Table of Contents

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

Chapter 10. Sockets for Servers

The last chapter discussed sockets from the standpoint of *clients*: programs that open a socket to a server that's listening for connections. However, client sockets themselves aren't enough; clients aren't much use unless they can talk to a server, and the `Socket` class discussed in the last chapter is not sufficient for writing servers. To create a `Socket`, you need to know the Internet host to which you want to connect. When you're writing a server, you don't know in advance who will contact you, and even if you did, you wouldn't know when that host wanted to contact you. In other words, servers are like receptionists who sit by the phone and wait for incoming calls. They don't know who will call or when, only that when the phone rings, they have to pick it up and talk to whoever is there. You can't program that behavior with the `Socket` class alone.

For servers that accept connections, Java provides a `ServerSocket` class that represents server sockets. In essence, a server socket's job is to sit by the phone and wait for incoming calls. More technically, a server socket runs on the server and listens for incoming TCP connections. Each server socket listens on a particular port on the server machine. When a client on a remote host attempts to connect to that port, the server wakes up, negotiates the connection between the client and the server, and returns a regular `Socket` object representing the socket between the two hosts. In other words, server sockets wait for connections while client sockets initiate connections. Once a `ServerSocket` has set up the connection, the server uses a regular `Socket` object to send data to the client. Data always travels over the regular socket.

10.1. The ServerSocket Class

The `ServerSocket` class contains everything needed to write servers in Java. It has constructors that create new `ServerSocket` objects, methods that listen for connections on a specified port, methods that configure the various server socket options, and the usual miscellaneous methods such as `toString( )`.

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

In Java, the basic life cycle of a server program is:

1. A new `ServerSocket` is created on a particular port using a `ServerSocket( )` constructor.
2. The `ServerSocket` listens for incoming connection attempts on that port using its `accept( )` method. `accept( )` blocks until a client attempts to make a connection, at which point `accept( )` returns a `Socket` object connecting the client and the server.
3. Depending on the type of server, either the `Socket`'s `getInputStream()` method, `getOutputStream( )` method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.
6. The server returns to step 2 and waits for the next connection.

If step 4 is likely to take a long or indefinite amount of time, traditional Unix servers such as wu-ftpd create a new process to handle each connection so that multiple clients can be serviced at the same time. Java programs should spawn a thread to interact with the client so that the server can be ready to process the next connection sooner. A thread places a far smaller load on the server than a complete child process. In fact, the overhead of forking too many processes is why the typical Unix FTP server can't handle more than roughly 400 connections without slowing to a crawl. On the other hand, if the protocol is simple and quick and allows the server to close the connection when it's through, then it will be more efficient for the server to process the client request immediately without spawning a thread.

Although threads are lighter-weight than processes on most systems (Linux is the notable exception), too many threads can still be a performance problem. For instance, on most VMs each thread requires about a megabyte of RAM above and beyond what the rest of the program needs. Thus, on a typical modern server with about a gigabyte of RAM, anything close to or beyond a thousand threads is likely to slow down dramatically and eventually crash as the CPU violently and frequently swaps data into and out of RAM. Spawning too many threads is one of the few ways you can reliably crash any Java virtual machine.

Java 1.4 introduces a `ServerSocketChannel` class that provides non-blocking, multiplexed I/O based on channels rather than streams. With channels, a single thread can process multiple connections, thereby requiring many fewer threads and placing a much smaller load on the VM. This can be highly advantageous for high volume servers on some operating systems. I'll discuss these kinds of servers in Chapter 12. For simple, low-volume servers or any servers that need to run with Java 1.3 or earlier, the techniques discussed in this chapter should be used.

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

The operating system stores incoming connection requests addressed to a particular port in a first-in, first-out queue. The default length of the queue is normally 50, although it can vary from operating system to operating system. Some operating systems (not Solaris) have a maximum queue length, typically five. On these systems, the queue length will be the largest possible value less than or equal to 50. After the queue fills to capacity with unprocessed connections, the host refuses additional connections on that port until slots in the queue open up. Many (though not all) clients will try to make a connection multiple times if their initial attempt is refused. The operating system manages incoming connections and the queue; your program does not need to worry about it. Several `ServerSocket` constructors allow you to change the length of the queue if its default length isn't large enough; however, you won't be able to increase the queue beyond the maximum size that the operating system supports.

10.1.1. The Constructors

There are four public `ServerSocket` constructors:

```
public ServerSocket(int port) throws BindException, IOException
public ServerSocket(int port, int queueLength)
 throws BindException, IOException
public ServerSocket(int port, int queueLength, InetAddress bindAddress)
 throws IOException
public ServerSocket( ) throws IOException  // Java 1.4
```

These constructors let you specify the port, the length of the queue used to hold incoming connection requests, and the local network interface to bind to. They pretty much all do the same thing, though some use default values for the queue length and the address to bind to. Let's explore these in order.

10.1.1.1. public ServerSocket(int port) throws BindException, IOException

This constructor creates a server socket on the port specified by the argument. If you pass 0 for the port number, the system selects an available port for you. A port chosen for you by the system is sometimes called an *anonymous port* since you don't know its number. For servers, anonymous ports aren't very useful because clients need to know in advance which port to connect to; however, there are a few situations (which we will discuss later) in which an anonymous port might be useful.

For example, to create a server socket that would be used by an HTTP server on port 80, you would write:

```
try {
  ServerSocket httpd = new ServerSocket(80);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

The constructor throws an `IOException` (specifically, a `BindException`) if the socket cannot be created and bound to the requested port. An `IOException` when creating a `ServerSocket` almost always means one of two things. Either another server socket, possibly from a completely different program, is already using the requested port, or you're trying to connect to a port from 1 to 1,023 on Unix (including Linux and Mac OS X) without root (superuser) privileges.

You can use this constructor to write a variation on the `PortScanner` programs of the previous chapter. Example 10-1 checks for ports on the local machine by attempting to create `ServerSocket` objects on them and seeing on which ports that fails. If you're using Unix and are not running as root, this program works only for ports 1,024 and above.

Example 10-1. Look for local ports

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

public class LocalPortScanner {

  public static void main(String[] args) {

    for (int port = 1; port <= 65535; port++) {

      try {
        // the next line will fail and drop into the catch block if
        // there is already a server running on the port
        ServerSocket server = new ServerSocket(port);
      }
      catch (IOException ex) {
        System.out.println("There is a server on port " + port + ".");
      } // end catch

    } // end for

  }

}
```

Here's the output I got when running `LocalPortScanner` on my Windows NT 4.0 workstation:

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

```
D:\JAVA\JNP2\examples\11>java LocalPortScanner
There is a server on port 135.
There is a server on port 1025.
There is a server on port 1026.
There is a server on port 1027.
There is a server on port 1028.
```

10.1.1.2. public ServerSocket(int port, int queueLength) throws IOException, BindException

This constructor opens a server socket on the specified port with a queue length of your choosing. If the machine has multiple network interfaces or IP addresses, then it listens on this port on all those interfaces and IP addresses. The `queueLength` argument sets the length of the queue for incoming connection requests—that is, how many incoming connections can be stored at one time before the host starts refusing connections. Some operating systems have a maximum queue length, typically five. If you try to expand the queue past that maximum number, the maximum queue length is used instead. If you pass 0 for the port number, the system selects an available port.

For example, to create a server socket on port 5,776 that would hold up to 100 incoming connection requests in the queue, you would write:

```
try {
  ServerSocket httpd = new ServerSocket(5776, 100);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

The constructor throws an `IOException` (specifically, a `BindException`) if the socket cannot be created and bound to the requested port. However, no exception is thrown if the queue length is larger than the host OS supports. Instead, the queue length is simply set to the maximum size allowed.

10.1.1.3. public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws BindException, IOException

This constructor binds a server socket to the specified port with the specified queue length. It differs from the other two constructors in binding only to the specified local IP address. This constructor is useful for servers that run on systems with several IP addresses because it allows you to choose the address to which you'll listen. That is, the server socket only listens for incoming connections on the specified address; it won't listen for connections that come in through the host's other addresses. The previous two constructors bind to all local IP addresses by default.

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

For example, *login.ibiblio.org* is a particular Linux box in North Carolina. It's connected to the Internet with the IP address 152.2.210.122. The same box has a second Ethernet card with the local IP address 192.168.210.122 that is not visible from the public Internet, only from the local network. If for some reason I wanted to run a server on this host that only responded to local connections from within the same network, I could create a server socket that listens on port 5,776 of 192.168.210.122 but not on port 5,776 of 152.2.210.122, like so:

```
try {
  ServerSocket httpd = new ServerSocket(5776, 10,
   InetAddress.getByName("192.168.210.122"));
}
catch (IOException ex) {
  System.err.println(ex);
}
```

The constructor throws an `IOException` (again, really a `BindException`) if the socket cannot be created and bound to the requested port or network interface.

10.1.1.4. public ServerSocket( ) throws IOException // Java 1.4

The public no-args constructor is new in Java 1.4. It creates a `ServerSocket` object but does not actually bind it to a port so it cannot initially accept any connections. It can be bound later using the `bind()` methods also introduced in Java 1.4:

```
public void bind(SocketAddress endpoint) throws IOException // Java 1.4
public void bind(SocketAddress endpoint, int queueLength) // Java 1.4
  throws IOException
```

The primary use for this feature is to allow programs to set server socket options before binding to a port. Some options are fixed after the server socket has been bound. The general pattern looks like this:

```
ServerSocket ss = new ServerSocket( );
// set socket options...
SocketAddress  http = new InetSocketAddress(80);
ss.bind(http);
```

You can also past null for the `SocketAddress` to select an arbitrary port. This is like passing 0 for the port number in the other constructors.

10.1.2. Accepting and Closing Connections

A `ServerSocket` customarily operates in a loop that repeatedly accepts connections. Each pass through the loop invokes the `accept( )` method. This returns a `Socket` object representing the connection between the remote client and the local server. Interaction with the client takes place through this `Socket` object. When the transaction is finished, the server should invoke the `Socket` object's `close()` method. If the client closes the connection while the server is still operating, the input and/or output streams that connect the server to the client throw an `InterruptedIOException` on the next read or write. In either case, the server should then get ready to process the next incoming connection. However, when the server needs to shut down and not process any further incoming connections, you should invoke the `ServerSocket` object's `close( )` method.

10.1.2.1. public Socket accept( ) throws IOException

When server setup is done and you're ready to accept a connection, call the `ServerSocket`'s `accept()` method. This method "blocks"; that is, it stops the flow of execution and waits until a client connects. When a client does connect, the `accept( )` method returns a `Socket` object. You use the streams returned by this `Socket`'s `getInputStream( )` and `getOutputStream( )` methods to communicate with the client. For example:

```
ServerSocket server = new ServerSocket(5776);
while (true) {
  Socket connection = server.accept( );
  OutputStreamWriter out
   = new OutputStreamWriter(connection.getOutputStream( ));
  out.write("You've connected to this server. Bye-bye now.\r\n");
  connection.close( );
}
```

If you don't want the program to halt while it waits for a connection, put the call to `accept ( )` in a separate thread.

If you're using Java 1.4 or later, you have the option to use channels and non-blocking I/O instead of threads. In some (not all) virtual machines, this is much faster than using streams and threads. These techniques will be discussed in Chapter 12.

When exception handling is added, the code becomes somewhat more convoluted. It's important to distinguish between exceptions that should probably shut down the server and log an error message, and exceptions that should just close that active connection. Exceptions thrown by `accept( )` or the input and output streams generally should not shut down the server. Most other exceptions probably should. To do this, you'll need to nest your `try` blocks.

Finally, most servers will want to make sure that all sockets they accept are closed when they're finished. Even if the protocol specifies that clients are responsible for closing connections, clients do not always strictly adhere to the protocol. The call to `close( )` also has to be wrapped in a `try` block that catches an `IOException`. However, if you do catch an `IOException` when closing the socket, ignore it. It just means that the client closed the socket before the server could. Here's a slightly more realistic example:

```
try {
  ServerSocket server = new ServerSocket(5776);
  while (true) {
    Socket connection = server.accept( );
    try {
      Writer out
       = new OutputStreamWriter(connection.getOutputStream( ));
      out.write("You've connected to this server. Bye-bye now.\r\n");
      out.flush( );
      connection.close( );
    }
    catch (IOException ex) {
      // This tends to be a transitory error for this one connection;
      // e.g. the client broke the connection early. Consequently,
      // you don't want to break the loop or print an error message.
      // However, you might choose to log this exception in an error log.
    }
    finally {
      // Guarantee that sockets are closed when complete.
      try {
        if (connection != null) connection.close( );
      }
      catch (IOException ex) {}
    }
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Example 10-2 implements a simple daytime server, as per RFC 867. Since this server just sends a single line of text in response to each connection, it processes each connection immediately. More complex servers should spawn a thread to handle each request. In this case, the overhead of spawning a thread would be greater than the time needed to process the request.

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

If you run this program on Unix (including Linux and Mac OS X), you need to run it as root in order to connect to port 13. If you don't want to or can't run it as root, change the port number to something above 1024—say, 1313.

Example 10-2. A daytime server

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

public class DaytimeServer {

  public final static int DEFAULT_PORT = 13;

  public static void main(String[] args) {

   int port = DEFAULT_PORT;
   if (args.length > 0) {
     try {
        port = Integer.parseInt(args[0]);
        if (port < 0 || port >= 65536) {
          System.out.println("Port must between 0 and 65535");
          return;
        }
     }
     catch (NumberFormatException ex) {
       // use default port
     }

   }

   try {

     ServerSocket server = new ServerSocket(port);

     Socket connection = null;
     while (true) {

       try {
         connection = server.accept( );
         Writer out = new OutputStreamWriter(connection.getOutputStream( ));
         Date now = new Date( );
         out.write(now.toString( ) +"\r\n");
         out.flush( );
         connection.close( );
       }
       catch (IOException ex) {}
       finally {
```


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

```
            try {
              if (connection != null) connection.close( );
            }
            catch (IOException ex) {}
          }

        }  // end while

      }  // end try
      catch (IOException ex) {
        System.err.println(ex);
      } // end catch

    } // end main

} // end DaytimeServer
```

Example 10-2 is straightforward. The first three lines import the usual packages, `java.io` and `java.net`, as well as `java.util.Date`, which provides the time as read by the server's internal clock. There is a single `public final static int` field (i.e., a constant) in the class `DEFAULT_PORT`, which is set to the well-known port for a daytime server (port 13). The class has a single method, `main( )`, which does all the work. If the port is specified on the command line, then it's read from `args[0]`. Otherwise, the default port is used.

The outer `try` block traps any `IOException`s that may arise while the `ServerSocket` object `server` is constructed on the daytime port or when it accepts connections. The inner `try` block watches for exceptions thrown while the connections are accepted and processed. The `accept( )` method is called within an infinite loop to watch for new connections; like many servers, this program never terminates but continues listening until an exception is thrown or you stop it manually.

The command for stopping a program manually depends on your system; under Unix, NT, and many other systems, CTRL-C will do the job. If you are running the server in the background on a Unix system, stop it by finding the server's process ID and killing it with the `kill` command (**`kill`** *`pid`*).

When a client connects, `accept( )` returns a `Socket`, which is stored in the local variable `connection`, and the program continues. It calls `getOutputStream( )` to get the output stream associated with that `Socket` and then chains that output stream to a new `OutputStreamWriter`, `out`. A new `Date` object provides the current time. The content is sent to the client by writing its string representation on `out` with `write()`.

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

Finally, after the data is sent or an exception has been thrown, the `finally` block closes the `connection`. Always close a socket when you're finished with it. In the previous chapter, I said that a client shouldn't rely on the other side of a connection to close the socket: that goes triple for servers. Clients time out or crash; users cancel transactions; networks go down in high-traffic periods. For any of these or a dozen more reasons, you cannot rely on clients to close sockets, even when the protocol requires them to, which this one doesn't.

Sending binary, nontext data is not significantly harder. Example 10-3 demonstrates with a time server that follows the time protocol outlined in RFC 868. When a client connects, the server sends a 4-byte, big-endian, unsigned integer specifying the number of seconds that have passed since 12:00 A.M., January 1, 1900 GMT (the epoch). Once again, the current time is found by creating a new `Date` object. However, since the `Date` class counts milliseconds since 12:00 A.M., January 1, 1970 GMT rather than seconds since 12:00 A.M., January 1, 1900 GMT, some conversion is necessary.

Example 10-3. A time server

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

public class TimeServer {

  public final static int DEFAULT_PORT = 37;

  public static void main(String[] args) {

   int port = DEFAULT_PORT;
   if (args.length > 0) {
     try {
        port = Integer.parseInt(args[0]);
        if (port < 0 || port >= 65536) {
          System.out.println("Port must between 0 and 65535");
          return;
        }
     }
     catch (NumberFormatException ex) {}
   }

   // The time protocol sets the epoch at 1900,
   // the Date class at 1970. This number
   // converts between them.

   long differenceBetweenEpochs = 2208988800L;

   try {
     ServerSocket server = new ServerSocket(port);
       while (true) {
```


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

```
            Socket connection = null;
            try {
              connection = server.accept( );
              OutputStream out = connection.getOutputStream( );
              Date now = new Date( );
              long msSince1970 = now.getTime( );
              long secondsSince1970 = msSince1970/1000;
              long secondsSince1900 = secondsSince1970
               + differenceBetweenEpochs;
              byte[] time = new byte[4];
              time[0]
               = (byte) ((secondsSince1900 & 0x00000000FF000000L) >> 24);
              time[1]
               = (byte) ((secondsSince1900 & 0x0000000000FF0000L) >> 16);
              time[2]
               = (byte) ((secondsSince1900 & 0x000000000000FF00L) >> 8);
              time[3] = (byte) (secondsSince1900 & 0x00000000000000FFL);
              out.write(time);
              out.flush( );
            } // end try
            catch (IOException ex) {
            } // end catch
            finally {
              if (connection != null) connection.close( );
            }
          }  // end while
        }  // end try
        catch (IOException ex) {
          System.err.println(ex);
        } // end catch

      } // end main

    } // end TimeServer
```

As with the `TimeClient` of the previous chapter, most of the effort here goes into working with a data format (32-bit unsigned integers) that Java doesn't natively support.

10.1.2.2. public void close( ) throws IOException

If you're finished with a server socket, you should close it, especially if the program is going to continue to run for some time. This frees up the port for other programs that may wish to use it. Closing a `ServerSocket` should not be confused with closing a `Socket`. Closing a `ServerSocket` frees a port on the local host, allowing another server to bind to the port; it also breaks all currently open sockets that the `ServerSocket` has accepted.

Server sockets are closed automatically when a program dies, so it's not absolutely necessary to close them in programs that terminate shortly after the `ServerSocket` is no longer needed. Nonetheless, it doesn't hurt. For example, the main loop of the

`LocalPortScanner` program might be better written like this so that it doesn't temporarily occupy most of the ports on the system:

```
for (int port = 1; port <= 65535; port++) {

  try {
    // the next line will fail and drop into the catch block if
    // there is already a server running on the port
    ServerSocket server = new ServerSocket(port);
    server.close( );
  }
  catch (IOException ex) {
    System.out.println("There is a server on port " + port + ".");
  }

} // end for
```

After the server socket has been closed, it cannot be reconnected, even to the same port.

Java 1.4 adds an `isClosed( )` method that returns true if the `ServerSocket` has been closed, false if it hasn't:

```
public boolean isClosed( ) // Java 1.4
```

`ServerSocket` objects that were created with the no-args `ServerSocket( )` constructor and not yet bound to a port are not considered to be closed. Invoking `isClosed( )` on these objects returns false. Java 1.4 also adds an `isBound( )` method that tells you whether the `ServerSocket` has been bound to a port:

```
public boolean isBound( ) // Java 1.4
```

As with the `isBound( )` method of the `Socket` class discussed in the last chapter, the name is a little misleading. `isBound( )` returns true if the `ServerSocket` has ever been bound to a port, even if it's currently closed. If you need to test whether a `ServerSocket` is open, you must check both that `isBound( )` returns true and that `isClosed( )` returns false. For example:

```
public static boolean isOpen(ServerSocket ss) {
  return ss.isBound( ) && ! ss.isClosed( );
}
```

10.1.3. The get Methods

The `ServerSocket` class provides two getter methods that tell you the local address and port occupied by the server socket. These are useful if you've opened a server socket on an anonymous port and/or an unspecified network interface. This would be the case, for one example, in the data connection of an FTP session.

10.1.3.1. public InetAddress getInetAddress( )

This method returns the address being used by the server (the local host). If the local host has a single IP address (as most do), this is the address returned by `InetAddress.getLocalHost( )`. If the local host has more than one IP address, the specific address returned is one of the host's IP addresses. You can't predict which address you will get. For example:

```
ServerSocket httpd = new ServerSocket(80);
InetAddress ia = httpd.getInetAddress( );
```

If the `ServerSocket` has not yet bound to a network interface, this method returns null.

10.1.3.2. public int getLocalPort( )

The `ServerSocket` constructors allow you to listen on an unspecified port by passing 0 for the port number. This method lets you find out what port you're listening on. You might use this in a peer-to-peer multisocket program where you already have a means to inform other peers of your location. Or a server might spawn several smaller servers to perform particular operations. The well-known server could inform clients what ports they can find the smaller servers on. Of course, you can also use `getLocalPort()` to find a non-anonymous port, but why would you need to? Example 10-4 demonstrates.

Example 10-4. A random port

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

public class RandomPort {

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


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

```
        try {
          ServerSocket server = new ServerSocket(0);
          System.out.println("This server runs on port "
           + server.getLocalPort( ));
        }
        catch (IOException ex) {
          System.err.println(ex);
        }

      }

    }
```

Here's the output of several runs:

```
D:\JAVA\JNP3\examples\10>java RandomPort
This server runs on port 1154
D:\JAVA\JNP3\examples\10>java RandomPort
This server runs on port 1155
D:\JAVA\JNP3\examples\10>java RandomPort
This server runs on port 1156
```

At least on this VM, the ports aren't really random, but they are at least indeterminate until runtime.

If the `ServerSocket` has not yet bound to a port, then this method returns -1.

10.1.4. Socket Options

Java 1.3 only supports one socket option for server sockets, SO_TIMEOUT. Java 1.4 adds two more, SO_REUSEADDR and SO_RCVBUF.

10.1.4.1. SO_TIMEOUT

SO_TIMEOUT is the amount of time, in milliseconds, that `accept( )` waits for an incoming connection before throwing a `java.io.InterruptedIOException`. If SO_TIMEOUT is 0, `accept( )` will never time out. The default is to never time out.

Using SO_TIMEOUT is rather rare. You might need it if you were implementing a complicated and secure protocol that required multiple connections between the client and the server where responses needed to occur within a fixed amount of time. However, most servers are designed to run for indefinite periods of time and therefore just use the default timeout value,

0 (never time out). If you want to change this, the `setSoTimeout( )` method sets the SO_TIMEOUT field for this server socket object.

```
public void setSoTimeout(int timeout) throws SocketException
public int  getSoTimeout( ) throws IOException
```

The countdown starts when `accept()` is invoked. When the timeout expires, `accept ( )` throws an `InterruptedIOException`. (In Java 1.4, it throws `SocketTimeoutException`, a subclass of `InterruptedIOException`.) You should set this option before calling `accept( )`; you cannot change the timeout value while `accept ( )` is waiting for a connection. The `timeout` argument must be greater than or equal to zero; if it isn't, the method throws an `IllegalArgumentException`. For example:

```
try {
  ServerSocket server = new ServerSocket(2048);
  server.setSoTimeout(30000); // block for no more than 30 seconds
  try {
    Socket s = server.accept( );
    // handle the connection
    // ...
  }
  catch (InterruptedIOException ex) {
    System.err.println("No connection within 30 seconds");
  }
  finally {
    server.close( );
  }
catch (IOException ex) {
  System.err.println("Unexpected IOException: " + e);
}
```

The `getSoTimeout()` method returns this server socket's current SO_TIMEOUT value. For example:

```
public void printSoTimeout(ServerSocket server) {

  int timeout = server.getSoTimeOut( );
  if (timeout > 0) {
    System.out.println(server + " will time out after "
     + timeout + "milliseconds.");
  }
  else if (timeout == 0) {
    System.out.println(server + " will never time out.");
  }
  else {
    System.out.println("Impossible condition occurred in " + server);
    System.out.println("Timeout cannot be less than zero." );
  }

}
```


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

10.1.4.2. SO_REUSEADDR // Java 1.4

The SO_REUSEADDR option for server sockets is very similar to the same option for client sockets, discussed in the last chapter. It determines whether a new socket will be allowed to bind to a previously used port while there might still be data traversing the network addressed to the old socket. As you probably expect, there are two methods to get and set this option:

```
public void setReuseAddress(boolean on) throws SocketException
public boolean getReuseAddress( ) throws SocketException
```

The default value is platform-dependent. This code fragment determines the default value by creating a new `ServerSocket` and then calling `getReuseAddress( )`:

```
ServerSocket ss = new ServerSocket(10240);
System.out.println("Reusable: " + ss.getReuseAddress( ));
```

On the Linux and Mac OS X boxes where I tested this code, server sockets were reusable.

10.1.4.3. SO_RCVBUF // Java 1.4

The SO_RCVBUF option sets the default receive buffer size for client sockets accepted by the server socket. It's read and written by these two methods:

```
public void setReceiveBufferSize(int size) throws SocketException
public int  getReceiveBufferSize( ) throws SocketException
```

Setting SO_RCVBUF on a server socket is like calling `setReceiveBufferSize( )` on each individual socket returned by `accept( )` (except that you can't change the receive buffer size after the socket has been accepted). Recall from the last chapter that this option suggests a value for the size of the individual IP packets in the stream. Faster connections will want to use larger packets, although most of the time the default value is fine.

You can set this option before or after the server socket is bound, unless you want to set a receive buffer size larger than 64K. In that case, you must set the option on an unbound `ServerSocket` before binding it. For example:

```
ServerSocket ss = new ServerSocket( );
int receiveBufferSize = ss.getReceiveBufferSize( );
if (receiveBufferSize < 131072) {
  ss.setReceiveBufferSize(131072);
}
```


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

```
ss.bind(new InetSocketAddress(8000));
//...
```

10.1.4.4. public void setPerformancePreferences(int connectionTime, int latency, int bandwidth) // Java 1.5

Java 1.5 adds a slightly different method for setting socket options—the `setPerformancePreferences()` method:

```
public void setPerformancePreferences(int connectionTime, int latency,
                                       int bandwidth)
```

This method expresses the relative preferences given to connection time, latency, and bandwidth. For instance, if `connectionTime` is 2 and `latency` is 1 and `bandwidth` is 3, then maximum bandwidth is the most important characteristic, minimum latency is the least important, and connection time is in the middle. Exactly how any given VM implements this is implementation-dependent. Indeed, it may be a no-op in some implementations. The API documentation for `ServerSocket` even suggests using non-TCP/IP sockets, although it's not at all clear what that means.

10.1.5. The Object Methods

`ServerSocket` overrides only one of the standard methods from `java.lang.Object`, `toString( )`. Thus, equality comparisons test for strict identity and server sockets are problematic in hash tables. Normally, this isn't a large problem.

10.1.5.1. public String toString( )

A `String` returned by `ServerSocket`'s `toString( )` method looks like this:

```
ServerSocket[addr=0.0.0.0,port=0,localport=5776]
```

`addr` is the address of the local network interface to which the server socket is bound. This will be 0.0.0.0 if it's bound to all interfaces, as is commonly the case. `port` is always 0. The `localport` is the local port on which the server is listening for connections. This method is sometimes useful for debugging, but not much more. Don't rely on it.

10.1.6. Implementation

The `ServerSocket` class provides two methods for changing the default implementation of server sockets. I'll describe them only briefly here, since they're primarily intended for implementers of Java virtual machines rather than application programmers.

10.1.6.1. public static void setSocketFactory(SocketImplFactory factory) throws IOException

This method sets the *system's* server `SocketImplFactory`, which is the factory used to create `ServerSocket` objects. This is not the same factory that is used to create client `Socket` objects, though the syntax is similar; you can have one factory for `Socket` objects and a different factory for `ServerSocket` objects. You can set this factory only once in a program, however. A second attempt to set the `SocketImplFactory` throws a `SocketException`.

10.1.6.2. protected final void implAccept(Socket s) throws IOException

Subclasses of `ServerSocket` use this method when they want to override `accept( )` so that it returns an instance of their own custom `Socket` subclass rather than a plain `java.net.Socket`. The overridden `accept( )` method passes its own unconnected `Socket` object to this method to actually make the connection. You pass an unconnected `Socket` object to `implAccept( )`. When `implAccept( )` returns, the `Socket` argument `s` is connected to a client. For example:

```
public Socket accept( )  throws IOException {
  Socket s = new MySocketSubclass( );
  implAccept(s);
  return s;
}
```

If the server needs to know that the `Socket` returned by `accept( )` has a more specific type than just `java.net.Socket`, it must cast the return value appropriately. For example:

```
ServerSocket server = new MyServerSocketSubclass(80);
while (true) {
    MySocketSubclass  socket = (MySocketSubclass) server.accept( );;
    // ...
  }
```

10.2. Some Useful Servers

This section shows several servers you can build with server sockets. It starts with a server you can use to test client responses and requests, much as you use Telnet to test server behavior. Then three different HTTP servers are presented, each with a different special purpose and each slightly more complex than the previous one.

10.2.1. Client Tester

In the previous chapter, you learned how to use Telnet to experiment with servers. There's no equivalent program to test clients, so let's create one. Example 10-5 is a program called `ClientTester` that runs on a port specified on the command line, shows all data sent by the client, and allows you to send a response to the client by typing it on the command line. For example, you can use this program to see the commands that Internet Explorer sends to a server.

Clients are rarely as forgiving about unexpected server responses as servers are about unexpected client responses. If at all possible, try to run the clients that connect to this program on a Unix system or some other platform that is moderately crash-proof. Don't run them on Mac OS 9 or Windows ME, which are less stable.

This program uses two threads: one to handle input from the client and the other to send output from the server. Using two threads allows the program to handle input and output simultaneously: it can send a response to the client while receiving a request—or, more to the point, it can send data to the client while waiting for the client to respond. This is convenient because different clients and servers talk in unpredictable ways. With some protocols, the server talks first; with others, the client talks first. Sometimes the server sends a one-line response; often, the response is much larger. Sometimes the client and the server talk at each other simultaneously. Other times, one side of the connection waits for the other to finish before it responds. The program must be flexible enough to handle all these cases. Example 10-5 shows the code.

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

Example 10-5. A client tester

```
import java.net.*;
import java.io.*;
import com.macfaq.io.SafeBufferedReader; // from Chapter 4

public class ClientTester {

  public static void main(String[] args) {

    int port;

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
      port = 0;
    }

    try {
      ServerSocket server = new ServerSocket(port, 1);
      System.out.println("Listening for connections on port "
       + server.getLocalPort( ));

      while (true) {
        Socket connection = server.accept( );
        try {
          System.out.println("Connection established with "
           + connection);
          Thread input = new InputThread(connection.getInputStream( ));
          input.start( );
          Thread output
           = new OutputThread(connection.getOutputStream( ));
          output.start( );
          // wait for output and input to finish
          try {
            input.join( );
            output.join( );
          }
          catch (InterruptedException ex) {
          }
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
        finally {
          try {
            if (connection != null) connection.close( );
          }
          catch (IOException ex) {}
        }
      }
    }
    catch (IOException ex) {
      e.printStackTrace( );
    }
```


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

```
    }

  }

  class InputThread extends Thread {

    InputStream in;

     public InputThread(InputStream in) {
       this.in = in;
     }

     public void run( )  {

       try {
         while (true) {
           int i = in.read( );
           if (i == -1) break;
           System.out.write(i);
         }
       }
       catch (SocketException ex) {
         // output thread closed the socket
       }
       catch (IOException ex) {
         System.err.println(ex);
       }
       try {
         in.close( );
       }
       catch (IOException ex) {
       }

    }

  }

  class OutputThread extends Thread {

    private Writer out;

    public OutputThread(OutputStream out) {
      this.out = new OutputStreamWriter(out);
    }

    public void run( ) {

      String line;
      BufferedReader in
       = new SafeBufferedReader(new InputStreamReader(System.in));
      try {
        while (true) {
          line = in.readLine( );
          if (line.equals(".")) break;
          out.write(line +"\r\n");
          out.flush( );
        }
```


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

```
        }
        catch (IOException ex) {
        }
        try {
          out.close( );
        }
        catch (IOException ex) {
        }

      }

  }
```

The client tester application is split into three classes: `ClientTester`, `InputThread`, and `OutputThread`. The `ClientTester` class reads the port from the command line, opens a `ServerSocket` on that port, and listens for incoming connections. Only one connection is allowed at a time, because this program is designed for experimentation, and a slow human being has to provide all responses. Consequently, it sets an unusually short queue length of 1. Further connections will be refused until the first one has been closed.

An infinite `while` loop waits for connections with the `accept( )` method. When a connection is detected, its `InputStream` is used to construct a new `InputThread` and its `OutputStream` is used to construct a new `OutputThread`. After starting these threads, the program waits for them to finish by calling their `join()` methods.

The `InputThread` is contained almost entirely in the `run( )` method. It has a single field, `in`, which is the `InputStream` from which data will be read. Data is read from `in` one byte at a time. Each `byte` read is written on `System.out`. The `run( )` method ends when the end of stream is encountered or an `IOException` is thrown. The most likely exception here is a `SocketException` thrown because the corresponding `OutputThread` closed the connection.

The `OutputThread` reads input from the local user sitting at the terminal and sends that data to the client. Its constructor has a single argument, an output stream for sending data to the client. `OutputThread` reads input from the user on `System.in`, which is chained to an instance of the `SafeBufferedReader` class developed in Chapter 4. The `OutputStream` that was passed to the constructor is chained to an `OutputStreamWriter` for convenience. The `run( )` method for `OutputThread` reads lines from the `SafeBufferedReader` and copies them onto the `OutputStreamWriter`, which sends them to the client. A period typed on a line by itself signals the end of user input. When this occurs, `run( )` exits the loop and `out` is closed. This has the effect of also closing the socket so that a `SocketException` is thrown in the input thread, which also exits.

For example, here's the output when Netscape Communicator 4.6 for Windows connected to this server:

```
D:\JAVA\JNP3\examples\10>java ClientTester 80
Listening for connections on port 80
Connection established with
Socket[addr=localhost/127.0.0.1,port=1033,localport=80]
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 [en] (WinNT; I)
Host: localhost
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8

<html><body><h1>Hello Client!</h1></body></html>
.
```

Even minimal exploration of clients can reveal some surprising things. For instance, I didn't know until I wrote this example that Netscape Navigator 4.6 can read *.gz* files just as easily as it can read HTML files. That might be useful for serving large text files full of redundant data.

10.2.2. HTTP Servers

HTTP is a large protocol. As you saw in Chapter 3, a full-featured HTTP server must respond to requests for files, convert URLs into filenames on the local system, respond to POST and GET requests, handle requests for files that don't exist, interpret MIME types, and much, much more. However, many HTTP servers don't need all of these features. For example, many sites simply display an "under construction" message. Clearly, Apache is overkill for a site like this. Such a site is a candidate for a custom server that does only one thing. Java's network class library makes writing simple servers like this almost trivial.

Custom servers aren't useful only for small sites. High-traffic sites like Yahoo! are also candidates for custom servers because a server that does only one thing can often be much faster than a general purpose server such as Apache or Microsoft IIS. It is easy to optimize a special purpose server for a particular task; the result is often much more efficient than a general purpose server that needs to respond to many different kinds of requests. For instance, icons and images that are used repeatedly across many pages or on high-traffic pages might be better handled by a server that read all the image files into memory on startup and then served them straight out of RAM, rather than having to read them off disk for each request. Furthermore, this server could avoid wasting time on logging if you didn't want to track the image requests separately from the requests for the pages they were included in.

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

Finally, Java isn't a bad language for full-featured web servers meant to compete with the likes of Apache or IIS. Even if you believe CPU-intensive Java programs are slower than CPU-intensive C and C++ programs (something I very much doubt is true in modern VMs), most HTTP servers are limited by bandwidth, not by CPU speed. Consequently, Java's other advantages, such as its half-compiled/half-interpreted nature, dynamic class loading, garbage collection, and memory protection really get a chance to shine. In particular, sites that make heavy use of dynamic content through servlets, PHP pages, or other mechanisms can often run much faster when reimplemented on top of a pure or mostly pure Java web server. Indeed, there are several production web servers written in Java, such as the W3C's testbed server Jigsaw (http://www.w3.org/Jigsaw/). Many other web servers written in C now include substantial Java components to support the Java Servlet API and Java Server Pages. On many sites, these are replacing the traditional CGIs, ASPs, and server-side includes, mostly because the Java equivalents are faster and less resource-intensive. I'm not going to explore these technologies here since they easily deserve a book of their own. I refer interested readers to Jason Hunter's *Java Servlet Programming* (O'Reilly). However, it is important to note that servers in general and web servers in particular are one area where Java really is competitive with C.

10.2.2.1. A single-file server

Our investigation of HTTP servers begins with a server that always sends out the same file, no matter what the request. It's called `SingleFileHTTPServer` and is shown in Example 10-6. The filename, local port, and content encoding are read from the command line. If the port is omitted, port 80 is assumed. If the encoding is omitted, ASCII is assumed.

Example 10-6. An HTTP server that chunks out the same file

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

public class SingleFileHTTPServer extends Thread {

  private byte[] content;
  private byte[] header;
  private int port = 80;

  public SingleFileHTTPServer(String data, String encoding,
   String MIMEType, int port) throws UnsupportedEncodingException {
    this(data.getBytes(encoding), encoding, MIMEType, port);
  }
```


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

```
    public SingleFileHTTPServer(byte[] data, String encoding,
     String MIMEType, int port) throws UnsupportedEncodingException {

      this.content = data;
      this.port = port;
      String header = "HTTP/1.0 200 OK\r\n"
       + "Server: OneFile 1.0\r\n"
       + "Content-length: " + this.content.length + "\r\n"
       + "Content-type: " + MIMEType + "\r\n\r\n";
      this.header = header.getBytes("ASCII");

    }

    public void run( ) {

      try {
        ServerSocket server = new ServerSocket(this.port);
        System.out.println("Accepting connections on port "
          + server.getLocalPort( ));
        System.out.println("Data to be sent:");
        System.out.write(this.content);
        while (true) {

          Socket connection = null;
          try {
            connection = server.accept( );
            OutputStream out = new BufferedOutputStream(
                                connection.getOutputStream( )
                               );
            InputStream in   = new BufferedInputStream(
                                connection.getInputStream( )
                               );
            // read the first line only; that's all we need
            StringBuffer request = new StringBuffer(80);
            while (true) {
              int c = in.read( );
              if (c == '\r' || c == '\n' || c == -1) break;
              request.append((char) c);

            }
            // If this is HTTP/1.0 or later send a MIME header
            if (request.toString( ).indexOf("HTTP/") != -1) {
              out.write(this.header);
            }
            out.write(this.content);
            out.flush( );
          }  // end try
          catch (IOException ex) {
          }
          finally {
            if (connection != null) connection.close( );
          }

        } // end while
      } // end try
      catch (IOException ex) {
        System.err.println("Could not start server. Port Occupied");
      }
```

```
    } // end run


    public static void main(String[] args) {

      try {

        String contentType = "text/plain";
        if (args[0].endsWith(".html") || args[0].endsWith(".htm")) {
          contentType = "text/html";
        }

        InputStream in = new FileInputStream(args[0]);
        ByteArrayOutputStream out = new ByteArrayOutputStream( );
        int b;
        while ((b = in.read( )) != -1) out.write(b);
        byte[] data = out.toByteArray( );

        // set the port to listen on
        int port;
        try {
          port = Integer.parseInt(args[1]);
          if (port < 1 || port > 65535) port = 80;
        }
        catch (Exception ex) {
          port = 80;
        }

        String encoding = "ASCII";
        if (args.length >= 2) encoding = args[2];

        Thread t = new SingleFileHTTPServer(data, encoding,
         contentType, port);
        t.start( );

      }
      catch (ArrayIndexOutOfBoundsException ex) {
        System.out.println(
         "Usage: java SingleFileHTTPServer filename port encoding");
      }
      catch (Exception ex) {
        System.err.println(ex);
      }

    }

  }
```

The constructors set up the data to be sent along with an HTTP header that includes information about content length and content encoding. The header and the body of the response are stored in byte arrays in the desired encoding so that they can be blasted to clients very quickly.

The `SingleFileHTTPServer` class itself is a subclass of `Thread`. Its `run( )` method processes incoming connections. Chances are this server will serve only small files and will

support only low-volume web sites. Since all the server needs to do for each connection is check whether the client supports HTTP/1.0 and spew one or two relatively small byte arrays over the connection, chances are this will be sufficient. On the other hand, if you find clients are getting refused, you could use multiple threads instead. A lot depends on the size of the file served, the peak number of connections expected per minute, and the thread model of Java on the host machine. Using multiple threads would be a clear win for a server that was even slightly more sophisticated than this one.

The `run( )` method creates a `ServerSocket` on the specified port. Then it enters an infinite loop that continually accepts connections and processes them. When a socket is accepted, an `InputStream` reads the request from the client. It looks at the first line to see whether it contains the string `HTTP`. If it sees this string, the server assumes that the client understands HTTP/1.0 or later and therefore sends a MIME header for the file; then it sends the data. If the client request doesn't contain the string `HTTP`, the server omits the header, sending the data by itself. Finally, the server closes the connection and tries to accept the next connection.

The `main( )` method just reads parameters from the command line. The name of the file to be served is read from the first command-line argument. If no file is specified or the file cannot be opened, an error message is printed and the program exits. Assuming the file *can* be read, its contents are read into the `byte` array `data`. A reasonable guess is made about the content type of the file, and that guess is stored in the `contentType` variable. Next, the port number is read from the second command-line argument. If no port is specified or if the second argument is not an integer from 0 to 65,535, port 80 is used. The encoding is read from the third command-line argument, if present. Otherwise, ASCII is assumed. (Surprisingly, some VMs don't support ASCII, so you might want to pick 8859-1 instead.) Then these values are used to construct a `SingleFileHTTPServer` object and start it running. This is only one possible interface. You could easily use this class as part of some other program. If you added a setter method to change the content, you could easily use it to provide simple status information about a running server or system. However, that would raise some additional issues of thread safety that Example 10-6 doesn't have to address because it's immutable.

Here's what you see when you connect to this server via Telnet; the specifics depend on the exact server and file:

```
% telnet macfaq.dialup.cloud9.net 80
Trying 168.100.203.234...
Connected to macfaq.dialup.cloud9.net.
Escape character is '^]'.
GET / HTTP/1.0
HTTP/1.0 200 OK
Server: OneFile 1.0
```

```
Content-length: 959
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
<TITLE>Under Construction</TITLE>
</HEAD>

<BODY>
...
```

10.2.2.2. A redirector

Redirection is another simple but useful application for a special-purpose HTTP server. In this section, we develop a server that redirects users from one web site to another—for example, from *cnet.com* to *www.cnet.com*. Example 10-7 reads a URL and a port number from the command line, opens a server socket on the port, and redirects all requests that it receives to the site indicated by the new URL using a 302 FOUND code. Chances are this server is fast enough not to require multiple threads. Nonetheless, threads might be mildly advantageous, especially for a high volume site on a slow network connection. But really for purposes of example more than anything, I've made the server multithreaded. In this example, I chose to use a new thread rather than a thread pool for each connection. This is perhaps a little simpler to code and understand but somewhat less efficient. In Example 10-8, we'll look at an HTTP server that uses a thread pool.

Example 10-7. An HTTP redirector

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

public class Redirector implements Runnable {

  private int port;
  private String newSite;

  public Redirector(String site, int port) {
    this.port = port;
    this.newSite = site;
  }

  public void run( ) {

    try {

      ServerSocket server = new ServerSocket(this.port);
```

```
          System.out.println("Redirecting connections on port "
            + server.getLocalPort( ) + " to " + newSite);

          while (true) {

            try {
              Socket s = server.accept( );
              Thread t = new RedirectThread(s);
              t.start( );
            }  // end try
            catch (IOException ex) {
            }

          } // end while

        } // end try
        catch (BindException ex) {
          System.err.println("Could not start server. Port Occupied");
        }
        catch (IOException ex) {
          System.err.println(ex);
        }

      }  // end run

      class RedirectThread extends Thread {

        private Socket connection;

        RedirectThread(Socket s) {
          this.connection = s;
        }

        public void run( ) {

          try {

            Writer out = new BufferedWriter(
                          new OutputStreamWriter(
                           connection.getOutputStream( ), "ASCII"
                          )
                         );
            Reader in = new InputStreamReader(
                         new BufferedInputStream(
                          connection.getInputStream( )
                         )
                        );

            // read the first line only; that's all we need
            StringBuffer request = new StringBuffer(80);
            while (true) {
              int c = in.read( );
              if (c == '\r' || c == '\n' || c == -1) break;
              request.append((char) c);
            }
            // If this is HTTP/1.0 or later send a MIME header
            String get = request.toString( );
            int firstSpace = get.indexOf(' ');
```

```
            int secondSpace = get.indexOf(' ', firstSpace+1);
            String theFile = get.substring(firstSpace+1, secondSpace);
            if (get.indexOf("HTTP") != -1) {
              out.write("HTTP/1.0 302 FOUND\r\n");
              Date now = new Date( );
              out.write("Date: " + now + "\r\n");
              out.write("Server: Redirector 1.0\r\n");
              out.write("Location: " + newSite + theFile + "\r\n");
              out.write("Content-type: text/html\r\n\r\n");
              out.flush( );
            }
            // Not all browsers support redirection so we need to
            // produce HTML that says where the document has moved to.
            out.write("<HTML><HEAD><TITLE>Document moved</TITLE></HEAD>\r\n");
            out.write("<BODY><H1>Document moved</H1>\r\n");
            out.write("The document " + theFile
             + " has moved to\r\n<A HREF=\"" + newSite + theFile + "\">"
             + newSite  + theFile
             + "</A>.\r\n Please update your bookmarks<P>");
            out.write("</BODY></HTML>\r\n");
            out.flush( );

          } // end try
          catch (IOException ex) {
          }
          finally {
            try {
              if (connection != null) connection.close( );
            }
            catch (IOException ex) {}
          }

        }  // end run

      }

      public static void main(String[] args) {

        int thePort;
        String theSite;

        try {
          theSite = args[0];
          // trim trailing slash
          if (theSite.endsWith("/")) {
            theSite = theSite.substring(0, theSite.length( )-1);
          }
        }
        catch (Exception ex) {
          System.out.println(
           "Usage: java Redirector http://www.newsite.com/ port");
          return;
        }

        try {
          thePort = Integer.parseInt(args[1]);
        }
        catch (Exception ex) {
```

```
          thePort = 80;
        }

        Thread t = new Thread(new Redirector(theSite, thePort));
        t.start( );

      }  // end main

    }
```

In order to start the redirector on port 80 and redirect incoming requests to http://www.ibiblio.org/xml/, type:

```
D:\JAVA\JNP3\examples\10>java Redirector http://www.ibiblio.org/xml/
Redirecting connections on port 80 to http://www.ibiblio.org/xml/
```

If you connect to this server via Telnet, this is what you'll see:

```
% telnet macfaq.dialup.cloud9.net 80
Trying 168.100.203.234...
Connected to macfaq.dialup.cloud9.net.
Escape character is '^]'.
GET / HTTP/1.0
HTTP/1.0 302 FOUND
Date: Wed Sep 08 11:59:42 PDT 1999
Server: Redirector 1.0
Location: http://www.ibiblio.org/xml/
Content-type: text/html

<HTML><HEAD><TITLE>Document moved</TITLE></HEAD>
<BODY><H1>Document moved</H1>
The document / has moved to
<A HREF="http://www.ibiblio.org/xml/">http://www.ibiblio.org/xml/</A>.
 Please update your bookmarks<P></BODY></HTML>
Connection closed by foreign host.
```

If, however, you connect with a reasonably modern web browser, you should be sent to http://www.ibiblio.org/xml/ with only a slight delay. You should never see the HTML added after the response code; this is only provided to support older browsers that don't do redirection automatically.

The `main( )` method provides a very simple interface that reads the URL of the new site to redirect connections to and the local port to listen on. It uses this information to construct a `Redirector` object. Then it uses the resulting `Runnable` object (`Redirector` implements `Runnable`) to spawn a new thread and start it. If the port is not specified, `Redirector` listens on port 80. If the site is omitted, `Redirector` prints an error message and exits.

The `run()` method of `Redirector` binds the server socket to the port, prints a brief status message, and then enters an infinite loop in which it listens for connections. Every time a connection is accepted, the resulting `Socket` object is used to construct a `RedirectThread`. This `RedirectThread` is then started. All further interaction with the client takes place in this new thread. The `run( )` method of `Redirector` then simply waits for the next incoming connection.

The `run( )` method of `RedirectThread` does most of the work. It begins by chaining a `Writer` to the `Socket`'s output stream and a `Reader` to the `Socket`'s input stream. Both input and output are buffered. Then the `run( )` method reads the first line the client sends. Although the client will probably send a whole MIME header, we can ignore that. The first line contains all the information we need. The line looks something like this:

```
GET /directory/filename.html HTTP/1.0
```

It is possible that the first word will be POST or PUT instead or that there will be no HTTP version. The second "word" is the file the client wants to retrieve. This *must* begin with a slash (/). Browsers are responsible for converting relative URLs to absolute URLs that begin with a slash; the server does not do this. The third word is the version of the HTTP protocol the browser understands. Possible values are nothing at all (pre-HTTP/1.0 browsers), HTTP/1.0, or HTTP/1.1.

To handle a request like this, `Redirector` ignores the first word. The second word is attached to the URL of the target server (stored in the field `newSite`) to give a full redirected URL. The third word is used to determine whether to send a MIME header; MIME headers are not used for old browsers that do not understand HTTP/1.0. If there is a version, a MIME header is sent; otherwise, it is omitted.

Sending the data is almost trivial. The `Writer out` is used. Since all the data we send is pure ASCII, the exact encoding isn't too important. The only trick here is that the end-of-line character for HTTP requests is `\r\n`--a carriage return followed by a linefeed.

The next lines each send one line of text to the client. The first line printed is:

```
HTTP/1.0 302 FOUND
```

This is an HTTP/1.0 response code that tells the client to expect to be redirected. The second line is a `Date:` header that gives the current time at the server. This line is optional. The third line is the name and version of the server; this line is also optional but is used by spiders that try to keep statistics about the most popular web servers. (It would be very surprising to ever see `Redirector` break into single digits in lists of the most popular servers.) The next line is the `Location:` header, which is required for this server. It tells the client where it is being

redirected to. Last is the standard `Content-type:` header. We send the content type `text/html` to indicate that the client should expect to see HTML. Finally, a blank line is sent to signify the end of the header data.

Everything after this will be HTML, which is processed by the browser and displayed to the user. The next several lines print a message for browsers that do not support redirection, so those users can manually jump to the new site. That message looks like:

```
<HTML><HEAD><TITLE>Document moved</TITLE></HEAD>
<BODY><H1>Document moved</H1>
The document / has moved to
<A HREF="http://www.ibiblio.org/xml/">http://www.ibiblio.org/xml/</A>.
 Please update your bookmarks<P></BODY></HTML>
```

Finally, the connection is closed and the thread dies.

10.2.2.3. A full-fledged HTTP server

Enough special-purpose HTTP servers. This next section develops a full-blown HTTP server, called `JHTTP`, that can serve an entire document tree, including images, applets, HTML files, text files, and more. It will be very similar to the `SingleFileHTTPServer`, except that it pays attention to the GET requests. This server is still fairly lightweight; after looking at the code, we'll discuss other features we might want to add.

Since this server may have to read and serve large files from the filesystem over potentially slow network connections, we'll change its approach. Rather than processing each request as it arrives in the main thread of execution, we'll place incoming connections in a pool. Separate instances of a `RequestProcessor` class will remove the connections from the pool and process them. Example 10-8 shows the main `JHTTP` class. As in the previous two examples, the `main( )` method of `JHTTP` handles initialization, but other programs can use this class to run basic web servers.

Example 10-8. The JHTTP web server

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

public class JHTTP extends Thread {

  private File documentRootDirectory;
```


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

```
    private String indexFileName = "index.html";
    private ServerSocket server;
    private int numThreads = 50;

    public JHTTP(File documentRootDirectory, int port,
     String indexFileName) throws IOException {

      if (!documentRootDirectory.isDirectory( )) {
        throw new IOException(documentRootDirectory
         + " does not exist as a directory");
      }
      this.documentRootDirectory = documentRootDirectory;
      this.indexFileName = indexFileName;
      this.server = new ServerSocket(port);
    }

    public JHTTP(File documentRootDirectory, int port)
     throws IOException {
      this(documentRootDirectory, port, "index.html");
    }

    public JHTTP(File documentRootDirectory) throws IOException {
      this(documentRootDirectory, 80, "index.html");
    }

    public void run( ) {

      for (int i = 0; i < numThreads; i++) {
        Thread t = new Thread(
         new RequestProcessor(documentRootDirectory, indexFileName));
        t.start( );
      }
      System.out.println("Accepting connections on port "
       + server.getLocalPort( ));
      System.out.println("Document Root: " + documentRootDirectory);
      while (true) {
        try {
          Socket request = server.accept( );
          RequestProcessor.processRequest(request);
        }
        catch (IOException ex) {
        }
      }

    }

    public static void main(String[] args) {

      // get the Document root
      File docroot;
      try {
        docroot = new File(args[0]);
      }
      catch (ArrayIndexOutOfBoundsException ex) {
        System.out.println("Usage: java JHTTP docroot port indexfile");
        return;
      }
```


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

```
        // set the port to listen on
        int port;
        try {
          port = Integer.parseInt(args[1]);
          if (port < 0 || port > 65535) port = 80;
        }
        catch (Exception ex) {
          port = 80;
        }

        try {
          JHTTP webserver = new JHTTP(docroot, port);
          webserver.start( );
        }
        catch (IOException ex) {
          System.out.println("Server could not start because of an "
           + e.getClass( ));
          System.out.println(e);
        }

      }

    }
```

The `main( )` method of the `JHTTP` class sets the document root directory from `args[0]`. The port is read from `args[1]` or 80 is used for a default. Then a new `JHTTP` thread is constructed and started. The `JHTTP` thread spawns 50 `RequestProcessor` threads to handle requests, each of which retrieves incoming connection requests from the `RequestProcessor` pool as they become available. The JHTTP thread repeatedly accepts incoming connections and puts them in the `RequestProcessor` pool.

Each connection is handled by the `run( )` method of the `RequestProcessor` class shown in Example 10-9. This method waits until it can get a `Socket` out of the pool. Once it does that, it gets input and output streams from the socket and chains them to a reader and a writer. The reader reads the first line of the client request to determine the version of HTTP that the client supports—we want to send a MIME header only if this is HTTP/1.0 or later—and the requested file. Assuming the method is `GET`, the file that is requested is converted to a filename on the local filesystem. If the file requested is a directory (i.e., its name ends with a slash), we add the name of an index file. We use the canonical path to make sure that the requested file doesn't come from outside the document root directory. Otherwise, a sneaky client could walk all over the local filesystem by including `..` in URLs to walk up the directory hierarchy. This is all we'll need from the client, although a more advanced web server, especially one that logged hits, would read the rest of the MIME header the client sends.

Next, the requested file is opened and its contents are read into a byte array. If the HTTP version is 1.0 or later, we write the appropriate MIME headers on the output stream. To figure out the content type, we call the `guessContentTypeFromName()` method to map file

extensions such as *.html* onto MIME types such as text/html. The `byte` array containing the file's contents is written onto the output stream and the connection is closed. Exceptions may be thrown at various places if, for example, the file cannot be found or opened. If an exception occurs, we send an appropriate HTTP error message to the client instead of the file's contents.

Example 10-9. The thread pool that handles HTTP requests

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

public class RequestProcessor implements Runnable {

  private static List pool = new LinkedList( );
  private File documentRootDirectory;
  private String indexFileName = "index.html";

  public RequestProcessor(File documentRootDirectory,
   String indexFileName) {

    if (documentRootDirectory.isFile( )) {
      throw new IllegalArgumentException(
       "documentRootDirectory must be a directory, not a file");
    }
    this.documentRootDirectory = documentRootDirectory;
    try {
      this.documentRootDirectory
       = documentRootDirectory.getCanonicalFile( );
    }
    catch (IOException ex) {
    }
    if (indexFileName != null) this.indexFileName = indexFileName;
  }

  public static void processRequest(Socket request) {

    synchronized (pool) {
      pool.add(pool.size( ), request);
      pool.notifyAll( );
    }

  }

  public void run( ) {

    // for security checks
    String root = documentRootDirectory.getPath( );

    while (true) {
      Socket connection;
      synchronized (pool) {
        while (pool.isEmpty( )) {
          try {
```


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

```
                pool.wait( );
              }
              catch (InterruptedException ex) {
              }
            }
            connection = (Socket) pool.remove(0);
          }

          try {
            String filename;
            String contentType;
            OutputStream raw = new BufferedOutputStream(
                               connection.getOutputStream( )
                              );
            Writer out = new OutputStreamWriter(raw);
            Reader in = new InputStreamReader(
                        new BufferedInputStream(
                         connection.getInputStream( )
                        ),"ASCII"
                       );
            StringBuffer requestLine = new StringBuffer( );
            int c;
            while (true) {
              c = in.read( );
              if (c == '\r' || c == '\n') break;
              requestLine.append((char) c);
            }

            String get = requestLine.toString( );

            // log the request
            System.out.println(get);

            StringTokenizer st = new StringTokenizer(get);
            String method = st.nextToken( );
            String version = "";
            if (method.equals("GET")) {
              filename = st.nextToken( );
              if (filename.endsWith("/")) filename += indexFileName;
              contentType = guessContentTypeFromName(filename);
              if (st.hasMoreTokens( )) {
                version = st.nextToken( );
              }

              File theFile = new File(documentRootDirectory,
               filename.substring(1,filename.length( )));
              if (theFile.canRead( )
                  // Don't let clients outside the document root
               && theFile.getCanonicalPath( ).startsWith(root)) {
                DataInputStream fis = new DataInputStream(
                                      new BufferedInputStream(
                                       new FileInputStream(theFile)
                                      )
                                     );
                byte[] theData = new byte[(int) theFile.length( )];
                fis.readFully(theData);
                fis.close( );
                if (version.startsWith("HTTP ")) {  // send a MIME header
```


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

```
                out.write("HTTP/1.0 200 OK\r\n");
                Date now = new Date( );
                out.write("Date: " + now + "\r\n");
                out.write("Server: JHTTP/1.0\r\n");
                out.write("Content-length: " + theData.length + "\r\n");
                out.write("Content-type: " + contentType + "\r\n\r\n");
                out.flush( );
              }  // end if

              // send the file; it may be an image or other binary data
              // so use the underlying output stream
              // instead of the writer
              raw.write(theData);
              raw.flush( );
            }  // end if
            else {  // can't find the file
              if (version.startsWith("HTTP ")) {  // send a MIME header
                out.write("HTTP/1.0 404 File Not Found\r\n");
                Date now = new Date( );
                out.write("Date: " + now + "\r\n");
                out.write("Server: JHTTP/1.0\r\n");
                out.write("Content-type: text/html\r\n\r\n");
              }
              out.write("<HTML>\r\n");
              out.write("<HEAD><TITLE>File Not Found</TITLE>\r\n");
              out.write("</HEAD>\r\n");
              out.write("<BODY>");
              out.write("<H1>HTTP Error 404: File Not Found</H1>\r\n");
              out.write("</BODY></HTML>\r\n");
              out.flush( );
            }
          }
          else {  // method does not equal "GET"
            if (version.startsWith("HTTP ")) {  // send a MIME header
              out.write("HTTP/1.0 501 Not Implemented\r\n");
              Date now = new Date( );
              out.write("Date: " + now + "\r\n");
              out.write("Server: JHTTP 1.0\r\n");
              out.write("Content-type: text/html\r\n\r\n");
            }
            out.write("<HTML>\r\n");
            out.write("<HEAD><TITLE>Not Implemented</TITLE>\r\n");
            out.write("</HEAD>\r\n");
            out.write("<BODY>");
            out.write("<H1>HTTP Error 501: Not Implemented</H1>\r\n");
            out.write("</BODY></HTML>\r\n");
            out.flush( );
          }
        }
        catch (IOException ex) {
        }
        finally {
          try {
            connection.close( );
          }
          catch (IOException ex) {}
        }
```


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

```
        } // end while

      } // end run

      public static String guessContentTypeFromName(String name) {
        if (name.endsWith(".html") || name.endsWith(".htm")) {
          return "text/html";
        }
        else if (name.endsWith(".txt") || name.endsWith(".java")) {
          return "text/plain";
        }
        else if (name.endsWith(".gif")) {
          return "image/gif";
        }
        else if (name.endsWith(".class")) {
          return "application/octet-stream";
        }
        else if (name.endsWith(".jpg") || name.endsWith(".jpeg")) {
          return "image/jpeg";
        }
        else return "text/plain";
      }

    } // end RequestProcessor
```

This server is functional but still rather austere. Here are a few features that could be added:

- A server administration interface
- Support for CGI programs and/or the Java Servlet API
- Support for other request methods, such as POST, HEAD, and PUT
- A log file in the common web log file format
- Server-side includes and/or Java Server Pages
- Support for multiple document roots so individual users can have their own sites

Finally, spend a little time thinking about ways to optimize this server. If you really want to use `JHTTP` to run a high-traffic site, there are a couple of things that can speed this server up. The first and most important is to use a Just-in-Time (JIT) compiler such as HotSpot. JITs can improve program performance by an order of magnitude or more. The second thing to do is implement smart caching. Keep track of the requests you've received and store the data from the most frequently requested files in a `Hashtable` so that they're kept in memory. Use a low-priority thread to update this cache. Another option for developers using Java 1.4 or later is to use non-blocking I/O and channels instead of threads and streams. We'll explore this possibility in Chapter 12.