The following pages are (most of) Chapter 20 from

Cay Horstmann, Big Java, 3rd ed., Wiley, 2008.

Chapter 20

Multithreading

CHAPTER GOALS

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

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 of 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 you can implement this behavior by running tasks in multiple threads, and how you can ensure that the tasks access shared data in a controlled fashion.

Chapter Contents

20.1 Running Threads

A thread is a program unit that is executed independently of 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. Implement 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 go here
      . . .
   }
}
```

The start m
Thread class
thread that e
run method
associated R
object.

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);
```

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

```
Thu Dec 28 23:12:03 PST 2006 Hello, World!
Thu Dec 28 23:12:04 PST 2006 Hello, World!
Thu Dec 28 23:12:05 PST 2006 Hello, World!
Thu Dec 28 23:12:06 PST 2006 Hello, World!
Thu Dec 28 23:12:07 PST 2006 Hello, World!
Thu Dec 28 23:12:08 PST 2006 Hello, World!
Thu Dec 28 23:12:09 PST 2006 Hello, World!
Thu Dec 28 23:12:10 PST 2006 Hello, World!
Thu Dec 28 23:12:11 PST 2006 Hello, World!
Thu Dec 28 23:12:12 PST 2006 Hello, World!
```

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

```
public class GreetingRunnable implements Runnable
{
   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

   public void run()
   {
      // Task statements go here
      . . .
   }
   // Fields used by the task statements
   private String greeting;
}
```

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. Its default constructor 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.

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

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.

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 the runnable class:

ch20/greeting/GreetingRunnable.java

```
import java.util.Date;

/**
   A runnable that repeatedly prints a greeting.
*/
public class GreetingRunnable implements Runnable
{
   /**
      Constructs the runnable object.
      @param aGreeting the greeting to display
   */
   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

   public void run()
   {
      try
      {
```

```java
         for (int i = 1; i <= REPETITIONS; i++)
         {
            Date now = new Date();
            System.out.println(now + " " + greeting);
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException exception)
      {
      }
   }

   private String greeting;

   private static final int REPETITIONS = 10;
   private static final int DELAY = 1000;
}
```

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

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

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

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

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

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

ch20/greeting/GreetingThreadRunner.java

```java
/**
   This program runs two greeting threads in parallel.
*/
public class GreetingThreadRunner
{
   public static void main(String[] args)
   {
      GreetingRunnable r1 = new GreetingRunnable("Hello, World!");
      GreetingRunnable r2 = new GreetingRunnable("Goodbye, World!");
      Thread t1 = new Thread(r1);
      Thread t2 = new Thread(r2);
      t1.start();
      t2.start();
   }
}
```

Output

```
Tue Dec 19 12:04:46 PST 2006 Hello, World!
Tue Dec 19 12:04:46 PST 2006 Goodbye, World!
Tue Dec 19 12:04:47 PST 2006 Hello, World!
Tue Dec 19 12:04:47 PST 2006 Goodbye, World!
Tue Dec 19 12:04:48 PST 2006 Hello, World!
Tue Dec 19 12:04:48 PST 2006 Goodbye, World!
Tue Dec 19 12:04:49 PST 2006 Hello, World!
Tue Dec 19 12:04:49 PST 2006 Goodbye, World!
Tue Dec 19 12:04:50 PST 2006 Hello, World!
Tue Dec 19 12:04:50 PST 2006 Goodbye, World!
Tue Dec 19 12:04:51 PST 2006 Hello, World!
Tue Dec 19 12:04:51 PST 2006 Goodbye, World!
Tue Dec 19 12:04:52 PST 2006 Goodbye, World!
Tue Dec 19 12:04:52 PST 2006 Hello, World!
Tue Dec 19 12:04:53 PST 2006 Hello, World!
Tue Dec 19 12:04:53 PST 2006 Goodbye, World!
Tue Dec 19 12:04:54 PST 2006 Hello, World!
Tue Dec 19 12:04:54 PST 2006 Goodbye, World!
Tue Dec 19 12:04:55 PST 2006 Hello, World!
Tue Dec 19 12:04:55 PST 2006 Goodbye, World!
```

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.

Self Check

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?

Quality 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 go here
      . . .
   }
}
```

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. Advanced 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.

Advanced 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 space and run-time cost for each thread that is created. 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, World!");
Runnable r2 = new GreetingRunnable("Goodbye, World!");
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.

A thread terminates when the `run` method of the associated runnable object returns. 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 try 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` field 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 `Interrupted-Exception` 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
      }
   }
   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.

SELF CHECK

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?

Quality 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.

Simply 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, out of ignorance and desperation, muzzle it by surrounding 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` field is still 0, and the value of the `newBalance` local variable is 100.
2. Immediately afterwards, 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 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` field.

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 field. 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 field, 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);
}
```

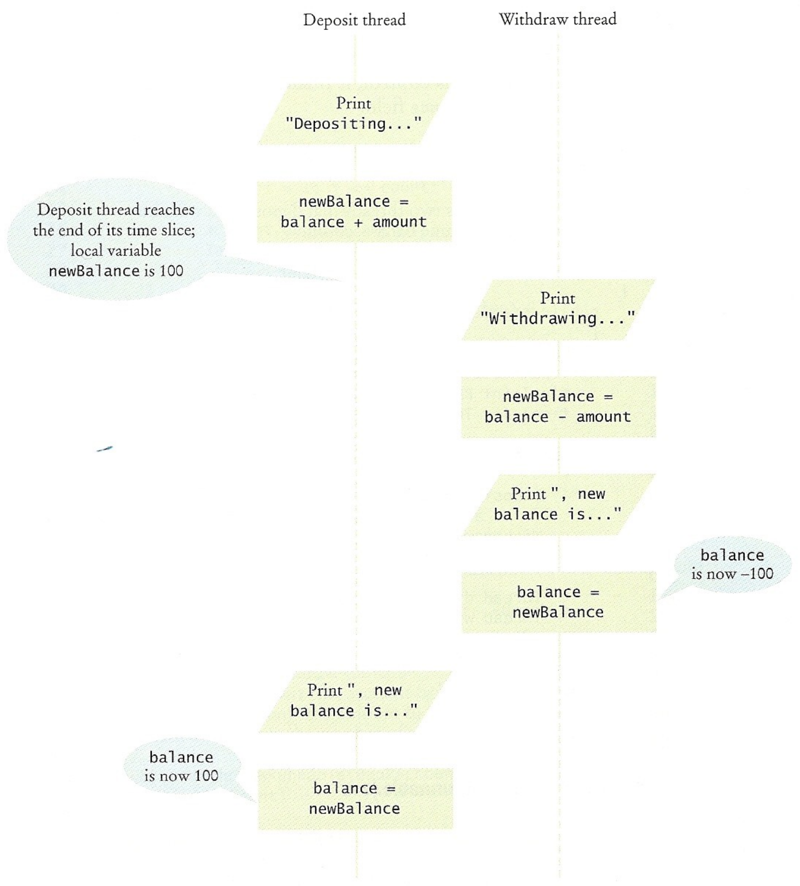
Deposit thread
Withdraw thread
Print "Depositing..."
newBalance = balance + amount
Deposit thread reaches the end of its time slice; local variable newBalance is 100
Print "Withdrawing..."
newBalance = balance - amount
Print ", new balance is..."
balance is now −100
balance = newBalance
Print ", new balance is..."
balance is now 100
balance = newBalance

Figure 1 Corrupting the Contents of the `balance` Field

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` field.

ch20/unsynch/BankAccountThreadRunner.java

```java
/**
   This program runs two 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);

         dt.start();
         wt.start();
      }
   }
}
```

ch20/unsynch/DepositRunnable.java

```java
/**
   A deposit runnable makes periodic deposits to a bank account.
*/
public class DepositRunnable implements Runnable
{
   /**
      Constructs a deposit runnable.
      @param anAccount the account into which to deposit money
      @param anAmount the amount to deposit in each repetition
      @param aCount the number of repetitions
   */
   public DepositRunnable(BankAccount anAccount, double anAmount,
         int aCount)
   {
      account = anAccount;
      amount = anAmount;
      count = aCount;
   }

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

   private static final int DELAY = 1;
   private BankAccount account;
   private double amount;
   private int count;
}
```

ch20/unsynch/WithdrawRunnable.java

```
/**
   A withdraw runnable makes periodic withdrawals from a bank account.
*/
public class WithdrawRunnable implements Runnable
{
   /**
      Constructs a withdraw runnable.
      @param anAccount the account from which to withdraw money
      @param anAmount the amount to deposit in each repetition
      @param aCount the number of repetitions
   */
   public WithdrawRunnable(BankAccount anAccount, double anAmount,
         int aCount)
   {
      account = anAccount;
      amount = anAmount;
      count = aCount;
   }

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

```java
      }
      catch (InterruptedException exception) {}
   }

   private static final int DELAY = 1;
   private BankAccount account;
   private double amount;
   private int count;
}
```

ch20/unsynch/BankAccount.java

```java
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      System.out.print("Depositing " + amount);
      double newBalance = balance + amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }

   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
      System.out.print("Withdrawing " + amount);
      double newBalance = balance - amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }

   private double balance;
}
```

Output

```
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
```

SELF CHECK

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 15, explain how the list can be damaged in the process.

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 Advanced Topic 20.2)

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

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

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

```
balanceChangeLock.lock();
```
Code that manipulates 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
{
   Code that manipulates 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, it 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.

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

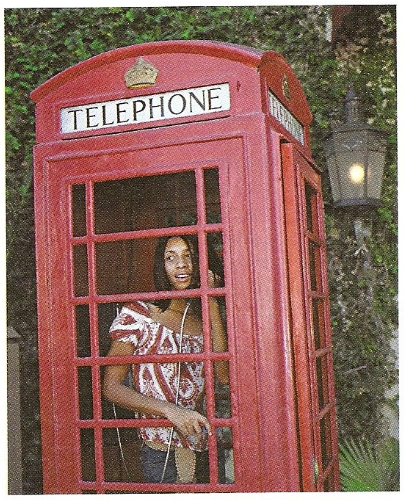

Figure 2
Visualizing Object Locks

at one 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. (Some computer programmers think of a rest-room stall instead of a telephone booth in order to visualize object locks. However, in the interest of good taste, we will not develop that analogy any further.)

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.

Self Check

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?

20.5 Avoiding Deadlocks

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

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.

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

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. 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();
      }
   }
```

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

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

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.

ch20/synch/BankAccountThreadRunner.java

```
/**
   This program runs four 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;

```

```java
      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);

         dt.start();
         wt.start();
      }
   }
}
```

ch20/synch/BankAccount.java

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition = balanceChangeLock.newCondition();
   }

   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   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;
         sufficientFundsCondition.signalAll();
      }
      finally
      {
         balanceChangeLock.unlock();
      }
   }

   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
         throws InterruptedException
   {
      balanceChangeLock.lock();
      try
      {
         while (balance < amount)
            sufficientFundsCondition.await();
         System.out.print("Withdrawing " + amount);
         double newBalance = balance - amount;
         System.out.println(", new balance is " + newBalance);
         balance = newBalance;
      }
      finally
      {
         balanceChangeLock.unlock();
      }
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }

   private double balance;
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
}
```

Output

```
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
```

Self Check

9. What is the essential difference between calling `sleep` and `await`?

10. Why is the `sufficientFundsCondition` object a field of the `BankAccount` class and not a local variable of the `withdraw` and `deposit` methods?

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`.

Advanced 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` keyword.

```
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 the lock of that object 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 other thread is automatically deactivated, and it needs to wait until the first thread has unlocked the object again.

In other words, the `synchronized` keyword 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` keyword. If you need more control over threads, the `Lock` and `Condition` interfaces give you additional flexibility.

ADVANCED 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. Since 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 Case Study: 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 P20.12 for an example. However, more advanced animations are best implemented with threads.