Objectives

- To get an overview of multithreading (§32.2).
- To develop task classes by implementing the `Runnable` interface (§32.3).
- To create threads to run tasks using the `Thread` class (§32.3).
- To control threads using the methods in the `Thread` class (§32.4).
- To control animations using threads (§32.5, §32.7).
- To run code in the event dispatch thread (§32.6).
- To execute tasks in a thread pool (§32.8).
- To use synchronized methods or blocks to synchronize threads to avoid race conditions (§32.9).
- To synchronize threads using locks (§32.10).
- To facilitate thread communications using conditions on locks (§§32.11–32.12).
- To use blocking queues to synchronize access to an array queue, linked queue, and priority queue (§32.13).
- To restrict the number of accesses to a shared resource using semaphores (§32.14).
- To use the resource-ordering technique to avoid deadlocks (§32.15).
- To describe the life cycle of a thread (§32.16).
- To create synchronized collections using the static methods in the `Collections` class (§32.17).
- To develop parallel programs using the Fork/Join Framework (§32.18).
Chapter 32 Multithreading and Parallel Programming

32.1 Introduction

Multithreading enables multiple tasks in a program to be executed concurrently.

One of the powerful features of Java is its built-in support for multithreading—the concurrent running of multiple tasks within a program. In many programming languages, you have to invoke system-dependent procedures and functions to implement multithreading. This chapter introduces the concepts of threads and how to develop multithreading programs in Java.

32.2 Thread Concepts

A program may consist of many tasks that can run concurrently. A thread is the flow of execution, from beginning to end, of a task.

A thread provides the mechanism for running a task. With Java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multiprocessor systems, as shown in Figure 32.1a.

![Figure 32.1](a) Here multiple threads are running on multiple CPUs. (b) Here multiple threads share a single CPU.

In single-processor systems, as shown in Figure 32.1b, the multiple threads share CPU time, known as time sharing, and the operating system is responsible for scheduling and allocating resources to them. This arrangement is practical, because most of the time the CPU is idle. It does nothing, for example, while waiting for the user to enter data.

Multithreading can make your program more responsive and interactive, as well as enhance performance. For example, a good word processor lets you print or save a file while you are typing. In some cases, multithreaded programs run faster than single-threaded programs even on single-processor systems. Java provides exceptionally good support for creating and running threads and for locking resources to prevent conflicts.

When your program executes as an application, the Java interpreter starts a thread for the main method. When your program executes as an applet, the Web browser starts a thread to run the applet. You can create additional threads to run concurrent tasks in the program. In Java, each task is an instance of the Runnable interface, also called a runnable object. A thread is essentially an object that facilitates the execution of a task.

32.1 Why is multithreading needed? How can multiple threads run simultaneously in a single-processor system?

32.2 What is a runnable object? What is a thread?

32.3 Creating Tasks and Threads

A task class must implement the Runnable interface. A task must be run from a thread.

Tasks are objects. To create tasks, you have to first define a class for tasks, which implements the Runnable interface. The Runnable interface is rather simple. All it contains is
32.3 Creating Tasks and Threads

the `run` method. You need to implement this method to tell the system how your thread is going to run. A template for developing a task class is shown in Figure 32.2a.

![Figure 32.2](image)

**Figure 32.2** Define a task class by implementing the `Runnable` interface.

Once you have defined a `TaskClass`, you can create a task using its constructor. For example,

```java
TaskClass task = new TaskClass(...);
```

A task must be executed in a thread. The `Thread` class contains the constructors for creating threads and many useful methods for controlling threads. To create a thread for a task, use

```java
Thread thread = new Thread(task);
```

You can then invoke the `start()` method to tell the JVM that the thread is ready to run, as follows:

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

The JVM will execute the task by invoking the task’s `run()` method. Figure 32.2b outlines the major steps for creating a task, a thread, and starting the thread.

Listing 32.1 gives a program that creates three tasks and three threads to run them.

- The first task prints the letter `a` 100 times.
- The second task prints the letter `b` 100 times.
- The third task prints the integers 1 through 100.

When you run this program, the three threads will share the CPU and take turns printing letters and numbers on the console. Figure 32.3 shows a sample run of the program.

### Listing 32.1 TaskThreadDemo.java

```java
public class TaskThreadDemo {
    public static void main(String[] args) {
        // Create tasks
        Runnable printA = new PrintChar('a', 100);
        Runnable printB = new PrintChar('b', 100);
        Runnable print100 = new PrintNum(100);
```
Figure 32.3  Tasks printA, printB, and print100 are executed simultaneously to display the letter a 100 times, the letter b 100 times, and the numbers from 1 to 100.

```java
// Create threads
Thread thread1 = new Thread(printA);
Thread thread2 = new Thread(printB);
Thread thread3 = new Thread(print100);

// Start threads
thread1.start();
thread2.start();
thread3.start();
```

class PrintChar implements Runnable {
  private char charToPrint;  // The character to print
  private int times;  // The number of times to repeat

  /** Construct a task with specified character and number of times to print the character */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  @Override  /** Override the run() method to tell the system what task to perform */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ..., n */
  public PrintNum(int n) {
    lastNum = n;
  }

  @Override  /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
```
32.3 Creating Tasks and Threads

The program creates three tasks (lines 4–6). To run them concurrently, three threads are created (lines 9–11). The `start()` method (lines 14–16) is invoked to start a thread that causes the `run()` method in the task to be executed. When the `run()` method completes, the thread terminates.

Because the first two tasks, `printA` and `printB`, have similar functionality, they can be defined in one task class `PrintChar` (lines 21–41). The `PrintChar` class implements `Runnable` and overrides the `run()` method (lines 36–40) with the print-character action. This class provides a framework for printing any single character a given number of times. The runnable objects `printA` and `printB` are instances of the `PrintChar` class.

The `PrintNum` class (lines 44–58) implements `Runnable` and overrides the `run()` method (lines 53–57) with the print-number action. This class provides a framework for printing numbers from 1 to n, for any integer n. The runnable object `print100` is an instance of the class `PrintNum` class.

**Note**
If you don't see the effect of these three threads running concurrently, increase the number of characters to be printed. For example, change line 4 to:

```
Runnable printA = new PrintChar('a', 10000);
```

**Important Note**
The `run()` method in a task specifies how to perform the task. This method is automatically invoked by the JVM. You should not invoke it. Invoking `run()` directly merely executes this method in the same thread; no new thread is started.

32.3 How do you define a task class? How do you create a thread for a task?

32.4 What would happen if you replaced the `start()` method with the `run()` method in lines 14–16 in Listing 32.1?

```
print100.start();
printA.start();
printB.start();
```

Replaced by

```
print100.run();
printA.run();
printB.run();
```

32.5 What is wrong in the following two programs? Correct the errors.

```
public class Test implements Runnable {
    public static void main(String[] args) {
        new Test();
    }

    public Test() {
        Test task = new Test();
        new Thread(task).start();
    }

    public void run() {
        System.out.println("test");
    }
}

(a)
```

```
public class Test implements Runnable {
    public static void main(String[] args) {
        new Test();
    }

    public Test() {
        Thread t = new Thread(this);
        t.start();
        t.start();
    }

    public void run() {
        System.out.println("test");
    }
}

(b)
```
32.4 The Thread Class

The Thread class contains the constructors for creating threads for tasks and the methods for controlling threads.

Figure 32.4 shows the class diagram for the Thread class.

```
interface Runnable

class Thread
+Thread()
+Thread(task: Runnable)
+start(): void
+isAlive(): boolean
+setPriority(p: int): void
+join(): void
+yield(): void
+interrupt(): void
```

Creates an empty thread.
Creates a thread for a specified task.
Starts the thread that causes the run() method to be invoked by the JVM.
Tests whether the thread is currently running.
Sets priority p (ranging from 1 to 10) for this thread.
Waits for this thread to finish.
Puts a thread to sleep for a specified time in milliseconds.
Causes a thread to pause temporarily and allow other threads to execute.
Interrupts this thread.

Note
Since the Thread class implements Runnable, you could define a class that extends Thread and implements the run method, as shown in Figure 32.5a, and then create an object from the class and invoke its start method in a client program to start the thread, as shown in Figure 32.5b.

```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }
  
  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }

  // Create a thread
  CustomThread thread1 = new CustomThread(...);
  // Start a thread
  thread1.start();

  // Create another thread
  CustomThread thread2 = new CustomThread(...);
  // Start a thread
  thread2.start();
}
```

(a)

(b)

Figure 32.5 Define a thread class by extending the Thread class.
This approach is, however, not recommended, because it mixes the task and the mechanism of running the task. Separating the task from the thread is a preferred design.

**Note**
The Thread class also contains the `stop()`, `suspend()`, and `resume()` methods. As of Java 2, these methods were deprecated (or outdated) because they are known to be inherently unsafe. Instead of using the `stop()` method, you should assign `null` to a Thread variable to indicate that it is stopped.

You can use the `yield()` method to temporarily release time for other threads. For example, suppose you modify the code in the `run()` method in lines 53–57 for PrintNum in Listing 32.1 as follows:

```java
public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" "+ i);
        Thread.yield();
    }
}
```

Every time a number is printed, the thread of the print100 task is yielded to other threads.

The `sleep(long millis)` method puts the thread to sleep for the specified time in milliseconds to allow other threads to execute. For example, suppose you modify the code in lines 53–57 in Listing 32.1 as follows:

```java
public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" "+ i);
        if (i >= 50) ;
    }
}
```

Every time a number (>= 50) is printed, the thread of the print100 task is put to sleep for 1 millisecond.

The `sleep` method may throw an `InterruptedException`, which is a checked exception. Such an exception may occur when a sleeping thread’s `interrupt()` method is called. The `interrupt()` method is very rarely invoked on a thread, so an `InterruptedException` is unlikely to occur. But since Java forces you to catch checked exceptions, you have to put it in a `try-catch` block. If a `sleep` method is invoked in a loop, you should wrap the loop in a `try-catch` block, as shown in (a) below. If the loop is outside the `try-catch` block, as shown in (b), the thread may continue to execute even though it is being interrupted.

```java
try {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" "+ i);
        if (i >= 50) Thread.sleep(1);
    }
} catch (InterruptedException ex) {
    ex.printStackTrace();
}
```

```java
public void run() {
    try {
        while (...) {
            Thread.sleep(1000);
        }
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

(a) Correct

```
public void run() {
    try {
        while (...) {
            Thread.sleep(sleepTime);
        }
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

(b) Incorrect
You can use the `join()` method to force one thread to wait for another thread to finish. For example, suppose you modify the code in lines 53–57 in Listing 32.1 as follows:

```java
public void run() {
    Thread thread4 = new Thread(
        new PrintChar('c', 40));
    thread4.start();
    try {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print (" "+i);
            if (i == 50) thread4.join();
        }
    } catch (InterruptedException ex) {
    }
}
```

A new `thread4` is created, and it prints character c 40 times. The numbers from 50 to 100 are printed after thread `thread4` is finished.

Java assigns every thread a priority. By default, a thread inherits the priority of the thread that spawned it. You can increase or decrease the priority of any thread by using the `setPriority` method, and you can get the thread’s priority by using the `getPriority` method. Priorities are numbers ranging from 1 to 10. The `Thread` class has the `int` constants `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`, representing 1, 5, and 10, respectively. The priority of the main thread is `Thread.NORM_PRIORITY`.

The JVM always picks the currently runnable thread with the highest priority. A lower-priority thread can run only when no higher-priority threads are running. If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue. This is called `round-robin scheduling`. For example, suppose you insert the following code in line 16 in Listing 32.1:

```
thread3.setPriority(Thread.MAX_PRIORITY);
```

The thread for the `print100` task will be finished first.

**Tip**

The priority numbers may be changed in a future version of Java. To minimize the impact of any changes, use the constants in the `Thread` class to specify thread priorities.

**Tip**

A thread may never get a chance to run if there is always a higher-priority thread running or a same-priority thread that never yields. This situation is known as `contention` or `starvation`. To avoid contention, the thread with higher priority must periodically invoke the `sleep` or `yield` method to give a thread with a lower or the same priority a chance to run.

32.6 Which of the following methods are instance methods in `java.lang.Thread`? Which method may throw an `InterruptedException`? Which of them are deprecated in Java?

- `run`, `start`, `stop`, `suspend`, `resume`, `sleep`, `interrupt`, `yield`, `join`

32.7 If a loop contains a method that throws an `InterruptedException`, why should the loop be placed inside a `try-catch` block?

32.8 How do you set a priority for a thread? What is the default priority?
32.5 Case Study: Flashing Text

You can use a thread to control an animation.

The use of a Timer object to control animations was introduced in Section 16.11, Animation Using the Timer Class. You can also use a thread to control animation. Listing 32.2 gives an example that displays flashing text on a label, as shown in Figure 32.6.

![Figure 32.6](image)

The text “Welcome” blinks.

Listing 32.2 FlasingText.java

```java
import javax.swing.*;

public class FlasingText extends JApplet implements Runnable {
    private JLabel jlblText = new JLabel("Welcome", JLabel.CENTER);
    private Thread thread;

    public FlasingText() {
        add(jlblText);
        new Thread(this).start();
    }

    @Override
    public void run() {
        try {
            while (run()) {
                if (jlblText.getText() == null)
                    jlblText.setText("Welcome");
                else
                    jlblText.setText(null);

                Thread.sleep(200);
            }

            catch (InterruptedException ex) {
            }
        }
    }
}
```

FlasingText implements Runnable (line 3), so it is a task class. Line 8 wraps the task in a thread and starts the thread. The run method dictates how to run the thread. It sets a text in the label if the label does not have one (line 15), and sets its text as null (line 18) if the label has a text. The text is set and unset to simulate a flashing effect.

You can use a timer or a thread to control animation. Which one is better? A timer is a source component that fires an(ActionEvent at a “fixed rate.” When an action event occurs, the timer invokes the listener’s actionPerformed method to handle the event. The timer and event handling run on the same thread. If it takes a long time to handle the event, the actual delay time between the two events will be longer than the requested delay time. In this case, you should run event handling on a separate thread. (The next section gives an example to illustrate the problem and how to fix it by running the event handling on a separate thread.) In general, threads are more reliable and responsive than timers. If you need a precise delay...
time or a quick response, it is better to use a thread. Otherwise, using a timer is simpler and more efficient. Timers consume less system resources because they run on the GUI event dispatch thread, so you don’t need to spawn new threads for timers.

32.9 What causes the text to flash?

32.10 Is an instance of FlashingText a runnable object?

32.6 GUI Event Dispatch Thread

GUI event handling code is executed on a special thread called the event dispatch thread.

This special thread is also used to run most of Swing methods. Running GUI event handling code and the most of Swing methods in the same thread is necessary because most Swing methods are not thread-safe. Invoking them from multiple threads may cause conflicts.

In certain situations, you need to run the code in the event dispatch thread to avoid possible conflicts. You can use the static methods invokeLater and invokeAndWait in the javax.swing.SwingUtilities class to run the code in the event dispatch thread. You must put this code in the run method of a Runnable object and specify the Runnable object as the argument to invokeLater and invokeAndWait. The invokeLater method returns immediately, without waiting for the event dispatch thread to execute the code. The invokeAndWait method is just like invokeLater, except that invokeAndWait doesn’t return until the event dispatching thread has executed the specified code.

So far, you have launched your GUI application from the main method by creating a frame and making it visible. This works fine for most applications, but if it takes a long time to launch a GUI application, problems may occur. To avoid possible problems in this situation, you should launch the GUI creation from the event dispatch thread, as follows:

```java
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            // The code for creating a frame and setting its properties
        }
    });
}
```

For example, Listing 32.3 gives a simple program that launches the frame from the event dispatch thread.

**LISTING 32.3 EventDispatcherThreadDemo.java**

```java
import javax.swing.*;

public class EventDispatcherThreadDemo extends JApplet {
    public EventDispatcherThreadDemo() {
        add(new JLabel("Hi, it runs from an event dispatch thread"));
    }

    /** Main method */
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                // Center the frame
                frame.setLocationRelativeTo(null);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setSize(200, 200);
                frame.add(new EventDispatcherThreadDemo());
                frame.add(new EventDispatcherThreadDemo());
                frame.setSize(200, 200);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setLocationRelativeTo(null); // Center the frame
            }
        });
    }
}
```
What is the event dispatch thread?

How do you let a task run from the event dispatch thread?

This case study shows the necessity of using threads for certain GUI animations.

This case study creates an applet that displays a running clock that announces the time at one-minute intervals. For example, if the current time is 6:30:00, the applet announces, “six o’clock thirty minutes A.M.” If the current time is 20:20:00, the applet announces, “eight o’clock twenty minutes P.M.” The program also has a label that displays the digital time, as shown in Figure 32.7.

![Figure 32.7](image)

The applet displays a clock and announces the time every minute.

To announce the time, the applet plays three audio clips. The first clip announces the hour, the second announces the minute, and the third announces A.M. or P.M. All of the audio files are stored in the directory audio, a subdirectory of the applet’s class directory. The 12 audio files used to announce the hours are stored in the files hour0.au, hour1.au, and so on, to hour11.au. The 60 audio files used to announce the minutes are stored in the files minute0.au, minute1.au, and so on, to minute59.au. The two audio files used to announce A.M. or P.M. are stored in the file am.au and pm.au.

You need to play three audio clips on a separate thread to avoid animation delays. To illustrate the problem, let us first write a program without playing the audio on a separate thread.

In Section 13.9, the StillClock class was developed to draw a still clock to show the current time. Create an applet named ClockWithAudio (Listing 32.4) that contains an instance of StillClock to display an analog clock, and an instance of JLabel to display the digital time. Override the init method to load the audio files. Use a Timer object to set and display the current time continuously at every second. When the second is zero, announce the current time.

### Listing 32.4  ClockWithAudio.java

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

public class ClockWithAudio extends JApplet {
    protected AudioClip[] hourAudio = new AudioClip[12];

```

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

```java
32.11 What is the event dispatch thread?
32.12 How do you let a task run from the event dispatch thread?

### Key Point

My Programming Lab

### Check Point

audio clips
audio files
```java
protected AudioClip[] minuteAudio = new AudioClip[60];

// Create audio clips for pronouncing am and pm
protected AudioClip amAudio = Applet.newAudioClip(this.getClass().getResource("audio/am.au"));
protected AudioClip pmAudio = Applet.newAudioClip(this.getClass().getResource("audio/pm.au"));

// Create a clock
private StillClock clock = new StillClock();

// Create a timer
private Timer timer = new Timer(1000, new TimerListener());

// Create a label to display time
private JLabel jlblDigitTime = new JLabel("", JLabel.CENTER);

@Override /** Initialize the applet */
public void init() {
    // Create audio clips for pronouncing hours
    for (int i = 0; i < 12; i++)
        hourAudio[i] = Applet.newAudioClip(this.getClass().getResource("audio/hour" + i + ".au"));

    // Create audio clips for pronouncing minutes
    for (int i = 0; i < 60; i++)
        minuteAudio[i] = Applet.newAudioClip(this.getClass().getResource("audio/minute" + i + ".au"));

    // Add clock and time label to the content pane of the applet
    add(clock, BorderLayout.CENTER);
    add(jlblDigitTime, BorderLayout.SOUTH);
}

@Override /** Override the applet's start method */
public void start() {
    // Resume clock
    timer.start(); // Resume clock
}

@Override /** Override the applet's stop method */
public void stop() {
    // Suspend clock
    timer.stop(); // Suspend clock
}

private class TimerListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        clock.setCurrentTime();
        clock.repaint();
        jlblDigitTime.setText(clock.getHour() + ":" + clock.getMinute() + ":" + clock.getSecond());
        if (clock.getSecond() == 0)
            announceTime(clock.getHour(), clock.getMinute());
    }
}

/** Announce the current time at every minute */
public void announceTime(int hour, int minute) {
    // Announce hour
    hourAudio[hour % 12].play();
```
32.7 Case Study: Clock with Audio

```java
try {
    // Time delay to allow hourAudio play to finish
    Thread.sleep(1500);

    // Announce minute
    minuteAudio[minute].play();

    // Time delay to allow minuteAudio play to finish
    Thread.sleep(1500);
} catch (InterruptedException ex) {
}
```

The `hourAudio` is an array of twelve audio clips that are used to announce the 12 hours of the day (line 7); the `minuteAudio` is an audio clip that is used to announce the minutes in an hour (line 8). The `amAudio` announces “A.M.” (line 11); the `pmAudio` announces “P.M.” (line 13).

The `init()` method creates hour audio clips (lines 29–30) and minute audio clips (lines 34–35), and places a clock and a label in the applet (lines 38–39).

An `ActionEvent` is fired by the timer every second. In the listener’s `actionPerformed` method (lines 54–61), the clock is repainted with the new current time, and the digital time is displayed in the label.

In the `announceTime` method (lines 65–87), the `sleep()` method (lines 71, 77) is purposely invoked to ensure that one clip finishes before the next clip starts, so that the clips do not interfere with each other.

The applet’s `start()` and `stop()` methods (lines 43–50) are overridden to ensure that the timer starts or stops when the applet is restarted or stopped.

When you run the preceding program, you will notice that the second hand does not display at the first, second, and third seconds of the minute. This is because `sleep(1500)` is invoked twice in the `announceTime()` method, which takes three seconds to announce the time at the beginning of each minute. Thus, the next action event is delayed for three seconds during the first three seconds of each minute. As a result of this delay, the time is not updated and the clock is not repainted for these three seconds. To fix this problem, you should announce the time on a separate thread. This can be accomplished by modifying the `announceTime` method. Listing 32.5 gives the new program.

**LISTING 32.5 ClockWithAudioOnSeparateThread.java**

```java
public class ClockWithAudioOnSeparateThread extends JApplet {

    // ** Announce the current time at every minute */
    public void announceTime(int h, int m) {
        new Thread(new AnnounceTimeOnSeparateThread(h, m)).start();
    }
}
```
The new class `ClockWithAudioOnSeparateThread` is the same as `ClockWithAudio` except that the `announceTime` method is new. The new `announceTime` method creates a thread (line 8) for the task of announcing the time. The task class is defined as an inner class (lines 12–44). The `run` method (line 21) announces the time on a separate thread.

When running this program, you will discover that the audio does not interfere with the clock animation because an instance of `AnnounceTimeOnSeparateThread` starts on a separate thread to announce the current time. This thread is independent of the thread on which the `actionPerformed` method runs.

### 32.13 When should you use a timer or a thread to control animation? What are the advantages and disadvantages of using a thread and a timer?

### 32.8 Thread Pools

A thread pool can be used to execute tasks efficiently.

In Section 32.3, Creating Tasks and Threads, you learned how to define a task class by implementing `java.lang.Runnable`, and how to create a thread to run a task like this:

```java
Runnable task = new TaskClass(task);
new Thread(task).start();
```
This approach is convenient for a single task execution, but it is not efficient for a large number of tasks, because you have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance. Using a thread pool is an ideal way to manage the number of tasks executing concurrently. Java provides the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks. ExecutorService is a subinterface of Executor, as shown in Figure 32.8.

To create an Executor object, use the static methods in the Executors class, as shown in Figure 32.9. The newFixedThreadPool(int) method creates a fixed number of threads in a pool. If a thread completes executing a task, it can be reused to execute another task. If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution. The newCachedThreadPool() method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution. A thread in a cached pool will be terminated if it has not been used for 60 seconds. A cached pool is efficient for many short tasks.

Listing 32.6 shows how to rewrite Listing 32.1 using a thread pool.

Listing 32.6 ExecutorDemo.java

```
1 import java.util.concurrent.*;
2
3 public class ExecutorDemo {
```
public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);
    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));
    // Shut down the executor
    executor.shutdown();
}

Line 6 creates a thread pool executor with a total of three threads maximum. Classes PrintChar
and PrintNum were defined in Listing 32.1. Line 9 creates a task, new PrintChar('a', 100), and adds it to the pool. Similarly, another two runnable tasks are
created and added to the same pool in lines 10–11. The executor creates three threads to
eexecute three tasks concurrently.

Suppose that you replace line 6 with

    ExecutorService executor = Executors.newFixedThreadPool(1);

What will happen? The three runnable tasks will be executed sequentially, because there is
only one thread in the pool.

Suppose you replace line 6 with

    ExecutorService executor = Executors.newCachedThreadPool();

What will happen? New threads will be created for each waiting task, so all the tasks will be
executed concurrently.

The shutdown() method in line 14 tells the executor to shut down. No new tasks can be
accepted, but any existing tasks will continue to finish.

Tip
If you need to create a thread for just one task, use the Thread class. If you need to cre-
ate threads for multiple tasks, it is better to use a thread pool.

What are the benefits of using a thread pool?

How do you create a thread pool with three fixed threads? How do you submit a task
to a thread pool? How do you know that all the tasks are finished?

32.9 Thread Synchronization

Thread synchronization is to coordinate the execution of the dependent threads.

A shared resource may become corrupted if it is accessed simultaneously by multiple threads.
The following example demonstrates the problem.

Suppose that you create and launch 100 threads, each of which adds a penny to an account.
Define a class named Account to model the account, a class named AddAPennyTask to add
a penny to the account, and a main class that creates and launches threads. The relationships
of these classes are shown in Figure 32.10. The program is given in Listing 32.7.
import java.util.concurrent.*;

class AccountWithoutSync {
    private static Account account = new Account();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        // Create and launch 100 threads
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPennyTask());
        }

        executor.shutdown();

        // Wait until all tasks are finished
        while (!executor.isTerminated()) {
        }

        System.out.println("What is balance? "+account.getBalance());
    }

    // A thread for adding a penny to the account
    private static class AddAPennyTask implements Runnable {
        public void run() {
            account.deposit(1);
        }
    }

    // An inner class for account
    private static class Account {
        private int balance = 0;

        public int getBalance() {
            return balance;
        }

        public void deposit(int amount) {
            int newBalance = balance + amount;
        }
    }
}
The classes AddAPennyTask and Account in lines 24–51 are inner classes. Line 4 creates an Account with initial balance 0. Line 11 creates a task to add a penny to the account and submit the task to the executor. Line 11 is repeated 100 times in lines 10–12. The program repeatedly checks whether all tasks are completed in lines 17–18. The account balance is displayed in line 20 after all tasks are completed.

The program creates 100 threads executed in a thread pool executor (lines 10–12). The isTerminated() method (line 17) is used to test whether the thread is terminated.

The balance of the account is initially 0 (line 32). When all the threads are finished, the balance should be 100, but the output is unpredictable. As can be seen in Figure 32.11, the answers are wrong in the sample run. This demonstrates the data-corruption problem that occurs when all the threads have access to the same data source simultaneously.

Lines 39–49 could be replaced by one statement:

\[
\text{balance} = \text{balance} + \text{amount};
\]

It is highly unlikely, although plausible, that the problem can be replicated using this single statement. The statements in lines 39–49 are deliberately designed to magnify the data-corruption problem and make it easy to see. If you run the program several times but still do not see the problem, increase the sleep time in line 44. This will increase the chances for showing the problem of data inconsistency.

What, then, caused the error in this program? A possible scenario is shown in Figure 32.12.
In Step 1, Task 1 gets the balance from the account. In Step 2, Task 2 gets the same balance from the account. In Step 3, Task 1 writes a new balance to the account. In Step 4, Task 2 writes a new balance to the account.

The effect of this scenario is that Task 1 does nothing, because in Step 4 Task 2 overrides Task 1’s result. Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes a conflict. This is a common problem, known as a race condition, in multithreaded programs. A class is said to be thread-safe if an object of the class does not cause a race condition in the presence of multiple threads. As demonstrated in the preceding example, the Account class is not thread-safe.

32.9.1 The synchronized Keyword

To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the critical region. The critical region in Listing 32.7 is the entire deposit method. You can use the keyword synchronized to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in Listing 32.7. One approach is to make Account thread-safe by adding the keyword synchronized in the deposit method in line 38, as follows:

```java
public synchronized void deposit(double amount)
```

A synchronized method acquires a lock before it executes. A lock is a mechanism for exclusive use of a resource. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

With the deposit method synchronized, the preceding scenario cannot happen. If Task 1 enters the method, Task 2 is blocked until Task 1 finishes the method, as shown in Figure 32.13.

![Figure 32.13](image)

Task 1 and Task 2 are synchronized.

32.9.2 Synchronizing Statements

Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class. A synchronized statement can be used to acquire a lock on any object, not just this object, when executing a
synchronized block

block of the code in a method. This block is referred to as a synchronized block. The general form of a synchronized statement is as follows:

```java
synchronized (expr) {
    statements;
}
```

The expression `expr` must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronized statements enable you to synchronize part of the code in a method instead of the entire method. This increases concurrency. You can make Listing 32.7 thread-safe by placing the statement in line 26 inside a synchronized block:

```java
synchronized (account) {
    account.deposit(1);
}
```

Note

Any synchronized instance method can be converted into a synchronized statement. For example, the following synchronized instance method in (a) is equivalent to (b):

```java
(a) public synchronized void xMethod() {
    // method body
}
```

```java
(b) public void xMethod() {
    synchronized (this) {
        // method body
    }
}
```

32.16 Give some examples of possible resource corruption when running multiple threads. How do you synchronize conflicting threads?

32.17 Suppose you place the statement in line 26 of Listing 32.7 inside a synchronized block to avoid race conditions, as follows:

```java
synchronized (this) {
    account.deposit(1);
}
```

Will it work?

32.10 Synchronization Using Locks

*Locks and conditions can be explicitly used to synchronize threads.*

Recall that in Listing 32.7, 100 tasks deposit a penny to the same account concurrently, which causes conflicts. To avoid it, you used the `synchronized` keyword in the `deposit` method, as follows:

```java
public synchronized void deposit(double amount)
```

A synchronized instance method implicitly acquires a lock on the instance before it executes the method.

Java enables you to acquire locks explicitly, which give you more control for coordinating threads. A lock is an instance of the `Lock` interface, which defines the methods for
acquiring and releasing locks, as shown in Figure 32.14. A lock may also use the `newCondition()` method to create any number of `Condition` objects, which can be used for thread communications.

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

public class AccountWithSyncUsingLock {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPennyTask());
        }
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("What is balance? " + account.getBalance());
        }
    }
}
```

**Figure 32.14** The `ReentrantLock` class implements the `Lock` interface to represent a lock.

`ReentrantLock` is a concrete implementation of `Lock` for creating mutually exclusive locks. You can create a lock with the specified fairness policy. True fairness policies guarantee that the longest-waiting thread will obtain the lock first. False fairness policies grant a lock to a waiting thread arbitrarily. Programs using fair locks accessed by many threads may have poorer overall performance than those using the default setting, but they have smaller variances in times to obtain locks and prevent starvation.

Listing 32.8 revises the program in Listing 32.7 to synchronize the account modification using explicit locks.
public static class AddAPennyTask implements Runnable {
    public void run() {
        account.deposit(1);
    }
}

// An inner class for Account
public static class Account {
    private int balance = 0;
    public int getBalance() {
        return balance;
    }
    public void deposit(int amount) {
        try {
            int newBalance = balance + amount;
            // This delay is deliberately added to magnify the
            // data-corruption problem and make it easy to see.
            Thread.sleep(5);
            balance = newBalance;
        } catch (InterruptedException ex) {
        } finally {
            lock.unlock(); // Release the lock
        }
    }
}

Line 33 creates a lock, line 41 acquires the lock, and line 55 releases the lock.

Tip
It is a good practice to always immediately follow a call to lock() with a try-catch block and release the lock in the finally clause, as shown in lines 41–56, to ensure that the lock is always released.

Listing 32.8 can be implemented using a synchronize method for deposit rather than using a lock. In general, using synchronized methods or statements is simpler than using explicit locks for mutual exclusion. However, using explicit locks is more intuitive and flexible to synchronize threads with conditions, as you will see in the next section.

32.18 How do you create a lock object? How do you acquire a lock and release a lock?

32.11 Cooperation among Threads
Conditions on locks can be used to coordinate thread interactions.

Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region, but sometimes you also need a way for threads to cooperate. Conditions can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the
newCondition() method on a Lock object. Once a condition is created, you can use its `await()`, `signal()`, and `signalAll()` methods for thread communications, as shown in Figure 32.15. The `await()` method causes the current thread to wait until the condition is signaled. The `signal()` method wakes up one waiting thread, and the `signalAll()` method wakes all waiting threads.

**Figure 32.15** The Condition interface defines the methods for performing synchronization.

Let us use an example to demonstrate thread communications. Suppose that you create and launch two tasks: one that deposits into an account, and one that withdraws from the same account. The withdraw task has to wait if the amount to be withdrawn is more than the current balance. Whenever new funds are deposited into the account, the deposit task notifies the withdraw thread to resume. If the amount is still not enough for a withdrawal, the withdraw thread has to continue to wait for a new deposit.

To synchronize the operations, use a lock with a condition: `newDeposit` (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the `newDeposit` condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown in Figure 32.16.

**Figure 32.16** The condition `newDeposit` is used for communications between the two threads.

You create a condition from a Lock object. To use a condition, you have to first obtain a lock. The `await()` method causes the thread to wait and automatically releases the lock on the condition. Once the condition is right, the thread reacquires the lock and continues executing.

Assume that the initial balance is 0 and the amounts to deposit and withdraw are randomly generated. Listing 32.9 gives the program. A sample run of the program is shown in Figure 32.17.
Figure 32.17  The withdraw task waits if there are not sufficient funds to withdraw.

Listing 32.9  ThreadCooperation.java

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

public class ThreadCooperation {
    private static Account account = new Account();

    public static void main(String[] args) {
        // Create a thread pool with two threads
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new DepositTask());
        executor.execute(new WithdrawTask());
        executor.shutdown();

        System.out.println("Thread 1	Thread 2	Balance");
    }

    public static class DepositTask implements Runnable {
        @Override // Keep adding an amount to the account
        public void run() {
            try { // Purposely delay it to let the withdraw method proceed
                while (true) {
                    account.deposit((int)(Math.random() * 10) + 1);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

    public static class WithdrawTask implements Runnable {
        @Override // Keep subtracting an amount from the account
        public void run() {
            while (true) {
                account.withdraw((int)(Math.random() * 10) + 1);
            }
        }
    }

    // An inner class for account
    private static class Account {
        // Create a new lock
        private static Lock lock = new ReentrantLock();

        // Create a condition
        private static Condition newDeposit = lock.newCondition();
    }
```
32.11 Cooperation among Threads

```java
private int balance = 0;

public int getBalance() {
    return balance;
}

public void withdraw(int amount) {
    lock.lock(); // Acquire the lock
    try {
        while (balance < amount) {
            System.out.println("\t\t\tWait for a deposit");
            newDeposit.await();
        }
        balance -= amount;
        System.out.println("\t\t\tWithdraw " + amount + "\t\t" + getBalance());
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    } finally {
        lock.unlock(); // Release the lock
    }
}

public void deposit(int amount) {
    lock.lock(); // Acquire the lock
    try {
        balance += amount;
        System.out.println("Deposit " + amount + "\t\t\t\t" + getBalance());
        // Signal thread waiting on the condition
        newDeposit.signalAll();
    } finally {
        lock.unlock(); // Release the lock
    }
}
```

The example creates a new inner class named `Account` to model the account with two methods, `deposit(int)` and `withdraw(int)`, a class named `DepositTask` to add an amount to the balance, a class named `WithdrawTask` to withdraw an amount from the balance, and a main class that creates and launches two threads.

The program creates and submits the deposit task (line 10) and the withdraw task (line 11). The deposit task is purposely put to sleep (line 23) to let the withdraw task run. When there are not enough funds to withdraw, the withdraw task waits (line 59) for notification of the balance change from the deposit task (line 83).

A lock is created in line 44. A condition named `newDeposit` on the lock is created in line 47. A condition is bound to a lock. Before waiting or signaling the condition, a thread must first acquire the lock for the condition. The withdraw task acquires the lock in line 56, waits for the `newDeposit` condition (line 60) when there is not a sufficient amount to withdraw, and releases the lock in line 71. The deposit task acquires the lock in line 76, and signals all waiting threads (line 83) for the `newDeposit` condition after a new deposit is made.
What will happen if you replace the `while` loop in lines 58–61 with the following `if` statement?

```java
if (balance < amount) {
    System.out.println("\t\tWait for a deposit");
    newDeposit.await();
}
```

The deposit task will notify the withdraw task whenever the balance changes. `(balance < amount)` may still be true when the withdraw task is awakened. Using the `if` statement, the withdraw task may wait forever. Using the loop statement, the withdraw task will have a chance to recheck the condition. Thus you should always test the condition in a loop.

**Caution**

Once a thread invokes `await()` on a condition, the thread waits for a signal to resume. If you forget to call `signal()` or `signalAll()` on the condition, the thread will wait forever.

**Caution**

A condition is created from a `Lock` object. To invoke its method (e.g., `await()`, `signal()`, and `signalAll()`), you must first own the lock. If you invoke these methods without acquiring the lock, an `IllegalMonitorStateException` will be thrown.

Locks and conditions were introduced in Java 5. Prior to Java 5, thread communications were programmed using the object’s built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor, so you can ignore this section. However, if you are working with legacy Java code, you may encounter Java’s built-in monitor.

A `monitor` is an object with mutual exclusion and synchronization capabilities. Only one thread at a time can execute a method in the monitor. A thread enters the monitor by acquiring a lock on it and exits by releasing the lock. Any object can be a monitor. An object becomes a monitor once a thread locks it. Locking is implemented using the `synchronized` keyword on a method or a block. A thread must acquire a lock before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor. You can invoke the `wait()` method on the monitor object to release the lock so that some other thread can get in the monitor and perhaps change the monitor’s state. When the condition is right, the other thread can invoke the `notify()` or `notifyAll()` method to signal one or all waiting threads to regain the lock and resume execution. The template for invoking these methods is shown in Figure 32.18.

---

**Figure 32.18** The `wait()`, `notify()`, and `notifyAll()` methods coordinate thread communication.
The `wait()`, `notify()`, and `notifyAll()` methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an `IllegalMonitorStateException` will occur.

When `wait()` is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.

The `wait()`, `notify()`, and `notifyAll()` methods on an object are analogous to the `await()`, `signal()`, and `signalAll()` methods on a condition.

32.19 How do you create a condition on a lock? What are the `await()`, `signal()`, and `signalAll()` methods for?

32.20 What would happen if the `while` loop in line 58 of Listing 32.9 were changed to an `if` statement?

32.21 Why does the following class have a syntax error?

```java
import javax.swing.*;

public class Test extends JApplet implements Runnable {
    public void init() throws InterruptedException {
        Thread thread = new Thread(this);
        thread.sleep(1000);
    }

    public synchronized void run() {
    }
}
```

32.22 What is a possible cause for `IllegalMonitorStateException`?

32.23 Can the `wait()`, `notify()`, and `notifyAll()` be invoked from any object? What is the purpose of these methods?

32.24 What is wrong in the following code?

```java
synchronized (object1) {
    try {
        while (!condition) object2.wait();
    } catch (InterruptedException ex) {
    }
}
```

32.12 Case Study: Producer/Consumer

This section gives the classic Consumer/Producer example for demonstrating thread coordination.

Suppose you use a buffer to store integers, and that the buffer size is limited. The buffer provides the method `write(int)` to add an `int` value to the buffer and the method `read()` to read and delete an `int` value from the buffer. To synchronize the operations, use a lock with two conditions: `notEmpty` (i.e., the buffer is not empty) and `notFull` (i.e., the buffer is not full). When a task adds an `int` to the buffer, if the buffer is full, the task will wait for the `notFull` condition. When a task deletes an `int` from the buffer, if the buffer is empty, the
task will wait for the notEmpty condition. The interaction between the two tasks is shown in Figure 32.19.

![Figure 32.19](image)

The conditions notFull and notEmpty are used to coordinate task interactions.

Listing 32.10 presents the complete program. The program contains the Buffer class (lines 48–95) and two tasks for repeatedly adding and consuming numbers to and from the buffer (lines 16–45). The write(int) method (line 60) adds an integer to the buffer. The read() method (line 77) deletes and returns an integer from the buffer.

The buffer is actually a first-in, first-out queue (lines 50–51). The conditions notEmpty and notFull on the lock are created in lines 57–58. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the wait() and notify() methods to rewrite this example, you have to designate two objects as monitors.

**LISTING 32.10 ConsumerProducer.java**

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

public class ConsumerProducer {

  private static Buffer buffer = new Buffer();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }

  // A task for adding an int to the buffer
  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer writes "+ i);
          buffer.write(i++); // Add a value to the buffer
          // Put the thread to sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      } catch (InterruptedException ex) {
```
ex.printStackTrace();
}
}

// A task for reading and deleting an int from the buffer
private static class ConsumerTask implements Runnable {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer reads " + buffer.read());
                Thread.sleep((int)(Math.random() * 10000));
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

// An inner class for buffer
private static class Buffer {
    private static final int CAPACITY = 1; // buffer size
    private java.util.LinkedList<Integer> queue =
        new java.util.LinkedList<Integer>();

    // Create a new lock
    private static Lock lock = new ReentrantLock();

    // Create two conditions
    private static Condition notEmpty = lock.newCondition();
    private static Condition notFull = lock.newCondition();

    public void write(int value) {
        lock.lock(); // Acquire the lock
        try {
            while (queue.size() == CAPACITY) {
                System.out.println("Wait for notFull condition");
                notFull.await();
            }

            queue.offer(value);
            notFull.signal(); // Signal notFull condition
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        } finally {
            lock.unlock(); // Release the lock
        }
    }

    public int read() {
        int value = 0;
        lock.lock(); // Acquire the lock
        try {
            while (queue.isEmpty()) {
                System.out.println("Wait for notEmpty condition");
                notEmpty.await();
            }

            value = queue.remove();
            notEmpty.signal(); // Signal notEmpty condition
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        } finally {
            lock.unlock(); // Release the lock
        }
    }
}
A sample run of the program is shown in Figure 32.20.

![Figure 32.20](image_url)  

**Figure 32.20**  Locks and conditions are used for communications between the Producer and Consumer threads.

### 32.25 Check Point

**Can the read and write methods in the Buffer class be executed concurrently?**

### 32.26 Check Point

**When invoking the read method, what happens if the queue is empty?**

### 32.27 Check Point

**When invoking the write method, what happens if the queue is full?**

### 32.13 Blocking Queues

*Java Collections Framework provides ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue for supporting blocking queues.*

Queues and priority queues were introduced in Section 22.9. A **blocking queue** causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue. The **BlockingQueue** interface extends java.util.Queue and provides the synchronized put and take methods for adding an element to the tail of the queue and for removing an element from the head of the queue, as shown in Figure 32.21.

Three concrete blocking queues—ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue—are provided in Java, as shown in Figure 32.22. All are in the java.util.concurrent package. ArrayBlockingQueue implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue. LinkedBlockingQueue implements a blocking queue using a linked list. You can create an unbounded or bounded LinkedBlockingQueue. PriorityBlockingQueue is a priority queue. You can create an unbounded or bounded priority queue.

**Note**

The put method will never block an unbounded LinkedBlockingQueue or PriorityBlockingQueue.
Listing 32.11 gives an example of using an `ArrayBlockingQueue` to simplify the Consumer/Producer example in Listing 32.10. Line 5 creates an `ArrayBlockingQueue` to store integers. The Producer thread puts an integer into the queue (line 22), and the Consumer thread takes an integer from the queue (line 37).

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

public class ConsumerProducerUsingBlockingQueue {
  private static ArrayBlockingQueue<Integer> buffer =
      new ArrayBlockingQueue<Integer>(2);

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }

  // A task for adding an int to the buffer

  public class ProducerTask implements Runnable {
    @Override
    public void run() {
      // The Producer writes 20 integers to the buffer
      for (int i = 0; i < 20; i++) {
        buffer.put(i);
      }
    }
  }

  public class ConsumerTask implements Runnable {
    @Override
    public void run() {
      // The Consumer reads the integers from the buffer
      for (int i = 0; i < 20; i++) {
        int item = buffer.take();
        System.out.println(item);
      }
    }
  }
}
```

**Figure 32.21**  `BlockingQueue` is a subinterface of `Queue`.

**Figure 32.22**  `ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue` are concrete blocking queues.

Listing 32.11 gives an example of using an `ArrayBlockingQueue` to simplify the Consumer/Producer example in Listing 32.10. Line 5 creates an `ArrayBlockingQueue` to store integers. The Producer thread puts an integer into the queue (line 22), and the Consumer thread takes an integer from the queue (line 37).
private static class ProducerTask implements Runnable {
    public void run() {
        try {
            int i = 1;
            while (true) {
                System.out.println("Producer writes " + i);
                buffer.put(i++); // Add any value to the buffer, say, 1
                // Put the thread to sleep
                Thread.sleep((int)(Math.random() * 10000));
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

private static class ConsumerTask implements Runnable {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer reads " + buffer.take());
                // Put the thread to sleep
                Thread.sleep((int)(Math.random() * 10000));
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

In Listing 32.10, you used locks and conditions to synchronize the Producer and Consumer threads. In this program, hand coding is not necessary, because synchronization is already implemented in ArrayBlockingQueue.

32.28 What is a blocking queue? What blocking queues are supported in Java?
32.29 What method do you use to add an element to an ArrayBlockingQueue? What happens if the queue is full?
32.30 What method do you use to retrieve an element from an ArrayBlockingQueue? What happens if the queue is empty?

32.14 Semaphores

Semaphores can be used to restrict the number of threads that access a shared resource.

In computer science, a semaphore is an object that controls the access to a common resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown in Figure 32.23.

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure 32.24. A task acquires a permit by invoking the semaphore’s acquire() method and releases the permit by invoking the semaphore’s release() method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1. Once a permit is released, the total number of available permits in a semaphore is increased by 1.
A semaphore with just one permit can be used to simulate a mutually exclusive lock.

Listing 32.12 revises the `Account` inner class in Listing 32.9 using a semaphore to ensure that only one thread at a time can access the `deposit` method.

**LISTING 32.12 New Account Inner Class**

```java
// An inner class for Account
private static class Account {
    // Create a semaphore
    private static Semaphore semaphore = new Semaphore(1);
    private int balance = 0;

    public int getBalance() {
        return balance;
    }

    public void deposit(int amount) {
        try {
            semaphore.acquire(); // Acquire a permit
            int newBalance = balance + amount;
            // This delay is deliberately added to magnify the
data-corruption problem and make it easy to see
            Thread.sleep(5);
            balance = newBalance;
        } catch (InterruptedException ex) {
            semaphore.release(); // Release the permit
        }
    }
}
```

A semaphore with just one permit can be used to simulate a mutually exclusive lock. Listing 32.12 revises the `Account` inner class in Listing 32.9 using a semaphore to ensure that only one thread at a time can access the `deposit` method.
32.15 Avoiding Deadlocks

Deadlocks can be avoided by using a proper resource ordering.

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause a deadlock, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown in Figure 32.25. Thread 1 has acquired a lock on object1, and Thread 2 has acquired a lock on object2. Now Thread 1 is waiting for the lock on object2, and Thread 2 for the lock on object1. Each thread waits for the other to release the lock it needs, and until that happens, neither can continue to run.

Deadlock is easily avoided by using a simple technique known as resource ordering. With this technique, you assign an order to all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For the example in Figure 32.25, suppose that the objects are ordered as object1 and object2. Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2. Once Thread 1 acquires a lock on object1, Thread 2 has to wait for a lock on object1. Thus, Thread 1 will be able to acquire a lock on object2 and no deadlock will occur.
32.16 Thread States

A thread state indicates the status of thread.

Tasks are executed in threads. Threads can be in one of five states: New, Ready, Running, Blocked, or Finished (see Figure 32.26).

When a thread is newly created, it enters the New state. After a thread is started by calling its start() method, it enters the Ready state. A ready thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.

When a ready thread begins executing, it enters the Running state. A running thread can enter the Ready state if its given CPU time expires or its yield() method is called.

A thread can enter the Blocked state (i.e., become inactive) for several reasons. It may have invoked the join(), sleep(), or wait() method. It may be waiting for an I/O operation to finish. A blocked thread may be reactivated when the action inactivating it is reversed. For example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the Ready state.

Finally, a thread is Finished if it completes the execution of its run() method.

The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.

The interrupt() method interrupts a thread in the following way: If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and a java.lang.InterruptedException is thrown.

32.34 What is a thread state? Describe the states for a thread.

32.17 Synchronized Collections

Java Collections Framework provides synchronized collections for lists, sets, and maps.

The classes in the Java Collections Framework are not thread-safe; that is, their contents may become corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or by using synchronized collections.
The `Collections` class provides six static methods for wrapping a collection into a synchronized version, as shown in Figure 32.27. The collections created using these methods are called *synchronization wrappers*.

**Figure 32.27** You can obtain synchronized collections using the methods in the `Collections` class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>synchronizedCollection(c: Collection): Collection</code></td>
<td>Returns a synchronized collection.</td>
</tr>
<tr>
<td><code>synchronizedList(list: List): List</code></td>
<td>Returns a synchronized list from the specified list.</td>
</tr>
<tr>
<td><code>synchronizedMap(m: Map): Map</code></td>
<td>Returns a synchronized map from the specified map.</td>
</tr>
<tr>
<td><code>synchronizedSet(s: Set): Set</code></td>
<td>Returns a synchronized set from the specified set.</td>
</tr>
<tr>
<td><code>synchronizedSortedMap(s: SortedMap): SortedMap</code></td>
<td>Returns a synchronized sorted map from the specified sorted map.</td>
</tr>
<tr>
<td><code>synchronizedSortedSet(s: SortedSet): SortedSet</code></td>
<td>Returns a synchronized sorted set.</td>
</tr>
</tbody>
</table>

Invoking `synchronizedCollection(Collection c)` returns a new `Collection` object, in which all the methods that access and update the original collection `c` are synchronized. These methods are implemented using the `synchronized` keyword. For example, the `add` method is implemented like this:

```java
public boolean add(E o) {
    synchronized (this) {
        return c.add(o);
    }
}
```

Synchronized collections can be safely accessed and modified by multiple threads concurrently.

**Note**

The methods in `java.util.Vector`, `java.util.Stack`, and `java.util.Hashtable` are already synchronized. These are old classes introduced in JDK 1.0. Starting with JDK 1.5, you should use `java.util.ArrayList` to replace `Vector`, `java.util.LinkedList` to replace `Stack`, and `java.util.Map` to replace `Hashtable`. If synchronization is needed, use a synchronization wrapper.

The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing `java.util.ConcurrentModificationException`, which is a subclass of `RuntimeException`. To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it. For example, to traverse a set, you have to write the code like this:

```java
Set hashSet = Collections.synchronizedSet(new HashSet());

synchronized (hashSet) {
    Iterator iterator = hashSet.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```
Failure to do so may result in nondeterministic behavior, such as a `ConcurrentModificationException`.

32.35 What is a synchronized collection? Is `ArrayList` synchronized? How do you make it synchronized?

32.36 Explain why an iterator is fail-fast.

32.18 Parallel Programming

*The Fork/Join Framework is used for parallel programming in Java.*

The widespread use of multicore systems has created a revolution in software. In order to benefit from multiple processors, software needs to run in parallel. JDK 7 introduces the new Fork/Join Framework for parallel programming, which utilizes the multicore processors.

The *Fork/Join Framework* is illustrated in Figure 32.28 (the diagram resembles a fork, hence its name). A problem is divided into nonoverlapping subproblems, which can be solved independently in parallel. The solutions to all subproblems are then joined to obtain the overall solution for the problem. This is the parallel implementation of the divide-and-conquer approach. In JDK 7’s Fork/Join Framework, a *fork* can be viewed as an independent task that runs on a thread.

![Fork/Join Framework Diagram](image)

**Figure 32.28** The nonoverlapping subproblems are solved in parallel.

The framework defines a task using the `ForkJoinTask` class, as shown in Figure 32.29, and executes a task in an instance of `ForkJoinPool`, as shown in Figure 32.30.

`ForkJoinTask` is the abstract base class for tasks. A `ForkJoinTask` is a thread-like entity, but it is much lighter than a normal thread, because huge numbers of tasks and subtasks can be executed by a small number of actual threads in a `ForkJoinPool`. The tasks are primarily coordinated using *fork()* and *join()* methods. Invoking *fork()* on a task arranges asynchronous execution, and invoking *join()* waits until the task is completed. The *invoke()* and *invokeAll*(tasks) methods implicitly invoke *fork()* to execute the task and *join()* to wait for the tasks to complete, and return the result, if any. Note that the static method *invokeAll* takes a variable number of `ForkJoinTask` arguments using the ... syntax, which was introduced in Section 6.9.

The Fork/Join Framework is designed to parallelize divide-and-conquer solutions, which are naturally recursive. `RecursiveAction` and `RecursiveTask` are two subclasses of `ForkJoinTask`. To define a concrete task class, your class should extend `RecursiveAction` or `RecursiveTask`. `RecursiveAction` is for a task that doesn’t return a value, and `RecursiveTask` is for a task that does return a value. Your task class should override the *compute()* method to specify how a task is performed.

We now use a merge sort to demonstrate how to develop parallel programs using the Fork/Join Framework. The merge sort algorithm (introduced in Section 25.3) divides the
Figure 32.29 The ForkJoinTask class defines a task for asynchronous execution.

Figure 32.30 The ForkJoinPool executes Fork/Join tasks.

array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm merges them. Listing 32.13 gives a parallel implementation of the merge sort algorithm and compares its execution time with a sequential sort.

Listing 32.13 ParallelMergeSort.java

```
1 import java.util.concurrent.RecursiveAction;
2 import java.util.concurrent.ForkJoinPool;
3
4 public class ParallelMergeSort {
5   public static void main(String[] args) {
```
final int SIZE = 7000000;
int[] list1 = new int[SIZE];
int[] list2 = new int[SIZE];

for (int i = 0; i < list1.length; i++)
    list1[i] = list2[i] = (int)(Math.random() * 10000000);

long startTime = System.currentTimeMillis();
parallelMergeSort(list1); // Invoke parallel merge sort
long endTime = System.currentTimeMillis();
System.out.println("Parallel time with "+
    Runtime.getRuntime().availableProcessors() + 
    " processors is "+ (endTime - startTime) + " milliseconds");

startTime = System.currentTimeMillis();
MergeSort.mergeSort(list2); // MergeSort is in Listing 25.5
endTime = System.currentTimeMillis();
System.out.println("Sequential time is" +
    (endTime - startTime) + " milliseconds");

public static void parallelMergeSort(int[] list) {

RecursiveAction mainTask = new SortTask(list);
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(mainTask);
}

private static class SortTask extends RecursiveAction {

private final int THRESHOLD = 500;
private int[] list;

SortTask(int[] list) {
    this.list = list;
}

@Override
protected void compute() {
    if (list.length < THRESHOLD)
        java.util.Arrays.sort(list);
    else {
        // Obtain the first half
        int[] firstHalf = new int[list.length / 2];
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);

        // Obtain the second half
        int secondHalfLength = list.length - list.length / 2;
        int[] secondHalf = new int[secondHalfLength];
        System.arraycopy(list, list.length / 2, secondHalf, 0, secondHalfLength);

        // Recursively sort the two halves
        invokeAll(new SortTask(firstHalf),
            new SortTask(secondHalf));

        // Merge firstHalf with secondHalf into list
        MergeSort.merge(firstHalf, secondHalf, list);
    }
}
}
Parallel time with 2 processors is 2829 milliseconds
Sequential time is 4751 milliseconds

Since the sort algorithm does not return a value, we define a concrete ForkJoinTask class by extending RecursiveAction (lines 33–64). The compute method is overridden to implement a recursive merge sort (lines 42–63). If the list is small, it is more efficient to be solved sequentially (line 44). For a large list, it is split into two halves (lines 47–54). The two halves are sorted concurrently (lines 57–58) and then merged (line 61).

The program creates a main ForkJoinTask (line 28), a ForkJoinPool (line 29), and places the main task for execution in a ForkJoinPool (line 30). The invoke method will return after the main task is completed.

When executing the main task, the task is split into subtasks and the subtasks are invoked using the invokeAll method (lines 57–58). The invokeAll method will return after all the subtasks are completed. Note that each subtask is further split into smaller tasks recursively. Huge numbers of subtasks may be created and executed in the pool. The ForkJoin Framework automatically executes and coordinates all the tasks efficiently.

The MergeSort class is defined in Listing 25.5. The program invokes MergeSort.merge to merge two sorted sublists (line 61). The program also invokes MergeSort.mergeSort (line 21) to sort a list using merge sort sequentially. You can see that the parallel sort is much faster than the sequential sort.

Note that the loop for initializing the list can also be parallelized. However, you should avoid using Math.random() in the code, because it is synchronized and cannot be executed in parallel (see Programming Exercise 32.12). The parallelMergeSort method only sorts an array of int values, but you can modify it to become a generic method (see Programming Exercise 32.13).

In general, a problem can be solved in parallel using the following pattern:

```java
if (the program is small)
    solve it sequentially;
else {
    divide the problem into nonoverlapping subproblems;
    solve the subproblems concurrently;
    combine the results from subproblems to solve the whole problem;
}
```

Listing 32.14 develops a parallel method that finds the maximal number in a list.

**LISTING 32.14  ParallelMax.java**

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

public class ParallelMax {
    public static void main(String[] args) {
        // Create a list
        final int N = 9000000;
        int[] list = new int[N];
        for (int i = 0; i < list.length; i++)
            list[i] = i;

        long startTime = System.currentTimeMillis();
        System.out.println("The maximal number is " + max(list));
        long endTime = System.currentTimeMillis();
        System.out.println("The number of processors is " + Runtime.getRuntime().availableProcessors());
        System.out.println("Time is " + (endTime - startTime));
    }
}
```

invoke max
32.18 Parallel Programming

+ " milliseconds"

public static int max(int[] list) {
    RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
    ForkJoinPool pool = new ForkJoinPool();
    return pool.invoke(task);
}

private static class MaxTask extends RecursiveTask<Integer> {
    private final static int THRESHOLD = 1000;
    private int[] list;
    private int low;
    private int high;

    public MaxTask(int[] list, int low, int high) {
        this.list = list;
        this.low = low;
        this.high = high;
    }

    @Override
    public Integer compute() {
        if (high - low < THRESHOLD) {
            int max = list[0];
            for (int i = low; i < high; i++)
                if (list[i] > max)
                    max = list[i];
            return new Integer(max);
        } else {
            int mid = (low + high) / 2;
            RecursiveTask<Integer> left = new MaxTask(list, low, mid);
            RecursiveTask<Integer> right = new MaxTask(list, mid, high);
            right.fork();
            left.fork();
            return new Integer(Math.max(left.join().intValue(),
                                         right.join().intValue()));
        }
    }
}

The maximal number is 8999999
The number of processors is 2
Time is 44 milliseconds

Since the algorithm returns an integer, we define a task class for fork join by extending RecursiveTask<Integer> (lines 26–58). The compute method is overridden to return the max element in a list[low..high] (lines 39–57). If the list is small, it is more efficient to be solved sequentially (lines 40–46). For a large list, it is split into two halves (lines 48–50). The tasks left and right find the maximal element in the left half and right half, respectively. Invoking fork() on the task causes the task to be executed (lines 52–53). The join() method awaits for the task to complete and then returns the result (lines 54–55).
### 32.37 How do you define a `ForkJoinTask`? What are the differences between `RecursiveAction` and `RecursiveTask`?

### 32.38 How do you tell the system to execute a task?

### 32.39 What method can you use to test if a task has been completed?

### 32.40 How do you create a `ForkJoinPool`? How do you place a task into a `ForkJoinPool`?

#### Key Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>1150</td>
</tr>
<tr>
<td>deadlock</td>
<td>1162</td>
</tr>
<tr>
<td>event dispatch thread</td>
<td>1138</td>
</tr>
<tr>
<td>fail-fast</td>
<td>1164</td>
</tr>
<tr>
<td>fairness policy</td>
<td>1149</td>
</tr>
<tr>
<td>Fork/Join Framework</td>
<td>1165</td>
</tr>
<tr>
<td>lock</td>
<td>1148</td>
</tr>
<tr>
<td>monitor</td>
<td>1154</td>
</tr>
<tr>
<td>multithreading</td>
<td>1130</td>
</tr>
<tr>
<td>race condition</td>
<td>1147</td>
</tr>
<tr>
<td>semaphore</td>
<td>1160</td>
</tr>
<tr>
<td>synchronization wrapper</td>
<td>1164</td>
</tr>
<tr>
<td>synchronized block</td>
<td>1148</td>
</tr>
<tr>
<td>thread</td>
<td>1130</td>
</tr>
<tr>
<td>thread-safe</td>
<td>1147</td>
</tr>
</tbody>
</table>

#### Chapter Summary

1. Each task is an instance of the `Runnable` interface. A `thread` is an object that facilitates the execution of a task. You can define a task class by implementing the `Runnable` interface and create a thread by wrapping a task using a `Thread` constructor.

2. After a thread object is created, use the `start()` method to start a thread, and the `sleep(long)` method to put a thread to sleep so that other threads get a chance to run.

3. A thread object never directly invokes the `run` method. The JVM invokes the `run` method when it is time to execute the thread. Your class must override the `run` method to tell the system what the thread will do when it runs.

4. To prevent threads from corrupting a shared resource, use `synchronized` methods or blocks. A `synchronized method` acquires a `lock` before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static (class) method, the lock is on the class.

5. A `synchronized statement` can be used to acquire a lock on any object, not just `this` object, when executing a block of the code in a method. This block is referred to as a `synchronized block`.

6. You can use explicit locks and `conditions` to facilitate communications among threads, as well as using the built-in monitor for objects.

7. `Deadlock` occurs when two or more threads acquire locks on multiple objects and each has a lock on one object and is waiting for the lock on the other object. The `resource ordering technique` can be used to avoid deadlock.

8. The JDK 7’s Fork/Join Framework is designed for developing parallel programs. You can define a task class that extends `RecursiveAction` or `RecursiveTask` and execute the tasks concurrently in `ForkJoinPool`, and obtains the overall solution after all tasks are completed.
**Test Questions**

Do the test questions for this chapter online at [www.cs.armstrong.edu/liang/intro9e/test.html](http://www.cs.armstrong.edu/liang/intro9e/test.html).

**Programming Exercises**

**Sections 32.1–32.5**

*32.1* (Revise Listing 32.1) Rewrite Listing 32.1 to display the output in a text area, as shown in Figure 32.31.

![Figure 32.31](image)

The output from three threads is displayed in a text area.

32.2 *(Racing cars)* Rewrite Programming Exercise 18.17 using a thread to control car racing. Compare the program with Programming Exercise 18.17 by setting the delay time to 10 in both programs. Which one runs the animation faster?

32.3 *(Raise flags)* Rewrite Programming Exercise 18.23 using a thread to animate a flag being raised. Compare the program with Programming Exercise 18.23 by setting the delay time to 10 in both programs. Which one runs the animation faster?

**Sections 32.8–32.12**

32.4 *(Synchronize threads)* Write a program that launches 1,000 threads. Each thread adds 1 to a variable `sum` that initially is 0. You need to pass `sum` by reference to each thread. In order to pass it by reference, define an `Integer` wrapper object to hold `sum`. Run the program with and without synchronization to see its effect.

32.5 *(Run fans)* Rewrite Programming Exercise 18.11 using a thread to control the fan animation.

32.6 *(Bouncing balls)* Rewrite Programming Exercise 18.19 using a thread to animate bouncing ball movements.

32.7 *(Control a group of clocks)* Rewrite Programming Exercise 18.14 using a thread to control the clock animation.

32.8 *(Account synchronization)* Rewrite Listing 32.9, ThreadCooperation.java, using the object’s `wait()` and `notifyAll()` methods.

32.9 *(Demonstrate ConcurrentModificationException)* The iterator is *fail-fast*. Write a program to demonstrate it by creating two threads that concurrently access and modify a set. The first thread creates a hash set filled with numbers, and adds a new number to the set every second. The second thread obtains an iterator for the set and traverses the set back and forth through the iterator every second. You will receive a `ConcurrentModificationException` because the underlying set is being modified in the first thread while the set in the second thread is being traversed.
*32.10  (Use synchronized sets) Using synchronization, correct the problem in the preceding exercise so that the second thread does not throw a `ConcurrentModificationException`.

Section 32.15
*32.11  (Demonstrate deadlock) Write a program that demonstrates deadlock.

Section 32.18
*32.12  (Parallel array initializer) Implement the following method using the Fork/Join Framework to assign random values to the list.

```java
public static void parallelAssignValues(double[] list)
```

Write a test program that creates a list with 9,000,000 elements and invokes `parallelAssignValues` to assign random values to the list. Also implement a sequential algorithm and compare the execution time of the two. Note that if you use `Math.random()`, your parallel code execution time will be worse than the sequential code execution time, because `Math.random()` is synchronized and cannot be executed in parallel. To fix this problem, create a `Random` object for assigning random values to a small list.

32.13  (Generic parallel merge sort) Revise Listing 32.13, `ParallelMergeSort.java`, to define a generic `parallelMergeSort` method as follows:

```java
public static void <E extends Comparable<E>> parallelMergeSort(E[] list)
```

*32.14  (Parallel quick sort) Implement the following method in parallel to sort a list using quick sort (see Listing 25.7).

```java
public static void parallelQuickSort(int[] list)
```

Write a test program that times the execution time for a list of size 9,000,000 using this parallel method and a sequential method.

32.15  (Parallel sum) Implement the following method using Fork/Join to find the sum of a list.

```java
public static double parallelSum(double[] list)
```

Write a test program that finds the sum in a list of 9,000,000 double values.

*32.16  (Parallel matrix addition) Programming Exercise 7.5 describes how to perform matrix addition. Suppose you have multiple processors, so you can speed up the matrix addition. Implement the following method in parallel.

```java
public static double[][] parallelAddMatrix(
    double[][] a,
    double[][] b)
```

Write a test program that times the execution time for adding two $2,000 \times 2,000$ matrices.

*32.17  (Parallel matrix multiplication) Programming Exercise 7.6 describes how to perform matrix multiplication. Suppose you have multiple processors, so you can speed up the matrix multiplication. Implement the following method in parallel.

```java
public static double[][] parallelMultiplyMatrix(
    double[][] a,
    double[][] b)
```
Write a test program that times the execution time for multiplying two 2,000 × 2,000 matrices.

**32.18** *(Parallel Eight Queens)* Revise Listing 24.10, EightQueens.java, to develop a parallel algorithm that finds all solutions for the Eight Queens problem. *(Hint: Launch eight subtasks, each of which places the queen in a different column in the first row.)*

**Comprehensive**

**32.19** *(Sorting animation)* Write an animation applet for selection sort, insertion sort, and bubble sort, as shown in Figure 32.32. Create an array of integers 1, 2, . . . , 50. Shuffle it randomly. Create a panel to display the array in a histogram. You should invoke each sort method in a separate thread. Each algorithm uses two nested loops. When the algorithm completes an iteration in the outer loop, put the thread to sleep for 0.5 seconds, and redisplay the array in the histogram. Color the last bar in the sorted subarray.

![Sorting animation](image1)

(a) Sorting in progress

![Sorting animation](image2)

(b) Sorted

**Figure 32.32** Three sorting algorithms are illustrated in the animation.

**32.20** *(Sudoku search animation)* Modify Programming Exercise 24.21 to display the intermediate results of the search. As shown in Figure 32.33a, the number 2 is placed in the first row and last column, because 2 already appears in the same row. This number is invalid, so the next value, 3, is placed in Figure 32.33b. This number is also invalid, because 3 already appears in the same row; so the next value, 4, is placed in Figure 32.33c. The animation displays all the search steps.

**32.21** *(Eight Queens animation)* Modify Listing 24.10, EightQueens.java, to display the intermediate results of the search. As shown in Figure 32.34a, the current row being searched is highlighted. When the user clicks the mouse button, a position for the row is found and a queen is placed in the row, as shown in Figure 32.34b.
Figure 32.33 The intermediate search steps are displayed in the animation for the Sudoku problem.

Figure 32.34 The intermediate search steps are displayed in the animation for the Eight Queens problem.