CHAPTER

30

MULTITHREADING AND PARALLEL PROGRAMMING

Objectives

- To get an overview of multithreading (§30.2).
- To develop task classes by implementing the **Runnable** interface (§30.3).
- To create threads to run tasks using the **Thread** class (§30.3).
- To control threads using the methods in the **Thread** class (§30.4).
- To control animations using threads and use **Platform.runLater** to run the code in the application thread (§30.5).
- To execute tasks in a thread pool (§30.6).
- To use synchronized methods or blocks to synchronize threads to avoid race conditions (§30.7).
- To synchronize threads using locks (§30.8).
- To facilitate thread communications using conditions on locks (§§30.9 and 30.10).
- To use blocking queues (**ArrayBlockingQueue**, **LinkedBlockingQueue**, **PriorityBlockingQueue**) to synchronize access to a queue (§30.11).
- To restrict the number of concurrent tasks that access a shared resource using semaphores (§30.12).
- To use the resource-ordering technique to avoid deadlocks (§30.13).
- To describe the life cycle of a thread (§30.14).
- To create synchronized collections using the static methods in the **Collections** class (§30.15).
- To develop parallel programs using the Fork/Join Framework (§30.16).

30.1 Introduction

Key Point

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.

30.2 Thread Concepts

Key Point

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 30.1a.

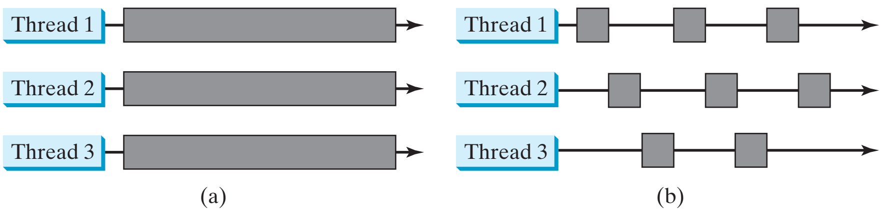


FIGURE 30.1 (a) Here multiple threads are running on multiple CPUs. (b) Here multiple threads share a single CPU.

time sharing

In single-processor systems, as shown in Figure 30.1b, 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.

Check Point

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

30.2 What is a runnable object? What is a thread?

30.3 Creating Tasks and Threads

Key Point

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 30.2a.

java.lang.Runnable ◁- - - - TaskClass

```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

(a)

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

(b)

FIGURE 30.2 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 30.2b outlines the major steps for creating a task, a thread, and starting the thread.

start a thread

Listing 30.1 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 30.3 shows a sample run of the program.

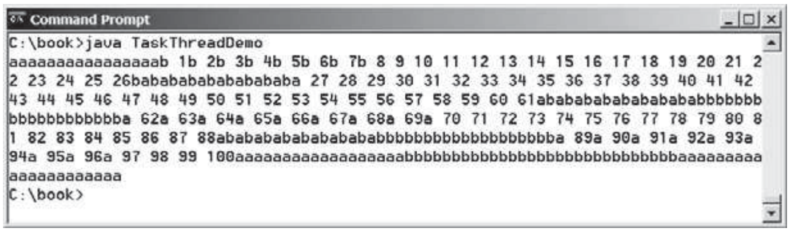


FIGURE 30.3 Tasks **printA**, **printB**, and **print100** are executed simultaneously to display the letter **a** 100 times, the letter **b** 100 times, and the numbers from 1 to 100.

LISTING 30.1 TaskThreadDemo.java

create tasks
create threads
start threads
task class
run
task class
run

```java
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

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

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

// The task for printing a character a specified number of times
class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The number of times to repeat

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

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

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

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

  @Override /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}
```

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

effect of concurrency

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

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

Important Note

run() method

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.

Check Point

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

30.4 What would happen if you replace the `start()` method with the `run()` method in lines 14–16 in Listing 30.1?

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

Replaced by

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

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

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

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

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

(a)

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

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

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

(b)

30.4 The Thread Class

Key Point

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

Figure 30.4 shows the class diagram for the **Thread** class.

«interface»
java.lang.Runnable

java.lang.Thread

java.lang.Thread	
`+Thread()`	Creates an empty thread.
`+Thread(task: Runnable)`	Creates a thread for a specified task.
`+start(): void`	Starts the thread that causes the `run()` method to be invoked by the JVM.
`+isAlive(): boolean`	Tests whether the thread is currently running.
`+setPriority(p: int): void`	Sets priority `p` (ranging from 1 to 10) for this thread.
`+join(): void`	Waits for this thread to finish.
`+sleep(millis: long): void`	Puts a thread to sleep for a specified time in milliseconds.
`+yield(): void`	Causes a thread to pause temporarily and allow other threads to execute.
`+interrupt(): void`	Interrupts this thread.

FIGURE 30.4 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 30.5a, and then create an object from the class and invoke its **start** method in a client program to start the thread, as shown in Figure 30.5b.

separating task from thread

`java.lang.Thread` ◁— `CustomThread`

```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }

  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
```

(a)

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create a thread
    CustomThread thread1 = new CustomThread(...);

    // Start a thread
    thread1.start();
    ...

    // Create another thread
    CustomThread thread2 = new CustomThread(...);

    // Start a thread
    thread2.start();
  }
  ...
}
```

(b)

FIGURE 30.5 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 has stopped.

deprecated method

You can use the `yield()` method to temporarily release time for other threads. For example, suppose you modify the code in the `run()` method in lines 53–57 for `PrintNum` in Listing 30.1 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 you modify the code in lines 53–57 in Listing 30.1, 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

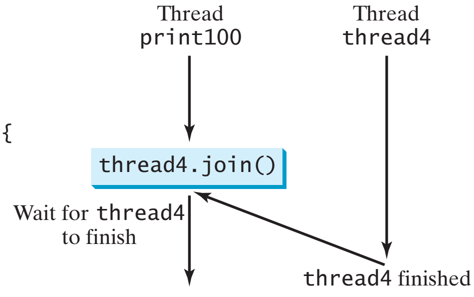
```
public void run() {
  while (...) {
    try {
      ...
      Thread.sleep(sleepTime);
    }
    catch (InterruptedException ex)  {
      ex.printStackTrace();
    }
  }
}
```

(b) Incorrect

join()

You can use the `join()` method to force one thread to wait for another thread to finish. For example, suppose you modify the code in lines 53–57 in Listing 30.1 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.

setPriority(int)

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`.

round-robin scheduling

The JVM always picks the currently runnable thread with the highest priority. A lower-priority thread can run only when no higher-priority threads are running. If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue. This is called *round-robin scheduling*. For example, suppose you insert the following code in line 16 in Listing 30.1:

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

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

Tip

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

contention or starvation

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.

Check Point

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

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

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

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

30.5 Case Study: Flashing Text

You can use a thread to control an animation.

Key Point

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 30.2 gives an example that displays flashing text on a label, as shown in Figure 30.6.

FIGURE 30.6 The text "Welcome" blinks.

LISTING 30.2 FlashText.java

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class FlashText extends Application {
  private String text = "";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    Label lblText = new Label("Programming is fun");
    pane.getChildren().add(lblText);

    new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          while (true) {
            if (lblText.getText().trim().length() == 0)
              text = "Welcome";
            else
              text = "";

            Platform.runLater(new Runnable() { // Run from JavaFX GUI
              @Override
              public void run() {
                lblText.setText(text);
              }
            });

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

```

create a label
label in a pane
create a thread
run thread
change text
Platform.runLater
update GUI
sleep

```
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.

Platform.runLater

Invoking **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();
```

Check Point

30.9 What causes the text to flash?

30.10 Is an instance of **FlashText** a runnable object?

30.11 What is the purpose of using **Platform.runLater**?

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

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

30.13 What happens if line 34 (**Thread.sleep(200)**) is not used?

30.6 Thread Pools

Key Point

A thread pool can be used to execute tasks efficiently.

In Section 30.3, 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(task);
new Thread(task).start();
```

This approach is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance. Using a *thread pool* is an ideal way to manage the number of tasks executing concurrently. Java provides the `Executor` interface for executing tasks in a thread pool and the `ExecutorService` interface for managing and controlling tasks. `ExecutorService` is a subinterface of `Executor`, as shown in Figure 30.7.

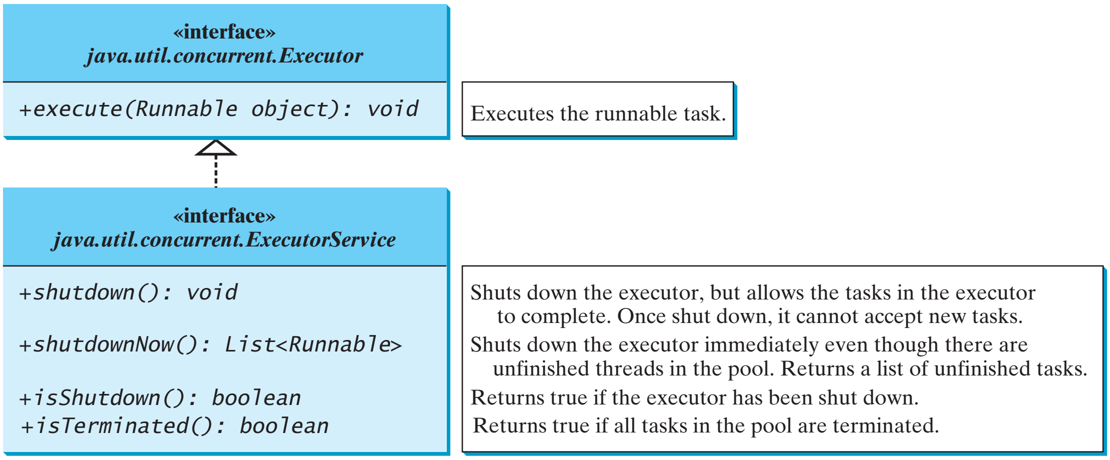


FIGURE 30.7 The `Executor` interface executes threads, and the `ExecutorService` subinterface manages threads.

To create an `Executor` object, use the static methods in the `Executors` class, as shown in Figure 30.8. 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.

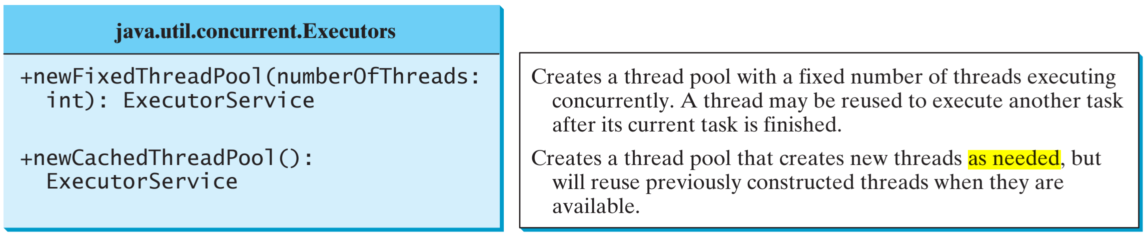


FIGURE 30.8 The `Executors` class provides static methods for creating `Executor` objects.

Listing 30.3 shows how to rewrite Listing 30.1 using a thread pool.

LISTING 30.3 ExecutorDemo.java

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

public class ExecutorDemo {
```

create executor

submit task

shut down executor

```
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  }
```

Line 6 creates a thread pool executor with a total of three threads maximum. Classes **PrintChar** and **PrintNum** are defined in Listing 30.1. 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 that you replace line 6 with

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

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

Suppose you replace line 6 with

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

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

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

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

Check Point

30.14 What are the benefits of using a thread pool?

30.15 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?

30.7 Thread Synchronization

Key Point

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 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 30.9. The program is given in Listing 30.4.

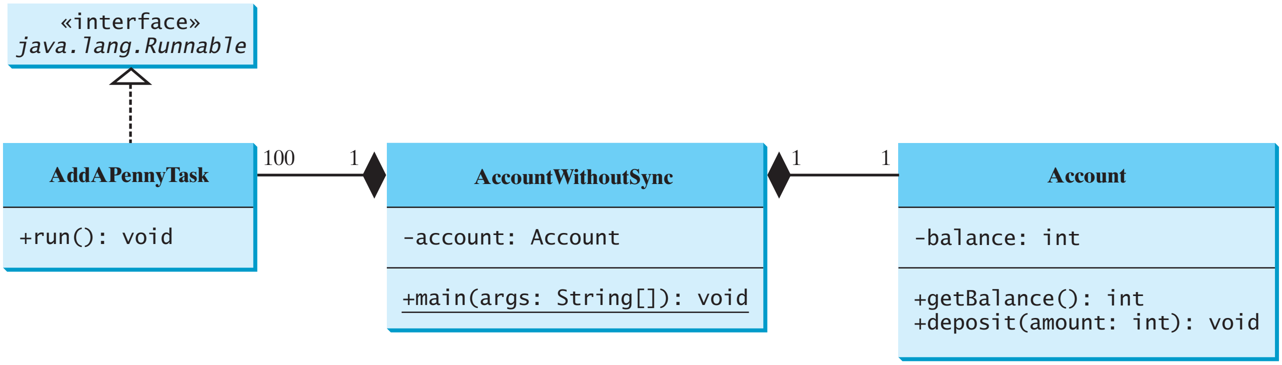


FIGURE 30.9 `AccountWithoutSync` contains an instance of `Account` and 100 threads of `AddAPennyTask`.

LISTING 30.4 AccountWithoutSync.java

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

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

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

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

    executor.shutdown();

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

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

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

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

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      int newBalance = balance + amount;

      // This delay is deliberately added to magnify the
```

create executor

submit task

shut down executor

wait for all tasks to terminate

```
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 the thread is terminated.

The balance of the account is initially **0** (line 32). When all the threads are finished, the balance should be **100** but the output is unpredictable. As can be seen in Figure 30.10, 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.

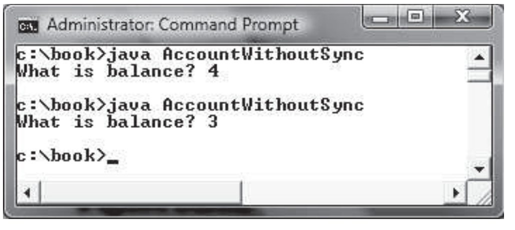


FIGURE 30.10 The **AccountWithoutSync** program causes data inconsistency.

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 30.11.

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 30.11 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

30.7.1 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 30.4 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 30.4. 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 30.12.

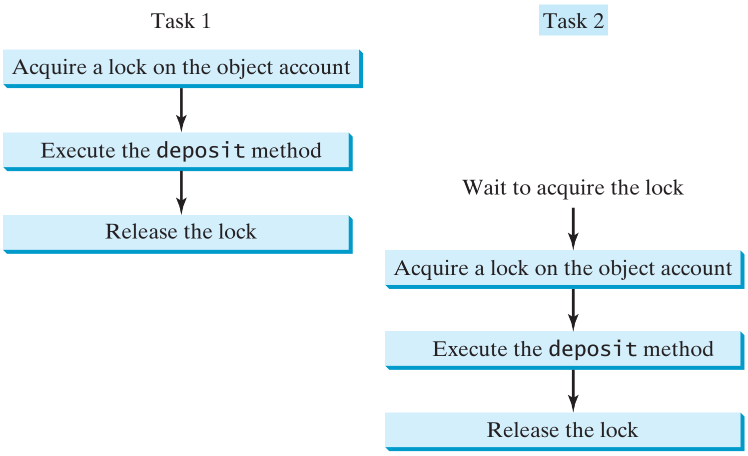


FIGURE 30.12 Task 1 and Task 2 are synchronized.

30.7.2 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 synchronized block

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

```
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 30.4 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)

Check Point

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

30.17 Suppose you place the statement in line 26 of Listing 30.4 inside a synchronized block to avoid race conditions, as follows:

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

Will it work?

30.8 Synchronization Using Locks

Key Point

Locks and conditions can be explicitly used to synchronize threads.

Recall that in Listing 30.4, 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 30.13. A lock may also use the **newCondition()** method to create any number of **Condition** objects, which can be used for thread communications.

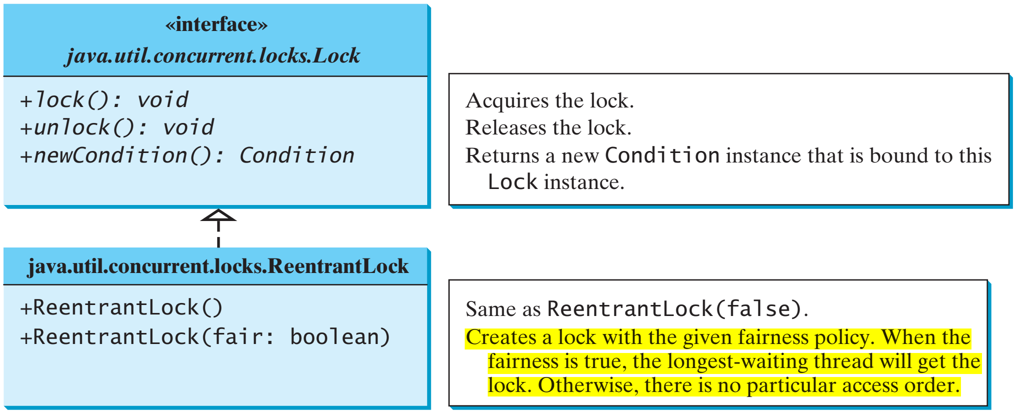


FIGURE 30.13 The `ReentrantLock` class implements the `Lock` interface to represent a lock.

`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 30.5 revises the program in Listing 30.7 to synchronize the account modification using explicit locks.

LISTING 30.5 AccountWithSyncUsingLock.java

package for locks

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

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

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

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

    executor.shutdown();

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

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

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

```

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 30.5 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.

Check Point

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

30.9 Cooperation among Threads

Key Point

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 30.14. 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.

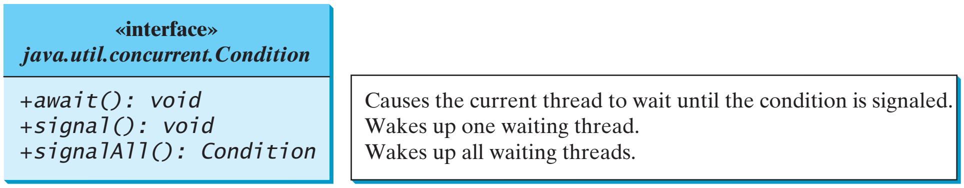


FIGURE 30.14 The `Condition` interface defines the methods for performing synchronization.

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

thread cooperation example

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 30.15.

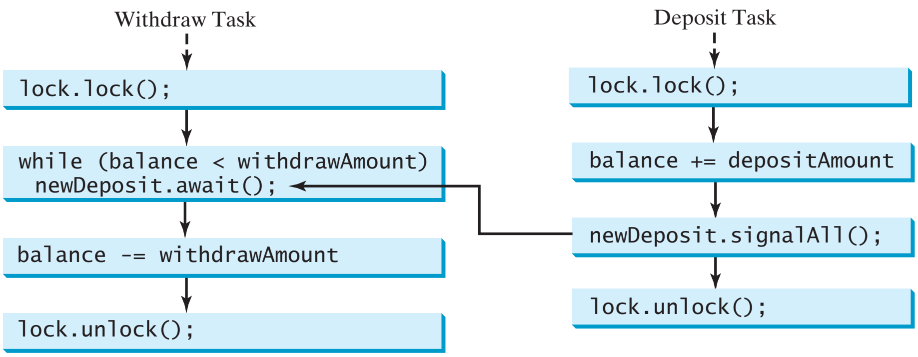


FIGURE 30.15 The condition `newDeposit` is used for communications between the two threads.

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 that the initial balance is `0` and the amount to deposit and withdraw are randomly generated. Listing 30.6 gives the program. A sample run of the program is shown in Figure 30.16.

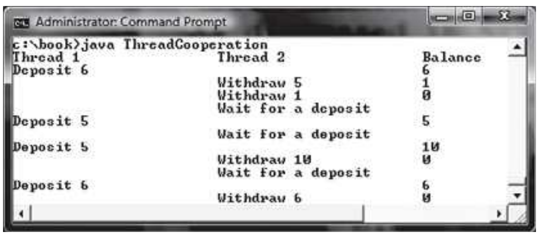


FIGURE 30.16 The withdraw task waits if there are not sufficient funds to withdraw.

LISTING 30.6 ThreadCooperation.java

create two threads

create a lock

create a condition

acquire the lock

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

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

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

    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }

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

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

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

    // Create a condition
    private static Condition newDeposit = lock.newCondition();

    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void withdraw(int amount) {
      lock.lock(); // Acquire the lock
      try {
        while (balance < amount) {
```

```
59           System.out.println("\t\t\tWait for a deposit");
60           newDeposit.await();
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
72       }
73     }
74
75     public void deposit(int amount) {
76       lock.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();
84       }
85       finally {
86         lock.unlock(); // Release the lock
87       }
88     }
89   }
90 }
```

wait on the condition

release the lock

acquire the lock

signal threads

release the lock

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

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

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

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

```
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 may lead to an incorrect withdraw. Using the loop statement, the withdraw task will have a chance to recheck the condition before performing a withdraw.

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.

`IllegalMonitorState-Exception`

Caution

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

Locks and conditions were introduced in Java 5. Prior to Java 5, thread communications were programmed using the object's built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor, so you 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 that some other thread can get in the monitor and perhaps change the monitor's state. When the condition is right, the other thread can invoke the **`notify()`** or **`notifyAll()`** method to signal one or all waiting threads to regain the lock and resume execution. The template for invoking these methods is shown in Figure 30.17.

Task 1

```java
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

resume

Task 2

```java
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or anObject.notifyAll();
  ...
}
```

FIGURE 30.17 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.

Check Point

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

30.20 What would happen if the `while` loop in line 58 of Listing 30.6 was changed to an `if` statement?

`while (balance < amount)` —Replaced by→ `if (balance < amount)`

30.21 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() {
  }
}
```

30.22 What is a possible cause for `IllegalMonitorStateException`?

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

30.24 What is wrong in the following code?

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

30.10 Case Study: Producer/Consumer

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

Key Point

Suppose you use a buffer to store integers and that the buffer size is limited. The buffer provides the method `write(int)` to add an `int` value to the buffer and the method `read()` to read and delete an `int` value from the buffer. To synchronize the operations, use a lock with two conditions: `notEmpty` (i.e., the buffer is not empty) and `notFull` (i.e., the buffer is not full). When a task adds an `int` to the buffer, if the buffer is full, the task will wait for the `notFull` condition. When a task 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 30.18.

Listing 30.7 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.

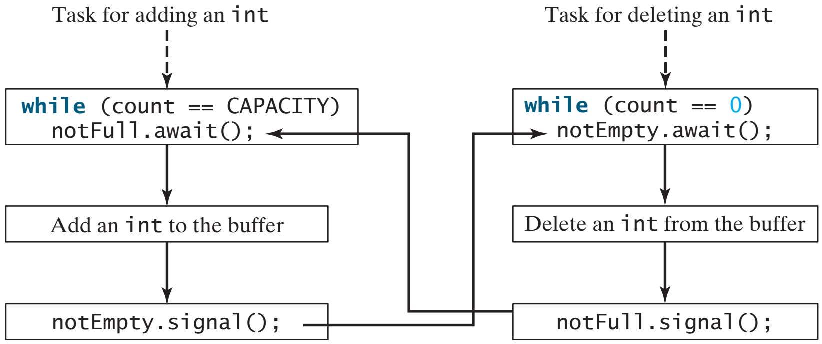


FIGURE 30.18 The conditions **notFull** and **notEmpty** are used to coordinate task interactions.

The buffer is actually a first-in, first-out queue (lines 52–53). The conditions **notEmpty** and **notFull** on the lock are created in lines 59–60. 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 30.7 ConsumerProducer.java

create a buffer (line 5)

create two threads (line 9)

producer task (line 16)

consumer task (line 34)

```
 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() {
```

```
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.read());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

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

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

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

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

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

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

        value = queue.remove();
        notFull.signal(); // Signal notFull condition
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
```

create a lock

create a condition

create a condition

acquire the lock

wait for `notFull`

signal `notEmpty`

release the lock

acquire the lock

wait for `notEmpty`

signal `notFull`

release the lock

```
        finally {
          lock.unlock(); // Release the lock
          return value;
        }
      }
    }
  }
```

A sample run of the program is shown in Figure 30.19.

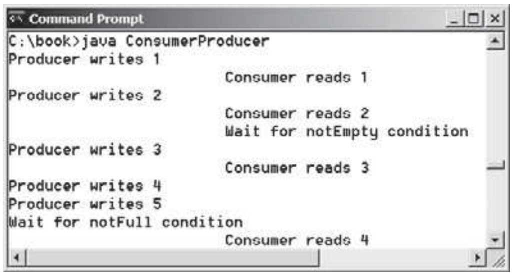


FIGURE 30.19 Locks and conditions are used for communications between the Producer and Consumer threads.

Check Point

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

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

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

30.11 Blocking Queues

Key Point

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

blocking queue

Queues and priority queues are 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 30.20.

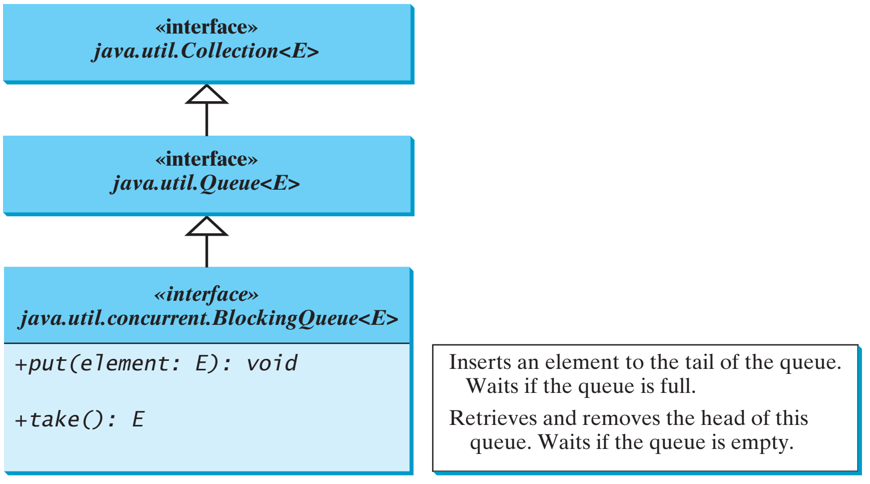


FIGURE 30.20 **BlockingQueue** is a subinterface of **Queue**.

Three concrete blocking queues—**ArrayBlockingQueue**, **LinkedBlockingQueue**, and **PriorityBlockingQueue**—are provided in Java, as shown in Figure 30.21. 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.

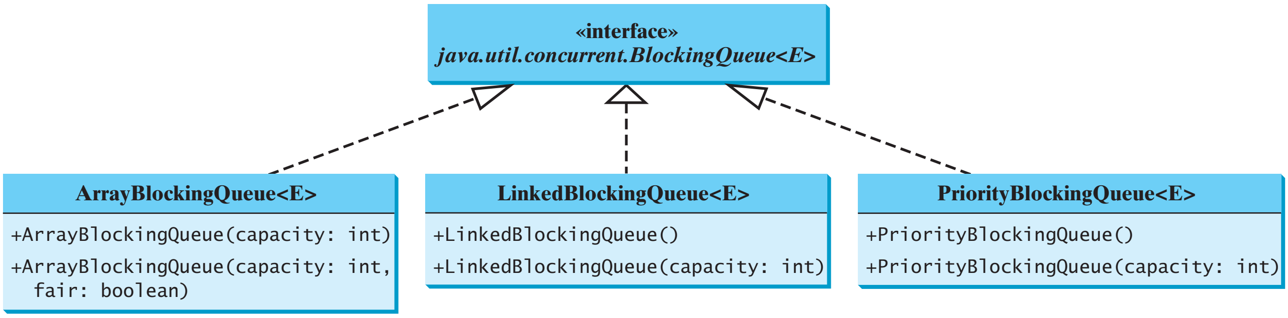


FIGURE 30.21 **ArrayBlockingQueue**, **LinkedBlockingQueue**, and **PriorityBlockingQueue** are concrete blocking queues.

> **Note**
> The **put** method will never block an unbounded **LinkedBlockingQueue** or **PriorityBlockingQueue**.

unbounded queue

Listing 30.8 gives an example of using an **ArrayBlockingQueue** to simplify the Consumer/Producer example in Listing 30.10. 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 30.8 ConsumerProducerUsingBlockingQueue.java

```
1  import java.util.concurrent.*;
2
3  public class ConsumerProducerUsingBlockingQueue {
4    private static ArrayBlockingQueue<Integer> buffer =
5      new ArrayBlockingQueue<>(2);
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 staticclass 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.put(i++); // Add any value to the buffer, say, 1
23           // Put the thread into sleep
```

create a buffer

create two threads

producer task

put

```
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

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

consumer task

take

In Listing 30.7, 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`.

Check Point

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

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

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

30.12 Semaphores

Key Point

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 30.22.

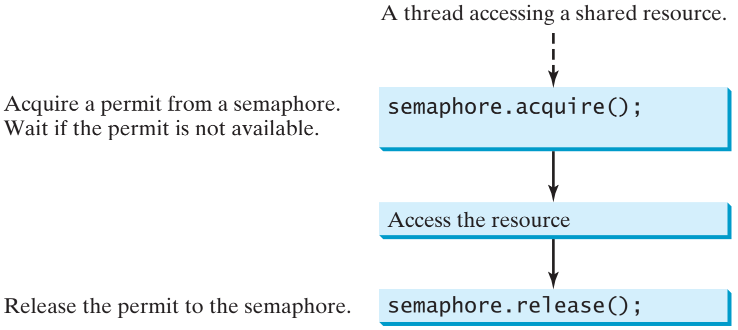


FIGURE 30.22 A limited number of threads can access a shared resource controlled by a semaphore.

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure 30.23. 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 30.23 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 30.9 revises the **Account** inner class in Listing 30.9 using a semaphore to ensure that only one thread at a time can access the **deposit** method.

LISTING 30.9 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);
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
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
26     }
27   }
28 }
```

create a semaphore

acquire a permit

release a permit

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.

Check Point

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

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

30.13 Avoiding Deadlocks

Key Point

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 30.24. 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.

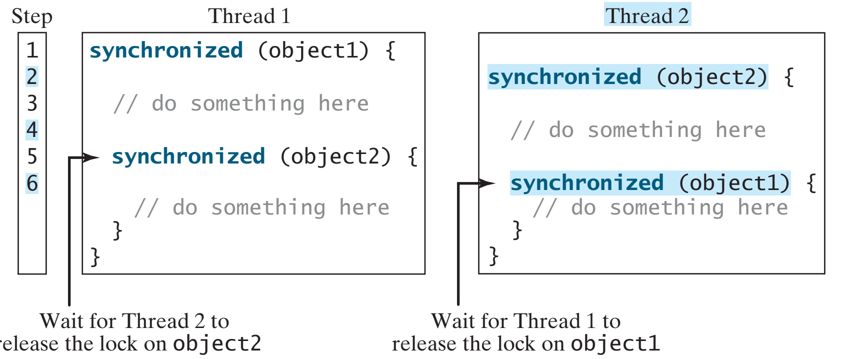


FIGURE 30.24 Thread 1 and Thread 2 are deadlocked.

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 the example in Figure 30.24, suppose that the objects are ordered as **object1** and **object2**. Using the resource ordering technique, Thread 2 must acquire a lock on **object1** first, then on **object2**. Once Thread 1 acquires a lock on **object1**, Thread 2 has to wait for a lock on **object1**. Thus, Thread 1 will be able to acquire a lock on **object2** and no deadlock will occur.

Check Point

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

30.14 Thread States

Key Point

A thread state indicates the status of thread.

Tasks are executed in threads. Threads can be in one of five states: New, Ready, Running, Blocked, or Finished (see Figure 30.25).

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.

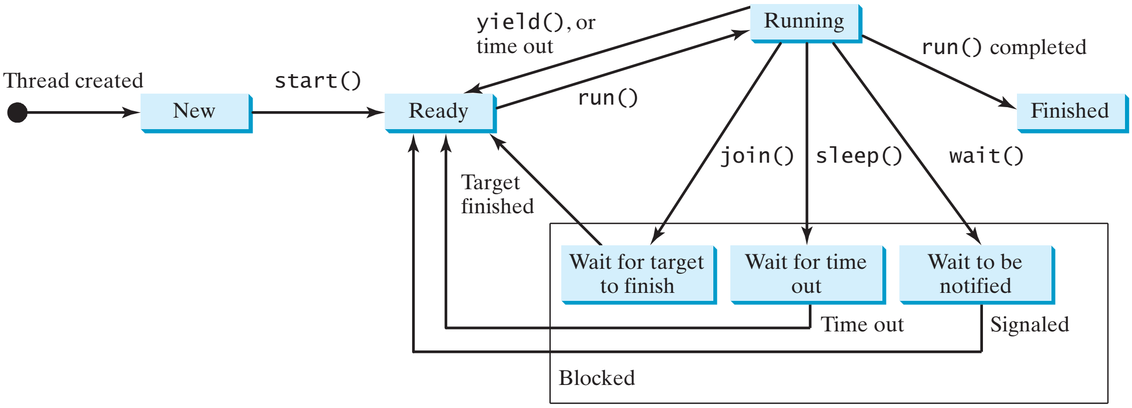


FIGURE 30.25 A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.

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.

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

Check Point

30.15 Synchronized Collections

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

Key Point

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 30.26. 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 30.26 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`.

Check Point

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

30.36 Explain why an iterator is fail-fast.

30.16 Parallel Programming

Key Point

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

JDK 7 feature

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

Fork/Join Framework

The *Fork/Join Framework* is illustrated in Figure 30.27 (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.

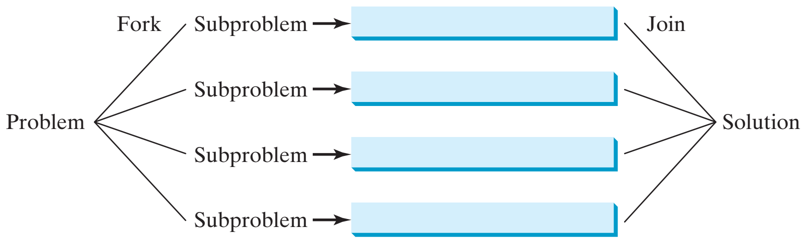


FIGURE 30.27 The nonoverlapping subproblems are solved in parallel.

The framework defines a task using the **ForkJoinTask** class, as shown in Figure 30.28 and executes a task in an instance of **ForkJoinPool**, as shown in Figure 30.29.

ForkJoinTask
ForkJoinPool

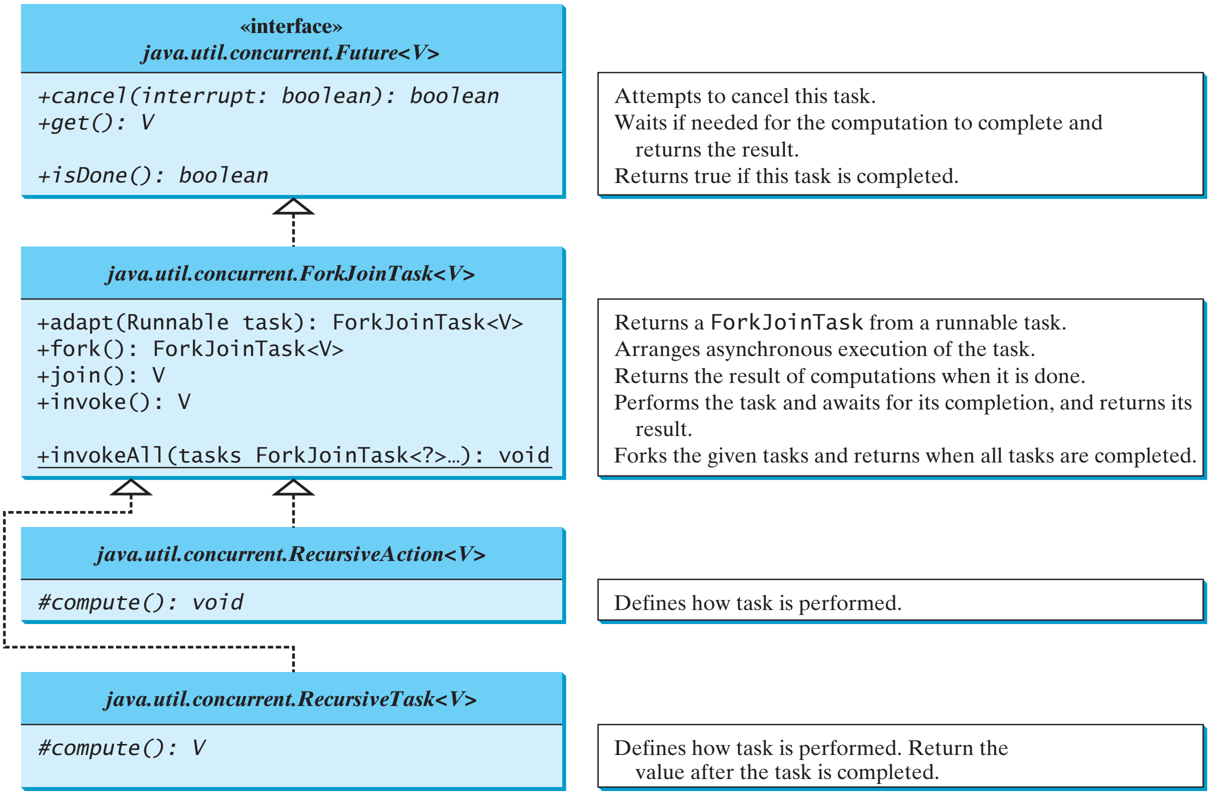


FIGURE 30.28 The **ForkJoinTask** class defines a task for asynchronous execution.

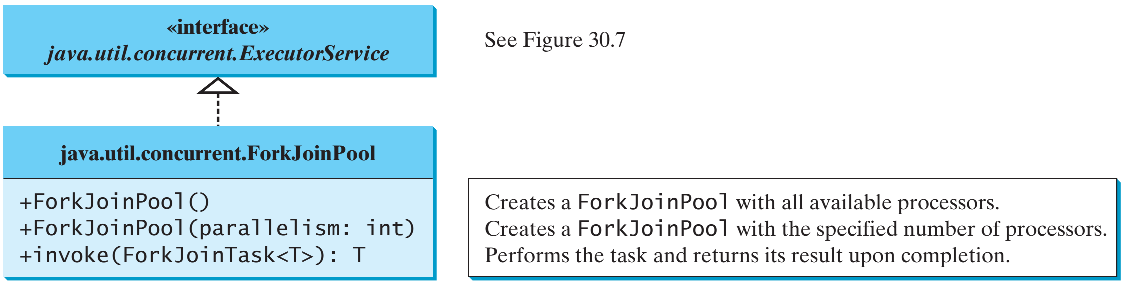


FIGURE 30.29 The **ForkJoinPool** executes Fork/Join tasks.

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 that the static method **invokeAll** takes a variable number of **ForkJoinTask** arguments using the **...** syntax, which is introduced in Section 7.9.

RecursiveAction
RecursiveTask

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

We now use a merge sort to demonstrate how to develop parallel programs using the Fork/Join Framework. The merge sort algorithm (introduced in Section 25.3) divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm merges them. Listing 30.10 gives a parallel implementation of the merge sort algorithm and compares its execution time with a sequential sort.

LISTING 30.10 ParallelMergeSort.java

invoke parallel sort
invoke sequential sort
create a ForkJoinTask
create a ForkJoinPool
execute a task
define concrete ForkJoinTask

```java
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;

public class ParallelMergeSort {
  public static void main(String[] args) {
    final int SIZE = 7000000;
    int[] list1 = new int[SIZE];
    int[] list2 = new int[SIZE];

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

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

    startTime = System.currentTimeMillis();
    MergeSort.mergeSort(list2); // MergeSort is in Listing 23.5
    endTime = System.currentTimeMillis();
    System.out.println("\nSequential time is " +
      (endTime - startTime) + " milliseconds");
  }

  public static void parallelMergeSort(int[] list) {
    RecursiveAction mainTask = new SortTask(list);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(mainTask);
  }

  private static class SortTask extends RecursiveAction {
    private final int THRESHOLD = 500;
```

```
35      private int[] list;
36
37      SortTask(int[] list) {
38        this.list = list;
39      }
40
41      @Override
42      protected void compute() {
43        if (list.length < THRESHOLD)
44          java.util.Arrays.sort(list);
45        else {
46          // Obtain the first half
47          int[] firstHalf = new int[list.length / 2];
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),
58            new SortTask(secondHalf));
59
60          // Merge firstHalf with secondHalf into list
61          MergeSort.merge(firstHalf, secondHalf, list);
62        }
63      }
64    }
65  }
```

perform the task (line 42)
sort a small list (line 44)
split into two parts (line 47)
solve each part (line 57)
merge two parts (line 61)

```
Parallel time with 2 processors is 2829 milliseconds
Sequential time is 4751 milliseconds
```

Since the sort algorithm does not return a value, we define a concrete `ForkJoinTask` class by extending `RecursiveAction` (lines 33–64). The `compute` method is overridden to implement a recursive merge sort (lines 42–63). If the list is small, it is more efficient to be solved sequentially (line 44). For a large list, it is split into two halves (lines 47–54). The two halves are sorted concurrently (lines 57 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 that 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 that the loop for initializing the list can also be parallelized. However, you should avoid using `Math.random()` in the code because it is synchronized and cannot be executed in parallel (see Programming Exercise 30.12). The `parallelMergeSort` method only sorts an array of `int` values, but you can modify it to become a generic method (see Programming Exercise 30.13).

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

LISTING 30.11 ParallelMax.java

invoke max

create a ForkJoinTask

create a ForkJoinPool

execute a task

define concrete ForkJoinTask

perform the task

solve a small problem

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

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

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

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

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

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

    @Override
    public Integer compute() {
      if (high - low < THRESHOLD) {
        int max = list[0];
        for (int i = low; i < high; i++)
          if (list[i] > max)
            max = list[i];
        return new Integer(max);
```

```
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).

Check Point

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

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

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

30.40 How do you create a **`ForkJoinPool`**? How do you place a task into a **`ForkJoinPool`**?

Key Terms

condition 1114	multithreading 1098
deadlock 1126	race condition 1111
fail-fast 1128	semaphore 1124
fairness policy 1113	synchronization wrapper 1127
Fork/Join Framework 1128	synchronized block 1112
lock 1112	thread 1098
monitor 1118	thread-safe 1111

Chapter Summary

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

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

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

4. To prevent threads from corrupting a shared resource, use *synchronized* methods or blocks. A *synchronized method* acquires a *lock* before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static 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**, **PriorityBlockingQueue**) provided in the Java Collections Framework automatically synchronize access to a queue.

8. You can use semaphores to restrict the number of concurrent tasks that access 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 obtains the overall solution after all tasks are completed.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/intro10e/quiz.html.

MyProgrammingLab™

Programming Exercises

Sections 30.1–30.5

***30.1** (*Revise Listing 30.1*) Rewrite Listing 30.1 to display the output in a text area, as shown in Figure 30.30.

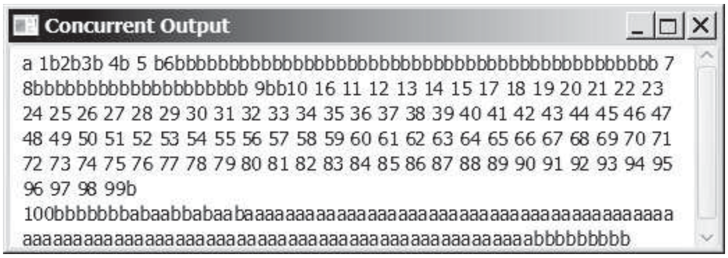


Figure 30.30 The output from three threads is displayed in a text area.

30.2 (*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 programs. Which one runs the animation faster?

30.3 (*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 30.8–30.12

30.4 (*Synchronize threads*) Write a program that launches 1,000 threads. Each thread adds `1` to a variable `sum` that initially is `0`. Define an `Integer` wrapper object to hold `sum`. Run the program with and without synchronization to see its effect.

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

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

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

30.8 (*Account synchronization*) Rewrite Listing 30.6, ThreadCooperation.java, using the object's `wait()` and `notifyAll()` methods.

30.9 (*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.

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

Section 30.15

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

Section 30.18

***30.12** (*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 that if you use `Math.random()`, your parallel code execution time will be worse than the sequential code execution time because `Math.random()` is synchronized and cannot be executed in parallel. To fix this problem, create a `Random` object for assigning random values to a small list.

30.13 (*Generic parallel merge sort*) Revise Listing 30.10, ParallelMergeSort.java, to define a generic `parallelMergeSort` method as follows:

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

***30.14** (*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.

***30.15** (*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.

***30.16** (*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 × 2,000 matrices using the parallel method and sequential method, respectively.

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

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

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

***30.18** (*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

*****30.19** (*Sorting animation*) Write an animation for selection sort, insertion sort, and bubble sort, as shown in Figure 30.31. 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.

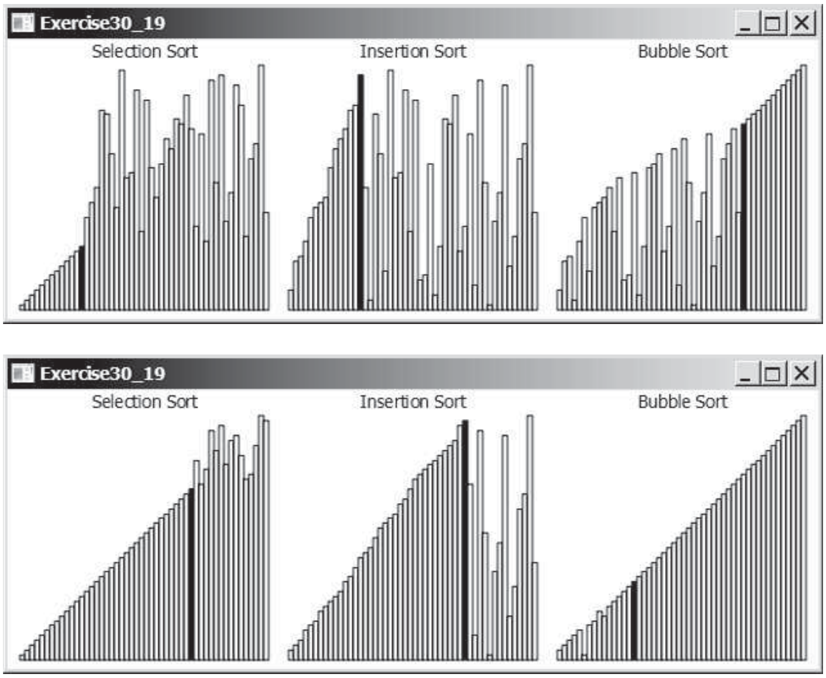
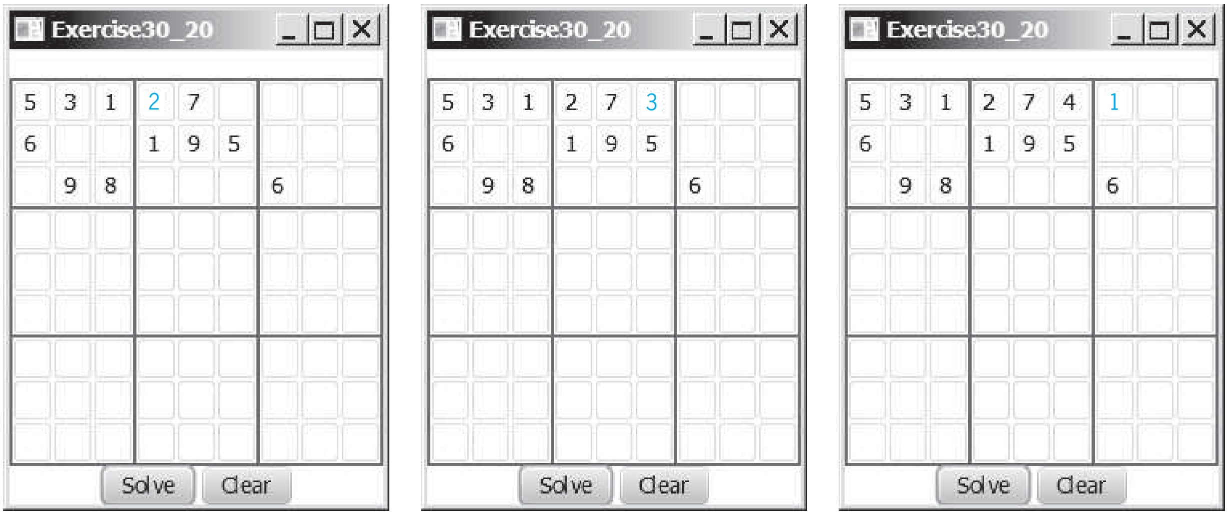


FIGURE 30.31 Three sorting algorithms are illustrated in the animation.

*****30.20** (*Sudoku search animation*) Modify Programming Exercise 22.21 to display the intermediate results of the search. Figure 30.32 gives a snapshot of an animation in progress with number **2** placed in the cell in Figure 30.32a, number **3** placed in the cell in Figure 30.32b, and number **3** placed in the cell in Figure 30.32c. The animation displays all the search steps.



(a) (b) (c)

FIGURE 30.32 The intermediate search steps are displayed in the animation for the Sudoku problem.

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

*****30.22** (*Eight Queens animation*) Modify Listing 22.11, EightQueens.java, to display the intermediate results of the search. As shown in Figure 30.33, the current row being searched is highlighted. Every one second, a new state of the chess board is displayed.

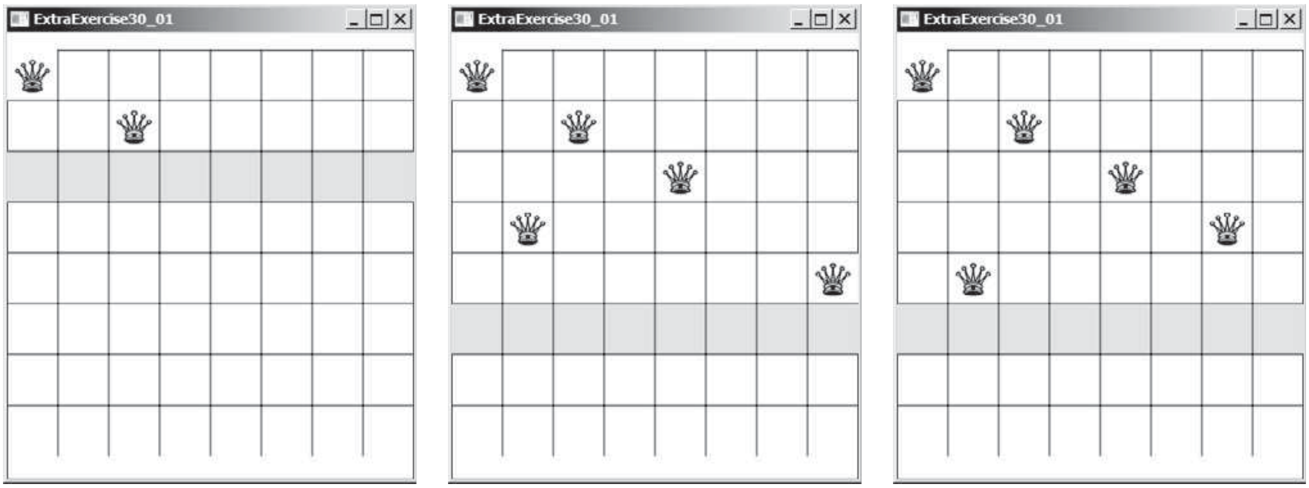


FIGURE 30.33 The intermediate search steps are displayed in the animation for the Eight Queens problem.

CHAPTER

31

NETWORKING

Objectives

- To explain terms: TCP, IP, domain name, domain name server, stream-based communications, and packet-based communications (§31.2).
- To create servers using server sockets (§31.2.1) and clients using client sockets (§31.2.2).
- To implement Java networking programs using stream sockets (§31.2.3).
- To develop an example of a client/server application (§31.2.4).
- To obtain Internet addresses using the `InetAddress` class (§31.3).
- To develop servers for multiple clients (§31.4).
- To send and receive objects on a network (§31.5).
- To develop an interactive tic-tac-toe game played on the Internet (§31.6).

31.1 Introduction

Computer networking is used to send and receive messages among computers on the Internet.

To browse the Web or send an email, your computer must be connected to the Internet. The *Internet* is the global network of millions of computers. Your computer can connect to the Internet through an Internet Service Provider (ISP) using a dialup, DSL, or cable modem, or through a local area network (LAN).

IP address

When a computer needs to communicate with another computer, it needs to know the other computer's address. An *Internet Protocol* (IP) address uniquely identifies the computer on the Internet. An IP address consists of four dotted decimal numbers between **`0`** and **`255`**, such as **`130.254.204.31`**. Since it is not easy to remember so many numbers, they are often mapped to meaningful names called *domain names*, such as liang.armstrong.edu. Special servers called *Domain Name Servers* (DNS) on the Internet translate host names into IP addresses. When a computer contacts liang.armstrong.edu, it first asks the DNS to translate this domain name into a numeric IP address and then sends the request using the IP address.

domain name
domain name server

TCP
UDP

The Internet Protocol is a low-level protocol for delivering data from one computer to another across the Internet in packets. Two higher-level protocols used in conjunction with the IP are the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent. UDP is a standard, low-overhead, connectionless, host-to-host protocol that is used over the IP. UDP allows an application program on one computer to send a datagram to an application program on another computer.

stream-based communication
packet-based communication

Java supports both stream-based and packet-based communications. *Stream-based communications* use TCP for data transmission, whereas *packet-based communications* use UDP. Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission. Stream-based communications are used in most areas of Java programming and are the focus of this chapter. Packet-based communications are introduced in Supplement III.P, Networking Using Datagram Protocol.

31.2 Client/Server Computing

Java provides the **`ServerSocket`** *class for creating a server socket and the* **`Socket`** *class for creating a client socket. Two programs on the Internet communicate through a server socket and a client socket using I/O streams.*

socket

Networking is tightly integrated in Java. The Java API provides the classes for creating sockets to facilitate program communications over the Internet. *Sockets* are the endpoints of logical connections between two hosts and can be used to send and receive data. Java treats socket communications much as it treats I/O operations; thus, programs can read from or write to sockets as easily as they can read from or write to files.

Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds. The client begins by attempting to establish a connection to the server. The server can accept or deny the connection. Once a connection is established, the client and the server communicate through sockets.

The server must be running when a client attempts to connect to the server. The server waits for a connection request from the client. The statements needed to create sockets on a server and on a client are shown in Figure 31.1.

31.2.1 Server Sockets

server socket
port

To establish a server, you need to create a *server socket* and attach it to a *port*, which is where the server listens for connections. The port identifies the TCP service on the socket. Port numbers range from 0 to 65536, but port numbers 0 to 1024 are reserved for privileged services.

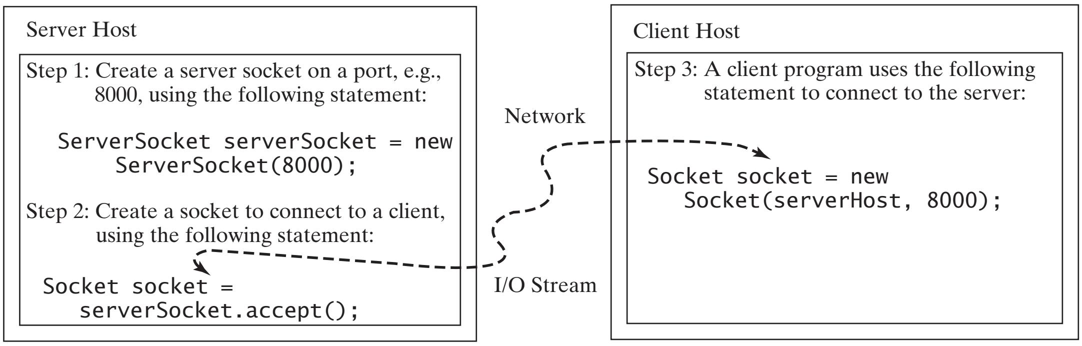


FIGURE 31.1 The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket.

For instance, the email server runs on port 25, and the Web server usually runs on port 80. You can choose any port number that is not currently used by other programs. The following statement creates a server socket **serverSocket**:

```
ServerSocket serverSocket = new ServerSocket(port);
```

Note

Attempting to create a server socket on a port already in use would cause a **java.net.BindException**. BindException

31.2.2 Client Sockets

After a server socket is created, the server can use the following statement to listen for connections:

```
Socket socket = serverSocket.accept();
```

This statement waits until a client connects to the server socket. The client issues the following statement to request a connection to a server: connect to client

```
Socket socket = new Socket(serverName, port);
```

This statement opens a socket so that the client program can communicate with the server. ***serverName*** is the server's Internet host name or IP address. The following statement creates a socket on the client machine to connect to the host 130.254.204.33 at port 8000: client socket use IP address

```
Socket socket = new Socket("130.254.204.33", 8000)
```

Alternatively, you can use the domain name to create a socket, as follows: use domain name

```
Socket socket = new Socket("liang.armstrong.edu", 8000);
```

When you create a socket with a host name, the JVM asks the DNS to translate the host name into the IP address.

Note

A program can use the host name **localhost** or the IP address **127.0.0.1** to refer to the machine on which a client is running. localhost

UnknownHostException

Note

The **Socket** constructor throws a **java.net.UnknownHostException** if the host cannot be found.

31.2.3 Data Transmission through Sockets

After the server accepts the connection, communication between the server and the client is conducted in the same way as for I/O streams. The statements needed to create the streams and to exchange data between them are shown in Figure 31.2.

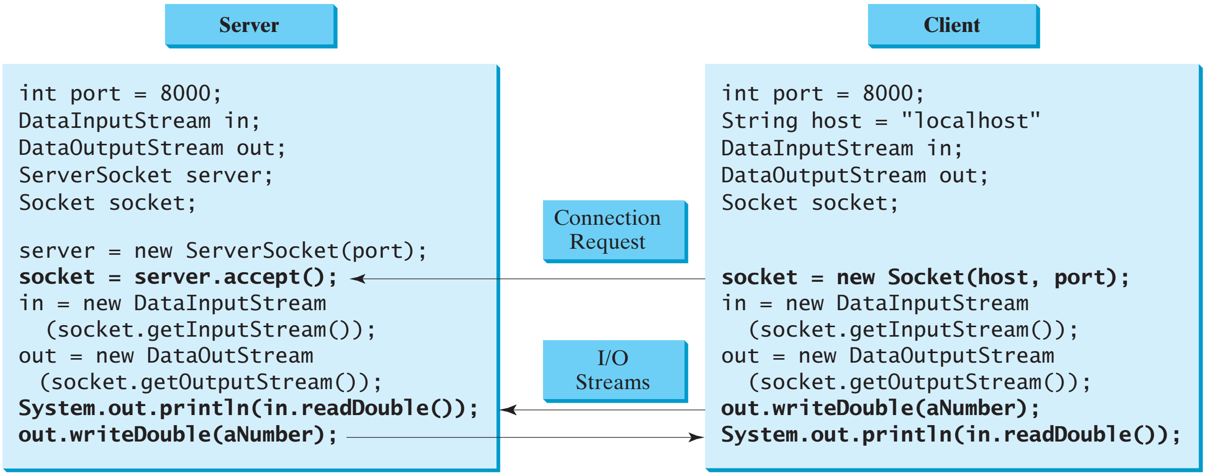


FIGURE 31.2 The server and client exchange data through I/O streams on top of the socket.

To get an input stream and an output stream, use the **getInputStream()** and **getOutputStream()** methods on a socket object. For example, the following statements create an **InputStream** stream called **input** and an **OutputStream** stream called **output** from a socket:

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

The **InputStream** and **OutputStream** streams are used to read or write bytes. You can use **DataInputStream**, **DataOutputStream**, **BufferedReader**, and **PrintWriter** to wrap on the **InputStream** and **OutputStream** to read or write data, such as **int**, **double**, or **String**. The following statements, for instance, create the **DataInputStream** stream **input** and the **DataOutput** stream **output** to read and write primitive data values:

```
DataInputStream input = new DataInputStream
  (socket.getInputStream());
DataOutputStream output = new DataOutputStream
  (socket.getOutputStream());
```

The server can use **input.readDouble()** to receive a **double** value from the client and **output.writeDouble(d)** to send the **double** value **d** to the client.

Tip

Recall that binary I/O is more efficient than text I/O because text I/O requires encoding and decoding. Therefore, it is better to use binary I/O for transmitting data between a server and a client to improve performance.

31.2.4 A Client/Server Example

This example presents a client program and a server program. The client sends data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console. In this example, the data sent from the client comprise the radius of a circle, and the result produced by the server is the area of the circle (see Figure 31.3).

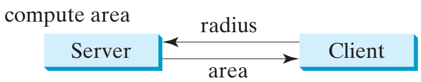


FIGURE 31.3 The client sends the radius to the server; the server computes the area and sends it to the client.

The client sends the radius through a `DataOutputStream` on the output stream socket, and the server receives the radius through the `DataInputStream` on the input stream socket, as shown in Figure 31.4a. The server computes the area and sends it to the client through a `DataOutputStream` on the output stream socket, and the client receives the area through a `DataInputStream` on the input stream socket, as shown in Figure 31.4b. The server and client programs are given in Listings 31.1 and 31.2. Figure 31.5 contains a sample run of the server and the client.

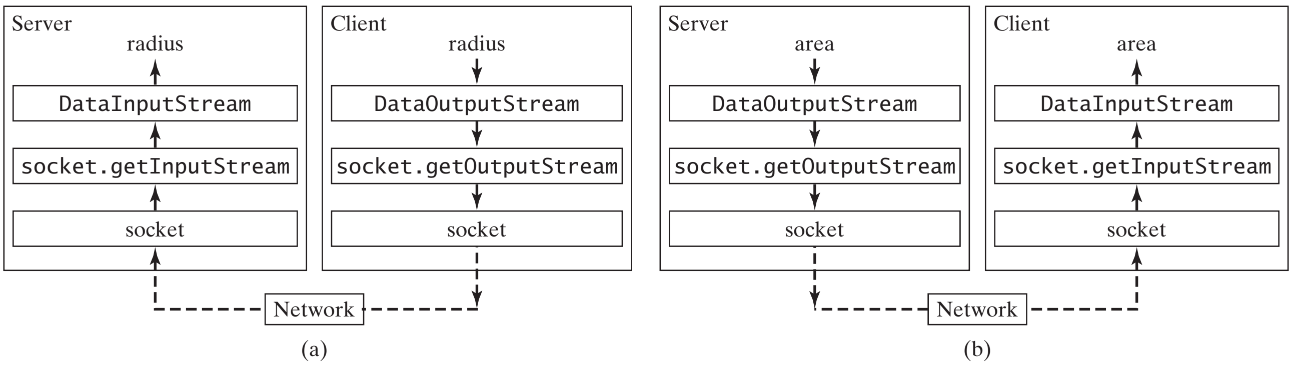


FIGURE 31.4 (a) The client sends the radius to the server. (b) The server sends the area to the client.

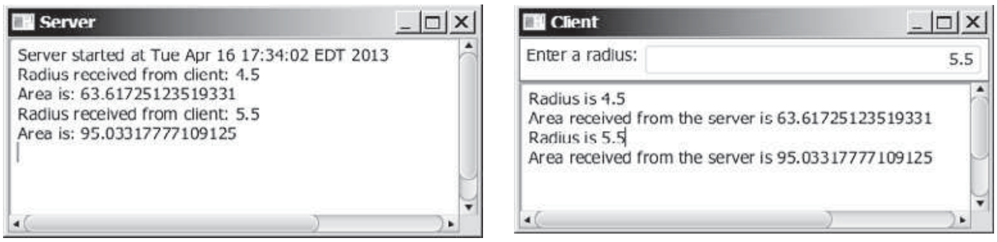


FIGURE 31.5 The client sends the radius to the server. The server receives it, computes the area, and sends the area to the client.

LISTING 31.1 Server.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
```

create server UI
server socket
update UI
connect client
input from client
output to client
read radius
write area
update UI

```java
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class Server extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Text area for displaying contents
    TextArea ta = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(ta), 450, 200);
    primaryStage.setTitle("Server"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread(() -> {
      try {
        // Create a server socket
        ServerSocket serverSocket = new ServerSocket(8000);
        Platform.runLater(() ->
          ta.appendText("Server started at " + new Date() + '\n'));

        // Listen for a connection request
        Socket socket = serverSocket.accept();

        // Create data input and output streams
        DataInputStream inputFromClient = new DataInputStream(
          socket.getInputStream());
        DataOutputStream outputToClient = new DataOutputStream(
          socket.getOutputStream());

        while (true) {
          // Receive radius from the client
          double radius = inputFromClient.readDouble();

          // Compute area
          double area = radius * radius * Math.PI;

          // Send area back to the client
          outputToClient.writeDouble(area);

          Platform.runLater(() -> {
            ta.appendText("Radius received from client: "
              + radius + '\n');
            ta.appendText("Area is: " + area + '\n');
          });
        }
      }
      catch(IOException ex) {
        ex.printStackTrace();
      }
    }).start();
  }
}
```

LISTING 31.2 Client.java

```java
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Client extends Application {
  // IO streams
  DataOutputStream toServer = null;
  DataInputStream fromServer = null;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Panel p to hold the label and text field
    BorderPane paneForTextField = new BorderPane();
    paneForTextField.setPadding(new Insets(5, 5, 5, 5));
    paneForTextField.setStyle("-fx-border-color: green");
    paneForTextField.setLeft(new Label("Enter a radius: "));

    TextField tf = new TextField();
    tf.setAlignment(Pos.BOTTOM_RIGHT);
    paneForTextField.setCenter(tf);

    BorderPane mainPane = new BorderPane();
    // Text area to display contents
    TextArea ta = new TextArea();
    mainPane.setCenter(new ScrollPane(ta));
    mainPane.setTop(paneForTextField);

    // Create a scene and place it in the stage
    Scene scene = new Scene(mainPane, 450, 200);
    primaryStage.setTitle("Client"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    tf.setOnAction(e -> {
      try {
        // Get the radius from the text field
        double radius = Double.parseDouble(tf.getText().trim());

        // Send the radius to the server
        toServer.writeDouble(radius);
        toServer.flush();

        // Get area from the server
        double area = fromServer.readDouble();

        // Display to the text area
        ta.appendText("Radius is " + radius + "\n");
        ta.appendText("Area received from the server is "
          + area + '\n');
```

create UI

handle action event

read radius

write radius

read area

request connection

input from server

output to server

```
59         }
60         catch (IOException ex) {
61           System.err.println(ex);
62         }
63       });
64 
65       try {
66         // Create a socket to connect to the server
67         Socket socket = new Socket("localhost", 8000);
68         // Socket socket = new Socket("130.254.204.36", 8000);
69         // Socket socket = new Socket("drake.Armstrong.edu", 8000);
70 
71         // Create an input stream to receive data from the server
72         fromServer = new DataInputStream(socket.getInputStream());
73 
74         // Create an output stream to send data to the server
75         toServer = new DataOutputStream(socket.getOutputStream());
76       }
77       catch (IOException ex) {
78         ta.appendText(ex.toString() + '\n');
79       }
80     }
81   }
```

You start the server program first and then start the client program. In the client program, enter a radius in the text field and press *Enter* to send the radius to the server. The server computes the area and sends it back to the client. This process is repeated until one of the two programs terminates.

The networking classes are in the package `java.net`. You should import this package when writing Java network programs.

The `Server` class creates a `ServerSocket serverSocket` and attaches it to port 8000 using this statement (line 26 in Server.java):

```
ServerSocket serverSocket = new ServerSocket(8000);
```

The server then starts to listen for connection requests, using the following statement (line 31 in Server.java):

```
Socket socket = serverSocket.accept();
```

The server waits until the client requests a connection. After it is connected, the server reads the radius from the client through an input stream, computes the area, and sends the result to the client through an output stream. The `ServerSocket accept()` method takes time to execute. It is not appropriate to run this method in the JavaFX application thread. So, we place it in a separate thread (lines 23–59). The statements for updating GUI need to run from the JavaFX application thread using the `Platform.runLater` method (lines 27–28, 49–53).

The `Client` class uses the following statement to create a socket that will request a connection to the server on the same machine (localhost) at port 8000 (line 67 in Client.java).

```
Socket socket = new Socket("localhost", 8000);
```

If you run the server and the client on different machines, replace `localhost` with the server machine's host name or IP address. In this example, the server and the client are running on the same machine.

If the server is not running, the client program terminates with a `java.net.ConnectException`. After it is connected, the client gets input and output streams—wrapped by data input and output streams—in order to receive and send data to the server.

If you receive a `java.net.BindException` when you start the server, the server port is currently in use. You need to terminate the process that is using the server port and then restart the server.

Note

When you create a server socket, you have to specify a port (e.g., 8000) for the socket. When a client connects to the server (line 67 in Client.java), a socket is created on the client. This socket has its own local port. This port number (e.g., 2047) is automatically chosen by the JVM, as shown in Figure 31.6.

client socket port

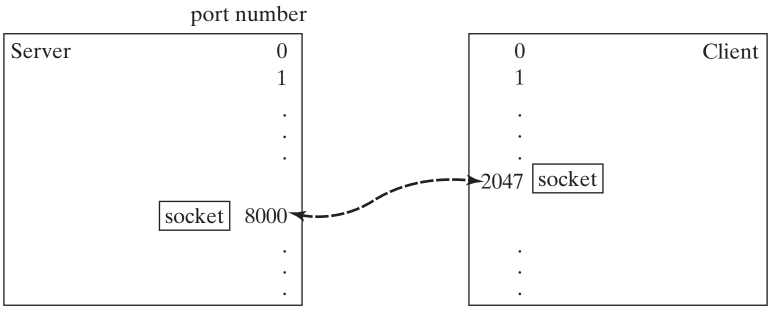


FIGURE 31.6 The JVM automatically chooses an available port to create a socket for the client.

To see the local port on the client, insert the following statement in line 70 in Client.java.

```
System.out.println("local port: " + socket.getLocalPort());
```

Check Point

31.1 How do you create a server socket? What port numbers can be used? What happens if a requested port number is already in use? Can a port connect to multiple clients?

31.2 What are the differences between a server socket and a client socket?

31.3 How does a client program initiate a connection?

31.4 How does a server accept a connection?

31.5 How are data transferred between a client and a server?

31.3 The **InetAddress** Class

Key Point

The server program can use the `InetAddress` *class to obtain the information about the IP address and host name for the client.*

Occasionally, you would like to know who is connecting to the server. You can use the `InetAddress` class to find the client's host name and IP address. The `InetAddress` class models an IP address. You can use the following statement in the server program to get an instance of `InetAddress` on a socket that connects to the client.

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " +
  inetAddress.getHostName());
```

```
System.out.println("Client's IP Address is " +
  inetAddress.getHostAddress());
```

You can also create an instance of `InetAddress` from a host name or IP address using the static `getByName` method. For example, the following statement creates an `InetAddress` for the host `liang.armstrong.edu`.

```
InetAddress address = InetAddress.getByName("liang.armstrong.edu");
```

Listing 31.3 gives a program that identifies the host name and IP address of the arguments you pass in from the command line. Line 7 creates an `InetAddress` using the `getByName` method. Lines 8 and 9 use the `getHostName` and `getHostAddress` methods to get the host's name and IP address. Figure 31.7 shows a sample run of the program.

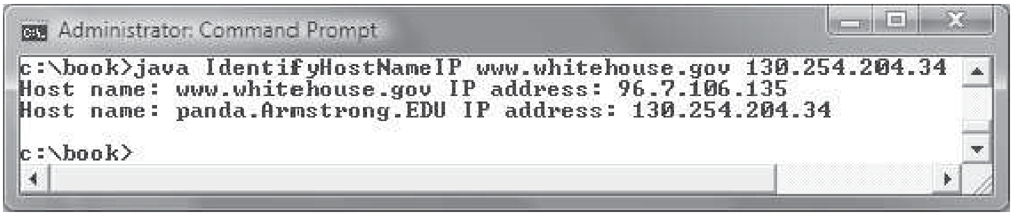


FIGURE 31.7 The program identifies host names and IP addresses.

LISTING 31.3 IdentifyHostNameIP.java

get an `InetAddress`
get host name
get host IP

```
1  import java.net.*;
2
3  public class IdentifyHostNameIP {
4    public static void main(String[] args) {
5      for (int i = 0; i < args.length; i++) {
6        try {
7          InetAddress address = InetAddress.getByName(args[i]);
8          System.out.print("Host name: " + address.getHostName() + " ");
9          System.out.println("IP address: " + address.getHostAddress());
10       }
11       catch (UnknownHostException ex) {
12         System.err.println("Unknown host or IP address " + args[i]);
13       }
14     }
15   }
16 }
```

Check Point

31.6 How do you obtain an instance of `InetAddress`?

31.7 What methods can you use to get the IP address and hostname from an `InetAddress`?

31.4 Serving Multiple Clients

Key Point

A server can serve multiple clients. The connection to each client is handled by one thread.

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs continuously on a server computer, and clients from all over the Internet can connect to it. You can use threads to handle the server's multiple clients simultaneously—simply create a thread for each connection. Here is how the server handles the establishment of a connection:

```
while (true) {
  Socket socket = serverSocket.accept(); // Connect to a client
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

The server socket can have many connections. Each iteration of the `while` loop creates a new connection. Whenever a connection is established, a new thread is created to handle communication between the server and the new client, and this allows multiple connections to run at the same time.

Listing 31.4 creates a server class that serves multiple clients simultaneously. For each connection, the server starts a new thread. This thread continuously receives input (the radius of a circle) from clients and sends the results (the area of the circle) back to them (see Figure 31.8). The client program is the same as in Listing 31.2. A sample run of the server with two clients is shown in Figure 31.9.

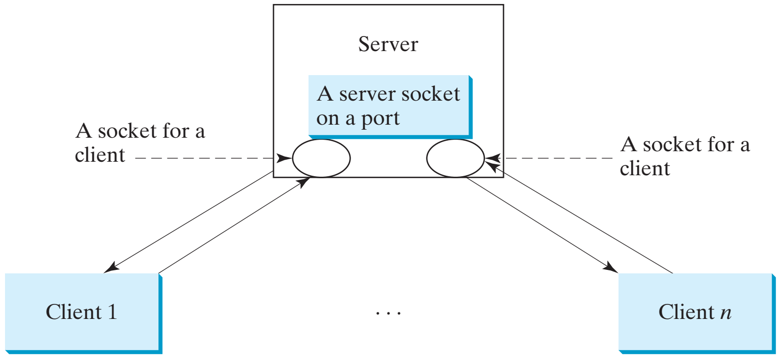


FIGURE 31.8 Multithreading enables a server to handle multiple independent clients.

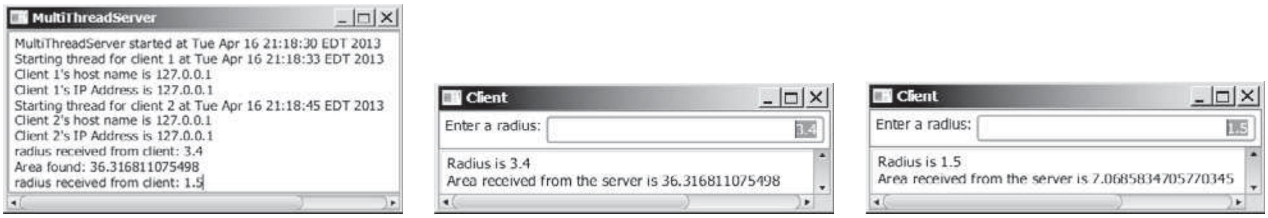


FIGURE 31.9 The server spawns a thread in order to serve a client.

LISTING 31.4 `MultiThreadServer.java`

```
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

```

server socket

connect client

update GUI

network information

create task

start thread

task class

```
public class MultiThreadServer extends Application {
  // Text area for displaying contents
  private TextArea ta = new TextArea();

  // Number a client
  private int clientNo = 0;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(ta), 450, 200);
    primaryStage.setTitle("MultiThreadServer"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
      try {
        // Create a server socket
        ServerSocket serverSocket = new ServerSocket(8000);
        ta.appendText("MultiThreadServer started at "
          + new Date() + '\n');

        while (true) {
          // Listen for a new connection request
          Socket socket = serverSocket.accept();

          // Increment clientNo
          clientNo++;

          Platform.runLater( () -> {
            // Display the client number
            ta.appendText("Starting thread for client " + clientNo +
              " at " + new Date() + '\n');

            // Find the client's host name, and IP address
            InetAddress inetAddress = socket.getInetAddress();
            ta.appendText("Client " + clientNo + "'s host name is "
              + inetAddress.getHostName() + "\n");
            ta.appendText("Client " + clientNo + "'s IP Address is "
              + inetAddress.getHostAddress() + "\n");
          });

          // Create and start a new thread for the connection
          new Thread(new HandleAClient(socket)).start();
        }
      }
      catch(IOException ex) {
        System.err.println(ex);
      }
    }).start();
  }

  // Define the thread class for handling new connection
  class HandleAClient implements Runnable {
    private Socket socket; // A connected socket

    /** Construct a thread */
    public HandleAClient(Socket socket) {
      this.socket = socket;
    }

```

```
72        /** Run a thread */
73        public void run() {
74          try {
75            // Create data input and output streams
76            DataInputStream inputFromClient = new DataInputStream(
77              socket.getInputStream());
78            DataOutputStream outputToClient = new DataOutputStream(
79              socket.getOutputStream());
80
81            // Continuously serve the client
82            while (true) {
83              // Receive radius from the client
84              double radius = inputFromClient.readDouble();
85
86              // Compute area
87              double area = radius * radius * Math.PI;
88
89              // Send area back to the client
90              outputToClient.writeDouble(area);
91
92              Platform.runLater(() -> {
93                ta.appendText("radius received from client: " +
94                  radius + '\n');
95                ta.appendText("Area found: " + area + '\n');
96              });
97            }
98          }
99          catch(IOException e) {
100           ex.printStackTrace();
101         }
102       }
103     }
104   }
```

I/O

update GUI

The server creates a server socket at port 8000 (line 29) and waits for a connection (line 35). When a connection with a client is established, the server creates a new thread to handle the communication (line 54). It then waits for another connection in an infinite `while` loop (lines 33–55).

The threads, which run independently of one another, communicate with designated clients. Each thread creates data input and output streams that receive and send data to a client.

31.8 How do you make a server serve multiple clients?

Check Point

31.5 Sending and Receiving Objects

A program can send and receive objects from another program.

Key Point

In the preceding examples, you learned how to send and receive data of primitive types. You can also send and receive objects using `ObjectOutputStream` and `ObjectInputStream` on socket streams. To enable passing, the objects must be serializable. The following example demonstrates how to send and receive objects.

The example consists of three classes: StudentAddress.java (Listing 31.5), StudentClient.java (Listing 31.6), and StudentServer.java (Listing 31.7). The client program collects student information from the client and sends it to a server, as shown in Figure 31.10.

The `StudentAddress` class contains the student information: name, street, city, state, and zip. The `StudentAddress` class implements the `Serializable` interface. Therefore, a `StudentAddress` object can be sent and received using the object output and input streams.

FIGURE 31.10 The client sends the student information in an object to the server.

LISTING 31.5 StudentAddress.java

serialized

```
public class StudentAddress implements java.io.Serializable {
  private String name;
  private String street;
  private String city;
  private String state;
  private String zip;

  public StudentAddress(String name, String street, String city,
    String state, String zip) {
    this.name = name;
    this.street = street;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getName() {
    return name;
  }

  public String getStreet() {
    return street;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

The client sends a `StudentAddress` object through an `ObjectOutputStream` on the output stream socket, and the server receives the `Student` object through the `ObjectInputStream` on the input stream socket, as shown in Figure 31.11. The client uses the `writeObject` method in the `ObjectOutputStream` class to send data about a student to the server, and the server receives the student's information using the `readObject` method in the `ObjectInputStream` class. The server and client programs are given in Listings 31.6 and 31.7.

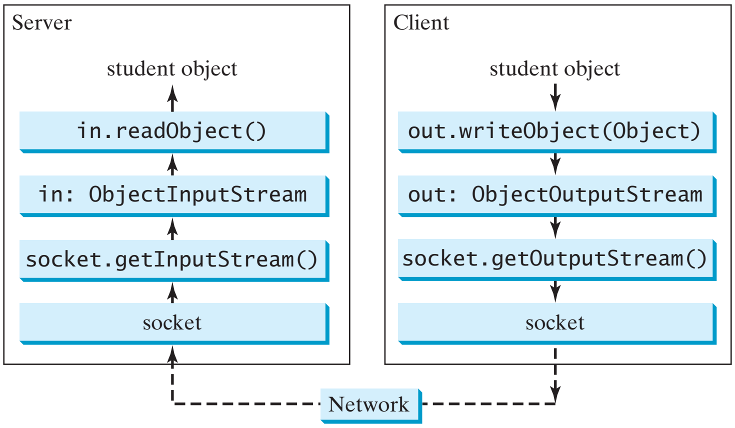


FIGURE 31.11 The client sends a **StudentAddress** object to the server.

LISTING 31.6 StudentClient.java

```
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class StudentClient extends Application {
  private TextField tfName = new TextField();
  private TextField tfStreet = new TextField();
  private TextField tfCity = new TextField();
  private TextField tfState = new TextField();
  private TextField tfZip = new TextField();

  // Button for sending a student to the server
  private Button btRegister = new Button("Register to the Server");

  // Host name or ip
  String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    GridPane pane = new GridPane();
    pane.add(new Label("Name"), 0, 0);
    pane.add(tfName, 1, 0);
    pane.add(new Label("Street"), 0, 1);
    pane.add(tfStreet, 1, 1);
    pane.add(new Label("City"), 0, 2);

```

create UI

```
    HBox hBox = new HBox(2);
    pane.add(hBox, 1, 2);
    hBox.getChildren().addAll(tfCity, new Label("State"), tfState,
      new Label("Zip"), tfZip);
    pane.add(btRegister, 1, 3);
    GridPane.setHalignment(btRegister, HPos.RIGHT);

    pane.setAlignment(Pos.CENTER);
    tfName.setPrefColumnCount(15);
    tfStreet.setPrefColumnCount(15);
    tfCity.setPrefColumnCount(10);
    tfState.setPrefColumnCount(2);
    tfZip.setPrefColumnCount(3);

```

register listener

```
    btRegister.setOnAction(new ButtonListener());

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 450, 200);
    primaryStage.setTitle("StudentClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Handle button action */
  private class ButtonListener implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
      try {
        // Establish connection with the server
```

server socket

```
        Socket socket = new Socket(host, 8000);

        // Create an output stream to the server
```

output stream

```
        ObjectOutputStream toServer =
          new ObjectOutputStream(socket.getOutputStream());

        // Get text field
        String name = tfName.getText().trim();
        String street = tfStreet.getText().trim();
        String city = tfCity.getText().trim();
        String state = tfState.getText().trim();
        String zip = tfZip.getText().trim();

        // Create a Student object and send to the server
        StudentAddress s =
          new StudentAddress(name, street, city, state, zip);
```

send to server

```
        toServer.writeObject(s);
      }
      catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

LISTING 31.7 StudentServer.java

```
import java.io.*;
import java.net.*;

public class StudentServer {
```

```
5     private ObjectOutputStream outputToFile;
6     private ObjectInputStream inputFromClient;
7
8     public static void main(String[] args) {
9       new StudentServer();
10    }
11
12    public StudentServer() {
13      try {
14        // Create a server socket
15        ServerSocket serverSocket = new ServerSocket(8000);
16        System.out.println("Server started ");
17
18        // Create an object output stream
19        outputToFile = new ObjectOutputStream(
20          new FileOutputStream("student.dat", true));
21
22        while (true) {
23          // Listen for a new connection request
24          Socket socket = serverSocket.accept();
25
26          // Create an input stream from the socket
27          inputFromClient =
28            new ObjectInputStream(socket.getInputStream());
29
30          // Read from input
31          Object object = inputFromClient.readObject();
32
33          // Write to the file
34          outputToFile.writeObject(object);
35          System.out.println("A new student object is stored");
36        }
37      }
38      catch(ClassNotFoundException ex) {
39        ex.printStackTrace();
40      }
41      catch(IOException ex) {
42        ex.printStackTrace();
43      }
44      finally {
45        try {
46          inputFromClient.close();
47          outputToFile.close();
48        }
49        catch (Exception ex) {
50          ex.printStackTrace();
51        }
52      }
53    }
54  }
```

server socket

output to file

connect to client

input stream

get from client

write to file

On the client side, when the user clicks the *Register to the Server* button, the client creates a socket to connect to the host (line 67), creates an **ObjectOutputStream** on the output stream of the socket (lines 70 and 71), and invokes the **writeObject** method to send the **StudentAddress** object to the server through the object output stream (line 83).

On the server side, when a client connects to the server, the server creates an **ObjectInputStream** on the input stream of the socket (lines 27 and 28), invokes the **readObject** method to receive the **StudentAddress** object through the object input stream (line 31), and writes the object to a file (line 34).

Check Point

31.9 How does a server receive connection from a client? How does a client connect to a server?

31.10 How do you find the host name of a client program from the server?

31.11 How do you send and receive an object?

31.6 Case Study: Distributed Tic-Tac-Toe Games

Key Point

This section develops a program that enables two players to play the tic-tac-toe game on the Internet.

In Section 16.12, Case Study: Developing a Tic-Tac-Toe Game, you developed a program for a tic-tac-toe game that enables two players to play the game on the same machine. In this section, you will learn how to develop a distributed tic-tac-toe game using multithreads and networking with socket streams. A distributed tic-tac-toe game enables users to play on different machines from anywhere on the Internet.

You need to develop a server for multiple clients. The server creates a server socket and accepts connections from every two players to form a session. Each session is a thread that communicates with the two players and determines the status of the game. The server can establish any number of sessions, as shown in Figure 31.13.

For each session, the first client connecting to the server is identified as player 1 with token `X`, and the second client connecting is identified as player 2 with token `O`. The server notifies the players of their respective tokens. Once two clients are connected to it, the server starts a thread to facilitate the game between the two players by performing the steps repeatedly, as shown in Figure 31.13.

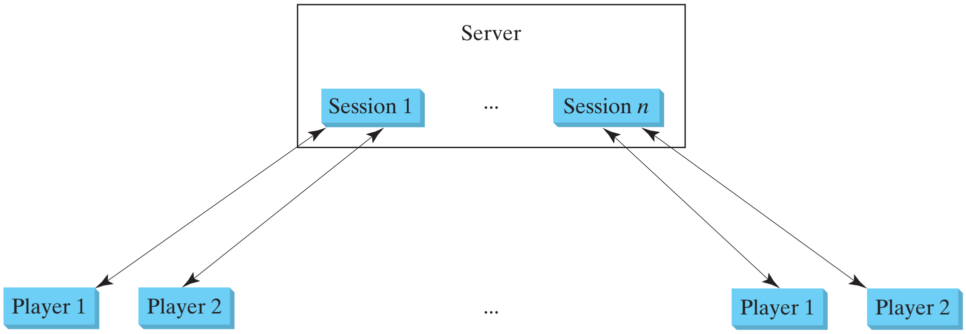


FIGURE 31.12 The server can create many sessions, each of which facilitates a tic-tac-toe game for two players.

The server does not have to be a graphical component, but creating it in a GUI in which game information can be viewed is user-friendly. You can create a scroll pane to hold a text area in the GUI and display game information in the text area. The server creates a thread to handle a game session when two players are connected to the server.

The client is responsible for interacting with the players. It creates a user interface with nine cells and displays the game title and status to the players in the labels. The client class is very similar to the `TicTacToe` class presented in the case study in Listing 16.13. However, the client in this example does not determine the game status (win or draw); it simply passes the moves to the server and receives the game status from the server.

Based on the foregoing analysis, you can create the following classes:

- `TicTacToeServer` serves all the clients in Listing 31.9.
- `HandleASession` facilitates the game for two players. This class is defined in TicTacToeServer.java.

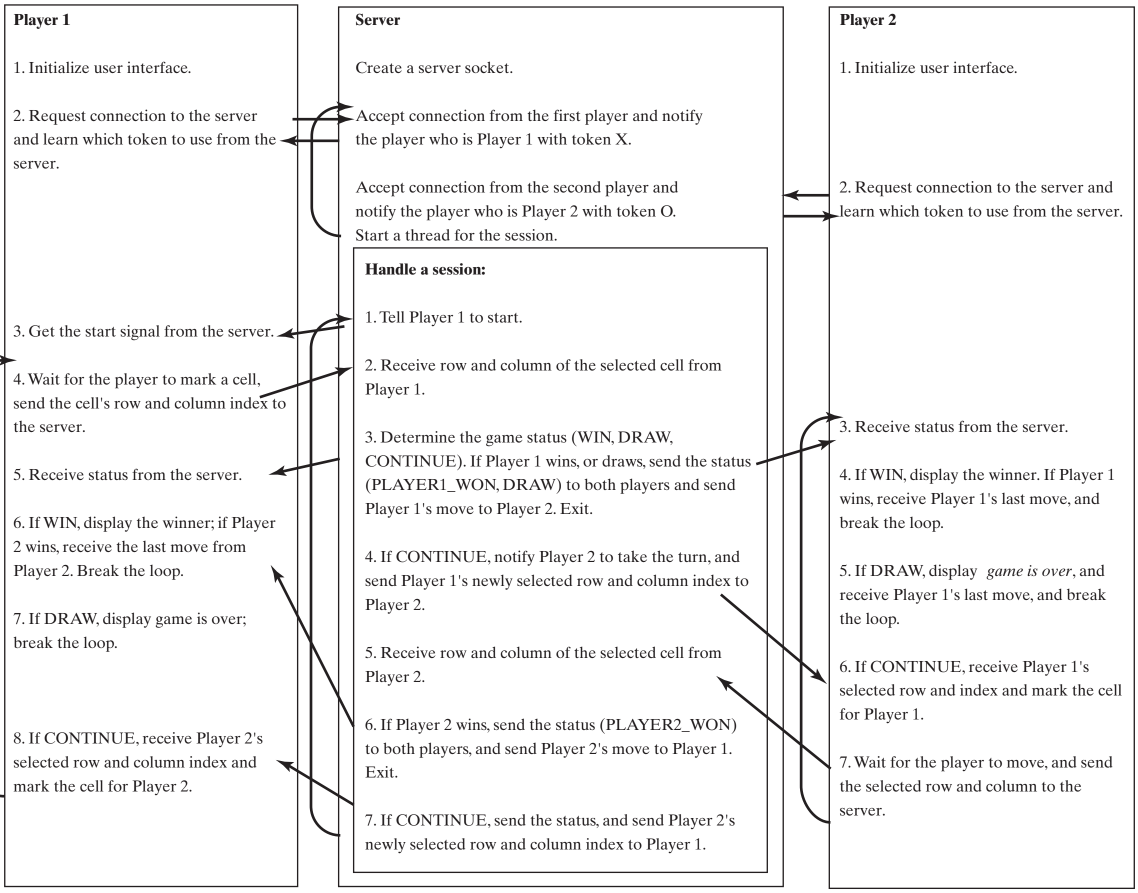


FIGURE 31.13 The server starts a thread to facilitate communications between the two players.

- `TicTacToeClient` models a player in Listing 31.10.
- `Cell` models a cell in the game. It is an inner class in `TicTacToeClient`.
- `TicTacToeConstants` is an interface that defines the constants shared by all the classes in the example in Listing 31.8.

The relationships of these classes are shown in Figure 31.14.

LISTING 31.8 TicTacToeConstants.java

```java
public interface TicTacToeConstants {
  public static int PLAYER1 = 1; // Indicate player 1
  public static int PLAYER2 = 2; // Indicate player 2
  public static int PLAYER1_WON = 1; // Indicate player 1 won
  public static int PLAYER2_WON = 2; // Indicate player 2 won
  public static int DRAW = 3; // Indicate a draw
  public static int CONTINUE = 4; // Indicate to continue
}
```

LISTING 31.9 TicTacToeServer.java

```java
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
```

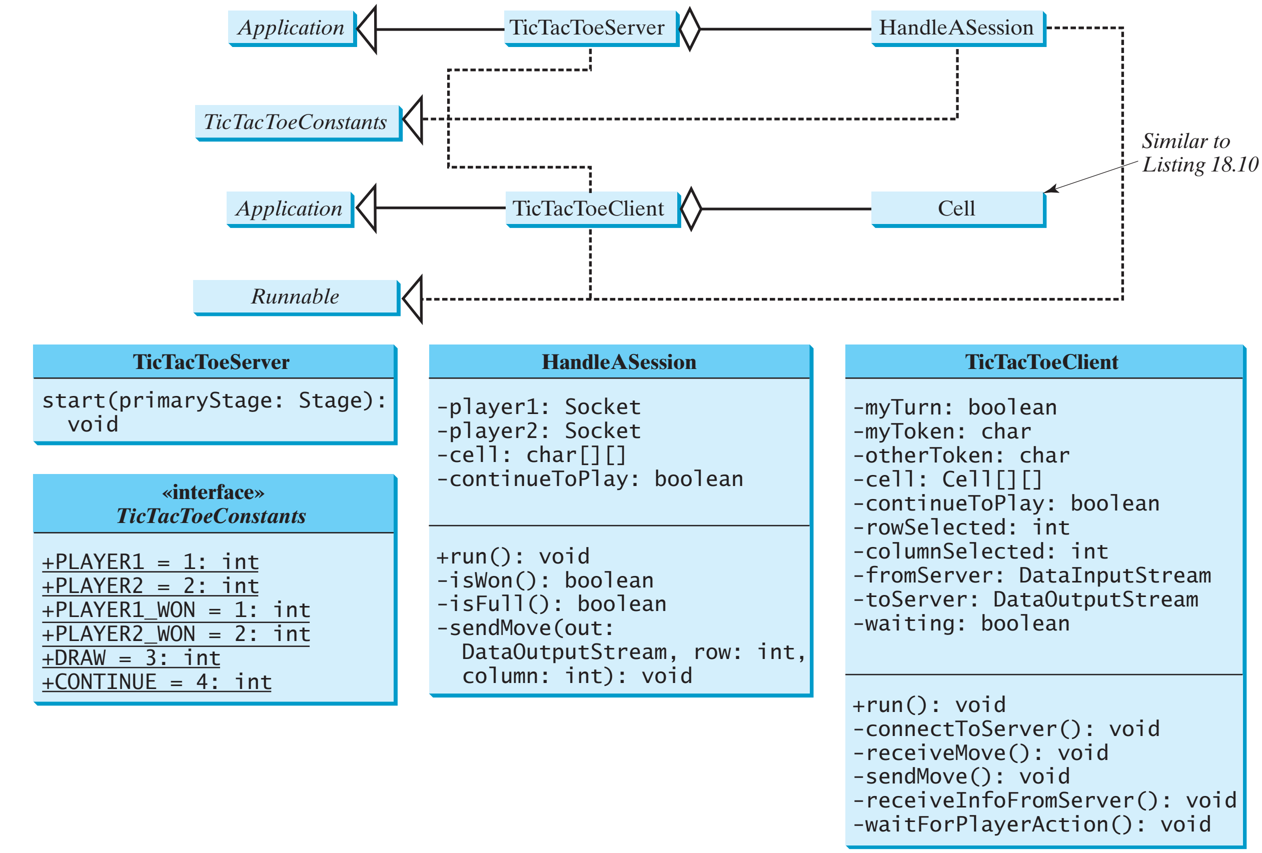


FIGURE 31.14 `TicTacToeServer` creates an instance of `HandleASession` for each session of two players. `TicTacToeClient` creates nine cells in the UI.

create UI

server socket

```
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class TicTacToeServer extends Application
    implements TicTacToeConstants {
  private int sessionNo = 1; // Number a session

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TextArea taLog = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(taLog), 450, 200);
    primaryStage.setTitle("TicTacToeServer"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
      try {
        // Create a server socket
        ServerSocket serverSocket = new ServerSocket(8000);
```

```
     Platform.runLater(() -> taLog.appendText(new Date() +
       ": Server started at socket 8000\n"));

     // Ready to create a session for every two players
     while (true) {
       Platform.runLater(() -> taLog.appendText(new Date() +
        ": Wait for players to join session " + sessionNo + '\n'));

       // Connect to player 1
       Socket player1 = serverSocket.accept();

       Platform.runLater(() -> {
         taLog.appendText(new Date() + ": Player 1 joined session "
           + sessionNo + '\n');
         taLog.appendText("Player 1's IP address" +
           player1.getInetAddress().getHostAddress() + '\n');
       });

       // Notify that the player is Player 1
       new DataOutputStream(
         player1.getOutputStream()).writeInt(PLAYER1);

       // Connect to player 2
       Socket player2 = serverSocket.accept();

       Platform.runLater(() -> {
         taLog.appendText(new Date() +
           ": Player 2 joined session " + sessionNo + '\n');
         taLog.appendText("Player 2's IP address" +
           player2.getInetAddress().getHostAddress() + '\n');
       });

       // Notify that the player is Player 2
       new DataOutputStream(
         player2.getOutputStream()).writeInt(PLAYER2);

       // Display this session and increment session number
       Platform.runLater(() ->
         taLog.appendText(new Date() +
          ": Start a thread for session " + sessionNo++ + '\n'));

       // Launch a new thread for this session of two players
       new Thread(new HandleASession(player1, player2)).start();
     }
    }
    catch(IOException ex) {
     ex.printStackTrace();
    }
   }).start();
  }

  // Define the thread class for handling a new session for two players
  class HandleASession implements Runnable, TicTacToeConstants {
   private Socket player1;
   private Socket player2;

   // Create and initialize cells
   private char[][] cell =  new char[3][3];

   private DataInputStream fromPlayer1;
```

connect to client

to player1

connect to client

to player2

a session for two players

IO streams

X won?

Is full?

```
  private DataOutputStream toPlayer1;
  private DataInputStream fromPlayer2;
  private DataOutputStream toPlayer2;

  // Continue to play
  private boolean continueToPlay = true;

  /** Construct a thread */
  public HandleASession(Socket player1, Socket player2) {
    this.player1 = player1;
    this.player2 = player2;

    // Initialize cells
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        cell[i][j] = ' ';
  }

  /** Implement the run() method for the thread */
  public void run() {
    try {
      // Create data input and output streams
      DataInputStream fromPlayer1 = new DataInputStream(
        player1.getInputStream());
      DataOutputStream toPlayer1 = new DataOutputStream(
        player1.getOutputStream());
      DataInputStream fromPlayer2 = new DataInputStream(
        player2.getInputStream());
      DataOutputStream toPlayer2 = new DataOutputStream(
        player2.getOutputStream());

      // Write anything to notify player 1 to start
      // This is just to let player 1 know to start
      toPlayer1.writeInt(1);

      // Continuously serve the players and determine and report
      // the game status to the players
      while (true) {
        // Receive a move from player 1
        int row = fromPlayer1.readInt();
        int column = fromPlayer1.readInt();
        cell[row][column] = 'X';

        // Check if Player 1 wins
        if (isWon('X')) {
          toPlayer1.writeInt(PLAYER1_WON);
          toPlayer2.writeInt(PLAYER1_WON);
          sendMove(toPlayer2, row, column);
          break; // Break the loop
        }
        else if (isFull()) { // Check if all cells are filled
          toPlayer1.writeInt(DRAW);
          toPlayer2.writeInt(DRAW);
          sendMove(toPlayer2, row, column);
          break;
        }
        else {
          // Notify player 2 to take the turn
          toPlayer2.writeInt(CONTINUE);

```

```
          // Send player 1's selected row and column to player 2
          sendMove(toPlayer2, row, column);
        }

        // Receive a move from Player 2
        row = fromPlayer2.readInt();
        column = fromPlayer2.readInt();
        cell[row][column] = 'O';

        // Check if Player 2 wins
```

O won?

```
        if (isWon('O')) {
          toPlayer1.writeInt(PLAYER2_WON);
          toPlayer2.writeInt(PLAYER2_WON);
          sendMove(toPlayer1, row, column);
          break;
        }
        else {
          // Notify player 1 to take the turn
          toPlayer1.writeInt(CONTINUE);

          // Send player 2's selected row and column to player 1
          sendMove(toPlayer1, row, column);
        }
      }
    }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }

  /** Send the move to other player */
```

send a move

```
  private void sendMove(DataOutputStream out, int row, int column)
      throws IOException {
    out.writeInt(row); // Send row index
    out.writeInt(column); // Send column index
  }

  /** Determine if the cells are all occupied */
  private boolean isFull() {
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        if (cell[i][j] == ' ')
          return false; // At least one cell is not filled

    // All cells are filled
    return true;
  }

  /** Determine if the player with the specified token wins */
  private boolean isWon(char token) {
    // Check all rows
    for (int i = 0; i < 3; i++)
      if ((cell[i][0] == token)
          && (cell[i][1] == token)
          && (cell[i][2] == token)) {
        return true;
      }

    /** Check all columns */
    for (int j = 0; j < 3; j++)
```

```java
      if ((cell[0][j] == token)
          && (cell[1][j] == token)
          && (cell[2][j] == token)) {
        return true;
      }

    /** Check major diagonal */
    if ((cell[0][0] == token)
        && (cell[1][1] == token)
        && (cell[2][2] == token)) {
      return true;
    }

    /** Check subdiagonal */
    if ((cell[0][2] == token)
        && (cell[1][1] == token)
        && (cell[2][0] == token)) {
      return true;
    }

    /** All checked, but no winner */
    return false;
  }
 }
}
```

Listing 31.10 TicTacToeClient.java

```java
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class TicTacToeClient extends Application
    implements TicTacToeConstants {
  // Indicate whether the player has the turn
  private boolean myTurn = false;

  // Indicate the token for the player
  private char myToken = ' ';

  // Indicate the token for the other player
  private char otherToken = ' ';

  // Create and initialize cells
  private Cell[][] cell =  new Cell[3][3];

```

```java
  // Create and initialize a title label
  private Label lblTitle = new Label();

  // Create and initialize a status label
  private Label lblStatus = new Label();

  // Indicate selected row and column by the current move
  private int rowSelected;
  private int columnSelected;

  // Input and output streams from/to server
  private DataInputStream fromServer;
  private DataOutputStream toServer;

  // Continue to play?
  private boolean continueToPlay = true;

  // Wait for the player to mark a cell
  private boolean waiting = true;

  // Host name or ip
  private String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Pane to hold cell
    GridPane pane = new GridPane();
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        pane.add(cell[i][j] = new Cell(i, j), j, i);

    BorderPane borderPane = new BorderPane();
    borderPane.setTop(lblTitle);
    borderPane.setCenter(pane);
    borderPane.setBottom(lblStatus);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 320, 350);
    primaryStage.setTitle("TicTacToeClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Connect to the server
    connectToServer();
  }

  private void connectToServer() {
    try {
      // Create a socket to connect to the server
      Socket socket = new Socket(host, 8000);

      // Create an input stream to receive data from the server
      fromServer = new DataInputStream(socket.getInputStream());

      // Create an output stream to send data to the server
      toServer = new DataOutputStream(socket.getOutputStream());
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
```

create UI

connect to server

input from server

output to server

```
    // Control the game on a separate thread
    new Thread(() -> {
      try {
        // Get notification from the server
        int player = fromServer.readInt();

        // Am I player 1 or 2?
        if (player == PLAYER1) {
          myToken = 'X';
          otherToken = 'O';
          Platform.runLater(() -> {
            lblTitle.setText("Player 1 with token 'X'");
            lblStatus.setText("Waiting for player 2 to join");
          });

          // Receive startup notification from the server
          fromServer.readInt(); // Whatever read is ignored

          // The other player has joined
          Platform.runLater(() ->
            lblStatus.setText("Player 2 has joined. I start first"));

          // It is my turn
          myTurn = true;
        }
        else if (player == PLAYER2) {
          myToken = 'O';
          otherToken = 'X';
          Platform.runLater(() -> {
            lblTitle.setText("Player 2 with token 'O'");
            lblStatus.setText("Waiting for player 1 to move");
          });
        }

        // Continue to play
        while (continueToPlay) {
          if (player == PLAYER1) {
            waitForPlayerAction(); // Wait for player 1 to move
            sendMove(); // Send the move to the server
            receiveInfoFromServer(); // Receive info from the server
          }
          else if (player == PLAYER2) {
            receiveInfoFromServer(); // Receive info from the server
            waitForPlayerAction(); // Wait for player 2 to move
            sendMove(); // Send player 2's move to the server
          }
        }
      }
      catch (Exception ex) {
        ex.printStackTrace();
      }
    }).start();
  }

  /** Wait for the player to mark a cell */
  private void waitForPlayerAction() throws InterruptedException {
    while (waiting) {
      Thread.sleep(100);
    }
```

```java

    waiting = true;
  }

  /** Send this player's move to the server */
  private void sendMove() throws IOException {
    toServer.writeInt(rowSelected); // Send the selected row
    toServer.writeInt(columnSelected); // Send the selected column
  }

  /** Receive info from the server */
  private void receiveInfoFromServer() throws IOException {
    // Receive game status
    int status = fromServer.readInt();

    if (status == PLAYER1_WON) {
      // Player 1 won, stop playing
      continueToPlay = false;
      if (myToken == 'X') {
        Platform.runLater(() -> lblStatus.setText("I won! (X)"));
      }
      else if (myToken == 'O') {
        Platform.runLater(() ->
          lblStatus.setText("Player 1 (X) has won!"));
        receiveMove();
      }
    }
    else if (status == PLAYER2_WON) {
      // Player 2 won, stop playing
      continueToPlay = false;
      if (myToken == 'O') {
        Platform.runLater(() -> lblStatus.setText("I won! (O)"));
      }
      else if (myToken == 'X') {
        Platform.runLater(() ->
          lblStatus.setText("Player 2 (O) has won!"));
        receiveMove();
      }
    }
    else if (status == DRAW) {
      // No winner, game is over
      continueToPlay = false;
      Platform.runLater(() ->
        lblStatus.setText("Game is over, no winner!"));

      if (myToken == 'O') {
        receiveMove();
      }
    }
    else {
      receiveMove();
      Platform.runLater(() -> lblStatus.setText("My turn"));
      myTurn = true; // It is my turn
    }
  }

  private void receiveMove() throws IOException {
    // Get the other player's move
    int row = fromServer.readInt();
    int column = fromServer.readInt();
```

```
      Platform.runLater(() -> cell[row][column].setToken(otherToken));
  }

  // An inner class for a cell
```

model a cell

```
  public class Cell extends Pane {
    // Indicate the row and column of this cell in the board
    private int row;
    private int column;

    // Token used for this cell
    private char token = ' ';

    public Cell(int row, int column) {
      this.row = row;
      this.column = column;
      this.setPrefSize(2000, 2000); // What happens without this?
      setStyle("-fx-border-color: black"); // Set cell's border
```

register listener

```
      this.setOnMouseClicked(e -> handleMouseClick());
    }

    /** Return token */
    public char getToken() {
      return token;
    }

    /** Set a new token */
    public void setToken(char c) {
      token = c;
      repaint();
    }

    protected void repaint() {
      if (token == 'X') {
```

draw X

```
        Line line1 = new Line(10, 10,
          this.getWidth() - 10, this.getHeight() - 10);
        line1.endXProperty().bind(this.widthProperty().subtract(10));
        line1.endYProperty().bind(this.heightProperty().subtract(10));
        Line line2 = new Line(10, this.getHeight() - 10,
          this.getWidth() - 10, 10);
        line2.startYProperty().bind(
          this.heightProperty().subtract(10));
        line2.endXProperty().bind(this.widthProperty().subtract(10));

        // Add the lines to the pane
        this.getChildren().addAll(line1, line2);
      }
      else if (token == 'O') {
```

draw O

```
        Ellipse ellipse = new Ellipse(this.getWidth() / 2,
          this.getHeight() / 2, this.getWidth() / 2 - 10,
          this.getHeight() / 2 - 10);
        ellipse.centerXProperty().bind(
          this.widthProperty().divide(2));
        ellipse.centerYProperty().bind(
            this.heightProperty().divide(2));
        ellipse.radiusXProperty().bind(
            this.widthProperty().divide(2).subtract(10));
        ellipse.radiusYProperty().bind(
            this.heightProperty().divide(2).subtract(10));
        ellipse.setStroke(Color.BLACK);
        ellipse.setFill(Color.WHITE);
```

```
        getChildren().add(ellipse); // Add the ellipse to the pane
      }
    }

    /* Handle a mouse click event */
    private void handleMouseClick() {
      // If cell is not occupied and the player has the turn
      if (token == ' ' && myTurn) {
        setToken(myToken);  // Set the player's token in the cell
        myTurn = false;
        rowSelected = row;
        columnSelected = column;
        lblStatus.setText("Waiting for the other player to move");
        waiting = false; // Just completed a successful move
      }
    }
  }
}
```

mouse clicked handler

The server can serve any number of sessions simultaneously. Each session takes care of two players. The client can be deployed to run as a Java applet. To run a client as a Java applet from a Web browser, the server must run from a Web server. Figures 31.15 and 31.16 show sample runs of the server and the clients.

FIGURE 31.15 `TicTacToeServer` accepts connection requests and creates sessions to serve pairs of players.

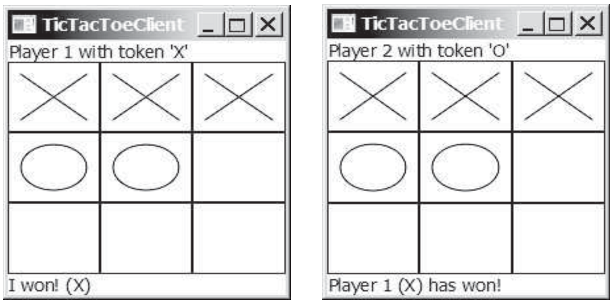


FIGURE 31.16 `TicTacToeClient` can run as an applet or standalone.

The `TicTacToeConstants` interface defines the constants shared by all the classes in the project. Each class that uses the constants needs to implement the interface. Centrally defining constants in an interface is a common practice in Java.

Once a session is established, the server receives moves from the players in alternation. Upon receiving a move from a player, the server determines the status of the game. If the game is not finished, the server sends the status (`CONTINUE`) and the player's move to the other player. If the game is won or a draw, the server sends the status (`PLAYER1_WON`, `PLAYER2_WON`, or `DRAW`) to both players.

The implementation of Java network programs at the socket level is tightly synchronized. An operation to send data from one machine requires an operation to receive data from the other machine. As shown in this example, the server and the client are tightly synchronized to send or receive data.

Check Point

33.11 What would happen if the preferred size for a cell is not set in line 227 in Listing 31.10?

33.12 If a player does not have the turn but clicks on an empty cell, what will the client program in Listing 31.10 do?

Key Terms

client socket 1141
domain name 1140
domain name server 1140
localhost 1141
IP address 1140
port 1140
packet-based communication 1140
server socket 1140
socket 1140
stream-based communication 1140
TCP 1140
UDP 1140

Chapter Summary

1. Java supports stream sockets and datagram sockets. *Stream sockets* use TCP (Transmission Control Protocol) for data transmission, whereas *datagram sockets* use UDP (User Datagram Protocol). Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission.

2. To create a server, you must first obtain a server socket, using `new ServerSocket(port)`. After a server socket is created, the server can start to listen for connections, using the `accept()` method on the server socket. The client requests a connection to a server by using `new Socket(serverName, port)` to create a client socket.

3. Stream socket communication is very much like input/output stream communication after the connection between a server and a client is established. You can obtain an input stream using the `getInputStream()` method and an output stream using the `getOutputStream()` method on the socket.

4. A server must often work with multiple clients at the same time. You can use threads to handle the server's multiple clients simultaneously by creating a thread for each connection.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/intro10e/quiz.html.