19 Java Never Ends

And thick and fast they came at last,
And more, and more, and more—

LEWIS CARROLL, *Through the Looking-Glass*

Introduction

Of course there is only a finite amount of Java, but when you consider all the standard libraries and other accompanying software, the amount of power and the amount to learn seem to be endless. In this chapter, we give you a brief introduction to five topics to give you a flavor of some of the directions you can take in extending your knowledge of Java. The five topics are multithreading, networking with stream sockets, JavaBeans, the interaction of Java with database systems, and Web programming with Java Server Pages.

Prerequisites

You really should cover most of the book before covering this chapter. However, Section 19.1 requires only Chapters 17 and 18 and their prerequisites. Section 19.2 requires Chapters 1 through 5, 9, and 10. Sections 19.3 and 19.4 require only Chapters 1 through 6. Section 19.5 requires an understanding of HTML, which is given in Chapter 20. Chapter 20 is distributed as a file on the website included in this book. Aside from references to Section 19.1 in Section 19.2, all sections are independent of each other and may be read in any order.

19.1 Multithreading

"Can you do two things at once?"
"I have trouble doing one thing at once."

Part of a job interview

thread

A **thread** is a separate computation process. In Java, you can have programs with multiple threads. You can think of the threads as computations that execute in parallel. On a computer with enough processors, the threads might indeed execute in parallel. However, in most normal computing situations, the threads do not really do this. Instead, the computer switches resources between threads so that each thread in turn does a little bit of computing. To the user, this looks like the processes are executing in parallel.

You have already experienced threads. Modern operating systems allow you to run more than one program at the same time. For example, rather than waiting for your virus

scanning program to finish its computation, you can go on to, say, read your e-mail while the virus scanning program is still executing. The operating system is using threads to make this happen. There may or may not be some work being done in parallel depending on your computer and operating system. Most likely, the two computation threads are simply sharing computer resources so that they take turns using the computer's resources. When reading your e-mail, you may or may not notice that response is slower because resources are being shared with the virus scanning program. Your e-mail reading program is indeed slowed down, but because humans are so much slower than computers, any apparent slowdown is likely to be unnoticed.

EXAMPLE: A Nonresponsive GUI

Display 19.1 contains a very simple action GUI. When the `"Start"` button is clicked, the GUI draws circles one after the other until a large portion of the window is filled with circles. There is 1/10 of a second pause between the drawing of each circle. So, you can see the circles appear one after the other. If you are interested in Java programming, this can be pretty exciting for the first few circles, but it quickly becomes boring. You are likely to want to end the program early, but if you click the close-window button, nothing will happen until the program is finished drawing all its little circles. We will use threads to fix this problem, but first let us understand this program, which does not really use threads in any essential way, despite the occurrence of the word `Thread` in the program. We explain this Swing program in the next few subsections.

`Thread.sleep`

In Display 19.1, the following method invocation produces a 1/10 of a second pause after drawing each of the circles:

```
doNothing(PAUSE);
```

which is equivalent to

```
doNothing(100);
```

The method `doNothing` is a private helping method that does nothing except call the method `Thread.sleep` and take care of catching any thrown exception. So, the pause is really created by the method invocation

`Thread.sleep`

```
Thread.sleep(100);
```

This is a static method in the class `Thread` that pauses whatever thread includes the invocation. It pauses for the number of milliseconds (thousandths of a second) given as an argument. So, this pauses the computation of the program in Display 19.1 for 100 milliseconds or 1/10 of a second.

"Wait a minute," you may think, "the program in Display 19.1 was not supposed to use threads in any essential way." That is basically true, but every Java program uses threads in some way. If there is only one stream of computation, as in Display 19.1, then that is treated as a single thread by Java. So, threads are always used by Java, but not in an interesting way until more than one thread is used.

You can safely think of the invocation of

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

as a pause in the computation that lasts (approximately) the number of milliseconds given as the argument. (If this invocation is in a thread of a multithreaded program, then the pause, like anything else in the thread, applies only to the thread in which it occurs.)

The method `Thread.sleep` can sometimes be handy even if you do not do any multithreaded programming. The class `Thread` is in the package `java.lang` and so requires no import statement.

Display 19.1 Nonresponsive GUI (part 1 of 3)

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JButton;
4  import java.awt.BorderLayout;
5  import java.awt.FlowLayout;
6  import java.awt.Graphics;
7  import java.awt.event.ActionListener;
8  import java.awt.event.ActionEvent;

9  /**
10 Packs a section of the frame window with circles, one at a time.
11 */
12 public class FillDemo extends JFrame implements ActionListener
13 {
14     public static final int WIDTH = 300;
15     public static final int HEIGHT = 200;
16     public static final int FILL_WIDTH = 300;
17     public static final int FILL_HEIGHT = 100;
18     public static final int CIRCLE_SIZE = 10;
19     public static final int PAUSE = 100; //milliseconds

20     private JPanel box;

21     public static void main(String[] args)
22     {
23         FillDemo gui = new FillDemo();
24         gui.setVisible(true);
25     }
```

Display 19.1 Nonresponsive GUI (part 2 of 3)

```
26      public FillDemo()
27      {
28          setSize(WIDTH, HEIGHT);
29          setTitle("FillDemo");
30          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31          setLayout(new BorderLayout());
32          box = new JPanel();
33          add(box, "Center");

34          JPanel buttonPanel = new JPanel();
35          buttonPanel.setLayout(new FlowLayout());
36          JButton startButton = new JButton("Start");
37          startButton.addActionListener(this);
38          buttonPanel.add(startButton);
39          add(buttonPanel, "South");
40      }

41      public void actionPerformed(ActionEvent e)
42      {
43          fill( );
44      }

45      public void fill()
46      {
47          Graphics g = box.getGraphics();

48          for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
49              for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
50              {
51                  g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
52                  doNothing(PAUSE);
53              }
54      }

55      public void doNothing(int milliseconds)
56      {
57          try
58          {
59              Thread.sleep(milliseconds);
60          }
61          catch(InterruptedException e)
62          {
63              System.out.println("Unexpected interrupt");
64              System.exit(0);
65          }
66      }
67  }
```

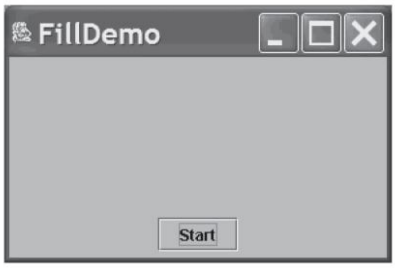
Nothing else can happen until **`actionPerformed`** *returns, which does not happen until* **`fill`** *returns.*

Everything stops for 100 milliseconds (1/10 of a second).

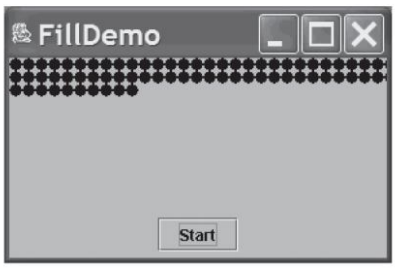
(continued)

Display 19.1 Nonresponsive GUI (part 3 of 3)

RESULTING GUI (When started)



RESULTING GUI (While drawing circles)



If you click the close-window button while the circles are being drawn, the window will not close until all the circles are drawn.

RESULTING GUI (After all circles are drawn)

The method `Thread.sleep` can throw an `InterruptedException`, which is a checked exception—that is, it must be either caught in a `catch` block or declared in a `throws` clause. We do not discuss `InterruptedException` in this book, leaving it for more advanced books on multithreaded programming, but it has to do with one thread interrupting another thread. We will simply note that an `InterruptedException` may be thrown by `Thread.sleep` and so must be accounted for—in our case, by a simple `catch` block. The class `InterruptedException` is in the `java.lang` package and so requires no import statement.

Thread.sleep

`Thread.sleep` is a static method in the class `Thread` that pauses the thread that includes the invocation. It pauses for the number of milliseconds (thousandths of a second) given as an argument.

The method `Thread.sleep` may throw an `InterruptedException`, which is a checked exception and so must be either caught in a `catch` block or declared in a `throws` clause.

The classes `Thread` and `InterruptedException` are both in the package `java.lang`, so neither requires any import statement.

Note that `Thread.sleep` can be invoked in an ordinary (single thread) program of the kind we have seen before this chapter. It will insert a pause in the single thread of that program.

SYNTAX

```
Thread.sleep(Number_Of_Milliseconds);
```

EXAMPLE

```
try
{
    Thread.sleep(100); //Pause of 1/10 of a second
}
catch(InterruptedException e)
{
    System.out.println("Unexpected interrupt");
}
```

The `getGraphics` Method

The other new method in Display 19.1 is the `getGraphics` method, which is used in the following line from the method `fill`:

`getGraphics`

```
Graphics g = box.getGraphics();
```

The `getGraphics` method is almost self-explanatory. As we already noted in Chapter 18, almost every item displayed on the screen (more precisely, every `JComponent`) has an associated `Graphics` object. The method `getGraphics` is an accessor method that returns the associated `Graphics` object (of the calling object for `getGraphics`)—in this case, the `Graphics` object associated with the panel box. This gives us a `Graphics` object that can draw circles (or anything else) in the panel box.

We still need to say a bit more about why the program in Display 19.1 makes you wait before it will respond to the close-window button, but otherwise this concludes our explanation of Display 19.1. The rest of the code consists of standard things we have seen before.

getGraphics

Every `JComponent` has an associated `Graphics` object. The method `getGraphics` is an accessor method that returns the associated `Graphics` object of its calling object.

SYNTAX

```
Component.getGraphics( );
```

EXAMPLE (see Display 19.1 for context)

```
Graphics g = box.getGraphics( );
```

Fixing a Nonresponsive Program Using Threads

Now that we have discussed the new items in the program in Display 19.1, we are ready to explain why it is nonresponsive and to show you how to use threads to write a responsive version of that program.

Recall that when you run the program in Display 19.1, it draws circles one after the other to fill a portion of the frame. Although there is only a 1/10 of a second pause between drawing each circle, it can still seem like it takes a long time to finish. So, you are likely to want to abort the program and close the window early. But, if you click the close-window button, the window will not close until the GUI is finished drawing all the circles.

Here is why the close-window button is nonresponsive: The method `fill`, which draws the circles, is invoked in the body of the method `actionPerformed`. So, the method `actionPerformed` does not end until after the method `fill` ends. And, until the method `actionPerformed` ends, the GUI cannot go on to do the next thing, which is probably to respond to the close-window button.

Here is how we fixed the problem: We have the method `actionPerformed` create a new (independent) thread to draw the circles. Once `actionPerformed` does this, the new thread is an independent process that proceeds on its own. The method `actionPerformed` has nothing more to do with this new thread; the work of `actionPerformed` is ended. So, the main thread (the one with `actionPerformed`) is ready to move on to the next thing, which will probably be to respond promptly to a click of the close-window button. At the same time, the new thread draws the circles. So, the circles are drawn, but at the same time a click of the close-window button will end the program. The program that implements this multithreaded solution is given in the next Programming Example.

EXAMPLE: A Multithreaded Program

Display 19.2 contains a program that uses a main thread and a second thread to implement the technique discussed in the previous subsection. The general approach was outlined in the previous subsection, but we need to explain the Java code details. We do that in the next few subsections.

The Class `Thread`

`Thread`

In Java, a thread is an object of the class `Thread`. The normal way to program a thread is to define a class that is a derived class of the class `Thread`. An object of this derived class will be a thread that follows the programming given in the definition of the derived (thread) class.

`run()`

Where do you do the programming of a thread? The class `Thread` has a method named `run`. The definition of the method `run` is the code for the thread. When the thread is executed, the method `run` is executed. Of course, the method defined in the class `Thread` and inherited by any derived class of `Thread` does not do what you want your thread to do. So, when you define a derived class of `Thread`, you override the definition of the method `run` to do what you want the thread to do.

In Display 19.2, the inner class `Packer` is a derived class of the class `Thread`. The method `run` for the class `Packer` is defined to be exactly the same as the method `fill` in our previous, unresponsive GUI (Display 19.1). So, an object of the class `Packer` is a thread that will do what `fill` does, namely draw the circles to fill up a portion of the window.

The method `actionPerformed` in Display 19.2 differs from the method `actionPerformed` in our older, nonresponsive program (Display 19.1) in that the

`start()`

invocation of the method `fill` is replaced with the following:

```
Packer packerThread = new Packer( );
packerThread.start( );
```

This creates a new, independent thread named `packerThread` and starts it processing. Whatever `packerThread` does, it does as an independent thread. The main thread can then allow `actionPerformed` to end and the main thread will be ready to respond to any click of the close-window button.

Display 19.2 **Threaded Version of `FillDemo`** (part 1 of 3)

The GUI produced is identical to the GUI produced by Display 19.1 except that in this version the close-window button works even while the circles are being drawn, so you can end the GUI early if you get bored.

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JButton;
4  import java.awt.BorderLayout;
5  import java.awt.FlowLayout;
6  import java.awt.Graphics;
7  import java.awt.event.ActionListener;
8  import java.awt.event.ActionEvent;

9  public class ThreadedFillDemo extends JFrame implements ActionListener
10 {
11     public static final int WIDTH = 300;
12     public static final int HEIGHT = 200;
13     public static final int FILL_WIDTH = 300;
14     public static final int FILL_HEIGHT = 100;
15     public static final int CIRCLE_SIZE = 10;
16     public static final int PAUSE = 100; //milliseconds
```

(continued)

Display 19.2 Threaded Version of `FillDemo` (part 2 of 3)

```
17      private JPanel box;

18      public static void main(String[] args)
19      {
20          ThreadedFillDemo gui = new ThreadedFillDemo();
21          gui.setVisible(true);
22      }

23      public ThreadedFillDemo()
24      {
25          setSize(WIDTH, HEIGHT);
26          setTitle("Threaded Fill Demo");
27          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

28          setLayout(new BorderLayout());

29          box = new JPanel();
30          add(box, "Center");

31          JPanel buttonPanel = new JPanel();
32          buttonPanel.setLayout(new FlowLayout());
33          JButton startButton = new JButton("Start");
34          startButton.addActionListener(this);
35          buttonPanel.add(startButton);
36          add(buttonPanel, "South");
37      }

38      public void actionPerformed(ActionEvent e)
39      {
40          Packer packerThread = new Packer();
41          packerThread.start();
42      }

43      private class Packer extends Thread
44      {
45          public void run()
46          {
47              Graphics g = box.getGraphics();
48              for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
49                for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
50                {
51                    g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
52                    doNothing(PAUSE);
53                }
54          }
```

You need a thread object, even if there are no instance variables in the class definition of Packer.

`start` "starts" the thread and calls `run`.

`run` is inherited from `Thread` but needs to be overridden. This definition of `run` is identical to that of `fill` in Display 19.1.

Display 19.2 Threaded Version of `FillDemo` (part 3 of 3)

```
55          public void doNothing(int milliseconds)
56          {
57              try
58              {
59                  Thread.sleep(milliseconds);
60              }
61              catch(InterruptedException e)
62              {
63                  System.out.println("Unexpected interrupt");
64                  System.exit(0);
65              }
66          }
67      } //End Packer inner class

68  }
```

run()

We need only to discuss the method `start` and we will be through with our explanation. The method `start` initiates the computation (process) of the calling thread. It performs some overhead associated with starting a thread and then it invokes the `run` method for the thread. As we have already seen, the `run` method of the class `Packer` in Display 19.2 draws the circles we want, so the invocation

```
packerThread.start( );
```

does this as well, because it calls `run`. Note that you do not invoke `run` directly. Instead, you invoke `start`, which does some other needed things and then invokes `run`.

This ends our explanation of the multithreaded program in Display 19.2, but there is still one, perhaps puzzling, thing about the class `Packer` that we should explain. The definition of the class `Packer` includes no instance variables. So, why do we need to bother with an object of the class `Packer`? Why not simply make all the methods static and call them with the class name `Packer`? The answer is that the only way to get a new thread is to create a new `Thread` object. The things inherited from the class `Thread` are what the object needs to be a thread. Static methods do not a thread make. In fact, not only will static methods not work, the compiler will not even allow you to define `run` to be static. This is because `run` is inherited from `Thread` as a nonstatic method; this cannot be changed to static when overriding a method definition. The compiler will not let you even try to do this without creating an object of the class `Packer`.

The `Thread` Class

A thread is an object of the class `Thread`. The normal way to program a thread is to define a class that is a derived class of the class `Thread`. An object of this derived class will be a thread that follows the programming given in the definition of the derived (thread) class's method named `run`.

Any thread class inherits the method start from the class `Thread`. An invocation of `start` by an object of a thread class will start the thread and invoke the method `run` for that thread.

See Display 19.2 for an example.

The `Runnable` Interface ★

There are times when you would rather not make a thread class a derived class of the class `Thread`. The alternative to making your class a derived class of the class `Thread` is to have your class instead implement the `Runnable` interface. The `Runnable` interface has only one method heading:

```
public void run()
```

A class that implements the `Runnable` interface must still be run from an instance of the class `Thread`. This is usually done by passing the `Runnable` object as an argument to the thread constructor. The following is an outline of one way to do this:

```
public class ClassToRun extends SomeClass implements Runnable
{
     ....
    public void run()
    {
        //Fill this just as you would if ClassToRun
        //were derived from Thread.
    }
     ....
    public void startThread()
    {
        Thread theThread = new Thread(this);
        theThread.run();
    }
     ....
}
```

The previous method `startThread` is not compulsory, but it is one way to produce a thread that will in turn run the `run` method of an object of the class `ClassToRun`. In Display 19.3, we have rewritten the program in Display 19.2 using the `Runnable` interface. The program behaves exactly the same as the one in Display 19.2.

Display 19.3 The `Runnable` Interface (part 1 of 2)

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JButton;
4  import java.awt.BorderLayout;
5  import java.awt.FlowLayout;
6  import java.awt.Graphics;
7  import java.awt.event.ActionListener;
8  import java.awt.event.ActionEvent;

9  public class ThreadedFillDemo2 extends JFrame
10                                 implements ActionListener, Runnable
11 {
12     public static final int WIDTH = 300;
13     public static final int HEIGHT = 200;
14     public static final int FILL_WIDTH = 300;
15     public static final int FILL_HEIGHT = 100;
16     public static final int CIRCLE_SIZE = 10;
17     public static final int PAUSE = 100; //milliseconds
18     private JPanel box;

19     public static void main(String[] args)
20     {
21         ThreadedFillDemo2 gui = new ThreadedFillDemo2();
22         gui.setVisible(true);
23     }

24     public ThreadedFillDemo2()
25     {
26         setSize(WIDTH, HEIGHT);
27         setTitle("Threaded Fill Demo");
28         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

29         setLayout(new BorderLayout());

30         box = new JPanel();
31         add(box, "Center");

32         JPanel buttonPanel = new JPanel();
33         buttonPanel.setLayout(new FlowLayout());

34         JButton startButton = new JButton("Start");
35         startButton.addActionListener(this);
36         buttonPanel.add(startButton);
37         add(buttonPanel, "South");
38     }
```

(continued)

Display 19.3 The `Runnable` Interface (part 2 of 2)

```
39      public void actionPerformed(ActionEvent e)
40      {
41          startThread();
42      }

43      public void run()
44      {
45          Graphics g = box.getGraphics();
46          for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
47            for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
48            {
49                g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
50                doNothing(PAUSE);
51            }
52      }

53      public void startThread()
54      {
55          Thread theThread = new Thread(this);
56          theThread.start();
57      }

58      public void doNothing(int milliseconds)
59      {
60          try
61          {
62              Thread.sleep(milliseconds);
63          }
64          catch (InterruptedException e)
65          {
66              System.out.println("Unexpected interrupt");
67              System.exit(0);
68          }
69      }
70  }
```

MyProgrammingLab™

Self-Test Exercises

1. Because `sleep` is a static method, how can it possibly know what thread it needs to pause?
2. Where was polymorphism used in the program in Display 19.2? (*Hint*: We are looking for an answer involving the class `Packer`.)

Race Conditions and Thread Synchronization ★

race condition

When multiple threads change a shared variable, it is sometimes possible that the variable will end up with the wrong (and often unpredictable) value. This is called a **race condition** because the final value depends on the sequence in which the threads access the shared value.

For example, consider two threads where each thread runs the following code:

```
int local;
local = sharedVariable;
local++;
sharedVariable = local;
```

The intent is for each thread to increment `sharedVariable` by one so if there are two threads, then `sharedVariable` should be incremented by two. However, consider the case where `sharedVariable` is 0. The first thread runs and executes the first two statements, so its variable `local` is set to 0. Now there is a context switch to the second thread. The second thread executes all four statements, so its variable `local` is set to 0 and incremented, and `sharedVariable` is set to 1. Now we return to the first thread and it continues where it left off, which is the third statement. The variable `local` is 0 so it is incremented to 1 and then the value 1 is copied into `sharedVariable`. The end result after both threads are done is that `sharedVariable` has the value 1, and we lost the value written by thread two!

You might think that this problem could be avoided by replacing our code with a single statement such as

```
sharedVariable++;
```

Unfortunately, this will not solve our problem because the statement is not guaranteed to be an "atomic" action and there could still be a context switch to another thread "in the middle" of executing the statement.

To demonstrate this problem, consider the `Counter` class shown in Display 19.4. This simple class merely stores a variable that increments a counter. It uses the somewhat roundabout way to increment the counter on purpose to increase the likelihood of a race condition.

The way we will demonstrate the race condition is to do the following:

1. Create a single instance of the `Counter` class.
2. Create an array of many threads (30,000 in the example) where each thread references the single instance of the `Counter` class.
3. Each thread runs and invokes the `increment()` method.
4. Wait for each thread to finish and then output the value of the counter. If there are no race conditions, then its value should be 30,000. If there are race conditions, then the value will be less than 30,000.

We create many threads to increase the likelihood that the race condition occurs. With only a few threads, it is not likely that there will be a switch to another thread inside the `increment()` method at the right point to cause a problem.

Display 19.4 The `Counter` Class

```
1  public class Counter
2  {
3    private int counter;
4    public Counter()
5    {
6           counter = 0;
7    }
8    public int value()
9    {
10          return counter;
11   }
12   public void increment()
13   {
14          int local;
15          local = counter;
16          local++;
17          counter = local;
18   }
19 }
```

The only new tool that we need for our demonstration program is a way to wait for all the threads to finish. If we do not wait, then our program might output the counter before all the threads have had a chance to increment the value. We can wait by invoking the `join()` method for every thread we create. This method waits for the thread to complete. The `join()` method throws `InterruptedException`. This is a checked exception so we must use the try/catch mechanism.

VideoNote Walkthrough of a Program with Race Conditions

The class `RaceConditionTest` in Display 19.5 illustrates the race condition. You may have to run the program several times before you get a value less than 30,000. Problems as a result of race conditions are often rare occurrences. This makes them extremely hard to find and debug!

Display 19.5 The `RaceConditionTest` Class (part 1 of 2)

```
1  public class RaceConditionTest extends Thread
2  {
3    private Counter countObject;
4    public RaceConditionTest(Counter ctr)
5    {
6         countObject = ctr;
7    }
```

Stores a reference to a single `Counter` object.

Display 19.5 The `RaceConditionTest` Class (part 2 of 2)

```
 8     public void run()
 9     {
10       countObject.increment();
11     }

12     public static void main(String[] args)
13     {
14       int i;
15       Counter masterCounter = new Counter();
16       RaceConditionTest[] threads = new RaceConditionTest[30000];

17       System.out.println("The counter is " + masterCounter.value());
18       for (i = 0; i < threads.length; i++)
19       {
20             threads[i] = new RaceConditionTest(masterCounter);
21             threads[i].start();
22       }

23        // Wait for the threads to finish
24        for (i = 0; i < threads.length; i++)
25        {
26             try
27             {
28               threads[i].join();
29             }
30             catch (InterruptedException e)
31             {
32               System.out.println(e.getMessage());
33             }
34        }
35        System.out.println("The counter is " + masterCounter.value());
37     }
38  }
```

Invokes the code in Display 19.4 where the race condition occurs.

The single instance of the `Counter` object.

Array of 30,000 threads.

Give each thread a reference to the single `Counter` object and start each thread.

Waits for the thread to complete.

Sample Dialogue (output will vary)

```
The counter is 0
The counter is 29998
```

So how do we fix this problem? The solution is to make each thread wait so only one thread can run the code in `increment()` at a time. This section of code is called a **critical region**. Java allows you to add the keyword **synchronized** around a critical region to enforce the requirement that only one thread is allowed to execute in this region at a time. All other threads will wait until the thread inside the region is finished.

critical region

synchronized

In this particular case, we can add the keyword `synchronized` to either the method or around the specific code. If we add `synchronized` to the `increment()` method in the `Counter` class, then it looks like this:

```
public synchronized void increment()
{
  int local;
  local = counter;
  local++;
  counter = local;
}
```

If we add `synchronized` inside the code, then we can write

```
public void increment()
{
  int local;
  synchronized (this)
  {
   local = counter;
   local++;
   counter = local;
  }
}
```

Either version will result in a counter whose final value is always 30,000. There are many other issues involved in thread management, concurrency, and synchronization. These concepts are often covered in more detail in an operating systems or parallel programming course.

MyProgrammingLab™

Self-Test Exercises

3. In the `run()` method of Display 19.5, make the thread sleep a random amount of time between one and five milliseconds. You should see an increase in the number of problems caused by race conditions. Can you explain why?

4. Here is some code that synchronizes thread access to a shared variable. How come it is not guaranteed to output 30,000 every time it is run?

```
public class Counter
{
  private int counter;
  public Counter()
  {
       counter = 0;
  }
```

MyProgrammingLab™

Self-Test Exercises (continued)

```
    public int value()
    {
      return counter;
    }
    public synchronized void increment()
    {
      counter++;
    }
}
public class RaceConditionTest extends Thread
{
  private Counter countObject;
  public RaceConditionTest(Counter ctr)
  {
      countObject = ctr;
  }
  public void run()
  {
    countObject.increment();
  }
  public static void main(String[] args)
  {
    int i;
    Counter masterCounter = new Counter();
    RaceConditionTest[] threads = new RaceConditionTest[30000];
    System.out.println("The counter is " + masterCounter.
    value());
    for (i = 0; i < threads.length; i++)
    {
        threads[i] = new RaceConditionTest(masterCounter);
        threads[i].start();
    }
    System.out.println("The counter is " + masterCounter.
    value());
  }
}
```

19.2 Networking with Stream Sockets

Since in order to speak, one must first listen, learn to speak by listening.

MEVLANA RUMI

Transmission Control Protocol (TCP)

When computers want to communicate with each other over a network, each computer must speak the same "language." In other words, the computers need to communicate using the same *protocol*. One of the most common protocols today is **TCP**, or the **Transmission Control Protocol**. For example, the HTTP protocol used to transmit Web pages is based on TCP. TCP is a stream-based protocol in which a stream of data is transmitted from the sender to the receiver. TCP is considered a reliable protocol because it guarantees that data from the sender is received in the same order in which it was sent. An analogy to TCP is the telephone system. A connection is made when the phone is dialed and the participants communicate by speaking back and forth. In TCP, the receiver must first be listening for a connection, the sender initiates the connection, and then the sender and receiver can transmit data. The program that is waiting for a connection is called the **server** and the program that initiates the connection is called the **client**.

server

client

User Datagram Protocol (UDP)

An alternate protocol is **UDP**, or the **User Datagram Protocol**. In UDP, packets of data are transmitted but no guarantee is made regarding the order in which the packets are received. An analogy to UDP is the postal system. Letters that are sent might be received in an unpredictable order, or lost entirely with no notification. Although Java provides support for UDP, we will only introduce TCP in this section.

Sockets

sockets

port

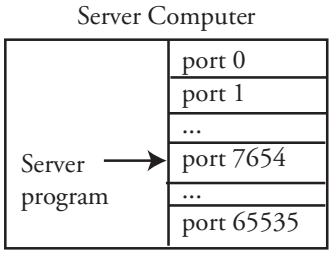
Network programming is implemented in Java using **sockets**. A socket describes one end of the connection between two programs over the network. A socket consists of an address that identifies the remote computer and a **port** for both the local and remote computer. The port is assigned an integer value between 0 and 65,535 that is used to identify which program should handle data received from the network. Two applications may not bind to the same port. Typically, ports 0 to 1,024 are reserved for use by well-known services implemented by your operating system.

The process of client/server communication is shown in Display 19.6. First, the server waits for a connection by listening on a specific port. When a client connects to this port, a new socket is created that identifies the remote computer, the remote port, and the local port. A similar socket is created on the client. Once the sockets are created on both the client and the server, data can be transmitted using streams in a manner very similar to the way we implemented file I/O in Chapter 10.

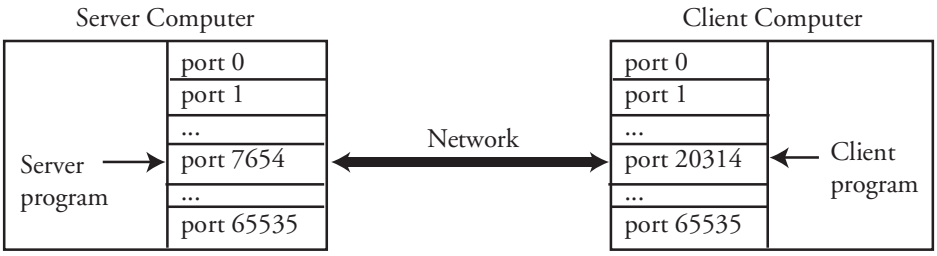
Display 19.7 shows how to create a simple server that listens on port 7654 for a connection. Once it receives a connection, a new socket is returned by the `accept( )` method. From this socket, we create a `BufferedReader`, just as if we were reading from a text file described in Chapter 10. Data is transmitted to the socket using a `DataOutputStream`, which is similar to a `FileOutputStream`. The `ServerSocket`

Display 19.6 Client/Server Network Communication through Sockets

1. The server listens and waits for a connection on port 7654.



2. The client connects to the server on port 7654. It uses a local port that is assigned automatically, in this case, port 20314.



The server program can now communicate over a socket bound locally to port 7654 and remotely to the client's address at port 20314.

The client program can now communicate over a socket bound locally to port 20314 and remotely to the server's address at port 7654.

and `Socket` classes are in the `java.net` package, while the `BufferedReader` and `DataOutputStream` classes are in the `java.io` package. Once the streams are created, the server expects the client to send a name. The server waits for the name with a call to `readLine( )` on the `BufferedReader` object and then sends back the name concatenated with the current date and time. Finally, the server closes the streams and sockets.

Display 19.6 shows how to create a client that connects to our date and time server. First, we create a socket with the name of the computer running the server along with the corresponding port of 7654. If the server program and client program are running on the same computer, then you can use **`localhost`** as the name of the machine. Your computer understands that any attempt to connect across a network to the machine named `localhost` really corresponds to a connection with itself. Otherwise, the hostname should be set to the name of the computer (e.g., my.server.com). After a connection is made, the client creates stream objects, sends its name, waits for a reply, and prints the reply.

`localhost`

Display 19.7 **Date and Time Server** (part 1 of 2)

```
1  import java.util.Date;
2  import java.net.ServerSocket;
3  import java.net.Socket;
4  import java.io.DataOutputStream;
5  import java.io.BufferedReader;
6  import java.io.InputStreamReader;
7  import java.io.IOException;

8  public class DateServer
9  {
10  public static void main(String[] args)
11  {
12       Date now = new Date();

13       try
14       {
15         System.out.println("Waiting for a connection on port 7654.");
16         ServerSocket serverSock = new ServerSocket(7654);
17         Socket connectionSock = serverSock.accept();

18         BufferedReader clientInput = new BufferedReader(
19             new InputStreamReader(connectionSock.getInputStream()));
20         DataOutputStream clientOutput = new DataOutputStream(
21             connectionSock.getOutputStream());

22         System.out.println("Connection made, waiting for client " +
23             "to send their name.");
24         String clientText = clientInput.readLine();
25         String replyText = "Welcome, " + clientText +
26             ", Today is " + now.toString() + "\n";
27         clientOutput.writeBytes(replyText);
28         System.out.println("Sent: " + replyText);

29         clientOutput.close();
30         clientInput.close();
31         connectionSock.close();
32         serverSock.close();
33       }
34       catch (IOException e)
35       {
           System.out.println(e.getMessage());
36       }
37  }
38 }
```

Display 19.7 **Date and Time Server** (part 2 of 2)

Sample Dialogue *Output when the client program in Display 19.8 connects to the server program.*

```
Waiting for a connection on port 7654.
Connection made, waiting for client to send their name.
Sent: Welcome, Dusty Rhodes, Today is Sun Nov 20 12:18:21 AKDT 2011
```

Display 19.8 **Date and Time Client** (part 1 of 2)

```
1  import java.net.Socket;
2  import java.io.DataOutputStream;
3  import java.io.BufferedReader;
4  import java.io.InputStreamReader;
5  import java.io.IOException;

6  public class DateClient
7  {
8    public static void main(String[] args)
9    {
10       try
11       {
12         String hostname = "localhost";
13         int port = 7654;

14         System.out.println("Connecting to server on port " + port);
15         Socket connectionSock = new Socket(hostname, port);

16         BufferedReader serverInput = new BufferedReader(
17             new InputStreamReader(connectionSock.getInputStream()));
18         DataOutputStream serverOutput = new DataOutputStream(
19             connectionSock.getOutputStream());

20         System.out.println("Connection made, sending name.");
21         serverOutput.writeBytes("Dusty Rhodes\n");

22         System.out.println("Waiting for reply.");
23         String serverData = serverInput.readLine();
24         System.out.println("Received: " + serverData);

25         serverOutput.close();
26         serverInput.close();
27         connectionSock.close();
28       }
```

`localhost` *refers to the same, or local, machine that the client is running on. Change this string to the appropriate hostname (e.g., my.server.com) if the server is running on a remote machine.*

(continued)

Display 19.8 **Date and Time Client** (part 2 of 2)

```
29          catch (IOException e)
30          {
31            System.out.println(e.getMessage());
32          }
33      }
34  }
```

Sample Dialogue *Output when client program connects to the server program in Display 19.7.*

```
Connecting to server on port 7654
Connection made, sending name.
Waiting for reply.
Received: Welcome, Dusty Rhodes, Today is Fri Oct 13 03:03:21 AKDT 2011
```

Note that the socket and stream objects throw checked exceptions. This means that their exceptions must be caught or declared in a `throws` block.

Sockets and Threading

blocking

If you run the program in Display 19.7, then you will notice that the server waits, or **blocks**, at the `serverSock.accept( )` call until a client connects to it. Both the client and server also block at the `readLine( )` call if data from the socket is not yet available. In a client with a GUI, you would notice this as a nonresponsive program while it is waiting for data. For the server, this behavior makes it difficult to handle connections with more than one client. After a connection is made with the first client, the server will become nonresponsive to the client's requests while it waits for a second client.

The solution to this problem is to use threads. One thread will listen for new connections while another thread handles an existing connection. Section 19.1 describes how to create threads and make a GUI program responsive. On the server, the `accept( )` call is typically placed in a loop and a new thread is created to handle each client connection:

```
while (true)
{
  Socket connectionSock = serverSock.accept( );
  ClientHandler handler = new ClientHandler(connectionSock);
  Thread theThread = new Thread(handler);
  theThread.start( );
}
```

In this code, `ClientHandler` is a class that implements `Runnable`. The constructor keeps a reference to the socket in an instance variable, and the `run( )` method would handle all communications. A complete implementation of a threaded server is left as Programming Projects 19.7 and 19.8.