20 MULTITHREADING

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.

20.1 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 subclass:

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

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

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

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

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

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

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.

```
Fri Dec 28 23:12:03 PST 2012 Hello, World!
Fri Dec 28 23:12:04 PST 2012 Hello, World!
Fri Dec 28 23:12:05 PST 2012 Hello, World!
Fri Dec 28 23:12:06 PST 2012 Hello, World!
Fri Dec 28 23:12:07 PST 2012 Hello, World!
Fri Dec 28 23:12:08 PST 2012 Hello, World!
```

```
Fri Dec 28 23:12:09 PST 2012 Hello, World!
Fri Dec 28 23:12:10 PST 2012 Hello, World!
Fri Dec 28 23:12:11 PST 2012 Hello, World!
Fri Dec 28 23:12:12 PST 2012 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 20.2, 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

```
Fri Dec 28 12:04:46 PST 2012 Hello
Fri Dec 28 12:04:46 PST 2012 Goodbye
Fri Dec 28 12:04:47 PST 2012 Hello
Fri Dec 28 12:04:47 PST 2012 Goodbye
Fri Dec 28 12:04:48 PST 2012 Hello
Fri Dec 28 12:04:48 PST 2012 Goodbye
Fri Dec 28 12:04:49 PST 2012 Hello
Fri Dec 28 12:04:49 PST 2012 Goodbye
Fri Dec 28 12:04:50 PST 2012 Hello
Fri Dec 28 12:04:50 PST 2012 Goodbye
Fri Dec 28 12:04:51 PST 2012 Hello
Fri Dec 28 12:04:51 PST 2012 Goodbye
Fri Dec 28 12:04:52 PST 2012 Goodbye
Fri Dec 28 12:04:52 PST 2012 Hello
Fri Dec 28 12:04:53 PST 2012 Hello
Fri Dec 28 12:04:53 PST 2012 Goodbye
Fri Dec 28 12:04:54 PST 2012 Hello
Fri Dec 28 12:04:54 PST 2012 Goodbye
Fri Dec 28 12:04:55 PST 2012 Goodbye
Fri Dec 28 12:04:55 PST 2012 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: R20.2, R20.3, P20.7.

Programming Tip 20.1

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 20.1 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 20.1

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.

20.2 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 20.3.) 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: R20.4, R20.5, R20.6.

Programming Tip 20.2

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
}
```

20.3 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

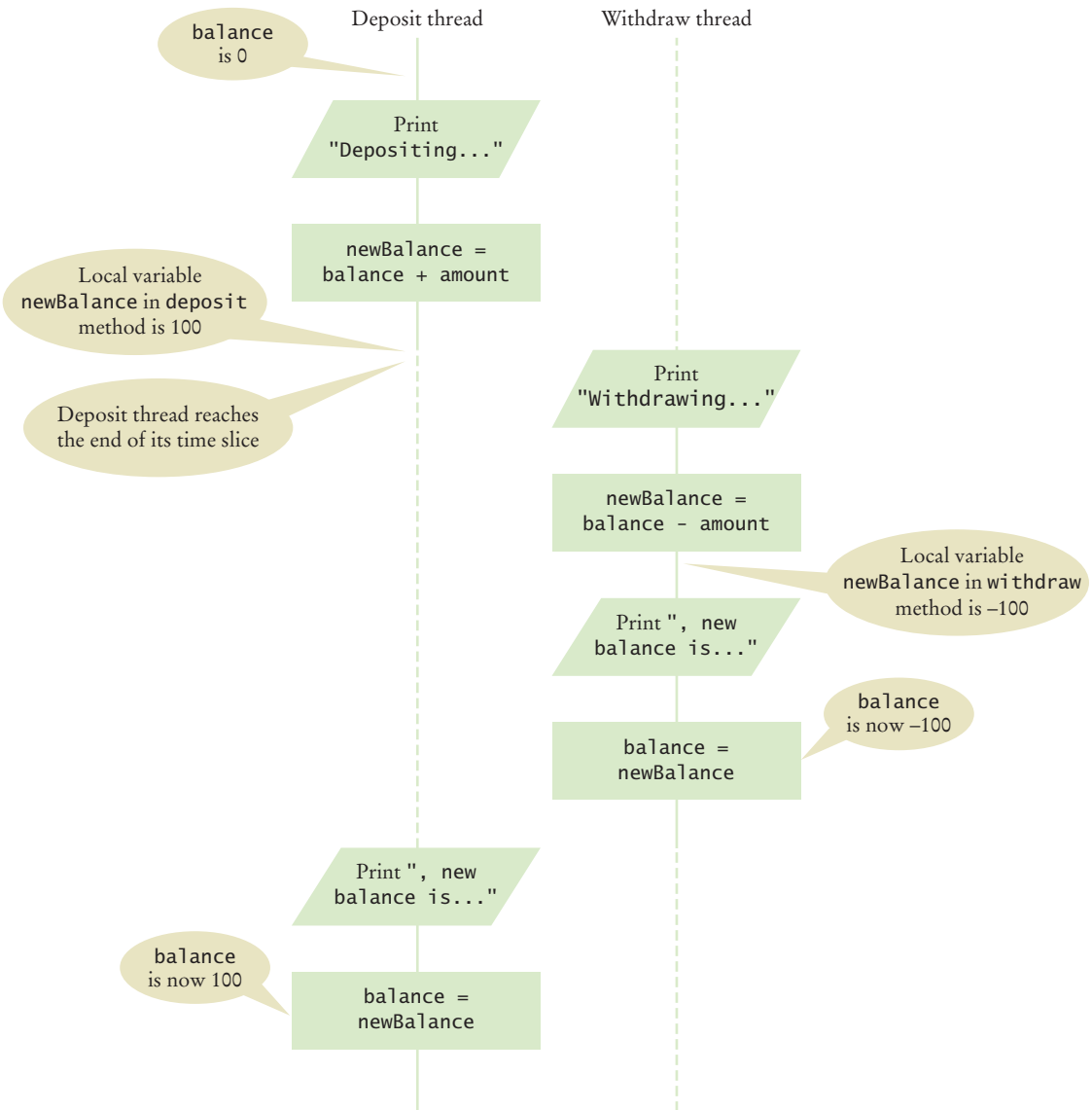


Figure 1 Corrupting the Contents of the `balance` Variable

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

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.

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: R20.8, R20.9, P20.1.

20.4 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 20.2).

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

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: P20.2, P20.6, P20.8.

20.5 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: R20.12, P20.3, P20.4, P20.5.

Common Error 20.1

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 20.2

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 20.2

Object Locks and Synchronized Methods

The `Lock` and `Condition` classes 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 20.3

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.

20.6 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 P20.19 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();
   }
}
```

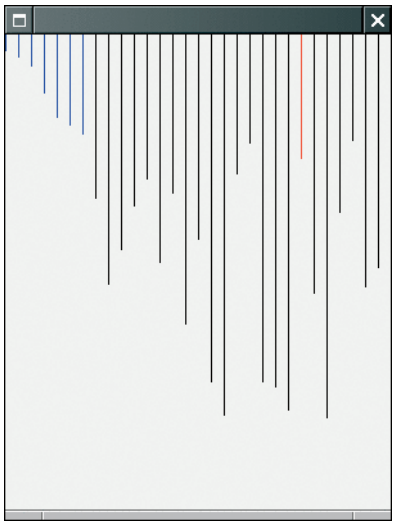


Figure 3 A Step in the Animation of the Selection Sort Algorithm

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 P20.17 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: R20.14, P20.14, P20.16.

Random Fact 20.1 Embedded Systems

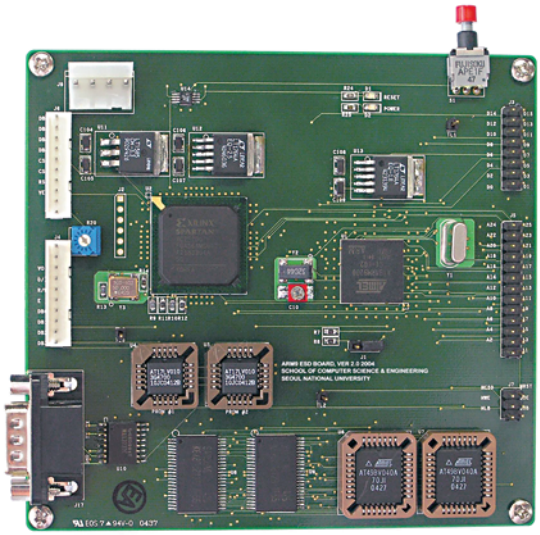
An *embedded system* is a computer system that controls a device. The device contains a processor and other hardware and is controlled by a computer program. Unlike a personal computer, which has been designed to be flexible and run many different computer programs, the hardware and software of an embedded system are tailored to a specific device. Computer-controlled devices are becoming increasingly common, ranging from washing machines to medical equipment, automobile engines, and spacecraft.

Several challenges are specific to programming embedded systems. Most importantly, a much higher standard of quality control applies. Vendors are often unconcerned about bugs in personal computer software, because they can always make you install a patch or upgrade to the next version. But in an embedded system, that is not an option. Few consumers would feel comfortable upgrading the software in their washing machines or automobile engines. If you ever handed in a programming assignment that you believed to be correct, only to have the instructor or grader find bugs in it, then you know how hard it is to write software that can reliably do its task for many years without a chance of changing it.

Quality standards are especially important in devices whose failure would destroy property or human life—see Random Fact 7.2.

Many personal computer purchasers buy computers that are fast and have a lot of storage, because the investment is paid back over time when many programs are run on the same equipment. But the hardware for an embedded device is not shared—it is dedicated to one device. A separate processor, memory, and so on, are built for every copy of the device (see the photo). If it is possible to shave a few pennies off the manufacturing cost of every unit, the savings can add up quickly for devices that are produced in large volumes. Thus, the embedded-system programmer has a much larger economic incentive to conserve resources than the programmer of desktop software. Unfortunately, trying to conserve resources usually makes it harder to write programs that work correctly.

Generally, embedded systems are written in lower-level programming languages to avoid the overhead of a complex run-time system. The Java run-time system, with its safety mechanisms, garbage collector, support for multithreading, and so on, would be too costly to add to every washing machine. However, some devices are now being built with a scaled-down version of Java: the Java 2 Micro Edition. Examples are smart cell phones and onboard computers for automobiles. The Java 2 Micro Edition is a good candidate for devices that are connected to a network and that need to be able to run new applications safely. For example, you can download a program into a Java-enabled cell phone and be assured that it cannot corrupt other parts of the cell phone software.



The Controller of an Embedded System

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

■ **R20.1** 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.

■■■ **R20.2** In the program of Section 20.1, 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.

■■■ **R20.3** In Java, a graphical user interface program has more than one thread. Explain how you can prove that.

■■■ **R20.4** Why is the `stop` method for stopping a thread deprecated? How do you terminate a thread?

■ **R20.5** Give an example of why you would want to terminate a thread.

■■ **R20.6** 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?

■■ **R20.7** What is a race condition? How can you avoid it?

■■ **R20.8** Consider the `ArrayList` implementation from Section 16.2. Describe two different scenarios in which race conditions can corrupt the data structure.

■■ **R20.9** 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.

■■ **R20.10** 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.

■■ **R20.11** What is a deadlock? How can you avoid it?

■ **R20.12** What is the difference between a thread that sleeps by calling `sleep` and a thread that waits by calling `await`?

■ **R20.13** What happens when a thread calls `await` and no other thread calls `signalAll` or `signal`?

■■ **R20.14** In the algorithm animation program of Section 20.6, we do not use any conditions. Why not?

PROGRAMMING EXERCISES

■■ **P20.1** Write a program in which multiple threads add and remove elements from a `java.util.LinkedList`. Demonstrate that the list is being corrupted.

■■ **P20.2** Implement a stack as a linked list in which the `push`, `pop`, and `isEmpty` methods can be safely accessed from multiple threads.

■■■ **P20.3** 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 10 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.

■ **P20.4** Enhance the program of Exercise P20.3 by supplying a variable number of producer and consumer threads. Prompt the program user for the numbers.

■ **P20.5** Reimplement Exercise P20.4 by using the `ArrayBlockingQueue` class from the standard library.

■■ **P20.6** Modify the `ArrayList` implementation of Section 16.2 so that all methods can be safely accessed from multiple threads.

■■ **P20.7** 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
```

■■■ **P20.8** Enhance the program of Exercise P20.7 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.

■■ **P20.9** 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
```

■■ **P20.10** Add a condition to the `deposit` method of the `BankAccount` class in Section 20.5, 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.

■■■ **P20.11** 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 P20.12** Write a program that shows two cars moving across a window. Use a separate thread for each car.

■■■ **Graphics P20.13** Modify Exercise P20.12 so that the cars change direction when they hit an edge of the window.

■ **Graphics P20.14** Enhance the `SelectionSorter` of Section 20.6 so that the current minimum is painted in yellow.

■■ **Graphics P20.15** Enhance the `SelectionSortViewer` of Section 20.6 so that the sorting only starts when the user clicks a "Start" button.

■■ **Graphics P20.16** Instead of using a thread and a `pause` method, use the `Timer` class introduced in Chapter 11 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 P20.17** 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 P20.18** Enhance Exercise P20.17 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 P20.19** Reorganize the code of the sorting animation in Section 20.6 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.