Chapter 14

Threads and Concurrent Programming

OBJECTIVES

After studying this chapter, you will

- Understand the concept of a thread.
- Know how to design and write multithreaded programs.
- Be able to use the `Thread` class and the `Runnable` interface.
- Understand the life cycle of a thread.
- Know how to synchronize threads.

OUTLINE

14.1 Introduction

This chapter is about doing more than one thing at a time. Doing more than one thing at once is commonplace in our everyday lives. For example, let's say your breakfast today consists of cereal, toast, and a cup of java. You have to do three things at once to have breakfast: eat cereal, eat toast, and drink coffee.

Actually, you do these things "at the same time" by alternating among them: You take a spoonful of cereal, then a bite of toast, and then sip some coffee. Then you have another bite of toast, or another spoonful of cereal, more coffee, and so on, until breakfast is finished. If the phone rings while you're having breakfast, you will probably answer it—and continue to have breakfast, or at least to sip the coffee. This means you're doing even more "at the same time." Everyday life is full of examples where we do more than one task at the same time.

The computer programs we have written so far have performed one task at a time. But there are plenty of applications where a program needs to do several things at once, or **concurrently**. For example, if you wrote an Internet chat program, it would let several users take part in a discussion group. The program would have to read messages from several users at the same time and broadcast them to the other participants in the group. The reading and broadcasting tasks would have to take place concurrently. In Java, concurrent programming is handled by *threads*, the topic of this chapter.

14.2 What Is a Thread?

A **thread** (or a *thread of execution* or a *thread of control*) is a single sequence of executable statements within a program. For Java applications, the flow of control begins at the first statement in `main()` and continues sequentially through the program statements. For Java applets, the flow of control begins with the first statement in `init()`. Loops within a program cause a certain block of statements to be repeated. If-else structures cause certain statements to be selected and others to be skipped. Method calls cause the flow of execution to jump to another part of the program, from which it returns after the method's statements are executed. Thus, within a single thread, you can trace the sequential flow of execution from one statement to the next.

Visualizing a thread

One way to visualize a thread is to imagine that you could make a list of the program's statements as they are executed by the computer's central processing unit (CPU). Thus, for a particular execution of a program with loops, method calls, and selection statements, you could list each instruction that was executed, beginning at the first, and continuing until the program stopped, as a single sequence of executed statements. That's a thread!

Now imagine that we break a program up into two or more independent threads. Each thread will have its own sequence of instructions. Within a single thread, the statements are executed one after the other, as usual. However, by alternately executing the statements from one thread and another, the computer can run several threads *concurrently*. Even

though the CPU executes one instruction at at time, it can run multiple threads concurrently by rapidly alternating among them. The main advantage of concurrency is that it allows the computer to do more than one task at a time. For example, the CPU could alternate between downloading an image from the Internet and running a spreadsheet calculation. This is the same way you ate toast and cereal and drank coffee in our earlier breakfast example. From our perspective, it might look as if the computer had several CPUs working in parallel, but that's just the illusion created by an effectively scheduling threads.

JAVA LANGUAGE RULE JVM Threads. The Java Virtual Machine (JVM) is itself an example of a multithreaded program. JVM threads perform tasks that are essential to the successful execution of Java programs.

JAVA LANGUAGE RULE Garbage Collector Thread. One of the JVM threads, the *garbage collector thread*, automatically reclaims memory taken up by objects that are not used in your programs. This happens at the same time that the JVM is interpreting your program.

14.2.1 Concurrent Execution of Threads

Multitasking

The technique of concurrently executing several tasks within a program is known as **multitasking**. A **task** in this sense is a computer operation of some sort, such as reading or saving a file, compiling a program, or displaying an image on the screen. Multitasking requires the use of a separate thread for each of the tasks. The methods available in the Java `Thread` class make it possible (and quite simple) to implement **multithreaded** programs.

Most computers, including personal computers, are *sequential* machines that consist of a single CPU, which is capable of executing one machine instruction at a time. In contrast, *parallel computers*, used primarily for large scale scientific and engineering applications, are made up of multiple CPUs working in tandem.

Today's personal computers, running at clock speeds over 1 gigahertz—1 *gigahertz* equals 1 billion cycles per second—are capable of executing millions of machine instructions per second. Despite its great speed, however, a single CPU can process only one instruction at a time.

Each CPU uses a **fetch-execute cycle** to retrieve the next instruction from memory and execute it. Since CPUs can execute only one instruction at a time, multithreaded programs are made possible by dividing the CPU's time and sharing it among the threads. The CPU's schedule is managed by a **scheduling algorithm**, which is an algorithm that schedules threads for execution on the CPU. The choice of a scheduling algorithm depends on the platform on which the program is running. Thus, thread scheduling might be handled differently on Unix, Windows, and Macintosh systems.

CPUs are sequential

One common scheduling technique is known as **time slicing**, in which

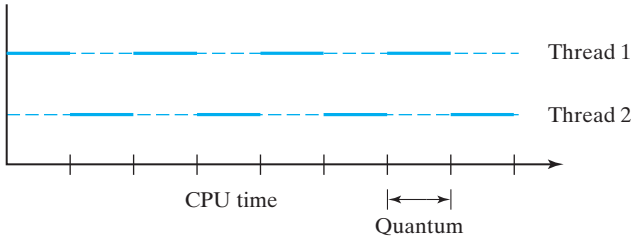

Figure 14.1: Each thread gets a slice of the CPU's time.

each thread alternatively gets a slice of the CPU's time. For example, suppose we have a program that consists of two threads. Using this technique, the system would give each thread a small **quantum** of CPU time—say, one thousandth of a second (one *millisecond*)—to execute its instructions. When its quantum expires, the thread would be *preempted* and the other thread would be given a chance to run. The algorithm would then alternate in this **round-robin** fashion between one thread and the other (Fig. 14.1). During each millisecond on a 300-megahertz CPU, a thread can execute 300,000 machine instructions. One **megahertz** equals 1 million cycles per second. Thus, within each second of real time, each thread will receive 500 time slices and will be able to execute something like 150 million machine instructions.

Time slicing

Priority scheduling

Under **priority scheduling**, threads of higher priority are allowed to run to completion before lower-priority threads are given a chance. An example of a high-priority thread would be one that is processing keyboard input or any other kind of interactive input from the user. If such tasks were given low priority, users would experience noticeable delays in their interaction, which would be quite unacceptable.

The only way a high-priority thread can be preempted is if a thread of still higher priority becomes available to run. In many cases, higher-priority threads are those that can complete their task within a few milliseconds, so they can be allowed to run to completion without starving the lower-priority threads. An example would be processing a user's keystroke, a task that can begin as soon as the key is struck and can be completed very quickly. Starvation occurs when one thread is repeatedly preempted by other threads.

JAVA LANGUAGE RULE Thread Support. Depending on the hardware platform, Java threads can be supported by assigning different threads to different processors, by time slicing a single processor, or by time slicing many hardware processors.

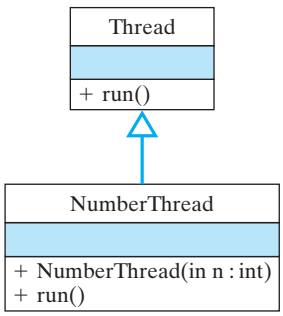

FIGURE 14.2 The `NumberThread` class overrides the inherited `run()` method.

14.2.2 Multithreaded Numbers

Let's consider a simple example of a threaded program. Suppose we give every individual thread a unique ID number, and each time it runs, it prints its ID ten times. For example, when the thread with ID 1 runs the output produced would just be a sequence of ten 1's: 1111111111.

As shown in Figure 14.2, the `NumberThread` class is defined as a subclass of `Thread` and overrides the `run()` method. To set the thread's ID

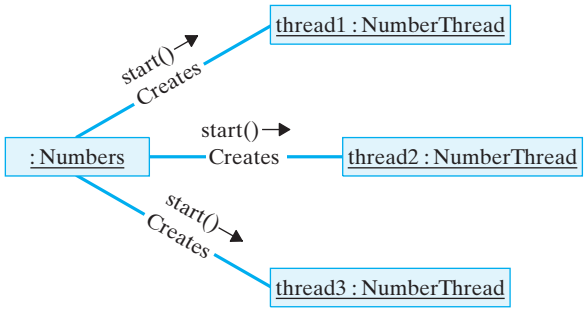

Figure 14.3: The `Numbers` object creates several instances of `NumberThread` and tells each one to `start()`.

number, the constructor takes a single parameter that is used to set the thread's ID number. In the `run()` method, the thread simply executes a loop that prints its own number ten times:

```
public class NumberThread extends Thread {
    int num;

    public NumberThread(int n) {
        num = n;
    }
    public void run() {
        for (int k=0; k < 10; k++) {
            System.out.print(num);
        } //for
    } // run()
} // NumberThread
```

`Thread` subclass

Now let's define another class whose task will be to create many `NumberThread`s and get them all running at the same time (Fig. 14.3). For each `NumberThread`, we want to call its constructor and then `start()` it:

```
public class Numbers {
  public static void main(String args[]) {
                                    // 5 threads
    NumberThread number1, number2, number3, number4, number5;

     // Create and start each thread
    number1 = new NumberThread(1); number1.start();
    number2 = new NumberThread(2); number2.start();
    number3 = new NumberThread(3); number3.start();
    number4 = new NumberThread(4); number4.start();
    number5 = new NumberThread(5); number5.start();
  } // main()
} // Numbers
```

When a thread is started by calling its `start()` method, it automatically calls its `run()` method. The output generated by this version of

Starting a thread

the `Numbers` application is as follows:

```
1111111111222222222233333333334444444444555555555
```

From this output, it appears that the individual threads were run in the order in which they were created. In this case, each thread was able to run to completion before the next thread started running.

What if we increase the number of iterations that each thread performs? Will each thread still run to completion? The following output was generated for 200 iterations per thread:

```
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111112222222
22222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222233333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333444444444444444444444444444444444444444444
44444444444444444444444444444444444444444444444444444444444444444444
44444444445555555555555555555555555555555555555555555555555555555555
55555555555555555555555555555555555555555555555555555555552222222
22223333333333333333333333333333333333333333333333333333333333333333
33333333333334444444444444444444444444444555555555555555555555555555
55555555555555555555555555555555555555555555555555555555444444444444444
44444444444444444444444444444444
```

In this case, only thread 1 managed to run to completion. Threads 2, 3, 4, and 5 did not. As this example illustrates, the order and timing of a thread's execution are highly unpredictable. This example also serves to illustrate one way of creating a multithreaded program:

- Create a subclass of the `Thread` class.
- Within the subclass, implement a method with the signature `void run()` that contains the statements to be executed by that thread.
- Create several instances of the subclass and start each thread by invoking the `start()` method on each instance.

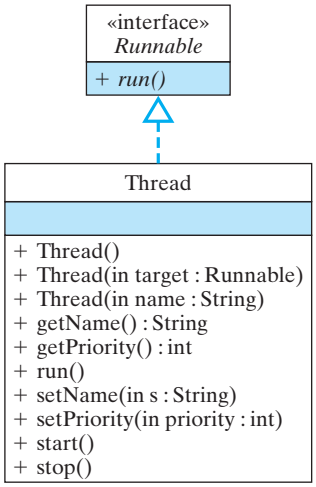

FIGURE 14.4 The `java.lang.Thread` class. NOTE: NEEDS REVISION TO ADD PRIORITY, YIELD() and SLEEP().

JAVA LANGUAGE RULE Thread Creation. One way to create a thread in Java is to define a subclass of `Thread` and override the default `run()` method.

14.3 From the Java Library: `java.lang.Thread`

The `java.lang.Thread` class contains the public methods shown in Figure 14.4 (the figure contains only a partial list). Note that `Thread` implements the `Runnable` interface, which consists simply of the `run()` method. As we will now see, another way to create a thread is to instantiate a `Thread` object and pass it a `Runnable` object that will become its body. This approach allows you to turn an existing class into a separate thread.

A `Runnable` object is any object that implements the `Runnable` interface—that is, any object that implements the `run()` method

(Fig. 14.5). The following example provides an alternative way to implement the NumberThread program:

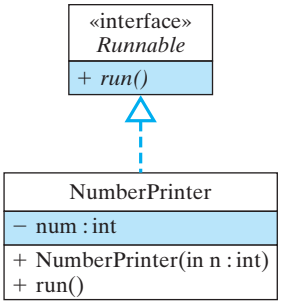

FIGURE 14.5 Any object that implements the Runnable interface can be run as a separate thread.

```
public class NumberPrinter implements Runnable {
    int num;

    public NumberPrinter(int n) {
        num = n;
    }

    public void run() {
        for (int k=0; k < 10; k++)
            System.out.print(num);
    } // run()
} // NumberPrinter
```

Given this definition, we would then pass instances of this class to the individual threads as we create them:

```
public class Numbers {
  public static void main(String args[]) {

    Thread number1, number2, number3, number4, number5;
     // Create and start each thread
    number1 = new Thread(new NumberPrinter(1)); number1.start();
    number2 = new Thread(new NumberPrinter(2)); number2.start();
    number3 = new Thread(new NumberPrinter(3)); number3.start();
    number4 = new Thread(new NumberPrinter(4)); number4.start();
    number5 = new Thread(new NumberPrinter(5)); number5.start();
  } // main()
} // Numbers
```

The NumberPrinter class implements Runnable by defining exactly the same run() that was used previously in the NumberThread class. We then pass instances of NumberPrinter when we create the individual threads. Doing things this way gives exactly the same output as earlier. This example serves to illustrate another way of creating a multithreaded program:

- Implement the Runnable interface for an existing class by implementing the void run() method, which contains the statements to be executed by that thread.
- Create several Thread instances by first creating instances of the Runnable class and passing each instance as an argument to the Thread() constructor.
- For each thread instance, start it by invoking the start() method on it.

JAVA LANGUAGE RULE Thread Creation. A thread can be created by passing a Runnable object to a new Thread instance. The object's run() method will be invoked automatically as soon as the thread's start() method is called.

JAVA EFFECTIVE DESIGN **Converting a Class to a Thread.** Using the `Runnable` interface to create threads enables you to turn an existing class into a thread. For most applications, using the `Runnable` interface is preferable to redefining the class as a `Thread` subclass.

SELF-STUDY EXERCISE

EXERCISE 14.1 Use the `Runnable` interface to convert the following class into a thread. You want the thread to print all the odd numbers up to its bound:

```
public class PrintOdds {
    private int bound;
    public PrintOdds(int b) {
        bound = b;
    }

    public void print() {
        if (int k = 1; k < bound; k+=2)
            System.out.println(k);
    }
} // PrintOdds
```

14.3.1 Thread Control

Controlling threads

The various methods in the `Thread` class (Fig. 14.4) can be used to exert some control over a thread's execution. The `start()` and `stop()` methods play the obvious roles of starting and stopping a thread. These methods will sometimes be called automatically. For example, an applet is treated as a thread by the browser, or appletviewer, which is responsible for starting and stopping it.

As we saw in the `NumberThread` example, the `run()` method encapsulates the thread's basic algorithm. It is usually not called directly. Instead, it is called by the thread's `start()` method, which handles any system-dependent initialization tasks before calling `run()`.

14.3.2 Thread Priority

The `setPriority(int)` method lets you set a thread's priority to an integer value between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`, the bounds defined as constants in the `Thread` class. Using `setPriority()` gives you some control over a thread's execution. In gen-

eral, higher-priority threads get to run before, and longer than, lower-priority threads.

JAVA LANGUAGE RULE Preemption. A higher-priority thread that wants to run will *preempt* any threads of lower priority.

To see how `setPriority()` works, suppose we change `NumberThread`'s constructor to the following:

```
public NumberThread(int n) {
    num = n;
    setPriority(n);
}
```

In this case, each thread sets its priority to its ID number. So, thread five will have priority five, a higher priority than all the other threads. Suppose we now run 2 million iterations of each of these threads. Because 2 million iterations will take a long time if we print the thread's ID on each iteration, let's modify the `run()` method, so that the ID is printed every 1 million iterations:

Thread priority

```
for (int k = 0; k < 10; k++)
    if (k % 1000000 == 0)
        System.out.print(num);
```

Given this modification, we get the following output when we run `Numbers`:

```
5544332211
```

It appears from this output that the threads ran to completion in priority order. Thus, thread five completed 2 million iterations before thread four started to run, and so on. This shows that, on my system at least, the Java Virtual Machine (JVM) supports priority scheduling.

JAVA PROGRAMMING TIP Platform Dependence. Thread implementation in Java is platform dependent. Adequate testing is necessary to ensure that a program will perform correctly on a given platform.

JAVA EFFECTIVE DESIGN Thread Coordination. One way to coordinate the behavior of two threads is to give one thread higher priority than another.

JAVA DEBUGGING TIP Starvation. A high-priority thread that never gives up the CPU can starve lower-priority threads by preventing them from accessing the CPU.

14.3.3 Forcing Threads to Sleep

Sleep versus yield

The `Thread.sleep()` and `Thread.yield()` methods also provide some control over a thread's behavior. When executed by a thread, the `yield()` method causes the thread to yield the CPU, allowing the thread scheduler to choose another thread. The `sleep()` method causes the thread to yield and not to be scheduled until a certain amount of real time has passed.

JAVA LANGUAGE RULE Sleep Versus Yield. Both the `yield()` and `sleep()` methods yield the CPU, but the `sleep()` method keeps the thread from being rescheduled for a fixed amount of real time.

The `sleep()` method can halt a running thread for a given number of milliseconds, allowing other waiting threads to run. The `sleep()` method throws an `InterruptedException`, which is a checked exception. This means that the `sleep()` call must be embedded within a `try/catch` block or the method it's in must throw an `InterruptedException`. Try/catch blocks were covered in Chapter 10.

```
try {
    sleep(100);
} catch (InterruptedException e) {
    System.out.println(e.getMessage());
}
```

For example, consider the following version of the `NumberPrinter.run()`:

```
public void run() {
    for (int k=0; k < 10; k++) {
        try {
            Thread.sleep((long)(Math.random() * 1000));
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.print(num);
    } // for
} // run()
```

In this example, each thread is forced to sleep for a random number of milliseconds between 0 and 1,000. When a thread sleeps, it gives up the CPU, which allows one of the other waiting threads to run. As you would expect, the output we get from this example will reflect the randomness in the amount of time that each thread sleeps:

```
14522314532143154232152423541243235415523113435451
```

As we will see, the `sleep()` method provides a rudimentary form of thread synchronization, in which one thread yields control to another.

SELF-STUDY EXERCISES

EXERCISE 14.2 What happens if you run five `NumberThread`s of equal priority through 2 million iterations each? Run this experiment and note the output. Don't print after every iteration! What sort of scheduling algorithm (round-robin, priority scheduling, or something else) was used to schedule threads of equal priority on your system?

EXERCISE 14.3 Try the following experiment and note the output. Let each thread sleep for 50 milliseconds (rather than a random number of milliseconds). How does this affect the scheduling of the threads? To make things easier to see, print each thread's ID after every 100,000 iterations.

EXERCISE 14.4 The purpose of the Java garbage collector is to recapture memory that was used by objects that are no longer being used by your program. Should its thread have higher or lower priority than your program?

14.3.4 The Asynchronous Nature of Threaded Programs

Threads are **asynchronous**. This means that the order of execution and the timing of a set of threads are unpredictable, at least from the programmer's point of view. Threads are executed under the control of the scheduling algorithm used by the operating system and the Java Virtual Machine. In general, unless threads are explicitly synchronized, it is impossible for the programmer to predict when and for how long an individual thread will run. In some systems, under some circumstances, a thread might run to completion before any other thread can run. In other systems, or under different circumstances, a thread might run for a short time and then be suspended while another thread runs. Of course, when a thread is preempted by the system, its state is saved so that its execution can be resumed without losing any information.

Thread preemptions are unpredictable

One implication of a thread's asynchronicity is that it is not generally possible to determine where in its source code an individual thread might be preempted. You can't even assume that a thread will be able to complete a simple Java arithmetic operation once it has started it. For example, suppose a thread had to execute the following operation:

```
int N = 5 + 3;
```

This operation computes the sum of 5 and 3 and assigns the result to `N`. It would be tempting to think that once the thread started this operation, it would be able to complete it, but that is not necessarily so. You have to remember that Java code is compiled into a rudimentary bytecode, which is translated still further into the computer's machine language. In machine language, this operation would break down into something like the following three steps:

An arithmetic operation can be interrupted

```
Fetch 5 from memory and store it in register A.
Add 3 to register A.
Assign the value in register A to N.
```

Although none of the individual machine instructions can be preempted, the thread could be interrupted between any two machine instructions. The point here is that not even a single Java language instruction can be assumed to be indivisible or unpreemptible. Therefore, it is impossible to make any assumptions about when a particular thread will run and when it will give up the CPU. This suggests the following important principle of multithreaded programs:

Threads are asynchronous

JAVA LANGUAGE RULE Asynchronous Thread Principle. Unless they are explicitly prioritized or synchronized, threads behave in a completely *asynchronous* fashion.

JAVA PROGRAMMING TIP Thread Timing. Unless they are explicitly synchronized, you cannot make any assumptions about when, or in what order, individual threads will execute, or where a thread might be interrupted or preempted during its execution.

As we will see, this principle plays a large role in the design of multithreaded programs.

14.4 Thread States and Life Cycle

Ready, running, and sleeping

Controlling a thread

Each thread has a **life cycle** that consists of several different states, which are summarized in Figure 14.6 and Table 14.1. Thread states are represented by labeled ovals, and the transitions between states are represented by labeled arrows. Much of a thread's life cycle is under the control of the operating system and the Java Virtual Machine. Those transitions represented by method names—such as `start()`, `stop()`, `wait()`, `sleep()`, `notify()`—can be controlled by the program. Of these methods, the `stop()` method has been deprecated in JDK 1.2 because it is inherently unsafe to stop a thread in the middle of its execution. Other transitions—such as *dispatch*, *I/O request*, *I/O done*, *time expired*, *done sleeping*—are under the control of the CPU scheduler. When first created a thread is in the ready state, which means that it is ready to run. In the

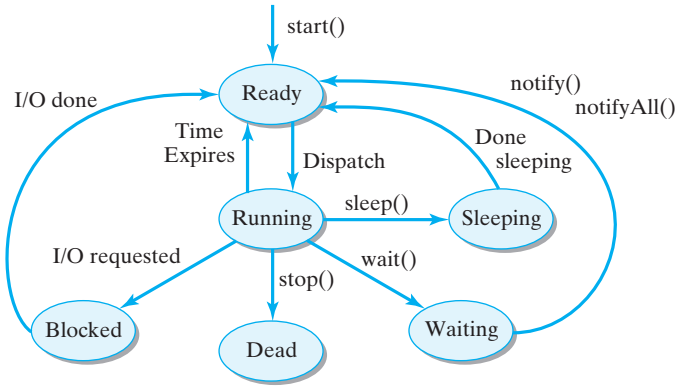

Figure 14.6: A depiction of a thread's life cycle.

TABLE 14.1 A summary of the different thread states.

State	Description
Ready	The thread is ready to run and waiting for the CPU.
Running	The thread is executing on the CPU.
Waiting	The thread is waiting for some event to happen.
Sleeping	The thread has been told to sleep for a time.
Blocked	The thread is waiting for I/O to finish.
Dead	The thread is terminated.

ready state, a thread is waiting, perhaps with other threads, in the **ready queue**, for its turn on the CPU. A **queue** is like a waiting line. When the CPU becomes available, the first thread in the ready queue will be **dispatched**—that is, it will be given the CPU. It will then be in the running state.

The ready queue

Transitions between the ready and running states happen under the control of the CPU scheduler, a fundamental part of the Java runtime system. The job of scheduling many threads in a fair and efficient manner is a little like sharing a single bicycle among several children. Children who are ready to ride the bike wait in line for their turn. The grown up (scheduler) lets the first child (thread) ride for a period of time before the bike is taken away and given to the next child in line. In round-robin scheduling, each child (thread) gets an equal amount of time on the bike (CPU).

CPU scheduler

When a thread calls the `sleep()` method, it voluntarily gives up the CPU, and when the sleep period is over, it goes back into the ready queue. This would be like one of the children deciding to rest for a moment during his or her turn. When the rest was over, the child would get back in line.

When a thread calls the `wait()` method, it voluntarily gives up the CPU, but this time it won't be ready to run again until it is notified by some other thread.

Threads can give up the CPU

This would be like one child giving his or her turn to another child. When the second child's turn is up, it would notify the first child, who would then get back in line.

The system also manages transitions between the **blocked** and ready states. A thread is put into a blocked state when it does some kind of I/O operation. I/O devices, such as disk drives, modems, and keyboards, are very slow compared to the CPU. Therefore, I/O operations are handled by separate processors known as *controllers*. For example, when a thread wants to read data from a disk drive, the system will give this task to the disk controller, telling it where to place the data. Because the thread can't do anything until the data are read, it is blocked, and another thread is allowed to run. When the disk controller completes the I/O operation, the blocked thread is unblocked and placed back in the ready queue.

Threads block on I/O operations

In terms of the bicycle analogy, blocking a thread would be like giving the bicycle to another child when the rider has to stop to tie his or her shoe. Instead of letting the bicycle just sit there, we let another child ride it. When the shoe is tied, the child is ready to ride again and goes back

into the ready line. Letting other threads run while one thread is waiting for an I/O operation to complete improves the overall utilization of the CPU.

SELF-STUDY EXERCISE

EXERCISE 14.5 Round-robin scheduling isn't always the best idea. Sometimes *priority scheduling* leads to a better system. Can you think of ways that priority scheduling—higher-priority threads go to the head of the line—can be used to improve the responsiveness of an interactive program?

14.5 Using Threads to Improve Interface Responsiveness

One good use for a *multithreaded* program is to help make a more responsive user interface. In a single-threaded program, a program that is executing statements in a long (perhaps even infinite) loop remains unresponsive to the user's actions until the loop is exited. Thus, the user will experience a noticeable and sometimes frustrating delay between the time an action is initiated and the time it is actually handled by the program.

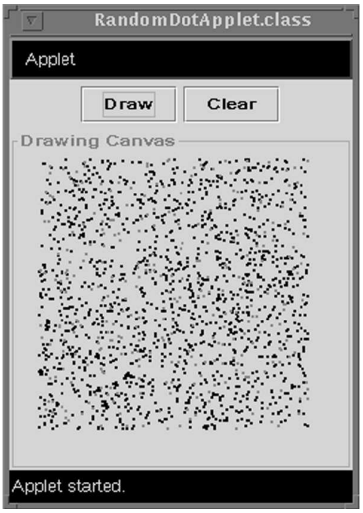

FIGURE 14.7 Random dots are drawn until the user clicks the Clear button.

14.5.1 Single-Threaded Design

It's always a good idea that the interface be responsive to user input, but sometimes it is crucial to an application. For example, suppose a psychology experiment is trying to measure how quickly a user responds to a certain stimulus presented by a program. Obviously, for this kind of application, the program should take action as soon as the user clicks a button to indicate a response to the stimulus. Let's work through an appropriate program design for the experiment. First, we will formally state the situation and describe what the program should do. Then, we will examine the components that would make up an effective program.

Problem Statement

Problem specification

A psychologist is conducting a psychometric experiment to measure user response to a visual cue and asks you to create the following program. The program should have two buttons. When the Draw button is clicked, the program begins drawing thousands of black dots at random locations within a rectangular region of the screen (Fig. 14.7). After a random time interval, the program begins drawing red dots. This change corresponds to the presentation of the stimulus. As soon as the stimulus is presented the user is supposed to click on a Clear button, which clears the drawing area. To provide a measure of the user's reaction time, the program should report how many red dots were drawn before the user clicked the Clear button.

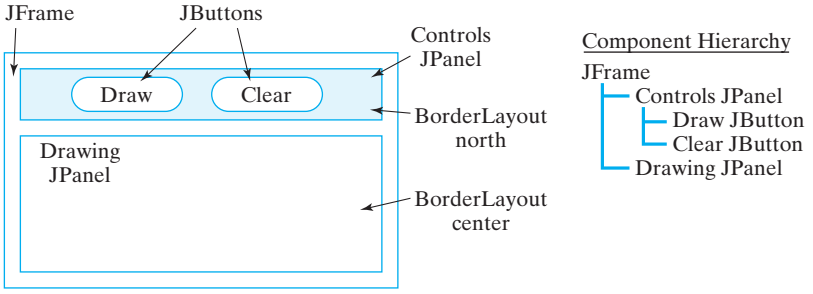

Figure 14.8: GUI design for the dot-drawing program.

Figure 14.8 shows a design for this program's GUI. It contains a control `JPanel` that contains the two `JButton`s. The dots are drawn on a `JPanel`, which is positioned in the center of a `BorderLayout` design.

GUI design

Problem Decomposition

This program should be decomposed into two classes, a GUI to handle the user interface and a drawing class to manage the drawing. The main features of its classes are as follows:

Interface class and drawing class

- `RandomDotGUI` Class: This class manages the user interface, responding to user actions by calling methods of the `Dotty` class (Fig. 14.9).
- `Dotty` Class: This class contains `draw()` and `clear()` methods for drawing on the GUI's drawing panel (Fig. 14.10).

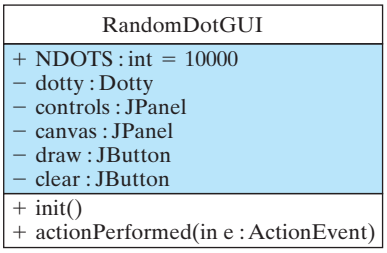

FIGURE 14.9 The `RandomDotGUI`.

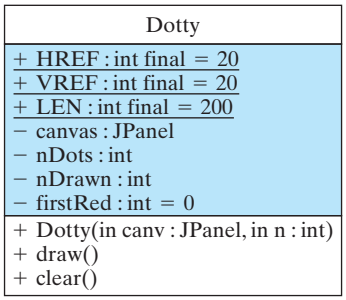

FIGURE 14.10 The `Dotty` class manages the drawing actions.

The `RandomDotGUI` Class

The implementation of `RandomDotGUI` is shown in Figure 14.11. The GUI arranges the control and drawing panels in a `BorderLayout` and listens for action events on its `JButton`s. When the user clicks the Draw button, the GUI's `actionPerformed()` method will create a new `Dotty` instance and call its `draw()` method:

```
dotty = new Dotty(canvas, NDOTS);
dotty.draw();
```

Note that `Dotty` is passed a reference to the drawing `canvas` as well as the number of dots to be drawn. When the user clicks the Clear button, the GUI should call the `dotty.clear()` method. Of course, the important question is, how responsive will the GUI be to the user's action?

The `Dotty` Class

The purpose of the `Dotty` class will be to draw the dots and to report how many red dots were drawn before the canvas was cleared. Because it will be passed a reference to the drawing panel and the number of dots to draw, the `Dotty` class will need instance variables to store these two values. It will also need a variable to keep track of how many dots were drawn. Finally, since it will be drawing within a fixed rectangle on the

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

public class RandomDotGUI extends JFrame
                          implements ActionListener  {
  public final int NDOTS = 10000;
  private Dotty dotty;        // The drawing class
  private JPanel controls = new JPanel();
  private JPanel canvas = new JPanel();
  private JButton draw = new JButton("Draw");
  private JButton clear = new JButton("Clear");

  public RandomDotGUI() {
    getContentPane().setLayout(new BorderLayout());
    draw.addActionListener(this);
    clear.addActionListener(this);
    controls.add(draw);
    controls.add(clear);
    canvas.setBorder(
      BorderFactory.createTitledBorder("Drawing Canvas"));
    getContentPane().add("North", controls);
    getContentPane().add("Center", canvas);
    getContentPane().setSize(400, 400);
  }
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == draw) {
      dotty = new Dotty(canvas, NDOTS);
      dotty.draw();
    } else {
      dotty.clear();
    }
  } // actionPerformed()
  public static void main(String args[]){
    RandomDotGUI gui = new RandomDotGUI();
    gui.setSize(400,400);
    gui.setVisible(true);
  }
} // RandomDotGUI
```

Figure 14.11: The `RandomDotGUI` class.

panel, the reference coordinates and dimensions of the drawing area are declared as class constants.

The `Dotty()` constructor method will be passed a reference to a drawing panel as well as the number of dots to be drawn and will merely assign these parameters to its instance variables. In addition to its constructor method, the `Dotty` class will have public `draw()` and `clear()` methods, which will be called from the GUI. The `draw()` method will use a loop to draw random dots. The `clear()` will clear the canvas and report the number of dots drawn.

```
import java.awt.*;
import javax.swing.*;    // Import Swing classes

public class Dotty {
                              // Coordinates
  private static final int HREF = 20, VREF = 20, LEN = 200;
  private JPanel canvas;
  private int nDots;          // Number of dots to draw
  private int nDrawn;         // Number of dots drawn
  private int firstRed = 0; // Number of the first red dot

  public Dotty(JPanel canv, int dots) {
    canvas = canv;
    nDots = dots;
  }
  public void draw() {
    Graphics g = canvas.getGraphics();
    for (nDrawn = 0; nDrawn < nDots; nDrawn++) {
      int x = HREF + (int)(Math.random() * LEN);
      int y = VREF + (int)(Math.random() * LEN);
      g.fillOval(x, y, 3, 3);              // Draw a dot

      if ((Math.random() < 0.001) && (firstRed == 0)) {
        g.setColor(Color.red); // Change color to red
        firstRed = nDrawn;
      }
    } //for
  } // draw()
  public void clear() {  // Clear screen and report result
    Graphics g = canvas.getGraphics();
    g.setColor(canvas.getBackground());
    g.fillRect(HREF, VREF, LEN + 3, LEN + 3);
    System.out.println(
     "Number of dots drawn since first red = " + (nDrawn-firstRed));
  } // clear()
} // Dotty
```

Figure 14.12: The `Dotty` class, single-threaded version.

The complete implementation of `Dotty` is shown in Figure 14.12. Note how its `draw()` method is designed. The drawing loop is bounded by the number of dots to be drawn. On each iteration, the `draw()` method picks a random location within the rectangle defined by the coordinates (HREF,VREF) and (HREF+LEN, VREF+LEN), and draws a dot there. On each iteration it also generates a random number. If the random number is less than 0.001, it changes the drawing color to red and keeps track of the number of dots drawn up to that point.

The problem with this design is that as long as the `draw()` method is executing, the program will be unable to respond to the GUI's Clear button. In a single-threaded design, both the GUI and `dotty` are combined into a single thread of execution (Fig. 14.13). When the user clicks the Draw button, the GUI's `actionPerformed()` method is invoked.

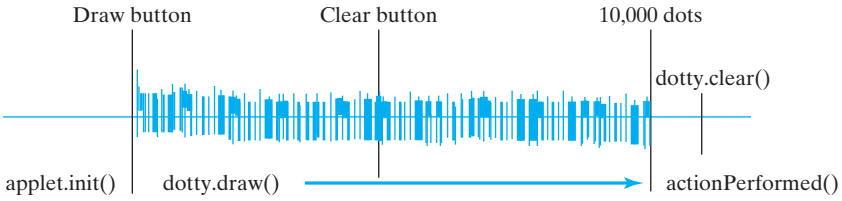

Figure 14.13: A single-threaded execution of random dot drawing.

It then invokes `Dotty`'s `draw()` method, which must run to completion before anything else can be done. If the user clicks the Clear button while the dots are being drawn, the GUI won't be able to get to this until all the dots are drawn.

If you run this program with `nDots` set to 10,000, the program will not clear the drawing panel until all 10,000 dots are drawn, no matter when the Clear button is pressed. Therefore, the values reported for the user's reaction time will be wrong. Obviously, since it is so unresponsive to user input, this design completely fails to satisfy the program's specifications.

JAVA LANGUAGE RULE Single-Threaded Loop. In a single-threaded design, a loop that requires lots of iterations will completely dominate the CPU during its execution, which forces other tasks, including user I/O tasks, to wait.

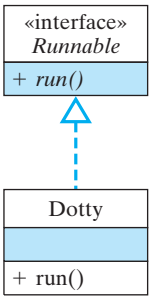

FIGURE 14.14 In a multithreaded design, the `Dotty` class implements `Runnable`.

SELF-STUDY EXERCISE

EXERCISE 14.6 Suppose the Java Virtual Machine (JVM) was single threaded and your program got stuck in an infinite loop. Would you be able to break out of the loop by typing some special command (such as Control-C) from the keyboard?

14.5.2 Multithreaded Drawing: The `Dotty` Thread

Multithreaded design: Interrupt the drawing loop

One way to remedy this problem is to create a second thread (in addition to the GUI itself) to do the drawing. The drawing thread will be responsible just for drawing, while the GUI thread will be responsible for handling user actions in the interface. The trick to making the user interface more responsive will be to interrupt the drawing thread periodically so that the GUI thread has a chance to handle any events that have occurred.

As Figure 14.14 illustrates, the easiest way to convert `Dotty` into a thread is to have it implement the `Runnable` interface:

```
public class Dotty implements Runnable {

    // Everything else remains the same

    public void run() {
        draw();
    }
}
```

This version of `Dotty` will perform the same task as before except that it will now run as a separate thread of execution. Note that its `run()` method just calls the `draw()` method that we defined in the previous version. When the `Dotty` thread is started by the `RandomDotGUI`, we will have a multithreaded program.

However, just because this program has two threads doesn't necessarily mean that it will be any more responsive to the user. There's no guarantee that the drawing thread will stop as soon as the Clear button is clicked. On most systems, if both threads have equal priority, the GUI thread won't run until the drawing thread finishes drawing all *N* dots.

Thread control

JAVA DEBUGGING TIP Thread Control. Just breaking a program into two separate threads won't necessarily give you the desired performance. It might be necessary to *coordinate* the threads.

Therefore, we have to modify our design in order to guarantee that the GUI thread will get a chance to handle the user's actions. One good way to do this is to have `Dotty` sleep for a short instance after it draws each dot. When a thread sleeps, any other threads that are waiting their turn will get a chance to run. If the GUI thread is waiting to handle the user's click on Clear, it will now be able to call `Dotty`'s `clear()` method.

Using `sleep()` *to interrupt the drawing*

The new version of `draw()` is shown in Figure 14.15. In this version of `draw()`, the thread sleeps for 1 millisecond on each iteration of the loop. This will make it possible for the GUI to run on every iteration, so it will handle user actions immediately.

Another necessary change is that once the `clear()` method is called, the `Dotty` thread should stop running (drawing). The correct way to stop a thread is to use some variable whose value will cause the run loop (or in this case the drawing loop) to exit, so the new version of `Dotty` uses the `boolean` variable `isCleared` to control when drawing is stopped. Note that the variable is initialized to `false` and then set to `true` in the `clear()` method. The for loop in `draw()` will exit when `isCleared` becomes `true`. This causes the `draw()` method to return, which causes the `run()` method to return, which causes the thread to stop in an orderly fashion.

JAVA EFFECTIVE DESIGN Threaded GUIs. Designing a multithreaded GUI involves creating a secondary thread that will run concurrently with the main GUI thread. The GUI thread handles the user interface, while the secondary thread performs CPU-intensive calculations.

JAVA PROGRAMMING TIP Threading an GUI. Creating a second thread within a GUI requires three steps: (1) Define the secondary thread to implement the `Runnable` interface, (2) override its `run()` method, and (3) incorporate some mechanism, such as a `sleep()` state, into the thread's run algorithm so that the GUI thread will have a chance to run periodically.

```
import java.awt.*;
import javax.swing.*;      // Import Swing classes

public class Dotty implements Runnable {
                                        // Coordinates
  private static final int HREF = 20, VREF = 20, LEN = 200;
  private JPanel canvas;
  private int nDots;              // Number of dots to draw
  private int nDrawn;             // Number of dots drawn
  private int firstRed = 0;       // Number of the first red dot
  private boolean isCleared = false; // Panel is cleared

  public void run() {
    draw();
  }
  public Dotty(JPanel canv, int dots) {
    canvas = canv;
    nDots = dots;
  }
  public void draw() {
    Graphics g = canvas.getGraphics();
    for (nDrawn = 0; !isCleared && nDrawn < nDots; nDrawn++) {
      int x = HREF + (int)(Math.random() * LEN);
      int y = VREF + (int)(Math.random() * LEN);
      g.fillOval(x, y, 3, 3);           // Draw a dot

      if (Math.random() < 0.001 && firstRed == 0) {
        g.setColor(Color.red);  // Change color to red
        firstRed = nDrawn;
      }
      try {
        Thread.sleep(1);          // Sleep for an instant
      } catch (InterruptedException e) {
        System.out.println(e.getMessage());
      }
    } //for
  } // draw()
  public void clear() {
    isCleared = true;
    Graphics g = canvas.getGraphics();
    g.setColor( canvas.getBackground() );
    g.fillRect(HREF,VREF,LEN+3,LEN+3);
    System.out.println("Number of dots drawn since first red = "
                                              + (nDrawn-firstRed));
  } // clear()
} // Dotty
```

Figure 14.15: By implementing the `Runnable` interface, this version of `Dotty` can run as a separate thread.

Modifications to RandomDotGUI

We don't need to make many changes in `RandomDotGUI` to get it to work with the new version of `Dotty`. The primary change comes in the `actionPerformed()` method. Each time the Draw button was clicked in the original version of this method, we created a `dotty` instance and then called its `draw()` method. In the revised version we must create a new `Thread` and pass it an instance of `Dotty`, which will then run as a separate thread:

Starting the drawing thread

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == draw) {
        dotty = new Dotty(canvas, NDOTS);
        dottyThread = new Thread(dotty);
        dottyThread.start();
    } else {
        dotty.clear();
    }
} // actionPerformed()
```

Note that in addition to a reference to `dotty` we also have a reference to a `Thread` named `dottyThread`. This additional variable must be declared within the GUI.

Remember that when you call the `start()` method, it automatically calls the thread's `run()` method. When `dottyThread` starts to run, it will immediately call the `draw()` method and start drawing dots. After each dot is drawn, `dottyThread` will sleep for an instant.

Notice how the GUI stops the drawing thread. In the new version, `Dotty.clear()` will set the `isCleared` variable, which will cause the drawing loop to terminate. Once again, this is the proper way to stop a thread. Thus, as soon as the user clicks the Clear button, the `Dotty` thread will stop drawing and report its result.

JAVA DEBUGGING TIP Stopping a Thread. The best way to stop a thread is to use a `boolean` control variable whose value can be set to true or false to exit the `run()` loop.

14.5.3 Advantages of Multithreaded Design

By creating a separate thread for `Dotty`, we have turned a single-threaded program into a multithreaded program. One thread, the GUI, handles the user interface. The second thread handles the drawing task. By forcing the drawing to sleep on each iteration, we guarantee that the GUI thread will remain responsive to the user's actions. Figure 14.16 illustrates the difference between the single- and multithreaded designs. Note that the GUI thread starts and stops the drawing thread, and the GUI thread executes `dotty.clear()`. The drawing thread simply executes its `draw()` method. In the single-threaded version, all of these actions are done by one thread.

Divide and conquer!

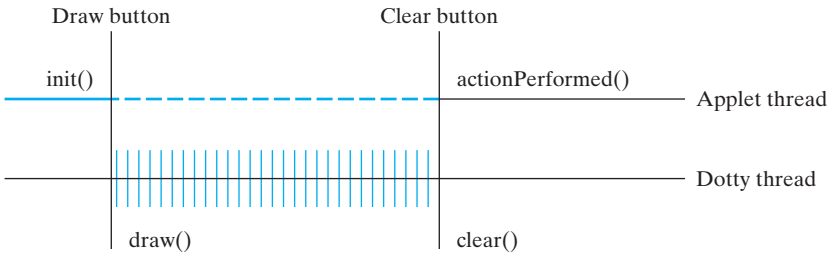

Figure 14.16: Two independent threads: one for drawing, the other for the GUI.

Trade-off: speed vs. responsiveness

The trade-off involved in this design is that it will take longer to draw *N* random dots, since `dottyThread.draw()` will sleep for an instant on each iteration. However, the extra time is hardly noticeable. By breaking the program into two separate threads of control, one to handle the drawing task and one to handle the user interface, the result is a much more responsive program.

JAVA EFFECTIVE DESIGN Responsive Interfaces. In order to give a program a more responsive user interface, divide it into separate threads of control. Let one thread handle interactive tasks, such as user input, and let the second thread handle CPU-intensive computations.

SELF-STUDY EXERCISES

EXERCISE 14.7 Someone might argue that because the Java Virtual Machine uses a round-robin scheduling algorithm, it's redundant to use the `sleep()` method, since the GUI thread will get its chance to run. What's wrong with this argument for interface responsiveness?

EXERCISE 14.8 Instead of sleeping on each iteration, another way to make the interface more responsive would be to set the threaded `Dotty`'s priority to a low number, such as 1. Make this change, and experiment with its effect on the program's responsiveness. Is it more or less responsive than sleeping on each iteration? Why?

14.6 CASE STUDY: Cooperating Threads

For some applications it is necessary to synchronize and coordinate the behavior of threads to enable them to carry out a cooperative task. Many cooperative applications are based on the **producer/consumer model**. According to this model, two threads cooperate at producing and consuming a particular resource or piece of data. The producer thread creates some message or result, and the consumer thread reads or uses the result. The consumer has to wait for a result to be produced, and the producer has to take care not to overwrite a result that hasn't yet been consumed. Many types of coordination problems fit the producer/consumer model.

Producer and consumer threads

One example of an application for this model would be to control the display of data that is read by your browser. As information arrives from the Internet, it is written to a buffer by the producer thread. A separate consumer thread reads information from the buffer and displays it

in your browser window. Obviously, the two threads must be carefully synchronized.

14.6.1 Problem Statement

Simulating a waiting line

To illustrate how to address the sorts of problems that can arise when you try to synchronize threads, let's consider a simple application in which several threads use a shared resource. You're familiar with those take-a-number devices that are used in bakeries to manage a waiting line. Customers take a number when they arrive, and the clerk announces who's next by looking at the device. As customers are called, the clerk increments the "next customer" counter by one.

There are some obvious potential coordination problems here. The device must keep proper count and can't skip customers. Nor can it give the same number to two different customers. Nor can it allow the clerk to serve nonexistent customers.

Our task is to build a multithreaded simulation that uses a model of a take-a-number device to coordinate the behavior of customers and a (single) clerk in a bakery waiting line. To help illustrate the various issues involved in trying to coordinate threads, we will develop more than one version of the program.

Problem Decomposition

What classes do we need?

This simulation will use four classes of objects. Figure 14.17 provides a UML representation of the interactions among the objects. The

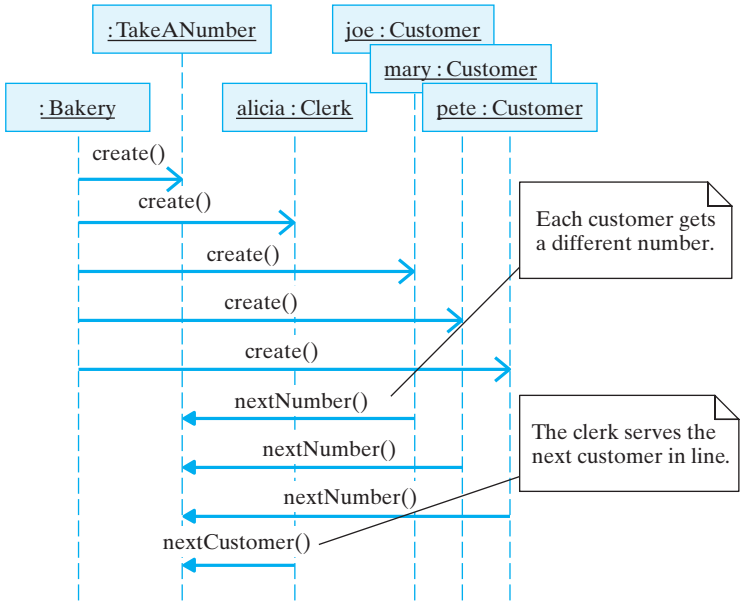

Figure 14.17: The `Bakery` creates the `Customer` and `Clerk` threads and the `TakeANumber` gadget. Then `Customer`s request and receive waiting numbers and the `Clerk` requests and receives the number of the next customer to serve.

`TakeANumber` object will serve as a model of a take-a-number device. This is the resource that will be shared by the threads, but it is not a thread itself. The `Customer` class, a subclass of `Thread`, will model the behavior of a customer who arrives on line and takes a number from the

`TakeANumber` device. There will be several `Customer` threads created that then compete for a space in line. The `Clerk` thread, which simulates the behavior of the store clerk, should use the `TakeANumber` device to determine who the next customer is and should serve that customer. Finally, there will be a main program that will have the task of creating and starting the various threads. Let's call this the `Bakery` class, which gives us the following list of classes:

- `Bakery`—creates the threads and starts the simulation.
- `TakeANumber`—represents the gadget that keeps track of the next customer to be served.
- `Clerk`—uses the `TakeANumber` to determine the next customer and will serve the customer.
- `Customer`—represents the customers who will use the `TakeANumber` to take their place in line.

14.6.2 Design: The `TakeANumber` Class

The `TakeANumber` class must track two things: Which customer will be served next, and which waiting number the next customer will be given. This suggests that it should have at least two public methods: `nextNumber()`, which will be used by customers to get their waiting numbers, and `nextCustomer()`, which will be used by the clerk to determine who should be served (Fig. 14.18). Each of these methods will simply retrieve the values of the instance variables, `next` and `serving`, which keep track of these two values. As part of the object's state, these variables should be `private`.

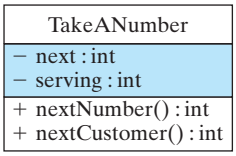

FIGURE 14.18 The `TakeANumber` object keeps track of numbers and customers.

Passing a reference to a shared object

How should we make this `TakeANumber` object accessible to all of the other objects—that is, to all of the customers and to the clerk? The easiest way to do that is to have the main program pass a reference to the `TakeANumber` when it constructs the `Customer`s and the `Clerk`. They can each store the reference as an instance variable. In this way, all the objects in the simulation can share a `TakeANumber` object as a common resource. Our design considerations lead to the definition of the `TakeANumber` class shown in Figure 14.19.

Synchronized methods

Note that the `nextNumber()` method is declared `synchronized`. As we will discuss in more detail, this ensures that only one customer at a time can take a number. Once a thread begins executing a `synchronized` method, no other thread can execute that method until the first thread finishes. This is important because, otherwise, several `Customer`s could call the `nextNumber` method at the same time. It's important that the customer threads have access only one at a time, also called *mutually exclusive access* to the `TakeANumber` object. This form of **mutual exclusion** is important for the correctness of the simulation.

SELF-STUDY EXERCISE

EXERCISE 14.9 What is the analogue to mutual exclusion in the real-world bakery situation?

```
class TakeANumber {
    private int next = 0;      // Next place in line
    private int serving = 0;   // Next customer to serve

    public synchronized int nextNumber() {
        next = next + 1;
        return next;
    } // nextNumber()

    public int nextCustomer() {
        ++serving;
        return serving;
    } // nextCustomer()

} // TakeANumber
```

Figure 14.19: Definition of the `TakeANumber` class, Version 1.

14.6.3 Java Monitors and Mutual Exclusion

The monitor concept

An object that contains `synchronized` methods has a **monitor** associated with it. A monitor is a widely used synchronization mechanism that ensures that only one thread at a time can execute a `synchronized` method. When a `synchronized` method is called, a **lock** is acquired on that object. For example, if one of the `Customer` threads calls `nextNumber()`, a lock will be placed on that `TakeANumber` object. While an object is *locked*, no other `synchronized` method can run in that object. Other threads must wait for the lock to be released before they can execute a `synchronized` method.

Mutually exclusive access to a shared object

While one `Customer` is executing `nextNumber()`, all other `Customers` will be forced to wait until the first `Customer` is finished. When the `synchronized` method is exited, the lock on the object is released, allowing other `Customer` threads to access their `synchronized` methods. In effect, a `synchronized` method can be used to guarantee mutually exclusive access to the `TakeANumber` object among the competing customers.

JAVA LANGUAGE RULE `synchronized`. Once a thread begins to execute a `synchronized` method in an object, the object is *locked* so that no other thread can gain access to that object's synchronized methods.

JAVA EFFECTIVE DESIGN Synchronization. In order to restrict access of a method or set of methods to one object at a time (mutual exclusion), declare the methods `synchronized`.

One cautionary note here is that although a synchronized method blocks access to other synchronized methods, it does not block access to nonsynchronized methods. This could cause problems. We will return to this

issue in the next part of our case study when we discuss the testing of our program.

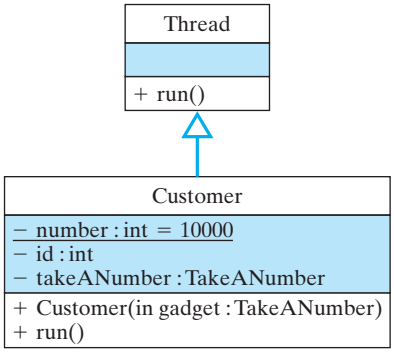

FIGURE 14.20 The `Customer` thread.

14.6.4 The `Customer` Class

A `Customer` thread should model the behavior of taking a number from the `TakeANumber` gadget. For the sake of this simulation, let's suppose that after taking a number, the `Customer` object just prints it out. This will serve as a simple model of "waiting on line." What about the `Customer`'s state? To help distinguish one customer from another, let's give each customer a unique ID number starting at 10001, which will be set in the constructor method. Also, as we noted earlier, each `Customer` needs a reference to the `TakeANumber` object, which is passed as a constructor parameter (Fig. 14.20). This leads to the definition of `Customer` shown in Figure 14.21. Note that before taking a number the customer sleeps for a random interval of up to 1,000 milliseconds. This will introduce a bit of randomness into the simulation.

```
public class Customer extends Thread {

  private static int number = 10000; // Initial ID number
  private int id;
  private TakeANumber takeANumber;

  public Customer( TakeANumber gadget ) {
    id = ++number;
    takeANumber = gadget;
  }

  public void run() {
    try {
      sleep( (int)(Math.random() * 1000 ) );
      System.out.println("Customer " + id +
        " takes ticket " + takeANumber.nextNumber());
    } catch (InterruptedException e) {
      System.out.println("Exception " + e.getMessage());
    }
  } // run()
} // Customer
```

Figure 14.21: Definition of the `Customer` class, Version 1.

Static (class) variables

Another important feature of this definition is the use of the `static` variable `number` to assign each customer a unique ID number. Remember that a `static` variable belongs to the class itself, not to its instances. Therefore, each `Customer` that is created can share this variable. By

incrementing it and assigning its new value as the `Customer`'s ID, we guarantee that each customer has a unique ID number.

JAVA LANGUAGE RULE Static (Class) Variables. Static variables are associated with the class itself and not with its instances.

JAVA EFFECTIVE DESIGN Unique IDs. Static variables are often used to assign a unique ID number or a unique initial value to each instance of a class.

14.6.5 The `Clerk` Class

The `Clerk` thread should simulate the behavior of serving the next customer in line, so the `Clerk` thread will repeatedly access `TakeANumber.nextCustomer()` and then serve that customer. For the sake of this simulation, we'll just print a message to indicate which customer is being served. Because there's only one clerk in this simulation, the only variable in its internal state will be a reference to the `TakeANumber` object (Fig. 14.22). In addition to the constructor, all we really need to define for this class is the `run()` method. This leads to the definition of `Clerk` shown in Figure 14.23. In this case, the `sleep()` method is necessary to allow the `Customer` threads to run. The `Clerk` will sit in an infinite loop serving the next customer on each iteration.

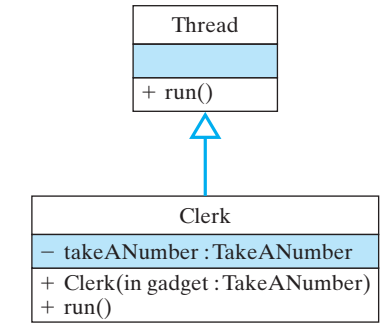

FIGURE 14.22 The `Clerk` thread.

```
public class Clerk extends Thread {
  private TakeANumber takeANumber;

  public Clerk(TakeANumber gadget) {
    takeANumber = gadget;
  }

  public void run() {
    while (true) {
      try {
        sleep( (int)(Math.random() * 50));
        System.out.println("Clerk serving ticket " +
                           takeANumber.nextCustomer());
      } catch (InterruptedException e) {
        System.out.println("Exception " + e.getMessage() );
      }
    } //while
  } //run()
} // Clerk
```

Figure 14.23: Definition of `Clerk`, Version 1.

14.6.6 The `Bakery` Class

The main program

Finally, `Bakery` is the simplest class to design. It contains the `main()` method, which gets the whole simulation started. As we said, its role will be to create one `Clerk` thread and several `Customer` threads, and get them all started (Fig. 14.24). Notice that the `Customer`s and the `Clerk` are each passed a reference to the shared `TakeANumber` gadget.

```
public class Bakery {
  public static void main(String args[]) {
    System.out.println("Starting clerk and customer threads");
    TakeANumber numberGadget = new TakeANumber();
    Clerk clerk = new Clerk(numberGadget);
    clerk.start();
    for (int k = 0; k < 5; k++) {
      Customer customer = new Customer(numberGadget);
      customer.start();
    }
  } // main()
} // Bakery
```

Figure 14.24: Definition of the `Bakery` class.

Problem: Nonexistent Customers

Testing and debugging

Now that we have designed and implemented the classes, let's run several experiments to test that everything works as intended. Except for the `synchronized nextNumber()` method, we've made little attempt to make sure that the `Customer` and `Clerk` threads will work together cooperatively, without violating the real-world constraints that should be satisfied by the simulation. If we run the simulation as it is presently coded, it will generate five customers and the clerk will serve all of them. But we get something like the following output:

```
Starting clerk and customer threads
  Clerk serving ticket 1
  Clerk serving ticket 2
  Clerk serving ticket 3
  Clerk serving ticket 4
  Clerk serving ticket 5
Customer 10004 takes ticket 1
Customer 10002 takes ticket 2
  Clerk serving ticket 6
Customer 10005 takes ticket 3
  Clerk serving ticket 7
  Clerk serving ticket 8
  Clerk serving ticket 9
  Clerk serving ticket 10
Customer 10001 takes ticket 4
Customer 10003 takes ticket 5
```

Problem: The clerk thread doesn't wait for customer threads

Our current solution violates an important real-world constraint: You can't serve customers before they enter the line! How can we ensure that the clerk doesn't serve a customer unless there's actually a customer waiting?

The wrong way to address this issue would be to increase the amount of sleeping that the `Clerk` does between serving customers. Indeed, this would allow more customer threads to run, so it might appear to have the desired effect, but it doesn't truly address the main problem: A clerk cannot serve a customer if no customer is waiting.

The clerk checks the line

The correct way to solve this problem is to have the clerk check that there are customers waiting before taking the next customer. One way to model this would be to add a `customerWaiting()` method to our `TakeANumber` object. This method would return `true` whenever `next` is greater than `serving`. That will correspond to the real-world situation in which the clerk can see customers waiting in line. We can make the following modification to `Clerk.run()`:

```
public void run() {
  while (true) {
    try {
      sleep((int)(Math.random() * 50));
      if (takeANumber.customerWaiting())
        System.out.println("Clerk serving ticket "
                    + takeANumber.nextCustomer());
    } catch (InterruptedException e) {
      System.out.println("Exception " + e.getMessage());
    }
  } // while
} // run()
```

And we add the following method to `TakeANumber` (Fig. 14.25):

```
public boolean customerWaiting() {
    return next > serving;
}
```

In other words, the `Clerk` won't serve a customer unless there are customers waiting—that is, unless `next` is greater than `serving`. Given

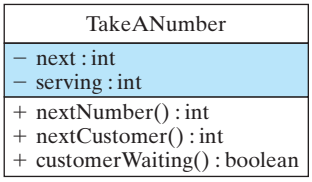

FIGURE 14.25 The revised `TakeANumber` class.

these changes, we get the following type of output when we run the simulation:

```
Starting clerk and customer threads
Customer 10003 takes ticket 1
  Clerk serving ticket 1
Customer 10005 takes ticket 2
  Clerk serving ticket 2
Customer 10001 takes ticket 3
  Clerk serving ticket 3
Customer 10004 takes ticket 4
  Clerk serving ticket 4
Customer 10002 takes ticket 5
  Clerk serving ticket 5
```

This example illustrates that when application design involves cooperating threads, the algorithm used must ensure the proper cooperation and coordination among the threads.

JAVA EFFECTIVE DESIGN Thread Coordination. When two or more threads must behave cooperatively, their interaction must be carefully coordinated by the algorithm.

14.6.7 Problem: Critical Sections

Thread interruptions are unpredictable

It is easy to forget that thread behavior is asynchronous. You can't predict when a thread might be interrupted or might have to give up the CPU to another thread. In designing applications that involve cooperating threads, it's important that the design incorporates features to guard against problems caused by asynchronicity. To illustrate this problem, consider the following statement from the `Customer.run()` method:

```
System.out.println("Customer " + id +
        " takes ticket " + takeANumber.nextNumber());
```

Even though this is a single Java statement, it breaks up into several Java bytecode statements. A `Customer` thread could certainly be interrupted between getting the next number back from `TakeANumber` and printing it

out. We can simulate this by breaking the `println()` into two statements and putting a `sleep()` in their midst:

```
public void run() {
  try {
    int myturn = takeANumber.nextNumber();
    sleep( (int)(Math.random() * 1000 ) );
    System.out.println("Customer " + id +
                       " takes ticket " + myturn);
  } catch (InterruptedException e) {
    System.out.println("Exception " + e.getMessage());
  }
} // run()
```

If this change is made in the simulation, you might get the following output:

```
Starting clerk and customer threads
  Clerk serving ticket 1
  Clerk serving ticket 2
  Clerk serving ticket 3
Customer 10004 takes ticket 4
  Clerk serving ticket 4
  Clerk serving ticket 5
Customer 10001 takes ticket 1
Customer 10002 takes ticket 2
Customer 10003 takes ticket 3
Customer 10005 takes ticket 5
```

Because the `Customer` threads are now interrupted in between taking a number and reporting their number, it looks as if they are being served in the wrong order. Actually, they are being served in the correct order. It's their reporting of their numbers that is wrong!

Problem: An interrupt in a critical section

The problem here is that the `Customer.run()` method is being interrupted in such a way that it invalidates the simulation's output. A method that displays the simulation's state should be designed so that once a thread begins reporting its state, that thread will be allowed to finish reporting before another thread can start reporting its state. Accurate reporting of a thread's state is a critical element of the simulation's overall integrity.

A **critical section** is any section of a thread that should not be interrupted during its execution. In the bakery simulation, all of the statements that report the simulation's progress are critical sections. Even though the chances are small that a thread will be interrupted in the midst of a `println()` statement, the faithful reporting of the simulation's state should not be left to chance. Therefore, we must design an algorithm that prevents the interruption of critical sections.

Creating a Critical Section

The correct way to address this problem is to treat the reporting of the customer's state as a critical section. As we saw earlier when we discussed

Making a critical section uninterruptible

the concept of a monitor, a `synchronized` method within a shared object ensures that once a thread starts the method, it will be allowed to finish it before any other thread can start it. Therefore, one way out of this dilemma is to redesign the `nextNumber()` and `nextCustomer()` methods in the `TakeANumber` class so that they report which customer receives a ticket and which customer is being served (Fig. 14.26). In this version all of the methods are `synchronized`, so all the actions of the `TakeANumber` object are treated as critical sections.

```
public class TakeANumber {
  private int next = 0;      // Next place in line
  private int serving = 0;   // Next customer to serve

  public synchronized int nextNumber(int custId) {
    next = next + 1;
    System.out.println( "Customer " + custId + "
                                  takes ticket " + next );
    return next;
  } // nextNumber()
  public synchronized int nextCustomer() {
    ++serving;
    System.out.println("  Clerk serving ticket "
                                              + serving );
    return serving;
  } // nextCustomer()
  public synchronized boolean customerWaiting() {
    return next > serving;
  } // customerWaiting()
} // TakeANumber
```

Figure 14.26: Definition of the `TakeANumber` class, Version 2.

Note that the reporting of both the next number and the next customer to be served are now handled by `TakeANumber` in Figure 14.26 . Because the methods that handle these actions are `synchronized`, they cannot be interrupted by any threads involved in the simulation. This guarantees that the simulation's output will faithfully report the simulation's state.

Given these changes to `TakeANumber`, we must remove the `println()` statements from the `run()` methods in `Customer`:

```
public void run() {
  try {
    sleep((int)(Math.random() * 2000));
    takeANumber.nextNumber(id);
  } catch (InterruptedException e) {
    System.out.println("Exception: "+ e.getMessage());
  }
} // run()
```

and from the `run()` method in `Clerk`:

```
public void run() {
  while (true) {
    try {
      sleep( (int)(Math.random() * 1000));
      if (takeANumber.customerWaiting())
        takeANumber.nextCustomer();
    } catch (InterruptedException e) {
      System.out.println("Exception: "+e.getMessage());
    }
  } // while
} // run()
```

Rather than printing their numbers, these methods now just call the appropriate methods in `TakeANumber`. Given these design changes, our simulation now produces the following correct output:

```
Starting clerk and customer threads
Customer 10001 takes ticket 1
  Clerk serving ticket 1
Customer 10003 takes ticket 2
Customer 10002 takes ticket 3
  Clerk serving ticket 2
Customer 10005 takes ticket 4
Customer 10004 takes ticket 5
  Clerk serving ticket 3
  Clerk serving ticket 4
  Clerk serving ticket 5
```

Preventing undesirable interrupts

The lesson to be learned from this is that in designing multithreaded programs, it is important to assume that if a thread can be interrupted at a certain point, it will be interrupted at that point. The fact that an interrupt

is unlikely to occur is no substitute for the use of a critical section. This is something like "Murphy's Law of Thread Coordination."

JAVA EFFECTIVE DESIGN The Thread Coordination Principle. Use critical sections to coordinate the behavior of cooperating threads. By designating certain methods as synchronized, you can ensure their mutually exclusive access. Once a thread starts a synchronized method, no other thread will be able to execute the method until the first thread is finished.

In a multithreaded application, the classes and methods should be designed so that undesirable interrupts will not affect the correctness of the algorithm.

JAVA PROGRAMMING TIP Critical Sections. Java's monitor mechanism will ensure that while one thread is executing a synchronized method, no other thread can gain access to it. Even if the first thread is interrupted, when it resumes execution again it will be allowed to finish the synchronized method before other threads can access synchronized methods in that object.

SELF-STUDY EXERCISE

EXERCISE 14.10 Given the changes we've described, the bakery simulation should now run correctly regardless of how slow or fast the `Customer` and `Clerk` threads run. Verify this by placing different-sized sleep intervals in their `run()` methods. (*Note*: You don't want to put a `sleep()` in the synchronized methods because that would undermine the whole purpose of making them `synchronized` in the first place.)

14.6.8 Using `wait/notify` to Coordinate Threads

The examples in the previous sections were designed to illustrate the issue of thread asynchronicity and the principles of mutual exclusion and critical sections. Through the careful design of the algorithm and the appropriate use of the `synchronized` qualifier, we have managed to design a program that correctly coordinates the behavior of the `Customer`s and `Clerk` in this bakery simulation.

The Busy-Waiting Problem

Busy waiting

One problem with our current design of the Bakery algorithm is that it uses *busy waiting* on the part of the `Clerk` thread. **Busy waiting** occurs when a thread, while waiting for some condition to change, executes a loop instead of giving up the CPU. Because busy waiting is wasteful of CPU time, we should modify the algorithm.

As it is presently designed, the `Clerk` thread sits in a loop that repeatedly checks whether there's a customer to serve:

```
public void run() {
  while (true) {
    try {
      sleep( (int)(Math.random() * 1000));
      if (takeANumber.customerWaiting())
        takeANumber.nextCustomer();
    } catch (InterruptedException e) {
      System.out.println("Exception: " + e.getMessage());
    }
  } // while
} // run()
```

A far better solution would be to force the `Clerk` thread to wait until a customer arrives without using the CPU. Under such a design, the `Clerk` thread can be notified and enabled to run as soon as a `Customer` becomes available. Note that this description views the customer/clerk relationship as one-half of the producer/consumer relationship. When a customer takes a number, it *produces* a customer in line that must be served (that is, *consumed*) by the clerk.

Producer/consumer

This is only half the producer/consumer relationship because we haven't placed any constraint on the size of the waiting line. There's no real limit to how many customers can be produced. If we did limit the line size, customers might be forced to wait before taking a number if, say, the tickets ran out, or the bakery filled up. In that case, customers would have to wait until the line resource became available and we would have a full-fledged producer/consumer relationship.

The `wait/notify` Mechanism

So, let's use Java's `wait/notify` mechanism to eliminate busy waiting from our simulation. As noted in Figure 14.6, the `wait()` method puts a thread into a waiting state, and `notify()` takes a thread out of waiting and places it back in the ready queue. To use these methods in this program we need to modify the `nextNumber()` and `nextCustomer()`

methods. If there is no customer in line when the `Clerk` calls the `nextCustomer()` method, the `Clerk` should be made to `wait()`:

```
public synchronized int nextCustomer() {
  try {
    while (next <= serving)
      wait();
  } catch(InterruptedException e) {
    System.out.println("Exception: " + e.getMessage());
  } finally {
    ++serving;
    System.out.println(" Clerk serving ticket " + serving);
    return serving;
  }
}
```

A waiting thread gives up the CPU

Note that the `Clerk` still checks whether there are customers waiting. If there are none, the `Clerk` calls the `wait()` method. This removes the `Clerk` from the CPU until some other thread notifies it, at which point it will be ready to run again. When it runs again, it should check that there is in fact a customer waiting before proceeding. That's why we use a while loop here. In effect, the `Clerk` will wait until there's a customer to serve. This is not busy waiting because the `Clerk` thread loses the CPU and must be notified each time a customer becomes available.

When and how will the `Clerk` be notified? Clearly, the `Clerk` should be notified as soon as a customer takes a number. Therefore, we put a `notify()` in the `nextNumber()` method, which is the method called by each `Customer` as it gets in line:

```
public synchronized int nextNumber( int custId) {
  next = next + 1;
  System.out.println("Customer " + custId +
                     " takes ticket " + next);
  notify();
  return next;
}
```

Thus, as soon as a `Customer` thread executes the `nextNumber()` method, the `Clerk` will be notified and allowed to proceed.

What happens if more than one `Customer` has executed a `wait()`? In that case, the JVM will maintain a queue of waiting `Customer` threads. Then, each time a `notify()` is executed, the JVM will take the first `Customer` out of the queue and allow it to proceed.

If we use this model of thread coordination, we no longer need to test `customerWaiting()` in the `Clerk.run()` method. It is to be tested in

the `TakeANumber.nextCustomer()`. Thus, the `Clerk.run()` can be simplified to

```
public void run() {
  while (true) {
    try {
      sleep((int)(Math.random() * 1000));
      takeANumber.nextCustomer();
    } catch (InterruptedException e) {
      System.out.println("Exception: "+ e.getMessage());
    }
  } // while
} // run()
```

The `Clerk` thread may be forced to wait when it calls the `nextCustomer` method.

Because we no longer need the `customerWaiting()` method, we end up with the new definition of `TakeANumber` shown in Figures 14.27 and 14.28.

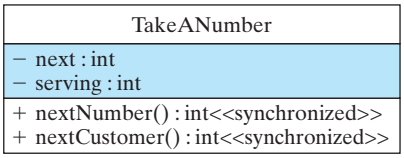

FIGURE 14.27 In the final design of `TakeANumber`, its methods are synchronized.

```
public class TakeANumber {
  private int next = 0;
  private int serving = 0;

  public synchronized int nextNumber(int custId) {
    next = next + 1;
    System.out.println( "Customer " + custId +
                        " takes ticket " + next);
    notify();
    return next;
  } // nextNumber()

  public synchronized int nextCustomer() {
    try {
      while (next <= serving)  {
        System.out.println("  Clerk waiting ");
        wait();
      }
    } catch(InterruptedException e) {
       System.out.println("Exception " + e.getMessage());
    } finally {
       ++serving;
       System.out.println("  Clerk serving ticket "
                                          + serving);
       return serving;
    }
  } // nextCustomer()
} // TakeANumber
```

Figure 14.28: The `TakeANumber` class, Version 3.

Given this version of the program, the following kind of output will be generated:

```
Starting clerk and customer threads
Customer 10004 takes ticket 1
Customer 10002 takes ticket 2
  Clerk serving ticket 1
  Clerk serving ticket 2
Customer 10005 takes ticket 3
Customer 10003 takes ticket 4
  Clerk serving ticket 3
Customer 10001 takes ticket 5
  Clerk serving ticket 4
  Clerk serving ticket 5
  Clerk waiting
```

JAVA PROGRAMMING TIP Busy Waiting. Java's `wait/notify` mechanism can be used effectively to eliminate busy waiting from a multithreaded application.

JAVA EFFECTIVE DESIGN Producer/Consumer. The producer/consumer model is a useful design for coordinating the wait/notify interaction.

SELF-STUDY EXERCISE

EXERCISE 14.11 An interesting experiment to try is to make the `Clerk` a little slower by making it sleep for up to 2,000 milliseconds. Take a guess at what would happen if you ran this experiment. Then run the experiment and observe the results.

The `wait/notify` Mechanism

Wait/notify go into synchronized methods

There are a number of important restrictions that must be observed when using the `wait/notify` mechanism:

- Both `wait()` and `notify()` are methods of the `Object` class, not the `Thread` class. This enables them to lock objects, which is the essential feature of Java's monitor mechanism.
- A `wait()` method can be used within a `synchronized` method. The method doesn't have to be part of a `Thread`.
- You can only use `wait()` and `notify()` within `synchronized` methods. If you use them in other methods, you will cause an `IllegalMonitorStateException` with the message "current thread not owner."
- When a `wait()`—or a `sleep()`—is used within a `synchronized` method, the lock on that object is released so that other methods can access the object's `synchronized` methods.

JAVA DEBUGGING TIP **Wait/Notify.** It's easy to forget that the `wait()` and `notify()` methods can only be used within `synchronized` methods.

14.7 CASE STUDY: The Game of Pong

The game of Pong was one of the first computer video games and was all the rage in the 1970s. The game consists of a ball that moves horizontally and vertically within a rectangular region, and a single paddle, which is located at the right edge of the region that can be moved up and down by the user. When the ball hits the top, left, or bottom walls or the paddle, it bounces off in the opposite direction. If the ball misses the paddle, it passes through the right wall and re-emerges at the left wall. Each time the ball bounces off a wall or paddle, it emits a pong sound.

14.7.1 A Multithreaded Design

Let's develop a multithreaded applet to play the game of Pong. Figure 14.29 shows how the game's GUI should appear. There are three objects involved in this program: the applet, which serves as the GUI, the ball, which is represented as a blue circle in the applet, and the paddle, which is represented by a red rectangle along the right edge of the applet. What cannot be seen in this figure is that the ball moves autonomously, bouncing off the walls and paddle. The paddle's motion is controlled by the user by pressing the up- and down-arrow keys on the keyboard.

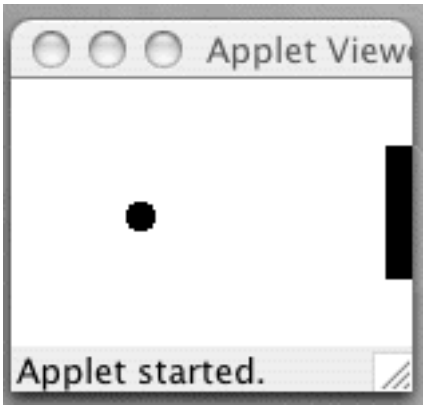

FIGURE 14.29 The UI for Pong.

We will develop class definitions for the ball, paddle, and the applet. Following the example of our dot-drawing program earlier in the Chapter, we will employ two independent threads, one for the GUI and one for the ball. Because the user will control the movements of the paddle, the applet will employ a listener object to listen for and respond to the user's key presses.

Figure 14.30 provides an overview of the object-oriented design of the Pong program. The `PongApplet` class is the main class. It uses instances of the `Ball` and `Paddle` classes. `PongApplet` is a subclass of `JApplet` and implements the `KeyListener` interface. This is another of the several event handlers provided in the `java.awt` library. This one handles `KeyEvents` and the `KeyListener` interface consists of three abstract methods: `keyPressed()`, `keyTyped()`, and `keyReleased()`, all of which are associated with the act of pressing a key on the keyboard. All three of these methods are implemented in the `PongApplet` class. A key-typed event occurs when a key is pressed down. A key-release event occurs when a key that has been pressed down is released. A key-press event is a combination of both of these events.

The `Ball` class is a `Thread` subclass. Its data and methods are designed mainly to keep track of its motion within the applet's drawing panel. The design strategy employed here leaves the drawing of the ball up to the applet. The `Ball` thread itself just handles the movement within the applet's drawing panel. Note that the `Ball()` constructor takes a reference to the `PongApplet`. As we will see, the `Ball` uses this reference

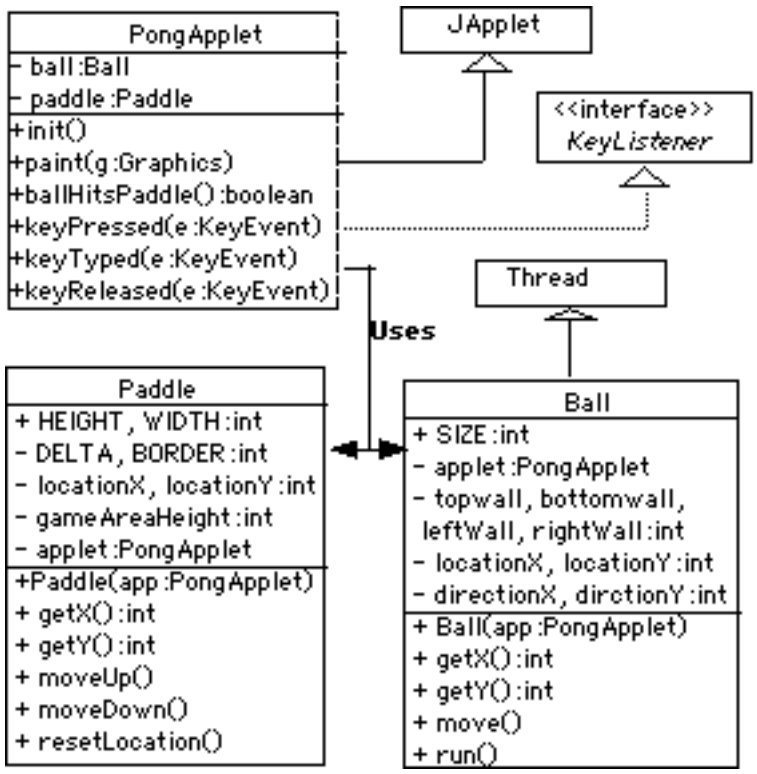

Figure 14.30: Design of the Pong program.

to set the dimensions of the applet's drawing panel. Also, as the `Ball` moves, it will repeatedly call the applet's `repaint()` method to draw the ball.

The `Paddle` class is responsible for moving the paddle up and down along the drawing panel's right edge. Its public methods, `moveUP()` and `moveDown()`, will be called by the applet in response to the user pressing the up and down arrows on the keyboard. Because the applet needs to know where to draw the applet, the paddle class contains several public methods, `getX()`, `getY()`, and `resetLocation()`, whose tasks are to report the paddle's location or to adjust its location in case the applet is resized.

The `PongApplet` controls the overall activity of the program. Note in particular its `ballHitsPaddle()` method. This method has the task of determining when the ball and paddle come in contact as the ball continuously moves around in the applet's drawing panel. As in the `ThreadedDotty` example earlier in the chapter, it is necessary for the `Ball` and the the applet to be implemented as separated threads so that the applet can be responsive to the user's key presses.

14.7.2 Implementation of the Pong Program

We begin our discussion of the program's implementation with the `Paddle` class implementation (Fig. 14.31).

```
public class Paddle {
    public static final int HEIGHT = 50;  // Paddle size
    public static final int WIDTH = 10;
    private static final int DELTA = HEIGHT/2; // Move size
    private static final int BORDER = 0;
    private int gameAreaHeight;
    private int locationX, locationY;
    private PongApplet applet;

    public Paddle (PongApplet a) {
       applet = a;
       gameAreaHeight  = a.getHeight();
       locationX = a.getWidth()-WIDTH;
       locationY = gameAreaHeight/2;
    } // Paddle()
    public void resetLocation() {
       gameAreaHeight  = applet.getHeight();
       locationX = applet.getWidth()-WIDTH;
    }
    public int getX() {
         return locationX;
    }
    public int getY() {
         return locationY;
    }
    public void moveUp () {
       if (locationY > BORDER )
           locationY -= DELTA;
    } // moveUp()
    public void moveDown() {
       if (locationY + HEIGHT < gameAreaHeight - BORDER)
           locationY += DELTA;
    } // moveDown()
} // Paddle
```

Figure 14.31: Definition of the `Paddle` class.

Class constants, `HEIGHT` and `WIDTH` are used to define the size of the `Paddle`, which is represented on the applet as a simple rectangle. The applet will use the `Graphics.fillRect()` method to draw the paddle:

```
g.fillRect(pad.getX(),pad.getY(),Paddle.WIDTH,Paddle.HEIGHT);
```

Note how the applet uses the paddle's `getX()` and `getY()` methods to get the paddle's current location.

The class constants `DELTA` and `BORDER` are used to control the paddle's movement. `DELTA` represents the number of pixels that the paddle moves on each move up or down, and `BORDER` is used with `gameAreaHeight` to keep the paddle within the drawing area. The `moveUp()` and `moveDown()` methods are called by the applet each time the user presses an up- or down-arrow key. They simply change the paddle's location by `DELTA` pixels up or down.

The `Ball` class (Fig. 14.32) uses the class constant `SIZE` to determine the size of the oval that represents the ball, drawn by the applet as follows:

```
g.fillOval(ball.getX(),ball.getY(),ball.SIZE,ball.SIZE);
```

As with the paddle, the applet uses the ball's `getX()` and `getY()` method to determine the ball's current location.

Unlike the paddle, however, the ball moves autonomously. Its `run()` method, which is inherited from its `Thread` superclass, repeatedly moves the ball, draws the ball, and then sleeps for a brief interval (to slow down the speed of the ball's apparent motion). The `run()` method itself is quite simple because it consists of a short loop. We will deal with the details of how the ball is painted on the applet when we discuss the applet itself.

The most complex method in the `Ball` class is the `move()` method. This is the method that controls the ball's movement within the boundaries of the applet's drawing area. This method begins by moving the ball by one pixel left, right, up, or down by adjusting the values of its `locationX` and `locationY` coordinates:

```
locationX = locationX + directionX; // Calculate location
locationY = locationY + directionY;
```

The `directionX` and `directionY` variables are set to either $+1$ or -1, depending on whether the ball is moving left or right, up or down. After the ball is moved, the method uses a sequence of `if` statements to check whether the ball is touching one of the walls or the paddle. If the ball is in contact with the top, left, or bottom walls or the paddle, its direction is changed by reversing the value of the `directionX` or `directionY` variable. The direction changes depend on whether the ball has touched a horizontal or vertical wall. When the ball touches the right wall, having missed the paddle, it passes through the right wall and re-emerges from the left wall going in the same direction.

Note how the applet method, `ballHitsPaddle()` is used to determine whether the ball has hit the paddle. This is necessary because only the applet knows the locations of both the ball and the paddle.

14.7.3 The `KeyListener` Interface

The implementation of the `PongApplet` class is shown in figure 14.33. The applet's main task is to manage the drawing of the ball and paddle and to handle the user's key presses. Handling keyboard events is a simple matter of implementing the `KeyListener` interface. This works in much the same way as the `ActionListener` interface, which is used to handle button clicks and other `ActionEvents`. Whenever a key is pressed, it generates `KeyEvents`, which are passed to the appropriate methods of the `KeyListener` interface.

There's a bit of redundancy in the `KeyListener` interface in the sense that a single key press and release generates three `KeyEvents`: A key-typed event, when the key is pressed, a key-released event, when the key is released, and a key-pressed event, when the key is pressed and released.

```
import javax.swing.*;
import java.awt.Toolkit;

public class Ball extends Thread {
  public static final int SIZE = 10;    // Diameter of the ball
  private PongApplet applet;            // Reference to the applet
  private int topWall, bottomWall, leftWall, rightWall;  // Boundaries
  private int locationX, locationY;     // Current location of the ball
  private int directionX = 1, directionY = 1; //x- and y-direction (1 or -1)
  private Toolkit kit = Toolkit.getDefaultToolkit(); //For beep() method

  public Ball(PongApplet app) {
    applet = app;
    locationX = leftWall + 1;                  // Set initial location
    locationY = bottomWall/2;
  } // Ball()
  public int getX() {
    return locationX;
  } // getX()
  public int getY() {
    return locationY;
  } // getY()
  public void move() {
    rightWall = applet.getWidth() - SIZE;    // Define bouncing region
    leftWall = topWall = 0;                  // And location of walls
    bottomWall = applet.getHeight() - SIZE;
    locationX = locationX + directionX; // Calculate a new location
    locationY = locationY + directionY;

    if (applet.ballHitsPaddle()){
      directionX = -1;   //  move toward left wall
      kit.beep();
    } //if ball hits paddle
    if (locationX <= leftWall){
      directionX = + 1;   //  move toward right wall
      kit.beep();
    } //if ball hits left wall
    if (locationY + SIZE >= bottomWall || locationY <= topWall){
      directionY = -directionY;       //  reverse direction
      kit.beep();
    } //if ball hits top or bottom walls
    if (locationX  >= rightWall + SIZE) {
      locationX = leftWall + 1;   // jump back to left wall
    } //if ball goes through right wall
  } // move()
  public void run() {
    while (true) {
      move();                  // Move
      applet.repaint();
      try {  sleep(15);
      } catch (InterruptedException e) {}
    } // while
  } // run()
} // Ball
```

Figure 14.32: Definition of the `Ball` class.

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

public class PongApplet extends JApplet implements KeyListener {
     private Ball ball;
     private Paddle pad;

     public void init() {
         setBackground(Color.white);
         addKeyListener(this);
         pad = new Paddle(this);  // Create the paddle
         ball = new Ball(this);       // Create the ball
         ball.start();
         requestFocus();  // Required to receive key events
     } // init()

     public void paint (Graphics g ) {
          g.setColor(getBackground());    // Erase the drawing area
          g.fillRect(0,0,getWidth(),getHeight());

          g.setColor(Color.blue);         // Paint the ball
          g.fillOval(ball.getX(),ball.getY(),ball.SIZE,ball.SIZE);

          pad.resetLocation();            // Paint the paddle
          g.setColor(Color.red);
          g.fillRect(pad.getX(),pad.getY(),Paddle.WIDTH,Paddle.HEIGHT);
     } // paint()

      public boolean ballHitsPaddle() {
          return ball.getX() + Ball.SIZE >= pad.getX()
             && ball.getY() >= pad.getY()
             && ball.getY() <= pad.getY() + Paddle.HEIGHT;
     } // ballHitsPaddle()

     public void keyPressed( KeyEvent e) { // Check for arrow keys
         int keyCode = e.getKeyCode();
         if (keyCode == e.VK_UP)             // Up arrow
             pad.moveUp();
         else if (keyCode == e.VK_DOWN)      // Down arrow
             pad.moveDown();
     } // keyReleased()
     public void keyTyped(KeyEvent e) {}      // Unused
     public void keyReleased( KeyEvent e) {} // Unused
} // PongApplet
```

Figure 14.33: Definition of the `PongApplet` class.

While it is important for some programs to be able to distinguish between a key-typed and key-released event, for this program, we will take action whenever one of the arrow keys is pressed (typed and released). Therefore, we implement the `keyPressed()` method as follows:

```
public void keyPressed( KeyEvent e) { // Check arrow keys
   int keyCode = e.getKeyCode();
   if (keyCode == e.VK_UP)              // Up arrow
      pad.moveUp();
   else if (keyCode == e.VK_DOWN)       // Down arrow
      pad.moveDown();
} // keyReleased()
```

Each key on the keyboard has a unique code that identifies the key. The key's code is gotten from the `KeyEvent` object by means of the `getKeyCode()` method. Then it is compared with the codes for the up-arrow and down-arrow keys, which are implemented as class constants, `VK_UP` and `VK_DOWN`, in the `KeyEvent` class. If either of those keys were typed, the appropriate paddle method, `moveUP()` or `moveDown()`, is called.

Note that even though we are not using the `keyPressed()` and `keyReleased()` methods in this program, it is still necessary to provide implementations for these methods in the applet. In order to implement an interface, such as the `KeyListener` interface, you must implement *all* the abstract methods in the interface. That is why we provide trivial implementations of both the `keyPressed()` and `keyReleased()` methods.

14.7.4 Animating the Bouncing Ball

Computer animation is accomplished by repeatedly drawing, erasing, and re-drawing an object at different locations on the drawing panel. The applet's `paint()` method is used for drawing the ball and the paddle at their current locations. The `paint()` method is never called directly. Rather, it is called automatically after the `init()` method, when the applet is started. It is then invoked indirectly by the program by calling the `repaint()` method, which is called in the `run()` method of the `Ball` class. The reason that `paint()` is called indirectly is because Java needs to pass it the applet's current `Graphics` object. Recall that in Java all drawing is done using a `Graphics` object.

In order to animate the bouncing ball, we first erase the current image of the ball, then we draw the ball in its new location. We also draw the paddle in its current location. These steps are carried out in the applet's `paint()` method. First, the drawing area is cleared by painting its rectangle in the background color. Then the ball and paddle are painted at their current locations. Note that before painting the paddle, we first call its `resetLocation()` method. This causes the paddle to be relocated in case the user has resized the applet's drawing area. There is no need to do this for the ball because the ball's drawing area is updated within the `Ball.move()` method every time the ball is moved.

Double buffering

One problem with computer animations of this sort is that the repeated drawing and erasing of the drawing area can cause the screen to flicker. In some drawing environments a technique known as **double buffering** is used to reduce the flicker. In double buffering, an invisible, off-screen, buffer is used for the actual drawing operations and it is then used to replace the visible image all at once when the drawing is done. Fortunately, Java's Swing components, including `JApplet` and `JFrame`, perform an automatic form of double buffering, so we needn't worry about it. Some graphics environments, including Java's AWT environment, do not perform double buffering automatically, in which case the program itself must carry it out.

Like the other examples in this chapter, the game of Pong provides a simple illustration of how threads are used to coordinate concurrent actions in a computer program. As most computer game fans will realize, most modern interactive computer games utilize a multithreaded design. The use of threads allows our interactive programs to achieve a responsiveness and sophistication that is not possible in single-threaded programs. One of the great advantages of Java is that it simplifies the use of threads, thereby making thread programming accessible to programmers. However, one of the lessons that should be drawn from this chapter is that multithreaded programs must be carefully designed in order to work effectively.

SELF-STUDY EXERCISE

EXERCISE 14.12 Modify the `PongApplet` program so that it contains a second ball that starts at a different location from the first ball.

CHAPTER SUMMARY

Technical Terms

asynchronous
blocked
busy waiting
concurrent
critical section
dispatched
fetch-execute cycle
lock
monitor
multitasking
multithreaded
mutual exclusion
priority scheduling
producer/consumer model
quantum
queue
ready queue
round-robin scheduling
scheduling algorithm
task
thread
thread life cycle
time slicing

Summary of Important Points

- *Multitasking* is the technique of executing several tasks at the same time within a single program. In Java we give each task a separate *thread of execution*, thus resulting in a *multithreaded* program.
- A *sequential* computer with a single *central processing unit (CPU)* can execute only one machine instruction at a time. A *parallel* computer uses multiple CPUs operating simultaneously to execute more than one instruction at a time.

- Each CPU uses a *fetch-execute cycle* to retrieve the next machine instruction from memory and execute it. The cycle is under the control of the CPU's internal clock, which typically runs at several hundred *megahertz*—where 1 megahertz (MHz) is 1 million cycles per second.
- *Time slicing* is the technique whereby several threads can share a single CPU over a given time period. Each thread is given a small slice of the CPU's time under the control of some kind of scheduling algorithm.
- In *round-robin scheduling*, each thread is given an equal slice of time, in a first-come–first-served order. In *priority scheduling*, higher-priority threads are allowed to run before lower-priority threads are run.
- There are generally two ways of creating threads in a program. One is to create a subclass of `Thread` and implement a `run()` method. The other is to create a `Thread` instance and pass it a `Runnable` object—that is, an object that implements `run()`.
- The `sleep()` method removes a thread from the CPU for a determinate length of time, giving other threads a chance to run.
- The `setPriority()` method sets a thread's priority. Higher-priority threads have more and longer access to the CPU.
- Threads are *asynchronous*. Their timing and duration on the CPU are highly sporadic and unpredictable. In designing threaded programs, you must be careful not to base your algorithm on any assumptions about the threads' timing.
- To improve the responsiveness of interactive programs, you could give compute-intensive tasks, such as drawing lots of dots, to a lower-priority thread or to a thread that sleeps periodically.
- A thread's life cycle consists of ready, running, waiting, sleeping, and blocked states. Threads start in the ready state and are dispatched to the CPU by the scheduler, an operating system program. If a thread performs an I/O operation, it blocks until the I/O is completed. If it voluntarily sleeps, it gives up the CPU.
- According to the *producer/consumer* model, two threads share a resource, one serving to produce the resource and the other to consume the resource. Their cooperation must be carefully synchronized.
- An object that contains `synchronized` methods is known as a *monitor*. Such objects ensure that only one thread at a time can execute a synchronized method. The object is *locked* until the thread completes the method or voluntarily sleeps. This is one way to ensure mutually exclusive access to a resource by a collection of cooperating threads.
- The `synchronized` qualifier can also be used to designate a method as a *critical section*, whose execution should not be preempted by one of the other cooperating threads.
- In designing multithreaded programs, it is useful to assume that if a thread *can* be interrupted at a certain point, it *will* be interrupted there. Thread coordination should never be left to chance.
- One way of coordinating two or more cooperating threads is to use the `wait/notify` combination. One thread waits for a resource to be available, and the other thread notifies when a resource becomes available.

SOLUTIONS TO SELF-STUDY EXERCISES

SOLUTION 14.1

```
public class PrintOdds implements Runnable {
  private int bound;
  public PrintOdds(int b) {
   bound = b;
  }

  public void print() {
    if (int k = 1; k < bound; k+=2)
      System.out.println(k);
  }

  public void run() {
    print();
  }
}
```

SOLUTION 14.2 On my system, the experiment yielded the following output, if each thread printed its number after every 100,000 iterations:

```
11111122222222111111113333333222222211111133333332222244444444333333344444445555555544444455555555555
```

This suggests that round-robin scheduling is being used.

SOLUTION 14.3 If each thread is given 50 milliseconds of sleep on each iteration, they tend to run in the order in which they were created:

```
123451234512345...
```

SOLUTION 14.4 The garbage collector runs whenever the available memory drops below a certain threshold. It must have higher priority than the application, since the application won't be able to run if it runs out of memory.

SOLUTION 14.5 To improve the responsiveness of an interactive program, the system could give a high priority to the threads that interact with the user and a low priority to those that perform noninteractive computations, such as number crunching.

SOLUTION 14.6 If the JVM were single threaded, it wouldn't be possible to break out of an infinite loop, because the program's loop would completely consume the CPU's attention.

SOLUTION 14.7 If round-robin scheduling is used, each thread will be get a portion of the CPU's time, so the GUI thread will eventually get its turn. But you don't know how long it will be before the GUI gets its turn, so there might still be an unacceptably long wait before the user's actions are handled. Thus, to *guarantee* responsiveness, it is better to have the drawing thread sleep on every iteration.

SOLUTION 14.8 If `Dotty`'s priority is set to 1, a low value, this does improve the responsiveness of the interface, but it is significantly less responsive than using a `sleep()` on each iteration.

SOLUTION 14.9 In a real bakery only one customer at a time can take a number. The take-a-number gadget "enforces" mutual exclusion by virtue of its design: There's room for only one hand to grab the ticket and there's only one ticket per number. If two customers got "bakery rage" and managed to grab the same ticket, it would rip in half and neither would benefit.

SOLUTION 14.10 One experiment to run would be to make the clerk's performance very slow by using large sleep intervals. If the algorithm is correct, this should not affect the order in which customers are served. Another experiment would be to force the clerk to work fast but the customers to work slowly. This should still not affect the order in which the customers are served.

SOLUTION 14.11 You should observe that the waiting line builds up as customers enter the bakery, but the clerk should still serve the customers in the correct order.

SOLUTION 14.12 A two-ball version of Pong would require the following changes to the original version:

1. A new `Ball()` constructor that has parameters to set the initial location and direction of the ball.
2. The `PongApplet` should create a new `Ball` instance, start it, and draw it.

EXERCISES

Note: *For programming exercises,* **first** *draw a UML class diagram describing all classes and their inheritance relationships and/or associations.*

EXERCISE 14.1 Explain the difference between the following pairs of terms:

a. *Blocked* and *ready*.
b. *Priority* and *round-robin* scheduling.
c. *Producer* and *consumer*.
d. *Monitor* and *lock*.
e. *Concurrent* and *time slicing*.
f. *Mutual exclusion* and *critical section*.
g. *Busy* and *nonbusy* waiting.

EXERCISE 14.2 Fill in the blanks.

a. _____ happens when a CPU's time is divided among several different threads.
b. A method that should not be interrupted during its execution is known as a _____.
c. The scheduling algorithm in which each thread gets an equal portion of the CPU's time is known as _____.
d. The scheduling algorithm in which some threads can preempt other threads is known as _____.
e. A _____ is a mechanism that enforces mutually exclusive access to a synchronized method.
f. A thread that performs an I/O operation may be forced into the _____ state until the operation is completed.

EXERCISE 14.3 Describe the concept of *time slicing* as it applies to CPU scheduling.

EXERCISE 14.4 What's the difference in the way concurrent threads would be implemented on a computer with several processors and on a computer with a single processor?

EXERCISE 14.5 Why are threads put into the *blocked* state when they perform an I/O operation?

EXERCISE 14.6 What's the difference between a thread in the sleep state and a thread in the ready state?

EXERCISE 14.7 **Deadlock** is a situation that occurs when one thread is holding a resource that another thread is waiting for, while the other thread is holding a resource that the first thread is waiting for. Describe how deadlock can occur at a four-way intersection with cars entering from each branch. How can it be avoided?

EXERCISE 14.8 **Starvation** can occur if one thread is repeatedly preempted by other threads. Describe how starvation can occur at a four-way intersection and how it can be avoided.

EXERCISE 14.9 Use the `Runnable` interface to define a thread that repeatedly generates random numbers in the interval 2 through 12.

EXERCISE 14.10 Create a version of the `Bakery` program that uses two clerks to serve customers.

EXERCISE 14.11 Modify the `Numbers` program so that the user can interactively create `NumberThread`s and assign them a priority. Modify the `NumberThread`s so that they print their numbers indefinitely (rather than for a fixed number of iterations). Then experiment with the system by observing the effect of introducing threads with the same, lower, or higher priority. How do the threads behave when they all have the same priority? What happens when you introduce a higher-priority thread into the mix? What happens when you introduce a lower-priority thread into the mix?

EXERCISE 14.12 Create a bouncing ball simulation in which a single ball (thread) bounces up and down in a vertical line. The ball should bounce off the bottom and top of the enclosing frame.

EXERCISE 14.13 Modify the simulation in the previous exercise so that more than one ball can be introduced. Allow the user to introduce new balls into the simulation by pressing the space bar or clicking the mouse.

EXERCISE 14.14 Modify your solution to the previous problem by having the balls bounce off the wall at a random angle.

EXERCISE 14.15 **Challenge:** One type of producer/consumer problem is the *reader/writer* problem. Create a subclass of `JTextField` that can be shared by threads, one of which writes a random number to the text field, and the other of which reads the value in the text field. Coordinate the two threads so that the overall effect of the program will be to print the values from 0 to 100 in the proper order. In other words, the reader thread shouldn't read a value from the text field until there's a value to be read. The writer thread shouldn't write a value to the text field until the reader has read the previous value.

EXERCISE 14.16 **Challenge:** Create a streaming banner thread that moves a simple message across a panel. The message should repeatedly enter at the left edge of the panel and exit from the right edge. Design the banner as a subclass of `JPanel` and have it implement the `Runnable` interface. That way it can be added to any user interface. One of its constructors should take a `String` argument that lets the user set the banner's message.

EXERCISE 14.17 **Challenge:** Create a slide show applet, which repeatedly cycles through an array of images. The action of displaying the images should be a separate thread. The applet thread should handle the user interface. Give the user some controls that let it pause, stop, start, speed up, and slow down the images.

EXERCISE 14.18 **Challenge:** Create a horse race simulation, using separate threads for each of the horses. The horses should race horizontally across the screen, with each horse having a different vertical coordinate. If you don't have good horse images to use, just make each horse a colored polygon or some other shape. Have the horses implement the `Drawable` interface, which we introduced in Chapter 8.

EXERCISE 14.19 **Challenge:** Create a multithreaded digital clock application. One thread should keep time in an endless while loop. The other thread should be responsible for updating the screen each second.