CHAPTER 22

MULTITHREADING

© Rubén Hidalgo/iStockphoto.

CHAPTER GOALS

- To understand how multiple threads can execute in parallel
- To learn to implement threads
- To understand race conditions and deadlocks
- To avoid corruption of shared objects by using locks and conditions
- To use threads for programming animations

CHAPTER CONTENTS

It is often useful for a program to carry out two or more tasks at the same time. For example, a web browser can load multiple images on a web page at the same time. Or an animation program can show moving figures, with separate tasks computing the positions of each separate figure. In this chapter, you will see how to implement this behavior by running tasks in multiple threads, and how you can ensure that the tasks access shared data in a controlled fashion.

22.1 Running Threads

A thread is a program unit that is executed concurrently with other parts of the program.

A **thread** is a program unit that is executed independently of other parts of the program. The Java virtual machine executes each thread for a short amount of time and then switches to another thread. This gives the illusion of executing the threads in parallel to each other. Actually, if a computer has multiple central processing units (CPUs), then some of the threads *can* run in parallel, one on each processor.

Running a thread is simple in Java—follow these steps:

1. Write a class that implements the `Runnable` interface. That interface has a single method called `run`:

```
public interface Runnable
{
   void run();
}
```

2. Place the code for your task into the `run` method of your class:

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      Task statements.
      . . .
   }
}
```

3. Create an object of your runnable class:

```
Runnable r = new MyRunnable();
```

4. Construct a `Thread` object from the runnable object:

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

5. Call the `start` method to start the thread:

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

The `start` method of the `Thread` class starts a new thread that executes the `run` method of the associated `Runnable` object.

Let's look at a concrete example. We want to print ten greetings of "Hello, World!", one greeting every second. We add a time stamp to each greeting to see when it is printed.

```
Wed Apr 15 23:12:03 PST 2015 Hello, World!
Wed Apr 15 23:12:04 PST 2015 Hello, World!
Wed Apr 15 23:12:05 PST 2015 Hello, World!
Wed Apr 15 23:12:06 PST 2015 Hello, World!
Wed Apr 15 23:12:07 PST 2015 Hello, World!
Wed Apr 15 23:12:08 PST 2015 Hello, World!
```

```
Wed Apr 15 23:12:09 PST 2015 Hello, World!
Wed Apr 15 23:12:10 PST 2015 Hello, World!
Wed Apr 15 23:12:11 PST 2015 Hello, World!
Wed Apr 15 23:12:12 PST 2015 Hello, World!
```

Using the instructions for creating a thread, define a class that implements the `Runnable` interface:

```
public class GreetingRunnable implements Runnable
{
   private String greeting;

   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

   public void run()
   {
      Task statements.
      . . .
   }
}
```

The `run` method should loop ten times through the following task actions:

- Print a time stamp.
- Print the greeting.
- Wait a second.

Get the time stamp by constructing an object of the `java.util.Date` class. The `Date` constructor without arguments produces a date that is set to the current date and time.

```
Date now = new Date();
System.out.println(now + " " + greeting);
```

The `sleep` method puts the current thread to sleep for a given number of milliseconds.

To wait a second, we use the static `sleep` method of the `Thread` class. The call

```
Thread.sleep(milliseconds)
```

puts the current thread to sleep for a given number of milliseconds. In our case, it should sleep for 1,000 milliseconds, or one second.

When a thread is interrupted, the most common response is to terminate the `run` method.

There is, however, one technical problem. Putting a thread to sleep is potentially risky—a thread might sleep for so long that it is no longer useful and should be terminated. As you will see in Section 22.2, to terminate a thread, you interrupt it. When a sleeping thread is interrupted, an `InterruptedException` is generated. You need to catch that exception in your `run` method and terminate the thread.

The simplest way to handle thread interruptions is to give your `run` method the following form:

```
public void run()
{
   try
   {
      Task statements.
   }
   catch (InterruptedException exception)
   {
   }
   Clean up, if necessary.
}
```

We follow that structure in our example. Here is the complete code for our runnable class:

section_1/GreetingRunnable.java

```
1  import java.util.Date;
2
3  /**
4     A runnable that repeatedly prints a greeting.
5  */
6  public class GreetingRunnable implements Runnable
7  {
8     private static final int REPETITIONS = 10;
9     private static final int DELAY = 1000;
10
11    private String greeting;
12
13    /**
14       Constructs the runnable object.
15       @param aGreeting the greeting to display
16    */
17    public GreetingRunnable(String aGreeting)
18    {
19       greeting = aGreeting;
20    }
21
22    public void run()
23    {
24       try
25       {
26          for (int i = 1; i <= REPETITIONS; i++)
27          {
28             Date now = new Date();
29             System.out.println(now + " " + greeting);
30             Thread.sleep(DELAY);
31          }
32       }
33       catch (InterruptedException exception)
34       {
35       }
36    }
37 }
```

To start a thread, first construct an object of the runnable class.

```
Runnable r = new GreetingRunnable("Hello, World!");
```

Then construct a thread and call the `start` method.

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

Now a new thread is started, executing the code in the `run` method of your runnable class in parallel with any other threads in your program.

In the `GreetingThreadRunner` program, we start two threads: one that prints "Hello" and one that prints "Goodbye".

section_1/GreetingThreadRunner.java

```
1  /**
2     This program runs two greeting threads in parallel.
3  */
4  public class GreetingThreadRunner
5  {
6     public static void main(String[] args)
7     {
8        GreetingRunnable r1 = new GreetingRunnable("Hello");
9        GreetingRunnable r2 = new GreetingRunnable("Goodbye");
10       Thread t1 = new Thread(r1);
11       Thread t2 = new Thread(r2);
12       t1.start();
13       t2.start();
14    }
15 }
```

Program Run

```
Wed Apr 15 12:04:46 PST 2015 Hello
Wed Apr 15 12:04:46 PST 2015 Goodbye
Wed Apr 15 12:04:47 PST 2015 Hello
Wed Apr 15 12:04:47 PST 2015 Goodbye
Wed Apr 15 12:04:48 PST 2015 Hello
Wed Apr 15 12:04:48 PST 2015 Goodbye
Wed Apr 15 12:04:49 PST 2015 Hello
Wed Apr 15 12:04:49 PST 2015 Goodbye
Wed Apr 15 12:04:50 PST 2015 Hello
Wed Apr 15 12:04:50 PST 2015 Goodbye
Wed Apr 15 12:04:51 PST 2015 Hello
Wed Apr 15 12:04:51 PST 2015 Goodbye
Wed Apr 15 12:04:52 PST 2015 Goodbye
Wed Apr 15 12:04:52 PST 2015 Hello
Wed Apr 15 12:04:53 PST 2015 Hello
Wed Apr 15 12:04:53 PST 2015 Goodbye
Wed Apr 15 12:04:54 PST 2015 Hello
Wed Apr 15 12:04:54 PST 2015 Goodbye
Wed Apr 15 12:04:55 PST 2015 Goodbye
Wed Apr 15 12:04:55 PST 2015 Hello
```

The thread scheduler runs each thread for a short amount of time, called a time slice.

Because both threads are running in parallel, the two message sets are interleaved. However, if you look closely, you will find that the two threads aren't *exactly* interleaved. Sometimes, the second thread seems to jump ahead of the first thread. This shows an important characteristic of threads. The thread scheduler gives no guarantee about the order in which threads are executed. Each thread runs for a short amount of time, called a **time slice**. Then the scheduler activates another thread. However, there will always be slight variations in running times, especially when calling operating system services (such as input and output). Thus, you should expect that the order in which each thread gains control is somewhat random.

1. What happens if you change the call to the `sleep` method in the `run` method to `Thread.sleep(1)`?
2. What would be the result of the program if the `main` method called

```
r1.run();
r2.run();
```

instead of starting threads?

Practice It Now you can try these exercises at the end of the chapter: R22.2, R22.3, E22.7.

Programming Tip 22.1

Use the `Runnable` Interface

In Java, you can define the task statements of a thread in two ways. As you have seen already, you can place the statements into the `run` method of a class that implements the `Runnable` interface. Then you use an object of that class to construct a `Thread` object. You can also form a subclass of the `Thread` class, and place the task statements into the `run` method of your subclass:

```
public class MyThread extends Thread
{
   public void run()
   {
      Task statements.
      . . .
   }
}
```

Then you construct an object of the subclass and call the `start` method:

```
Thread t = new MyThread();
t.start();
```

This approach is marginally easier than using a `Runnable`, and it also seems quite intuitive. However, if a program needs a large number of threads, or if a program executes in a resource-constrained device, such as a cell phone, it can be quite expensive to construct a separate thread for each task. Special Topic 22.1 shows how to use a *thread pool* to overcome this problem. A thread pool uses a small number of threads to execute a larger number of runnables.

The `Runnable` interface is designed to encapsulate the concept of a sequence of statements that can run in parallel with other tasks, without equating it with the concept of a thread, a potentially expensive resource that is managed by the operating system.

Special Topic 22.1

Thread Pools

A program that creates a huge number of short-lived threads can be inefficient. Threads are managed by the operating system, and there is a cost for creating threads. Each thread requires memory, and thread creation takes time. This cost can be reduced by using a *thread pool*. A thread pool creates a number of threads and keeps them alive. When you add a `Runnable` object to the thread pool, the next idle thread executes its `run` method.

For example, the following statements submit two runnables to a thread pool:

```
Runnable r1 = new GreetingRunnable("Hello");
Runnable r2 = new GreetingRunnable("Goodbye");
ExecutorService pool = Executors.newFixedThreadPool(MAX_THREADS);
pool.execute(r1);
pool.execute(r2);
```

If many runnables are submitted for execution, then the pool may not have enough threads available. In that case, some runnables are placed in a queue until a thread is idle. As a result, the cost of creating threads is minimized. However, the runnables that are run by a particular thread are executed sequentially, not in parallel.

Thread pools are particularly important for server programs, such as database and web servers, that repeatedly execute requests from multiple clients. Rather than spawning a new thread for each request, the requests are implemented as runnable objects and submitted to a thread pool.

22.2 Terminating Threads

A thread terminates when its run method terminates.

When the run method of a thread has finished executing, the thread terminates. This is the normal way of terminating a thread—implement the run method so that it returns when it determines that no more work needs to be done.

However, sometimes you need to terminate a running thread. For example, you may have several threads trying to find a solution to a problem. As soon as the first one has succeeded, you may want to terminate the other ones. In the initial release of the Java library, the Thread class had a stop method to terminate a thread. However, that method is now *deprecated*—computer scientists have found that stopping a thread can lead to dangerous situations when multiple threads share objects. (We will discuss access to shared objects in Section 22.3.) Instead of simply stopping a thread, you should notify the thread that it should be terminated. The thread needs to cooperate, by releasing any resources that it is currently using and doing any other required cleanup. In other words, a thread should be in charge of terminating itself.

To notify a thread that it should clean up and terminate, you use the interrupt method.

```
t.interrupt();
```

The run method can check whether its thread has been interrupted by calling the interrupted method.

This method does not actually cause the thread to terminate—it merely sets a boolean variable in the thread data structure.

The run method can check whether that flag has been set, by calling the static interrupted method. In that case, it should do any necessary cleanup and exit. For example, the run method of the GreetingRunnable could check for interruptions at the beginning of each loop iteration:

```
public void run()
{
   for (int i = 1; i <= REPETITIONS && !Thread.interrupted(); i++)
   {
      Do work.
   }
   Clean up.
}
```

However, if a thread is sleeping, it can't execute code that checks for interruptions. Therefore, the sleep method is terminated with an InterruptedException whenever a sleeping thread is interrupted. The sleep method also throws an InterruptedException when it is called in a thread that is already interrupted. If your run method calls sleep in each loop iteration, simply use the InterruptedException to find out whether the thread is terminated. The easiest way to do that is to surround the entire work portion of the run method with a try block, like this:

```
public void run()
{
   try
   {
      for (int i = 1; i <= REPETITIONS; i++)
      {
         Do work.
         Sleep.
      }
   }
   catch (InterruptedException exception)
   {
   }
   Clean up.
}
```

Strictly speaking, there is nothing in the Java language specification that says that a thread must terminate when it is interrupted. It is entirely up to the thread what it does when it is interrupted. Interrupting is a general mechanism for getting the thread's attention, even when it is sleeping. However, in this chapter, we will always terminate a thread that is being interrupted.

3. Suppose a web browser uses multiple threads to load the images on a web page. Why should these threads be terminated when the user hits the "Back" button?
4. Consider the following runnable.

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      try
      {
         System.out.println(1);
         Thread.sleep(1000);
         System.out.println(2);
      }
      catch (InterruptedException exception)
      {
         System.out.println(3);
      }
      System.out.println(4);
   }
}
```

Suppose a thread with this runnable is started and immediately interrupted:

```
Thread t = new Thread(new MyRunnable());
t.start();
t.interrupt();
```

What output is produced?

Practice It Now you can try these exercises at the end of the chapter: R22.4, R22.5, R22.6.

Programming Tip 22.2

Check for Thread Interruptions in the `run` Method of a Thread

By convention, a thread should terminate itself (or at least act in some other well-defined way) when it is interrupted. You should implement your threads to follow this convention.

To do so, put the thread action inside a `try` block that catches the `InterruptedException`. That exception occurs when your thread is interrupted while it is not running, for example inside a call to `sleep`. When you catch the exception, do any required cleanup and exit the `run` method.

Some programmers don't understand the purpose of the `InterruptedException` and muzzle it by placing only the call to `sleep` inside a `try` block:

```
public void run()
{
   while (. . .)
   {
      . . .
      try
      {
         Thread.sleep(delay);
      }
      catch (InterruptedException exception) {} // DON'T
      . . .
   }
}
```

Don't do that. If you do, users of your thread class can't get your thread's attention by interrupting it. It is just as easy to place the entire thread action inside a single `try` block. Then interrupting the thread terminates the thread action.

```
public void run()
{
   try
   {
      while (. . .)
      {
         . . .
         Thread.sleep(delay);
         . . .
      }
   }
   catch (InterruptedException exception) {} // OK
}
```

22.3 Race Conditions

When threads share access to a common object, they can conflict with each other. To demonstrate the problems that can arise, we will investigate a sample program in which multiple threads manipulate a bank account.

We construct a bank account that starts out with a zero balance. We create two sets of threads:

- Each thread in the first set repeatedly deposits $100.
- Each thread in the second set repeatedly withdraws $100.

Here is the `run` method of the `DepositRunnable` class:

```
public void run()
{
   try
   {
      for (int i = 1; i <= count; i++)
      {
         account.deposit(amount);
         Thread.sleep(DELAY);
      }
   }
   catch (InterruptedException exception)
   {
   }
}
```

The `WithdrawRunnable` class is similar—it withdraws money instead.

The `deposit` and `withdraw` methods of the `BankAccount` class have been modified to print messages that show what is happening. For example, here is the code for the `deposit` method:

```
public void deposit(double amount)
{
   System.out.print("Depositing " + amount);
   double newBalance = balance + amount;
   System.out.println(", new balance is " + newBalance);
   balance = newBalance;
}
```

You can find the complete source code at the end of this section.

Normally, the program output looks somewhat like this:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
. . .
Withdrawing 100.0, new balance is 0.0
```

In the end, the balance should be zero. However, when you run this program repeatedly, you may sometimes notice messed-up output, like this:

```
Depositing 100.0Withdrawing 100.0, new balance is 100.0
, new balance is -100.0
```

And if you look at the last line of the output, you will notice that the final balance is not always zero. Clearly, something problematic is happening. You may have to try the program several times to see this effect.

Here is a scenario that explains how a problem can occur.

1. A deposit thread executes the lines

   ```
   System.out.print("Depositing " + amount);
   double newBalance = balance + amount;
   ```

   in the `deposit` method of the `BankAccount` class. The value of the `balance` variable is still 0, and the value of the `newBalance` local variable is 100.
2. Immediately afterward, the deposit thread reaches the end of its time slice, and the second thread gains control.

3. A withdraw thread calls the `withdraw` method, which prints a message and withdraws $100 from the `balance` variable. It is now –100.
4. The withdraw thread goes to sleep.
5. The deposit thread regains control and picks up where it was interrupted. It now executes the lines

   ```
   System.out.println(", new balance is " + newBalance);
   balance = newBalance;
   ```

   The value of `balance` is now 100 (see Figure 1).

Thus, not only are the messages interleaved, but the balance is wrong. The balance after a withdrawal and deposit should again be 0, not 100. Because the `deposit` method

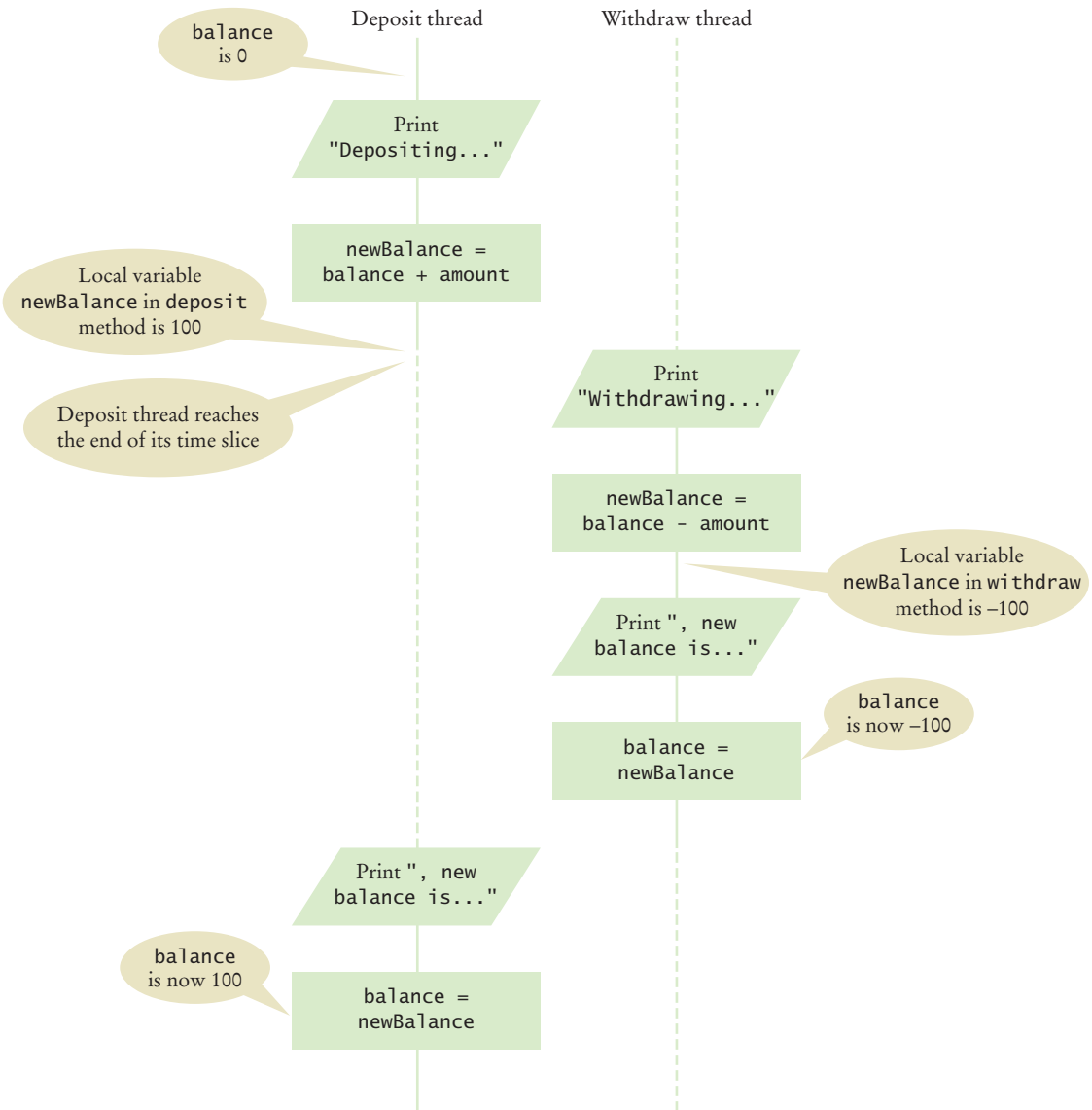


Figure 1 Corrupting the Contents of the `balance` Variable

was interrupted, it used the *old* balance (before the withdrawal) to compute the value of its local `newBalance` variable. Later, when it was activated again, it used that `newBalance` value to overwrite the changed `balance` variable.

A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

As you can see, each thread has its own local variables, but all threads share access to the `balance` instance variable. That shared access creates a problem. This problem is often called a **race condition**. All threads, in their race to complete their respective tasks, manipulate a shared variable, and the end result depends on which of them happens to win the race.

You might argue that the reason for this problem is that we made it too easy to interrupt the balance computation. Suppose the code for the `deposit` method is reorganized like this:

```
public void deposit(double amount)
{
   balance = balance + amount;
   System.out.print("Depositing " + amount
         + ", new balance is " + balance);
}
```

Suppose further that you make the same change in the `withdraw` method. If you run the resulting program, everything seems to be fine.

However, that is a *dangerous illusion*. The problem hasn't gone away; it has become much less frequent, and, therefore, more difficult to observe. It is still possible for the `deposit` method to reach the end of its time slice after it has computed the right-hand-side value

```
balance + amount
```

but before it performs the assignment

```
balance = the right-hand-side value
```

When the method regains control, it finally carries out the assignment, putting the wrong value into the `balance` variable.

section_3/BankAccountThreadRunner.java

```
/**
   This program runs threads that deposit and withdraw
   money from the same bank account.
*/
public class BankAccountThreadRunner
{
   public static void main(String[] args)
   {
      BankAccount account = new BankAccount();
      final double AMOUNT = 100;
      final int REPETITIONS = 100;
      final int THREADS = 100;

      for (int i = 1; i <= THREADS; i++)
      {
         DepositRunnable d = new DepositRunnable(
            account, AMOUNT, REPETITIONS);
         WithdrawRunnable w = new WithdrawRunnable(
            account, AMOUNT, REPETITIONS);

         Thread dt = new Thread(d);
         Thread wt = new Thread(w);
```

```
23
24          dt.start();
25          wt.start();
26       }
27    }
28 }
```

section_3/DepositRunnable.java

```
1  /**
2     A deposit runnable makes periodic deposits to a bank account.
3  */
4  public class DepositRunnable implements Runnable
5  {
6     private static final int DELAY = 1;
7     private BankAccount account;
8     private double amount;
9     private int count;
10
11    /**
12       Constructs a deposit runnable.
13       @param anAccount the account into which to deposit money
14       @param anAmount the amount to deposit in each repetition
15       @param aCount the number of repetitions
16    */
17    public DepositRunnable(BankAccount anAccount, double anAmount,
18          int aCount)
19    {
20       account = anAccount;
21       amount = anAmount;
22       count = aCount;
23    }
24
25    public void run()
26    {
27       try
28       {
29          for (int i = 1; i <= count; i++)
30          {
31             account.deposit(amount);
32             Thread.sleep(DELAY);
33          }
34       }
35       catch (InterruptedException exception) {}
36    }
37 }
```

section_3/WithdrawRunnable.java

```
1  /**
2     A withdraw runnable makes periodic withdrawals from a bank account.
3  */
4  public class WithdrawRunnable implements Runnable
5  {
6     private static final int DELAY = 1;
7     private BankAccount account;
8     private double amount;
9     private int count;
10
```

```
11    /**
12       Constructs a withdraw runnable.
13       @param anAccount the account from which to withdraw money
14       @param anAmount the amount to withdraw in each repetition
15       @param aCount the number of repetitions
16    */
17    public WithdrawRunnable(BankAccount anAccount, double anAmount,
18          int aCount)
19    {
20       account = anAccount;
21       amount = anAmount;
22       count = aCount;
23    }
24
25    public void run()
26    {
27       try
28       {
29          for (int i = 1; i <= count; i++)
30          {
31             account.withdraw(amount);
32             Thread.sleep(DELAY);
33          }
34       }
35       catch (InterruptedException exception) {}
36    }
37 }
```

section_3/BankAccount.java

```
1  /**
2     A bank account has a balance that can be changed by
3     deposits and withdrawals.
4  */
5  public class BankAccount
6  {
7     private double balance;
8
9     /**
10       Constructs a bank account with a zero balance.
11    */
12    public BankAccount()
13    {
14       balance = 0;
15    }
16
17    /**
18       Deposits money into the bank account.
19       @param amount the amount to deposit
20    */
21    public void deposit(double amount)
22    {
23       System.out.print("Depositing " + amount);
24       double newBalance = balance + amount;
25       System.out.println(", new balance is " + newBalance);
26       balance = newBalance;
27    }
28
```

```
29      /**
30         Withdraws money from the bank account.
31         @param amount the amount to withdraw
32      */
33      public void withdraw(double amount)
34      {
35         System.out.print("Withdrawing " + amount);
36         double newBalance = balance - amount;
37         System.out.println(", new balance is " + newBalance);
38         balance = newBalance;
39      }
40
41      /**
42         Gets the current balance of the bank account.
43         @return the current balance
44      */
45      public double getBalance()
46      {
47         return balance;
48      }
49   }
```

Program Run

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
. . .
Withdrawing 100.0, new balance is 400.0
Depositing 100.0, new balance is 500.0
Withdrawing 100.0, new balance is 400.0
Withdrawing 100.0, new balance is 300.0
```

5. Give a scenario in which a race condition causes the bank balance to be –100 after one iteration of a deposit thread and a withdraw thread.
6. Suppose two threads simultaneously insert objects into a linked list. Using the implementation in Chapter 16, explain how the list can be damaged in the process.

Practice It Now you can try these exercises at the end of the chapter: R22.8, R22.9, E22.1.

22.4 Synchronizing Object Access

To solve problems such as the one that you observed in the preceding section, use a **lock object**. The lock object is used to control the threads that want to manipulate a shared resource.

The Java library defines a `Lock` interface and several classes that implement this interface. The `ReentrantLock` class is the most commonly used lock class, and the only one that we cover in this book. (**Locks** are a feature added in Java version 5.0. Earlier versions of Java have a lower-level facility for thread synchronization—see Special Topic 22.2).

Typically, a lock object is added to a class whose methods access shared resources, like this:

```
public class BankAccount
{
   private Lock balanceChangeLock;
   . . .
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      . . .
   }
}
```

All code that manipulates the shared resource is surrounded by calls to lock and unlock the lock object:

```
balanceChangeLock.lock();
Manipulate the shared resource.
balanceChangeLock.unlock();
```

However, this sequence of statements has a potential flaw. If the code between the calls to `lock` and `unlock` throws an exception, the call to `unlock` never happens. This is a serious problem. After an exception, the current thread continues to hold the lock, and no other thread can acquire it. To overcome this problem, place the call to `unlock` into a `finally` clause:

```
balanceChangeLock.lock();
try
{
   Manipulate the shared resource.
}
finally
{
   balanceChangeLock.unlock();
}
```

For example, here is the code for the `deposit` method:

```
public void deposit(double amount)
{
   balanceChangeLock.lock();
   try
   {
      System.out.print("Depositing " + amount);
      double newBalance = balance + amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

By calling the `lock` method, a thread acquires a `Lock` object. Then no other thread can acquire the lock until the first thread releases the lock.

When a thread calls the `lock` method, it *owns the lock* until it calls the `unlock` method. If a thread calls `lock` while another thread owns the lock, the first thread is temporarily deactivated. The thread scheduler periodically reactivates such a thread so that it can again try to acquire the lock. If the lock is still unavailable, the thread is again deactivated. Eventually, when the lock is available because the original thread unlocked it, the waiting thread can acquire the lock.

Figure 2
Visualizing Object Locks

Creatas/Punchstock.

One way to visualize this behavior is to imagine that the lock object is the lock of an old-fashioned telephone booth and the threads are people wanting to make telephone calls (see Figure 2). The telephone booth can accommodate only one person at a time. If the booth is empty, then the first person wanting to make a call goes inside and closes the door. If another person wants to make a call and finds the booth occupied, then the second person needs to wait until the first person leaves the booth. If multiple people want to gain access to the telephone booth, they all wait outside. They don't necessarily form an orderly queue; a randomly chosen person may gain access when the telephone booth becomes available again.

With the `ReentrantLock` class, a thread can call the `lock` method on a lock object that it already owns. This can happen if one method calls another, and both start by locking the same object. The thread gives up ownership if the `unlock` method has been called as often as the `lock` method.

By surrounding the code in both the `deposit` and `withdraw` methods with lock and unlock calls, we ensure that our program will always run correctly. Only one thread at a time can execute either method on a given object. Whenever a thread acquires the lock, it is guaranteed to execute the method to completion before the other thread gets a chance to modify the balance of the same bank account object.

7. If you construct two `BankAccount` objects, how many lock objects are created?
8. What happens if we omit the call `unlock` at the end of the `deposit` method?

Practice It Now you can try these exercises at the end of the chapter: E22.2, E22.6, E22.8.

22.5 Avoiding Deadlocks

You can use lock objects to ensure that shared data are in a consistent state when several threads access them. However, locks can lead to another problem. It can happen that one thread acquires a lock and then waits for another thread to do some essential work. If that other thread is currently waiting to acquire the same lock, then

neither of the two threads can proceed. Such a situation is called a **deadlock** or **deadly embrace**. Let's look at an example.

A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.

Suppose we want to disallow negative bank balances in our program. Here's a naive way of doing that. In the `run` method of the `WithdrawRunnable` class, we can check the balance before withdrawing money:

```
if (account.getBalance() >= amount)
{
   account.withdraw(amount);
}
```

This works if there is only a single thread running that withdraws money. But suppose we have multiple threads that withdraw money. Then the time slice of the current thread may expire after the check `account.getBalance() >= amount` passes, but before the `withdraw` method is called. If, in the interim, another thread withdraws more money, then the test was useless, and we still have a negative balance.

Clearly, the test should be moved inside the `withdraw` method. That ensures that the test for sufficient funds and the actual withdrawal cannot be separated. Thus, the `withdraw` method could look like this:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
      {
         Wait for the balance to grow.
      }
      . . .
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

But how can we wait for the balance to grow? We can't simply call `sleep` inside the `withdraw` method. If a thread sleeps after acquiring a lock, it blocks all other threads that want to use the same lock. In particular, no other thread can successfully execute the `deposit` method. Other threads will call `deposit`, but they will simply be blocked until the `withdraw` method exits. But the `withdraw` method doesn't exit until it has funds available. This is the deadlock situation that we mentioned earlier.

To overcome this problem, we use a **condition object**. Condition objects allow a thread to temporarily release a lock, so that another thread can proceed, and to regain the lock at a later time.

In the telephone booth analogy, suppose that the coin reservoir of the telephone is completely filled, so that no further calls can be made until a service technician removes the coins. You don't want the person in the booth to go to sleep with the door closed. Instead, think of the person leaving the booth temporarily. That gives another person (hopefully a service technician) a chance to enter the booth.

Each condition object belongs to a specific lock object. You obtain a condition object with the `newCondition` method of the `Lock` interface. For example,

```
public class BankAccount
{
```

```
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
   . . .
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition = balanceChangeLock.newCondition();
      . . .
   }
}
```

It is customary to give the condition object a name that describes the condition that you want to test (such as "sufficient funds"). You need to implement an appropriate test. For as long as the test is not fulfilled, call the `await` method on the condition object:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
      {
         sufficientFundsCondition.await();
      }
      . . .
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Calling `await` on a condition object makes the current thread wait and allows another thread to acquire the lock object.

When a thread calls `await`, it is not simply deactivated in the same way as a thread that reaches the end of its time slice. Instead, it is in a blocked state, and it will not be activated by the thread scheduler until it is unblocked. To unblock, another thread must execute the `signalAll` method *on the same condition object*. The `signalAll` method unblocks all threads waiting on the condition. They can then compete with all other threads that are waiting for the lock object. Eventually, one of them will gain access to the lock, and it will exit from the `await` method.

In our situation, the `deposit` method calls `signalAll`:

```
public void deposit(double amount)
{
   balanceChangeLock.lock();
   try
   {
      . . .
      sufficientFundsCondition.signalAll();
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

The call to `signalAll` notifies the waiting threads that sufficient funds *may be* available, and that it is worth testing the loop condition again.

A waiting thread is blocked until another thread calls `signalAll` or `signal` on the condition object for which the thread is waiting.

In the telephone booth analogy, the thread calling `await` corresponds to the person who enters the booth and finds that the phone doesn't work. That person then leaves the booth and waits outside, depressed, doing absolutely nothing, even as other people enter and leave the booth. The person knows it is pointless to try again. At some point, a service technician enters the booth, empties the coin reservoir, and shouts a signal. Now all the waiting people stop being depressed and again compete for the telephone booth.

There is also a `signal` method, which randomly picks just one thread that is waiting on the object and unblocks it. The `signal` method can be more efficient, but it is useful only if you know that *every* waiting thread can actually proceed. In general, you don't know that, and `signal` can lead to deadlocks. For that reason, we recommend that you always call `signalAll`.

The `await` method can throw an `InterruptedException`. The `withdraw` method propagates that exception, because it has no way of knowing what the thread that calls the `withdraw` method wants to do if it is interrupted.

With the calls to `await` and `signalAll` in the `withdraw` and `deposit` methods, we can launch any number of withdrawal and deposit threads without a deadlock. If you run the sample program, you will note that all transactions are carried out without ever reaching a negative balance.

section_5/BankAccount.java

```
1  import java.util.concurrent.locks.Condition;
2  import java.util.concurrent.locks.Lock;
3  import java.util.concurrent.locks.ReentrantLock;
4
5  /**
6     A bank account has a balance that can be changed by
7     deposits and withdrawals.
8  */
9  public class BankAccount
10 {
11    private double balance;
12    private Lock balanceChangeLock;
13    private Condition sufficientFundsCondition;
14
15    /**
16       Constructs a bank account with a zero balance.
17    */
18    public BankAccount()
19    {
20       balance = 0;
21       balanceChangeLock = new ReentrantLock();
22       sufficientFundsCondition = balanceChangeLock.newCondition();
23    }
24
25    /**
26       Deposits money into the bank account.
27       @param amount the amount to deposit
28    */
29    public void deposit(double amount)
30    {
31       balanceChangeLock.lock();
32       try
33       {
```

```
34          System.out.print("Depositing " + amount);
35          double newBalance = balance + amount;
36          System.out.println(", new balance is " + newBalance);
37          balance = newBalance;
38          sufficientFundsCondition.signalAll();
39       }
40       finally
41       {
42          balanceChangeLock.unlock();
43       }
44    }
45
46    /**
47       Withdraws money from the bank account.
48       @param amount the amount to withdraw
49    */
50    public void withdraw(double amount)
51          throws InterruptedException
52    {
53       balanceChangeLock.lock();
54       try
55       {
56          while (balance < amount)
57          {
58             sufficientFundsCondition.await();
59          }
60          System.out.print("Withdrawing " + amount);
61          double newBalance = balance - amount;
62          System.out.println(", new balance is " + newBalance);
63          balance = newBalance;
64       }
65       finally
66       {
67          balanceChangeLock.unlock();
68       }
69    }
70
71    /**
72       Gets the current balance of the bank account.
73       @return the current balance
74    */
75    public double getBalance()
76    {
77       return balance;
78    }
79 }
```

section_5/BankAccountThreadRunner.java

```
1  /**
2     This program runs threads that deposit and withdraw
3     money from the same bank account.
4  */
5  public class BankAccountThreadRunner
6  {
7     public static void main(String[] args)
8     {
9        BankAccount account = new BankAccount();
10       final double AMOUNT = 100;
```

```
11         final int REPETITIONS = 100;
12         final int THREADS = 100;
13
14         for (int i = 1; i <= THREADS; i++)
15         {
16            DepositRunnable d = new DepositRunnable(
17               account, AMOUNT, REPETITIONS);
18            WithdrawRunnable w = new WithdrawRunnable(
19               account, AMOUNT, REPETITIONS);
20
21            Thread dt = new Thread(d);
22            Thread wt = new Thread(w);
23
24            dt.start();
25            wt.start();
26         }
27      }
28   }
```

Program Run

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
. . .
Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
```

9. What is the essential difference between calling `sleep` and `await`?
10. Why is the `sufficientFundsCondition` object an instance variable of the `BankAccount` class and not a local variable of the `withdraw` and `deposit` methods?

Practice It Now you can try these exercises at the end of the chapter: R22.12, E22.3, E22.4, E22.5.

Common Error 22.1

Calling `await` Without Calling `signalAll`

It is intuitively clear when to call `await`. If a thread finds out that it can't do its job, it has to wait. But once a thread has called `await`, it temporarily gives up all hope and doesn't try again until some other thread calls `signalAll` on the condition object for which the thread is waiting. In the telephone booth analogy, if the service technician who empties the coin reservoir doesn't notify the waiting people, they'll wait forever.

A common error is to have threads call `await` without matching calls to `signalAll` by other threads. Whenever you call `await`, ask yourself which call to `signalAll` will signal your waiting thread.

Common Error 22.2

Calling `signalAll` Without Locking the Object

The thread that calls `signalAll` must own the lock that belongs to the condition object on which `signalAll` is called. Otherwise, an `IllegalMonitorStateException` is thrown.

In the telephone booth analogy, the service technician must shout the signal while *inside* the telephone booth after emptying the coin reservoir.

In practice, this should not be a problem. Remember that `signalAll` is called by a thread that has just changed the state of some shared data in a way that may benefit waiting threads. That change should be protected by a lock in any case. As long as you use a lock to protect all access to shared data, and you are in the habit of calling `signalAll` after every beneficial change, you won't run into problems. But if you use `signalAll` in a haphazard way, you may encounter the `IllegalMonitorStateException`.

Special Topic 22.2

Object Locks and Synchronized Methods

The `Lock` and `Condition` interfaces were added in Java version 5.0. They overcome limitations of the thread synchronization mechanism in earlier Java versions. In this note, we discuss that classic mechanism.

Every Java object has one built-in lock and one built-in condition variable. The lock works in the same way as a `ReentrantLock` object. However, to acquire the lock, you call a **synchronized method**.

You simply tag all methods that contain thread-sensitive code (such as the `deposit` and `withdraw` methods of the `BankAccount` class) with the `synchronized` reserved word.

```
public class BankAccount
{
   public synchronized void deposit(double amount)
   {
      System.out.print("Depositing " + amount);
      double newBalance = balance + amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }

   public synchronized void withdraw(double amount)
   {
      . . .
   }
   . . .
}
```

When a thread calls a synchronized method on a `BankAccount` object, it owns that object's lock until it returns from the method and thereby unlocks the object. When an object is locked by one thread, no other thread can enter a synchronized method for that object. When another thread makes a call to a synchronized method for that object, the calling thread is automatically deactivated and needs to wait until the first thread has unlocked the object again.

In other words, the `synchronized` reserved word automatically implements the `lock/try/finally/unlock` idiom for the built-in lock.

The object lock has a single condition variable that you manipulate with the `wait`, `notifyAll`, and `notify` methods of the `Object` class. If you call `x.wait()`, the current thread is added to the

set of threads that is waiting for the condition of the object x. Most commonly, you will call `wait()`, which makes the current thread wait on `this`. For example,

```
public synchronized void withdraw(double amount)
      throws InterruptedException
{
   while (balance < amount)
   {
      wait();
   }
   . . .
}
```

The call `notifyAll()` unblocks all threads that are waiting for `this`:

```
public synchronized void deposit(double amount)
{
   . . .
   notifyAll();
}
```

This classic mechanism is undeniably simpler than using explicit locks and condition variables. However, there are limitations. Each object lock has one condition variable, and you can't test whether another thread holds the lock. If these limitations are not a problem, by all means, go ahead and use the `synchronized` reserved word. If you need more control over threads, the `Lock` and `Condition` interfaces give you additional flexibility.

Special Topic 22.3

The Java Memory Model

In a computer with multiple CPUs, you have to be particularly careful when multiple threads access shared data. Because modern processors are quite a bit faster than RAM memory, each CPU has its own *memory cache* that stores copies of frequently used memory locations. If a thread changes shared data, another thread may not see the change until both processor caches are synchronized. The same effect can happen even on a computer with a single CPU—occasionally, memory values are cached in CPU registers.

The Java language specification contains a set of rules, called the *memory model*, that describes under which circumstances the virtual machine must ensure that changes to shared data are visible in other threads. One of the rules states the following:

- If a thread changes shared data and then releases a lock, and another thread acquires the same lock and reads the same data, then it is guaranteed to see the changed data.

However, if the first thread does not release a lock, then the virtual machine is not required to write cached data back to memory. Similarly, if the second thread does not acquire the lock, the virtual machine is not required to refresh its cache from memory.

Thus, you should always use locks or synchronized methods when you access data that is shared among multiple threads, even if you are not concerned about race conditions.

22.6 Application: Algorithm Animation

One popular use for thread programming is animation. A program that displays an animation shows different objects moving or changing in some way as time progresses. This is often achieved by launching one or more threads that compute how parts of the animation change.

You can use the Swing `Timer` class for simple animations without having to do any thread programming—see Exercise P22.7 for an example. However, more advanced animations are best implemented with threads.

In this section you will see a particular kind of animation, namely the visualization of the steps of an algorithm. Algorithm animation is an excellent technique for gaining a better understanding of how an algorithm works. Many algorithms can be animated—type "Java algorithm animation" into your favorite web search engine, and you'll find lots of links to web pages with animations of various algorithms.

Use a separate thread for running the algorithm that is being animated.

All algorithm animations have a similar structure. The algorithm runs in a separate thread that periodically updates an image of the current state of the algorithm and then pauses so that the user can view the image. After a short amount of time, the algorithm thread wakes up again and runs to the next point of interest in the algorithm. It then updates the image and pauses again. This sequence is repeated until the algorithm has finished.

Let's take the selection sort algorithm of Chapter 14 as an example. That algorithm sorts an array of values. It first finds the smallest element, by inspecting all elements in the array and bringing the smallest element to the leftmost position. It then finds the smallest element among the remaining elements and brings it into the second position. It keeps going in that way. As the algorithm progresses, the sorted part of the array grows.

How can you visualize this algorithm? It is useful to show the part of the array that is already sorted in a different color. Also, we want to show how each step of the algorithm inspects another element in the unsorted part. That demonstrates why the selection sort algorithm is so slow—it first inspects all elements of the array, then all but one, and so on. If the array has n elements, the algorithm inspects

$$n + (n - 1) + (n - 2) + \cdots = \frac{n(n + 1)}{2}$$

or $O(n^2)$ elements. To demonstrate that, we mark the currently visited element in red.

Thus, the algorithm state is described by three items:

- The array of values
- The size of the already sorted area
- The currently marked element

The algorithm state needs to be safely accessed by the algorithm and painting threads.

We add this state to the `SelectionSorter` class.

```
public class SelectionSorter
{
   // This array is being sorted
   private int[] a;
   // These instance variables are needed for drawing
   private int markedPosition = -1;
   private int alreadySorted = -1;
   . . .
}
```

The array that is being sorted is now an instance variable, and we will change the `sort` method from a static method to an instance method.

This state is accessed by two threads: the thread that sorts the array and the thread that paints the frame. We use a lock to synchronize access to the shared state.

Finally, we add a `component` instance variable to the algorithm class and augment the constructor to set it. That instance variable is needed for repainting the component and finding out the dimensions of the component when drawing the algorithm state.

```
public class SelectionSorter
{
   private JComponent component;
   . . .
   public SelectionSorter(int[] anArray, JComponent aComponent)
   {
      a = anArray;
      sortStateLock = new ReentrantLock();
      component = aComponent;
   }
}
```

At each point of interest, the algorithm needs to pause so that the user can admire the graphical output. We supply the `pause` method shown below, and call it at various places in the algorithm. The `pause` method repaints the component and sleeps for a small delay that is proportional to the number of steps involved.

```
public void pause(int steps) throws InterruptedException
{
   component.repaint();
   Thread.sleep(steps * DELAY);
}
```

We add a `draw` method to the algorithm class that can draw the current state of the data structure, with the items of special interest highlighted. The `draw` method is specific to the particular algorithm. This `draw` method draws the array elements as a sequence of sticks in different colors. The already sorted portion is blue, the marked position is red, and the remainder is black (see Figure 3).

```
public void draw(Graphics g)
{
   sortStateLock.lock();
   try
   {
      int deltaX = component.getWidth() / a.length;
      for (int i = 0; i < a.length; i++)
      {
         if (i == markedPosition)
         {
            g.setColor(Color.RED);
         }
         else if (i <= alreadySorted)
         {
            g.setColor(Color.BLUE);
         }
         else
         {
            g.setColor(Color.BLACK);
         }
         g.drawLine(i * deltaX, 0, i * deltaX, a[i]);
      }
   }
   finally
   {
      sortStateLock.unlock();
   }
}
```

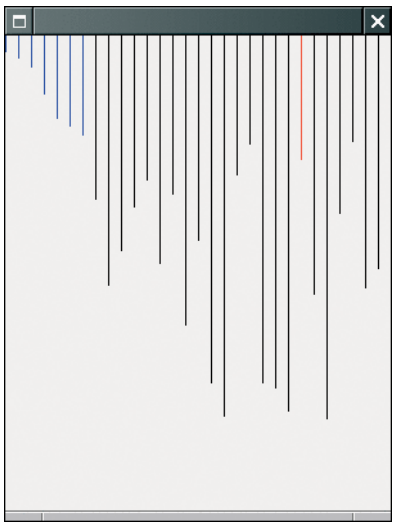


Figure 3 A Step in the Animation of the Selection Sort Algorithm

You need to update the special positions as the algorithm progresses and pause the animation whenever something interesting happens. The pause should be proportional to the number of steps that are being executed. For a sorting algorithm, pause one unit for each visited array element.

Here is the `minimumPosition` method from Chapter 14:

```
public static int minimumPosition(int[] a, int from)
{
   int minPos = from;
   for (int i = from + 1; i < a.length; i++)
   {
      if (a[i] < a[minPos]) { minPos = i; }
   }
   return minPos;
}
```

After each iteration of the `for` loop, update the marked position of the algorithm state; then pause the program. To measure the cost of each step fairly, pause for two units of time, because two array elements were inspected. Because we need to access the marked position and call the `pause` method, we need to change the method to an instance method:

```
private int minimumPosition(int from)
      throws InterruptedException
{
   int minPos = from;
   for (int i = from + 1; i < a.length; i++)
   {
      sortStateLock.lock();
      try
      {
         if (a[i] < a[minPos]) { minPos = i; }
         // For animation
         markedPosition = i;
```

```
            }
            finally
            {
               sortStateLock.unlock();
            }
            pause(2);
         }
         return minPos;
      }
```

The `sort` method is augmented in the same way. You will find the code at the end of this section. This concludes the modification of the algorithm class. Let us now turn to the component class.

The component's `paintComponent` method calls the `draw` method of the algorithm object.

```
public class SelectionSortComponent extends JComponent
{
   private SelectionSorter sorter;
   . . .
   public void paintComponent(Graphics g)
   {
      sorter.draw(g);
   }
}
```

The `SelectionSortComponent` constructor constructs a `SelectionSorter` object, which supplies a new array and the `this` reference to the component that displays the sorted values:

```
public SelectionSortComponent()
{
   int[] values = ArrayUtil.randomIntArray(30, 300);
   sorter = new SelectionSorter(values, this);
}
```

The `startAnimation` method constructs a thread that calls the sorter's `sort` method:

```
public void startAnimation()
{
   class AnimationRunnable implements Runnable
   {
      public void run()
      {
         try
         {
            sorter.sort();
         }
         catch (InterruptedException exception)
         {
         }
      }
   }

   Runnable r = new AnimationRunnable();
   Thread t = new Thread(r);
   t.start();
}
```

The class for the viewer program that displays the animation is at the end of this example. Run the program and the animation starts.

Exercise P22.8 asks you to animate the merge sort algorithm of Chapter 14. If you do that exercise, then start both programs and run them in parallel to see which algorithm is faster. Actually, you may find the result surprising. If you build fair delays into the merge sort animation to account for the copying from and to the temporary array, you will find that it doesn't perform all that well for small arrays. But if you increase the array size, then the advantage of the merge sort algorithm becomes clear.

section_6/SelectionSortViewer.java

```
1  import java.awt.BorderLayout;
2  import javax.swing.JButton;
3  import javax.swing.JFrame;
4
5  public class SelectionSortViewer
6  {
7     public static void main(String[] args)
8     {
9        JFrame frame = new JFrame();
10
11       final int FRAME_WIDTH = 300;
12       final int FRAME_HEIGHT = 400;
13
14       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
15       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16
17       final SelectionSortComponent component
18             = new SelectionSortComponent();
19       frame.add(component, BorderLayout.CENTER);
20
21       frame.setVisible(true);
22       component.startAnimation();
23    }
24 }
```

section_6/SelectionSortComponent.java

```
1  import java.awt.Graphics;
2  import javax.swing.JComponent;
3
4  /**
5     A component that displays the current state of the selection sort algorithm.
6  */
7  public class SelectionSortComponent extends JComponent
8  {
9     private SelectionSorter sorter;
10
11    /**
12       Constructs the component.
13    */
14    public SelectionSortComponent()
15    {
16       int[] values = ArrayUtil.randomIntArray(30, 300);
17       sorter = new SelectionSorter(values, this);
18    }
19
20    public void paintComponent(Graphics g)
21    {
22       sorter.draw(g);
```

```
23    }
24
25    /**
26       Starts a new animation thread.
27    */
28    public void startAnimation()
29    {
30       class AnimationRunnable implements Runnable
31       {
32          public void run()
33          {
34             try
35             {
36                sorter.sort();
37             }
38             catch (InterruptedException exception)
39             {
40             }
41          }
42       }
43
44       Runnable r = new AnimationRunnable();
45       Thread t = new Thread(r);
46       t.start();
47    }
48 }
```

section_6/SelectionSorter.java

```
1  import java.awt.Color;
2  import java.awt.Graphics;
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import javax.swing.JComponent;
6
7  /**
8     This class sorts an array, using the selection sort algorithm.
9  */
10 public class SelectionSorter
11 {
12    // This array is being sorted
13    private int[] a;
14    // These instance variables are needed for drawing
15    private int markedPosition = -1;
16    private int alreadySorted = -1;
17
18    private Lock sortStateLock;
19
20    // The component is repainted when the animation is paused
21    private JComponent component;
22
23    private static final int DELAY = 100;
24
25    /**
26       Constructs a selection sorter.
27       @param anArray the array to sort
28       @param aComponent the component to be repainted when the animation
29       pauses
30    */
```

```
31    public SelectionSorter(int[] anArray, JComponent aComponent)
32    {
33       a = anArray;
34       sortStateLock = new ReentrantLock();
35       component = aComponent;
36    }
37
38    /**
39       Sorts the array managed by this selection sorter.
40    */
41    public void sort()
42          throws InterruptedException
43    {
44       for (int i = 0; i < a.length - 1; i++)
45       {
46          int minPos = minimumPosition(i);
47          sortStateLock.lock();
48          try
49          {
50             ArrayUtil.swap(a, minPos, i);
51             // For animation
52             alreadySorted = i;
53          }
54          finally
55          {
56             sortStateLock.unlock();
57          }
58          pause(2);
59       }
60    }
61
62    /**
63       Finds the smallest element in a tail range of the array.
64       @param from the first position in a to compare
65       @return the position of the smallest element in the
66       range a[from] ... a[a.length - 1]
67    */
68    private int minimumPosition(int from)
69          throws InterruptedException
70    {
71       int minPos = from;
72       for (int i = from + 1; i < a.length; i++)
73       {
74          sortStateLock.lock();
75          try
76          {
77             if (a[i] < a[minPos]) { minPos = i; }
78             // For animation
79             markedPosition = i;
80          }
81          finally
82          {
83             sortStateLock.unlock();
84          }
85          pause(2);
86       }
87       return minPos;
88    }
89
```

```
90     /**
91        Draws the current state of the sorting algorithm.
92        @param g the graphics context
93     */
94     public void draw(Graphics g)
95     {
96        sortStateLock.lock();
97        try
98        {
99           int deltaX = component.getWidth() / a.length;
100          for (int i = 0; i < a.length; i++)
101          {
102             if (i == markedPosition)
103             {
104                g.setColor(Color.RED);
105             }
106             else if (i <= alreadySorted)
107             {
108                g.setColor(Color.BLUE);
109             }
110             else
111             {
112                g.setColor(Color.BLACK);
113             }
114             g.drawLine(i * deltaX, 0, i * deltaX, a[i]);
115          }
116       }
117       finally
118       {
119          sortStateLock.unlock();
120       }
121    }
122
123    /**
124       Pauses the animation.
125       @param steps the number of steps to pause
126    */
127    public void pause(int steps)
128          throws InterruptedException
129    {
130       component.repaint();
131       Thread.sleep(steps * DELAY);
132    }
133 }
```

11. Why is the `draw` method added to the `SelectionSorter` class and not the `SelectionSortComponent` class?

12. Would the animation still work if the `startAnimation` method simply called `sorter.sort()` instead of spawning a thread that calls that method?

Practice It Now you can try these exercises at the end of the chapter: R22.14, P22.5, P22.7.

CHAPTER SUMMARY

Describe how multiple threads execute concurrently.

- A thread is a program unit that is executed concurrently with other parts of the program.
- The `start` method of the `Thread` class starts a new thread that executes the `run` method of the associated `Runnable` object.
- The `sleep` method puts the current thread to sleep for a given number of milliseconds.
- When a thread is interrupted, the most common response is to terminate the `run` method.
- The thread scheduler runs each thread for a short amount of time, called a time slice.

Choose appropriate mechanisms for terminating threads.

- A thread terminates when its `run` method terminates.
- The `run` method can check whether its thread has been interrupted by calling the `interrupted` method.

Recognize the causes and effects of race conditions.

- A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

Use locks to control access to resources that are shared by multiple threads.

- By calling the `lock` method, a thread acquires a `Lock` object. Then no other thread can acquire the lock until the first thread releases the lock.

Explain how deadlocks occur and how they can be avoided with condition objects.

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.
- Calling `await` on a condition object makes the current thread wait and allows another thread to acquire the lock object.
- A waiting thread is blocked until another thread calls `signalAll` or `signal` on the condition object for which the thread is waiting.

Use multiple threads to display an animation of an algorithm.

- Use a separate thread for running the algorithm that is being animated.
- The algorithm state needs to be safely accessed by the algorithm and painting threads.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.lang.InterruptedException
java.lang.Object
   notify
   notifyAll
   wait
java.lang.Runnable
   run
java.lang.Thread
   interrupted
   sleep
   start
java.util.Date
java.util.concurrent.locks.Condition
   await
   signal
   signalAll
java.util.concurrent.locks.Lock
   lock
   newCondition
   unlock
java.util.concurrent.locks.ReentrantLock
```

REVIEW EXERCISES

■ **R22.1** Run a program with the following instructions:

```
GreetingRunnable r1 = new GreetingRunnable("Hello");
GreetingRunnable r2 = new GreetingRunnable("Goodbye");
r1.run();
r2.run();
```

Note that the threads don't run in parallel. Explain.

■■■ **R22.2** In the program of Section 22.1, is it possible that both threads are sleeping at the same time? Is it possible that neither of the two threads is sleeping at a particular time? Explain.

■■■ **R22.3** In Java, a program with a graphical user interface has more than one thread. Explain how you can prove that.

■■■ **R22.4** Why is the `stop` method for stopping a thread deprecated? How do you terminate a thread?

■ **R22.5** Give an example of why you would want to terminate a thread.

■■ **R22.6** Suppose you surround each call to the `sleep` method with a `try/catch` block to catch an `InterruptedException` and ignore it. What problem do you create?

■■ **R22.7** What is a race condition? How can you avoid it?

■■ **R22.8** Consider the `ArrayList` implementation from Section 16.2. Describe two different scenarios in which race conditions can corrupt the data structure.

■■ **R22.9** Consider a stack that is implemented as a linked list, as in Section 16.3.1. Describe two different scenarios in which race conditions can corrupt the data structure.

■■ **R22.10** Consider a queue that is implemented as a circular array, as in Section 16.3.4. Describe two different scenarios in which race conditions can corrupt the data structure.

■■ **R22.11** What is a deadlock? How can you avoid it?

■ **R22.12** What is the difference between a thread that sleeps by calling `sleep` and a thread that waits by calling `await`?

■ **R22.13** What happens when a thread calls `await` and no other thread calls `signalAll` or `signal`?

■■ **R22.14** In the algorithm animation program of Section 22.6, we do not use any conditions. Why not?

PRACTICE EXERCISES

■■ **E22.1** Write a program in which multiple threads add and remove elements from a `java.util.LinkedList`. Demonstrate that the list is being corrupted.

■■ **E22.2** Implement a stack as a linked list in which the `push`, `pop`, and `isEmpty` methods can be safely accessed from multiple threads.

■■■ **E22.3** Implement a `Queue` class whose `add` and `remove` methods are synchronized. Supply one thread, called the producer, which keeps inserting strings into the queue as long as there are fewer than ten elements in it. When the queue gets too full, the thread waits. As sample strings, simply use time stamps `new Date().toString()`. Supply a second thread, called the consumer, that keeps removing and printing strings from the queue as long as the queue is not empty. When the queue is empty, the thread waits. Both the consumer and producer threads should run for 100 iterations.

■ **E22.4** Enhance the program of Exercise E22.3 by supplying a variable number of producer and consumer threads. Prompt the program user for the numbers.

■ **E22.5** Reimplement Exercise E22.4 by using the `ArrayBlockingQueue` class from the standard library.

■■ **E22.6** Modify the `ArrayList` implementation of Section 16.2 so that all methods can be safely accessed from multiple threads.

■■ **E22.7** Write a program `WordCount` that counts the words in one or more files. Start a new thread for each file. For example, if you call

```
java WordCount report.txt address.txt Homework.java
```

then the program might print

```
address.txt: 1052
Homework.java: 445
report.txt: 2099
```

■■■ **E22.8** Enhance the program of Exercise E22.7 so that the last active thread also prints a combined count. Use locks to protect the combined word count and a counter of active threads.

■■ **E22.9** Add a condition to the `deposit` method of the `BankAccount` class in Section 22.5, restricting deposits to $100,000 (the insurance limit of the U.S. government). The method should block until sufficient money has been withdrawn by another thread. Test your program with a large number of deposit threads.

PROGRAMMING PROJECTS

■■ **P22.1** Write a program `Find` that searches all files specified on the command line and prints out all lines containing a reserved word. Start a new thread for each file. For example, if you call

```
java Find Buff report.txt address.txt Homework.java
```

then the program might print

```
report.txt: Buffet style lunch will be available at the
address.txt: Buffet, Warren|11801 Trenton Court|Dallas|TX
Homework.java: BufferedReader in;
address.txt: Walters, Winnie|59 Timothy Circle|Buffalo|MI
```

■■■ **P22.2** Implement the merge sort algorithm of Chapter 14 by spawning a new thread for each smaller `MergeSorter`. *Hint:* Use the `join` method of the `Thread` class to wait for the spawned threads to finish. Look up the method's behavior in the API documentation.

■■ **Graphics P22.3** Write a program that shows two cars moving across a window. Use a separate thread for each car.

■■■ **Graphics P22.4** Modify Exercise P22.3 so that the cars change direction when they hit an edge of the window.

■ **Graphics P22.5** Enhance the `SelectionSorter` of Section 22.6 so that the current minimum is painted in yellow.

■■ **Graphics P22.6** Enhance the `SelectionSortViewer` of Section 22.6 so that the sorting only starts when the user clicks a "Start" button.

■■ **Graphics P22.7** Instead of using a thread and a `pause` method, use the `Timer` class introduced in Chapter 10 to animate an algorithm. Whenever the timer sends out an action event, run the algorithm to the next step and display the state. That requires a more extensive recoding of the algorithm. You need to implement a `runToNextStep` method that is capable of running the algorithm one step at a time. Add sufficient instance variables to the algorithm to remember where the last step left off. For example, in the case of the selection sort algorithm, if you know the values of `alreadySorted` and `markedPosition`, you can determine the next step.

■■■ **Graphics P22.8** Implement an animation of the merge sort algorithm of Chapter 14. Reimplement the algorithm so that the recursive calls sort the elements inside a subrange of the original array, rather than in their own arrays:

```
public void mergeSort(int from, int to)
{
   if (from == to) { return; }
   int mid = (from + to) / 2;
   mergeSort(from, mid);
   mergeSort(mid + 1, to);
   merge(from, mid, to);
}
```

The `merge` method merges the sorted ranges `a[from] ... a[mid]` and `a[mid + 1] ... a[to]`. Merge the ranges into a temporary array, then copy back the temporary array into the combined range.

Pause in the `merge` method whenever you inspect an array element. Color the range `a[from] ... a[to]` in blue and the currently inspected element in red.

■■■ **Graphics P22.9** Enhance Exercise P22.8 so that it shows two frames, one for a merge sorter and one for a selection sorter. They should both sort arrays with the same values.

■■■ **Graphics P22.10** Reorganize the code of the sorting animation in Section 22.6 so that it can be used for generic animations. Provide a class `Animated` with abstract methods

```
public void run()
public void draw(Graphics g, int width, int height)
```

and concrete methods

```
public void lock()
public void unlock(int steps)
public void setComponent(JComponent component)
```

so that the `SelectionSorter` can be implemented as

```
public class SelectionSorter extends Animated
{
   private int[] a;
   private int markedPosition = -1;
   private int alreadySorted = -1;

   public SelectionSorter(int[] anArray) { a = anArray; }

   public void run()
   {
      for (int i = 0; i < a.length - 1; i++)
      {
         int minPos = minimumPosition(i);
         lock();
         ArrayUtil.swap(a, minPos, i);
         alreadySorted = i;
         unlock(2);
      }
   }

   private int minimumPosition(int from)
   {
      int minPos = from;
      for (int i = from + 1; i < a.length; i++)
      {
         lock();
         if (a[i] < a[minPos]) { minPos = i; }
         markedPosition = i;
         unlock(2);
      }
      return minPos;
   }

   public void draw(Graphics g, int width, int height)
   {
      int deltaX = width / a.length;
      for (int i = 0; i < a.length; i++)
      {
         if (i == markedPosition) { g.setColor(Color.RED); }
         else if (i <= alreadySorted) { g.setColor(Color.BLUE); }
         else { g.setColor(Color.BLACK); }
         g.drawLine(i * deltaX, 0, i * deltaX, a[i]);
```

```
            }
         }
      }
```

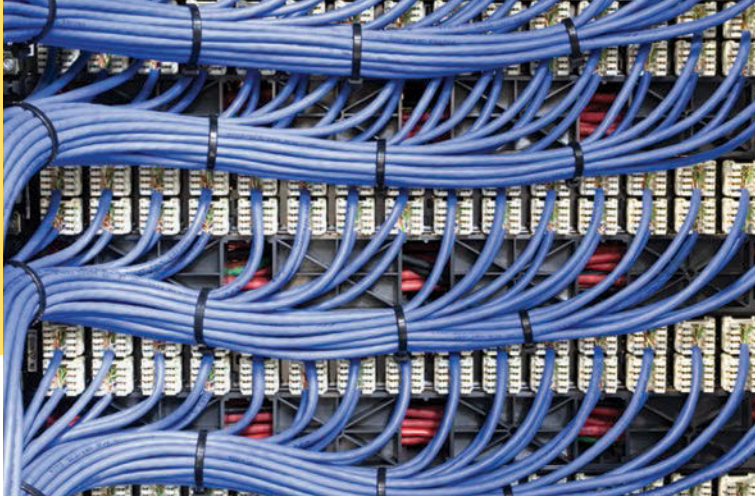
The remaining classes should be independent of any particular animation.

ANSWERS TO SELF-CHECK QUESTIONS

1. The messages are printed about one millisecond apart.
2. The first call to `run` would print ten "Hello" messages, and then the second call to `run` would print ten "Goodbye" messages.
3. If the user hits the "Back" button, the current web page is no longer displayed, and it makes no sense to expend network resources to fetch additional image data.
4. The `run` method prints the values 1, 3, and 4. The call to `interrupt` merely sets the interruption flag, but the `sleep` method immediately throws an `InterruptedException`.
5. There are many possible scenarios. Here is one:
   a. The first thread loses control after the first `print` statement.
   b. The second thread loses control just before the assignment `balance = newBalance`.
   c. The first thread completes the `deposit` method.
   d. The second thread completes the `withdraw` method.
6. One thread calls `addFirst` and is preempted just before executing the assignment `first = newNode`. Then the next thread calls `addFirst`, using the old value of `first`. Then the first thread completes the process, setting `first` to its new node. As a result, the links are not in sequence.
7. Two, one for each bank account object. Each lock protects a separate `balance` variable.
8. When a thread calls `deposit`, it continues to own the lock, and any other thread trying to deposit or withdraw money in the same bank account is blocked forever.
9. A sleeping thread is reactivated when the sleep delay has passed. A waiting thread is only reactivated if another thread has called `signalAll` or `signal`.
10. The calls to `await` and `signal`/`signalAll` must be made *to the same object*.
11. The `draw` method uses the array values and the values that keep track of the algorithm's progress. These values are available only in the `SelectionSorter` class.
12. Yes, provided you only show a single frame. If you modify the `SelectionSortViewer` program to show two frames, you want the sorters to run in parallel.

CHAPTER 23

INTERNET NETWORKING



© Felix Alim/iStockphoto.

CHAPTER GOALS

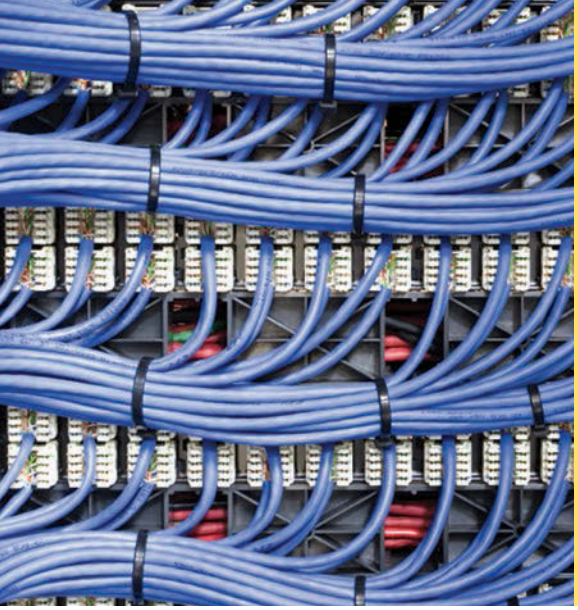
To understand the concept of sockets

To send and receive data through sockets

To implement network clients and servers

To communicate with web servers and server-side applications through the Hypertext Transfer Protocol (HTTP)

CHAPTER CONTENTS



© Felix Alim/iStockphoto.

You probably have quite a bit of experience with the Internet, the global network that links together millions of computers. In particular, you use the Internet whenever you browse the World Wide Web. Note that the Internet is not the same as the "Web". The World Wide Web is only one of many services offered over the Internet. E-mail, another popular service, also uses the Internet, but its implementation differs from that of the Web. In this chapter, you will see what goes on "under the hood" when you send an e-mail message or when you retrieve a web page from a remote server. You will also learn how to write programs that fetch data from sites across the Internet and how to write server programs that can serve information to other programs.

23.1 The Internet Protocol

The *Internet* is a worldwide collection of networks, routing equipment, and computers using a common set of protocols to define how each party will interact with each other.

Computers can be connected with each other through a variety of physical media. In a computer lab, for example, computers are connected by network cabling. Electrical impulses representing information flow across the cables. If you use a DSL modem to connect your computer to the Internet, the signals travel across a regular telephone wire, encoded as tones. On a wireless network, signals are sent by transmitting a modulated radio frequency. The physical characteristics of these transmissions differ widely, but they ultimately consist of sending and receiving streams of zeroes and ones along the network connection.

These zeroes and ones represent two kinds of information: *application data,* the data that one computer actually wants to send to another, and *network protocol data,* the data that describe how to reach the intended recipient and how to check for errors and data loss in the transmission. The protocol data follow certain rules set forth by the Internet Protocol Suite, also called TCP/IP, after the two most important protocols in the suite. These protocols have become the basis for connecting computers around the world over the Internet. We will discuss TCP and IP in this chapter.

Suppose that a computer A wants to send data to a computer B, both on the Internet. The computers aren't connected directly with a cable, as they could be if both were on the same local area network. Instead, A may be someone's home computer and connected to an *Internet service provider (ISP),* which is in turn connected to an *Internet access point;* B might be a computer on a local area network belonging to a large firm that has an Internet access point of its own, which may be half a world away from A. The **Internet** itself, finally, is a complex collection of pathways on which a message can travel from one Internet access point to, eventually, any other Internet access point (see Figure 1). Those connections carry millions of messages, not just the data that A is sending to B.

For the data to arrive at its destination, it must be marked with a *destination address.* In IP, addresses are denoted by sequences of four numbers, each one byte (that is, between 0 and 255); for example, 130.65.86.66. (Because there aren't enough four-byte addresses for all devices that would like to connect to the Internet, these addresses have been extended to sixteen bytes. For simplicity, we use the classic four-byte addresses in this chapter.) In order to send data, A needs to know the Internet

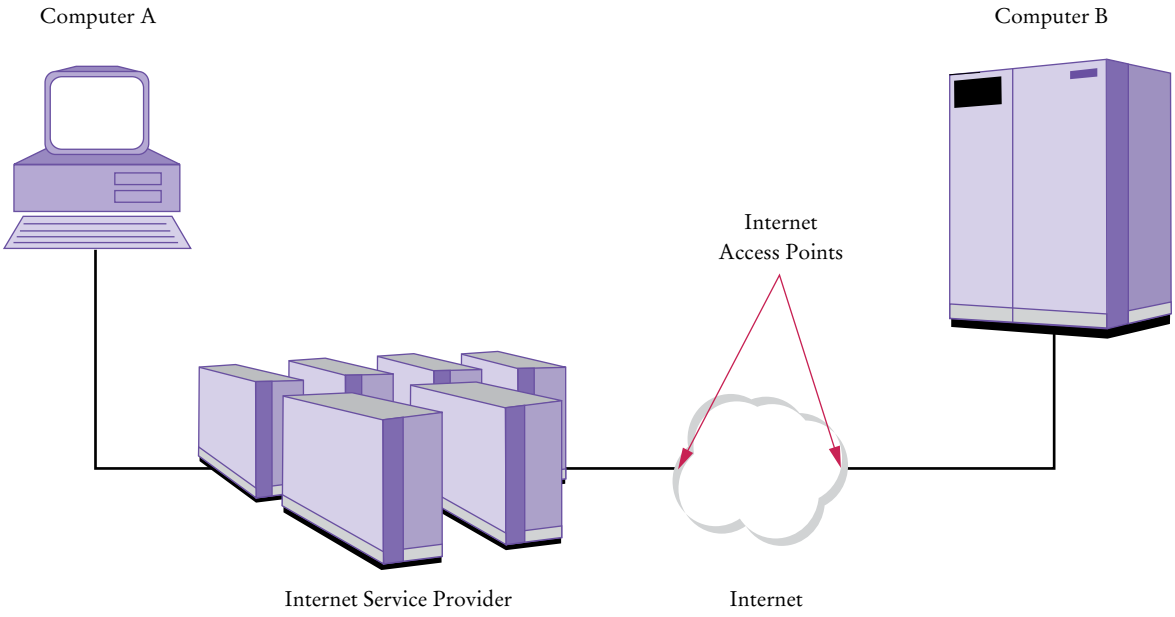


Figure 1 Two Computers Communicating Across the Internet

address of B and include it in the protocol portion when sending the data across the Internet. The routing software that is distributed across the Internet can then deliver the data to B.

Of course, addresses such as 130.65.86.66 are not easy to remember. You would not be happy if you had to use number sequences every time you sent e-mail or requested information from a web server. On the Internet, computers can have so-called *domain names* that are easier to remember, such as `cs.sjsu.edu` or `horstmann.com`. A special service called the *Domain Name System (DNS)* translates between domain names and Internet addresses. Thus, if computer A wants to have information from `horstmann.com`, it first asks the DNS to translate this domain name into a numeric Internet address; then it includes the numeric address with the request.

One interesting aspect of IP is that it breaks large chunks of data up into more manageable *packets*. Each packet is delivered separately, and different packets that are part of the same transmission can take different routes through the Internet. Packets are numbered, and the recipient reassembles them in the correct order.

TCP/IP is the abbreviation for *Transmission Control Protocol and Internet Protocol*, the pair of communication protocols designed to establish reliable transmission of data between two computers on the Internet.

The Internet Protocol is used when attempting to deliver data from one computer to another across the Internet. If some data get lost or garbled in the process, IP has safeguards built in to make sure that the recipient is aware of that unfortunate fact and doesn't rely on incomplete data. However, IP has no provision for retrying an incomplete transmission. That is the job of a higher-level protocol, the *Transmission Control Protocol (TCP)*. This protocol attempts reliable delivery of data, with retries if there are failures, and it notifies the sender whether or not the attempt succeeded. Most, but not all, Internet programs use TCP for reliable delivery. (Exceptions are "streaming media" services, which bypass the slower TCP for the highest possible throughput and tolerate occasional information loss. However, the most popular Internet services—the World Wide Web and e-mail—use TCP.) TCP is independent of the Internet Protocol; it could in principle be used with another lower-level

network protocol. However, in practice, TCP over IP (often called TCP/IP) is the most commonly used combination. We will focus on TCP/IP networking in this chapter.

A TCP connection requires the Internet addresses and port numbers of both end points.

A computer that is connected to the Internet may have programs for many different purposes. For example, a computer may run both a web server program and a mail server program. When data are sent to that computer, they need to be marked so that they can be forwarded to the appropriate program. TCP uses *port numbers* for this purpose. A port number is an integer between 0 and 65,535. The sending computer must know the port number of the receiving program and include it with the transmitted data. Some applications use "well-known" port numbers. For example, by convention, web servers use port 80, whereas mail servers running the Post Office Protocol (POP) use port 110. A TCP connection, therefore, requires

- The Internet address of the recipient.
- The port number of the recipient.
- The Internet address of the sender.
- The port number of the sender.

You can think of a TCP connection as a "pipe" between two computers that links the two ports together. Data flow in either direction through the pipe. In practical programming situations, you simply establish a connection and send data across it without worrying about the details of the TCP/IP mechanism. You will see how to establish such a connection in Section 23.3.

1. What is the difference between an IP address and a domain name?
2. Why do some streaming media services not use TCP?

Practice It Now you can try these exercises at the end of the chapter: R23.1, R23.2, R23.3.

23.2 Application Level Protocols

HTTP, or *Hypertext Transfer Protocol*, is the protocol that defines communication between web browsers and web servers.

A URL, or *Uniform Resource Locator*, is a pointer to an information resource (such as a web page or an image) on the World Wide Web.

In the preceding section you saw how the TCP/IP mechanism can establish an Internet connection between two ports on two computers so that the two computers can exchange data. Each Internet application has a different *application protocol*, which describes how the data for that particular application are transmitted.

Consider, for example, HTTP: the **Hypertext Transfer Protocol**, which is used for the World Wide Web. Suppose you type a web address, called a **Uniform Resource Locator** (URL), such as `http://horstmann.com/index.html`, into the address window of your browser and ask the browser to load the page.

The browser now takes the following steps:

1. It examines the part of the URL between the double slash and the first single slash ("`horstmann.com`"), which identifies the computer to which you want to connect. Because this part of the URL contains letters, it must be a domain name rather than an Internet address, so the browser sends a request to a DNS

server to obtain the Internet address of the computer with domain name `horstmann.com`.

2. From the `http:` prefix of the URL, the browser deduces that the protocol you want to use is HTTP, which by default uses port 80.
3. It establishes a TCP/IP connection to port 80 at the Internet address it obtained in Step 1.
4. It deduces from the `/index.html` suffix that you want to see the file `/index.html`, so it sends a request, formatted as an HTTP command, through the connection that was established in Step 3. The request looks like this:

   ```
   GET /index.html HTTP/1.1
   Host: horstmann.com
   blank line
   ```

   (The host is needed because a web server can host multiple domains with the same Internet address.)
5. The web server running on the computer whose Internet address is the one the browser obtained in Step 1 receives the request and decodes it. It then fetches the file `/index.html` and sends it back to the browser on your computer.
6. The browser displays the contents of the file. Because it happens to be an HTML file, the browser translates the HTML tags into fonts, bullets, separator lines, and so on. If the HTML file contains images, then the browser makes more `GET` requests, one for each image, through the same connection, to fetch the image data. (Appendix J contains a summary of the most frequently used HTML tags.)

The Telnet program is a useful tool for establishing test connections with servers.

You can try the following experiment to see this process in action. The "Telnet" program enables a user to type characters for sending to a remote computer and view characters that the remote computer sends back. On Windows, you need to enable the Telnet program in the control panel. UNIX, Linux, and Mac OS X systems normally have Telnet preinstalled.

For this experiment, you want to start Telnet with a host of `horstmann.com` and port 80. To start the program from the command line, simply type

```
telnet horstmann.com 80
```

Table 1 HTTP Commands

Command	Meaning
`GET`	Return the requested item
`HEAD`	Request only the header information of an item
`OPTIONS`	Request communications options of an item
`POST`	Supply input to a server-side command and return the result
`PUT`	Store an item on the server
`DELETE`	Delete an item on the server
`TRACE`	Trace server communication

Once the program starts, type very carefully, without making any typing errors and without pressing the backspace key,

```
GET / HTTP/1.1
Host: horstmann.com
```

Then press the Enter key twice.

The first / denotes the root page of the web server. Note that there are spaces before and after the first /, but there are no spaces in `HTTP/1.1`.

On Windows, you will not see what you type, so you should be extra careful when typing in the commands.

The server now sends a response to the request—see Figure 2. The response, of course, consists of the root web page that you requested. The Telnet program is not a browser and does not understand HTML tags, so it simply displays the HTML file—text, tags, and all.

The HTTP `GET` command requests information from a web server. The web server returns the requested item, which may be a web page, an image, or other data.

The `GET` command is one of the commands of HTTP. Table 1 shows the other commands of the protocol. As you can see, the protocol is pretty simple.

By the way, be sure not to confuse HTML with HTTP. **HTML** is a *document format* (with commands such as `<h1>` or `<ul>`) that describes the structure of a document, including headings, bulleted lists, images, hyperlinks, and so on. **HTTP** is a *protocol* (with commands such as `GET` and `POST`) that describes the command set for web server requests. Web *browsers* know how to display HTML documents and how to issue HTTP commands. Web *servers* know nothing about HTML. They merely understand HTTP and know how to fetch the requested items. Those items may be HTML documents, GIF or JPEG images, or any other data that a web browser can display.

HTTP is just one of many application protocols in use on the Internet. Another commonly used protocol is the Post Office Protocol (POP), which is used to download received messages from e-mail servers. To *send* messages, you use yet another protocol called the Simple Mail Transfer Protocol (SMTP). We don't want to go into

Figure 2 Using Telnet to Connect to a Web Server

```
+OK San Quentin State POP server
USER harryh
+OK Password required for harryh
PASS secret
+OK harryh has 2 messages (320 octets)
STAT
+OK 2 320
RETR 1
+OK 120 octets
the message is included here
DELE 1
+OK message 1 deleted
QUIT
+OK POP server signing off
```

Black = mail client requests
Color = mail server responses

Figure 3 A Sample POP Session

the details of these protocols, but Figure 3 gives you a flavor of the commands used by the Post Office Protocol.

Both HTTP and POP use plain text, which makes it particularly easy to test and debug client and server programs (see How To 23.1).

3. Why don't you need to know about HTTP when you use a web browser?
4. Why is it important that you don't make typing errors when you type HTTP commands in Telnet?

Practice It Now you can try these exercises at the end of the chapter: R23.13, R23.14, R23.15.

23.3 A Client Program

In this section you will see how to write a Java program that establishes a TCP connection to a server, sends a request to the server, and prints the response.

A socket is an object that encapsulates a TCP connection. To communicate with the other end point of the connection, use the input and output streams attached to the socket.

In the terminology of TCP/IP, there is a **socket** on each side of the connection (see Figure 4). In Java, a client establishes a socket with a call

```
Socket s = new Socket(hostname, portnumber);
```

For example, to connect to the HTTP port of the server `horstmann.com`, you use

```
final int HTTP_PORT = 80;
Socket s = new Socket("horstmann.com", HTTP_PORT);
```

The socket constructor throws an `UnknownHostException` if it can't find the host.

Once you have a socket, you obtain its input and output streams:

```
InputStream instream = s.getInputStream();
OutputStream outstream = s.getOutputStream();
```

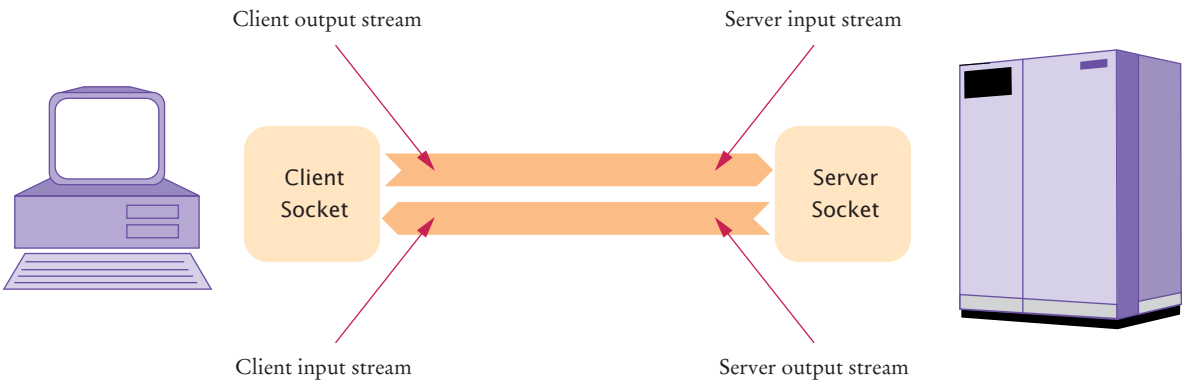


Figure 4 Client and Server Sockets

When you send data to `outstream`, the socket automatically forwards it to the server. The socket catches the server's response, and you can read the response through `instream` (see Figure 4).

When transmission over a socket is complete, remember to close the socket.

When you are done communicating with the server, you should close the socket. This is best done with a `try`-with-resources statement:

```
try (Socket s = . . . )
{
   . . .
} // s.close() called here
```

In Chapter 21, you saw that the `InputStream` and `OutputStream` classes are used for reading and writing bytes. If you want to communicate with the server by sending and receiving text, you should turn the streams into scanners and writers, as follows:

For text protocols, turn the socket streams into scanners and writers.

```
Scanner in = new Scanner(instream);
PrintWriter out = new PrintWriter(outstream);
```

A print writer *buffers* the characters that you send to it. That is, characters are not immediately sent to their destination. Instead, they are placed into an array. When the array is full, then the print writer sends all characters in the array to its destination. The advantage of buffering is increased performance—it takes some amount of time to contact the destination and send it data, and it is expensive to pay for that contact time for every character. However, when communicating with a server that responds to requests, you want to make sure that the server gets a complete request. Therefore, you need to *flush* the buffer manually whenever you send a command:

```
out.print(command);
out.flush();
```

Flush the writer attached to a socket at the end of every command. Then the command is sent to the server, even if the writer's buffer is not completely filled.

The `flush` method empties the buffer and forwards all waiting characters to the destination.

The `WebGet` program at the end of this section lets you retrieve any item from a web server. You need to specify the host and the item from the command line. For example,

```
java WebGet horstmann.com /
```

The `/` item denotes the root page of the web server that listens to port 80 of the host `horstmann.com`. Note that there is a space before the `/`.

The `WebGet` program establishes a connection to the host, sends a `GET` command to the host, and then receives input from the server until the server closes its connection.

section_3/WebGet.java

```
1  import java.io.InputStream;
2  import java.io.IOException;
3  import java.io.OutputStream;
4  import java.io.PrintWriter;
5  import java.net.Socket;
6  import java.util.Scanner;
7
8  /**
9     This program demonstrates how to use a socket to communicate
10    with a web server. Supply the name of the host and the
11    resource on the command line, for example,
12    java WebGet horstmann.com index.html.
13 */
14 public class WebGet
15 {
16    public static void main(String[] args) throws IOException
17    {
18       // Get command-line arguments
19
20       String host;
21       String resource;
22
23       if (args.length == 2)
24       {
25          host = args[0];
26          resource = args[1];
27       }
28       else
29       {
30          System.out.println("Getting / from horstmann.com");
31          host = "horstmann.com";
32          resource = "/";
33       }
34
35       // Open socket
36
37       final int HTTP_PORT = 80;
38       try (Socket s = new Socket(host, HTTP_PORT))
39       {
40          // Get streams
41
42          InputStream instream = s.getInputStream();
43          OutputStream outstream = s.getOutputStream();
44
45          // Turn streams into scanners and writers
46
47          Scanner in = new Scanner(instream);
48          PrintWriter out = new PrintWriter(outstream);
49
50          // Send command
51
52          String command = "GET " + resource + " HTTP/1.1\n"
53             + "Host: " + host + "\n\n";
54          out.print(command);
55          out.flush();
56
57          // Read server response
58
```

```
59          while (in.hasNextLine())
60          {
61             String input = in.nextLine();
62             System.out.println(input);
63          }
64       }
65
66       // The try-with-resources statement closes the socket
67    }
68 }
```

Program Run

```
Getting / from horstmann.com
HTTP/1.1 200 OK
Date: Thu, 09 Apr 2015 14:15:04 GMT
Server: Apache/1.3.41 (Unix) Sun-ONE-ASP/4.0.2
. . .
Content-Length: 6654
Content-Type: text/html

<html>
<head><title>Cay Horstmann's Home Page</title></head>
<body>
<h1>Welcome to Cay Horstmann's Home Page</h1>
. . .
</body>
</html>
```

5. What happens if you call `WebGet` with a nonexistent resource, such as `wombat.html` at `horstmann.com`?
6. How do you open a socket to read e-mail from the POP server at `e-mail.sjsu.edu`?

Practice It Now you can try these exercises at the end of the chapter: R23.7, R23.8, E23.1, E23.2.

23.4 A Server Program

Now that you have seen how to write a network client, we will turn to the server side. In this section we will develop a server program that enables clients to manage a set of bank accounts in a bank.

Whenever you develop a server application, you need to specify some application-level protocol that clients can use to interact with the server. For the purpose of this example, we will create a "Simple Bank Access Protocol". Table 2 shows the protocol format. Of course, this is just a toy protocol to show you how to implement a server.

The server program waits for clients to connect to a particular port. We choose port 8888 for this service. This number has not been preassigned to another service, so it is unlikely to be used by another server program. To listen to incoming connec-

tions, you use a *server socket.* To construct a server socket, you need to supply the port number:

```
ServerSocket server = new ServerSocket(8888);
```

The ServerSocket class is used by server applications to listen for client connections.

The `accept` method of the `ServerSocket` class waits for a client connection. When a client connects, then the server program obtains a socket through which it communicates with the client:

```
Socket s = server.accept();
BankService service = new BankService(s, bank);
```

The `BankService` class carries out the service. This class implements the `Runnable` interface, and its `run` method will be executed in each thread that serves a client connection. The `run` method gets a scanner and writer from the socket in the same way as we discussed in the preceding section. Then it executes the following method:

```
public void doService() throws IOException
{
   while (true)
   {
      if (!in.hasNext()) { return; }
      String command = in.next();
      if (command.equals("QUIT")) { return; }
      executeCommand(command);
   }
}
```

The `executeCommand` method processes a single command. If the command is `DEPOSIT`, then it carries out the deposit:

```
int account = in.nextInt();
double amount = in.nextDouble();
bank.deposit(account, amount);
```

The `WITHDRAW` command is handled in the same way. After each command, the account number and new balance are sent to the client:

```
out.println(account + " " + bank.getBalance(account));
```

The `doService` method returns to the `run` method if the client closed the connection or the command equals `"QUIT"`. Then the `run` method closes the socket and exits.

Let us go back to the point where the server socket accepts a connection and constructs the `BankService` object. At this point, we could simply call the `run` method. But then our server program would have a serious limitation: only one client could connect to it at any point in time. To overcome that limitation, server programs spawn a new thread whenever a client connects. Each thread is responsible for serving one client.

Table 2 A Simple Bank Access Protocol

Client Request	Server Response	Description
`BALANCE` *n*	*n* and the balance	Get the balance of account *n*
`DEPOSIT` *n a*	*n* and the new balance	Deposit amount *a* into account *n*
`WITHDRAW` *n a*	*n* and the new balance	Withdraw amount *a* from account *n*
`QUIT`	None	Quit the connection

Our `BankService` class implements the `Runnable` interface. Therefore, the server program `BankServer` simply starts a thread with the following instructions:

```
Thread t = new Thread(service);
t.start();
```

The thread dies when the client quits or disconnects and the `run` method exits. In the meantime, the `BankServer` loops back to accept the next connection.

```
while (true)
{
   try (Socket s = server.accept())
   {
      BankService service = new BankService(s, bank);
      Thread t = new Thread(service);
      t.start();
   }
}
```

The server program never stops. When you are done running the server, you need to kill it. For example, if you started the server in a shell window, press Ctrl+C.

To try out the program, run the server. Then use Telnet to connect to `localhost`, port number 8888. Start typing commands. Here is a typical dialog (see Figure 5):

```
DEPOSIT 3 1000
3 1000.0
WITHDRAW 3 500
3 500.0
QUIT
```

Alternatively, you can use a client program that connects to the server. You will find a sample client program at the end of this section.

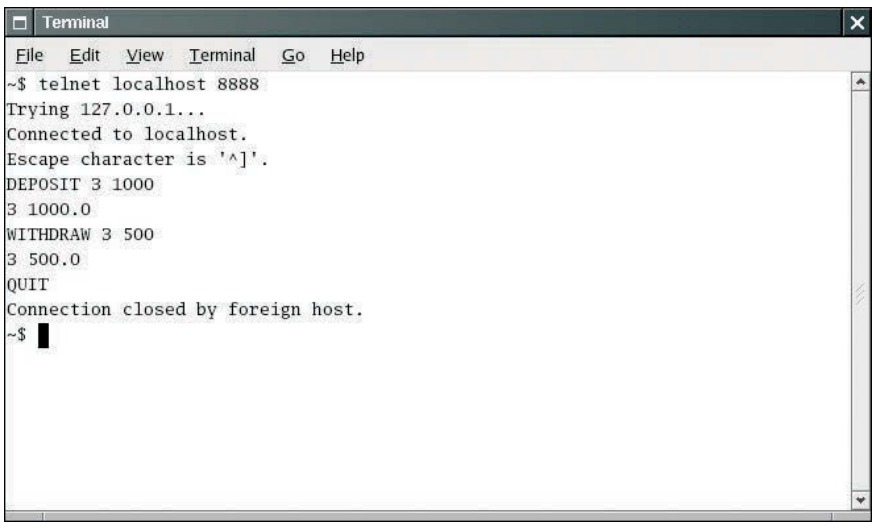


Figure 5 Using the Telnet Program to Connect to the Bank Server

section_4/BankServer.java

```
1  import java.io.IOException;
2  import java.net.ServerSocket;
3  import java.net.Socket;
4
5  /**
6     A server that executes the Simple Bank Access Protocol.
7  */
8  public class BankServer
9  {
10    public static void main(String[] args) throws IOException
11    {
12       final int ACCOUNTS_LENGTH = 10;
13       Bank bank = new Bank(ACCOUNTS_LENGTH);
14       final int SBAP_PORT = 8888;
15       ServerSocket server = new ServerSocket(SBAP_PORT);
16       System.out.println("Waiting for clients to connect...");
17
18       while (true)
19       {
20          try (Socket s = server.accept())
21          {
22             System.out.println("Client connected.");
23             BankService service = new BankService(s, bank);
24             Thread t = new Thread(service);
25             t.start();
26          }
27       }
28    }
29 }
```

section_4/BankService.java

```
1  import java.io.InputStream;
2  import java.io.IOException;
3  import java.io.OutputStream;
4  import java.io.PrintWriter;
5  import java.net.Socket;
6  import java.util.Scanner;
7
8  /**
9     Executes Simple Bank Access Protocol commands
10    from a socket.
11 */
12 public class BankService implements Runnable
13 {
14    private Socket s;
15    private Scanner in;
16    private PrintWriter out;
17    private Bank bank;
18
19    /**
20       Constructs a service object that processes commands
21       from a socket for a bank.
22       @param aSocket the socket
23       @param aBank the bank
24    */
25    public BankService(Socket aSocket, Bank aBank)
26    {
```

```
27          s = aSocket;
28          bank = aBank;
29       }
30
31       public void run()
32       {
33          try
34          {
35             in = new Scanner(s.getInputStream());
36             out = new PrintWriter(s.getOutputStream());
37             doService();
38          }
39          catch (IOException exception)
40          {
41             exception.printStackTrace();
42          }
43       }
44
45       /**
46          Executes all commands until the QUIT command or the
47          end of input.
48       */
49       public void doService() throws IOException
50       {
51          while (true)
52          {
53             if (!in.hasNext()) { return; }
54             String command = in.next();
55             if (command.equals("QUIT")) { return; }
56             else { executeCommand(command); }
57          }
58       }
59
60       /**
61          Executes a single command.
62          @param command the command to execute
63       */
64       public void executeCommand(String command)
65       {
66          int account = in.nextInt();
67          if (command.equals("DEPOSIT"))
68          {
69             double amount = in.nextDouble();
70             bank.deposit(account, amount);
71          }
72          else if (command.equals("WITHDRAW"))
73          {
74             double amount = in.nextDouble();
75             bank.withdraw(account, amount);
76          }
77          else if (!command.equals("BALANCE"))
78          {
79             out.println("Invalid command");
80             out.flush();
81             return;
82          }
83          out.println(account + " " + bank.getBalance(account));
84          out.flush();
85       }
86    }
```

section_4/Bank.java

```
1  /**
2     A bank consisting of multiple bank accounts.
3  */
4  public class Bank
5  {
6     private BankAccount[] accounts;
7
8     /**
9        Constructs a bank account with a given number of accounts.
10       @param size the number of accounts
11    */
12    public Bank(int size)
13    {
14       accounts = new BankAccount[size];
15       for (int i = 0; i < accounts.length; i++)
16       {
17          accounts[i] = new BankAccount();
18       }
19    }
20
21    /**
22       Deposits money into a bank account.
23       @param accountNumber the account number
24       @param amount the amount to deposit
25    */
26    public void deposit(int accountNumber, double amount)
27    {
28       BankAccount account = accounts[accountNumber];
29       account.deposit(amount);
30    }
31
32    /**
33       Withdraws money from a bank account.
34       @param accountNumber the account number
35       @param amount the amount to withdraw
36    */
37    public void withdraw(int accountNumber, double amount)
38    {
39       BankAccount account = accounts[accountNumber];
40       account.withdraw(amount);
41    }
42
43    /**
44       Gets the balance of a bank account.
45       @param accountNumber the account number
46       @return the account balance
47    */
48    public double getBalance(int accountNumber)
49    {
50       BankAccount account = accounts[accountNumber];
51       return account.getBalance();
52    }
53 }
```

section_4/BankClient.java

```
1  import java.io.InputStream;
2  import java.io.IOException;
```

```
3  import java.io.OutputStream;
4  import java.io.PrintWriter;
5  import java.net.Socket;
6  import java.util.Scanner;
7
8  /**
9     This program tests the bank server.
10 */
11 public class BankClient
12 {
13    public static void main(String[] args) throws IOException
14    {
15       final int SBAP_PORT = 8888;
16       try (Socket s = new Socket("localhost", SBAP_PORT))
17       {
18          InputStream instream = s.getInputStream();
19          OutputStream outstream = s.getOutputStream();
20          Scanner in = new Scanner(instream);
21          PrintWriter out = new PrintWriter(outstream);
22
23          String command = "DEPOSIT 3 1000\n";
24          System.out.print("Sending: " + command);
25          out.print(command);
26          out.flush();
27          String response = in.nextLine();
28          System.out.println("Receiving: " + response);
29
30          command = "WITHDRAW 3 500\n";
31          System.out.print("Sending: " + command);
32          out.print(command);
33          out.flush();
34          response = in.nextLine();
35          System.out.println("Receiving: " + response);
36
37          command = "QUIT\n";
38          System.out.print("Sending: " + command);
39          out.print(command);
40          out.flush();
41       }
42    }
43 }
```

Program Run

```
Sending: DEPOSIT 3 1000
Receiving: 3 1000.0
Sending: WITHDRAW 3 500
Receiving: 3 500.0
Sending: QUIT
```

7. Why didn't we choose port 80 for the bank server?
8. Can you read data from a server socket?

Practice It Now you can try these exercises at the end of the chapter: E23.3, E23.4, P23.2.

HOW TO 23.1 Designing Client/Server Programs

The bank server of this section is a typical example of a client/server program. A web browser/web server is another example. This How To outlines the steps to follow when designing a client/server application.

Step 1 Determine whether it really makes sense to implement a stand-alone server and a matching client.

Many times it makes more sense to build a web application instead. Chapter 26 discusses the construction of web applications in detail. For example, the bank application of this section could easily be turned into a web application, using an HTML form with Withdraw and Deposit buttons. However, programs for chat or peer-to-peer file sharing cannot easily be implemented as web applications.

Step 2 Design a communication protocol.

Figure out exactly what messages the client and server send to each other and what the success and error responses are.

With each request and response, ask yourself how the *end of data* is indicated.

- Do the data fit on a single line? Then the end of the line serves as the data terminator.
- Can the data be terminated by a special line (such as a blank line after the HTTP header or a line containing a period in SMTP)?
- Does the sender of the data close the socket? That's what a web server does at the end of a `GET` request.
- Can the sender indicate how many bytes are contained in the request? Web browsers do that in `POST` requests.

Use text, not binary data, for the communication between client and server. A text-based protocol is easier to debug.

Step 3 Implement the server program.

The server listens for socket connections and accepts them. It starts a new thread for each connection. Supply a class that implements the `Runnable` interface. The `run` method receives commands, interprets them, and sends responses back to the client.

Step 4 Test the server with the Telnet program.

Try out all commands in the communication protocol.

Step 5 Once the server works, write a client program.

The client program interacts with the program user, turns user requests into protocol commands, sends the commands to the server, receives the response, and displays the response for the program user.

23.5 URL Connections

The URLConnection class makes it easy to communicate with a web server without having to issue HTTP commands.

In Section 23.3, you saw how to use sockets to connect to a web server and how to retrieve information from the server by sending HTTP commands. However, because HTTP is such an important protocol, the Java library contains a URLConnection class, which provides convenient support for the HTTP. The URLConnection class takes care of the socket connection, so you don't have to fuss with sockets when you want to retrieve from a web server. As an additional benefit, the URLConnection class can also handle FTP, the *file transfer protocol*.

The URLConnection class makes it very easy to fetch a file from a web server given the file's URL as a string. First, you construct a URL object from the URL in the familiar format, starting with the http or ftp prefix. Then you use the URL object's openConnection method to get the URLConnection object itself:

```
URL u = new URL("http://horstmann.com/index.html");
URLConnection connection = u.openConnection();
```

Then you call the getInputStream method to obtain an input stream:

```
InputStream instream = connection.getInputStream();
```

You can turn the stream into a scanner in the usual way, and read input from the scanner.

The URLConnection and HttpURLConnection classes can give you additional information about HTTP requests and responses.

The URLConnection class can give you additional useful information. To understand those capabilities, we need to have a closer look at HTTP requests and responses. You saw in Section 23.2 that the command for getting an item from the server is

```
GET item HTTP/1.1
Host: hostname
blank line
```

You may have wondered why you need to provide a blank line. This blank line is a part of the general request format. The first line of the request is a command, such as GET or POST. The command is followed by *request properties* (such as Host:). Some commands—in particular, the POST command—send input data to the server. The reason for the blank line is to denote the boundary between the request property section and the input data section.

A typical request property is If-Modified-Since. If you request an item with

```
GET item HTTP/1.1
Host: hostname
If-Modified-Since: date
blank line
```

the server sends the item only if it is newer than the date. Browsers use this feature to speed up redisplay of previously loaded web pages. When a web page is loaded, the browser stores it in a *cache* directory. When the user wants to see the same web page again, the browser asks the server to get a new page only if it has been modified since the date of the cached copy. If it hasn't been, the browser simply redisplays the cached copy and doesn't spend time downloading another identical copy.

The URLConnection class has methods to set request properties. For example, you can set the If-Modified-Since property with the setIfModifiedSince method:

```
connection.setIfModifiedSince(date);
```

You need to set request properties before calling the getInputStream method. The URLConnection class then sends to the web server all the request properties that you set.

Similarly, the response from the server starts with a status line followed by a set of response parameters. The response parameters are terminated by a blank line and followed by the requested data (for example, an HTML page). Here is a typical response:

```
HTTP/1.1 200 OK
Date: Thu, 09 Apr 2015 00:15:48 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Tue, 03 Mar 2015 20:53:38 GMT
Content-Length: 4813
Content-Type: text/html
blank line
requested data
```

Normally, you don't see the response code. However, you may have run across bad links and seen a page that contained a response code `404 Not Found`. (A successful response has status `200 OK`.)

To retrieve the response code, you need to cast the `URLConnection` object to the `HttpURLConnection` subclass. You can retrieve the response code (such as the number 200 in this example, or the code 404 if a page was not found) and response message with the `getResponseCode` and `getResponseMessage` methods:

```
HttpURLConnection httpConnection = (HttpURLConnection) connection;
int code = httpConnection.getResponseCode(); // e.g., 404
String message = httpConnection.getResponseMessage(); // e.g., "Not found"
```

As you can see from the response example, the server sends some information about the requested data, such as the content length and the content type. You can request this information with methods from the `URLConnection` class:

```
int length = connection.getContentLength();
String type = connection.getContentType();
```

You need to call these methods after calling the `getInputStream` method.

To summarize: You don't need to use sockets to communicate with a web server, and you need not master the details of the HTTP protocol. Simply use the `URLConnection` and `HttpURLConnection` classes to obtain data from a web server, to set request properties, or to obtain response information.

The program at the end of this section puts the `URLConnection` class to work. The program fulfills the same purpose as that of Section 23.3—to retrieve a web page from a server—but it works at a higher level of abstraction. There is no longer a need to issue an explicit `GET` command. The `URLConnection` class takes care of that. Similarly, the parsing of the HTTP request and response headers is handled transparently to the programmer. Our sample program takes advantage of that fact. It checks whether the server response code is 200. If not, it exits. You can try that out by testing the program with a bad URL, like `http://horstmann.com/wombat.html`. Then the program prints a server response, such as `404 Not Found`.

This program completes our introduction to Internet programming with Java. You have seen how to use sockets to connect client and server programs. You also saw how to use the higher-level `URLConnection` class to obtain information from web servers.

section_5/URLGet.java

```
1  import java.io.InputStream;
2  import java.io.IOException;
3  import java.io.OutputStream;
4  import java.io.PrintWriter;
```

```
5  import java.net.HttpURLConnection;
6  import java.net.URL;
7  import java.net.URLConnection;
8  import java.util.Scanner;
9
10 /**
11    This program demonstrates how to use a URL connection
12    to communicate with a web server. Supply the URL on
13    the command line, for example
14    java URLGet http://horstmann.com/index.html
15 */
16 public class URLGet
17 {
18    public static void main(String[] args) throws IOException
19    {
20       // Get command-line arguments
21
22       String urlString;
23       if (args.length == 1)
24       {
25          urlString = args[0];
26       }
27       else
28       {
29          urlString = "http://horstmann.com/";
30          System.out.println("Using " + urlString);
31       }
32
33       // Open connection
34
35       URL u = new URL(urlString);
36       URLConnection connection = u.openConnection();
37
38       // Check if response code is HTTP_OK (200)
39
40       HttpURLConnection httpConnection
41             = (HttpURLConnection) connection;
42       int code = httpConnection.getResponseCode();
43       String message = httpConnection.getResponseMessage();
44       System.out.println(code + " " + message);
45       if (code != HttpURLConnection.HTTP_OK)
46       {
47          return;
48       }
49
50       // Read server response
51
52       InputStream instream = connection.getInputStream();
53       Scanner in = new Scanner(instream);
54
55       while (in.hasNextLine())
56       {
57          String input = in.nextLine();
58          System.out.println(input);
59       }
60    }
61 }
```

Program Run

```
Using http://horstmann.com/
200 OK
<html>
<head><title>Cay Horstmann's Home Page</title></head>
<body>
<h1>Welcome to Cay Horstmann's Home Page</h1>
. . .
</body>
</html>
```

9. Why is it better to use a `URLConnection` instead of a socket when reading data from a web server?
10. What happens if you use the `URLGet` program to request an image (such as `http://horstmann.com/cay-tiny.gif`)?

Practice It Now you can try these exercises at the end of the chapter: P23.5, P23.6, P23.7.

Programming Tip 23.1

Use High-Level Libraries

When you communicate with a web server to obtain data, you have two choices. You can make a socket connection and send `GET` and `POST` commands to the server over the socket. Or you can use the `URLConnection` class and have it issue the commands on your behalf.

Similarly, to communicate with a mail server, you can write programs that send SMTP and POP commands, or you can learn how to use the Java mail extensions. (See `http://oracle.com/technetwork/java/javamail/index.html` for more information on the Java Mail API.)

In such a situation, you may be tempted to use the low-level approach and send commands over a socket connection. It seems simpler than learning a complex set of classes. However, that simplicity is often deceptive. Once you go beyond the simplest cases, the low-level approach usually requires hard work. For example, to send binary e-mail attachments, you may need to master complex data encodings. The high-level libraries have all that knowledge built in, so you don't have to reinvent the wheel.

For that reason, you should not actually use sockets to connect to web servers. Always use the `URLConnection` class instead. Why did this book teach you about sockets if you aren't expected to use them? There are two reasons. Some client programs don't communicate with web or mail servers, and you may need to use sockets when a high-level library is not available. And, just as importantly, knowing what the high-level library does under the hood helps you understand it better. For the same reason, you saw in Chapter 16 how to implement linked lists, even though you probably will never program your own lists and will just use the standard `LinkedList` class.

CHAPTER SUMMARY

Describe the IP and TCP protocols.

- The Internet is a worldwide collection of networks, routing equipment, and computers using a common set of protocols to define how each party will interact with each other.
- TCP/IP is the abbreviation for *Transmission Control Protocol and Internet Protocol*, the pair of communication protocols designed to establish reliable transmission of data between two computers on the Internet.
- A TCP connection requires the Internet addresses and port numbers of both end points.

Describe the HTTP protocol.

- HTTP, or *Hypertext Transfer Protocol*, is the protocol that defines communication between web browsers and web servers.
- A URL, or *Uniform Resource Locator,* is a pointer to an information resource (such as a web page or an image) on the World Wide Web.
- The Telnet program is a useful tool for establishing test connections with servers.
- The HTTP `GET` command requests information from a web server. The web server returns the requested item, which may be a web page, an image, or other data.

Implement programs that use network sockets for reading data.

- A socket is an object that encapsulates a TCP connection. To communicate with the other end point of the connection, use the input and output streams attached to the socket.
- When transmission over a socket is complete, remember to close the socket.
- For text protocols, turn the socket streams into scanners and writers.
- Flush the writer attached to a socket at the end of every command. Then the command is sent to the server, even if the writer's buffer is not completely filled.

Implement programs that serve data over a network.

- The `ServerSocket` class is used by server applications to listen for client connections.

Use the `URLConnection` class to read data from a web server.

- The `URLConnection` class makes it easy to communicate with a web server without having to issue HTTP commands.
- The `URLConnection` and `HttpURLConnection` classes can give you additional information about HTTP requests and responses.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.net.HttpURLConnection
   getResponseCode
   getResponseMessage
java.net.ServerSocket
   accept
   close
java.net.Socket
   close
   getInputStream
   getOutputStream
java.net.URL
   openConnection
java.net.URLConnection
   getContentLength
   getContentType
   getInputStream
   setIfModifiedSince
```

REVIEW EXERCISES

- **R23.1** What is the IP address of the computer that you are using at home? Does it have a domain name?
- **R23.2** Can a computer somewhere on the Internet establish a network connection with the computer at your home? If so, what information does the other computer need to establish the connection?
- **R23.3** What is a port number? Can the same computer receive data on two different ports?
- **R23.4** What is a server? What is a client? How many clients can connect to a server at one time?
- **R23.5** What is a socket? What is the difference between a `Socket` object and a `ServerSocket` object?
- **R23.6** Under what circumstances would an `UnknownHostException` be thrown?
- **R23.7** What happens if the `Socket` constructor's second argument is not the same as the port number at which the server waits for connections?
- **R23.8** When a socket is created, which of the following Internet addresses is used?
  - **a.** The address of the computer to which you want to connect
  - **b.** The address of your computer
  - **c.** The address of your ISP
- **R23.9** What is the purpose of the `accept` method of the `ServerSocket` class?
- **R23.10** After a socket establishes a connection, which of the following mechanisms will your client program use to read data from the server computer?
  - **a.** The `Socket` will fill a buffer with bytes.
  - **b.** You will use a `Reader` obtained from the `Socket`.
  - **c.** You will use an `InputStream` obtained from the `Socket`.
- **R23.11** Why is it not common to work directly with the `InputStream` and `OutputStream` objects obtained from a `Socket` object?
- **R23.12** When a client program communicates with a server, it sometimes needs to flush the output stream. Explain why.

■ **R23.13** What is the difference between HTTP and HTML?

■ **R23.14** Try out the `HEAD` command of the HTTP protocol. What command did you use? What response did you get?

■■ **R23.15** Connect to a POP server that hosts your e-mail and retrieve a message. Provide a record of your session (but remove your password). If your mail server doesn't allow access on port 110, access it through SSL encryption (usually on port 995). Get a copy of the `openssl` utility and use the command

```
openssl s_client -connect servername:995
```

■ **R23.16** How can you communicate with a web server without using sockets?

■ **R23.17** What is the difference between a `URL` instance and a `URLConnection` instance?

■ **R23.18** What is a URL? How do you create an object of class `URL`? How do you connect to a URL?

PRACTICE EXERCISES

■ **E23.1** Modify the `WebGet` program to print only the HTTP header of the returned HTML page. The HTTP header is the beginning of the response data. It consists of several lines, such as

```
HTTP/1.1 200 OK
Date: Tue, 14 Apr 2015 16:10:34 GMT
Server: Apache/1.3.19 (Unix)
Cache-Control: max-age=86400
Expires: Wed, 15 Apr 2015 16:10:34 GMT
Connection: close
Content-Type: text/html
```

followed by a blank line.

■ **E23.2** Modify the `WebGet` program to print only the *title* of the returned HTML page. An HTML page has the structure

```
<html><head><title> . . . </title></head><body> . . . </body></html>
```

For example, if you run the program by typing at the command line

```
java WebGet horstmann.com /
```

the output should be the title of the root web page at `horstmann.com`, such as `Cay Horstmann's Home Page`.

■■ **E23.3** Modify the `BankServer` program so that it can be terminated more elegantly. Provide another socket on port 8889 through which an administrator can log in. Support the commands `LOGIN` *password*, `STATUS`, `PASSWORD` *newPassword*, `LOGOUT`, and `SHUTDOWN`. The `STATUS` command should display the total number of clients that have logged in since the server started.

■■ **E23.4** Modify the `BankServer` program to provide complete error checking. For example, the program should check to make sure that there is enough money in the account when withdrawing. Send appropriate error reports back to the client. Enhance the protocol to be similar to HTTP, in which each server response starts with a number

indicating the success or failure condition, followed by a string with response data or an error description.

■■ **E23.5** Write a program to display the protocol, host, port, and file components of a URL. *Hint:* Look at the API documentation of the `URL` class.

PROGRAMMING PROJECTS

■■ **P23.1** Write a client application that executes an infinite loop that

a. Prompts the user for a number.

b. Sends that value to the server.

c. Receives a number from the server.

d. Displays the new number.

Also write a server that executes an infinite loop whose body accepts a client connection, reads a number from the client, computes its square root, and writes the result to the client.

■■ **P23.2** Implement a client-server program in which the client will print the date and time given by the server. Two classes should be implemented: `DateClient` and `DateServer`. The `DateServer` simply prints `new Date().toString()` whenever it accepts a connection and then closes the socket.

■■■ **P23.3** Write a simple web server that recognizes only the `GET` request (without the `Host:` request parameter and blank line). When a client connects to your server and sends a command, such as `GET` *filename* `HTTP/1.1`, then return a header

```
HTTP/1.1 200 OK
```

followed by a blank line and all lines in the file. If the file doesn't exist, return `404 Not Found` instead.

Your server should listen to port 8080. Test your web server by starting up your web browser and loading a page, such as `localhost:8080/c:\cs1\myfile.html`.

■■■ **P23.4** Write a chat server and client program. The chat server accepts connections from clients. Whenever one of the clients sends a chat message, it is displayed for all other clients to see. Use a protocol with three commands: `LOGIN` *name*, `CHAT` *message*, and `LOGOUT`.

■■ **P23.5** A query such as

```
http://aa.usno.navy.mil/cgi-bin/aa_moonphases.pl?year=2011
```

returns a page containing the moon phases in a given year. Write a program that asks the user for a year, month, and day and then prints the phase of the moon on that day.

■■■ **P23.6** A page such as

```
http://www.nws.noaa.gov/view/states.php
```

contains links to pages showing the weather reports for many cities in the fifty states. Write a program that asks the user for a state and city and then prints the weather report.

■■■ **P23.7** A page such as

```
https://www.cia.gov/library/publications/the-world-factbook/geos/
   countrytemplate_ca.html
```

contains information about a country (here Canada, with the symbol ca—see `https://www.cia.gov/library/publications/the-world-factbook/print/textversion.html` for the country symbols). Write a program that asks the user for a country name and then prints the area and population.

ANSWERS TO SELF-CHECK QUESTIONS

1. An IP address is a numerical address, consisting of four or sixteen bytes. A domain name is an alphanumeric string that is associated with an IP address.
2. TCP is reliable but somewhat slow. When sending sounds or images in real time, it is acceptable if a small amount of the data is lost. But there is no point in transmitting data that is late.
3. The browser software translates your requests (typed URLs and mouse clicks on links) into HTTP commands that it sends to the appropriate web servers.
4. Some Telnet implementations send all keystrokes that you type to the server, including the backspace key. The server does not recognize a character sequence such as G W Backspace E T as a valid command.
5. The program makes a connection to the server, sends the `GET` request, and prints the error message that the server returns.
6. `Socket s = new Socket("e-mail.sjsu.edu", 110);`
7. Port 80 is the standard port for HTTP. If a web server is running on the same computer, then one can't open a server socket on an open port.
8. No, a server socket just waits for a connection and yields a regular `Socket` object when a client has connected. You use that socket object to read the data that the client sends.
9. The `URLConnection` class understands the HTTP protocol, freeing you from assembling requests and analyzing response headers.
10. The bytes that encode the images are displayed on the console, but they will appear to be random gibberish.