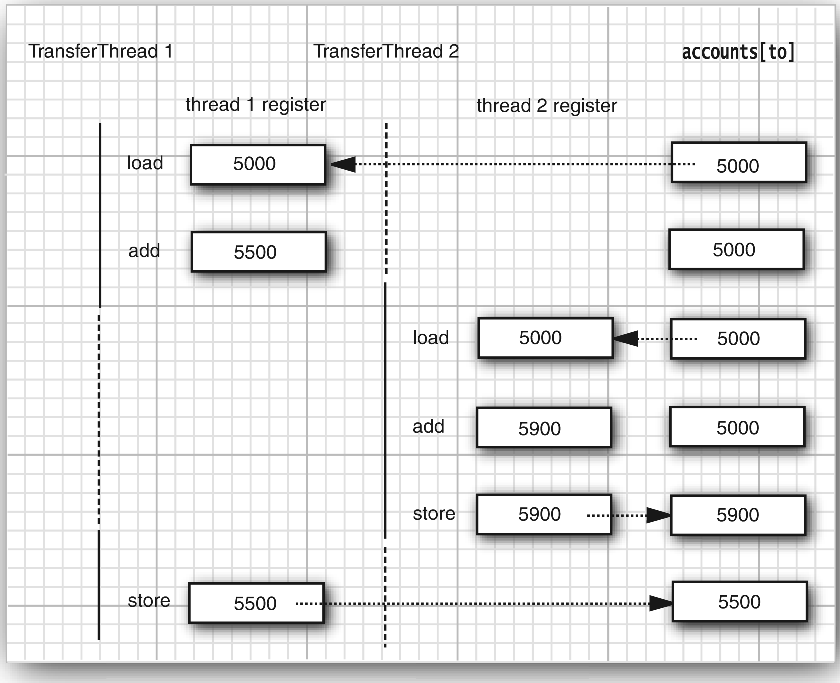


Figure 14.4 Simultaneous access by two threads

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 any instruction.

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 go away completely. If you run lots of threads on a heavily loaded machine, 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, the state of the bank account object would never be corrupted.

14.5.3 Lock Objects

There are two mechanisms for protecting a code block from concurrent access. The Java language provides a `synchronized` keyword for this purpose, and Java SE 5.0 introduced the `ReentrantLock` class. The `synchronized` keyword automatically provides a lock as well as an associated "condition," which makes it powerful and convenient for most cases that require explicit locking. However, we believe that it is easier to understand the `synchronized` keyword after you have seen locks and conditions in isolation. The `java.util.concurrent` framework provides separate classes for these fundamental mechanisms, which we explain here and in Section 14.5.4, "Condition Objects," on p. 872. Once you have understood these building blocks, we present the `synchronized` keyword in Section 14.5.5, "The `synchronized` Keyword," on p. 878.

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 deactivated until the first thread unlocks the lock object.

CAUTION: It is critically important that the `unlock` operation is enclosed in a `finally` clause. If the code in the critical section throws an exception, the lock must be unlocked. Otherwise, the other threads will be blocked forever.

NOTE: When you use locks, you cannot use the try-with-resources statement. First off, the unlock method isn't called `close`. But even if it was renamed, the try-with-resources statement wouldn't work. Its header expects the declaration of a new variable. But when you use a lock, you want to keep using the same variable that is shared among threads.

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

```
public class Bank
{
   private Lock bankLock = new ReentrantLock(); // ReentrantLock implements the Lock interface
   . . .
   public void transfer(int from, int to, int amount)
   {
      bankLock.lock();
```

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

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 14.5).

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, each thread acquires a different lock and neither thread is blocked. This is as it should be, because the threads cannot interfere with one 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 has 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 protected by a lock can call another method that uses the same locks.

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 update or inspect a shared object, so you can be assured that these operations run to completion before another thread can use the same object.

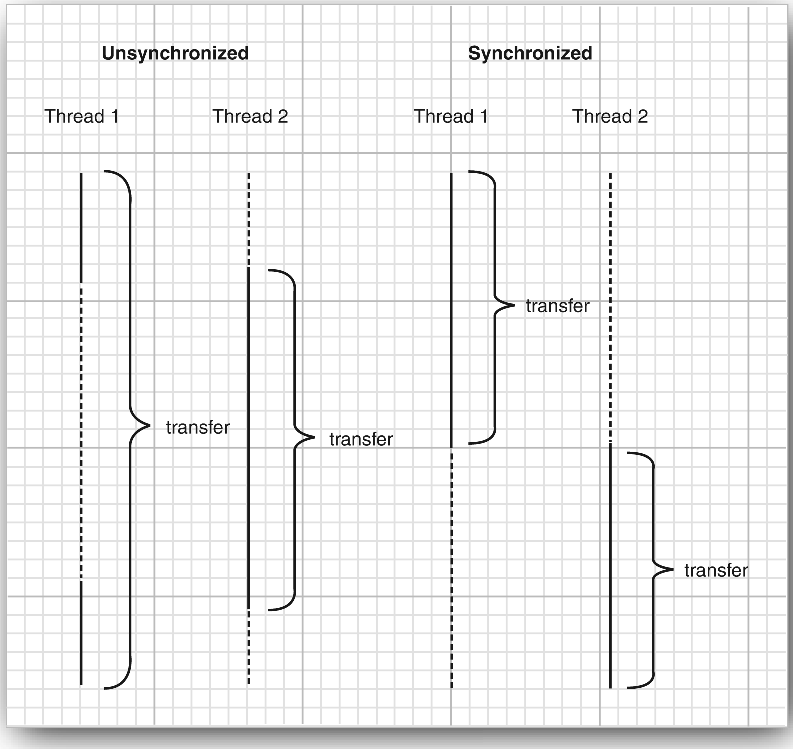


Figure 14.5 Comparison of unsynchronized and synchronized threads

CAUTION: Be careful to ensure that the code in a critical section is not bypassed by throwing an exception. If an exception is thrown before the end of the section, 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.
- `ReentrantLock(boolean fair)`

  constructs a lock with the given fairness policy. 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.

CAUTION: It sounds nice to be fair, but fair locks are *a lot slower* than regular locks. You should only enable fair locking if you truly know what you are doing and have a specific reason to consider fairness essential for your program. 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, it doesn't get the chance to be treated fairly by the lock.

14.5.4 Condition Objects

Often, a thread enters a critical section only to discover that it can't proceed until a condition is fulfilled. 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 no other thread can modify the balance between the test and the transfer action. 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
{
   private Condition sufficientFunds;
   . . .
   public Bank()
   {
      . . .
      sufficientFunds = bankLock.newCondition();
   }
}
```

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

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

The current thread is now deactivated 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* made runnable when the lock is available. Instead, it stays deactivated until another thread has called the signalAll method on the same condition.

When another thread has transferred money, it should call

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

This call reactivates all threads 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 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 reactivating itself. It puts its faith in the other threads. If none of them bother to reactivate 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, 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 relinquished the lock.

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, 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 if it owns the lock of the condition.

If you run the sample program in Listing 14.7, you will notice that nothing ever goes wrong. The total balance stays at $100,000 forever. No account ever has a negative balance. (Again, 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.

In practice, using conditions correctly can be quite challenging. Before you start implementing your own condition objects, you should consider using one of the constructs described in Section 14.10, "Synchronizers," on p. 934.

Listing 14.7 synch/Bank.java

```
package synch;

import java.util.*;
import java.util.concurrent.locks.*;

/**
 * A bank with a number of bank accounts that uses locks for serializing access.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
```

(Continues)

Listing 14.7 *(Continued)*

```
   private final double[] accounts;
   private Lock bankLock;
   private Condition sufficientFunds;

   /**
    * Constructs the bank.
    * @param n the number of accounts
    * @param initialBalance the initial balance for each account
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      Arrays.fill(accounts, initialBalance);
      bankLock = new ReentrantLock();
      sufficientFunds = bankLock.newCondition();
   }

   /**
    * Transfers money from one account to another.
    * @param from the account to transfer from
    * @param to the account to transfer to
    * @param amount the amount to transfer
    */
   public void transfer(int from, int to, double amount) throws InterruptedException
   {
      bankLock.lock();
      try
      {
         while (accounts[from] < amount)
            sufficientFunds.await();
         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());
         sufficientFunds.signalAll();
      }
      finally
      {
         bankLock.unlock();
      }
   }

   /**
    * Gets the sum of all account balances.
    * @return the total balance
    */
   public double getTotalBalance()
   {
      bankLock.lock();
      try
      {
         double sum = 0;

         for (double a : accounts)
            sum += a;

         return sum;
      }
      finally
      {
         bankLock.unlock();
      }
   }

   /**
    * Gets the number of accounts in the bank.
    * @return the number of accounts
    */
   public int size()
   {
      return accounts.length;
   }
}
```

java.util.concurrent.locks.Lock 5.0

- `Condition newCondition()`

  returns a condition object 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.

14.5.5 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.

The `Lock` and `Condition` interfaces give programmers a high degree of control over locking. However, in most situations, you don't need that control—you can use a mechanism that is built into the Java language. Ever since version 1.0, *every object* in Java has an intrinsic lock. If a method is declared with the `synchronized` keyword, the object's lock protects the entire method. That is, to call the method, a thread must acquire the intrinsic object lock.

In other words,

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

is the equivalent of

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

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

The intrinsic 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

```
intrinsicCondition.await();
intrinsicCondition.signalAll();
```

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

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

```
class Bank
{
   private double[] accounts;

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

   public synchronized double getTotalBalance() { . . . }
}
```

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 intrinsic lock, and that the lock has an intrinsic condition. The lock manages the threads that try to enter a synchronized method. The condition manages the threads that have called wait.

TIP: Synchronized methods are relatively straightforward. However, beginners often struggle with conditions. Before you use wait/notifyAll, you should consider using one of the constructs described in Section 14.10, "Synchronizers," on p. 934.

It is also legal to declare static methods as synchronized. If such a method is called, it acquires the intrinsic 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. As a result, no other thread can call this or any other synchronized static method of the same class.

The intrinsic locks and conditions have some limitations. Among them:

- 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.

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

- 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, in Section 14.6, "Blocking Queues," on p. 898, you will see how to use a blocking queue to synchronize threads that work on a common task. You should also explore parallel streams—see Volume II, Chapter 1.
- If the `synchronized` keyword works for your situation, by all means, use it. You'll write less code and have less room for error. Listing 14.8 shows the bank example, implemented with synchronized methods.
- Use `Lock`/`Condition` if you really need the additional power that these constructs give you.

Listing 14.8 `synch2/Bank.java`

```java
package synch2;

import java.util.*;

/**
 * A bank with a number of bank accounts that uses synchronization primitives.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
   private final double[] accounts;

   /**
    * Constructs the bank.
    * @param n the number of accounts
    * @param initialBalance the initial balance for each account
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      Arrays.fill(accounts, initialBalance);
   }

   /**
    * Transfers money from one account to another.
    * @param from the account to transfer from
    * @param to the account to transfer to
    * @param amount the amount to transfer
    */
   public synchronized void transfer(int from, int to, double amount) throws InterruptedException
   {
      while (accounts[from] < amount)
         wait();
      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());
      notifyAll();
   }

   /**
    * Gets the sum of all account balances.
    * @return the total balance
    */
   public synchronized double getTotalBalance()
   {
      double sum = 0;

      for (double a : accounts)
         sum += a;

      return sum;
   }

   /**
    * Gets the number of accounts in the bank.
    * @return the number of accounts
    */
   public int size()
   {
      return accounts.length;
   }
}
```

`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.

(Continues)

java.lang.Object 1.0 *(Continued)*

- `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 or block. 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 or block. 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, not exceeding 1,000,000

14.5.6 Synchronized Blocks

As we just discussed, every Java object has a lock. A thread can acquire the lock by calling a synchronized method. There is a second mechanism for acquiring the lock: by entering a *synchronized block*. When a thread enters a block of the form

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

then it acquires the lock for `obj`.

You will sometimes find "ad hoc" locks, such as

```
public class Bank
{
   private double[] accounts;
   private Object lock = new Object();
   . . .
```

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

      }
   }
```

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

Sometimes, programmers use the lock of an object to implement additional atomic operations—a practice known as *client-side locking*. Consider, for example, the `Vector` class, which is a list whose methods are synchronized. Now suppose we stored our bank balances in a `Vector<Double>`. Here is a naive implementation of a `transfer` method:

```
   public void transfer(Vector<Double> accounts, int from, int to, int amount) // Error
   {
      accounts.set(from, accounts.get(from) - amount);
      accounts.set(to, accounts.get(to) + amount);
      System.out.println(. . .);
   }
```

The `get` and `set` methods of the `Vector` class are synchronized, but that doesn't help us. It is entirely possible for a thread to be preempted in the `transfer` method after the first call to `get` has been completed. Another thread may then store a different value into the same position. However, we can hijack the lock:

```
   public void transfer(Vector<Double> accounts, int from, int to, int amount)
   {
      synchronized (accounts)
      {
         accounts.set(from, accounts.get(from) - amount);
         accounts.set(to, accounts.get(to) + amount);
      }
      System.out.println(. . .);
   }
```

This approach works, but it is entirely dependent on the fact that the `Vector` class uses the intrinsic lock for all of its mutator methods. However, is this really a fact? The documentation of the `Vector` class makes no such promise. You have to carefully study the source code and hope that future versions do not introduce unsynchronized mutators. As you can see, client-side locking is very fragile and not generally recommended.

14.5.7 The Monitor Concept

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. Since 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] >= amount` 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 intrinsic lock and an intrinsic condition. If a method is declared with the `synchronized` keyword, it acts like a monitor method. The condition variable is accessed by calling `wait`/`notifyAll`/`notify`.

However, a Java object differs from a monitor in three important ways, compromising thread safety:

- Fields are not required to be `private`.
- Methods are not required to be `synchronized`.
- The intrinsic lock is available to clients.

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].

14.5.8 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, 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 `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 `www.ibm.com/developerworks/library/j-jtp02244`.

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. As we already discussed, you can use a lock:

```
private boolean done;
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
```

Perhaps it is not a good idea to use the intrinsic object lock. The `isDone` and `setDone` methods can block if another thread has locked the object. If that is a concern, one can use a separate lock just for this variable. But this is getting to be a lot of trouble.

In this case, it is reasonable to declare the field as `volatile`:

```
private volatile boolean done;
public boolean isDone() { return done; }
public void setDone() { done = true; }
```

The compiler will insert the appropriate code to ensure that a change to the `done` variable in one thread is visible from any other thread that reads the variable.

CAUTION: Volatile variables do not provide any atomicity. For example, the method

```
public void flipDone() { done = !done; } // not atomic
```

is not guaranteed to flip the value of the field. There is no guarantee that the reading, flipping, and writing is uninterrupted.

14.5.9 Final Variables

As you saw in the preceding section, you cannot safely read a field from multiple threads unless you use locks or the `volatile` modifier.

There is one other situation in which it is safe to access a shared field—when it is declared `final`. Consider

```
final Map<String, Double> accounts = new HashMap<>();
```

Other threads get to see the `accounts` variable after the constructor has finished.

Without using `final`, there would be no guarantee that other threads would see the updated value of `accounts`—they might all see `null`, not the constructed `HashMap`.

Of course, the operations on the map are not thread safe. If multiple threads mutate and read the map, you still need synchronization.

14.5.10 Atomics

You can declare shared variables as `volatile` provided you perform no operations other than assignment.

There are a number of classes in the `java.util.concurrent.atomic` package that use efficient machine-level instructions to guarantee atomicity of other operations without using locks. For example, the `AtomicInteger` class has methods `incrementAndGet` and `decrementAndGet` that atomically increment or decrement an integer. For example, you can safely generate a sequence of numbers like this:

```
public static AtomicLong nextNumber = new AtomicLong();
// In some thread...
long id = nextNumber.incrementAndGet();
```

The `incrementAndGet` method atomically increments the `AtomicLong` and returns the post-increment value. That is, the operations of getting the value, adding 1, setting it, and producing the new value cannot be interrupted. It is guaranteed that the correct value is computed and returned, even if multiple threads access the same instance concurrently.

There are methods for atomically setting, adding, and subtracting values, but if you want to make a more complex update, you have to use the `compareAndSet` method. For example, suppose you want to keep track of the largest value that is observed by different threads. The following won't work:

```
public static AtomicLong largest = new AtomicLong();
// In some thread...
largest.set(Math.max(largest.get(), observed)); // Error--race condition!
```

This update is not atomic. Instead, compute the new value and use `compareAndSet` in a loop:

```
do {
   oldValue = largest.get();
   newValue = Math.max(oldValue, observed);
} while (!largest.compareAndSet(oldValue, newValue));
```

If another thread is also updating `largest`, it is possible that it has beat this thread to it. Then `compareAndSet` will return `false` without setting the new value. In that case, the loop tries again, reading the updated value and trying to change it. Eventually, it will succeed replacing the existing value with the new one. This sounds tedious, but the `compareAndSet` method maps to a processor operation that is faster than using a lock.

In Java SE 8, you don't have to write the loop boilerplate any more. Instead, you provide a lambda expression for updating the variable, and the update is done for you. In our example, we can call

```
largest.updateAndGet(x -> Math.max(x, observed));
```

or

```
largest.accumulateAndGet(observed, Math::max);
```

The `accumulateAndGet` method takes a binary operator that is used to combine the atomic value and the supplied argument.

There are also methods `getAndUpdate` and `getAndAccumulate` that return the old value.

NOTE: These methods are also provided for the classes `AtomicInteger`, `AtomicIntegerArray`, `AtomicIntegerFieldUpdater`, `AtomicLongArray`, `AtomicLongFieldUpdater`, `AtomicReference`, `AtomicReferenceArray`, and `AtomicReferenceFieldUpdater`.

When you have a very large number of threads accessing the same atomic values, performance suffers because the optimistic updates require too many retries. Java SE 8 provides classes `LongAdder` and `LongAccumulator` to solve this problem. A `LongAdder` is composed of multiple variables whose collective sum is the current value. Multiple threads can update different summands, and new summands are automatically provided when the number of threads increases. This is efficient in the common situation where the value of the sum is not needed until after all work has been done. The performance improvement can be substantial.

If you anticipate high contention, you should simply use a `LongAdder` instead of an `AtomicLong`. The method names are slightly different. Call `increment` to increment a counter or `add` to add a quantity, and `sum` to retrieve the total.

```
final LongAdder adder = new LongAdder();
for (. . .)
   pool.submit(() -> {
      while (. . .) {
         . . .
         if (. . .) adder.increment();
      }
   });
. . .
long total = adder.sum());
```

NOTE: Of course, the `increment` method does *not* return the old value. Doing that would undo the efficiency gain of splitting the sum into multiple summands.

The `LongAccumulator` generalizes this idea to an arbitrary accumulation operation. In the constructor, you provide the operation, as well as its neutral element. To incorporate new values, call `accumulate`. Call `get` to obtain the current value. The following has the same effect as a `LongAdder`:

```
LongAccumulator adder = new LongAccumulator(Long::sum, 0);
// In some thread...
adder.accumulate(value);
```

Internally, the accumulator has variables $a_1, a_2, \ldots, a_n$. Each variable is initialized with the neutral element (`0` in our example).

When `accumulate` is called with value v, then one of them is atomically updated as $a_i = a_i \ op \ v$, where op is the accumulation operation written in infix form. In our example, a call to `accumulate` computes $a_i = a_i + v$ for some i.

The result of `get` is $a_1 \ op \ a_2 \ op \ . . . \ op \ a_n$. In our example, that is the sum of the accumulators, $a_1 + a_2 + . . . + a_n$.

If you choose a different operation, you can compute maximum or minimum. In general, the operation must be associative and commutative. That means that the final result must be independent of the order in which the intermediate values were combined.

There are also `DoubleAdder` and `DoubleAccumulator` that work in the same way, except with `double` values.

14.5.11 Deadlocks

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

1. Account 1: $200
2. Account 2: $300
3. Thread 1: Transfer $300 from Account 1 to Account 2
4. Thread 2: Transfer $400 from Account 2 to Account 1

As Figure 14.6 indicates, Threads 1 and 2 are clearly blocked. Neither can proceed because the balances in Accounts 1 and 2 are insufficient.

It is possible that all threads get 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. Since there are 100 accounts and a total of $100,000 in them, at least one of the accounts must have must have at least $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 runnable with a `max` value of `2 * INITIAL_BALANCE` and run the program. The program will run for a while and then hang.

TIP: When the program hangs, press Ctrl+\. You will get a thread dump that lists all threads. Each thread has a stack trace, telling you where it is currently blocked. Alternatively, run `jconsole`, as described in Chapter 7, and consult the Threads panel (see Figure 14.7).

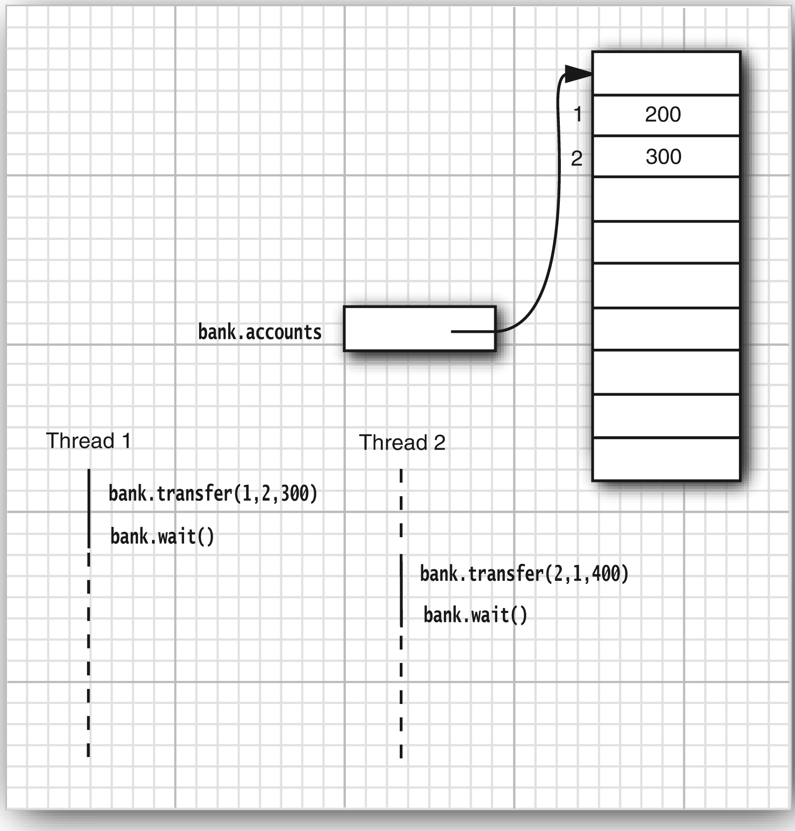
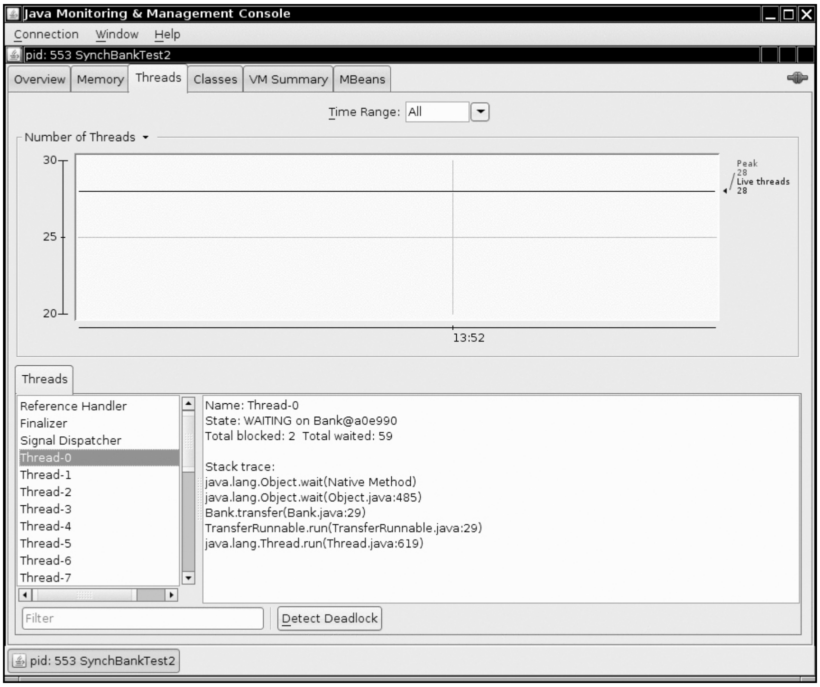


Figure 14.6 A deadlock situation

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 eventually hangs. (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,



```
Name: Thread-0
State: WAITING on Bank@a0e990
Total blocked: 2  Total waited: 59

Stack trace:
java.lang.Object.wait(Native Method)
java.lang.Object.wait(Object.java:485)
Bank.transfer(Bank.java:29)
TransferRunnable.run(TransferRunnable.java:29)
java.lang.Thread.run(Thread.java:619)
```

Figure 14.7 The Threads panel in jconsole

all threads can be blocked. Consider the following sample scenario of a developing deadlock:

1. Account 1: $1,990
2. All other accounts: $990 each
3. Thread 1: Transfer $995 from Account 1 to Account 2
4. 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:

1. Account 1: $995
2. Account 2: $1,985
3. 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,

1. Thread 1: Transfer $997 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.

14.5.12 Thread-Local Variables

In the preceding sections, we discussed the risks of sharing variables between threads. Sometimes, you can avoid sharing by giving each thread its own instance, using the `ThreadLocal` helper class. For example, the `SimpleDateFormat` class is not thread safe. Suppose we have a static variable

```
public static final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
```

If two threads execute an operation such as

```
String dateStamp = dateFormat.format(new Date());
```

then the result can be garbage since the internal data structures used by the `dateFormat` can be corrupted by concurrent access. You could use synchronization, which is expensive, or you could construct a local `SimpleDateFormat` object whenever you need it, but that is also wasteful.

To construct one instance per thread, use the following code:

```
public static final ThreadLocal<SimpleDateFormat> dateFormat =
   ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyy-MM-dd"));
```

To access the actual formatter, call

```
String dateStamp = dateFormat.get().format(new Date());
```

The first time you call `get` in a given thread, the lambda in the constructor is called. From then on, the `get` method returns the instance belonging to the current thread.

A similar problem is the generation of random numbers in multiple threads. The `java.util.Random` class is thread safe. But it is still inefficient if multiple threads need to wait for a single shared generator.

You could use the `ThreadLocal` helper to give each thread a separate generator, but Java SE 7 provides a convenience class for you. Simply make a call such as

```
int random = ThreadLocalRandom.current().nextInt(upperBound);
```

The call `ThreadLocalRandom.current()` returns an instance of the `Random` class that is unique to the current thread.

`java.lang.ThreadLocal<T>` 1.2

- `T get()`

  Gets the current value of this thread. If `get` is called for the first time, the value is obtained by calling `initialize`.
- `protected initialize()`

  Override this method to supply an initial value. By default, this method returns `null`.
- `void set(T t)`

  Sets a new value for this thread.
- `void remove()`

  Removes the value for this thread.
- `static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier)` 8

  Creates a thread local variable whose initial value is produced by invoking the given supplier.

`java.util.concurrent.ThreadLocalRandom` 7

- `static ThreadLocalRandom current()`

  returns an instance of the `Random` class that is unique to the current thread.

14.5.13 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.

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, 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`.

14.5.14 Read/Write Locks

The `java.util.concurrent.locks` package defines two lock classes, the `ReentrantLock` that we already discussed and the `ReentrantReadWriteLock`. 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 a read/write lock:

1. Construct a `ReentrantReadWriteLock` object:

```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```

2. Extract the 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.

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

The initial release of Java 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.

The `stop`, `suspend`, and `resume` methods have been deprecated. 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 `TransferRunnable` 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 intrinsic 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, 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(event -> {
   for (int i = 0; i < threads.length; i++)
      threads[i].suspend(); // Don't do this
});
resumeButton.addActionListener(event -> {
   for (int i = 0; i < threads.length; i++)
      threads[i].resume();
});
```

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

As you will see in Section 14.11, "Threads and Swing," on p. 937, 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—in a place where your thread doesn't lock objects that other threads need. When your thread finds that the `suspendRequested` variable has been set, it should keep waiting until it becomes available again.

14.6 Blocking Queues

You have now seen the low-level building blocks that form the foundations of concurrent programming in Java. However, for practical programming, you want to stay away from the low-level constructs whenever possible. It is much easier and safer to use higher-level structures that have been implemented by concurrency experts.

Many threading problems can be formulated elegantly and safely by using one or more queues. Producer threads insert items into the queue, and consumer threads retrieve them. The queue lets you safely hand over data from one thread to another. For example, consider our bank transfer program. Instead of accessing the bank object directly, the transfer threads insert transfer instruction objects into a queue. Another thread removes the instructions from the queue and carries out the transfers. Only that thread has access to the internals of the bank object. No synchronization is necessary. (Of course, the implementors of the thread-safe queue classes had to worry about locks and conditions, but that was their problem, not yours.)

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 into 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 14.1 shows the methods for blocking queues.

The blocking queue methods fall into three categories that differ by the action they perform when the queue is full or empty. If you use the queue as a thread management tool, use the `put` and `take` methods. 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.

Table 14.1 Blocking Queue Methods

Method	Normal Action	Action in Special Circumstances
`add`	Adds an element	Throws an `IllegalStateException` if the queue is full
`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
`peek`	Returns the head element	Returns `null` if the queue is empty
`poll`	Removes and returns the head element	Returns `null` if the queue is empty
`put`	Adds an element	Blocks if the queue is full
`remove`	Removes and returns the head element	Throws a `NoSuchElementException` if the queue is empty
`take`	Removes and returns the head element	Blocks if the queue is empty

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 returns `true`; otherwise, it returns `false` when it times out. Similarly, the call

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

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

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 several variations of blocking queues. By default, the `LinkedBlockingQueue` has no upper bound on its capacity, but a maximum capacity can be optionally specified. The `LinkedBlockingDeque` is a double-ended version. 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. (See Chapter 9 for more information on priority queues.)

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.

Java SE 7 adds a `TransferQueue` interface that allows a producer thread to wait until a consumer is ready to take on an item. When a producer calls

```
q.transfer(item);
```

the call blocks until another thread removes it. The `LinkedTransferQueue` class implements this interface.

The program in Listing 14.9 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.

Listing 14.9 `blockingQueue/BlockingQueueTest.java`

```
package blockingQueue;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;
```

```

/**
 * @version 1.02 2015-06-21
 * @author Cay Horstmann
 */
public class BlockingQueueTest
{
   private static final int FILE_QUEUE_SIZE = 10;
   private static final int SEARCH_THREADS = 100;
   private static final File DUMMY = new File("");
   private static BlockingQueue<File> queue = new ArrayBlockingQueue<>(FILE_QUEUE_SIZE);

   public static void main(String[] args)
   {
      try (Scanner in = new Scanner(System.in))
      {
         System.out.print("Enter base directory (e.g. /opt/jdk1.8.0/src): ");
         String directory = in.nextLine();
         System.out.print("Enter keyword (e.g. volatile): ");
         String keyword = in.nextLine();

         Runnable enumerator = () -> {
            try
            {
               enumerate(new File(directory));
               queue.put(DUMMY);
            }
            catch (InterruptedException e)
            {
            }
         };

         new Thread(enumerator).start();
         for (int i = 1; i <= SEARCH_THREADS; i++) {
            Runnable searcher = () -> {
               try
               {
                  boolean done = false;
                  while (!done)
                  {
                     File file = queue.take();
                     if (file == DUMMY)
                     {
                        queue.put(file);
                        done = true;
                     }
                     else search(file, keyword);
                  }
               }
```

(Continues)

Listing 14.9 *(Continued)*

```
               catch (IOException e)
               {
                  e.printStackTrace();
               }
               catch (InterruptedException e)
               {
               }
            };
            new Thread(searcher).start();
         }
      }
   }

   /**
    * Recursively enumerates all files in a given directory and its subdirectories.
    * @param directory the directory in which to start
    */
   public static void enumerate(File directory) throws InterruptedException
   {
      File[] files = directory.listFiles();
      for (File file : files)
      {
         if (file.isDirectory()) enumerate(file);
         else queue.put(file);
      }
   }

   /**
    * Searches a file for a given keyword and prints all matching lines.
    * @param file the file to search
    * @param keyword the keyword to search for
    */
   public static void search(File file, String keyword) throws IOException
   {
      try (Scanner in = new Scanner(file, "UTF-8"))
      {
         int lineNumber = 0;
         while (in.hasNextLine())
         {
            lineNumber++;
            String line = in.nextLine();
            if (line.contains(keyword))
               System.out.printf("%s:%d:%s%n", file.getPath(), lineNumber, line);
         }
      }
   }
}
```

java.util.concurrent.ArrayBlockingQueue<E> 5.0

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

  constructs 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
java.util.concurrent.LinkedBlockingDeque<E> 6

- `LinkedBlockingQueue()`
- `LinkedBlockingDeque()`

  constructs an unbounded blocking queue or deque, implemented as a linked list.
- `LinkedBlockingQueue(int capacity)`
- `LinkedBlockingDeque(int capacity)`

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

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

- `DelayQueue()`

  constructs an unbounded 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.
- `E take()`

  removes and returns the head element, blocking if necessary.
- `boolean offer(E element, long time, TimeUnit unit)`

  adds the given element and returns `true` if successful, blocking if necessary until the element has been added or the time has elapsed.
- `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.concurrent.BlockingDeque<E>`* 6**

- `void putFirst(E element)`
- `void putLast(E element)`

  adds the element, blocking if necessary.
- `E takeFirst()`
- `E takeLast()`

  removes and returns the head or tail element, blocking if necessary.

(Continues)

java.util.concurrent.BlockingDeque<E>* 6 *(Continued)

- `boolean offerFirst(E element, long time, TimeUnit unit)`
- `boolean offerLast(E element, long time, TimeUnit unit)`

  adds the given element and returns `true` if successful, blocking if necessary until the element has been added or the time has elapsed.

- `E pollFirst(long time, TimeUnit unit)`
- `E pollLast(long time, TimeUnit unit)`

  removes and returns the head or tail element, blocking if necessary until an element is available or the time has elapsed. Returns `null` upon failure.

***java.util.concurrent.TransferQueue<E>* 7**

- `void transfer(E element)`
- `boolean tryTransfer(E element, long time, TimeUnit unit)`

  transfers a value, or tries transferring it with a given timeout, blocking until another thread has removed the item. The second method returns `true` if successful.

14.7 Thread-Safe Collections

If multiple threads concurrently modify a data structure, such as a hash table, it is easy to damage that data structure. (See Chapter 9 for more information on hash tables.) For example, one thread may begin to insert a new element. Suppose it is preempted 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.

14.7.1 Efficient Maps, Sets, and Queues

The `java.util.concurrent` package supplies efficient implementations for maps, sorted sets, and queues: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, and `ConcurrentLinkedQueue`.

These collections use sophisticated algorithms that minimize contention by allowing concurrent access to different parts of the data structure.

Unlike most collections, the `size` method of these classes does not necessarily operate in constant time. Determining the current size of one of these collections usually requires traversal.

NOTE: Some applications use humongous concurrent hash maps, so large that the `size` method is insufficient because it returns an `int`. What is one to do with a map that has over two billion entries? Java SE 8 introduces a `mappingCount` method that returns the size as a `long`.

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 a `ConcurrentModificationException`.

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 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.

NOTE: A hash map keeps all entries with the same hash code in the same "bucket." Some applications use poor hash functions, and as a result all entries end up in a small number of buckets, severely degrading performance. Even generally reasonable hash functions, such as that of the `String` class, can be problematic. For example, an attacker can slow down a program by crafting a large number of strings that hash to the same value. As of Java SE 8, the concurrent hash map organizes the buckets as trees, not lists, when the key type implements `Comparable`, guaranteeing $O(\log(n))$ performance.

---

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.ConcurrentSkipListSet<E> 6

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`

  constructs a sorted set that can be safely accessed by multiple threads. The first constructor requires that the elements implement the `Comparable` interface.

java.util.concurrent.ConcurrentHashMap<K, V> 5.0
java.util.concurrent.ConcurrentSkipListMap<K, V> 6

- `ConcurrentHashMap<K, V>()`
- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

  constructs 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.

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

  constructs a sorted map that can be safely accessed by multiple threads. The first constructor requires that the keys implement the `Comparable` interface.

14.7.2 Atomic Update of Map Entries

The original version of `ConcurrentHashMap` only had a few methods for atomic updates, which made for somewhat awkward programming. Suppose we want to count how often certain features are observed. As a simple example, suppose multiple threads encounter words, and we want to count their frequencies.

Can we use a `ConcurrentHashMap<String, Long>`? Consider the code for incrementing a count. Obviously, the following is not thread safe:

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // Error--might not replace oldValue
```

Another thread might be updating the exact same count at the same time.

NOTE: Some programmers are surprised that a supposedly thread-safe data structure permits operations that are not thread safe. But there are two entirely different considerations. If multiple threads modify a plain `HashMap`, they can destroy the internal structure (an array of linked lists). Some of the links may go missing, or even go in circles, rendering the data structure unusable. That will never happen with a `ConcurrentHashMap`. In the example above, the code for `get` and `put` will never corrupt the data structure. But, since the sequence of operations is not atomic, the result is not predictable.

A classic trick is to use the `replace` operation, which atomically replaces an old value with a new one, provided that no other thread has come before and replaced the old value with something else. You have to keep doing it until `replace` succeeds:

```
do
{
   oldValue = map.get(word);
   newValue = oldValue == null ? 1 : oldValue + 1;
} while (!map.replace(word, oldValue, newValue));
```

Alternatively, you can use a `ConcurrentHashMap<String, AtomicLong>` or, with Java SE 8, a `ConcurrentHashMap<String, LongAdder>`. Then the update code is:

```
map.putIfAbsent(word, new LongAdder());
map.get(word).increment();
```

The first statement ensures that there is a `LongAdder` present that we can increment atomically. Since `putIfAbsent` returns the mapped value (either the existing one or the newly put one), you can combine the two statements:

```
map.putIfAbsent(word, new LongAdder()).increment();
```

Java SE 8 provides methods that make atomic updates more convenient. The `compute` method is called with a key and a function to compute the new value. That function receives the key and the associated value, or `null` if there is none, and it computes the new value. For example, here is how we can update a map of integer counters:

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```

NOTE: You cannot have `null` values in a `ConcurrentHashMap`. There are many methods that use a `null` value as an indication that a given key is not present in the map.

There are also variants `computeIfPresent` and `computeIfAbsent` that only compute a new value when there is already an old one, or when there isn't yet one. A map of `LongAdder` counters can be updated with

```
map.computeIfAbsent(word, k -> new LongAdder()).increment();
```

That is almost like the call to `putIfAbsent` that you saw before, but the `LongAdder` constructor is only called when a new counter is actually needed.

You often need to do something special when a key is added for the first time. The `merge` method makes this particularly convenient. It has a parameter for the initial value that is used when the key is not yet present. Otherwise, the function that you supplied is called, combining the existing value and the initial value. (Unlike `compute`, the function does *not* process the key.)

```
map.merge(word, 1L, (existingValue, newValue) -> existingValue + newValue);
```

or, more simply,

```
map.merge(word, 1L, Long::sum);
```

It doesn't get more concise than that.

NOTE: If the function that is passed to `compute` or `merge` returns `null`, the existing entry is removed from the map.

CAUTION: When you use `compute` or `merge`, keep in mind that the function that you supply should not do a lot of work. While that function runs, some other updates to the map may be blocked. Of course, that function should also not update other parts of the map.

14.7.3 Bulk Operations on Concurrent Hash Maps

Java SE 8 provides bulk operations on concurrent hash maps that can safely execute even while other threads operate on the map. The bulk operations traverse the map and operate on the elements they find as they go along. No effort is made to freeze a snapshot of the map in time. Unless you happen to know that the map is not being modified while a bulk operation runs, you should treat its result as an approximation of the map's state.

There are three kinds of operations:

- `search` applies a function to each key and/or value, until the function yields a non-null result. Then the search terminates and the function's result is returned.
- `reduce` combines all keys and/or values, using a provided accumulation function.
- `forEach` applies a function to all keys and/or values.

Each operation has four versions:

- *operation*`Keys`: operates on keys.
- *operation*`Values`: operates on values.
- *operation*: operates on keys and values.
- *operation*`Entries`: operates on `Map.Entry` objects.

With each of the operations, you need to specify a *parallelism threshold*. If the map contains more elements than the threshold, the bulk operation is parallelized. If you want the bulk operation to run in a single thread, use a threshold of `Long.MAX_VALUE`. If you want the maximum number of threads to be made available for the bulk operation, use a threshold of `1`.

Let's look at the `search` methods first. Here are the versions:

```
U searchKeys(long threshold, BiFunction<? super K, ? extends U> f)
U searchValues(long threshold, BiFunction<? super V, ? extends U> f)
U search(long threshold, BiFunction<? super K, ? super V,? extends U> f)
U searchEntries(long threshold, BiFunction<Map.Entry<K, V>, ? extends U> f)
```

For example, suppose we want to find the first word that occurs more than 1,000 times. We need to search keys and values:

```
String result = map.search(threshold, (k, v) -> v > 1000 ? k : null);
```

Then `result` is set to the first match, or to `null` if the search function returns `null` for all inputs.

The `forEach` methods have two variants. The first one simply applies a *consumer* function for each map entry, for example

```
map.forEach(threshold,
   (k, v) -> System.out.println(k + " -> " + v));
```

The second variant takes an additional *transformer* function, which is applied first, and its result is passed to the consumer:

```
map.forEach(threshold,
   (k, v) -> k + " -> " + v, // Transformer
   System.out::println); // Consumer
```

The transformer can be used as a filter. Whenever the transformer returns `null`, the value is silently skipped. For example, here we only print the entries with large values:

```
map.forEach(threshold,
   (k, v) -> v > 1000 ? k + " -> " + v : null, // Filter and transformer
   System.out::println); // The nulls are not passed to the consumer
```

The `reduce` operations combine their inputs with an accumulation function. For example, here is how you can compute the sum of all values:

```
Long sum = map.reduceValues(threshold, Long::sum);
```

As with `forEach`, you can also supply a transformer function. Here we compute the length of the longest key:

```
Integer maxlength = map.reduceKeys(threshold,
   String::length, // Transformer
   Integer::max); // Accumulator
```

The transformer can act as a filter, by returning `null` to exclude unwanted inputs. Here, we count how many entries have value > 1000:

```
Long count = map.reduceValues(threshold,
   v -> v > 1000 ? 1L : null,
   Long::sum);
```

NOTE: If the map is empty, or all entries have been filtered out, the `reduce` operation returns `null`. If there is only one element, its transformation is returned, and the accumulator is not applied.

There are specializations for `int`, `long`, and `double` outputs with suffixes `ToInt`, `ToLong`, and `ToDouble`. You need to transform the input to a primitive value and specify a default value and an accumulator function. The default value is returned when the map is empty.

```
long sum = map.reduceValuesToLong(threshold,
   Long::longValue, // Transformer to primitive type
   0, // Default value for empty map
   Long::sum); // Primitive type accumulator
```

CAUTION: These specializations act differently from the object versions where there is only one element to be considered. Instead of returning the transformed element, it is accumulated with the default. Therefore, the default must be the neutral element of the accumulator.

14.7.4 Concurrent Set Views

Suppose you want a large, thread-safe set instead of a map. There is no `ConcurrentHashSet` class, and you know better than trying to create your own. Of course, you can use a `ConcurrentHashMap` with bogus values, but then you get a map, not a set, and you can't apply operations of the `Set` interface.

The static `newKeySet` method yields a `Set<K>` that is actually a wrapper around a `ConcurrentHashMap<K, Boolean>`. (All map values are `Boolean.TRUE`, but you don't actually care since you just use it as a set.)

```
Set<String> words = ConcurrentHashMap.<String>newKeySet();
```

Of course, if you have an existing map, the `keySet` method yields the set of keys. That set is mutable. If you remove the set's elements, the keys (and their values) are removed from the map. But it doesn't make sense to add elements to the key set, because there would be no corresponding values to add. Java SE 8 adds a second `keySet` method to `ConcurrentHashMap`, with a default value, to be used when adding elements to the set:

```
Set<String> words = map.keySet(1L);
words.add("Java");
```

If `"Java"` wasn't already present in `words`, it now has a value of one.

14.7.5 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 threads that iterate over the collection greatly outnumber 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.

14.7.6 Parallel Array Algorithms

As of Java SE 8, the `Arrays` class has a number of parallelized operations. The static `Arrays.parallelSort` method can sort an array of primitive values or objects. For example,

```
String contents = new String(Files.readAllBytes(
    Paths.get("alice.txt")), StandardCharsets.UTF_8); // Read file into string
String[] words = contents.split("[\\P{L}]+"); // Split along nonletters
Arrays.parallelSort(words);
```

When you sort objects, you can supply a `Comparator`.

```
Arrays.parallelSort(words, Comparator.comparing(String::length));
```

With all methods, you can supply the bounds of a range, such as

```
values.parallelSort(values.length / 2, values.length); // Sort the upper half
```

NOTE: At first glance, it seems a bit odd that these methods have `parallel` in their name, since the user shouldn't care how the sorting happens. However, the API designers wanted to make it clear that the sorting is parallelized. That way, users are on notice to avoid comparators with side effects.

The `parallelSetAll` method fills an array with values that are computed from a function. The function receives the element index and computes the value at that location.

```
Arrays.parallelSetAll(values, i -> i % 10);
   // Fills values with 0 1 2 3 4 5 6 7 8 9 0 1 2 . . .
```

Clearly, this operation benefits from being parallelized. There are versions for all primitive type arrays and for object arrays.

Finally, there is a `parallelPrefix` method that replaces each array element with the accumulation of the prefix for a given associative operation. Huh? Here is an example. Consider the array `[1, 2, 3, 4, . . .]` and the × operation. After executing `Arrays.parallelPrefix(values, (x, y) -> x * y)`, the array contains

```
[1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, . . .]
```

Perhaps surprisingly, this computation can be parallelized. First, join neighboring elements, as indicated here:

```
[1, 1 × 2, 3, 3 × 4, 5, 5 × 6, 7, 7 × 8]
```

The gray values are left alone. Clearly, one can make this computation in parallel in separate regions of the array. In the next step, update the indicated elements by multiplying them with elements that are one or two positions below:

```
[1, 1 × 2, 1 x 2 x 3, 1 x 2 x 3 x 4, 5, 5 x 6, 5 x 6 x 7, 5 x 6 x 7 x 8]
```

This can again be done in parallel. After log(n) steps, the process is complete. This is a win over the straightforward linear computation if sufficient processors are available. On special-purpose hardware, this algorithm is commonly used, and users of such hardware are quite ingenious in adapting it to a variety of problems.

14.7.7 Older Thread-Safe Collections

Ever since the initial release of Java, the Vector and Hashtable classes provided thread-safe implementations of a dynamic array and a hash table. These classes are now considered obsolete, having been 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<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

The methods of the resulting collections are protected by a lock, providing thread safe access.

You should make sure that no thread accesses the data structure through the original unsynchronized methods. The easiest way to ensure this is not to save any reference to the original object. Simply construct a collection and immediately pass it to the wrapper, as we did in our examples.

You still need to use "client-side" locking if you want to *iterate* over the collection while another thread has the opportunity to mutate it:

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

You must use the same code if you use a "for each" loop because the loop uses an iterator. Note that the iterator actually fails with a ConcurrentModificationException if another thread mutates the collection while the iteration is in progress. The synchronization is still required so that the concurrent modification can be reliably detected.

You are usually better off using the collections defined in the java.util.concurrent package instead of the synchronization wrappers. In particular, the ConcurrentHashMap map has been carefully implemented so that multiple threads can access it without blocking each other, provided they access different buckets. One exception is an array list that is frequently mutated. In that case, a synchronized ArrayList can outperform a CopyOnWriteArrayList.

java.util.Collections 1.2

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

  constructs a view of the collection whose methods are synchronized.

14.8 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 can start a computation, give someone the `Future` object, 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, `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, 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 Listing 14.10 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.

Listing 14.10 future/FutureTest.java

```java
package future;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class FutureTest
{
   public static void main(String[] args)
   {
      try (Scanner in = new Scanner(System.in))
      {
         System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
         String directory = in.nextLine();
         System.out.print("Enter keyword (e.g. volatile): ");
         String keyword = in.nextLine();

         MatchCounter counter = new MatchCounter(new File(directory), keyword);
         FutureTask<Integer> task = new FutureTask<>(counter);
         Thread t = new Thread(task);
         t.start();
         try
         {
            System.out.println(task.get() + " matching files.");
         }
         catch (ExecutionException e)
         {
            e.printStackTrace();
         }
         catch (InterruptedException e)
         {
         }
      }
   }
}

/**
 * This task counts the files in a directory and its subdirectories that contain a given keyword.
 */
class MatchCounter implements Callable<Integer>
{
```

(Continues)

Listing 14.10 *(Continued)*

```
   private File directory;
   private String keyword;

   /**
    * Constructs a MatchCounter.
    * @param directory the directory in which to start the search
    * @param keyword the keyword to look for
    */
   public MatchCounter(File directory, String keyword)
   {
      this.directory = directory;
      this.keyword = keyword;
   }

   public Integer call()
   {
      int count = 0;
      try
      {
         File[] files = directory.listFiles();
         List<Future<Integer>> results = new ArrayList<>();

         for (File file : files)
            if (file.isDirectory())
            {
               MatchCounter counter = new MatchCounter(file, keyword);
               FutureTask<Integer> task = new FutureTask<>(counter);
               results.add(task);
               Thread t = new Thread(task);
               t.start();
            }
            else
            {
               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
      {
         try (Scanner in = new Scanner(file, "UTF-8"))
         {
            boolean found = false;
            while (!found && in.hasNextLine())
            {
               String line = in.nextLine();
               if (line.contains(keyword)) found = true;
            }
            return found;
         }
      }
      catch (IOException e)
      {
         return false;
      }
   }
}
```

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.

(Continues)

java.util.concurrent.Future<V>* 5.0 *(Continued)

- `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`.

14.9 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, it should use a *thread pool* instead. 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, 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 14.2 for a summary.

Table 14.2 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 (similar to the Swing event dispatch thread).
`newScheduledThreadPool`	A fixed-thread pool for scheduled execution; a replacement for `java.util.Timer`.
`newSingleThreadScheduledExecutor`	A single-thread "pool" for scheduled execution.

14.9.1 Thread Pools

Let us look at the first three methods in Table 14.2 (we will discuss the remaining methods in Section 14.9.2, "Scheduled Execution," on p. 926). 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, 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 where 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 thread 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 thread 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 Listing 14.11 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.

Listing 14.11 `threadPool/ThreadPoolTest.java`

```java
package threadPool;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.02 2015-06-21
 * @author Cay Horstmann
 */
public class ThreadPoolTest
{
   public static void main(String[] args) throws Exception
   {
      try (Scanner in = new Scanner(System.in))
      {
         System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
         String directory = in.nextLine();
         System.out.print("Enter keyword (e.g. volatile): ");
         String keyword = in.nextLine();

         ExecutorService pool = Executors.newCachedThreadPool();

         MatchCounter counter = new MatchCounter(new File(directory), keyword, pool);
         Future<Integer> result = pool.submit(counter);

         try
         {
            System.out.println(result.get() + " matching files.");
         }
         catch (ExecutionException e)
         {
            e.printStackTrace();
         }
         catch (InterruptedException e)
         {
         }
         pool.shutdown();

         int largestPoolSize = ((ThreadPoolExecutor) pool).getLargestPoolSize();
         System.out.println("largest pool size=" + largestPoolSize);
      }
   }
}

/**
 * This task counts the files in a directory and its subdirectories that contain a given keyword.
 */
class MatchCounter implements Callable<Integer>
{
   private File directory;
   private String keyword;
   private ExecutorService pool;
   private int count;

   /**
    * Constructs a MatchCounter.
    * @param directory the directory in which to start the search
    * @param keyword the keyword to look for
    * @param pool the thread pool for submitting subtasks
    */
   public MatchCounter(File directory, String keyword, ExecutorService pool)
   {
      this.directory = directory;
      this.keyword = keyword;
      this.pool = pool;
   }

```

(Continues)

Listing 14.11 *(Continued)*

```
public Integer call()
{
   count = 0;
   try
   {
      File[] files = directory.listFiles();
      List<Future<Integer>> results = new ArrayList<>();

      for (File file : files)
         if (file.isDirectory())
         {
            MatchCounter counter = new MatchCounter(file, keyword, pool);
            Future<Integer> result = pool.submit(counter);
            results.add(result);
         }
         else
         {
            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
   {
         try (Scanner in = new Scanner(file, "UTF-8"))
         {
            boolean found = false;
            while (!found && in.hasNextLine())
            {
               String line = in.nextLine();
               if (line.contains(keyword)) found = true;
            }
            return found;
         }
      }
      catch (IOException e)
      {
         return false;
      }
   }
}
```

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.

14.9.2 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.

14.9.3 Controlling Groups of Tasks

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 is the one that finished most quickly. 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 attempting a factorization with 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, blocks until all of them complete, and returns a list of `Future` objects that represent the solutions to all tasks. You can process the results of the computation when they are available, like this:

```
List<Callable<T>> tasks = . . .;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
   processFurther(result.get());
```

A disadvantage of 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<T> service = new ExecutorCompletionService<>(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
   processFurther(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<V> 5.0

- `ExecutorCompletionService(Executor e)`

  constructs an executor completion service that collects the results of the given executor.
- `Future<V> submit(Callable<V> task)`
- `Future<V> submit(Runnable task, V result)`

  submits a task to the underlying executor.
- `Future<V> take()`

  removes the next completed result, blocking if no completed results are available.
- `Future<V> poll()`
- `Future<V> poll(long time, TimeUnit unit)`

  removes and returns the next completed result, or return `null` if no completed results are available. The second method waits for the given time.

14.9.4 The Fork-Join Framework

Some applications use a large number of threads that are mostly idle. An example would be a web server that uses one thread per connection. Other applications use one thread per processor core, in order to carry out computationally intensive tasks, such as image or video processing. The fork-join framework, which appeared in Java SE 7, is designed to support the latter. Suppose you have a processing task that naturally decomposes into subtasks, like this:

```
if (problemSize < threshold)
   solve problem directly
else
{
    break problem into subproblems
    recursively solve each subproblem
    combine the results
}
```

One example is image processing. To enhance an image, you can transform the top half and the bottom half. If you have enough idle processors, those operations can run in parallel. (You will need to do a bit of extra work along the strip that separates the two halves, but that's a technical detail.)

Here, we will discuss a simpler example. Suppose we want to count how many elements of an array fulfill a particular property. We cut the array in half, compute the counts of each half, and add them up.

To put the recursive computation in a form that is usable by the framework, supply a class that extends `RecursiveTask<T>` (if the computation produces a result of type `T`) or `RecursiveAction` (if it doesn't produce a result). Override the `compute` method to generate and invoke subtasks, and to combine their results.

```
class Counter extends RecursiveTask<Integer>
{
   . . .
   protected Integer compute()
   {
      if (to - from < THRESHOLD)
      {
         solve problem directly
      }
      else
      {
         int mid = (from + to) / 2;
         Counter first = new Counter(values, from, mid, filter);
         Counter second = new Counter(values, mid, to, filter);
         invokeAll(first, second);
         return first.join() + second.join();
      }
   }
}
```

Here, the `invokeAll` method receives a number of tasks and blocks until all of them have completed. The `join` method yields the result. Here, we apply `join` to each subtask and return the sum.

NOTE: There is also a `get` method for getting the current result, but it is less attractive since it can throw checked exceptions that we are not allowed to throw in the `compute` method.

Listing 14.12 shows the complete example.

Behind the scenes, the fork-join framework uses an effective heuristic for balancing the workload among available threads, called *work stealing*. Each worker thread has a deque (double-ended queue) for tasks. A worker thread pushes subtasks onto the head of its own deque. (Only one thread accesses the head, so no locking is required.) When a worker thread is idle, it "steals" a task from the tail of another deque. Since large subtasks are at the tail, such stealing is rare.

Listing 14.12 forkJoin/ForkJoinTest.java

```
package forkJoin;

import java.util.concurrent.*;
import java.util.function.*;

/**
 * This program demonstrates the fork-join framework.
 * @version 1.01 2015-06-21
 * @author Cay Horstmann
 */
public class ForkJoinTest
{
   public static void main(String[] args)
   {
      final int SIZE = 10000000;
      double[] numbers = new double[SIZE];
      for (int i = 0; i < SIZE; i++) numbers[i] = Math.random();
      Counter counter = new Counter(numbers, 0, numbers.length, x -> x > 0.5);
      ForkJoinPool pool = new ForkJoinPool();
      pool.invoke(counter);
      System.out.println(counter.join());
   }
}

class Counter extends RecursiveTask<Integer>
{
   public static final int THRESHOLD = 1000;
   private double[] values;
   private int from;
   private int to;
   private DoublePredicate filter;

   public Counter(double[] values, int from, int to, DoublePredicate filter)
   {
      this.values = values;
      this.from = from;
      this.to = to;
      this.filter = filter;
   }

   protected Integer compute()
   {
      if (to - from < THRESHOLD)
      {
         int count = 0;
         for (int i = from; i < to; i++)
         {
            if (filter.test(values[i])) count++;
         }
         return count;
      }
      else
      {
         int mid = (from + to) / 2;
         Counter first = new Counter(values, from, mid, filter);
         Counter second = new Counter(values, mid, to, filter);
         invokeAll(first, second);
         return first.join() + second.join();
      }
   }
}
```

14.9.5 Completable Futures

The traditional approach for dealing with nonblocking calls is to use event handlers, where the programmer registers a handler for the action that should occur after a task completes. Of course, if the next action is also asynchronous, the next action after that is in a different event handler. Even though the programmer thinks in terms of "first do step 1, then step 2, then step 3," the program logic becomes dispersed in different handlers. It gets worse when one has to add error handling. Suppose step 2 is "the user logs in." You may need to repeat that step since the user can mistype the credentials. Trying to implement such a control flow in a set of event handlers, or to understand it once it has been implemented, is challenging.

The `CompletableFuture` class of Java SE 8 provides an alternative approach. Unlike event handlers, completable futures can be *composed*.

For example, suppose we want to extract all links from a web page in order to build a web crawler. Let's say we have a method

```
public void CompletableFuture<String> readPage(URL url)
```

that yields the text of a web page when it becomes available. If the method

```
public static List<URL> getLinks(String page)
```

yields the URLs in an HTML page, you can schedule it to be called when the page is available:

```
CompletableFuture<String> contents = readPage(url);
CompletableFuture<List<URL>> links = contents.thenApply(Parser::getLinks);
```

The `thenApply` method doesn't block either. It returns another future. When the first future has completed, its result is fed to the `getLinks` method, and the return value of that method becomes the final result.

With completable futures, you just specify what you want to have done and in which order. It won't all happen right away, of course, but what is important is that all the code is in one place.

Conceptually, `CompletableFuture` is a simple API, but there are many variants of methods for composing completable futures. Let us first look at those that deal with a single future (see Table 14.3). (For each method shown, there are also two `Async` variants that I don't show. One of them uses a shared `ForkJoinPool`, and the other has an `Executor` parameter.) In the table, I use a shorthand notation for the ponderous functional interfaces, writing `T -> U` instead of `Function<? super T, U>`. These aren't actual Java types, of course.

You have already seen the `thenApply` method. The calls

```
CompletableFuture<U> future.thenApply(f);
CompletableFuture<U> future.thenApplyAsync(f);
```

return a future that applies `f` to the result of `future` when it is available. The second call runs `f` in yet another thread.

The `thenCompose` method, instead of taking a function `T -> U`, takes a function `T -> CompletableFuture<U>`. That sounds rather abstract, but it can be quite natural. Consider the action of reading a web page from a given URL. Instead of supplying a method

```
public String blockingReadPage(URL url)
```

it is more elegant to have that method return a future:

```
public CompletableFuture<String> readPage(URL url)
```

Now, suppose we have another method that gets the URL from user input, perhaps from a dialog that won't reveal the answer until the user has clicked the OK button. That, too, is an event in the future:

```
public CompletableFuture<URL> getURLInput(String prompt)
```

Here we have two functions `T -> CompletableFuture<U>` and `U -> CompletableFuture<V>`. Clearly, they compose to a function `T -> CompletableFuture<V>` if the second function is called when the first one has completed. That is exactly what `thenCompose` does.

The third method in Table 14.3 focuses on a different aspect that I have ignored so far: failure. When an exception is thrown in a `CompletableFuture`, it is captured and wrapped in an unchecked `ExecutionException` when the `get` method is called. But perhaps `get` is never called. In order to handle an exception, use the `handle` method. The supplied function is called with the result (or `null` if none) and the exception (or `null` if none), and it gets to make sense of the situation.

The remaining methods have `void` result and are normally used at the end of a processing pipeline.

Table 14.3 Adding an Action to a `CompletableFuture<T>` Object

Method	Parameter	Description
`thenApply`	`T -> U`	Apply a function to the result.
`thenCompose`	`T -> CompletableFuture<U>`	Invoke the function on the result and execute the returned future.
`handle`	`(T, Throwable) -> U`	Process the result or error.
`thenAccept`	`T -> void`	Like `thenApply`, but with `void` result.
`whenComplete`	`(T, Throwable) -> void`	Like `handle`, but with `void` result.
`thenRun`	`Runnable`	Execute the `Runnable` with `void` result.

Now let us turn to methods that combine multiple futures (see Table 14.4).

The first three methods run a `CompletableFuture<T>` and a `CompletableFuture<U>` action in parallel and combine the results.

The next three methods run two `CompletableFuture<T>` actions in parallel. As soon as one of them finishes, its result is passed on, and the other result is ignored.

Finally, the static `allOf` and `anyOf` methods take a variable number of completable futures and yield a `CompletableFuture<Void>` that completes when all of them, or any one of them, completes. No results are propagated.

Table 14.4 Combining Multiple Composition Objects

Method	Parameter	Description
`thenCombine`	`CompletableFuture<U>, (T, U) -> V`	Execute both and combine the results with the given function.
`thenAcceptBoth`	`CompletableFuture<U>, (T, U) -> void`	Like `thenCombine`, but with `void` result.
`runAfterBoth`	`CompletableFuture<?>, Runnable`	Execute the runnable after both complete.
`applyToEither`	`CompletableFuture<T>, T -> V`	When a result is available from one or the other, pass it to the given function.
`acceptEither`	`CompletableFuture<T>, T -> void`	Like `applyToEither`, but with `void` result.
`runAfterEither`	`CompletableFuture<?>, Runnable`	Execute the runnable after one or the other completes.
`static allOf`	`CompletableFuture<?>...`	Complete with `void` result after all given futures complete.
`static anyOf`	`CompletableFuture<?>...`	Complete with `void` result after any of the given futures completes.

NOTE: Technically speaking, the methods in this section accept parameters of type `CompletionStage`, not `CompletableFuture`. That is an interface with almost forty abstract methods, implemented only by `CompletableFuture`. The interface is provided so that third-party frameworks can implement it.

14.10 Synchronizers

The `java.util.concurrent` package contains several classes that help manage a set of collaborating threads—see Table 14.5. These mechanisms have "canned functionality" for common rendezvous patterns between threads. If you have a set of collaborating threads that follow 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 and conditions.

Table 14.5 Synchronizers

Class	What It Does	Notes
`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.	Use when a number of threads need to complete before their results can be used. The barrier can be reused after the waiting threads have been released.
`Phaser`	Like a cyclic barrier, but with a mutable party count.	Introduced in Java SE 7.
`CountDownLatch`	Allows a set of threads to wait until a count has been decremented to 0.	Use when one or more threads need to wait until a specified number of events have occurred.
`Exchanger`	Allows two threads to exchange objects when both are ready for the exchange.	Use when two threads work on two instances of the same data structure, with the first thread filling one instance and the second thread emptying the other.
`Semaphore`	Allows a set of threads to wait until permits are available for proceeding.	Use to restrict the total number of threads that can access a resource. If the permit count is one, use to block threads until another thread gives permission.
`SynchronousQueue`	Allows a thread to hand off an object to another thread.	Use to send an object from one thread to another when both are ready, without explicit synchronization.

14.10.1 Semaphores

Conceptually, a semaphore manages a number of *permits*. The number is supplied in the constructor. To proceed past the semaphore, a thread requests a permit by calling `acquire`. (There are no actual permit objects. The semaphore simply keeps a count.) Since only a fixed number of permits is available, a semaphore limits the number of threads that are allowed to pass. Other threads may issue permits by calling `release`. Moreover, a permit doesn't have to be released by the thread that acquires it. Any thread can release any number of permits, potentially increasing the number of permits beyond the initial count.

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. Usually, semaphores do not map directly to common application situations.

14.10.2 Countdown Latches

A `CountDownLatch` lets a set of threads wait until a count has reached zero. The countdown latch is one-time only. Once the count has reached 0, you cannot increment it again.

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.

You can then use a second latch to check when all worker threads are done. Initialize the latch with the number of threads. Each worker thread counts down that latch just before it terminates. Another thread that harvests the work results waits on the latch, and proceeds as soon as all workers have terminated.

14.10.3 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 results of the individual threads.

The barrier is called *cyclic* because it can be reused after all waiting threads have been released. In this regard, it differs from a CountDownLatch which can only be used once.

The Phaser class adds more flexibility, allowing you to vary the number of participating threads between phases.

14.10.4 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.

14.10.5 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.

14.11 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, 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, your user interface can become corrupted.

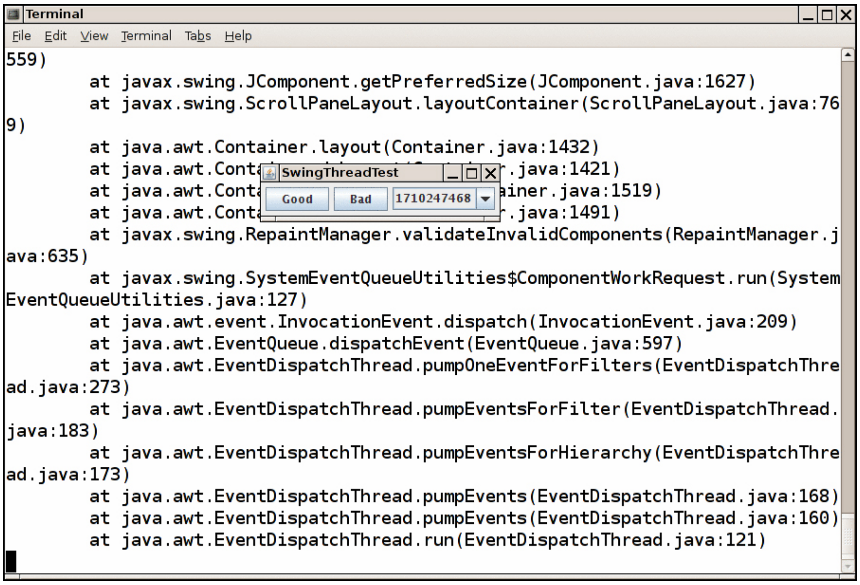
To see the problem, run the upcoming test program in Listing 14.13. When you click the Bad button, a new thread is started whose `run` method tortures a combo box, randomly adding and removing values.

```
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 scrollbar. Move the window. Click the Bad button again. Keep clicking the combo box. Eventually, you should see an exception report (Figure 14.8).

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

This situation could have been avoided if programmers could 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 importantly, the Swing team checked out the experience other teams had with thread-safe user interface toolkits. What they found was not encouraging. Programmers using thread-safe toolkits turned out to be confused by the demands for synchronization and often created deadlock-prone programs.



```
559)
        at javax.swing.JComponent.getPreferredSize(JComponent.java:1627)
        at javax.swing.ScrollPaneLayout.layoutContainer(ScrollPaneLayout.java:76
9)
        at java.awt.Container.layout(Container.java:1432)
        at java.awt.Cont                          .java:1421)
        at java.awt.Cont                          ainer.java:1519)
        at java.awt.Cont                          .java:1491)
        at javax.swing.RepaintManager.validateInvalidComponents(RepaintManager.j
ava:635)
        at javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.run(System
EventQueueUtilities.java:127)
        at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:209)
        at java.awt.EventQueue.dispatchEvent(EventQueue.java:597)
        at java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThre
ad.java:273)
        at java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.
java:183)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThre
ad.java:173)
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:168)
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:160)
        at java.awt.EventDispatchThread.run(EventDispatchThread.java:121)
```

Figure 14.8 Exception reports in the console

14.11.1 Running Time-Consuming Tasks

When you use threads together with Swing, you have to follow two simple rules.

1. If an action takes a long time, do it in a separate worker thread and never in the event dispatch thread.
2. Do not touch Swing components in any thread other than the event dispatch thread.

The reason for the first rule is easy to understand. If you take a long time in the event dispatch thread, the application seems "dead" because it cannot respond to any events. In particular, the event dispatch thread should never make input/output calls, which might block indefinitely, and it should never call `sleep`. (If you need to wait for a specific amount of time, use timer events.)

The second rule is often called the *single-thread rule* for Swing programming. We discuss it further on page 951.

These two rules seem to be in conflict with each other. Suppose you fire up a separate thread to run a time-consuming task. You would usually want to update the user interface to indicate progress while your thread is working. When your task is finished, you'd 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, you can't simply set its value from your thread.

To solve this problem, you can use, in any thread, two utility methods 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. Place the Swing code into the `run` method of a class that implements the `Runnable` interface. Then, 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(() -> {
   label.setText(percentage + "% complete");
});
```

The `invokeLater` method returns immediately when the event is posted to the event queue. The `run` method of the `Runnable` is executed asynchronously. The `invokeAndWait` method waits until the `run` method has actually been executed.

For 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.

Listing 14.13 demonstrates how to use the `invokeLater` method to safely modify the contents of a combo box. If you click the Good button, a thread inserts and removes numbers. However, the actual modification takes place in the event dispatching thread.

Listing 14.13 `swing/SwingThreadTest.java`

```
package swing;

import java.awt.*;
import java.util.*;

import javax.swing.*;

/**
```

```
 * This program demonstrates that a thread that runs in parallel with the event
 * dispatch thread can cause errors in Swing components.
 * @version 1.24 2015-06-21
 * @author Cay Horstmann
 */
public class SwingThreadTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() -> {
         JFrame frame = new SwingThreadFrame();
         frame.setTitle("SwingThreadTest");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * This frame has two buttons to fill a combo box from a separate thread. The
 * "Good" button uses the event queue, the "Bad" button modifies the combo box
 * directly.
 */
class SwingThreadFrame extends JFrame
{
   public SwingThreadFrame()
   {
      final JComboBox<Integer> combo = new JComboBox<>();
      combo.insertItemAt(Integer.MAX_VALUE, 0);
      combo.setPrototypeDisplayValue(combo.getItemAt(0));
      combo.setSelectedIndex(0);

      JPanel panel = new JPanel();

      JButton goodButton = new JButton("Good");
      goodButton.addActionListener(event ->
         new Thread(new GoodWorkerRunnable(combo)).start());
      panel.add(goodButton);
      JButton badButton = new JButton("Bad");
      badButton.addActionListener(event ->
         new Thread(new BadWorkerRunnable(combo)).start());
      panel.add(badButton);

      panel.add(combo);
      add(panel);
      pack();
   }
}

```

(Continues)

Listing 14.13 *(Continued)*

```
/**
 * This runnable modifies a combo box by randomly adding and removing numbers.
 * This can result in errors because the combo box methods are not synchronized
 * and both the worker thread and the event dispatch thread access the combo
 * box.
 */
class BadWorkerRunnable implements Runnable
{
   private JComboBox<Integer> combo;
   private Random generator;

   public BadWorkerRunnable(JComboBox<Integer> aCombo)
   {
      combo = aCombo;
      generator = new Random();
   }

   public void run()
   {
      try
      {
         while (true)
         {
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0)
               combo.insertItemAt(i, 0);
            else if (combo.getItemCount() > 0)
               combo.removeItemAt(i % combo.getItemCount());
            Thread.sleep(1);
         }
      }
      catch (InterruptedException e)
      {
      }
   }
}

/**
 * This runnable modifies a combo box by randomly adding and removing numbers.
 * In order to ensure that the combo box is not corrupted, the editing
 * operations are forwarded to the event dispatch thread.
 */
class GoodWorkerRunnable implements Runnable
{
   private JComboBox<Integer> combo;
   private Random generator;

   public GoodWorkerRunnable(JComboBox<Integer> aCombo)
   {
      combo = aCombo;
      generator = new Random();
   }

   public void run()
   {
      try
      {
         while (true)
         {
            EventQueue.invokeLater(() ->
               {
                  int i = Math.abs(generator.nextInt());
                  if (i % 2 == 0)
                     combo.insertItemAt(i, 0);
                  else if (combo.getItemCount() > 0)
                     combo.removeItemAt(i % combo.getItemCount());
               });
            Thread.sleep(1);
         }
      }
      catch (InterruptedException e)
      {
      }
   }
}
```

java.awt.EventQueue 1.1

- `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

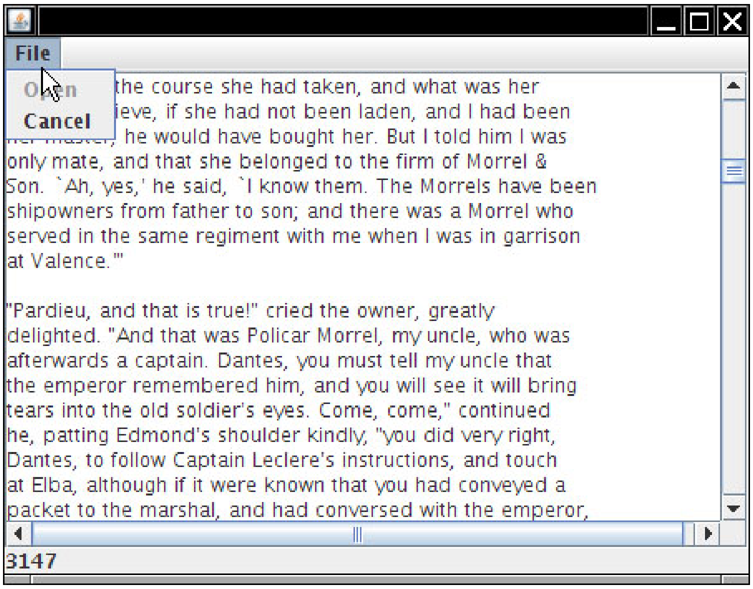
  causes the `run` method of the `runnable` object to be executed in the event dispatch thread after pending events have been processed. This call blocks until the `run` method has terminated.
- `static boolean isDispatchThread()` 1.2

  returns `true` if the thread executing this method is the event dispatch thread.

14.11.2 Using the 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. The `SwingWorker` class reduces the tedium of implementing background tasks.

The program in Listing 14.14 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 being read, the Open menu item is disabled and the Cancel item is enabled (see Figure 14.9). 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.



```
the course she had taken, and what was her
ieve, if she had not been laden, and I had been
, he would have bought her. But I told him I was
only mate, and that she belonged to the firm of Morrel &
Son. `Ah, yes,' he said, `I know them. The Morrels have been
shipowners from father to son; and there was a Morrel who
served in the same regiment with me when I was in garrison
at Valence.'"

"Pardieu, and that is true!" cried the owner, greatly
delighted. "And that was Policar Morrel, my uncle, who was
afterwards a captain. Dantes, you must tell my uncle that
the emperor remembered him, and you will see it will bring
tears into the old soldier's eyes. Come, come," continued
he, patting Edmond's shoulder kindly, "you did very right,
Dantes, to follow Captain Leclere's instructions, and touch
at Elba, although if it were known that you had conveyed a
packet to the marshal, and had conversed with the emperor,
3147
```

Figure 14.9 Loading a file in a separate thread

This example shows the typical UI activities of a background task:

- After each work unit, update the UI to show progress.
- After the work is finished, make a final change to the UI.

The `SwingWorker` class makes it easy to implement such a task. Override the `doInBackground` method to do the time-consuming work and occasionally call `publish` to communicate work progress. This method is executed in a worker thread. The `publish` method causes a `process` method to execute in the event dispatch thread to deal with the progress data. When the work is complete, the `done` method is called in the event dispatch thread so that you can finish updating the UI.

Whenever you want to do some work in the worker thread, construct a new worker. (Each worker object is meant to be used only once.) Then call the `execute` method. You will typically call `execute` on the event dispatch thread, but that is not a requirement.

It is assumed that a worker produces a result of some kind; therefore, `SwingWorker<T, V>` implements `Future<T>`. This result can be obtained by the `get` method of the `Future` interface. Since the `get` method blocks until the result is available, you don't want to call it immediately after calling `execute`. It is a good idea to call it only when you know that the work has been completed. Typically, you call `get` from the `done` method. (There is no requirement to call `get`. Sometimes, processing the progress data is all you need.)

Both the intermediate progress data and the final result can have arbitrary types. The `SwingWorker` class has these types as type parameters. A `SwingWorker<T, V>` produces a result of type `T` and progress data of type `V`.

To cancel the work in progress, use the `cancel` method of the `Future` interface. When the work is canceled, the `get` method throws a `CancellationException`.

As already mentioned, the worker thread's call to `publish` will cause calls to `process` on the event dispatch thread. For efficiency, the results of several calls to `publish` may be batched up in a single call to `process`. The `process` method receives a `List<V>` containing all intermediate results.

Let us put this mechanism to work for reading in a text file. As it turns out, a `JTextArea` is quite slow. Appending lines from a long text file (such as all lines in *The Count of Monte Cristo*) takes considerable time.

To show the user that progress is being made, we want to display the number of lines read in a status line. Thus, the progress data consist of the current line number and the current line of text. We package these into a trivial inner class:

```
private class ProgressData
{
   public int number;
   public String line;
}
```

The final result is the text that has been read into a `StringBuilder`. Thus, we need a `SwingWorker<StringBuilder, ProgressData>`.

In the `doInBackground` method, we read a file, a line at a time. After each line, we call `publish` to publish the line number and the text of the current line.

```
@Override public StringBuilder doInBackground() throws IOException, InterruptedException
{
   int lineNumber = 0;
   Scanner in = new Scanner(new FileInputStream(file), "UTF-8");
   while (in.hasNextLine())
   {
      String line = in.nextLine();
      lineNumber++;
      text.append(line).append("\n");
      ProgressData data = new ProgressData();
      data.number = lineNumber;
      data.line = line;
      publish(data);
      Thread.sleep(1); // to test cancellation; no need to do this in your programs
   }
   return text;
}
```

We also sleep for a millisecond after every line so that you can test cancellation without getting stressed out, but you wouldn't want to slow down your own programs by sleeping. If you comment out this line, you will find that *The Count of Monte Cristo* loads quite quickly, with only a few batched user interface updates.

NOTE: You can make this program behave quite smoothly by updating the text area from the worker thread, but this is not possible for most Swing components. We show you the general approach in which all component updates occur in the event dispatch thread.

In the `process` method, we ignore all line numbers but the last one, and we concatenate all lines for a single update of the text area.

```
@Override public void process(List<ProgressData> data)
{
   if (isCancelled()) return;
   StringBuilder b = new StringBuilder();
   statusLine.setText("" + data.get(data.size() - 1).number);
   for (ProgressData d : data) b.append(d.line).append("\n");
   textArea.append(b.toString());
}
```

In the `done` method, the text area is updated with the complete text, and the Cancel menu item is disabled.

Note how the worker is started in the event listener for the Open menu item.

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

Listing 14.14 `swingWorker/SwingWorkerTest.java`

```
package swingWorker;

import java.awt.*;
import java.io.*;
import java.util.*;
import java.util.List;
import java.util.concurrent.*;

import javax.swing.*;

/**
 * This program demonstrates a worker thread that runs a potentially time-consuming task.
 * @version 1.11 2015-06-21
 * @author Cay Horstmann
 */
public class SwingWorkerTest
{
   public static void main(String[] args) throws Exception
   {
      EventQueue.invokeLater(() -> {
         JFrame frame = new SwingWorkerFrame();
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * This frame has a text area to show the contents of a text file, a menu to open a file and
 * cancel the opening process, and a status line to show the file loading progress.
 */
class SwingWorkerFrame extends JFrame
{
   private JFileChooser chooser;
   private JTextArea textArea;
   private JLabel statusLine;
   private JMenuItem openItem;
   private JMenuItem cancelItem;
   private SwingWorker<StringBuilder, ProgressData> textReader;
   public static final int TEXT_ROWS = 20;
   public static final int TEXT_COLUMNS = 60;

```

(Continues)

Listing 14.14 *(Continued)*

```
   public SwingWorkerFrame()
   {
      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
      add(new JScrollPane(textArea));

      statusLine = new JLabel(" ");
      add(statusLine, BorderLayout.SOUTH);

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(event -> {
            // show file chooser dialog
            int result = chooser.showOpenDialog(null);

            // if file selected, set it as icon of the label
            if (result == JFileChooser.APPROVE_OPTION)
            {
               textArea.setText("");
               openItem.setEnabled(false);
               textReader = new TextReader(chooser.getSelectedFile());
               textReader.execute();
               cancelItem.setEnabled(true);
            }
         });

      cancelItem = new JMenuItem("Cancel");
      menu.add(cancelItem);
      cancelItem.setEnabled(false);
      cancelItem.addActionListener(event -> textReader.cancel(true));
      pack();
   }

   private class ProgressData
   {
      public int number;
      public String line;
   }

   private class TextReader extends SwingWorker<StringBuilder, ProgressData>
   {
      private File file;
      private StringBuilder text = new StringBuilder();

      public TextReader(File file)
      {
         this.file = file;
      }

      // The following method executes in the worker thread; it doesn't touch Swing components.

      @Override
      public StringBuilder doInBackground() throws IOException, InterruptedException
      {
         int lineNumber = 0;
         try (Scanner in = new Scanner(new FileInputStream(file), "UTF-8"))
         {
            while (in.hasNextLine())
            {
               String line = in.nextLine();
               lineNumber++;
               text.append(line).append("\n");
               ProgressData data = new ProgressData();
               data.number = lineNumber;
               data.line = line;
               publish(data);
               Thread.sleep(1); // to test cancellation; no need to do this in your programs
            }
         }
         return text;
      }

      // The following methods execute in the event dispatch thread.

      @Override
      public void process(List<ProgressData> data)
      {
         if (isCancelled()) return;
         StringBuilder b = new StringBuilder();
         statusLine.setText("" + data.get(data.size() - 1).number);
         for (ProgressData d : data) b.append(d.line).append("\n");
         textArea.append(b.toString());
      }

      @Override
      public void done()
      {
```

(Continues)

Listing 14.14 *(Continued)*

```
         try
         {
            StringBuilder result = get();
            textArea.setText(result.toString());
            statusLine.setText("Done");
         }
         catch (InterruptedException ex)
         {
         }
         catch (CancellationException ex)
         {
            textArea.setText("");
            statusLine.setText("Cancelled");
         }
         catch (ExecutionException ex)
         {
            statusLine.setText("" + ex.getCause());
         }

         cancelItem.setEnabled(false);
         openItem.setEnabled(true);
      }
   };
}
```

`javax.swing.SwingWorker<T, V>` 6

- `abstract T doInBackground()`

  is the method to override to carry out the background task and to return the result of the work.
- `void process(List<V> data)`

  is the method to override to process intermediate progress data in the event dispatch thread.
- `void publish(V... data)`

  forwards intermediate progress data to the event dispatch thread. Call this method from `doInBackground`.
- `void execute()`

  schedules this worker for execution on a worker thread.
- `SwingWorker.StateValue getState()`

  gets the state of this worker, one of `PENDING`, `STARTED`, or `DONE`.

14.11.3 The Single-Thread Rule

Every Java application starts with a `main` method that runs in the main thread. In a Swing program, the main thread is short lived. It schedules the construction of the user interface in the event dispatch thread and then exits. After the user interface construction, the event dispatch thread processes event notifications, such as calls to `actionPerformed` or `paintComponent`. 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.

Earlier in the chapter, we introduced the single-thread rule: "Do not touch Swing components in any thread other than the event dispatch thread." In this section, we investigate that rule further.

There are a few exceptions to the single-thread rule.

- You can safely add and remove event listeners in any thread. Of course, the listener methods will be invoked in the event dispatch thread.
- A small number of 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
JComponent.repaint
JComponent.revalidate
```

NOTE: We used the `repaint` method many times in 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`.)

Historically, the single-thread rule was more permissive. Any thread was allowed to construct components, set their properties, and add them into containers, as long as none of these components had been *realized*. A component is realized if it can receive paint or validation events. This is the case after the `setVisible(true)` or `pack` (!) methods have been invoked on the component, or after the component has been added to a container that has been realized.

That version of the single-thread rule was convenient. It allowed you to create the GUI in the `main` method and then call `setVisible(true)` on the top-level frame of the application. There was no bothersome scheduling of a `Runnable` on the event dispatch thread.

Unfortunately, some component implementors did not pay attention to the subtleties of the original single-thread rule. They launched activities on the event dispatch thread without ever bothering to check whether the component was realized. For example, if you call `setSelectionStart` or `setSelectionEnd` on a `JTextComponent`, a caret movement is scheduled in the event dispatch thread, even if the component is not visible.

It might well have been possible to detect and fix these problems, but the Swing designers took the easy way out. They decreed that it is never safe to access components from any thread other than the event dispatch thread. Therefore, you need to construct the user interface in the event dispatch thread, using the calls to `EventQueue.invokeLater` that you have seen in all our sample programs.

Of course, there are plenty of programs that are not so careful and live by the old version of the single-thread rule, initializing the user interface on the main thread. Those programs incur the slight risk that some of the user interface initialization may cause actions on the event dispatch thread that conflict with actions on the main thread. As we said in Chapter 10, you don't want to be one of the unlucky few who run into trouble and waste time debugging an intermittent threading bug. Therefore, you should simply follow the strict single-thread rule.

You have now reached the end of Volume I of *Core Java*. This volume covered the fundamentals of the Java programming language and the parts of the standard library that you need for most programming projects. We hope that you enjoyed your tour through the Java fundamentals and that you found useful information along the way. For advanced topics, such as networking, advanced AWT/Swing, security, and internationalization, please turn to Volume II.