CHAPTER

32

Multithreading and Parallel Programming

Objectives

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

32.1 Introduction

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

multithreading

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

32.2 Thread Concepts

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

thread
task

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

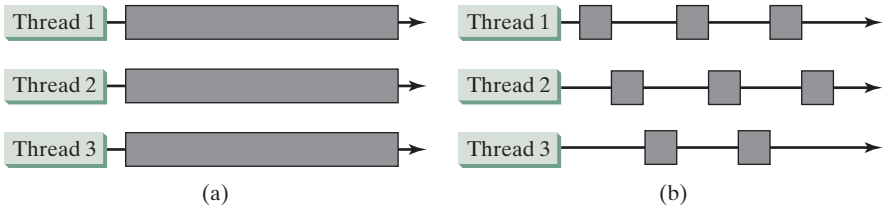


FIGURE 32.1 (a) Multiple threads running on multiple CPUs. (b) Multiple threads share a single CPU.

time sharing

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

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

task
runnable object
thread

You can create additional threads to run concurrent tasks in the program. In Java, each task is an instance of the `Runnable` interface, also called a *runnable object*. A *thread* is essentially an object that facilitates the execution of a task.

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

32.2.2 What is a runnable object? What is a thread?

32.3 Creating Tasks and Threads

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

Runnable interface
run() method

Tasks are objects. To create tasks, you have to first define a class for tasks, which implements the `Runnable` interface. The `Runnable` interface is rather simple. All it contains is the `run()` method. You need to implement this method to tell the system how your thread is going to run. A template for developing a task class is shown in Figure 32.2a.

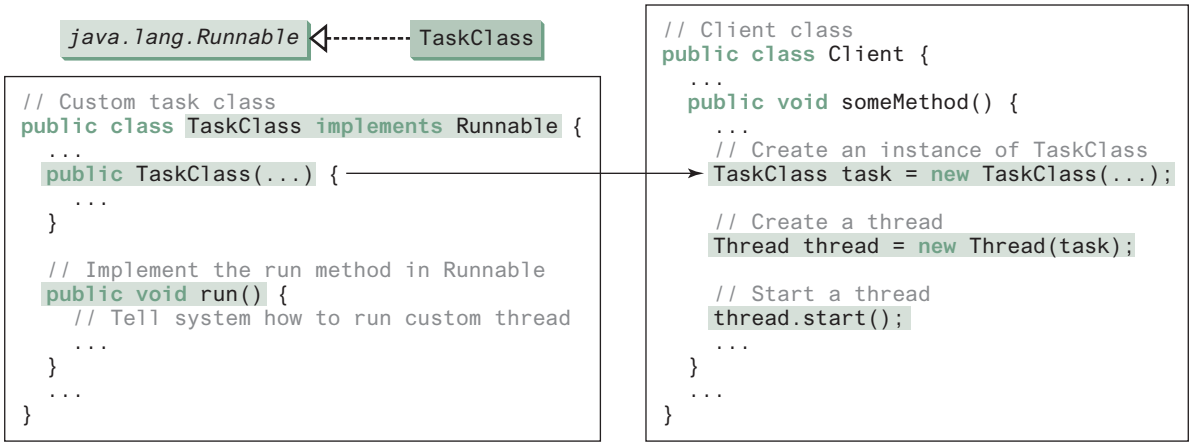


FIGURE 32.2 Define a task class by implementing the `Runnable` interface.

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

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

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

`Thread` class
create a task

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

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

create a thread

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

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

start a thread

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

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

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

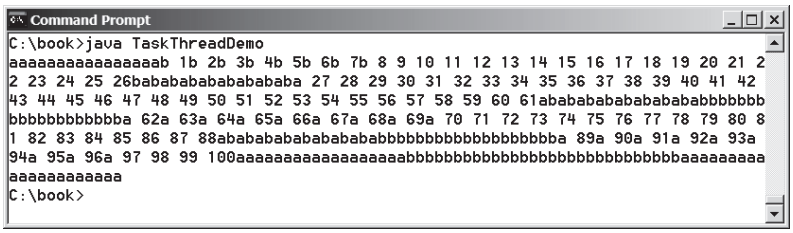


FIGURE 32.3 Tasks `printA`, `printB`, and `print100` are executed simultaneously to display the letter `a` 100 times, the letter `b` 100 times, and the numbers from 1 to 100.

LISTING 32.1 TaskThreadDemo.java

create tasks

create threads

start threads

task class

run

task class

run

```
1  public class TaskThreadDemo {
2    public static void main(String[] args) {
3      // Create tasks
4      Runnable printA = new PrintChar('a', 100);
5      Runnable printB = new PrintChar('b', 100);
6      Runnable print100 = new PrintNum(100);
7
8      // Create threads
9      Thread thread1 = new Thread(printA);
10     Thread thread2 = new Thread(printB);
11     Thread thread3 = new Thread(print100);
12
13     // Start threads
14     thread1.start();
15     thread2.start();
16     thread3.start();
17   }
18 }
19
20 // The task for printing a character a specified number of times
21 class PrintChar implements Runnable {
22   private char charToPrint; // The character to print
23   private int times; // The number of times to repeat
24
25   /** Construct a task with a specified character and number of
26     * times to print the character
27     */
28   public PrintChar(char c, int t) {
29     charToPrint = c;
30     times = t;
31   }
32
33   @Override /** Override the run() method to tell the system
34     * what task to perform
35     */
36   public void run() {
37     for (int i = 0; i < times; i++) {
38       System.out.print(charToPrint);
39     }
40   }
41 }
42
43 // The task class for printing numbers from 1 to n for a given n
44 class PrintNum implements Runnable {
45   private int lastNum;
46
47   /** Construct a task for printing 1, 2, ..., n */
48   public PrintNum(int n) {
49     lastNum = n;
50   }
51
52   @Override /** Tell the thread how to run */
53   public void run() {
54     for (int i = 1; i <= lastNum; i++) {
55       System.out.print(" " + i);
56     }
57   }
58 }
```

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

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

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

Note

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

effect of concurrency

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

Important Note

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

`run()` method

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

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

Check Point

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

Replaced by

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

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

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

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

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

(a)

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

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

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

(b)

32.4 The Thread Class

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

Figure 32.4 shows the class diagram for the **Thread** class.

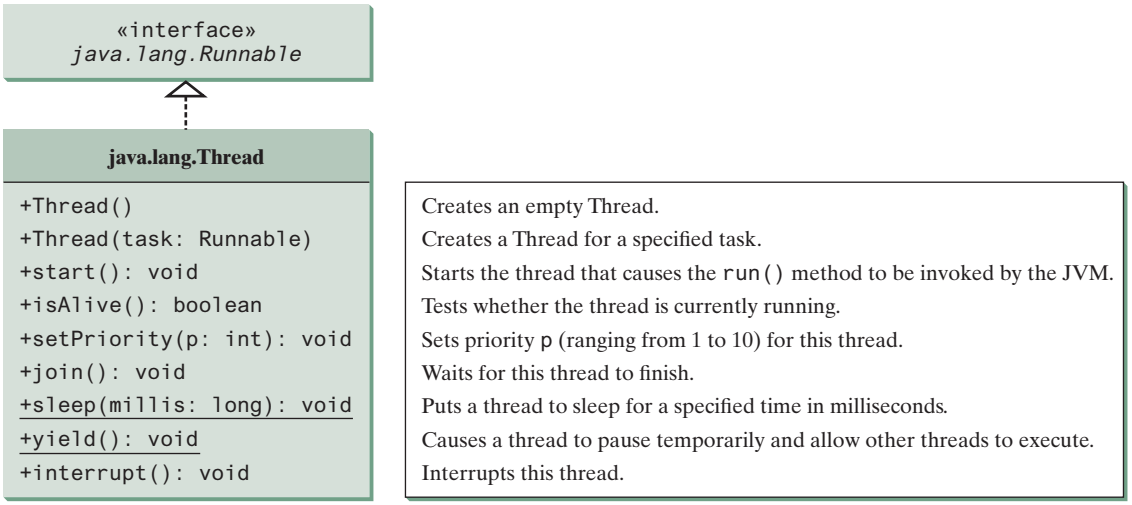


FIGURE 32.4 The **Thread** class contains the methods for controlling threads.

Note

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

separating task from thread

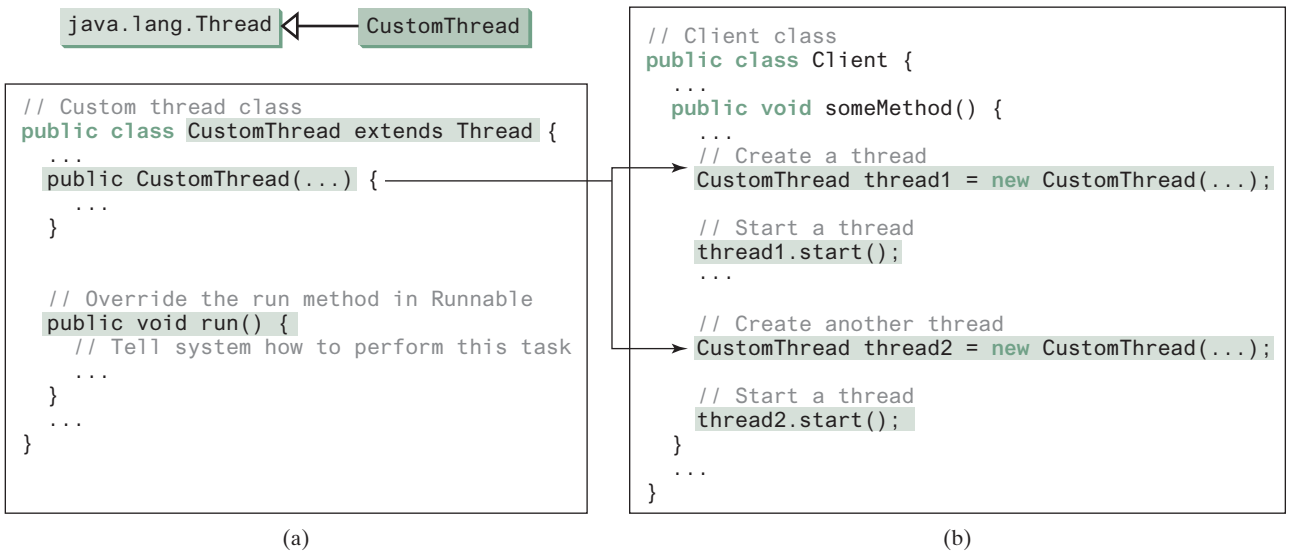


FIGURE 32.5 Define a thread class by extending the **Thread** class.

This approach is, however, not recommended because it mixes the task and the mechanism of running the task. Separating the task from the thread is a preferred design.

Note

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

deprecated method

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

`yield()`

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

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

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

`sleep(long)`

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

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

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

`InterruptedException`

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

(a) Correct

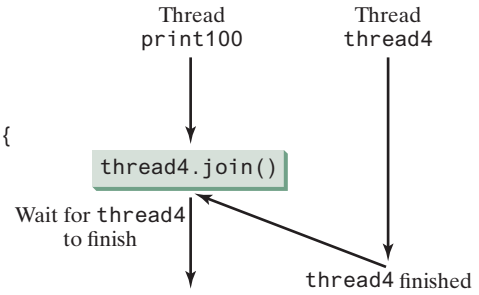
```
public void run() {
  while (...) {
    try {
      ...
      Thread.sleep(sleepTime);
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

(b) Incorrect

join()

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

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



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

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

setPriority(int)

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

round-robin scheduling

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

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

Tip

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

Tip

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

contention or starvation

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

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

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

32.4.3 How do you set a priority for a thread? What is the default priority?

32.5 Animation Using Threads and the `Platform.runLater` Method

You can use a thread to control an animation and run the code in JavaFX GUI thread using the `Platform.runLater` *method.*

The use of a `Timeline` object to control animations was introduced in Section 15.11, Animation. Alternatively, you can also use a thread to control animation. Listing 32.2 gives an example that displays flashing text on a label, as shown in Figure 32.6.

FIGURE 32.6 The text "Welcome" blinks.

LISTING 32.2 FlashText.java

```
1  import javafx.application.Application;
2  import javafx.application.Platform;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.layout.StackPane;
6  import javafx.stage.Stage;
7
8  public class FlashText extends Application {
9    private String text = "";
10
11   @Override // Override the start method in the Application class
12   public void start(Stage primaryStage) {
13     StackPane pane = new StackPane();
14     Label lblText = new Label("Programming is fun");                    create a label
15     pane.getChildren().add(lblText);                                    label in a pane
16
17     new Thread(new Runnable() {                                         create a thread
18       @Override
19       public void run() {                                               run thread
20         try {
21           while (true) {
22             if (lblText.getText().trim().length() == 0)                 change text
23               text = "Welcome";
24             else
25               text = "";
26
27             Platform.runLater(new Runnable() { // Run from JavaFX GUI    Platform.runLater
28               @Override
29               public void run() {
30                 lblText.setText(text);                                  update GUI
31               }
32             });
33
34             Thread.sleep(200);                                          sleep
35           }
36         }
37         catch (InterruptedException ex) {
38         }
39       }
40     }).start();
41
```

```
42      // Create a scene and place it in the stage
43      Scene scene = new Scene(pane, 200, 50);
44      primaryStage.setTitle("FlashText"); // Set the stage title
45      primaryStage.setScene(scene); // Place the scene in the stage
46      primaryStage.show(); // Display the stage
47    }
48  }
```

The program creates a `Runnable` object in an anonymous inner class (lines 17–40). This object is started in line 40 and runs continuously to change the text in the label. It sets a text in the label if the label is blank (line 23) and sets its text blank (line 25) if the label has a text. The text is set and unset to simulate a flashing effect.

JavaFX application thread

JavaFX GUI is run from the *JavaFX application thread*. The flashing control is run from a separate thread. The code in a nonapplication thread cannot update GUI in the application thread. To update the text in the label, a new `Runnable` object is created in lines 27–32. Invoking

`Platform.runLater`

`Platform.runLater(Runnable r)` tells the system to run this `Runnable` object in the application thread.

The anonymous inner classes in this program can be simplifed using lambda expressions as follows:

```
new Thread(() -> { // lambda expression
  try {
    while (true) {
      if (lblText.getText().trim().length() == 0)
        text = "Welcome";
      else
        text = "";

      Platform.runLater(() -> lblText.setText(text)); // lambda exp

      Thread.sleep(200);
    }
  }
  catch (InterruptedException ex) {
  }
}).start();
```

32.5.1 What causes the text to flash?

32.5.2 Is an instance of `FlashText` a runnable object?

32.5.3 What is the purpose of using `Platform.runLater`?

32.5.4 Can you replace the code in lines 27–32 using the following code?

```
Platform.runLater(e -> lblText.setText(text));
```

32.5.5 What happens if line 34 (`Thread.sleep(200)`) is not used?

32.5.6 There is an issue in Listing 16.9, ListViewDemo. If you press the CTRL key and select Canada, Demark, and China in this order, you will get an `ArrayIndexOutBoundsException`. What is the reason for this error and how do you fix it? (Thanks to Henri Heimonen of Finland for contributing this question).

32.6 Thread Pools

A thread pool can be used to execute tasks efficiently.

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

```
Runnable task = new TaskClass(...);
new Thread(task).start();
```

This approach is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance. Using a *thread pool* is an ideal way to manage the number of tasks executing concurrently. Java provides the `Executor` interface for executing tasks in a thread pool and the `ExecutorService` interface for managing and controlling tasks. `ExecutorService` is a subinterface of `Executor`, as shown in Figure 32.7.

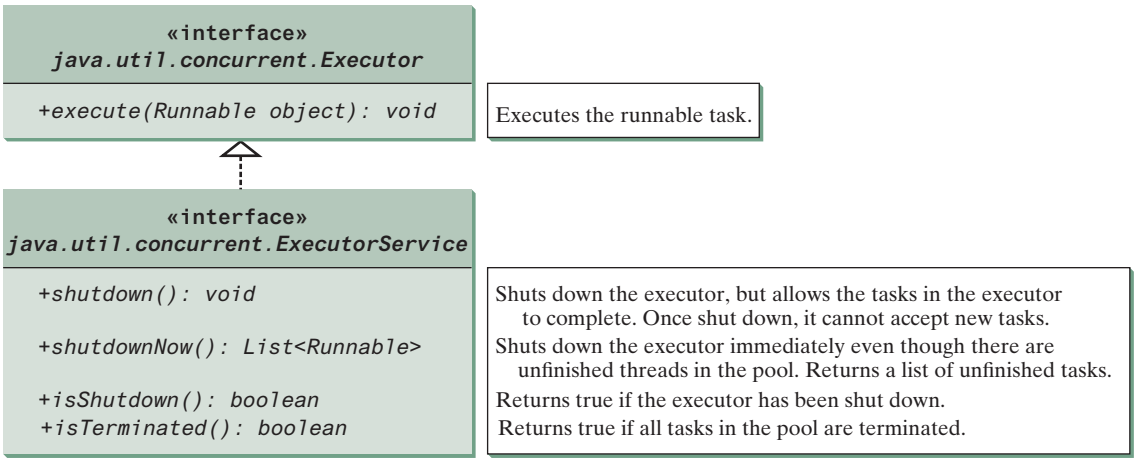


FIGURE 32.7 The `Executor` interface executes threads and the `ExecutorService` subinterface manages threads.

To create an `Executor` object, use the static methods in the `Executors` class, as shown in Figure 32.8. The `newFixedThreadPool(int)` method creates a fixed number of threads in a pool. If a thread completes executing a task, it can be reused to execute another task. If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution. The `newCachedThreadPool()` method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution. A thread in a cached pool will be terminated if it has not been used for 60 seconds. A cached pool is efficient for many short tasks.

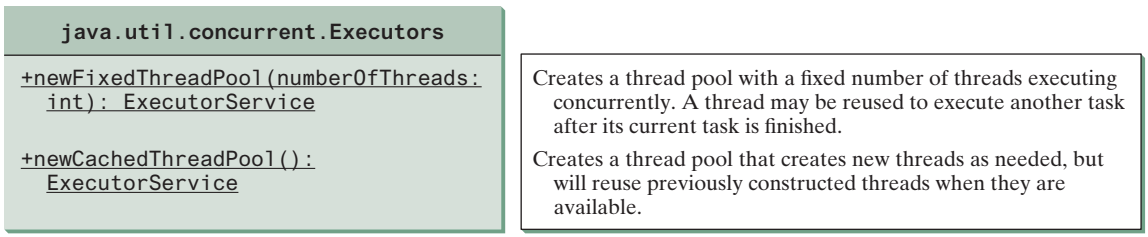


FIGURE 32.8 The `Executors` class provides static methods for creating `Executor` objects.

Listing 32.3 shows how to rewrite Listing 32.1 using a thread pool.

LISTING 32.3 ExecutorDemo.java

```
1  import java.util.concurrent.*;
2
3  public class ExecutorDemo {
```

```
 4   public static void main(String[] args) {
 5     // Create a fixed thread pool with maximum three threads
 6     ExecutorService executor = Executors.newFixedThreadPool(3);
 7
 8     // Submit runnable tasks to the executor
 9     executor.execute(new PrintChar('a', 100));
10     executor.execute(new PrintChar('b', 100));
11     executor.execute(new PrintNum(100));
12
13     // Shut down the executor
14     executor.shutdown();
15   }
16 }
```

create executor

submit task

shut down executor

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

Suppose you replace line 6 with

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

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

Suppose you replace line 6 with

```
ExecutorService executor = Executors.newCachedThreadPool();
```

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

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

Tip

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

32.6.1 What are the benefits of using a thread pool?

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

32.7 Thread Synchronization

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

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

Suppose that you create and launch 100 threads, each of which adds a penny to an account. Define a class named `Account` to model the account, a class named `AddAPennyTask` to add a penny to the account, and a main class that creates and launches threads. The relationships of these classes are shown in Figure 32.9. The program is given in Listing 32.4.

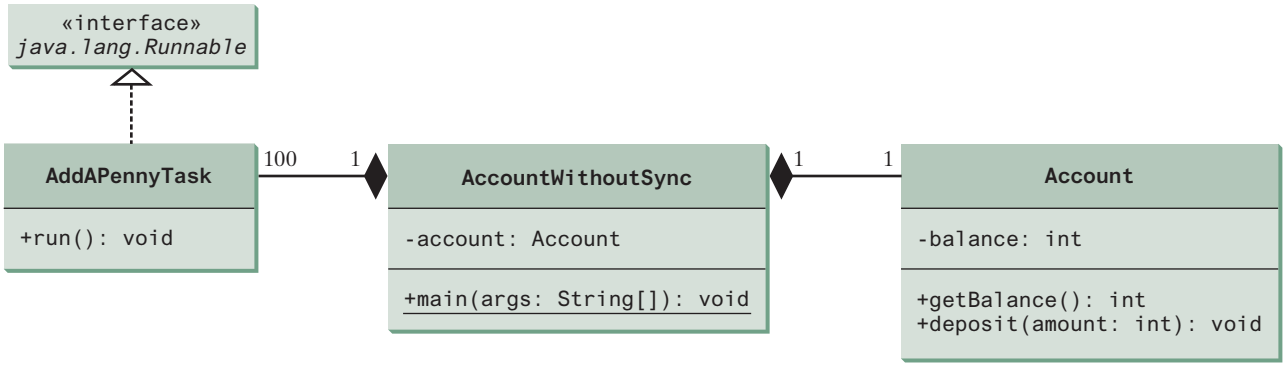


FIGURE 32.9 `AccountWithoutSync` contains an instance of `Account` and 100 threads of `AddAPennyTask`.

LISTING 32.4 AccountWithoutSync.java

```
1  import java.util.concurrent.*;
2
3  public class AccountWithoutSync {
4    private static Account account = new Account();
5
6    public static void main(String[] args) {
7      ExecutorService executor = Executors.newCachedThreadPool();          create executor
8
9      // Create and launch 100 threads
10     for (int i = 0; i < 100; i++) {
11       executor.execute(new AddAPennyTask());                             submit task
12     }
13
14     executor.shutdown();                                                 shut down executor
15
16     // Wait until all tasks are finished
17     while (!executor.isTerminated()) {                                   wait for all tasks to terminate
18     }
19
20     System.out.println("What is balance? " + account.getBalance());
21   }
22
23   // A thread for adding a penny to the account
24   private static class AddAPennyTask implements Runnable {
25     public void run() {
26       account.deposit(1);
27     }
28   }
29
30   // An inner class for account
31   private static class Account {
32     private int balance = 0;
33
34     public int getBalance() {
35       return balance;
36     }
37
38     public void deposit(int amount) {
39       int newBalance = balance + amount;
40
41       // This delay is deliberately added to magnify the
```

```
42          // data-corruption problem and make it easy to see.
43          try {
44            Thread.sleep(5);
45          }
46          catch (InterruptedException ex) {
47          }
48
49          balance = newBalance;
50        }
51      }
52    }
```

The classes `AddAPennyTask` and `Account` in lines 24–51 are inner classes. Line 4 creates an `Account` with initial balance `0`. Line 11 creates a task to add a penny to the account and submits the task to the executor. Line 11 is repeated 100 times in lines 10–12. The program repeatedly checks whether all tasks are completed in lines 17 and 18. The account balance is displayed in line 20 after all tasks are completed.

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

The balance of the account is initially `0` (line 32). When all the threads are finished, the balance should be `100` but the output is unpredictable. As can be seen in Figure 32.10, the answers are wrong in the sample run. This demonstrates the data-corruption problem that occurs when all the threads have access to the same data source simultaneously.

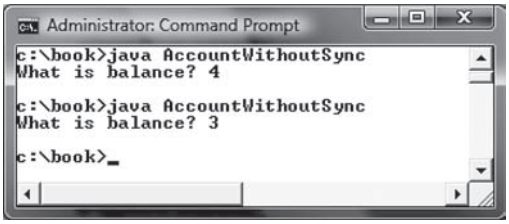


FIGURE 32.10 The `AccountWithoutSync` program causes data inconsistency.

Lines 39–49 could be replaced by one statement:

```
balance = balance + amount;
```

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

What, then, caused the error in this program? A possible scenario is shown in Figure 32.11.

Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

FIGURE 32.11 Task 1 and Task 2 both add 1 to the same balance.

In Step 1, Task 1 gets the balance from the account. In Step 2, Task 2 gets the same balance from the account. In Step 3, Task 1 writes a new balance to the account. In Step 4, Task 2 writes a new balance to the account.

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

race condition

thread-safe

32.7.1 The `synchronized` Keyword

To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the *critical region*. The critical region in Listing 32.4 is the entire `deposit` method. You can use the keyword `synchronized` to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in Listing 32.4. One approach is to make `Account` thread-safe by adding the keyword `synchronized` in the `deposit` method in line 38, as follows:

critical region

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

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

With the `deposit` method synchronized, the preceding scenario cannot happen. If Task 1 enters the method, Task 2 is blocked until Task 1 finishes the method, as shown in Figure 32.12.

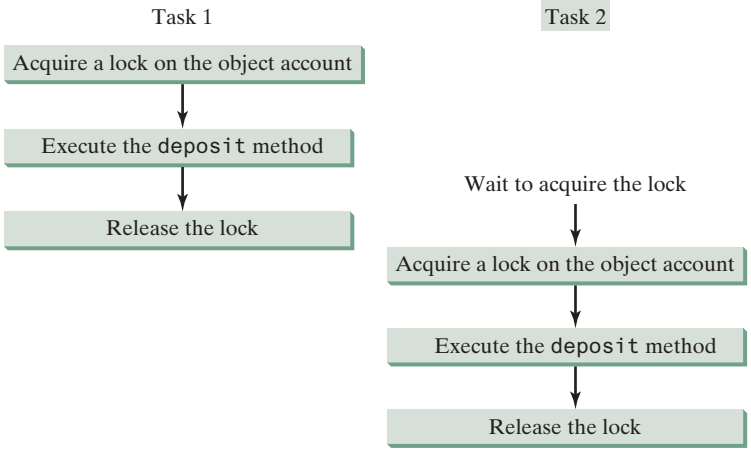


FIGURE 32.12 Task 1 and Task 2 are synchronized.

32.7.2 Synchronizing Statements

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

synchronized block

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

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

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

Synchronized statements enable you to synchronize part of the code in a method instead of the entire method. This increases concurrency. You can make Listing 32.4 thread-safe by placing the statement in line 26 inside a synchronized block:

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

Note

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

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

(a)

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

(b)

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

32.7.2 Suppose you place the statement in line 26 of Listing 32.4 inside a synchronized block to avoid race conditions, as follows:

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

Will it work?

32.8 Synchronization Using Locks

Locks and conditions can be explicitly used to synchronize threads.

Recall that in Listing 32.4, 100 tasks deposit a penny to the same account concurrently, which causes conflicts. To avoid it, you use the `synchronized` keyword in the `deposit` method, as follows:

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

lock

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

Java enables you to acquire locks explicitly, which give you more control for coordinating threads. A lock is an instance of the `Lock` interface, which defines the methods for acquiring and releasing locks, as shown in Figure 32.13. A lock may also use the `newCondition()` method to create any number of `Condition` objects, which can be used for thread communications.

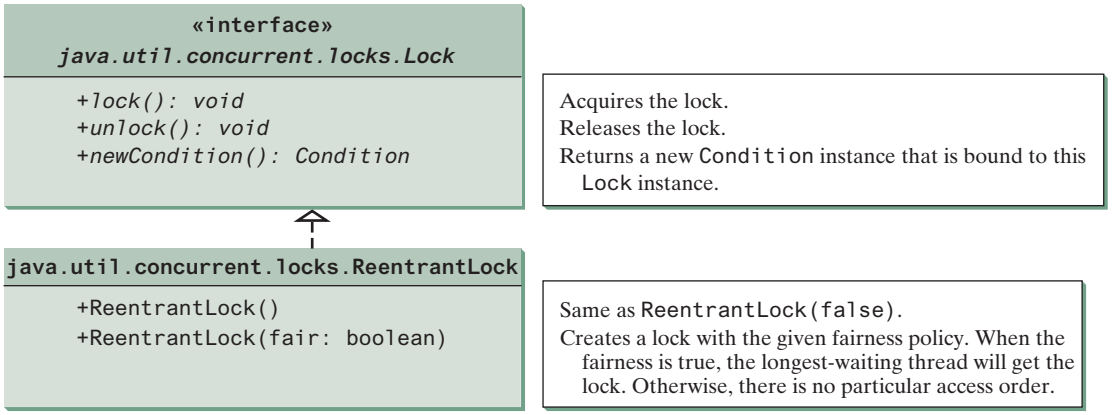


FIGURE 32.13 The `ReentrantLock` class implements the `Lock` interface to represent a lock.

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

fairness policy

Listing 32.5 revises the program in Listing 32.7 to synchronize the account modification using explicit locks.

LISTING 32.5 AccountWithSyncUsingLock.java

```
1  import java.util.concurrent.*;
2  import java.util.concurrent.locks.*;                    package for locks
3
4  public class AccountWithSyncUsingLock {
5    private static Account account = new Account();
6
7    public static void main(String[] args) {
8      ExecutorService executor = Executors.newCachedThreadPool();
9
10     // Create and launch 100 threads
11     for (int i = 0; i < 100; i++) {
12       executor.execute(new AddAPennyTask());
13     }
14
15     executor.shutdown();
16
17     // Wait until all tasks are finished
18     while (!executor.isTerminated()) {
19     }
20
21     System.out.println("What is balance? " + account.getBalance());
22   }
23
24   // A thread for adding a penny to the account
25   public static class AddAPennyTask implements Runnable {
26     public void run() {
27       account.deposit(1);
28     }
29   }
30
```

create a lock

acquire the lock

release the lock

```
31    // An inner class for Account
32    public static class Account {
33      private static Lock lock = new ReentrantLock(); // Create a lock
34      private int balance = 0;
35
36      public int getBalance() {
37        return balance;
38      }
39
40      public void deposit(int amount) {
41        lock.lock(); // Acquire the lock
42
43        try {
44          int newBalance = balance + amount;
45
46          // This delay is deliberately added to magnify the
47          // data-corruption problem and make it easy to see.
48          Thread.sleep(5);
49
50          balance = newBalance;
51        }
52        catch (InterruptedException ex) {
53        }
54        finally {
55          lock.unlock(); // Release the lock
56        }
57      }
58    }
59  }
```

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

Tip

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

Listing 32.5 can be implemented using a synchronize method for `deposit` rather than using a lock. In general, using `synchronized` methods or statements is simpler than using explicit locks for mutual exclusion. However, using explicit locks is more intuitive and flexible to synchronize threads with conditions, as you will see in the next section.

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

32.9 Cooperation among Threads

Conditions on locks can be used to coordinate thread interactions.

condition

Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region, but sometimes you also need a way for threads to cooperate. *Conditions* can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the `newCondition()` method on a `Lock` object. Once a condition is created, you can use its `await()`, `signal()`, and `signalAll()` methods for thread communications, as shown in Figure 32.14. The `await()` method causes the current thread to wait until the condition is signaled. The `signal()` method wakes up one waiting thread, and the `signalAll()` method wakes all waiting threads.

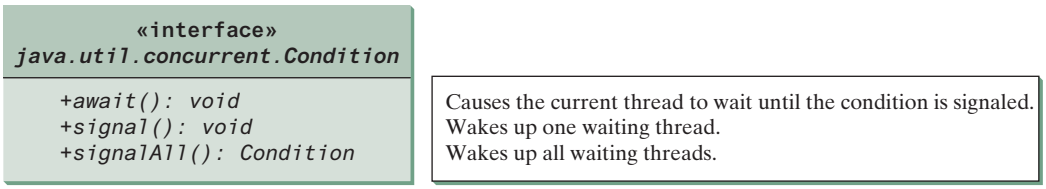


FIGURE 32.14 The `Condition` interface defines the methods for performing synchronization.

thread cooperation example

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

To synchronize the operations, use a lock with a condition: `newDeposit` (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the `newDeposit` condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown in Figure 32.15.

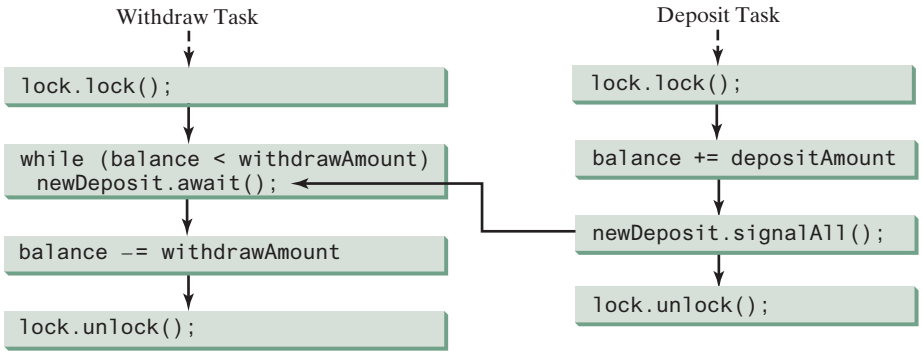


FIGURE 32.15 The condition `newDeposit` is used for communications between the two threads.

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

Assume the initial balance is `0` and the amount to deposit and withdraw are randomly generated. Listing 32.6 gives the program. A sample run of the program is shown in Figure 32.16.

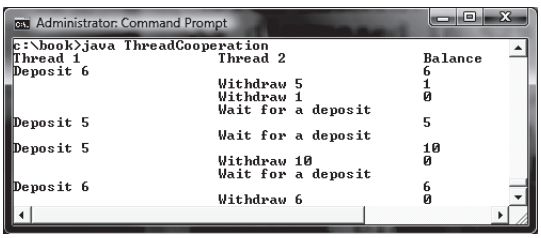


FIGURE 32.16 The withdraw task waits if there are not sufficient funds to withdraw.

LISTING 32.6 ThreadCooperation.java

```
1  import java.util.concurrent.*;
2  import java.util.concurrent.locks.*;
3
4  public class ThreadCooperation {
5    private static Account account = new Account();
6
7    public static void main(String[] args) {
8      // Create a thread pool with two threads
9      ExecutorService executor = Executors.newFixedThreadPool(2);
10     executor.execute(new DepositTask());
11     executor.execute(new WithdrawTask());
12     executor.shutdown();
13
14     System.out.println("Thread 1\t\tThread 2\t\tBalance");
15   }
16
17   public static class DepositTask implements Runnable {
18     @Override // Keep adding an amount to the account
19     public void run() {
20       try { // Purposely delay it to let the withdraw method proceed
21         while (true) {
22           account.deposit((int)(Math.random() * 10) + 1);
23           Thread.sleep(1000);
24         }
25       }
26       catch (InterruptedException ex) {
27         ex.printStackTrace();
28       }
29     }
30   }
31
32   public static class WithdrawTask implements Runnable {
33     @Override // Keep subtracting an amount from the account
34     public void run() {
35       while (true) {
36         account.withdraw((int)(Math.random() * 10) + 1);
37       }
38     }
39   }
40
41   // An inner class for account
42   private static class Account {
43     // Create a new lock
44     private static Lock lock = new ReentrantLock();
45
46     // Create a condition
47     private static Condition newDeposit = lock.newCondition();
48
49     private int balance = 0;
50
51     public int getBalance() {
52       return balance;
53     }
54
55     public void withdraw(int amount) {
56       lock.lock(); // Acquire the lock
57       try {
58         while (balance < amount) {
```

create two threads

create a lock

create a condition

acquire the lock

```
59            System.out.println("\t\t\tWait for a deposit");
60            newDeposit.await();                                    wait on the condition
61          }
62
63          balance -= amount;
64          System.out.println("\t\t\tWithdraw " + amount +
65            "\t\t" + getBalance());
66        }
67        catch (InterruptedException ex) {
68          ex.printStackTrace();
69        }
70        finally {
71          lock.unlock(); // Release the lock                      release the lock
72        }
73      }
74
75      public void deposit(int amount) {
76        lock.lock(); // Acquire the lock                          acquire the lock
77        try {
78          balance += amount;
79          System.out.println("Deposit " + amount +
80            "\t\t\t\t\t" + getBalance());
81
82          // Signal thread waiting on the condition
83          newDeposit.signalAll();                                signal threads
84        }
85        finally {
86          lock.unlock(); // Release the lock                      release the lock
87        }
88      }
89    }
90  }
```

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

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

A lock is created in line 44. A condition named `newDeposit` on the lock is created in line 47. A condition is bound to a lock. Before waiting or signaling the condition, a thread must first acquire the lock for the condition. The withdraw task acquires the lock in line 56, waits for the `newDeposit` condition (line 60) when there is not a sufficient amount to withdraw, and releases the lock in line 71. The deposit task acquires the lock in line 76 and signals all waiting threads (line 83) for the `newDeposit` condition after a new deposit is made.

What will happen if you replace the `while` loop in lines 58–61 with the following `if` statement?

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

The deposit task will notify the withdraw task whenever the balance changes. `(balance < amount)` may still be true when the withdraw task is awakened. Using the `if` statement would

lead to incorrect withdraw. Using the loop statement, the withdraw task will have a chance to recheck the condition.

ever-waiting threads

Caution

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

`IllegalMonitorStateException`

Caution

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

Locks and conditions were introduced in Java 5. Prior to Java 5, thread communications were programmed using the object's built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor, so will not need to use monitors. However, if you are working with legacy Java code, you may encounter Java's built-in monitor.

Java's built-in monitor

monitor

A *monitor* is an object with mutual exclusion and synchronization capabilities. Only one thread can execute a method at a time in the monitor. A thread enters the monitor by acquiring a lock on it and exits by releasing the lock. *Any object can be a monitor*. An object becomes a monitor once a thread locks it. Locking is implemented using the `synchronized` keyword on a method or a block. A thread must acquire a lock before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor. You can invoke the `wait()` method on the monitor object to release the lock so some other thread can get in the monitor and perhaps change the monitor's state. When the condition is right, the other thread can invoke the `notify()` or `notifyAll()` method to signal one or all waiting threads to regain the lock and resume execution. The template for invoking these methods is shown in Figure 32.17.

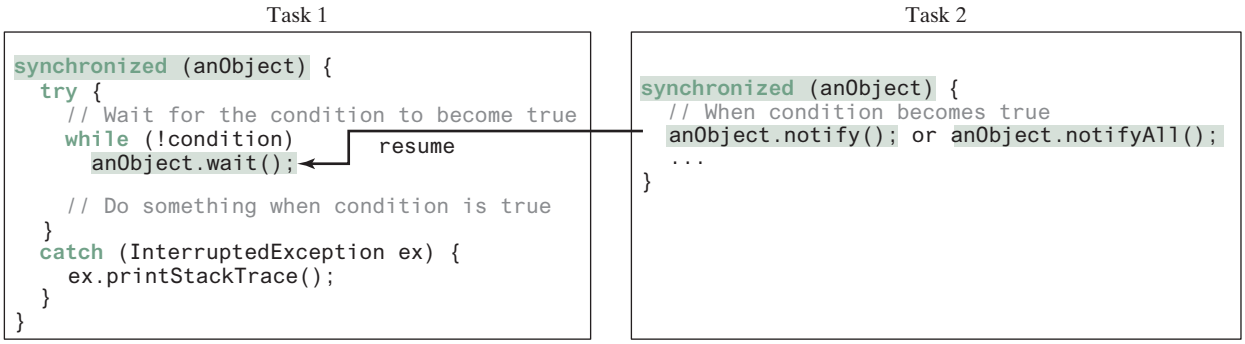


FIGURE 32.17 The `wait()`, `notify()`, and `notifyAll()` methods coordinate thread communication.

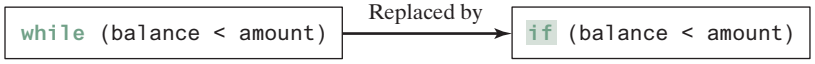
The `wait()`, `notify()`, and `notifyAll()` methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an `IllegalMonitorStateException` will occur.

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

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

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

32.9.2 What would happen if the `while` loop in line 58 of Listing 32.6 was changed to an `if` statement?



32.9.3 Why does the following class have a syntax error?

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

  public Test() throws InterruptedException {
    Thread thread = new Thread(this);
    thread.sleep(1000);
  }

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

32.9.4 What is a possible cause for `IllegalMonitorStateException`?

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

32.9.6 What is wrong in the following code?

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

32.10 Case Study: Producer/Consumer

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

Suppose that you use a buffer to store integers and that the buffer size is limited. The buffer provides the method `write(int)` to add an `int` value to the buffer and the method `read()` to read and delete an `int` value from the buffer. To synchronize the operations, use a lock with two conditions: `notEmpty` (i.e., the buffer is not empty) and `notFull` (i.e., the buffer is not full). When a task adds an `int` to the buffer, if the buffer is full, the task will wait for the `notFull` condition. When a task reads an `int` from the buffer, if the buffer is empty, the task will wait for the `notEmpty` condition. The interaction between the two tasks is shown in Figure 32.18.

Listing 32.7 presents the complete program. The program contains the `Buffer` class (lines 50–101) and two tasks for repeatedly adding and consuming numbers to and from the buffer (lines 16–47). The `write(int)` method (lines 62–79) adds an integer to the buffer. The `read()` method (lines 81–100) deletes and returns an integer from the buffer.

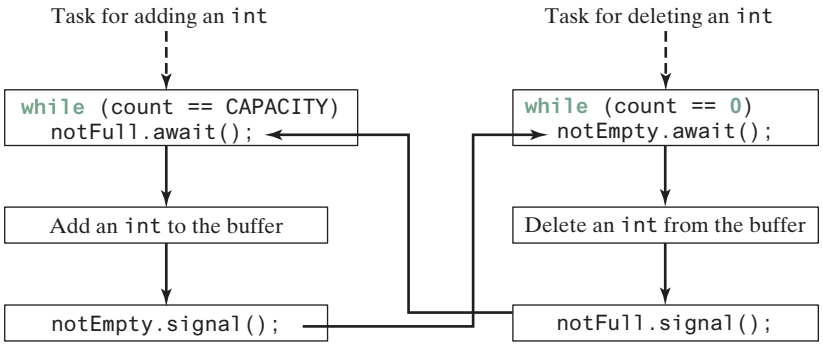


FIGURE 32.18 The conditions `notFull` and `notEmpty` are used to coordinate task interactions.

The buffer is actually a first-in, first-out queue (lines 52 and 53). The conditions `notEmpty` and `notFull` on the lock are created in lines 59 and 60. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the `wait()` and `notify()` methods to rewrite this example, you have to designate two objects as monitors.

LISTING 32.7 ConsumerProducer.java

```
1  import java.util.concurrent.*;
2  import java.util.concurrent.locks.*;
3
4  public class ConsumerProducer {
5    private static Buffer buffer = new Buffer();
6
7    public static void main(String[] args) {
8      // Create a thread pool with two threads
9      ExecutorService executor = Executors.newFixedThreadPool(2);
10     executor.execute(new ProducerTask());
11     executor.execute(new ConsumerTask());
12     executor.shutdown();
13   }
14
15   // A task for adding an int to the buffer
16   private static class ProducerTask implements Runnable {
17     public void run() {
18       try {
19         int i = 1;
20         while (true) {
21           System.out.println("Producer writes " + i);
22           buffer.write(i++); // Add a value to the buffer
23           // Put the thread into sleep
24           Thread.sleep((int)(Math.random() * 10000));
25         }
26       }
27       catch (InterruptedException ex) {
28         ex.printStackTrace();
29       }
30     }
31   }
32
33   // A task for reading and deleting an int from the buffer
34   private static class ConsumerTask implements Runnable {
35     public void run() {
```

create a buffer

create two threads

producer task

consumer task

```
36      try {
37        while (true) {
38          System.out.println("\t\t\tConsumer reads " + buffer.read());
39          // Put the thread into sleep
40          Thread.sleep((int)(Math.random() * 10000));
41        }
42      }
43      catch (InterruptedException ex) {
44        ex.printStackTrace();
45      }
46    }
47  }
48
49  // An inner class for buffer
50  private static class Buffer {
51    private static final int CAPACITY = 1; // buffer size
52    private java.util.LinkedList<Integer> queue =
53      new java.util.LinkedList<>();
54
55    // Create a new lock
56    private static Lock lock = new ReentrantLock();                    create a lock
57
58    // Create two conditions
59    private static Condition notEmpty = lock.newCondition();           create a condition
60    private static Condition notFull = lock.newCondition();            create a condition
61
62    public void write(int value) {
63      lock.lock(); // Acquire the lock                                 acquire the lock
64      try {
65        while (queue.size() == CAPACITY) {
66          System.out.println("Wait for notFull condition");
67          notFull.await();                                             wait for notFull
68        }
69
70        queue.offer(value);
71        notEmpty.signal(); // Signal notEmpty condition                 signal notEmpty
72      }
73      catch (InterruptedException ex) {
74        ex.printStackTrace();
75      }
76      finally {
77        lock.unlock(); // Release the lock                              release the lock
78      }
79    }
80
81    public int read() {
82      int value = 0;
83      lock.lock(); // Acquire the lock                                 acquire the lock
84      try {
85        while (queue.isEmpty()) {
86          System.out.println("\t\t\tWait for notEmpty condition");
87          notEmpty.await();                                            wait for notEmpty
88        }
89
90        value = queue.remove();
91        notFull.signal(); // Signal notFull condition                   signal notFull
92      }
93      catch (InterruptedException ex) {
94        ex.printStackTrace();
95      }
```

release the lock

```
96          finally {
97            lock.unlock(); // Release the lock
98            return value;
99          }
100       }
101     }
102   }
```

A sample run of the program is shown in Figure 32.19.

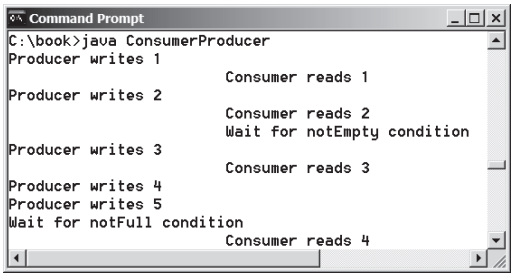


FIGURE 32.19 Locks and conditions are used for communications between the Producer and Consumer threads.

Check Point

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

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

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

Key Point

32.11 Blocking Queues

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

blocking queue

Queues and priority queues were introduced in Section 20.9. A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue. The `BlockingQueue` interface extends `java.util.Queue` and provides the synchronized `put` and `take` methods for adding an element to the tail of the queue and for removing an element from the head of the queue, as shown in Figure 32.20.

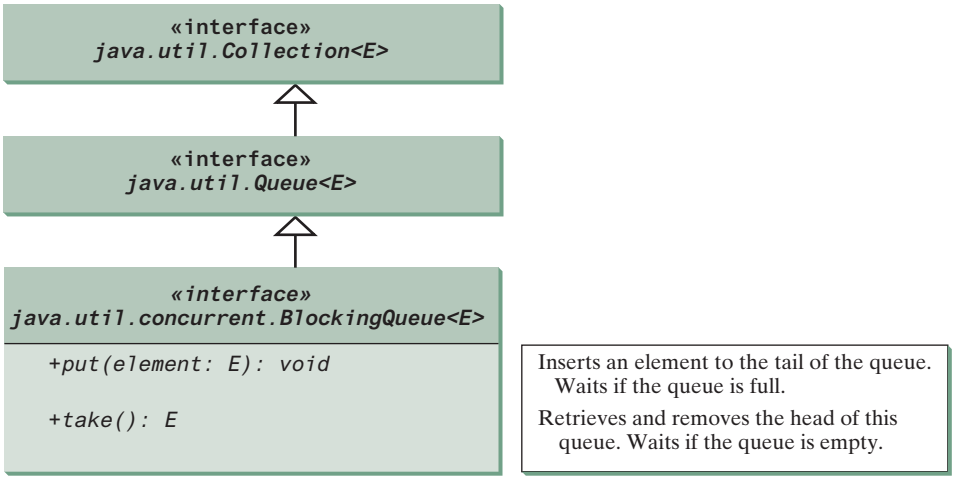


FIGURE 32.20 `BlockingQueue` is a subinterface of `Queue`.

Three concrete blocking queues—`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`—are provided in Java, as shown in Figure 32.21. All are in the `java.util.concurrent` package. `ArrayBlockingQueue` implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an `ArrayBlockingQueue`. `LinkedBlockingQueue` implements a blocking queue using a linked list. You can create an unbounded or bounded `LinkedBlockingQueue`. `PriorityBlockingQueue` is a priority queue. You can create an unbounded or bounded priority queue.

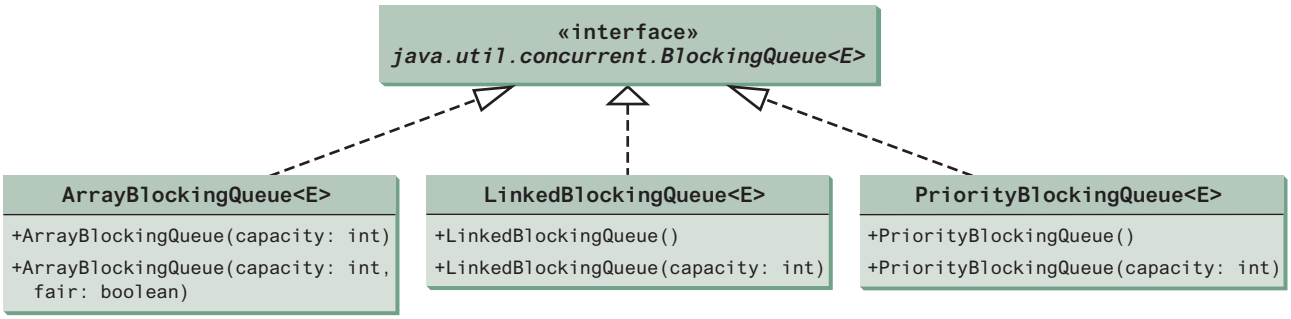


FIGURE 32.21 `ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue` are concrete blocking queues.

Note

The `put` method will never block an unbounded `LinkedBlockingQueue` or `PriorityBlockingQueue`.

unbounded queue

Listing 32.8 gives an example of using an `ArrayBlockingQueue` to simplify the Consumer/Producer example in Listing 32.10. Line 5 creates an `ArrayBlockingQueue` to store integers. The Producer thread puts an integer into the queue (line 22) and the Consumer thread takes an integer from the queue (line 38).

LISTING 32.8 ConsumerProducerUsingBlockingQueue.java

```
1  import java.util.concurrent.*;
2
3  public class ConsumerProducerUsingBlockingQueue {
4    private static ArrayBlockingQueue<Integer> buffer =
5      new ArrayBlockingQueue<>(2);                                  create a buffer
6
7    public static void main(String[] args) {
8      // Create a thread pool with two threads
9      ExecutorService executor = Executors.newFixedThreadPool(2);   create two threads
10     executor.execute(new ProducerTask());
11     executor.execute(new ConsumerTask());
12     executor.shutdown();
13   }
14
15   // A task for adding an int to the buffer
16   private static class ProducerTask implements Runnable {         producer task
17     public void run() {
18       try {
19         int i = 1;
20         while (true) {
21           System.out.println("Producer writes " + i);
22           buffer.put(i++); // Add any value to the buffer, say, 1   put
23           // Put the thread into sleep
```

```
24            Thread.sleep((int)(Math.random() * 10000));
25          }
26        }
27        catch (InterruptedException ex) {
28          ex.printStackTrace();
29        }
30      }
31    }
32
33    // A task for reading and deleting an int from the buffer
34    private static class ConsumerTask implements Runnable {
35      public void run() {
36        try {
37          while (true) {
38            System.out.println("\t\t\tConsumer reads " + buffer.take());
39            // Put the thread into sleep
40            Thread.sleep((int)(Math.random() * 10000));
41          }
42        }
43        catch (InterruptedException ex) {
44          ex.printStackTrace();
45        }
46      }
47    }
48  }
```

Consumer task

take

In Listing 32.7, you used locks and conditions to synchronize the Producer and Consumer threads. This program does not use locks and conditions because synchronization is already implemented in `ArrayBlockingQueue`.

32.11.1 What is a blocking queue? What blocking queues are supported in Java?

32.11.2 What method do you use to add an element to an `ArrayBlockingQueue`? What happens if the queue is full?

32.11.3 What method do you use to retrieve an element from an `ArrayBlockingQueue`? What happens if the queue is empty?

32.12 Semaphores

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

semaphore

In computer science, a *semaphore* is an object that controls the access to a common resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown in Figure 32.22.

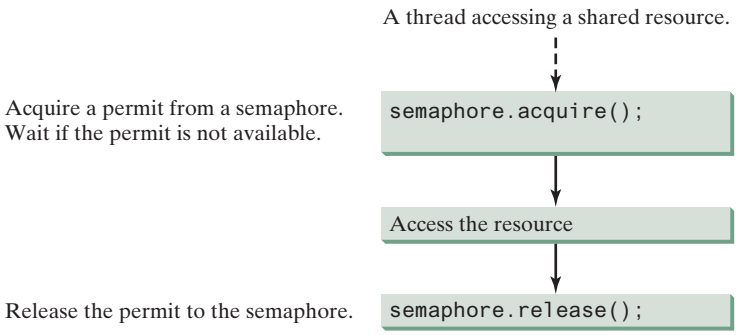


FIGURE 32.22 A limited number of threads can access a shared resource controlled by a semaphore.

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure 32.23. A task acquires a permit by invoking the semaphore's `acquire()` method and releases the permit by invoking the semaphore's `release()` method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by `1`. Once a permit is released, the total number of available permits in a semaphore is increased by `1`.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

FIGURE 32.23 The `Semaphore` class contains the methods for accessing a semaphore.

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

LISTING 32.9 New Account Inner Class

```
1  // An inner class for Account
2  private static class Account {
3    // Create a semaphore
4    private static Semaphore semaphore = new Semaphore(1);          create a semaphore
5    private int balance = 0;
6
7    public int getBalance() {
8      return balance;
9    }
10
11   public void deposit(int amount) {
12     try {
13       semaphore.acquire(); // Acquire a permit                    acquire a permit
14       int newBalance = balance + amount;
15
16       // This delay is deliberately added to magnify the
17       // data-corruption problem and make it easy to see
18       Thread.sleep(5);
19
20       balance = newBalance;
21     }
22     catch (InterruptedException ex) {
23     }
24     finally {
25       semaphore.release(); // Release a permit                    release a permit
26     }
27   }
28 }
```

A semaphore with one permit is created in line 4. A thread first acquires a permit when executing the deposit method in line 13. After the balance is updated, the thread releases the permit in line 25. It is a good practice to always place the `release()` method in the `finally` clause to ensure that the permit is finally released even in the case of exceptions.

32.12.1 What are the similarities and differences between a lock and a semaphore?

32.12.2 How do you create a semaphore that allows three concurrent threads? How do you acquire a semaphore? How do you release a semaphore?

32.13 Avoiding Deadlocks

Deadlocks can be avoided by using a proper resource ordering.

deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause a *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown in Figure 32.24. Thread 1 has acquired a lock on `object1`, and Thread 2 has acquired a lock on `object2`. Now Thread 1 is waiting for the lock on `object2`, and Thread 2 for the lock on `object1`. Each thread waits for the other to release the lock it needs and until that happens, neither can continue to run.

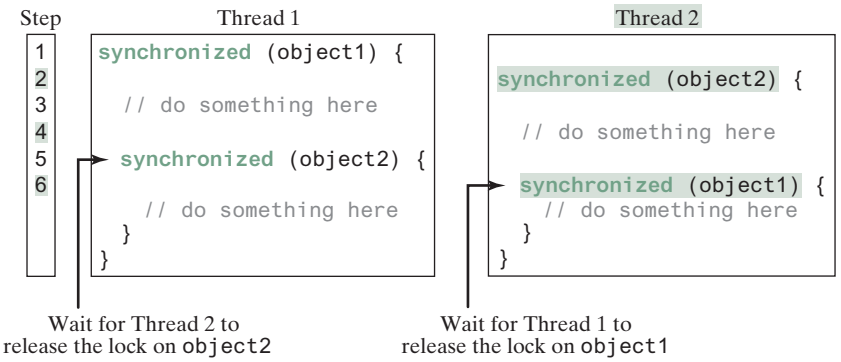


FIGURE 32.24 Thread 1 and Thread 2 are deadlocked.

resource ordering

Deadlock is easily avoided by using a simple technique known as *resource ordering*. With this technique, you assign an order to all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For example in Figure 32.24, suppose the objects are ordered as `object1` and `object2`. Using the resource-ordering technique, Thread 2 must acquire a lock on `object1` first, then on `object2`. Once Thread 1 acquires a lock on `object1`, Thread 2 has to wait for a lock on `object1`. Thus, Thread 1 will be able to acquire a lock on `object2` and no deadlock will occur.

32.13.1 What is a deadlock? How can you avoid deadlock?

32.14 Thread States

A thread state indicates the status of thread.

Tasks are executed in threads. Threads can be in one of the five states: New, Ready, Running, Blocked, or Finished (see Figure 32.25).

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

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

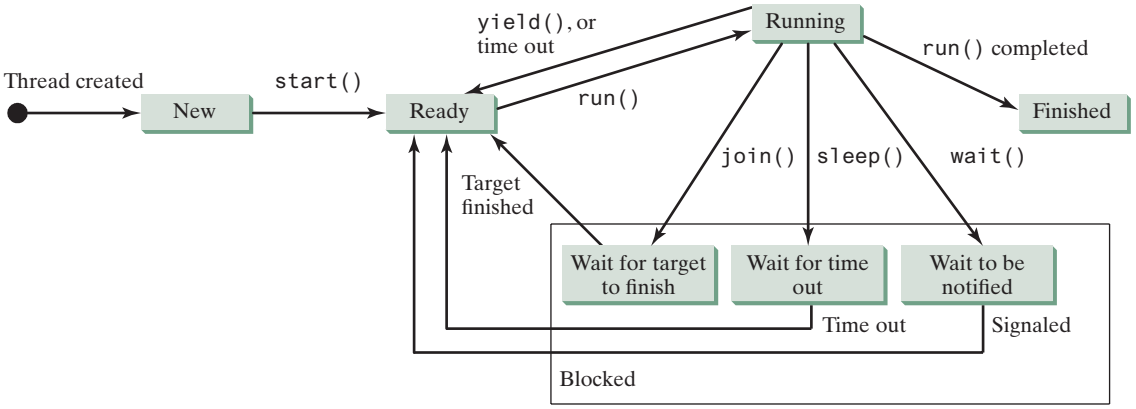


FIGURE 32.25 A thread can be in one of the five states: New, Ready, Running, Blocked, or Finished.

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

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

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

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

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

32.15 Synchronized Collections

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

The classes in the Java Collections Framework are not thread-safe; that is, their contents may become corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or by using synchronized collections.

synchronized collection

The `Collections` class provides six static methods for wrapping a collection into a synchronized version, as shown in Figure 32.26. The collections created using these methods are called *synchronization wrappers*.

synchronization wrapper

java.util.Collections	
`+synchronizedCollection(c: Collection): Collection`	Returns a synchronized collection.
`+synchronizedList(list: List): List`	Returns a synchronized list from the specified list.
`+synchronizedMap(m: Map): Map`	Returns a synchronized map from the specified map.
`+synchronizedSet(s: Set): Set`	Returns a synchronized set from the specified set.
`+synchronizedSortedMap(s: SortedMap): SortedMap`	Returns a synchronized sorted map from the specified sorted map.
`+synchronizedSortedSet(s: SortedSet): SortedSet`	Returns a synchronized sorted set.

FIGURE 32.26 You can obtain synchronized collections using the methods in the `Collections` class.

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

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

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

Note

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

fail-fast

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

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

synchronized (hashSet) { // Must synchronize it
  Iterator iterator = hashSet.iterator();

  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

Failure to do so may result in nondeterministic behavior, such as a `ConcurrentModificationException`.

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

32.15.2 Explain why an iterator is fail-fast.

32.16 Parallel Programming

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

Section 7.12 introduced the `Arrays.sort` and `Arrays.parallelSort` method for sorting an array. The `parallelSort` method utilizes multiple processors to reduce sort time. Chapter 22 introduced parallel streams for executing stream operations in parallel to speed up processing using multiple processors. The parallel processing are implemented using the Fork/Join Framework. This section, introduces the new Fork/Join Framework so you can write own code for parallel programming.

Fork/Join Framework

The *Fork/Join Framework* is illustrated in Figure 32.27 (the diagram resembles a fork, hence its name). A problem is divided into nonoverlapping subproblems, which can be solved independently in parallel. The solutions to all subproblems are then joined to obtain an overall solution for the problem. This is the parallel implementation of the divide-and-conquer approach. In JDK 7's Fork/Join Framework, a *fork* can be viewed as an independent task that runs on a thread.

JDK 7 feature

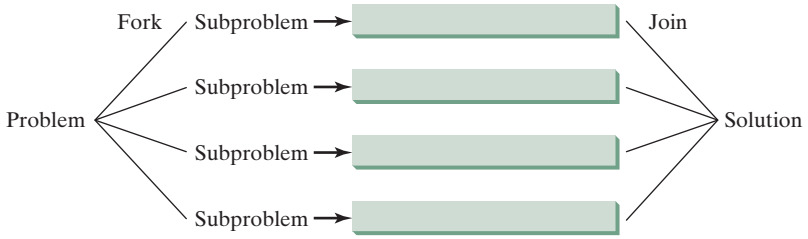


FIGURE 32.27 The nonoverlapping subproblems are solved in parallel.

The framework defines a task using the `ForkJoinTask` class, as shown in Figure 32.28 and executes a task in an instance of `ForkJoinPool`, as shown in Figure 32.29.

`ForkJoinTask`
`ForkJoinPool`

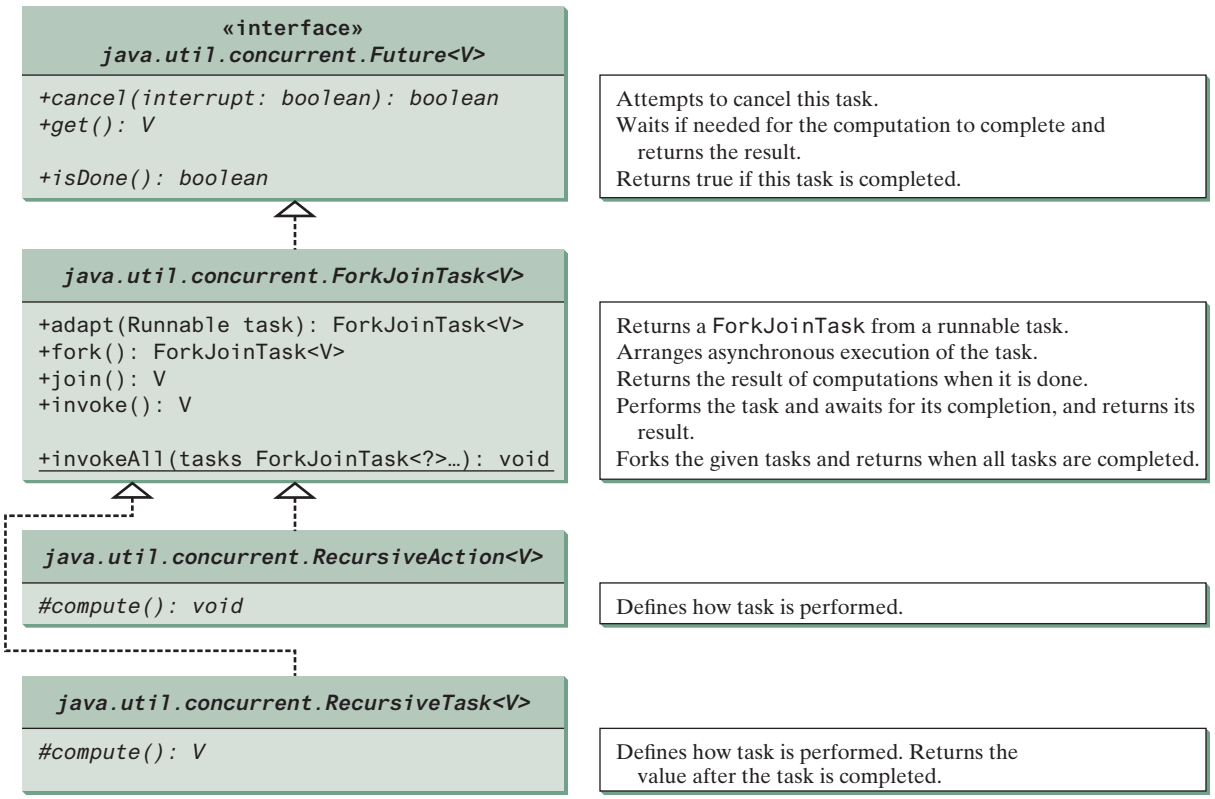


FIGURE 32.28 The `ForkJoinTask` class defines a task for asynchronous execution.

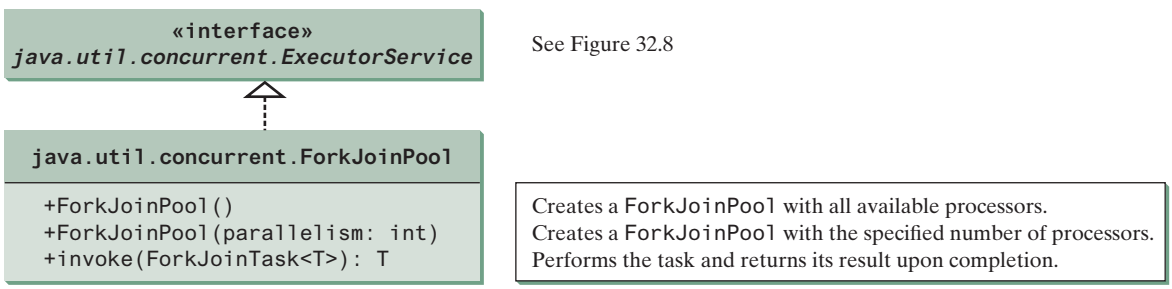


FIGURE 32.29 The `ForkJoinPool` executes Fork/Join tasks.

`ForkJoinTask` is the abstract base class for tasks. A `ForkJoinTask` is a thread-like entity, but it is much lighter than a normal thread because huge numbers of tasks and subtasks can be executed by a small number of actual threads in a `ForkJoinPool`. The tasks are primarily coordinated using `fork()` and `join()`. Invoking `fork()` on a task arranges asynchronous execution and invoking `join()` waits until the task is completed. The `invoke()` and `invokeAll(tasks)` methods implicitly invoke `fork()` to execute the task and `join()` to wait for the tasks to complete and return the result, if any. Note the static method `invokeAll` takes a variable number of `ForkJoinTask` arguments using the `...` syntax, which is introduced in Section 7.9.

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

`RecursiveAction`
`RecursiveTask`

We now use a merge sort to demonstrate how to develop parallel programs using the Fork/Join Framework. The merge sort algorithm (introduced in Section 25.3) divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm merges them. Listing 32.10 gives a parallel implementation of the merge sort algorithm and compares its execution time with a sequential sort.

LISTING 32.10 ParallelMergeSort.java

```
1  import java.util.concurrent.RecursiveAction;
2  import java.util.concurrent.ForkJoinPool;
3
4  public class ParallelMergeSort {
5    public static void main(String[] args) {
6      final int SIZE = 7000000;
7      int[] list1 = new int[SIZE];
8      int[] list2 = new int[SIZE];
9
10     for (int i = 0; i < list1.length; i++)
11       list1[i] = list2[i] = (int)(Math.random() * 10000000);
12
13     long startTime = System.currentTimeMillis();
14     parallelMergeSort(list1); // Invoke parallel merge sort
15     long endTime = System.currentTimeMillis();
16     System.out.println("\nParallel time with "
17       + Runtime.getRuntime().availableProcessors() +
18       " processors is " + (endTime - startTime) + " milliseconds");
19
20     startTime = System.currentTimeMillis();
21     MergeSort.mergeSort(list2); // MergeSort is in Listing 23.5
22     endTime = System.currentTimeMillis();
23     System.out.println("\nSequential time is " +
24       (endTime - startTime) + " milliseconds");
25   }
26
27   public static void parallelMergeSort(int[] list) {
28     RecursiveAction mainTask = new SortTask(list);
29     ForkJoinPool pool = new ForkJoinPool();
30     pool.invoke(mainTask);
31   }
32
33   private static class SortTask extends RecursiveAction {
34     private final int THRESHOLD = 500;
```

invoke parallel sort

invoke sequential sort

create a ForkJoinTask
create a ForkJoinPool
execute a task

define concrete ForkJoinTask

```
35      private int[] list;
36
37      SortTask(int[] list) {
38        this.list = list;
39      }
40
41      @Override
42      protected void compute() {                                    perform the task
43        if (list.length < THRESHOLD)
44          java.util.Arrays.sort(list);                            sort a small list
45        else {
46          // Obtain the first half
47          int[] firstHalf = new int[list.length / 2];             split into two parts
48          System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
49
50          // Obtain the second half
51          int secondHalfLength = list.length - list.length / 2;
52          int[] secondHalf = new int[secondHalfLength];
53          System.arraycopy(list, list.length / 2,
54            secondHalf, 0, secondHalfLength);
55
56          // Recursively sort the two halves
57          invokeAll(new SortTask(firstHalf),                      solve each part
58            new SortTask(secondHalf));
59
60          // Merge firstHalf with secondHalf into list
61          MergeSort.merge(firstHalf, secondHalf, list);           merge two parts
62        }
63      }
64    }
65  }
```

```
Parallel time with two processors is 2829 milliseconds
Sequential time is 4751 milliseconds
```

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

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

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

The `MergeSort` class is defined in Listing 23.5. The program invokes `MergeSort.merge` to merge two sorted sublists (line 61). The program also invokes `MergeSort.mergeSort` (line 21) to sort a list using merge sort sequentially. You can see that the parallel sort is much faster than the sequential sort.

Note the loop for initializing the list can also be parallelized. However, you should avoid using `Math.random()` in the code because it is synchronized and cannot be executed in parallel (see Programming Exercise 32.12). The `parallelMergeSort` method only sorts an

array of `int` values, but you can modify it to become a generic method (see Programming Exercise 32.13).

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

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

Listing 32.11 develops a parallel method that finds the maximal number in a list.

LISTING 32.11 ParallelMax.java

```
1  import java.util.concurrent.*;
2
3  public class ParallelMax {
4    public static void main(String[] args) {
5      // Create a list
6      final int N = 9000000;
7      int[] list = new int[N];
8      for (int i = 0; i < list.length; i++)
9        list[i] = i;
10
11     long startTime = System.currentTimeMillis();
12     System.out.println("\nThe maximal number is " + max(list));
13     long endTime = System.currentTimeMillis();
14     System.out.println("The number of processors is " +
15       Runtime.getRuntime().availableProcessors());
16     System.out.println("Time is " + (endTime - startTime)
17       + " milliseconds");
18   }
19
20   public static int max(int[] list) {
21     RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
22     ForkJoinPool pool = new ForkJoinPool();
23     return pool.invoke(task);
24   }
25
26   private static class MaxTask extends RecursiveTask<Integer> {
27     private final static int THRESHOLD = 1000;
28     private int[] list;
29     private int low;
30     private int high;
31
32     public MaxTask(int[] list, int low, int high) {
33       this.list = list;
34       this.low = low;
35       this.high = high;
36     }
37
38     @Override
39     public Integer compute() {
40       if (high - low < THRESHOLD) {
41         int max = list[0];
42         for (int i = low; i < high; i++)
43           if (list[i] > max)
44             max = list[i];
45         return new Integer(max);
```

invoke max

create a ForkJoinTask
create a ForkJoinPool
execute a task

define concrete ForkJoinTask

perform the task

solve a small problem

```
46        }
47        else {
48          int mid = (low + high) / 2;
49          RecursiveTask<Integer> left = new MaxTask(list, low, mid);
50          RecursiveTask<Integer> right = new MaxTask(list, mid, high);
51
52          right.fork();
53          left.fork();
54          return new Integer(Math.max(left.join().intValue(),
55            right.join().intValue()));
56        }
57      }
58    }
59  }
```

split into two parts

fork right

fork left

join tasks

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

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

32.16.1 How do you define a `ForkJoinTask`? What are the differences between `RecursiveAction` and `RecursiveTask`?

32.16.2 How do you tell the system to execute a task?

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

32.16.4 How do you create a `ForkJoinPool`? How do you place a task into a `ForkJoinPool`?

Key Terms

condition 32-18
deadlock 32-30
fail-fast 32-32
fairness policy 32-17
Fork/Join Framework 32-32
lock 32-16
monitor 32-22
multithreading 32-2
race condition 32-15
semaphore 32-28
synchronization wrapper 32-31
synchronized block 32-16
thread 32-2
thread-safe 32-15

Chapter Summary

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

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

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

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

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

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

7. The blocking queues (`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`) provided in the Java Collections Framework provide automatical synchronization for the access to a queue.

8. You can use semaphores to restrict the number of concurrent accesses to a shared resource.

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

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

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

Sections 32.1–32.5

***32.1** (*Revise Listing 32.1*) Rewrite Listing 32.1 to display the output in a text area, as shown in Figure 32.30.

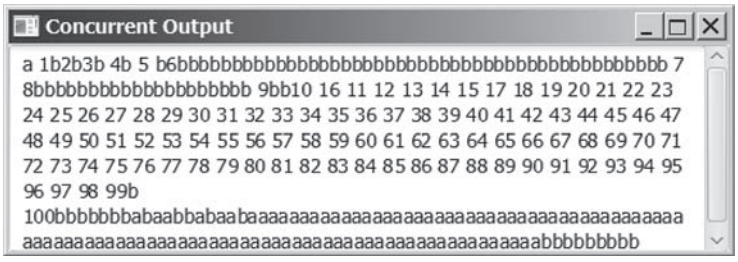


Figure 32.30 The output from three threads is displayed in a text area.

32.2 (*Racing cars*) Rewrite Programming Exercise 15.29 using a thread to control car racing. Compare the program with Programming Exercise 15.29 by setting the delay time to 10 in both the programs. Which one runs the animation faster?

32.3 (*Raise flags*) Rewrite Listing 15.13 using a thread to animate a flag being raised. Compare the program with Listing 15.13 by setting the delay time to 10 in both programs. Which one runs the animation faster?

Sections 32.8–32.12

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

32.5 (*Display a running fan*) Rewrite Programming Exercise 15.28 using a thread to control the fan animation.

32.6 (*Bouncing balls*) Rewrite Listing 15.17, BallPane.java using a thread to animate bouncing ball movements.

32.7 (*Control a clock*) Rewrite Programming Exercise 15.32 using a thread to control the clock animation.

32.8 (*Account synchronization*) Rewrite Listing 32.6, ThreadCooperation.java, using the object's `wait()` and `notifyAll()` methods.

32.9 (*Demonstrate* `ConcurrentModificationException`) The iterator is *fail-fast*. Write a program to demonstrate it by creating two threads that concurrently access and modify a set. The first thread creates a hash set filled with numbers and adds a new number to the set every second. The second thread obtains an iterator for the set and traverses the set back and forth through the iterator every second. You will receive a `ConcurrentModificationException` because the underlying set is being modified in the first thread while the set in the second thread is being traversed.

***32.10** (*Use synchronized sets*) Using synchronization, correct the problem in the preceding exercise so that the second thread does not throw a `ConcurrentModificationException`.

Section 32.15

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

Section 32.18

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

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

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

32.13 (*Generic parallel merge sort*) Revise Listing 32.10, ParallelMergeSort.java, to define a generic `parallelMergeSort` method as follows:

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

***32.14** (*Parallel quick sort*) Implement the following method in parallel to sort a list using quick sort (see Listing 23.7):

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

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

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

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

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

***32.16** (*Parallel matrix addition*) Programming Exercise 8.5 describes how to perform matrix addition. Suppose you have multiple processors, so you can speed up the matrix addition. Implement the following method in parallel:

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

Write a test program that measures the execution time for adding two 2,000 $\times$ 2,000 matrices using the parallel method and sequential method, respectively.

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

```
public static double[][] parallelMultiplyMatrix(
  double[][] a, double[][] b)
```

Write a test program that measures the execution time for multiplying two 2,000 $\times$ 2,000 matrices using the parallel method and sequential method, respectively.

***32.18** (*Parallel Eight Queens*) Revise Listing 22.11, EightQueens.java, to develop a parallel algorithm that finds all solutions for the Eight Queens problem. (*Hint*: Launch eight subtasks, each of which places the queen in a different column in the first row.)

Comprehensive

****32.19** (*Sorting animation*) Write an animation for selection sort, insertion sort, and bubble sort, as shown in Figure 32.31. Create an array of integers 1, 2, . . . , 50. Shuffle it randomly. Create a pane to display the array in a histogram. You should invoke each sort method in a separate thread. Each algorithm uses two nested loops. When the algorithm completes an iteration in the outer loop, put the thread to sleep for 0.5 seconds and redisplay the array in the histogram. Color the last bar in the sorted subarray.

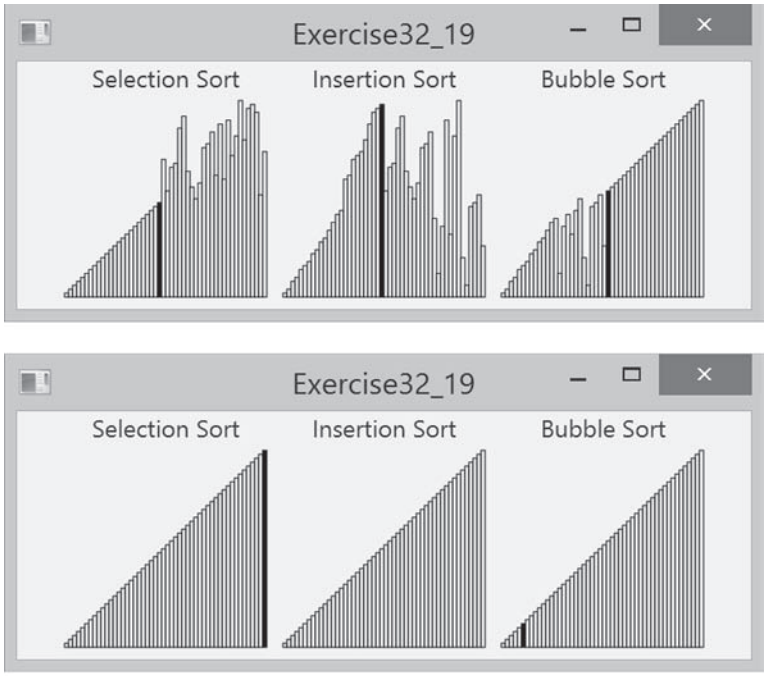
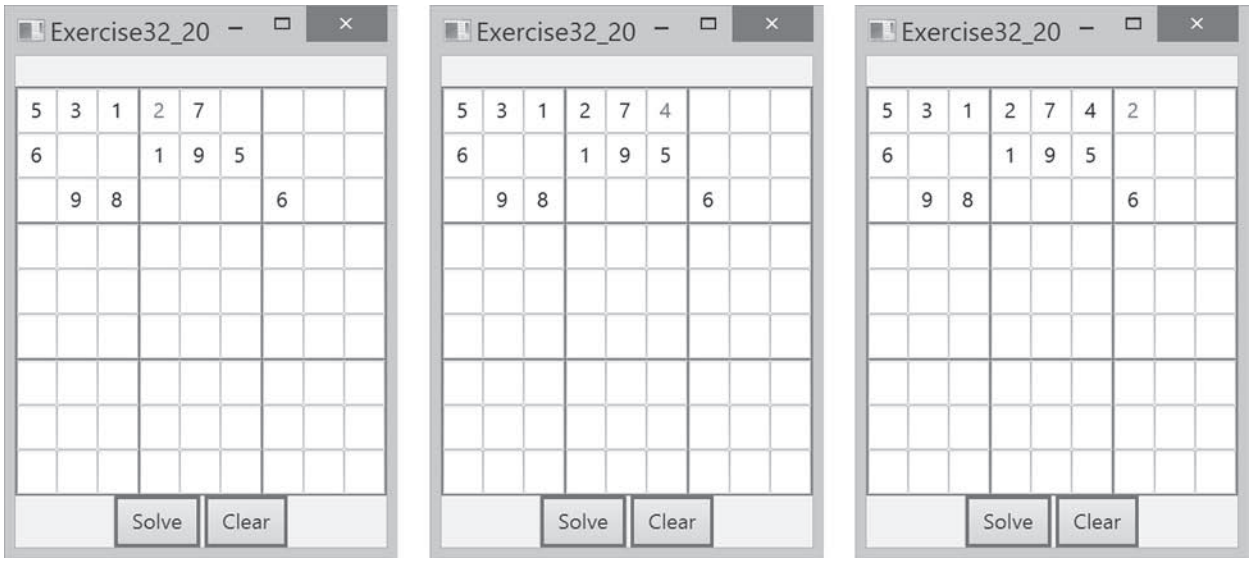


FIGURE 32.31 Three sorting algorithms are illustrated in the animation.

*****32.20** (*Sudoku search animation*) Modify Programming Exercise 22.21 to display the intermediate results of the search. Figure 32.32 gives a snapshot of an animation in progress with number 2 placed in the cell in Figure 32.32a, number 3 placed in the cell in Figure 32.32b, and number 3 placed in the cell in Figure 32.32c. The animation displays all the search steps.



(a) (b) (c)

FIGURE 32.32 The intermediate search steps are displayed in the animation for the Sudoku problem.

32.21 (*Combine colliding bouncing balls*) Rewrite Programming Exercise 20.5 using a thread to animate bouncing ball movements.

***32.22** (*Eight Queens animation*) Modify Listing 22.11, EightQueens.java, to display the intermediate results of the search. As shown in Figure 32.33, the current row being searched is highlighted. Every one second, a new state of the chess board is displayed.

FIGURE 32.33 The intermediate search steps are displayed in the animation for the Eight Queens problem.