Thread Pooling

CHAPTER

13

IN THIS CHAPTER

When design situations arise that could benefit by using many short-lived threads, thread *pooling* is a useful technique. Rather than create a brand new thread for each task, you can have one of the threads from the thread pool pulled out of the pool and assigned to the task. When the thread is finished with the task, it adds itself back to the pool and waits for another assignment.

In this chapter, I present two examples that use thread pooling. One creates a pool of threads that can be generically used to run `Runnable` objects. The other creates a pool of threads for servicing requests that come into a simple Hypertext Transfer Protocol (HTTP) server (a Web page server).

Benefits of Thread Pooling

Thread pooling saves the virtual machine the work of creating brand new threads for every short-lived task. In addition, it minimizes overhead associated with getting a thread started and cleaning it up after it dies. By creating a pool of threads, a single thread from the pool can be recycled over and over for different tasks.

With the thread pooling technique, you can reduce response time because a thread is already constructed and started and is simply waiting for its next task. In the case of an HTTP server, an available thread in the pool can deliver each new file requested. Without pooling, a brand new thread would have to be constructed and started before the request could be serviced.

Another characteristic of the thread pools discussed in this chapter is that they are fixed in size at the time of construction. All the threads are started, and then each goes into a wait state (which uses very few processor resources) until a task is assigned to it. This fixed size characteristic holds the number of assigned tasks to an upper limit. If all the threads are currently assigned a task, the pool is empty. New service requests can simply be rejected or can be put into a wait state until one of the threads finishes its task and returns itself to the pool. In the case of an HTTP server, this limit prevents a flood of requests from overwhelming the server to the point of servicing everyone very slowly or even crashing. You can expand on the designs presented in this chapter to include a method to support *growing* the size of the pool at runtime if you need this kind of dynamic tuning.

Considerations and Costs of Thread Pooling

Thread pooling works only when the tasks are relatively short-lived. An HTTP server fulfilling a request for a particular file is a perfect example of a task that is done best in another thread and does not run for very long. By using another thread to service each request, the server can simultaneously deliver multiple files. For tasks that run indefinitely, a normal thread is usually a better choice.

A cost of thread pooling is that all the threads in the pool are constructed and started in hopes that they will be needed. It is possible that the pool will have capacity far greater than necessary. Care should be taken to measure the utilization of the threads in the pool and tune the capacity to an optimal level.

The thread pool might also be too small. If tasks are rejected when the pool is empty (as is the case in the HTTP server example later in this chapter), a high rejection rate might be unacceptable. If the tasks are not rejected, but are held in a wait state, the waiting time could become too long. When the waiting time is long, response time worsens.

Also, some risk exists that one of the tasks assigned to a thread could cause it to deadlock or die. If thread pooling is not being used, this is still a problem. It is an even bigger problem if threads leave the pool and never return. Eventually, the pool will become empty and remain empty. You should code as carefully as possible to avoid this pitfall.

A Generic Thread Pool: ThreadPool

The class `ThreadPool`, shown in Listing 13.1, is used to pool a set of threads for generic tasks. The worker threads are running inside `ThreadPoolWorker` objects, shown in Listing 13.2. When a `ThreadPool` object is constructed, it constructs as many `ThreadPoolWorker` objects as are specified. To run a task, `ThreadPool` is passed a `Runnable` object through its `execute()` method. If a `ThreadPoolWorker` object is available, the `execute()` method removes it from the pool and hands off the `Runnable` to it for execution. If the pool is empty, the `execute()` method blocks until a worker becomes available. When the `run()` method of the `Runnable` task passed in returns, the `ThreadPoolWorker` has completed the task and puts itself back into the pool of available workers. There is no other signal that the task has been completed. If a signal is necessary, it should be coded in the task's `run()` method just before it returns.

NOTE

The `Runnable` interface is being used here in a slightly different manner than you've seen before. Earlier in the book, it was required that a `Runnable` object reference be passed to the constructor of `Thread`, and the `run()` method was the entry point for the new thread. The `run()` method was never called directly.

Here, instead of creating a new interface for thread pooling, the use of the existing `Runnable` interface is being expanded a little. Now, one of the worker threads will invoke the `run()` method directly (see line 72 of `ThreadPoolWorker` in Listing 13.2) when it is assigned to execute the `Runnable` task. I chose to use `Runnable` in this design so that passing a task to `execute()` would cause the `run()` method to be called by another thread in much the same way as `Thread`'s `start()` method causes a new thread to invoke `run()`.

LISTING 13.1 ThreadPool.java—A Thread Pool Used to Run Generic Tasks

```
 1: // uses ObjectFIFO from chapter 18
 2:
 3: public class ThreadPool extends Object {
 4:     private ObjectFIFO idleWorkers;
 5:     private ThreadPoolWorker[] workerList;
 6:
 7:     public ThreadPool(int numberOfThreads) {
 8:         // make sure that it's at least one
 9:         numberOfThreads = Math.max(1, numberOfThreads);
10:
11:         idleWorkers = new ObjectFIFO(numberOfThreads);
12:         workerList = new ThreadPoolWorker[numberOfThreads];
13:
14:         for ( int i = 0; i < workerList.length; i++ ) {
15:             workerList[i] = new ThreadPoolWorker(idleWorkers);
16:         }
17:     }
18:
19:     public void execute(Runnable target)
                        ➥throws InterruptedException {
20:         // block (forever) until a worker is available
21:         ThreadPoolWorker worker =
                       ➥(ThreadPoolWorker) idleWorkers.remove();
22:         worker.process(target);
23:     }
24:
25:     public void stopRequestIdleWorkers() {
26:         try {
27:             Object[] idle = idleWorkers.removeAll();
28:             for ( int i = 0; i < idle.length; i++ ) {
29:                 ( (ThreadPoolWorker) idle[i] ).stopRequest();
30:             }
31:         } catch ( InterruptedException x ) {
32:             Thread.currentThread().interrupt(); // re-assert
33:         }
34:     }
35:
36:     public void stopRequestAllWorkers() {
37:         // Stop the idle one's first
38:         // productive.
39:         stopRequestIdleWorkers();
40:
41:         // give the idle workers a quick chance to die
42:         try { Thread.sleep(250); }
```

```
                ➥catch ( InterruptedException x ) { }
43:
44:         // Step through the list of ALL workers.
45:         for ( int i = 0; i < workerList.length; i++ ) {
46:             if ( workerList[i].isAlive() ) {
47:                 workerList[i].stopRequest();
48:             }
49:         }
50:     }
51: }
```

`ThreadPool` serves as the central point of control for managing the worker threads. It holds a list of all the workers created in `workerList` (line 5). The current pool of idle `ThreadPoolWorker` objects is kept in a FIFO queue, `idleWorkers` (line 4).

NOTE

First-In-First-Out (FIFO) queues allow items to be *added* to one end of the queue and *removed* from the other end. Items are removed in the exact same order as they were added (the first item *in* is the first item *out*). A FIFO queue has a fixed capacity. If a thread invokes the `add()` method when the FIFO is full, it blocks waiting until another thread removes an item. If a thread invokes the `remove()` method when the FIFO is empty, it blocks waiting until another thread adds an item.

FIFO queues are explained and demonstrated in Chapter 18, "First-In-First-Out (FIFO) Queue." You can skip ahead to look at that technique at this time if you want to know more.

The constructor (lines 7–17) takes as its only parameter an `int` specifying the number of worker threads that should be created for this pool (line 7). The number of threads is silently forced to be at least `1` (line 9). A new `ObjectFIFO` is created with a capacity large enough to hold the entire pool of worker threads (line 11). This queue holds all the workers currently available for assignment to new tasks. A `ThreadPoolWorker[]`is created to keep a handle on all the workers—regardless of whether they are currently idle (line 12). The `for` loop (lines 14–16) is used to construct each of the `ThreadPoolWorker` objects. Each has a reference to the pool of available workers passed to its constructor (line 15). Each one will use this reference to add itself back to the pool when it is ready to service a new task.

When an external thread wants to run a task using one of the threads in the pool, it invokes the `execute()` method (lines 19–23). The `execute()` method takes a `Runnable` object as a parameter. This object will have its `run()` method invoked by the next available worker thread. The

external thread blocks waiting until an idle `ThreadPoolWorker` becomes available (line 21). When one is ready, the external thread passes the `Runnable` to the worker's `process()` method (line 22), which returns right away. The external thread returns from `execute()` and is free to continue with whatever else it has to do while the worker thread runs the `target`.

The `stopRequestIdleWorkers()` method (lines 25–34) is used to request that the internal threads of the idle workers stop as soon as possible. First, all the currently idle workers are removed from the queue (line 27). Each worker then has its `stopRequest()` method invoked (line 29). You should keep in mind that as other tasks finish, more idle workers could be added to the pool and will not be stopped until another `stopRequestIdleWorkers()` invocation occurs.

The `stopRequestAllWorkers()` method (lines 36–50) is used to request that all the workers stop as soon as possible, regardless of whether they are currently idle. First, a call to `stopRequestIdleWorkers()` is done because they can be stopped right away with negligible impact (line 39). A quarter-second break is taken to give the idle workers a chance to shut down. Next, the list of all the workers is stepped through using a `for` loop (lines 45–49). Each worker that is still alive (line 46) has its `stopRequest()` method invoked (line 47). It's possible that one or more of the idle threads will not have a chance to die before the `isAlive()` check. In this case, the `stopRequest()` method will be called twice, which should be harmless.

The `ThreadPoolWorker` class, shown in Listing 13.2, is in charge of providing the thread to run the specified task. In a real-world setting, this class should probably not be `public`, but should have package scope or be an inner class to `ThreadPool`. It is never accessed directly because `ThreadPool` acts as the sole interface to external code.

LISTING 13.2 ThreadPoolWorker.java—The Internal Assistant to ThreadPool Used to Run a Task

```
 1: // uses class ObjectFIFO from chapter 18
 2:
 3: public class ThreadPoolWorker extends Object {
 4:     private static int nextWorkerID = 0;
 5:
 6:     private ObjectFIFO idleWorkers;
 7:     private int workerID;
 8:     private ObjectFIFO handoffBox;
 9:
10:     private Thread internalThread;
11:     private volatile boolean noStopRequested;
12:
13:     public ThreadPoolWorker(ObjectFIFO idleWorkers) {
14:         this.idleWorkers = idleWorkers;
```

```
15:
16:         workerID = getNextWorkerID();
17:         handoffBox = new ObjectFIFO(1); // only one slot
18:
19:         // just before returning, the thread should be created.
20:         noStopRequested = true;
21:
22:         Runnable r = new Runnable() {
23:                 public void run() {
24:                     try {
25:                         runWork();
26:                     } catch ( Exception x ) {
27:                         // in case ANY exception slips through
28:                         x.printStackTrace();
29:                     }
30:                 }
31:             };
32:
33:         internalThread = new Thread(r);
34:         internalThread.start();
35:     }
36:
37:     public static synchronized int getNextWorkerID() {
38:         // notice: sync'd at the class level to ensure uniqueness
39:         int id = nextWorkerID;
40:         nextWorkerID++;
41:         return id;
42:     }
43:
44:     public void process(Runnable target)
                    ➥throws InterruptedException {
45:         handoffBox.add(target);
46:     }
47:
48:     private void runWork() {
49:         while ( noStopRequested ) {
50:             try {
51:                 System.out.println("workerID=" + workerID +
52:                         ", ready for work");
53:                 // Worker is ready work. This will never block
54:                 // because the idleWorker FIFO queue has
55:                 // enough capacity for all the workers.
56:                 idleWorkers.add(this);
57:
```

continues

LISTING 13.2 Continued

```
58:                 // wait here until the server adds a request
59:                 Runnable r = (Runnable) handoffBox.remove();
60:
61:                 System.out.println("workerID=" + workerID +
62:                    ", starting execution of new Runnable: " + r);
63:                 runIt(r); // catches all exceptions
64:             } catch ( InterruptedException x ) {
65:                 Thread.currentThread().interrupt(); // re-assert
66:             }
67:         }
68:     }
69:
70:     private void runIt(Runnable r) {
71:         try {
72:             r.run();
73:         } catch ( Exception runex ) {
74:             // catch any and all exceptions
75:             System.err.println(
                      ➥"Uncaught exception fell through from run()");
76:             runex.printStackTrace();
77:         } finally {
78:             // Clear the interrupted flag (in case it comes back
79:             // set) so that if the loop goes again, the
80:             // handoffBox.remove() does not mistakenly
81:             // throw an InterruptedException.
82:             Thread.interrupted();
83:         }
84:     }
85:
86:     public void stopRequest() {
87:         System.out.println("workerID=" + workerID +
88:                 ", stopRequest() received.");
89:         noStopRequested = false;
90:         internalThread.interrupt();
91:     }
92:
93:     public boolean isAlive() {
94:         return internalThread.isAlive();
95:     }
96: }
```

`ThreadPoolWorker` uses the active object technique discussed in Chapter 11, "Self-Running Objects." Each worker constructed is assigned a unique `workerID` (line 7) to help clarify the output messages. In a real-world setting, individual identity tracking is not always necessary.

At the class level, the next worker ID is held in a `static` member variable, `nextWorkerID` (line 4). This variable is retrieved and incremented inside the `getNextWorkerID()` method (lines 37–42). It is `static` and `synchronized` so that the class-level lock is acquired before changes are made (line 37). This ensures that no two instances of `ThreadPoolWorker` are accidentally assigned the same `workerID` value.

A reference to the list of currently unused workers is held in `idleWorkers` (line 6). This is a reference to an `ObjectFIFO` queue, and the worker adds itself back to `idleWorkers` when it is available for assignment. The `handoffBox` FIFO queue (line 8) is used to pass `Runnable` objects to the worker in a thread-safe manner.

In the constructor (lines 13–35), the passed reference to the pool of available workers is assigned to a member variable for later access (line 14). The `getNextWorkerID()` method is used to obtain a unique `int` to store in `workerID` (line 16). An `ObjectFIFO` with a capacity of only `1` is created to be used for handing off the next `Runnable` task to the internal thread. The rest of the code in the constructor uses the standard pattern for an active object (see Chapter 11).

The `process()` method (lines 44–46) is invoked by code inside the `execute()` method of `ThreadPool`. It is used to pass the `Runnable` task in to the worker for processing. It is put into the handoff box to be noticed and picked up by the internal thread (line 45). Although `add()` declares that it will throw an `InterruptedException` if it is interrupted while waiting for space, this should never happen in this scenario. The `handoffBox` FIFO queue should be empty when the worker is available and waiting for another assignment. I chose to use an `ObjectFIFO` here to encapsulate the `wait-notify` mechanism that is necessary to signal the internal thread that a new task has arrived. It's a simpler approach and uses well-tested code.

The `runWork()` method (lines 48–68) follows the active object pattern of looping using the internal thread as long as no stop has been requested (line 49). Each time through the loop, the internal thread adds itself to the pool of available workers (line 56). It then waits indefinitely for an external thread to invoke the `process()` method and put a `Runnable` into the handoff box. When assigned a request, the internal thread removes it from `handoffBox` and casts it down from `Object` to `Runnable` (line 59). The internal thread then passes the task to the `runIt()` method.

The `private` method `runIt()` (lines 70–84) takes the `Runnable` passed (line 70) and invokes its `run()` method (line 72). If any exceptions slip through—especially `RuntimeExceptions` such as `NullPointerException` that can occur unexpectedly just about anywhere—they are caught to protect the worker thread (line 73). Instances of `Error` (and its subclasses, such as `OutOfMemoryError`) will break the worker, but all instances of `Exception` (and its subclasses) will be safely caught. If one is caught, a message and a stack trace are printed to the console (lines 75–76). Regardless of how the internal thread returns from `run()`, the `finally` clause

(lines 77–83) ensures that the thread's interrupted flag is cleared (line 82) before returning to `runWork()`. This is important because if the flag comes back set, and `noStopRequested` is still `true`, an erroneous `InterruptedException` will be thrown by the `remove()` method on line 59.

If the interrupted flag was set by `stopRequest()`, no harm will be done by clearing it. This is because, after `runIt()` returns (line 63), the very next action is a check of the `noStopRequested` flag (line 49). Because `stopRequest()` sets this `false`, `runWork()` will return (line 25), and the worker thread will die quietly as requested. I give a full explanation of `stopRequest()` and `isAlive()` in Chapter 11.

`ThreadPoolMain`, shown in Listing 13.3, is used to demonstrate how `ThreadPool` can be used to run several tasks and then recycle its threads to run more tasks.

LISTING 13.3 ThreadPoolMain.java—Used to Demonstrate ThreadPool

```
 1: public class ThreadPoolMain extends Object {
 2:
 3:     public static Runnable makeRunnable(
 4:                 final String name,
 5:                 final long firstDelay
 6:             ) {
 7:
 8:         return new Runnable() {
 9:                 public void run() {
10:                     try {
11:                         System.out.println(name +": starting up");
12:                         Thread.sleep(firstDelay);
13:                         System.out.println(
                                    ➥name + ": doing some stuff");
14:                         Thread.sleep(2000);
15:                         System.out.println(name + ": leaving");
16:                     } catch ( InterruptedException ix ) {
17:                         System.out.println(
                                      ➥name + ": got interrupted!");
18:                         return;
19:                     } catch ( Exception x ) {
20:                         x.printStackTrace();
21:                     }
22:                 }
23:
24:                 public String toString() {
25:                     return name;
26:                 }
27:             };
28:     }
29:
30:     public static void main(String[] args) {
```

```
31:         try {
32:             ThreadPool pool = new ThreadPool(3);
33:
34:             Runnable ra = makeRunnable(“RA”, 3000);
35:             pool.execute(ra);
36:
37:             Runnable rb = makeRunnable(“RB”, 1000);
38:             pool.execute(rb);
39:
40:             Runnable rc = makeRunnable(“RC”, 2000);
41:             pool.execute(rc);
42:
43:             Runnable rd = makeRunnable(“RD”, 60000);
44:             pool.execute(rd);
45:
46:             Runnable re = makeRunnable(“RE”, 1000);
47:             pool.execute(re);
48:
49:             pool.stopRequestIdleWorkers();
50:             Thread.sleep(2000);
51:             pool.stopRequestIdleWorkers();
52:
53:             Thread.sleep(5000);
54:             pool.stopRequestAllWorkers();
55:         } catch ( InterruptedException ix ) {
56:             ix.printStackTrace();
57:         }
58:     }
59: }
```


`ThreadPoolMain` creates five `Runnable` objects and passes them to the `execute()` method of `ThreadPool`. The `static` method `makeRunnable()` (lines 3–28) is used to manufacture `Runnable` objects that are similar. It takes two parameters, the first being the `name` to use in output messages to differentiate the `Runnable` from the others (line 4). The second is the number of milliseconds to wait between printing the first and second messages (line 5). These two parameters are declared `final` so that they can be accessed from the anonymous inner class that is created (lines 8-27).

The `Runnable` interface is implemented on-the-fly. The two methods that are defined are `toString()` (lines 24–26) and `run()` (lines 9–22). The `toString()` method simply prints out `name`. The `run()` method prints several messages, all of which include the `name` to clarify the output (lines 11, 13, 15, and 17). The delay factor passed in is used to control the length of the first `sleep()` (line 12). If either `sleep()` is interrupted, a message is printed and the method returns (lines 16–18). If any other exception occurs, a stack trace is printed and the method returns (lines 19–21).

In `main()`, a `ThreadPool` object is constructed with the specification that it should create `3` instances of `ThreadPoolWorker` (line 32). The `makeRunnable()` method is invoked `5` times, and the results of each are passed to the `execute()` method (lines 34–47). All `5` will not be able to run at the same time because the pool has only `3` workers. The fourth and fifth calls to `execute()` will block briefly until a worker becomes available. After all `5` have been started (and at least `2` will have finished), the `stopRequestIdleWorkers()` method is invoked (line 49) on the pool to remove and shut down any and all workers that are currently not processing a request. After `2` seconds (line 50), another request is issued to stop all idle workers (line 51). After an additional `5` seconds have elapsed, the `stopRequestAllWorkers()` method is called to shut down any and all remaining workers, regardless of whether they are currently busy servicing a request (line 54).

Listing 13.4 shows possible output from running `ThreadPoolMain`. Your output should differ a bit because of the whims of the thread scheduler.

LISTING 13.4 Possible Output from ThreadPoolMain

```
 1: workerID=0, ready for work
 2: workerID=2, ready for work
 3: workerID=1, ready for work
 4: workerID=0, starting execution of new Runnable: RA
 5: RA: starting up
 6: workerID=2, starting execution of new Runnable: RB
 7: RB: starting up
 8: workerID=1, starting execution of new Runnable: RC
 9: RC: starting up
10: RB: doing some stuff
11: RC: doing some stuff
12: RA: doing some stuff
13: RB: leaving
14: workerID=2, ready for work
15: workerID=2, starting execution of new Runnable: RD
16: RD: starting up
17: RC: leaving
18: workerID=1, ready for work
19: workerID=1, starting execution of new Runnable: RE
20: RE: starting up
21: RA: leaving
22: workerID=0, ready for work
23: RE: doing some stuff
24: workerID=0, stopRequest() received.
25: RE: leaving
26: workerID=1, ready for work
27: workerID=1, stopRequest() received.
28: workerID=2, stopRequest() received.
29: RD: got interrupted!
```

Notice that the workers add themselves to the idle list in just about any order (output lines 1–3). However, the tasks are started in the requested order (lines 4–9). When the `RB` task is done (line 13), the worker that was running it, `2`, adds itself back to the idle queue (line 14). Task `RD` was blocked inside `execute()`, waiting for a worker to become available. As soon as `2` puts itself on the idle queue, it is recycled and removed to run task `RD` (line 15). When worker `1` finishes running task `RC` (line 17), it is recycled to run task `RE` (lines 18–19). Next, worker `0` finishes task `RA` and adds itself to the idle queue (line 22).

The first request to stop the currently idle threads gets idle worker `0` to stop (line 24). The next request gets idle worker `1` to stop. Task `RD` was started with a `60`-second delay and is still running. When the request to stop all the threads comes in (line 28), task `RD` is interrupted during its long sleep (line 29), but then returns to allow the thread to die.

A Specialized Worker Thread Pool: HttpServer

In this section, I'll show you how a simple Web page server can utilize thread-pooling techniques to service requests. In this case, the workers are specialized to handle requests for files from Web browsers.

Web browsers and Web servers communicate with each other using the Hypertext Transfer Protocol (HTTP). HTTP 1.0 (older, but simpler for this example than HTTP 1.1) is fully specified in RFC 1945, which is available at this URL:

```
http://www.w3.org/Protocols/rfc1945/rfc1945
```

The basics of this protocol consist of a *request* from the Web browser client, and a *response* from the Web server. The communication occurs over the `InputStream` and `OutputStream` pair available from a TCP/IP socket. The socket connection is initiated by the client and accepted by the server. The request-response cycle occurs while the socket is open. After the response is sent, the socket is closed. Each request uses a new socket. The client Web browser may make several simultaneous requests to a single server, each over its own socket.

The request consists of a required request line, followed by optional header lines, followed by a required blank line, followed by an optional message body. In this example, only the request line will be parsed. The request line consists of a request method, a space, the requested resource, a space, and finally the HTTP protocol version being used by the client. The only request method supported here is `GET`, so a sample request line would be

```
GET /dir1/dir2/file.html HTTP/1.0
```

The response consists of a required status line, followed by optional header lines, followed by a required blank line, followed by an optional message body. In this example, if the file is found, the server will return the status line, one header line with the content length, another

header line with the content type, and a message body with the bytes of the requested file. The status line consists of the HTTP protocol version, a space, a response code, a space, and finally a textual explanation of the response code. In response to a `GET` request, a response such as the following would be produced:

```
HTTP/1.0 200 OK
Content-Length: 1967
Content-Type: text/html
<blank line>
<the 1,967 bytes of the requested file>
```

This simple Web server supports three response status lines:

```
HTTP/1.0 200 OK
HTTP/1.0 404 Not Found
HTTP/1.0 503 Service Unavailable
```

The first is used when the requested file is found, the second if the file could not be found, and the third if the server is too busy to service the request properly.

Class HttpServer

The `HttpServer` class, shown in Listing 13.5, serves as the main interface to the Web server and creates several `HttpWorker` objects (see Listing 13.6). The `HttpServer` object and the `HttpWorker` objects each have their own internal thread. The workers add themselves to a pool when they are ready to accept another HTTP request. When a request comes in, the server checks the pool for available workers and if one is available, assigns it to the connection. If none are available, the terse `Service Unavailable` response is returned to the client.

LISTING 13.5 HttpServer.java—A Simple Web Page Server

```
 1: import java.io.*;
 2: import java.net.*;
 3:
 4: // uses ObjectFIFO from chapter 18
 5:
 6: public class HttpServer extends Object {
 7:
 8:     // currently available HttpWorker objects
 9:     private ObjectFIFO idleWorkers;
10:
11:     // all HttpWorker objects
12:     private HttpWorker[] workerList;
13:     private ServerSocket ss;
14:
15:     private Thread internalThread;
```

```
16:     private volatile boolean noStopRequested;
17:
18:     public HttpServer(
19:                 File docRoot,
20:                 int port,
21:                 int numberOfWorkers,
22:                 int maxPriority
23:             ) throws IOException {
24:
25:         // Allow a max of 10 sockets to queue up
26:         // waiting for accpet().
27:         ss = new ServerSocket(port, 10);
28:
29:         if ( ( docRoot == null ) ||
30:              !docRoot.exists() ||
31:              !docRoot.isDirectory()
32:            ) {
33:
34:             throw new IOException("specified docRoot is null " +
35:                 "or does not exist or is not a directory");
36:         }
37:
38:         // ensure that at least one worker is created
39:         numberOfWorkers = Math.max(1, numberOfWorkers);
40:
41:         // Ensure:
42:         // (minAllowed + 2) <= serverPriority <= (maxAllowed - 1)
43:         // which is generally:
44:         //   3 <= serverPriority <= 9
45:         int serverPriority = Math.max(
46:                 Thread.MIN_PRIORITY + 2,
47:                 Math.min(maxPriority, Thread.MAX_PRIORITY - 1)
48:             );
49:
50:         // Have the workers run at a slightly lower priority so
51:         // that new requests are handled with more urgency than
52:         // in-progress requests.
53:         int workerPriority = serverPriority - 1;
54:
55:         idleWorkers = new ObjectFIFO(numberOfWorkers);
56:         workerList = new HttpWorker[numberOfWorkers];
57:
58:         for ( int i = 0; i < numberOfWorkers; i++ ) {
59:             // Workers get a reference to the FIFO to add
```

continues

LISTING 13.5 Continued

```
60:             // themselves back in when they are ready to
61:             // handle a new request.
62:             workerList[i] = new HttpWorker(
63:                         docRoot, workerPriority, idleWorkers);
64:         }
65:
66:         // Just before returning, the thread should be
67:         // created and started.
68:         noStopRequested = true;
69:
70:         Runnable r = new Runnable() {
71:                 public void run() {
72:                     try {
73:                         runWork();
74:                     } catch ( Exception x ) {
75:                         // in case ANY exception slips through
76:                         x.printStackTrace();
77:                     }
78:                 }
79:             };
80:
81:         internalThread = new Thread(r);
82:         internalThread.setPriority(serverPriority);
83:         internalThread.start();
84:     }
85:
86:     private void runWork() {
87:         System.out.println(
88:                 “HttpServer ready to receive requests”);
89:
90:         while ( noStopRequested ) {
91:             try {
92:                 Socket s = ss.accept();
93:
94:                 if ( idleWorkers.isEmpty() ) {
95:                     System.out.println(
96:                         “HttpServer too busy, denying request”);
97:
98:                     BufferedWriter writer =
99:                         new BufferedWriter(
100:                            new OutputStreamWriter(
101:                                s.getOutputStream()));
102:
103:                    writer.write(“HTTP/1.0 503 Service “ +
```

```
104:                                    "Unavailable\r\n\r\n");
105:
106:                     writer.flush();
107:                     writer.close();
108:                     writer = null;
109:                 } else {
110:                     // No need to be worried that idleWorkers
111:                     // will suddenly be empty since this is the
112:                     // only thread removing items from the queue.
113:                     HttpWorker worker =
114:                             (HttpWorker) idleWorkers.remove();
115:
116:                     worker.processRequest(s);
117:                 }
118:             } catch ( IOException iox ) {
119:                 if ( noStopRequested ) {
120:                     iox.printStackTrace();
121:                 }
122:             } catch ( InterruptedException x ) {
123:                 // re-assert interrupt
124:                 Thread.currentThread().interrupt();
125:             }
126:         }
127:     }
128:
129:     public void stopRequest() {
130:         noStopRequested = false;
131:         internalThread.interrupt();
132:
133:         for ( int i = 0; i < workerList.length; i++ ) {
134:             workerList[i].stopRequest();
135:         }
136:
137:         if ( ss != null ) {
138:             try { ss.close(); } catch ( IOException iox ) { }
139:             ss = null;
140:         }
141:     }
142:
143:     public boolean isAlive() {
144:         return internalThread.isAlive();
145:     }
146:
```

continues

LISTING 13.5 Continued

```
147:     private static void usageAndExit(String msg, int exitCode) {
148:         System.err.println(msg);
149:         System.err.println("Usage: java HttpServer <port> " +
150:                            "<numWorkers> <documentRoot>");
151:         System.err.println("   <port> - port to listen on " +
152:                            "for HTTP requests");
153:         System.err.println("   <numWorkers> - number of " +
154:                            "worker threads to create");
155:         System.err.println("   <documentRoot> - base " +
156:                            "directory for HTML files");
157:         System.exit(exitCode);
158:     }
159:
160:     public static void main(String[] args) {
161:         if ( args.length != 3 ) {
162:             usageAndExit("wrong number of arguments", 1);
163:         }
164:
165:         String portStr = args[0];
166:         String numWorkersStr = args[1];
167:         String docRootStr = args[2];
168:
169:         int port = 0;
170:
171:         try {
172:             port = Integer.parseInt(portStr);
173:         } catch ( NumberFormatException x ) {
174:             usageAndExit("could not parse port number from '" +
175:                     portStr + "'", 2);
176:         }
177:
178:         if ( port < 1 ) {
179:             usageAndExit("invalid port number specified: " +
180:                     port, 3);
181:         }
182:
183:         int numWorkers = 0;
184:
185:         try {
186:             numWorkers = Integer.parseInt(numWorkersStr);
187:         } catch ( NumberFormatException x ) {
188:             usageAndExit(
189:                     "could not parse number of workers from '" +
190:                     numWorkersStr + "'", 4);
```

```
191:         }
192:
193:         File docRoot = new File(docRootStr);
194:
195:         try {
196:             new HttpServer(docRoot, port, numWorkers, 6);
197:         } catch ( IOException x ) {
198:             x.printStackTrace();
199:             usageAndExit("could not construct HttpServer", 5);
200:         }
201:     }
202: }
```

`HttpServer` keeps a pool of idle workers in `idleWorkers` by using an `ObjectFIFO` (line 9). In addition, it keeps a list of all the `HttpWorker` objects it created in an array (line 12). It also uses the self-running object pattern shown in Chapter 11.

The constructor (lines 18–84) takes four parameters. The `docRoot` parameter (line 19) is a `File` referring to the directory on the server machine that is the base directory for all HTTP file requests. The `port` parameter (line 20) is the TCP/IP port that the Web server will be listening to for new sockets (requests). The `numberOfWorkers` parameter (line 21) indicates the number of `HttpWorker` objects that should be created to service requests. The `maxPriority` parameter (line 22) is used to indicate the thread priority for the thread running inside `HttpServer`. The threads running inside the `HttpWorker` objects will run at a slightly lower priority: `(maxPriority - 1)`.

Inside the constructor, a `ServerSocket` is created to listen on `port` (line 27). It also specifies that the VM should accept up to `10` sockets more than what has been returned from the `accept()` method of `ServerSocket`. If there are any problems setting up the `ServerSocket`, an `IOException` will be thrown and will propagate out of the constructor (line 23). The `docRoot` parameter is then checked to be sure that it refers to an existing file and that it is also a directory (lines 29–36). The `numberOfWorkers` parameter is silently increased to `1` if it was less than that before (line 39). The priorities for the server thread and the worker threads are silently forced into a valid range (lines 45–53), so that

```
Thread.MIN_PRIORITY < workerPriority <
                                ➥serverPriority < Thread.MAX_PRIORITY
```

where `workerPriority` is just `1` less than `serverPriority`.

An `ObjectFIFO` is created with enough capacity to hold all the workers (line 55). A list of all the workers, idle or busy, is created (line 56), and each of the `HttpWorker` objects is constructed and added to this list (lines 58–64). Each `HttpWorker` is passed a reference to the `idleWorkers` FIFO queue so that it can add itself to the queue when it is ready to process a new request.

The rest of the constructor (lines 68–83) follows the pattern in Chapter 11, with just one minor addition: The priority of the internal thread is set before it is started (line 82).

The `runWork()` method (lines 86–127) is invoked by the internal thread. As long as no stop has been requested (line 90), the method continues to accept new sockets (line 92). When a socket is accepted, `runWork()` checks whether any idle workers are available to process the request (line 94). If the pool is empty, the request is denied, and a minimal response is created and sent back over the socket connection (lines 98–108). In HTTP message headers, an end-of-line is marked by a carriage-return, line-feed pair: `"\r\n"`.

If an idle worker is found in the pool, it is removed (lines 113–114). The `processRequest()` method of that `HttpWorker` is invoked, and the socket that was just accepted is passed to it for servicing (line 116). If an `IOException` occurs in the process of accepting a socket and handing it off to be processed, and no stop has been requested, the exception will have its stack trace dumped (lines 118–121). If an `InterruptedException` occurs, it is caught, and the interrupt is reasserted (lines 122–124).

The `stopRequest()`method (lines 129–141) follows the pattern of Chapter 11, but adds another two steps. After signaling the internal thread to stop, it invokes `stopRequest()` on each of the `HttpWorker` objects. Because the internal thread may be blocked on the `accept()` method of `ServerSocket` (line 92), steps have to be taken to unblock it. It does *not* respond to being interrupted, so if the internal thread is blocked on `accept()`, the `interrupt()` call (line 131) is ineffective. To unblock the `accept()` method, the `ServerSocket` is closed (line 138), which causes `accept()` to throw an `IOException`. This `IOException` is caught (line 118), and if a stop has been requested, the exception is ignored. In this case, a stop was requested, so the exception thrown by forcibly closing the `ServerSocket` is silently ignored. This technique can be used to unblock various I/O methods that do not respond to interrupts. I explain it in detail in Chapter 15, "Breaking Out of a Blocked I/O State."

The `static` method `usageAndExit()` (lines 147–158) assists `main()` in reporting command-line mistakes and printing the proper command usage. For example, if `HttpServer` is run with no command-line arguments,

```
java HttpServer
```

the following output is produced:

```
wrong number of arguments
Usage: java HttpServer <port> <numWorkers> <documentRoot>
   <port> - port to listen on for HTTP requests
   <numWorkers> - number of worker threads to create
   <documentRoot> - base directory for HTML files
```

After the error message (line 148) and the usage lines (lines 149–156) are printed, `usageAndExit()` causes the VM to exit with the exit code that was passed to it (line 157).

The `main()` method (lines 160–201) is used to parse and validate the command-line options and to construct an `HttpServer` instance. `HttpServer` can simply be used as a class in a larger application, or it can be run as its own application using `main()`. First, the port number passed on the command line is converted to an `int` (lines 169–176) and is checked to be a positive number (lines 178–181). If either step fails, `usageAndExit()` is used to halt the application with an appropriate message. Second, the number of `HttpWorker` objects to create is parsed and validated (lines 183–191). Third, the document root directly passed on the command line is converted into a platform-independent `File` object (line 193). Finally, an attempt is made to construct an `HttpServer` object with these parameters and a maximum thread priority of `6` (line 196).

The `HttpServer` object's internal thread will run at a priority of `6`, and each of the `HttpWorker` objects' internal threads will run at a priority of `5` (line 53). Constructing an `HttpServer` object might throw an `IOException`, especially if the port is already in use. If this or another problem occurs, a message is printed and the VM exits (lines 197–199). If all goes well, the constructor returns after starting the internal thread, and the `main()` method completes.

Class HttpWorker

`HttpWorker` objects, shown in Listing 13.6, are used as the specialized pool of threaded resources accessed from `HttpServer`. `HttpWorker` is similar to `ThreadPoolWorker` (refer to Listing 13.2), and I'll point out only the major differences.

LISTING 13.6 HttpWorker.java—The Helper Class for HttpServer

```
 1: import java.io.*;
 2: import java.net.*;
 3: import java.util.*;
 4:
 5: // uses class ObjectFIFO from chapter 18
 6:
 7: public class HttpWorker extends Object {
 8:     private static int nextWorkerID = 0;
 9:
10:     private File docRoot;
11:     private ObjectFIFO idleWorkers;
12:     private int workerID;
13:     private ObjectFIFO handoffBox;
14:
15:     private Thread internalThread;
16:     private volatile boolean noStopRequested;
```

continues

LISTING 13.6 Continued

```
17:
18:     public HttpWorker(
19:                 File docRoot,
20:                 int workerPriority,
21:                 ObjectFIFO idleWorkers
22:             ) {
23:
24:         this.docRoot = docRoot;
25:         this.idleWorkers = idleWorkers;
26:
27:         workerID = getNextWorkerID();
28:         handoffBox = new ObjectFIFO(1); // only one slot
29:
30:         // Just before returning, the thread should be
31:         // created and started.
32:         noStopRequested = true;
33:
34:         Runnable r = new Runnable() {
35:                 public void run() {
36:                     try {
37:                         runWork();
38:                     } catch ( Exception x ) {
39:                         // in case ANY exception slips through
40:                         x.printStackTrace();
41:                     }
42:                 }
43:             };
44:
45:         internalThread = new Thread(r);
46:         internalThread.setPriority(workerPriority);
47:         internalThread.start();
48:     }
49:
50:     public static synchronized int getNextWorkerID() {
51:         // synchronized at the class level to ensure uniqueness
52:         int id = nextWorkerID;
53:         nextWorkerID++;
54:         return id;
55:     }
56:
57:     public void processRequest(Socket s)
58:                 throws InterruptedException {
59:
60:         handoffBox.add(s);
```

```
61:     }
62:
63:     private void runWork() {
64:         Socket s = null;
65:         InputStream in = null;
66:         OutputStream out = null;
67:
68:         while ( noStopRequested ) {
69:             try {
70:                 // Worker is ready to receive new service
71:                 // requests, so it adds itself to the idle
72:                 // worker queue.
73:                 idleWorkers.add(this);
74:
75:                 // Wait here until the server puts a request
76:                 // into the handoff box.
77:                 s = (Socket) handoffBox.remove();
78:
79:                 in = s.getInputStream();
80:                 out = s.getOutputStream();
81:                 generateResponse(in, out);
82:                 out.flush();
83:             } catch ( IOException iox ) {
84:                 System.err.println(
85:                     “I/O error while processing request, “ +
86:                     “ignoring and adding back to idle “ +
87:                     “queue - workerID=” + workerID);
88:             } catch ( InterruptedException x ) {
89:                 // re-assert the interrupt
90:                 Thread.currentThread().interrupt();
91:             } finally {
92:                 // Try to close everything, ignoring
93:                 // any IOExceptions that might occur.
94:                 if ( in != null ) {
95:                     try {
96:                         in.close();
97:                     } catch ( IOException iox ) {
98:                         // ignore
99:                     } finally {
100:                        in = null;
101:                    }
102:                }
103:
104:                if ( out != null ) {
```

continues

LISTING 13.6 Continued

```
105:                     try {
106:                         out.close();
107:                     } catch ( IOException iox ) {
108:                         // ignore
109:                     } finally {
110:                         out = null;
111:                     }
112:                 }
113:
114:                 if ( s != null ) {
115:                     try {
116:                         s.close();
117:                     } catch ( IOException iox ) {
118:                         // ignore
119:                     } finally {
120:                         s = null;
121:                     }
122:                 }
123:             }
124:         }
125:     }
126:
127:     private void generateResponse(
128:                 InputStream in,
129:                 OutputStream out
130:             ) throws IOException {
131:
132:         BufferedReader reader =
133:                 new BufferedReader(new InputStreamReader(in));
134:
135:         String requestLine = reader.readLine();
136:
137:         if ( ( requestLine == null ) ||
138:              ( requestLine.length() < 1 )
139:            ) {
140:
141:             throw new IOException("could not read request");
142:         }
143:
144:         System.out.println("workerID=" + workerID +
145:                 ", requestLine=" + requestLine);
146:
147:         StringTokenizer st = new StringTokenizer(requestLine);
148:         String filename = null;
```

```
149:
150:        try {
151:            // request method, typically 'GET', but ignored
152:            st.nextToken();
153:
154:            // the second token should be the filename
155:            filename = st.nextToken();
156:        } catch ( NoSuchElementException x ) {
157:            throw new IOException(
158:                    "could not parse request line");
159:        }
160:
161:        File requestedFile = generateFile(filename);
162:
163:        BufferedOutputStream buffOut =
164:                new BufferedOutputStream(out);
165:
166:        if ( requestedFile.exists() ) {
167:            System.out.println("workerID=" + workerID +
168:                    ", 200 OK: " + filename);
169:
170:            int fileLen = (int) requestedFile.length();
171:
172:            BufferedInputStream fileIn =
173:                new BufferedInputStream(
174:                    new FileInputStream(requestedFile));
175:
176:            // Use this utility to make a guess obout the
177:            // content type based on the first few bytes
178:            // in the stream.
179:            String contentType =
180:                URLConnection.guessContentTypeFromStream(
181:                    fileIn);
182:
183:            byte[] headerBytes = createHeaderBytes(
184:                    "HTTP/1.0 200 OK",
185:                    fileLen,
186:                    contentType
187:                );
188:
189:            buffOut.write(headerBytes);
190:
191:            byte[] buf = new byte[2048];
```

continues

LISTING 13.6 Continued

```
            int blockLen = 0;

            while ( ( blockLen = fileIn.read(buf) ) != -1 ) {
                buffOut.write(buf, 0, blockLen);
            }

            fileIn.close();
        } else {
            System.out.println("workerID=" + workerID +
                    ", 404 Not Found: " + filename );

            byte[] headerBytes = createHeaderBytes(
                    "HTTP/1.0 404 Not Found",
                    -1,
                    null
                );

            buffOut.write(headerBytes);
        }

        buffOut.flush();
    }

    private File generateFile(String filename) {
        File requestedFile = docRoot; // start at the base

        // Build up the path to the requested file in a
        // platform independent way. URL's use '/' in their
        // path, but this platform may not.
        StringTokenizer st = new StringTokenizer(filename, "/");
        while ( st.hasMoreTokens() ) {
            String tok = st.nextToken();

            if ( tok.equals("..") ) {
                // Silently ignore parts of path that might
                // lead out of the document root area.
                continue;
            }

            requestedFile =
                new File(requestedFile, tok);
        }

        if ( requestedFile.exists() &&
```

```
236:              requestedFile.isDirectory()
237:            ) {
238:
239:             // If a directory was requested, modify the request
240:             // to look for the “index.html” file in that
241:             // directory.
242:             requestedFile =
243:                 new File(requestedFile, “index.html”);
244:         }
245:
246:         return requestedFile;
247:     }
248:
249:     private byte[] createHeaderBytes(
250:                 String resp,
251:                 int contentLen,
252:                 String contentType
253:             ) throws IOException {
254:
255:         ByteArrayOutputStream baos = new ByteArrayOutputStream();
256:         BufferedWriter writer = new BufferedWriter(
257:                 new OutputStreamWriter(baos));
258:
259:         // Write the first line of the response, followed by
260:         // the RFC-specified line termination sequence.
261:         writer.write(resp + “\r\n”);
262:
263:         // If a length was specified, add it to the header
264:         if ( contentLen != -1 ) {
265:             writer.write(
266:                 “Content-Length: “ + contentLen + “\r\n”);
267:         }
268:
269:         // If a type was specified, add it to the header
270:         if ( contentType != null ) {
271:             writer.write(
272:                 “Content-Type: “ + contentType + “\r\n”);
273:         }
274:
275:         // A blank line is required after the header.
276:         writer.write(“\r\n”);
277:         writer.flush();
278:
```

continues

LISTING 13.6 Continued

```
279:         byte[] data = baos.toByteArray();
280:         writer.close();
281:
282:         return data;
283:     }
284:
285:     public void stopRequest() {
286:         noStopRequested = false;
287:         internalThread.interrupt();
288:     }
289:
290:     public boolean isAlive() {
291:         return internalThread.isAlive();
292:     }
293: }
```

The constructor of `HttpWorker` (lines 18–48) is passed `docRoot`, the base directory for files (line 19), the priority to use for the internal thread (line 20), and a reference to the idle worker pool. The internal thread priority is set just before the thread is started (line 46). A one-slot handoff box is created (line 28) for passing the socket accepted in the `HttpServer` thread to this worker's internal thread.

The `processRequest()` method (lines 57–61) is invoked by the `HttpServer` object and puts the reference to the new socket into the handoff box. It should never block because a worker will add itself to the idle pool only when it is ready to process a new request. Also, the server will invoke the `processRequest()` methods only on workers it just removed from the idle pool.

The `runWork()` method (lines 63–125) is where the internal thread does the bulk of the processing. Basically, the internal thread loops until a stop is requested (lines 68–124). Each time through, the worker adds itself to the server's list of idle workers (line 73). It then blocks, indefinitely waiting for an assignment to be dropped off in its handoff box (line 77). When a `Socket` reference is picked up, the worker retrieves the `InputStream` and `OutputStream` from the reference and passes them to the `generateResponse()` method (lines 79–81). After `generateResponse()` has read the request from the `InputStream` and written the proper response to the `OutputStream`, the `OutputStream` is flushed to ensure that the data is pushed through any buffering (line 82). If an `IOException` occurs during any of this, a message is printed (lines 83–87). Whether or not an `IOException` occurs, the `finally` clause is used to ensure that the streams and socket are closed properly (lines 91–112). Next, the `while` expression is re-evaluated (line 68). If no stop request has been made, the worker adds itself back to the idle queue and waits for its next assignment.

The `generateResponse()` method (lines 127–213) is used to analyze the HTTP request on the `InputStream` and generate an appropriate HTTP response on the `OutputStream`. The raw `OutputStream` from the socket is wrapped in a `BufferedOutputStream` to data transfer efficiently (lines 163–164). The raw `InputStream` from the socket is wrapped in a `BufferedReader` (lines 132–133). The first line is read from the request (line 135–142) and broken up into tokens to pick out the filename (lines 147–159). The `generateFile()` method is used to create a new `File` instance that refers to the requested file (line 161).

Inside `generateFile()` (lines 215–247), the request is parsed on the `'/'` character to build up a new `File` object starting from `docRoot`. When the parsing is complete, a check is done to see if the requested file is a directory (lines 235–236). If a directory was requested, `"index.html"` is appended as the default file to return from a directory (lines 242–243).

If the requested file is found, it is sent back to the client (lines 166–198). First, the length of the file in bytes is determined for the response header (line 170). Next, the content type for the file is guessed by the use of the `static` method `guessContentTypeFromStream()` on the class `java.net.URLConnection` (lines 179–181). Then the `createHeaderBytes()` method (see the following description) is used to format the response header and convert it to an array of bytes (lines 183–187). This header information is written to the stream (line 189), followed by the contents of the file (lines 191–198).

If the requested file is not found, a brief response indicating that this is sent back to the client (lines 199–209). The `createHeaderBytes()` method is used to format a `404 Not Found` response (lines 203–207). These bytes are written back to the client (line 209).

The `createHeaderBytes()` method (lines 249–283) is used to format a response header and write it to an array of bytes. The `resp` string passed in is written as is (lines 255–261). If a valid content length is passed in, the `Content-Length:` header field is appended (lines 264–267). If a content type is specified, the `Content-Type:` header field is appended (lines 270–273). All headers end with a blank line (line 276). The resulting `byte[]` is passed back to the caller (lines 185–188).

Sample Files to Be Served

To demonstrate this simple Web server, I created a directory named `htmldir`, with the following files and subdirectory:

```
./htmldir/index.html
./htmldir/images/five.gif
./htmldir/images/four.gif
./htmldir/images/one.gif
./htmldir/images/three.gif
./htmldir/images/two.gif
```

The `index.html` file is the main file served, and it makes references to the graphics stored in the `images` subdirectory. When the server is launched, `htmldir` will be used as the document root directory.

The `index.html` file, shown in Listing 13.7, uses the Hypertext Markup Language (HTML) to specify the layout of a Web page. You can read more about it at this URL:

`http://www.w3.org/TR/REC-html32.html`

LISTING 13.7 index.html—An HTML File for Demonstrating HttpServer

```
 1: <html>
 2: <head><title>Thread Pooling - 1</title></head>
 3: <body bgcolor="#FFFFFF">
 4: <center>
 5: <table border="2" cellspacing="5" cellpadding="2">
 6: <tr>
 7:   <td valign="top"><img src="./images/one.gif"></td>
 8:   <td>Thread pooling helps to save the VM the work of creating and
 9:       destroying threads when they can be easily recycled.</td>
10: </tr>
11: <tr>
12:     <td valign="top"><img src="./images/twoDOESNOTEXIST.gif"></td>
13:     <td>Thread pooling reduces response time because the worker
14:         thread is already created, started, and running. It's only
15:         only waiting for the signal to <b><i>go</i></b>!</td>
16: </tr>
17: <tr>
18:   <td valign="top"><img src="./images/three.gif"></td>
19:   <td>Thread pooling holds resource usage to a predetermined upper
20:       limit. Instead of starting a new thread for every request
21:       received by an HTTP server, a set of workers is available to
22:       service requests. When this set is being completely used by
23:       other requests, the server does not increase its load, but
24:       rejects requests until a worker becomes available.</td>
25: </tr>
26: <tr>
27:     <td valign="top"><img src="./images/four.gif"></td>
28:     <td>Thread pooling generally works best when a thread is
29:         needed for only a brief period of time.</td>
30: </tr>
31: <tr>
32:     <td valign="top"><img src="./images/five.gif"></td>
33:     <td>When using the thread pooling technique, care must
34:         be taken to reasonably ensure that threads don't become
35:         deadlocked or die.<td>
36: </tr>
```

```
37: </table>
38: </body>
39: </html>
```

This HTML file makes references to `5` images that will subsequently be requested by the Web browser. All but one of them exists. Instead of `two.gif`, `twoDOESNOTEXIST.gif` is requested (line 12) to cause the simple server to generate a `404 Not Found` response.

Running HttpServer with 3 Workers

First, I'll show you what happens when the `HttpServer` application is run with only `3` workers in the pool:

```
java HttpServer 2001 3 htmldir
```

The Web server will be listening on port `2001` for connections. If you can't use this port on your system, specify a different one. (Generally, Web servers listen to port `80`, but on some systems, only privileged users can run processes that listen to ports less than `1024`.) The second command-line argument indicates that `3` workers should be created to service requests. The third argument is the directory where the HTML files are located. On my machine they are in `htmldir`, which is a subdirectory of the current directory. You can use a fully qualified pathname if necessary.

Possible output from this simple Web server application is shown in Listing 13.8.

LISTING 13.8 Possible Output from HttpServer with 3 Threads

```
 1: HttpServer ready to receive requests
 2: workerID=0, requestLine=GET / HTTP/1.0
 3: workerID=0, 200 OK: /index.html
 4: HttpServer too busy, denying request
 5: HttpServer too busy, denying request
 6: workerID=1, requestLine=GET /images/one.gif HTTP/1.0
 7: workerID=0, requestLine=GET /images/five.gif HTTP/1.0
 8: workerID=2, requestLine=GET /images/twoDOESNOTEXIST.gif HTTP/1.0
 9: workerID=1, 200 OK: /images/one.gif
10: workerID=0, 200 OK: /images/five.gif
11: workerID=2, 404 Not Found: /images/twoDOESNOTEXIST.gif
12: workerID=1, requestLine=GET / HTTP/1.0
13: workerID=1, 200 OK: /index.html
14: workerID=0, requestLine=GET /images/twoDOESNOTEXIST.gif HTTP/1.0
15: workerID=2, requestLine=GET /images/three.gif HTTP/1.0
16: workerID=1, requestLine=GET /images/four.gif HTTP/1.0
17: workerID=0, 404 Not Found: /images/twoDOESNOTEXIST.gif
18: workerID=2, 200 OK: /images/three.gif
19: workerID=1, 200 OK: /images/four.gif
```

When the server is up and running and ready to received requests, it prints a message (line 1). I used Netscape 4.5 to request the following URL:

```
http://localhost:2001/
```

NOTE

The hostname `localhost` is used in TCP/IP networking to generically refer to the current machine. If you don't have `localhost` defined on your system, you can supply another hostname or an IP address. Additionally, you can consider adding the following line to your machine's `hosts` file (it may be `/etc/hosts`, `C:\windows\hosts`, or something else):

```
127.0.0.1   localhost
```

The Web browser requests a connection on port `2001` of `localhost`. Then, it asks for filename `/`. This request is handed off to worker `0` (line 2). The worker responds with `/index.html` (line 3). When the browser parses this HTML file, it very quickly requests the `5` graphic files referenced. Only `3` workers are available, so when the server is flooded with these requests, it denies `2` of them (lines 4–5). Worker `1` is assigned to `/images/one.gif` (line 6), finds it, and sends it (line 9). Worker `0` is assigned to `/images/five.gif` (line 7), finds it, and sends it (line 10). Worker `2` looks for the nonexistent file `/images/twoDOESNOTEXIST.gif` (line 8), can't find it, and sends back the `404 Not Found` response (line 11).

After this, the browser looks like Figure 13.1. Notice that the graphics for `2`, `3`, and `4` are missing. The graphic for `2` does not exist and looks slightly different than the surrogate images supplied by Netscape for `3` and `4`.

To attempt to get the other graphics to load, I clicked on the Location field in the Web browser and pressed the Enter key to retrieve the page again. Referring back to Listing 13.8, you can see that the `/` file is requested again (line 12) and served back by worker `1` (line 13). This time, only `3` images are requested because the other `2` are already loaded. The `twoDOESNOTEXIST.gif` image is requested again (line 14) by the Web browser in the hope that the image is now there. It is not, and the `404 Not Found` response is sent again (line 17). Workers are available this time to send `three.gif` and `four.gif` (lines 15–16 and 18–19).

After this second try, the browser looks like Figure 13.2. Notice that everything is drawn, except for the missing `twoDOESNOTEXIST.gif` image.

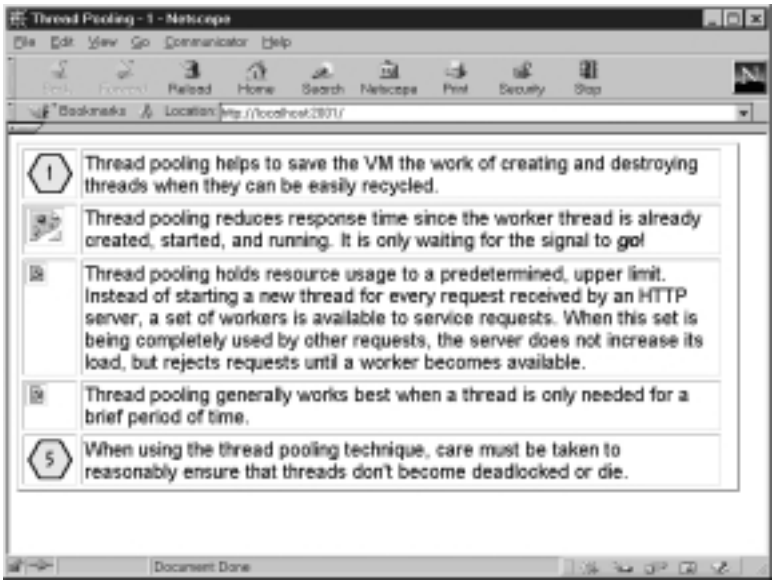

FIGURE 13.1
The first attempt to retrieve the page, with only 3 worker threads.

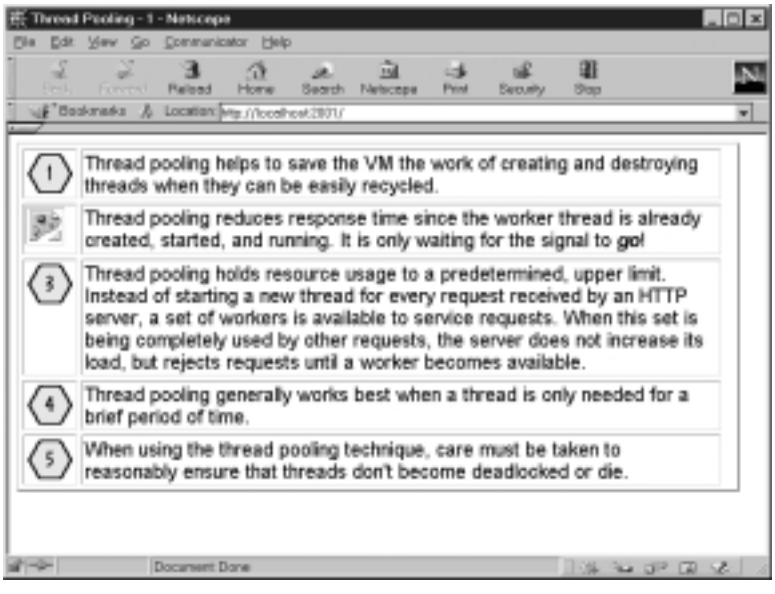

FIGURE 13.2
The second attempt to retrieve the page.

Running HttpServer with 10 Workers

If, instead, `HttpServer` is started with `10` workers,

```
java HttpServer 2001 10 htmldir
```

more resources are available to meet the demands. The Web browser is closed and restarted, and the same URL is requested. This time, the server output is shown in Listing 13.9. The Web browser looks like Figure 13.2 on the first try.

LISTING 13.9 Possible Output from HttpServer with 10 Threads

```
 1: HttpServer ready to receive requests
 2: workerID=0, requestLine=GET / HTTP/1.0
 3: workerID=0, 200 OK: /index.html
 4: workerID=4, requestLine=GET /images/four.gif HTTP/1.0
 5: workerID=1, requestLine=GET /images/one.gif HTTP/1.0
 6: workerID=2, requestLine=GET /images/twoDOESNOTEXIST.gif HTTP/1.0
 7: workerID=3, requestLine=GET /images/three.gif HTTP/1.0
 8: workerID=4, 200 OK: /images/four.gif
 9: workerID=1, 200 OK: /images/one.gif
10: workerID=3, 200 OK: /images/three.gif
11: workerID=2, 404 Not Found: /images/twoDOESNOTEXIST.gif
12: workerID=6, requestLine=GET /images/five.gif HTTP/1.0
13: workerID=6, 200 OK: /images/five.gif
```

With `10` workers running, none of the requests are denied—the simple Web server has idle workers to spare. When you run the server, you are likely to see slightly different ordering of requests and responses.

You can expand on this design to include a short wait before requests are denied. This would allow some time for a worker to finish up and put itself back into the idle queue.

Summary

Thread pooling can be used when a thread is needed for a relatively short time. Thread pooling allows a thread to be assigned to a task and, when the task completes, to be recycled for use in another task. Because threads in the pool are already up and running, response time is usually reduced. The number of threads in the pool can be fixed to an upper limit to prevent a sudden overloading of the application.

I showed you how to create a generic pool of threads to execute the `run()` method of objects that implement the `Runnable` interface. Then, I showed you how to create a specialized pool of threads to service HTTP requests. Both these techniques can be elaborated on to provide more specialized and robust solutions in your code.