Table of Contents

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 5. Threads

Back in the good old days of the Net, circa the early 1990s, we didn't have the Web and HTTP and graphical browsers. Instead, we had Usenet news and FTP and command-line interfaces, and we liked it that way! But as good as the good old days were, there were some problems. For instance, when we were downloading kilobytes of free software from a popular FTP site over our 2,400 bps modems using Kermit, we would often encounter error messages like this one:


```
% ftp eunl.java.sun.com
Connected to eunl.javasoft.com.
220 softwarenl FTP server (wu-2.4.2-academ[BETA- 16]+opie-2.32(1) 981105)
    ready.
Name (eunl.java.sun.com:elharo): anonymous
530-
530-    Server is busy.  Please try again later or try one of our other
530-    ftp servers at ftp.java.sun.com.  Thank you.
530-
530 User anonymous access denied.
Login failed.
```

In fact, in the days when the Internet had only a few million users instead of a few hundred million, we were far more likely to come across an overloaded and congested site than we are today. The problem was that both the FTP servers bundled with most Unixes and the third-party FTP servers, such as *wu-ftpd*, forked a new process for each connection. 100 simultaneous users meant 100 additional processes to handle. Since processes are fairly heavyweight items, too many could rapidly bring a server to its knees. The problem wasn't that the machines weren't powerful enough or the network fast enough; it was that the FTP servers were (and many still are) poorly implemented. Many more simultaneous users could be served if a new process wasn't needed for each connection.

Early web servers suffered from this problem as well, although the problem was masked a little by the transitory nature of HTTP connections. Since web pages and their embedded images tend to be small (at least compared to the software archives commonly retrieved by FTP) and since web browsers "hang up" the connection after each file is retrieved instead of staying connected for minutes or hours at a time, web users don't put nearly as much load

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

on a server as FTP users do. However, web server performance still degrades as usage grows. The fundamental problem is that while it's easy to write code that handles each incoming connection and each new task as a separate process (at least on Unix), this solution doesn't scale. By the time a server is attempting to handle a thousand or more simultaneous connections, performance slows to a crawl.

There are at least two solutions to this problem. The first is to reuse processes rather than spawning new ones. When the server starts up, a fixed number of processes (say, 300) are spawned to handle requests. Incoming requests are placed in a queue. Each process removes one request from the queue, services the request, then returns to the queue to get the next request. There are still 300 separate processes running, but because all the overhead of building up and tearing down the processes is avoided, these 300 processes can now do the work of 1,000. These numbers are rough estimates. Your exact mileage may vary, especially if your server hasn't yet reached the volume where scalability issues come into play. Still, whatever mileage you get out of spawning new processes, you should be able to do much better by reusing old processes.

The second solution to this problem is to use lightweight threads to handle connections instead of heavyweight processes. Whereas each separate process has its own block of memory, threads are easier on resources because they share memory. Using threads instead of processes can buy you another factor of three in server performance. By combining this with a pool of reusable threads (as opposed to a pool of reusable processes), your server can run nine times faster, all on the same hardware and network connection! While it's still the case that most Java virtual machines keel over somewhere between 700 and 2,000 simultaneous threads, the impact of running many different threads on the server hardware is relatively minimal since they all run within one process. Furthermore, by using a thread pool instead of spawning new threads for each connection, a server can use fewer than a hundred threads to handle thousands of connections per minute.

Unfortunately, this increased performance doesn't come for free. There's a cost in program complexity. In particular, multithreaded servers (and other multithreaded programs) require programmers to address concerns that aren't issues for single-threaded programs, particularly issues of safety and liveness. Because different threads share the same memory, it's entirely possible for one thread to stomp all over the variables and data structures used by another thread. This is similar to the way one program running on a non-memory-protected operating system such as Mac OS 9 or Windows 95 can crash the entire system. Consequently, different threads have to be extremely careful about which resources they use when. Generally, each thread must agree to use certain resources only when it's sure those resources can't change or that it has exclusive access to them. However, it's also possible for two threads to be too careful, each waiting for exclusive access to resources it will never get. This can lead to deadlock, in which two threads are each waiting for resources

the other possesses. Neither thread can proceed without the resources that the other thread has reserved, but neither is willing to give up the resources it has already.

There is a third solution to the problem, which in many cases is the most efficient of all, although it's only available in Java 1.4 and later. Selectors enable one thread to query a group of sockets to find out which ones are ready to be read from or written to, and then process the ready sockets sequentially. In this case, the I/O has to be designed around channels and buffers rather than streams. We'll discuss this in Chapter 12, which demonstrates selector-based solutions to the problems solved in this chapter with threads.

5.1. Running Threads

A *thread* with a little *t* is a separate, independent path of execution in the virtual machine. A `Thread` with a capital *T* is an instance of the `java.lang.Thread` class. There is a one-to-one relationship between threads executing in the virtual machine and `Thread` objects constructed by the virtual machine. Most of the time it's obvious from the context which one is meant if the difference is really important. To start a new thread running in the virtual machine, you construct an instance of the `Thread` class and invoke its `start()` method, like this:

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

Of course, this thread isn't very interesting because it doesn't have anything to do. To give a thread something to do, you either subclass the `Thread` class and override its `run()` method, or implement the `Runnable` interface and pass the `Runnable` object to the `Thread` constructor. I generally prefer the second option since it separates the task that the thread performs from the thread itself more cleanly, but you will see both techniques used in this book and elsewhere. In both cases, the key is the `run()` method, which has this signature:

```
public void run( )
```

You're going to put all the work the thread does in this one method. This method may invoke other methods; it may construct other objects; it may even spawn other threads. However,

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

the thread starts here and it stops here. When the `run()` method completes, the thread dies. In essence, the `run( )` method is to a thread what the `main( )` method is to a traditional nonthreaded program. A single-threaded program exits when the `main()` method returns. A multithreaded program exits when both the `main( )` method and the `run()` methods of all nondaemon threads return. (Daemon threads perform background tasks such as garbage collection and don't prevent the virtual machine from exiting.)

5.1.1. Subclassing Thread

For example, suppose you want to write a program that calculates the Secure Hash Algorithm (SHA) digest for many files. To a large extent, this program is I/O-bound; that is, its speed is limited by the amount of time it takes to read the files from the disk. If you write it as a standard program that processes the files in series, the program's going to spend a lot of time waiting for the hard drive to return the data. This is characteristic of a lot of network programs: they have a tendency to execute faster than the network can supply input. Consequently, they spend a lot of time blocked. This is time that other threads could use, either to process other input sources or to do something that doesn't rely on slow input. (Not all threaded programs share this characteristic. Sometimes, even if none of the threads have a lot of spare time to allot to other threads, it's simply easier to design a program by breaking it into multiple threads that perform independent operations.) Example 5-1 is a subclass of `Thread` whose `run( )` method calculates an SHA message digest for a specified file.

Example 5-1. FileDigestThread

```
import java.io.*;
import java.security.*;

public class DigestThread extends Thread {

  private File input;

  public DigestThread(File input) {
   this.input = input;
  }

  public void run( ) {
    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read( )) != -1) ;
      din.close( );
      byte[] digest = sha.digest( );
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
            StringBuffer result = new StringBuffer(input.toString( ));
            result.append(": ");
            for (int i = 0; i < digest.length; i++) {
              result.append(digest[i] + " ");
            }
            System.out.println(result);
          }
          catch (IOException ex) {
            System.err.println(ex);
          }
          catch (NoSuchAlgorithmException ex) {
            System.err.println(ex);
          }

        }

        public static void main(String[] args) {

          for (int i = 0; i < args.length; i++) {
            File f = new File(args[i]);
            Thread t = new DigestThread(f);
            t.start( );
          }

        }

      }
```

The `main( )` method reads filenames from the command-line and starts a new `DigestThread` for each one. The work of the thread is actually performed in the `run( )` method. Here, a `DigestInputStream` reads the file. Then the resulting digest is printed on `System.out`. Notice that the entire output from this thread is first built in a local `StringBuffer` variable `result`. This is then printed on the console with one method invocation. The more obvious path of printing the pieces one at a time using `System.out.print( )` is not taken. There's a reason for that, which we'll discuss soon.

Since the signature of the `run( )` method is fixed, you can't pass arguments to it or return values from it. Consequently, you need different ways to pass information into the thread and get information out of it. The simplest way to pass information in is to pass arguments to the constructor, which set fields in the `Thread` subclass, as done here.

Getting information out of a thread back into the original calling thread is trickier because of the asynchronous nature of threads. Example 5-1 sidesteps that problem by never passing any information back to the calling thread and simply printing the results on `System.out`. Most of the time, however, you'll want to pass the information to other parts of the program. You can store the result of the calculation in a field and provide a getter method to return the value of that field. However, how do you know when the calculation of that value is complete? What do you return if somebody calls the getter method before the value has been calculated? This is quite tricky, and we'll discuss it more later in this chapter.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

If you subclass `Thread`, you should override `run()` *and nothing else!* The various other methods of the `Thread` class, `start()`, `stop( )`, `interrupt( )`, `join( )`, `sleep ( )`, and so on, all have very specific semantics and interactions with the virtual machine that are difficult to reproduce in your own code. You should override `run( )` and provide additional constructors and other methods as necessary, but you should not replace any of the other standard `Thread` methods.

5.1.2. Implementing the Runnable Interface

One way to avoid overriding the standard `Thread` methods is not to subclass `Thread`. Instead, write the task you want the thread to perform as an instance of the `Runnable` interface. This interface declares the `run( )` method, exactly the same as the `Thread` class:

```
public void run( )
```

Other than this method, which any class implementing this interface must provide, you are completely free to create any other methods with any other names you choose, all without any possibility of unintentionally interfering with the behavior of the thread. This also allows you to place the thread's task in a subclass of some other class, such as `Applet` or `HTTPServlet`. To start a thread that performs the `Runnable`'s task, pass the `Runnable` object to the `Thread` constructor. For example:

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

It's easy to recast most problems that subclass `Thread` into `Runnable` forms. Example 5-2 demonstrates by rewriting Example 5-1 to use the `Runnable` interface rather than subclassing `Thread`. Aside from the name change, the only modifications that are necessary are changing `extends Thread` to `implements Runnable` and passing a `DigestRunnable` object to the `Thread` constructor in the `main()` method. The essential logic of the program is unchanged.

Example 5-2. DigestRunnable

```
import java.io.*;
import java.security.*;

public class DigestRunnable implements Runnable {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    private File input;

    public DigestRunnable(File input) {
     this.input = input;
    }

    public void run( ) {
      try {
        FileInputStream in = new FileInputStream(input);
        MessageDigest sha = MessageDigest.getInstance("SHA");
        DigestInputStream din = new DigestInputStream(in, sha);
        int b;
        while ((b = din.read( )) != -1) ;
        din.close( );
        byte[] digest = sha.digest( );
        StringBuffer result = new StringBuffer(input.toString( ));
        result.append(": ");
        for (int i = 0; i < digest.length; i++) {
          result.append(digest[i] + " ");
        }
        System.out.println(result);
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
      catch (NoSuchAlgorithmException ex) {
        System.err.println(ex);
      }

    }

    public static void main(String[] args) {

      for (int i = 0; i < args.length; i++) {
        File f = new File(args[i]);
        DigestRunnable dr = new DigestRunnable(f);
        Thread t = new Thread(dr);
        t.start( );
      }

    }

  }
```

There's no strong reason to prefer implementing `Runnable` to extending `Thread` or vice versa in the general case. In a few special cases, such as Example 5-14 later in this chapter, it may be useful to invoke some instance methods of the `Thread` class from within the constructor for each `Thread` object. This requires using a subclass. In some specific cases, it may be necessary to place the `run( )` method in a class that extends another class, such as `Applet`, in which case the `Runnable` interface is essential. Finally, some object-oriented purists argue that the task that a thread undertakes is not really a kind of `Thread`, and therefore should be placed in a separate class or interface such as `Runnable` rather than in a subclass of `Thread`. I half agree with them, although I don't think the argument is as strong

as it's sometimes made out to be. Consequently, I'll mostly use the `Runnable` interface in this book, but you should feel free to do whatever seems most convenient.

5.2. Returning Information from a Thread

One of the hardest things for programmers accustomed to traditional, single- threaded procedural models to grasp when moving to a multithreaded environment is how to return information from a thread. Getting information out of a finished thread is one of the most commonly misunderstood aspects of multithreaded programming. The `run( )` method and the `start( )` method don't return any values. For example, suppose that instead of simply printing out the SHA digest, as in Example 5-1 and Example 5-2, the digest thread needs to return the digest to the main thread of execution. Most people's first reaction is to store the result in a field and provide a getter method, as shown in Example 5-3 and Example 5-4. Example 5-3 is a `Thread` subclass that calculates a digest for a specified file. Example 5-4 is a simple command-line user interface that receives filenames and spawns threads to calculate digests for them.

Example 5-3. A thread that uses an accessor method to return the result

```
import java.io.*;
import java.security.*;

public class ReturnDigest extends Thread {

  private File input;
  private byte[] digest;

  public ReturnDigest(File input) {
   this.input = input;
  }

  public void run( ) {
    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read( )) != -1) ;
      din.close( );
      digest = sha.digest( );
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
```

```
        catch (NoSuchAlgorithmException ex) {
          System.err.println(ex);
        }

      }

      public byte[] getDigest( ) {
        return digest;
      }

    }
```

Example 5-4. A main program that uses the accessor method to get the output of the thread

```
import java.io.*;

public class ReturnDigestUserInterface {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {

      // Calculate the digest
      File f = new File(args[i]);
      ReturnDigest dr = new ReturnDigest(f);
      dr.start( );

      // Now print the result
      StringBuffer result = new StringBuffer(f.toString( ));
      result.append(": ");
      byte[] digest = dr.getDigest( );
      for (int j = 0; j < digest.length; j++) {
        result.append(digest[j] + " ");
      }
      System.out.println(result);

    }

  }

}
```

The `ReturnDigest` class stores the result of the calculation in the private field digest, which is accessed via `getDigest( )`. The `main( )` method in `ReturnDigestUserInterface` loops through a list of files from the command line. It starts a new `ReturnDigest` thread for each file and then tries to retrieve the result using `getDigest( )`. However, when you run this program, the result may not be what you expect:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
D:\JAVA\JNP3\examples\05>java ReturnDigestUserInterface *.java
Exception in thread "main" java.lang.NullPointerException
      at ReturnDigestUserInterface.main(ReturnDigestUserInterface.java,
      Compiled Code)
```

The problem is that the main program gets the digest and uses it before the thread has had a chance to initialize it. Although this flow of control would work in a single-threaded program in which `dr.start( )` simply invoked the `run()` method in the same thread, that's not what happens here. The calculations that `dr.start()` kicks off may or may not finish before the `main( )` method reaches the call to `dr.getDigest()`. If they haven't finished, `dr.getDigest( )` returns `null`, and the first attempt to access `digest` throws a `NullPointerException`.

5.2.1. Race Conditions

One possibility is to move the call to `dr.getDigest( )` later in the `main()` method, like this:

```
public static void main(String[] args) {

  ReturnDigest[] digests = new ReturnDigest[args.length];

  for (int i = 0; i < args.length; i++) {

    // Calculate the digest
    File f = new File(args[i]);
    digests[i] = new ReturnDigest(f);
    digests[i].start( );

  }

  for (int i = 0; i < args.length; i++) {

    // Now print the result
    StringBuffer result = new StringBuffer(args[i]);
    result.append(": ");
    byte[] digest = digests[i].getDigest( );
    for (int j = 0; j < digest.length; j++) {
      result.append(digest[j] + " ");
    }
    System.out.println(result);

  }

}
```

If you're lucky, this will work and you'll get the expected output, like this:

```
D:\JAVA\JNP3\examples\05>java ReturnDigest2 *.java
BadDigestRunnable.java: 73 -77 -74 111 -75 -14 70 13 -27 -28 32 68 -126
```

```
43 -27 55 -119 26 -77 6
BadDigestThread.java: 69 101 80 -94 -98 -113 29 -52 -124 -121 -38 -82 39
-4 8 -38 119 96 -37 -99
DigestRunnable.java: 61 116 -102 -120 97 90 53 37 -14 111 -60 -86 -112
124 -54 111 114 -42 -36 -111
DigestThread.java: 69 101 80 -94 -98 -113 29 -52 -124 -121 -38 -82 39
-4 8 -38 119 96 -37 -99
```

But let me emphasize that point about being lucky. You may not get this output. In fact, you may still get a `NullPointerException`. Whether this code works is completely dependent on whether every one of the `ReturnDigest` threads finishes before its `getDigest( )` method is called. If the first `for` loop is too fast and the second `for` loop is entered before the threads spawned by the first loop start finishing, we're back where we started:

```
D:\JAVA\JNP3\examples\05>java ReturnDigest2 ReturnDigest.java
Exception in thread "main" java.lang.NullPointerException
        at ReturnDigest2.main(ReturnDigest2.java, Compiled Code)
```

Whether you get the correct results or this exception depends on many factors, including how many threads the program spawns, the relative speeds of the CPU and disk on the system where this is run, and the algorithm the Java virtual machine uses to allot time to different threads. This is called a *race condition*. Getting the correct result depends on the relative speeds of different threads, and you can't control those! We need a better way to guarantee that the `getDigest()` method isn't called until the digest is ready.

5.2.2. Polling

The solution most novices adopt is to make the getter method return a flag value (or perhaps throw an exception) until the result field is set. Then the main thread periodically polls the getter method to see whether it's returning something other than the flag value. In this example, that would mean repeatedly testing whether the digest is null and using it only if it isn't. For example:

```
public static void main(String[] args) {

  ReturnDigest[] digests = new ReturnDigest[args.length];

  for (int i = 0; i < args.length; i++) {

    // Calculate the digest
    File f = new File(args[i]);
    digests[i] = new ReturnDigest(f);
    digests[i].start( );

  }
```

```
    for (int i = 0; i < args.length; i++) {
      while (true) {
        // Now print the result
        byte[] digest = digests[i].getDigest( );
        if (digest != null) {
          StringBuffer result = new StringBuffer(args[i]);
          result.append(": ");
          for (int j = 0; j < digest.length; j++) {
            result.append(digest[j] + " ");
          }
          System.out.println(result);
          break;
        }
      }
    }

  }
```

This solution works. It gives the correct answers in the correct order and it works irrespective of how fast the individual threads run relative to each other. However, it's doing a lot more work than it needs to.

5.2.3. Callbacks

In fact, there's a much simpler, more efficient way to handle the problem. The infinite loop that repeatedly polls each `ReturnDigest` object to see whether it's finished can be eliminated. The trick is that rather than having the main program repeatedly ask each `ReturnDigest` thread whether it's finished (like a five-year-old repeatedly asking, "Are we there yet?" on a long car trip, and almost as annoying), we let the thread tell the main program when it's finished. It does this by invoking a method in the main class that started it. This is called a *callback* because the thread calls its creator back when it's done. This way, the main program can go to sleep while waiting for the threads to finish and not steal time from the running threads.

When the thread's `run( )` method is nearly done, the last thing it does is invoke a known method in the main program with the result. Rather than the main program asking each thread for the answer, each thread tells the main program the answer. For instance, Example 5-5 shows a `CallbackDigest` class that is much the same as before. However, at the end of the `run( )` method, it passes off the `digest` to the static `CallbackDigestUserInterface.receiveDigest( )` method in the class that originally started the thread.

Example 5-5. CallbackDigest

```
import java.io.*;
import java.security.*;

public class CallbackDigest implements Runnable {

  private File input;

  public CallbackDigest(File input) {
   this.input = input;
  }

  public void run( ) {

    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read( )) != -1) ;
      din.close( );
      byte[] digest = sha.digest( );
      CallbackDigestUserInterface.receiveDigest(digest,
       input.getName( ));
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
    catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }

  }

}
```

The `CallbackDigestUserInterface` class shown in Example 5-6 provides the `main( )` method. However, unlike the `main( )` methods in the other variations of this program in this chapter, this one only starts the threads for the files named on the command line. It does not attempt to actually read, print out, or in any other way work with the results of the calculation. Those functions are handled by a separate method, `receiveDigest( )`. `receiveDigest( )` is not invoked by the `main( )` method or by any method that can be reached by following the flow of control from the `main( )` method. Instead, it is invoked by each thread separately. In effect, `receiveDigest( )` runs inside the digesting threads rather than inside the main thread of execution.

Example 5-6. CallbackDigestUserInterface

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


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public class CallbackDigestUserInterface {

  public static void receiveDigest(byte[] digest, String name) {

    StringBuffer result = new StringBuffer(name);
    result.append(": ");
    for (int j = 0; j < digest.length; j++) {
      result.append(digest[j] + " ");
    }
    System.out.println(result);

  }

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      // Calculate the digest
      File f = new File(args[i]);
      CallbackDigest cb = new CallbackDigest(f);
      Thread t = new Thread(cb);
      t.start( );
    }

  }

}
```

Example 5-5 and Example 5-6 use static methods for the callback so that `CallbackDigest` only needs to know the name of the method in `CallbackDigestUserInterface` to call. However, it's not much harder (and it's considerably more common) to call back to an instance method. In this case, the class making the callback must have a reference to the object it's calling back. Generally, this reference is provided as an argument to the thread's constructor. When the `run()` method is nearly done, the last thing it does is invoke the instance method on the callback object to pass along the result. For instance, Example 5-7 shows a `CallbackDigest` class that is much the same as before. However, it now has one additional field, a `CallbackDigestUserInterface` object called `callback`. At the end of the `run( )` method, the `digest` is passed to `callback`'s `receiveDigest( )` method. The `CallbackDigestUserInterface` object itself is set in the constructor.

Example 5-7. InstanceCallbackDigest

```
import java.io.*;
import java.security.*;


public class InstanceCallbackDigest implements Runnable {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
  private File input;
  private InstanceCallbackDigestUserInterface callback;

  public InstanceCallbackDigest(File input,
   InstanceCallbackDigestUserInterface callback) {
    this.input = input;
    this.callback = callback;
  }

  public void run( ) {

    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read( )) != -1) ;
      din.close( );
      byte[] digest = sha.digest( );
      callback.receiveDigest(digest);
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
    catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }

  }

}
```

The `CallbackDigestUserInterface` class shown in Example 5-8 holds the `main( )` method as well as the `receiveDigest( )` method used to handle an incoming digest. Example 5-8 just prints out the digest, but a more expansive class could do other things as well, such as storing the digest in a field, using it to start another thread, or performing further calculations on it.

Example 5-8. InstanceCallbackDigestUserInterface

```
import java.io.*;

public class InstanceCallbackDigestUserInterface {

  private File input;
  private byte[] digest;

  public InstanceCallbackDigestUserInterface(File input) {
    this.input = input;
  }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    public void calculateDigest( ) {
      InstanceCallbackDigest cb = new InstanceCallbackDigest(input, this);
      Thread t = new Thread(cb);
      t.start( );
    }

    void receiveDigest(byte[] digest) {
      this.digest = digest;
      System.out.println(this);
    }

    public String toString( ) {
      String result = input.getName( ) + ": ";
      if (digest != null) {
        for (int i = 0; i < digest.length; i++) {
          result += digest[i] + " ";
        }
      }
      else {
        result += "digest not available";
      }
      return result;
    }

    public static void main(String[] args) {

      for (int i = 0; i < args.length; i++) {
        // Calculate the digest
        File f = new File(args[i]);
        InstanceCallbackDigestUserInterface d
         = new InstanceCallbackDigestUserInterface(f);
        d.calculateDigest( );
      }

    }

  }
```

Using instance methods instead of static methods for callbacks is a little more complicated but has a number of advantages. First, each instance of the main class (`InstanceCallbackDigestUserInterface`, in this example) maps to exactly one file and can keep track of information about that file in a natural way without needing extra data structures. Furthermore, the instance can easily recalculate the digest for a particular file, if necessary. In practice, this scheme proves a lot more flexible. However, there is one caveat. Notice the addition of the `calculateDigest( )` method to start the thread. You might logically think that this belongs in a constructor. However, starting threads in a constructor is dangerous, especially threads that will call back to the originating object. There's a race condition here that may allow the new thread to call back before the constructor is finished and the object is fully initialized. It's unlikely in this case, because starting the new thread is the last thing this constructor does. Nonetheless, it's at least theoretically possible. Therefore, it's good form to avoid launching threads from constructors.

The first advantage of the callback scheme over the polling scheme is that it doesn't waste so many CPU cycles. But a much more important advantage is that callbacks are more flexible and can handle more complicated situations involving many more threads, objects, and classes. For instance, if more than one object is interested in the result of the thread's calculation, the thread can keep a list of objects to call back. Particular objects can register their interest by invoking a method in the `Thread` or `Runnable` class to add themselves to the list. If instances of more than one class are interested in the result, a new `interface` can be defined that all these classes implement. The `interface` would declare the callback methods. If you're experiencing déjà vu right now, that's probably because you have seen this scheme before. This is *exactly* how events are handled in Swing, the AWT, and JavaBeans. The AWT runs in a separate thread from the rest of the program; components and beans inform you of events by calling back to methods declared in particular interfaces, such as `ActionListener` and `PropertyChangeListener`. Your listener objects register their interests in events fired by particular components using methods in the `Component` class, such as `addActionListener( )` and `addPropertyChangeListener( )`. Inside the component, the registered listeners are stored in a linked list built out of `java.awt.AWTEventMulticaster` objects. It's easy to duplicate this pattern in your own classes. Example 5-9 shows one very simple possible interface class called `DigestListener` that declares the `digestCalculated( )` method.

Example 5-9. DigestListener interface

```
public interface DigestListener {

  public void digestCalculated(byte[] digest);

}
```

Example 5-10 shows the `Runnable` class that calculates the digest. Several new methods and fields are added for registering and deregistering listeners. For convenience and simplicity, a `java.util.Vector` manages the list. The `run( )` method no longer directly calls back the object that created it. Instead, it communicates with the private `sendDigest( )` method, which sends the digest to all registered listeners. The `run( )` method neither knows nor cares who's listening to it. This class no longer knows anything about the user interface class. It has been completely decoupled from the classes that may invoke it. This is one of the strengths of this approach.

Example 5-10. The ListCallbackDigest class

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
import java.io.*;
import java.security.*;
import java.util.*;

public class ListCallbackDigest implements Runnable {

  private File input;
  List listenerList = new Vector( );

  public ListCallbackDigest(File input) {
   this.input = input;
  }

  public synchronized void addDigestListener(DigestListener l) {
    listenerList.add(l);
  }

  public synchronized void removeDigestListener(DigestListener l) {
    listenerList.remove(l);
  }

  private synchronized  void sendDigest(byte[] digest) {

    ListIterator iterator = listenerList.listIterator( );
    while (iterator.hasNext( )) {
      DigestListener dl = (DigestListener) iterator.next( );
      dl.digestCalculated(digest);
    }

  }

  public void run( ) {

    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read( )) != -1) ;
      din.close( );
      byte[] digest = sha.digest( );
      this.sendDigest(digest);
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
    catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }

  }

}
```

Finally, Example 5-11 is a main program that implements the `DigestListener` interface and exercises the `ListCallbackDigest` class by calculating digests for all the files named

on the command line. However, this is no longer the only possible main program. There are now many more possible ways the digest thread could be used.

Example 5-11. ListCallbackDigestUserInterface interface

```
import java.io.*;

public class ListCallbackDigestUserInterface implements DigestListener {

  private File input;
  private byte[] digest;

  public ListCallbackDigestUserInterface(File input) {
    this.input = input;
  }

  public void calculateDigest( ) {
    ListCallbackDigest cb = new ListCallbackDigest(input);
    cb.addDigestListener(this);
    Thread t = new Thread(cb);
    t.start( );
  }

  public void digestCalculated(byte[] digest) {
    this.digest = digest;
    System.out.println(this);
  }

  public String toString( ) {
    String result = input.getName( ) + ": ";
    if (digest != null) {
      for (int i = 0; i < digest.length; i++) {
        result += digest[i] + " ";
      }
    }
    else {
      result += "digest not available";
    }
    return result;
  }

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      // Calculate the digest
      File f = new File(args[i]);
      ListCallbackDigestUserInterface d
       = new ListCallbackDigestUserInterface(f);
      d.calculateDigest( );
    }

  }

}
```

5.3. Synchronization

My shelves are overflowing with books, including many duplicate books, out-of-date books, and books I haven't looked at for 10 years and probably never will again. Over the years, these books have cost me tens of thousands of dollars, maybe more, to acquire. By contrast, two blocks down the street from my apartment, you'll find the Central Brooklyn Public Library. Its shelves are also overflowing with books; and over its 150 years, it's spent millions on its collection. But the difference is that its books are shared among all the residents of Brooklyn, and consequently the books have very high turnover. Most books in the collection are used several times a year. Although the public library spends a lot more money buying and storing books than I do, the cost per page read is much lower at the library than for my personal shelves. That's the advantage of a shared resource.

Of course, there are disadvantages to shared resources, too. If I need a book from the library, I have to walk over there. I have to find the book I'm looking for on the shelves. I have to stand in line to check the book out, or else I have to use it right there in the library rather than bringing it home with me. Sometimes, somebody else has checked the book out, and I have to fill out a reservation slip requesting that the book be saved for me when it's returned. And I can't write notes in the margins, highlight paragraphs, or tear pages out to paste on my bulletin board. (Well, I can, but if I do, it significantly reduces the usefulness of the book for future borrowers; and if the library catches me, I may lose my borrowing privileges.) There's a significant time and convenience penalty associated with borrowing a book from the library rather than purchasing my own copy, but it does save me money and storage space.

A thread is like a borrower at a library; the thread borrows from a central pool of resources. Threads make programs more efficient by sharing memory, file handles, sockets, and other resources. As long as two threads don't want to use the same resource at the same time, a multithreaded program is much more efficient than the multiprocess alternative, in which each process has to keep its own copy of every resource. The downside of a multithreaded program is that if two threads want the same resource at the same time, one of them will have to wait for the other to finish. If one of them doesn't wait, the resource may get corrupted. Let's look at a specific example. Consider the `run( )` method of Example 5-1 and Example 5-2. As previously mentioned, the method builds the result as a `String`, and then prints the `String` on the console using one call to `System.out.println( )`. The output looks like this:

```
DigestThread.java: 69 101 80 -94 -98 -113 29 -52 -124 -121 -38 -82 39
-4 8 -38 119 96 -37 -99
DigestRunnable.java: 61 116 -102 -120 97 90 53 37 -14 111 -60 -86 -112
124 -54 111 114 -42 -36 -111
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
DigestThread.class: -62 -99 -39 -19 109 10 -91 25 -54 -128 -101 17 13
-66 119 25 -114 62 -21 121
DigestRunnable.class: 73 15 7 -122 96 66 -107 -45 69 -36 86 -43 103
-104 25 -128 -97 60 14 -76
```

Four threads run in parallel to produce this output. Each writes one line to the console. The order in which the lines are written is unpredictable because thread scheduling is unpredictable, but each line is written as a unified whole. Suppose, however, we used this variation of the `run( )` method, which, rather than storing intermediate parts of the result in the `String` variable `result`, simply prints them on the console as they become available:

```
public void run( ) {

  try {
    FileInputStream in = new FileInputStream(input);
    MessageDigest sha = MessageDigest.getInstance("SHA");
    DigestInputStream din = new DigestInputStream(in, sha);
    int b;
    while ((b = din.read( )) != -1) ;
    din.close( );
    byte[] digest = sha.digest( );
    System.out.print(input + ": ");
    for (int i = 0; i < digest.length; i++) {
     System.out.print(digest[i] + " ");
    }
    System.out.println( );
  }
  catch (IOException ex) {
    System.err.println(ex);
  }
  catch (NoSuchAlgorithmException ex) {
    System.err.println(ex);
  }

}
```

When you run the program on the same input, the output looks something like this:

```
DigestRunnable.class: 73 15 7 -122 96 66 -107 -45 69 -36 86 -43 103 -104 25
-128 DigestRunnable.java: DigestThread.class: DigestThread.java:
61 -62 69 116 -99 101 -102 -39 80 -120 -19 -94 97 109 -98 90 -97 10 -113 53
60 -91 29 37 14 25 -52 -14 -76 -54 -124 111
-128 -121 -60 -101 -38 -86 17 -82 -112 13 39 124 -66 -4 -54 119 8 111 25 -38
114 -114 119 -42 62 96 -36 -21 -37 -111 121 -99
```

The digests of the different files are all mixed up! There's no telling which number belongs to which digest. Clearly, this is a problem.

The reason this mix-up occurs is that `System.out` is shared between the four different threads. When one thread starts writing to the console through several `System.out.print()` statements, it may not finish all its writes before another thread breaks in and starts writing its output. The exact order in which one thread preempts the

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

other threads is indeterminate. You'll probably see slightly different output every time you run this program.

We need a way to assign exclusive access to a shared resource to one thread for a specific series of statements. In this example, that shared resource is `System.out`, and the statements that need exclusive access are:

```
System.out.print(input + ": ");
for (int i = 0; i < digest.length; i++) {
  System.out.print(digest[i] + " ");
}
System.out.println( );
```

5.3.1. Synchronized Blocks

Java's means of assigning exclusive access to an object is the `synchronized` keyword. To indicate that these five lines of code should be executed together, wrap them in a `synchronized` block that synchronizes on the `System.out` object, like this:

```
synchronized (System.out) {
  System.out.print(input + ": ");
  for (int i = 0; i < digest.length; i++) {
    System.out.print(digest[i] + " ");
  }
  System.out.println( );
}
```

Once one thread starts printing out the values, all other threads will have to stop and wait for it to finish before they can print out their values. Synchronization is only a partial lock on an object. Other methods can use the synchronized object if they do so blindly, without attempting to synchronize on the object. For instance, in this case, there's nothing to prevent an unrelated thread from printing on `System.out` if it doesn't also try to synchronize on `System.out`. Java provides no means to stop all other threads from using a shared resource. It can only prevent other threads that synchronize on the same object from using the shared resource.

In fact, the `PrintStream` class internally synchronizes most methods on the `PrintStream` object, `System.out` in this example. In other words, every other thread that calls `System.out.println( )` will be synchronized on `System.out` and will have to wait for this code to

finish. `PrintStream` is unique in this respect. Most other `OutputStream` subclasses do not synchronize themselves.

Synchronization must be considered any time multiple threads share resources. These threads may be instances of the same `Thread` subclass or use the same `Runnable` class, or they may be instances of completely different classes. The key is the resources they share, not what classes they are. In Java, all resources are represented by objects that are instances of particular classes. Synchronization becomes an issue only when two threads both possess references to the same object. In the previous example, the problem was that several threads had access to the same `PrintStream` object, `System.out`. In this case, it was a static class variable that led to the conflict. However, instance variables can also have problems.

For example, suppose your web server keeps a log file. The log file may be represented by a class like the one shown in Example 5-12. This class itself doesn't use multiple threads. However, if the web server uses multiple threads to handle incoming connections, then each of those threads will need access to the same log file and consequently to the same `LogFile` object.

Example 5-12. LogFile

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

public class LogFile {

  private Writer out;

  public LogFile(File f) throws IOException {
    FileWriter fw = new FileWriter(f);
    this.out = new BufferedWriter(fw);
  }

  public void writeEntry(String message) throws IOException {
    Date d = new Date( );
    out.write(d.toString( ));
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }

  public void close( ) throws IOException {
    out.flush( );
    out.close( );
  }
```

```
    protected void finalize( ) {
      try {
        this.close( );
      }
      catch (IOException ex) {
      }
    }

  }
```

In this class, the `writeEntry()` method finds the current date and time, then writes into the underlying file using four separate invocations of `out.write( )`. A problem occurs if two or more threads each have a reference to the same `LogFile` object and one of those threads interrupts another in the process of writing the data. One thread may write the date and a tab, then the next thread might write three complete entries; then, the first thread could write the message, a carriage return, and a linefeed. The solution, once again, is synchronization. However, here there are two good choices for which object to synchronize on. The first choice is to synchronize on the `Writer` object `out`. For example:

```
public void writeEntry(String message) throws IOException {

  synchronized (out) {
    Date d = new Date( );
    out.write(d.toString( ));
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }

}
```

This works because all the threads that use this `LogFile` object also use the same `out` object that's part of that `LogFile`. It doesn't matter that `out` is private. Although it is used by the other threads and objects, it's referenced only within the `LogFile` class. Furthermore, although we're synchronizing here on the `out` object, it's the `writeEntry( )` method that needs to be protected from interruption. The `Writer` classes all have their own internal synchronization, which protects one thread from interfering with a `write( )` method in another thread. (This is not true of input and output streams, with the exception of `PrintStream`. It is possible for a write to an output stream to be interrupted by another thread.) Each `Writer` class has a `lock` field that specifies the object on which writes to that writer synchronize.

The second possibility is to synchronize on the `LogFile` object itself. This is simple enough to arrange with the `this` keyword. For example:

```
public void writeEntry(String message) throws IOException {

  synchronized (this) {
    Date d = new Date( );
    out.write(d.toString( ));
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }

}
```

5.3.2. Synchronized Methods

Since synchronizing the entire method body on the object itself is such a common thing to do, Java provides a shortcut. You can synchronize an entire method on the current object (the `this` reference) by adding the `synchronized` modifier to the method declaration. For example:

```
public synchronized void writeEntry(String message)
 throws IOException {

  Date d = new Date( );
  out.write(d.toString( ));
  out.write('\t');
  out.write(message);
  out.write("\r\n");

}
```

Simply adding the `synchronized` modifier to all methods is not a catchall solution for synchronization problems. For one thing, it exacts a severe performance penalty in many VMs (though more recent VMs have improved greatly in this respect), potentially slowing down your code by a factor of three or more. Second, it dramatically increases the chances of deadlock. Third, and most importantly, it's not always the object itself you need to protect from simultaneous modification or access, and synchronizing on the instance of the method's class may not protect the object you really need to protect. For instance, in this example, what we're really trying to prevent is two threads simultaneously writing onto `out`. If some other class had a reference to `out` completely unrelated to the `LogFile`, this attempt would fail. However, in this example, synchronizing on the `LogFile` object is sufficient because `out` is a private instance variable. Since we never expose a reference to this object, there's no way for any other object to invoke its methods except through the `LogFile` class. Therefore, synchronizing on the `LogFile` object has the same effect as synchronizing on `out`.

5.3.3. Alternatives to Synchronization

Synchronization is not always the best solution to the problem of inconsistent behavior caused by thread scheduling. There are a number of techniques that avoid the need for synchronization entirely. The first is to use local variables instead of fields wherever possible. Local variables do not have synchronization problems. Every time a method is entered, the virtual machine creates a completely new set of local variables for the method. These variables are invisible from outside the method and are destroyed when the method exits. As a result, it's impossible for one local variable to be used in two different threads. Every thread has its own separate set of local variables.

Method arguments of primitive types are also safe from modification in separate threads because Java passes arguments by value rather than by reference. A corollary of this is that methods such as `Math.sqrt( )` that simply take zero or more primitive data type arguments, perform some calculation, and return a value without ever interacting with the fields of any class are inherently thread-safe. These methods often either are or should be declared static.

Method arguments of object types are a little trickier because the actual argument passed by value is a reference to the object. Suppose, for example, you pass a reference to an array into a `sort( )` method. While the method is sorting the array, there's nothing to stop some other thread that also has a reference to the array from changing the values in the array.

`String` arguments are safe because they're *immutable*; that is, once a `String` object has been created, it cannot be changed by any thread. An immutable object never changes state. The values of its fields are set once when the constructor runs and never altered thereafter. `StringBuffer` arguments are not safe because they're not immutable; they can be changed after they're created.

A constructor normally does not have to worry about issues of thread safety. Until the constructor returns, no thread has a reference to the object, so it's impossible for two threads to have a reference to the object. (The most likely issue is if a constructor depends on another object in another thread that may change while the constructor runs, but that's uncommon. There's also a potential problem if a constructor somehow passes a reference to the object it's creating into a different thread, but this is also uncommon.)

You can take advantage of immutability in your own classes. It's often the easiest way to make a class thread-safe, often much easier than determining exactly which methods or code blocks to synchronize. To make an object immutable, simply declare all its fields private and don't write any methods that can change them. A lot of classes in the core Java library are immutable, for instance, `java.lang.String`, `java.lang.Integer`,

`java.lang.Double`, and many more. This makes these classes less useful for some purposes, but it does make them a lot more thread-safe.

A third technique is to use a thread-unsafe class but only as a private field of a class that is thread-safe. As long as the containing class accesses the unsafe class only in a thread-safe fashion and as long as it never lets a reference to the private field leak out into another object, the class is safe. An example of this technique might be a web server that uses an unsynchronized `LogFile` class but gives each separate thread its own separate log so no resources are shared between the individual threads.

5.4. Deadlock

Synchronization can lead to another possible problem: *deadlock*. Deadlock occurs when two threads need exclusive access to the same set of resources and each thread holds the lock on a different subset of those resources. If neither thread is willing to give up the resources it has, both threads come to an indefinite halt. This isn't quite a hang in the classical sense because the program is still active and behaving normally from the perspective of the OS, but to a user the difference is insignificant.

To return to the library example, deadlock is what occurs when Jack and Jill are each writing a term paper on Thomas Jefferson and they both need the two books *Thomas Jefferson and Sally Hemings: An American Controversy* and *Sally Hemings and Thomas Jefferson: History, Memory and Civic Culture*. If Jill has checked out the first book and Jack has checked out the second, and neither is willing to give up the book they have, neither can finish the paper. Eventually the deadline expires and they both get an F. That's the problem of deadlock.

Worse yet, deadlock can be a sporadic, hard-to-detect bug. Deadlock usually depends on unpredictable issues of timing. Most of the time, either Jack or Jill will get to the library first and get both books. In this case, the one who gets the books writes a paper and returns the books; then the other one gets the books and writes their paper. Only rarely will they arrive at the same time and each get one of the two books. 99 times out of 100 or 999 times out of 1,000, a program will run to completion perfectly normally. Only rarely will it hang for no apparent reason. Of course, if a multithreaded server is handling hundreds or thousands of connections a minute, even a problem that occurs only once every million requests can hang the server in short order.

The most important technique for preventing deadlock is to avoid unnecessary synchronization. If there's an alternative approach for ensuring thread safety, such as making objects immutable or keeping a local copy of an object, use it. Synchronization should be a

last resort for ensuring thread safety. If you do need to synchronize, keep the synchronized blocks small and try not to synchronize on more than one object at a time. This can be tricky, though, because many of the methods from the Java class library that your code may invoke synchronize on objects you aren't aware of. Consequently, you may in fact be synchronizing on many more objects than you expect.

The best you can do in the general case is carefully consider whether deadlock is likely to be a problem and design your code around it. If multiple objects need the same set of shared resources to operate, make sure they request them in the same order. For instance, if Class A and Class B need exclusive access to Object X and Object Y, make sure that both classes request X first and Y second. If neither requests Y unless it already possesses X, deadlock is not a problem.

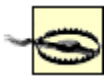
5.5. Thread Scheduling

When multiple threads are running at the same time (more properly, when multiple threads are available to be run at the same time), you have to consider issues of thread scheduling. You need to make sure that all important threads get at least some time to run and that the more important threads get more time. Furthermore, you want to ensure that the threads execute in a reasonable order. If your web server has 10 queued requests, each of which requires 5 seconds to process, you don't want to process them in series. If you do, the first request will finish in 5 seconds but the second will take 10, the third 15, and so on until the last request, which will have to wait almost a minute to be serviced. By that point, the user has likely gone to another page. By running threads in parallel, you might be able to process all 10 requests in only 10 seconds total. The reason this strategy works is that there's a lot of dead time in servicing a typical web request, time in which the thread is simply waiting for the network to catch up with the CPU—time the VM's thread scheduler can put to good use by other threads. However, CPU-bound threads (as opposed to the I/O-bound threads more common in network programs) may never reach a point where they have to wait for more input. It is possible for such a thread to starve all other threads by taking all the available CPU resources. With a little thought, you can avoid this problem. In fact, starvation is a considerably easier problem to avoid than either mis-synchronization or deadlock.

5.5.1. Priorities

Not all threads are created equal. Each thread has a priority, specified as an integer from 1 to 10. When multiple threads are able to run, the VM will generally run only the highest-priority thread, although that's not a hard-and-fast rule. In Java, 10 is the highest priority and 1 is the

lowest. The default priority is 5, and this is the priority that your threads will have unless you deliberately set them otherwise.



This is exact opposite of the normal Unix way of prioritizing processes, in which the higher the priority number of a process, the less CPU time the process gets.

These three priorities (1, 5, and 10) are often specified as the three named constants `Thread.MIN_PRIORITY`, `Thread.NORM_PRIORITY`, and `Thread.MAX_PRIORITY`:

```
public static final int MIN_PRIORITY  = 1;
public static final int NORM_PRIORITY = 5;
public static final int MAX_PRIORITY  = 10;
```

Sometimes you want to give one thread more time than another. Threads that interact with the user should get very high priorities so that perceived responsiveness will be very quick. On the other hand, threads that calculate in the background should get low priorities. Tasks that will complete quickly should have high priorities. Tasks that take a long time should have low priorities so that they won't get in the way of other tasks. The priority of a thread can be changed using the `setPriority( )` method:

```
public final void setPriority(int newPriority)
```

Attempting to exceed the maximum priority or set a nonpositive priority throws an `IllegalArgumentException`.

The `getPriority( )` method returns the current priority of the thread:

```
public final int getPriority( )
```

For instance, in Example 5-11, you might want to give higher priorities to the threads that do the calculating than the main program that spawns the threads. This is easily achieved by changing the `calculateDigest( )` method to set the priority of each spawned thread to 8:

```
public void calculateDigest( ) {

  ListCallbackDigest cb = new ListCallbackDigest(input);
  cb.addDigestListener(this);
  Thread t = new Thread(cb);
  t.setPriority(8);
  t.start( );
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    }
```

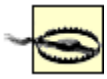
In general, though, try to avoid using too high a priority for threads, since you run the risk of starving other, lower-priority threads.

5.5.2. Preemption

Every virtual machine has a thread scheduler that determines which thread to run at any given time. There are two kinds of thread scheduling: *preemptive* and *cooperative*. A preemptive thread scheduler determines when a thread has had its fair share of CPU time, pauses that thread, and then hands off control of the CPU to a different thread. A cooperative thread scheduler waits for the running thread to pause itself before handing off control of the CPU to a different thread. A virtual machine that uses cooperative thread scheduling is much more susceptible to thread starvation than a virtual machine that uses preemptive thread scheduling, since one high-priority, uncooperative thread can hog an entire CPU.

All Java virtual machines are guaranteed to use preemptive thread scheduling between priorities. That is, if a lower-priority thread is running when a higher-priority thread becomes able to run, the virtual machine will sooner or later (and probably sooner) pause the lower-priority thread to allow the higher-priority thread to run. The higher-priority thread *preempts* the lower-priority thread.

The situation when multiple threads of the same priority are able to run is trickier. A preemptive thread scheduler will occasionally pause one of the threads to allow the next one in line to get some CPU time. However, a cooperative thread scheduler will not. It will wait for the running thread to explicitly give up control or come to a stopping point. If the running thread never gives up control and never comes to a stopping point and if no higher-priority threads preempt the running thread, all other threads will starve. This is a bad thing. It's important to make sure all your threads periodically pause themselves so that other threads have an opportunity to run.



A starvation problem can be hard to spot if you're developing on a VM that uses preemptive thread scheduling. Just because the problem doesn't arise on your machine doesn't mean it won't arise on your customers' machines if their VMs use cooperative thread scheduling. Most current virtual machines use preemptive thread scheduling, but some older virtual machines are cooperatively scheduled.

There are 10 ways a thread can pause in favor of other threads or indicate that it is ready to pause. These are:

- It can block on I/O.
- It can block on a synchronized object.
- It can yield.
- It can go to sleep.
- It can join another thread.
- It can wait on an object.
- It can finish.
- It can be preempted by a higher-priority thread.
- It can be suspended.
- It can stop.

You should inspect every `run( )` method you write to make sure that one of these conditions will occur with reasonable frequency. The last two possibilities are deprecated because they have the potential to leave objects in inconsistent states, so let's look at the other eight ways a thread can be a cooperative citizen of the virtual machine.

5.5.2.1. Blocking

Blocking occurs any time a thread has to stop and wait for a resource it doesn't have. The most common way a thread in a network program will voluntarily give up control of the CPU is by blocking on I/O. Since CPUs are much faster than networks and disks, a network program will often block while waiting for data to arrive from the network or be sent out to the network. Even though it may block for only a few milliseconds, this is enough time for other threads to do significant work.

Threads can also block when they enter a synchronized method or block. If the thread does not already possess the lock for the object being synchronized on and some other thread does possess that lock, the thread will pause until the lock is released. If the lock is never released, the thread is permanently stopped.

Neither blocking on I/O nor blocking on a lock will release any locks the thread already possesses. For I/O blocks, this is not such a big deal, since eventually the I/O will either unblock and the thread will continue or an `IOException` will be thrown and the thread will then exit the synchronized block or method and release its locks. However, a thread blocking on a lock that it doesn't possess will never give up its own locks. If one thread is waiting for a

lock that a second thread owns and the second thread is waiting for a lock that the first thread owns, deadlock results.

5.5.2.2. Yielding

The second way for a thread to give up control is to explicitly yield. A thread does this by invoking the static `Thread.yield()` method:

```
public static void yield( )
```

This signals the virtual machine that it can run another thread if one is ready to run. Some virtual machines, particularly on real-time operating systems, may ignore this hint.

Before yielding, a thread should make sure that it or its associated `Runnable` object is in a consistent state that can be used by other objects. Yielding does not release any locks the thread holds. Therefore, ideally, a thread should not be synchronized on anything when it yields. If the only other threads waiting to run when a thread yields are blocked because they need the synchronized resources that the yielding thread possesses, then the other threads won't be able to run. Instead, control will return to the only thread that can run, the one that just yielded, which pretty much defeats the purpose of yielding.

Making a thread yield is quite simple in practice. If the thread's `run( )` method simply consists of an infinite loop, just put a call to `Thread.yield( )` at the end of the loop. For example:

```
public void run( ) {

  while (true) {
    // Do the thread's work...
    Thread.yield( );
  }

}
```

This gives other threads of the same priority the opportunity to run.

If each iteration of the loop takes a significant amount of time, you may want to intersperse more calls to `Thread.yield()` in the rest of the code. This precaution should have minimal effect in the event that yielding isn't necessary.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

5.5.2.3. Sleeping

Sleeping is a more powerful form of yielding. Whereas yielding indicates only that a thread is willing to pause and let other equal-priority threads have a turn, a thread that goes to sleep will pause whether any other thread is ready to run or not. This gives an opportunity to run not only to other threads of the same priority but also to threads of lower priorities . However, a thread that goes to sleep does hold onto all the locks it's grabbed. Consequently, other threads that need the same locks will be blocked even if the CPU is available. Therefore, try to avoid having threads sleeping inside a synchronized method or block.

Sometimes sleeping is useful even if you don't need to yield to other threads. Putting a thread to sleep for a specified period of time lets you write code that executes once every second, every minute, every 10 minutes, and so forth. For instance, if you wrote a network monitor program that retrieved a page from a web server every five minutes and emailed the webmaster if the server had crashed, you could implement it as a thread that slept for five minutes between retrievals.

A thread goes to sleep by invoking one of two overloaded static `Thread.sleep( )` methods. The first takes the number of milliseconds to sleep as an argument. The second takes both the number of milliseconds and the number of nanoseconds:

```
public static void sleep(long milliseconds) throws InterruptedException
public static void sleep(long milliseconds, int nanoseconds)
 throws  InterruptedException
```

While most modern computer clocks have at least close-to-millisecond accuracy, nanosecond accuracy is rarer. There's no guarantee that you can actually time the sleep to within a nanosecond or even within a millisecond on any particular virtual machine. If the local hardware can't support that level of accuracy, the sleep time is simply rounded to the nearest value that can be measured. For example:

```
public void run( ) {

  while (true) {
    if (!getPage("http://www.cafeaulait.org/")) {
      mailError("elharo@metalab.unc.edu");
    }
    try {
      Thread.sleep(300000); // 300,000 milliseconds == 5 minutes
    }
    catch (InterruptedException ex) {
      break;
    }
  }

}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The thread is not absolutely guaranteed to sleep as long as it wants to. On occasion, the thread may not be woken up until some time after its requested wake-up call, simply because the VM is busy doing other things. It is also possible that some other thread will do something to wake up the sleeping thread before its time. Generally, this is accomplished by invoking the sleeping thread's `interrupt( )` method.

```
public void interrupt( )
```

This is one of those cases where the distinction between the thread and the `Thread` object is important. Just because the thread is sleeping doesn't mean that other threads that are awake can't work with the corresponding `Thread` object through its methods and fields. In particular, another thread can invoke the sleeping `Thread` object's `interrupt( )` method, which the sleeping thread experiences as an `InterruptedException`. From that point forward, the thread is awake and executes as normal, at least until it goes to sleep again. In the previous example, an `InterruptedException` is used to terminate a thread that would otherwise run forever. When the `InterruptedException` is thrown, the infinite loop is broken, the `run( )` method finishes, and the thread dies. The user interface thread can invoke this thread's `interrupt( )` method when the user selects Exit from a menu or otherwise indicates that he wants the program to quit.

5.5.2.4. Joining threads

It's not uncommon for one thread to need the result of another thread. For example, a web browser loading an HTML page in one thread might spawn a separate thread to retrieve every image embedded in the page. If the `IMG` elements don't have `HEIGHT` and `WIDTH` attributes, the main thread might have to wait for all the images to load before it can finish by displaying the page. Java provides three `join( )` methods to allow one thread to wait for another thread to finish before continuing. These are:

```
public final void join( ) throws InterruptedException
public final void join(long milliseconds) throws InterruptedException
public final void join(long milliseconds, int nanoseconds)
 throws InterruptedException
```

The first variant waits indefinitely for the *joined* thread to finish. The second two variants wait for the specified amount of time, after which they continue even if the joined thread has not finished. As with the `sleep()` method, nanosecond accuracy is not guaranteed.

The joining thread (that is, the one that invokes the `join()` method) waits for the joined thread (that is, the one whose `join( )` method is invoked) to finish. For instance, consider this code fragment. We want to find the minimum, maximum, and median of a random array of doubles. It's quicker to do this with a sorted array. We spawn a new thread to sort the array,

then join to that thread to await its results. Only when it's done do we read out the desired values.

```
double[] array = new double[10000];                          // 1
for (int i = 0; i < array.length; i++) {                    // 2
  array[i] = Math.random( );                                 // 3
}                                                            // 4
SortThread t = new SortThread(array);                        // 5
t.start( );                                                  // 6
try {                                                        // 7
  t.join( );                                                 // 8
  System.out.println("Minimum: " + array[0]);                 // 9
  System.out.println("Median: " + array[array.length/2]);    // 10
  System.out.println("Maximum: " + array[array.length-1]);   // 11
}                                                            // 12
catch (InterruptedException ex) {                            // 13
}                                                            // 14
```

First lines 1 through 4 execute, filling the array with random numbers. Then line 5 creates a new `SortThread`. Line 6 starts the thread that will sort the array. Before we can find the minimum, median, and maximum of the array, we need to wait for the sorting thread to finish. Therefore, line 8 joins the current thread to the sorting thread. At this point, the thread executing these lines of code stops in its tracks. It waits for the sorting thread to finish running. The minimum, median, and maximum are not retrieved in lines 9 through 10 until the sorting thread has finished running and died. Notice that at no point is there a reference to the thread that pauses. It's not the `Thread` object on which the `join()` method is invoked; it's not passed as an argument to that method. It exists implicitly only as the current thread. If this is within the normal flow of control of the `main( )` method of the program, there may not be any `Thread` variable anywhere that points to this thread.

A thread that's joined to another thread can be interrupted just like a sleeping thread if some other thread invokes its `interrupt( )` method. The thread experiences this invocation as an `InterruptedException`. From that point forward, it executes as normal, starting from the `catch` block that caught the exception. In the preceding example, if the thread is interrupted, it skips over the calculation of the minimum, median, and maximum because they won't be available if the sorting thread was interrupted before it could finish.

We can use `join( )` to fix up Example 5-4. Example 5-4s problem was that the `main( )` method tended to outpace the threads whose results the `main( )` method was using. It's straightforward to fix this by joining to each thread before trying to use its result. Example 5-13 demonstrates.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Example 5-13. Avoid a race condition by joining to the thread that has a result you need

```
import java.io.*;

public class JoinDigestUserInterface {

  public static void main(String[] args) {

    ReturnDigest[] digestThreads = new ReturnDigest[args.length];

    for (int i = 0; i < args.length; i++) {

      // Calculate the digest
      File f = new File(args[i]);
      digestThreads[i] = new ReturnDigest(f);
      digestThreads[i].start( );

    }

    for (int i = 0; i < args.length; i++) {

      try {
        digestThreads[i].join( );
        // Now print the result
        StringBuffer result = new StringBuffer(args[i]);
        result.append(": ");
        byte[] digest = digestThreads[i].getDigest( );
        for (int j = 0; j < digest.length; j++) {
          result.append(digest[j] + " ");
        }
        System.out.println(result);
      }
      catch (InterruptedException ex) {
        System.err.println("Thread Interrupted before completion");
      }

    }

  }

}
```

Since Example 5-13 joins to threads in the same order as the threads are started, this fix also has the side effect of printing the output in the same order as the arguments used to construct the threads, rather than in the order the threads finish. This modification doesn't make the program any slower, but it may occasionally be an issue if you want to get the output of a thread as soon as it's done, without waiting for other unrelated threads to finish first.

5.5.2.5. Waiting on an object

A thread can *wait* on an object it has locked. While waiting, it releases the lock on the object and pauses until it is notified by some other thread. Another thread changes the object in some way, notifies the thread waiting on that object, and then continues. This differs from

joining in that neither the waiting nor the notifying thread has to finish before the other thread can continue. Waiting is used to pause execution until an object or resource reaches a certain state. Joining is used to pause execution until a thread finishes.

Waiting on an object is one of the lesser-known ways a thread can pause. That's because it doesn't involve any methods in the `Thread` class. Instead, to wait on a particular object, the thread that wants to pause must first obtain the lock on the object using `synchronized` and then invoke one of the object's three overloaded `wait( )` methods:

```
public final void wait( ) throws InterruptedException
public final void wait(long milliseconds) throws InterruptedException
public final void wait(long milliseconds, int nanoseconds)
 throws InterruptedException
```

These methods are not in the `Thread` class; rather, they are in the `java.lang.Object` class. Consequently, they can be invoked on any object of any class. When one of these methods is invoked, the thread that invoked it releases the lock on the object it's waiting on (though not any locks it possesses on other objects) and goes to sleep. It remains asleep until one of three things happens:

- The timeout expires.
- The thread is interrupted.
- The object is notified.

The *timeout* is the same as for the `sleep( )` and `join( )` methods; that is, the thread wakes up after the specified amount of time has passed (within the limits of the local hardware clock accuracy). When the timeout expires, execution of the thread resumes with the statement immediately following the invocation of `wait()`. However, if the thread can't immediately regain the lock on the object it was waiting on, it may still be blocked for some time.

Interruption works the same way as `sleep( )` and `join( )`: some other thread invokes the thread's `interrupt( )` method. This causes an `InterruptedException`, and execution resumes in the `catch` block that catches the exception. The thread regains the lock on the object it was waiting on before the exception is thrown, however, so the thread may still be blocked for some time after the `interrupt( )` method is invoked.

The third possibility, *notification*, is new. Notification occurs when some other thread invokes the `notify( )` or `notifyAll( )` method on the object on which the thread is waiting. Both of these methods are in the `java.lang.Object` class:

```
public final void notify( )
public final void notifyAll( )
```

These must be invoked on the object the thread was waiting on, not generally on the `Thread` itself. Before notifying an object, a thread must first obtain the lock on the object using a synchronized method or block. The `notify( )` method selects one thread more or less at random from the list of threads waiting on the object and wakes it up. The `notifyAll ()` method wakes up every thread waiting on the given object.

Once a waiting thread is notified, it attempts to regain the lock of the object it was waiting on. If it succeeds, execution resumes with the statement immediately following the invocation of `wait()`. If it fails, it blocks on the object until its lock becomes available; then execution resumes with the statement immediately following the invocation of `wait( )`.

For example, suppose one thread is reading a JAR archive from a network connection. The first entry in the archive is the manifest file. Another thread might be interested in the contents of the manifest file even before the rest of the archive is available. The interested thread could create a custom `ManifestFile` object, pass a reference to this object to the thread that would read the JAR archive, and wait on it. The thread reading the archive would first fill the `ManifestFile` with entries from the stream, then notify the `ManifestFile`, then continue reading the rest of the JAR archive. When the reader thread notified the `ManifestFile`, the original thread would wake up and do whatever it planned to do with the now fully prepared `ManifestFile` object. The first thread works something like this:

```
ManifestFile m = new ManifestFile( );
JarThread    t = new JarThread(m, in);
synchronized (m) {
  t.start( );
  try {
    m.wait( );
    // work with the manifest file...
  }
  catch (InterruptedException ex) {
    // handle exception...
  }
}
```

The `JarThread` class works like this:

```
ManifestFile theManifest;
InputStream in;

public JarThread(Manifest m, InputStream in) {
  theManifest = m;
  this.in= in;
}

public void run( ) {

  synchronized (theManifest) {
    // read the manifest from the stream in...
    theManifest.notify( );
```

```
    }
    // read the rest of the stream...

  }
```

Waiting and notification are more commonly used when multiple threads want to wait on the same object. For example, one thread may be reading a web server log file in which each line contains one entry to be processed. Each line is placed in a `java.util.List` as it's read. Several threads wait on the `List` to process entries as they're added. Every time an entry is added, the waiting threads are notified using the `notifyAll()` method. If more than one thread is waiting on an object, `notifyAll( )` is preferred, since there's no way to select which thread to notify. When all threads waiting on one object are notified, all will wake up and try to get the lock on the object. However, only one can succeed immediately. That one continues; the rest are blocked until the first one releases the lock. If several threads are all waiting on the same object, a significant amount of time may pass before the last one gets its turn at the lock on the object and continues. It's entirely possible that the object on which the thread was waiting will once again have been placed in an unacceptable state during this time. Thus, you'll generally put the call to `wait( )` in a loop that checks the current state of the object. Do not assume that just because the thread was notified, the object is now in the correct state. Check it explicitly if you can't guarantee that once the object reaches a correct state it will never again reach an incorrect state. For example, this is how the client threads waiting on the log file entries might look:

```
private List entries;

public void processEntry( ) {

  synchronized (entries) { // must synchronize on the object we wait on
    while (entries.size( ) == 0) {
      try {
        entries.wait( );
        // We stopped waiting because entries.size( ) became non-zero
        // However we don't know that it's still non-zero so we
        // pass through the loop again to test its state now.
      }
      catch (InterruptedException ex) {
        // If interrupted, the last entry has been processed so
        return;
      }
    }
    String entry = (String) entries.remove(entries.size( )-1);
    // process this entry...
  }

}
```

The code reading the log file and adding entries to the vector might look something like this:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public void readLogFile( ) {

  String entry;

  while (true) {
    entry = log.getNextEntry( );
    if (entry == null) {
      // There are no more entries to add to the vector so
      // we have to interrupt all threads that are still waiting.
      // Otherwise, they'll wait forever.
      for (int i = 0; i < threads.length; i++) threads[i].interrupt( );
      break;
    }
    synchronized (entries) {
      entries.add(0, entry);
      entries.notifyAll( );
    }
  }

}
```

5.5.2.6. Priority-based preemption

Since threads are preemptive between priorities, you do not need to worry about giving up time to higher-priority threads. A high-priority thread will preempt lower-priority threads when it's ready to run. However, when the high-priority thread finishes running or blocks, it generally won't be the same low-priority thread that runs next. Instead, most non-real-time VMs use a round-robin scheduler so that the lower-priority thread that has been waiting longest will run next.

For example, suppose there are three threads with priority 5 named A, B, and C running in a cooperatively scheduled virtual machine. None of them will yield or block. Thread A starts running first. It runs for a while and is then preempted by thread D, which has priority 6. A stops running. Eventually, thread D blocks, and the thread scheduler looks for the next highest-priority thread to run. It finds three: A, B, and C. Thread A has already had some time to run, so the thread scheduler picks B (or perhaps C; this doesn't have to go in alphabetical order). B runs for a while when thread D suddenly unblocks. Thread D still has higher priority so the virtual machine pauses thread B and lets D run for a while. Eventually, D blocks again, and the thread scheduler looks for another thread to run. Again, it finds A, B, and C, but at this point, A has had some time and B has had some time, but C hasn't had any. So the thread scheduler picks thread C to run. Thread C runs until it is once again preempted by thread D. When thread D blocks again, the thread scheduler finds three threads ready to run. Of the three, however, A ran the longest ago, so the scheduler picks thread A. From this point forward, every time D preempts and blocks and the scheduler needs a new thread to run, it will run the threads A, B, and C in that order, circling back around to A after C.

If you'd rather avoid explicit yielding, you can use a higher-priority thread to force the lower-priority threads to give up time to each other. In essence, you can use a high-priority thread scheduler of your own devising to make all threading preemptive. The trick is to run a high-priority thread that does nothing but sleep and wake up periodically, say every 100 milliseconds. This will split the lower-priority threads into 100-millisecond time slices. It isn't necessary for the thread that's doing the splitting to know anything about the threads it's preempting. It's simply enough that it exists and is running. Example 5-14 demonstrates with a `TimeSlicer` class that allows you to guarantee preemption of threads with priorities less than a fixed value every `timeslice` milliseconds.

Example 5-14. A thread that forces preemptive scheduling for lower-priority threads

```
public class TimeSlicer extends Thread {

  private long timeslice;

  public TimeSlicer(long milliseconds, int priority) {

    this.timeslice = milliseconds;
    this.setPriority(priority);
    // If this is the last thread left, it should not
    // stop the VM from exiting
    this.setDaemon(true);

  }

  // Use maximum priority
  public TimeSlicer(long milliseconds) {
    this(milliseconds, 10);
  }

  // Use maximum priority and 100ms timeslices
  public TimeSlicer( ) {
    this(100, 10);
  }

  public void run( ) {

    while (true) {
      try {
        Thread.sleep(timeslice);
      }
      catch (InterruptedException ex) {
      }
    }

  }

}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

5.5.2.7. Finish

The final way a thread can give up control of the CPU in an orderly fashion is by *finishing*. When the `run()` method returns, the thread dies and other threads can take over. In network applications, this tends to occur with threads that wrap a single blocking operation, such as downloading a file from a server, so that the rest of the application won't be blocked.

Otherwise, if your `run( )` method is so simple that it always finishes quickly enough without blocking, there's a very real question of whether you should spawn a thread at all. There's a nontrivial amount of overhead for the virtual machine in setting up and tearing down threads. If a thread is finishing in a small fraction of a second anyway, chances are it would finish even faster if you used a simple method call rather than a separate thread.

5.6. Thread Pools

Adding multiple threads to a program dramatically improves performance, especially for I/O-bound programs such as most network programs. However, threads are not without overhead of their own. Starting a thread and cleaning up after a thread that has died takes a noticeable amount of work from the virtual machine, especially if a program spawns hundreds of threads—not an unusual occurrence for even a low- to medium-volume network server. Even if the threads finish quickly, this can overload the garbage collector or other parts of the VM and hurt performance, just like allocating thousands of any other kind of object every minute. Even more importantly, switching between running threads carries overhead. If the threads are blocking naturally—for instance, by waiting for data from the network—there's no real penalty to this, but if the threads are CPU-bound, then the total task may finish more quickly if you can avoid a lot of switching between threads. Finally, and most importantly, although threads help make more efficient use of a computer's limited CPU resources, there are still only a finite amount of resources to go around. Once you've spawned enough threads to use all the computer's available idle time, spawning more threads just wastes MIPS and memory on thread management.

Fortunately, you can get the best of both worlds by reusing threads. You cannot restart a thread once it's died, but you can engineer threads so that they don't die as soon as they've finished one task. Instead, put all the tasks you need to accomplish in a queue or other data structure and have each thread retrieve a new task from the queue when it's completed its previous task. This is called *thread pooling*, and the data structure in which the tasks are kept is called the *pool*.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The simplest way to implement a thread pool is by allotting a fixed number of threads when the pool is first created. When the pool is empty, each thread waits on the pool. When a task is added to the pool, all waiting threads are notified. When a thread finishes its assigned task, it goes back to the pool for a new task. If it doesn't get one, it waits until a new task is added to the pool.

An alternative is to put the threads themselves in the pool and have the main program pull threads out of the pool and assign them tasks. If no thread is in the pool when a task becomes necessary, the main program can spawn a new thread. As each thread finishes a task, it returns to the pool. (Imagine this scheme as a union hall in which new workers join the union only when full employment of current members is achieved.)

There are many data structures you can use for a pool, although a queue is probably the most efficient for ensuring that tasks are performed in a first-in, first-out order. Whichever data structure you use to implement the pool, however, you have to be extremely careful about synchronization, since many threads will interact with it very close together in time. The simplest way to avoid problems is to use either a `java.util.Vector` (which is fully synchronized) or a synchronized `Collection` from the Java Collections API.

Let's look at an example. Suppose you want to gzip every file in the current directory using a `java.util.zip.GZIPOutputStream`. On the one hand, this is an I/O-heavy operation because all the files have to be read and written. On the other hand, data compression is a very CPU-intensive operation, so you don't want too many threads running at once. This is a good opportunity to use a thread pool. Each client thread will compress files while the main program will determine which files to compress. In this example, the main program is likely to significantly outpace the compressing threads since all it has to do is list the files in a directory. Therefore, it's not out of the question to fill the pool first, then start the threads that compress the files in the pool. However, to make this example as general as possible, we'll allow the main program to run in parallel with the zipping threads.

Example 5-15 shows the `GZipThread` class. It contains a private field called `pool` containing a reference to the pool. Here that field is declared to have `List` type, but it's always accessed in a strictly queue-like first-in, first-out order. The `run( )` method removes `File` objects from the pool and gzips each one. If the pool is empty when the thread is ready to get something new from the pool, then the thread waits on the `pool` object.

Example 5-15. The GZipThread class

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class GZipThread extends Thread {

  private List pool;
  private static int filesCompressed = 0;

  public GZipThread(List pool) {
    this.pool = pool;
  }

  private static synchronized void incrementFilesCompressed( ) {
    filesCompressed++;
  }

  public void run( ) {

    while (filesCompressed != GZipAllFiles.
                              getNumberOfFilesToBeCompressed( )) {

      File input = null;

      synchronized (pool) {
        while (pool.isEmpty( )) {
          if (filesCompressed == GZipAllFiles.
                                   getNumberOfFilesToBeCompressed( )) {
            System.out.println("Thread ending");
            return;
          }
          try {
            pool.wait( );
          }
          catch (InterruptedException ex) {
          }
        }

        input = (File) pool.remove(pool.size( )-1);
        incrementFilesCompressed( );

      }

      // don't compress an already compressed file
      if (!input.getName( ).endsWith(".gz")) {
        try {
          InputStream in = new FileInputStream(input);
          in = new BufferedInputStream(in);

          File output = new File(input.getParent( ), input.getName( ) + ".gz");
          if (!output.exists( )) { // Don't overwrite an existing file
            OutputStream out = new FileOutputStream(output);
            out = new GZIPOutputStream(out);
            out = new BufferedOutputStream(out);
            int b;
            while ((b = in.read( )) != -1) out.write(b);
            out.flush( );
            out.close( );
```

```
                in.close( );
              }
            }
            catch (IOException ex) {
              System.err.println(ex);
            }

          } // end if

        } // end while

      } // end run

    } // end ZipThread
```

Example 5-16 is the main program. It constructs the pool as a `Vector` object, passes this to four newly constructed `GZipThread` objects, starts all four threads, and iterates through all the files and directories listed on the command line. Those files and files in those directories are added to the pool for eventual processing by the four threads.

Example 5-16. The GZipThread user interface class

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

public class GZipAllFiles {

  public final static int THREAD_COUNT = 4;
  private static int filesToBeCompressed = -1;

  public static void main(String[] args) {

    Vector pool = new Vector( );
    GZipThread[] threads = new GZipThread[THREAD_COUNT];

    for (int i = 0; i < threads.length; i++) {
      threads[i] = new GZipThread(pool);
      threads[i].start( );
    }

    int totalFiles = 0;
    for (int i = 0; i < args.length; i++) {

      File f = new File(args[i]);
      if (f.exists( )) {
        if (f.isDirectory( )) {
          File[] files = f.listFiles( );
          for (int j = 0; j < files.length; j++) {
            if (!files[j].isDirectory( )) { // don't recurse directories
              totalFiles++;
              synchronized (pool) {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
                    pool.add(0, files[j]);
                    pool.notifyAll( );
                  }
                }
              }
            }
            else {
              totalFiles++;
              synchronized (pool) {
                pool.add(0, f);
                pool.notifyAll( );
              }
            }

          } // end if

        } // end for

        filesToBeCompressed = totalFiles;

        // make sure that any waiting thread knows that no
        // more files will be added to the pool
        for (int i = 0; i < threads.length; i++) {
          threads[i].interrupt( );
        }

      }

      public static int getNumberOfFilesToBeCompressed( ) {
        return filesToBeCompressed;
      }

    }
```

The big question here is how to tell the program that it's done and should exit. You can't simply exit when all files have been added to the pool, because at that point most of the files haven't been processed. Neither can you exit when the pool is empty, because that may occur at the start of the program (before any files have been placed in the pool) or at various intermediate times when not all files have yet been put in the pool but all files that have been put there are processed. The latter possibility also prevents the use of a simple counter scheme.

The solution adopted here is to separately track the number of files that need to be processed (`GZipAllFiles.filesToBeCompressed`) and the number of files actually processed (`GZipThread.filesCompressed`). When these two values match, all threads' `run( )` methods return. Checks are made at the start of each of the `while` loops in the `run( )` method to see whether it's necessary to continue. This scheme is preferred to the deprecated `stop( )` method, because it won't suddenly stop the thread while it's halfway through compressing a file. This gives us much more fine-grained control over exactly when and where the thread stops.

Initially, `GZipAllFiles.filesToBeCompressed` is set to the impossible value -1. Only when the final number is known is it set to its real value. This prevents early coincidental matches between the number of files processed and the number of files to be processed. It's possible that when the final point of the `main( )` method is reached, one or more of the threads will be waiting. Thus, we interrupt each of the threads (an action that has no effect if the thread is merely processing and not waiting or sleeping) to make sure it checks one last time.

And finally, the last element of this program is the private `GZipThread.incrementFilesCompressed( )` method. This method is synchronized to ensure that if two threads try to update the `filesCompressed` field at the same time, one will wait. Otherwise, the `GZipThread.filesCompressed` field could end up one short of the true value and the program would never exit. Since the method is static, all threads synchronize on the same `Class` object. A synchronized instance method wouldn't be sufficient here.

The complexity of determining when to stop this program is mostly atypical of the more heavily threaded programs you'll write because it does have such a definite ending point: the point at which all files are processed. Most network servers continue indefinitely until some part of the user interface shuts them down. The real solution here is to provide some sort of simple user interface—such as typing a period on a line by itself—that ends the program.

This chapter has been a whirlwind tour of threading in Java, covering the bare minimum you need to know to write multithreaded network programs. For a more detailed and comprehensive look with many more examples, check out *Java Threads*, by Scott Oaks and Henry Wong (O'Reilly). Once you've mastered that book, Doug Lea's *Concurrent Programming in Java* (Addison Wesley) provides a comprehensive look at the traps and pitfalls of concurrent programming from a design patterns perspective.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004