Table of Contents

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

Chapter 9. Sockets for Clients

Data is transmitted across the Internet in packets of finite size called *datagrams*. Each datagram contains a *header* and a *payload*. The header contains the address and port to which the packet is going, the address and port from which the packet came, and various other housekeeping information used to ensure reliable transmission. The payload contains the data itself. However, since datagrams have a finite length, it's often necessary to split the data across multiple packets and reassemble it at the destination. It's also possible that one or more packets may be lost or corrupted in transit and need to be retransmitted or that packets arrive out of order and need to be reordered. Keeping track of this—splitting the data into packets, generating headers, parsing the headers of incoming packets, keeping track of what packets have and haven't been received, and so on—is a lot of work and requires a lot of intricate code.

Fortunately, you don't have to do the work yourself. Sockets allow the programmer to treat a network connection as just another stream onto which bytes can be written and from which bytes can be read. Sockets shield the programmer from low-level details of the network, such as error detection, packet sizes, packet retransmission, network addresses, and more.

9.1. Socket Basics

A socket is a connection between two hosts. It can perform seven basic operations:

- Connect to a remote machine
- Send data
- Receive data
- Close a connection
- Bind to a port
- Listen for incoming data
- Accept connections from remote machines on the bound port

Java's `Socket` class, which is used by both clients and servers, has methods that correspond to the first four of these operations. The last three operations are needed only by servers, which wait for clients to connect to them. They are implemented by the `ServerSocket` class, which is discussed in the next chapter. Java programs normally use client sockets in the following fashion:

1. The program creates a new socket with a constructor.
2. The socket attempts to connect to the remote host.
3. Once the connection is established, the local and remote hosts get input and output streams from the socket and use those streams to send data to each other. This connection is *full-duplex*; both hosts can send and receive data simultaneously. What the data means depends on the protocol; different commands are sent to an FTP server than to an HTTP server. There will normally be some agreed-upon hand-shaking followed by the transmission of data from one to the other.
4. When the transmission of data is complete, one or both sides close the connection. Some protocols, such as HTTP 1.0, require the connection to be closed after each request is serviced. Others, such as FTP, allow multiple requests to be processed in a single connection.

9.2. Investigating Protocols with Telnet

In this chapter, you'll see clients that use sockets to communicate with a number of well-known Internet services such as HTTP, echo, and more. The sockets themselves are simple enough; however, the protocols to communicate with different servers make life complex.

To get a feel for how a protocol operates, you can use Telnet to connect to a server, type different commands to it, and watch its responses. By default, Telnet attempts to connect to port 23. To connect to servers on different ports, specify the port you want to connect to like this:

```
% telnet localhost 25
```

This example assumes that you're using a Unix system. However, Telnet clients are available for all common operating systems, and they are all pretty similar; for example, on Windows, you might have to type the hostname and the port into a dialog box rather than on the command-line, but otherwise, the clients work the same.

This requests a connection to port 25, the SMTP port, on the local machine; SMTP is the protocol used to transfer email between servers or between a mail client and a server. If you know the commands to interact with an SMTP server, you can send email without going through a mail program. This trick can be used to forge email. For example, a few years ago, the summer students at the National Solar Observatory in Sunspot, New Mexico, made it appear that the party one of the scientists was throwing after the annual volleyball match between the staff and the students was in fact a victory party for the students. (Of course, the author of this book had absolutely nothing to do with such despicable behavior. ;-)) The interaction with the SMTP server went something like this; input the user types is shown in bold (the names have been changed to protect the gullible):

```
flare% telnet localhost 25
Trying 127.0.0.1 ...
Connected to localhost.sunspot.noao.edu.
Escape character is '^]'.
220 flare.sunspot.noao.edu Sendmail 4.1/SMI-4.1 ready at
Fri, 5 Jul 93 13:13:01 MDT
HELO sunspot.noao.edu
250 flare.sunspot.noao.edu Hello localhost [127.0.0.1], pleased to meet you
MAIL FROM: bart
250 bart... Sender ok
RCPT TO: local@sunspot.noao.edu
250 local@sunspot.noao.edu... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself

In a pitiful attempt to reingratiate myself with the students
            after their inevitable defeat of the staff on the volleyball
            court at 4:00 P.M., July 24, I will be throwing a victory
            party for the students at my house that evening at 7:00.
            Everyone is invited.
            Beer and Ben-Gay will be provided so the staff may drown
            their sorrows and assuage their aching muscles after their
            public humiliation.
            Sincerely,
            Bart
            .
250 Mail accepted
QUIT
221 flare.sunspot.noao.edu delivering mail
Connection closed by foreign host.
```

Several members of the staff asked Bart why he, a staff member, was throwing a victory party for the students. The moral of this story is that you should never trust email, especially patently ridiculous email like this, without independent verification. The other moral of this story is that you can use Telnet to simulate a client, see how the client and the server interact, and thus learn what your Java program needs to do. Although this session doesn't demonstrate all the features of the SMTP protocol, it's sufficient to enable you to deduce how a simple email client talks to a server.

9.3. The Socket Class

The `java.net.Socket` class is Java's fundamental class for performing client-side TCP operations. Other client-oriented classes that make TCP network connections such as `URL`, `URLConnection`, `Applet`, and `JEditorPane` all ultimately end up invoking the methods of this class. This class itself uses native code to communicate with the local TCP stack of the host operating system. The methods of the `Socket` class set up and tear down connections and set various socket options. Because TCP sockets are more or less reliable connections, the interface that the `Socket` class provides to the programmer is *streams*. The actual reading and writing of data over the socket is accomplished via the familiar stream classes.

9.3.1. The Constructors

The nondeprecated public `Socket` constructors are simple. Each lets you specify the host and the port you want to connect to. Hosts may be specified as an `InetAddress` or a `String`. Ports are always specified as `int` values from 0 to 65,535. Two of the constructors also specify the local address and local port from which data will be sent. You might need to do this when you want to select one particular network interface from which to send data on a multihomed host.

The `Socket` class also has two protected constructors (one of which is now public in Java 1.4) that create unconnected sockets. These are useful when you want to set socket options before making the first connection.

9.3.1.1. public Socket(String host, int port) throws UnknownHostException, IOException

This constructor creates a TCP socket to the specified port on the specified host and attempts to connect to the remote host. For example:

```
try {
  Socket toOReilly = new Socket("www.oreilly.com", 80);
  // send and receive data...
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

In this constructor, the `host` argument is just a hostname expressed as a `String`. If the domain name server cannot resolve the hostname or is not functioning, the constructor throws an `UnknownHostException`. If the socket cannot be opened for some other reason, the constructor throws an `IOException`. There are many reasons a connection attempt might fail: the host you're trying to reach may not be accepting connections, a dialup Internet connection may be down, or routing problems may be preventing your packets from reaching their destination.

Since this constructor doesn't just create a `Socket` object but also tries to connect the socket to the remote host, you can use the object to determine whether connections to a particular port are allowed, as in Example 9-1.

Example 9-1. Find out which of the first 1,024 ports seem to be hosting TCP servers on a specified host

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

public class LowPortScanner {

  public static void main(String[] args) {

    String host = "localhost";

    if (args.length > 0) {
      host = args[0];
    }
    for (int i = 1; i < 1024; i++) {
      try {
        Socket s = new Socket(host, i);
        System.out.println("There is a server on port " + i + " of "
         + host);
      }
      catch (UnknownHostException ex) {
        System.err.println(ex);
        break;
      }
      catch (IOException ex) {
        // must not be a server on this port
      }
    } // end for

  }  // end main

}  // end PortScanner
```

Here's the output this program produces on my local host. Your results will vary, depending on which ports are occupied. As a rule, more ports will be occupied on a Unix workstation than on a PC or a Mac:

```
% java LowPortScanner
There is a server on port 21 of localhost
There is a server on port 22 of localhost
There is a server on port 23 of localhost
There is a server on port 25 of localhost
There is a server on port 37 of localhost
There is a server on port 111 of localhost
There is a server on port 139 of localhost
There is a server on port 210 of localhost
There is a server on port 515 of localhost
There is a server on port 873 of localhost
```

If you're curious about what servers are running on these ports, try experimenting with Telnet. On a Unix system, you may be able to find out which services reside on which ports by looking in the file */etc/services*. If `LowPortScanner` finds any ports that are running servers but are not listed in */etc/services*, then that's interesting.

Although this program looks simple, it's not without its uses. The first step to securing a system is understanding it. This program helps you understand what your system is doing so you can find (and close) possible entrance points for attackers. You may also find rogue servers: for example, `LowPortScanner` might tell you that there's a server on port 800, which, on further investigation, turns out to be an HTTP server somebody is running to serve erotic GIFs, and which is saturating your T1. However, like most security tools, this program can be misused. Don't use `LowPortScanner` to probe a machine you do not own; most system administrators would consider that a hostile act.

9.3.1.2. public Socket(InetAddress host, int port) throws IOException

Like the previous constructor, this constructor creates a TCP socket to the specified port on the specified host and tries to connect. It differs by using an `InetAddress` object (discussed in Chapter 6) to specify the host rather than a hostname. It throws an `IOException` if it can't connect, but does not throw an `UnknownHostException`; if the host is unknown, you will find out when you create the `InetAddress` object. For example:

```
try {
  InetAddress oreilly = InetAddress.getByName("www.oreilly.com");
  Socket oreillySocket = new Socket(oreilly , 80);
  // send and receive data...
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

In the rare case where you open many sockets to the same host, it is more efficient to convert the hostname to an `InetAddress` and then repeatedly use that `InetAddress` to create sockets. Example 9-2 uses this technique to improve on the efficiency of Example 9-1.

Example 9-2. Find out which of the ports at or above 1,024 seem to be hosting TCP servers

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

public class HighPortScanner {

  public static void main(String[] args) {

    String host = "localhost";

    if (args.length > 0) {
      host = args[0];
    }

    try {
      InetAddress theAddress = InetAddress.getByName(host);
      for (int i = 1024; i < 65536; i++) {
        try {
          Socket theSocket = new Socket(theAddress, i);
          System.out.println("There is a server on port "
           + i + " of " + host);
        }
        catch (IOException ex) {
          // must not be a server on this port
        }
      } // end for
    } // end try
    catch (UnknownHostException ex) {
      System.err.println(ex);
    }

  }  // end main

}  // end HighPortScanner
```

The results of this example are similar to the previous ones, except that `HighPortScanner` checks ports above 1,023.

9.3.1.3. public Socket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException

This constructor creates a socket to the specified port on the specified host and tries to connect. It connects *to* the host and port specified in the first two arguments. It connects *from* the local network interface and port specified by the last two arguments. The network interface may be either physical (e.g., a different Ethernet card) or virtual (a multihomed host).

If 0 is passed for the `localPort` argument, Java chooses a random available port between 1,024 and 65,535.

One situation where you might want to explicitly choose the local address would be on a router/firewall that uses dual Ethernet ports. Incoming connections would be accepted on one interface, processed, and forwarded to the local network from the other interface. Suppose you were writing a program to periodically dump error logs to a printer or send them over an internal mail server. You'd want to make sure you used the inward-facing network interface instead of the outward-facing network interface. For example,

```
try {
  InetAddress inward = InetAddress.getByName("router");
  Socket socket = new Socket("mail", 25, inward, 0);
  // work with the sockets...
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

By passing 0 for the local port number, I say that I don't care which port is used but I do want to use the network interface bound to the local hostname router.

This constructor can throw an `IOException` for all the usual reasons given in the previous constructors. Furthermore, an `UnknownHostException` will also be thrown if the remote host cannot be located.

Finally, an `IOException` (probably a `BindException`, although again that's just a subclass of `IOException` and not specifically declared in the `throws` clause of this method) will be thrown if the socket is unable to bind to the requested local network interface, which tends to limit the portability of applications that use this constructor. You could take deliberate advantage of this to restrict a compiled program to run on only a predetermined host. It would require customizing distributions for each computer and is certainly overkill for cheap products. Furthermore, Java programs are so easy to disassemble, decompile, and reverse engineer that this scheme is far from foolproof. Nonetheless, it might be part of a scheme to enforce a software license.

9.3.1.4. public Socket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException

This constructor is identical to the previous one except that the host to connect to is passed as an `InetAddress`, not a `String`. It creates a TCP socket to the specified port on the specified host from the specified interface and local port, and tries to connect. If it fails, it throws an `IOException`. For example:

```
try {
  InetAddress inward = InetAddress.getByName("router");
  InetAddress mail = InetAddress.getByName("mail");
  Socket socket = new Socket(mail, 25, inward, 0);
  // work with the sockets...
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

9.3.1.5. protected Socket( )

The `Socket` class also has two (three in Java 1.5) constructors that create an object without connecting the socket. You use these if you're subclassing `Socket`, perhaps to implement a special kind of socket that encrypts transactions or understands your local proxy server. Most of your implementation of a new socket class will be written in a `SocketImpl` object.

The noargs `Socket()` constructor installs the default `SocketImpl` (from either the factory or a `java.net.PlainSocketImpl`). It creates a new `Socket` without connecting it, and is usually called by subclasses of `java.net.Socket`.

In Java 1.4, this constructor has been made public, and allows you to create a socket that is not yet connected to any host. You can connect later by passing a `SocketAddress` to one of the `connect( )` methods. The most common reason to create a `Socket` object without connecting is to set socket options; many of these cannot be changed after the connection has been made. I'll discuss this soon.

9.3.1.6. protected Socket(SocketImpl impl)

This constructor installs the `SocketImpl` object `impl` when it creates the new `Socket` object. The `Socket` object is created but is not connected. This constructor is usually called by subclasses of `java.net.Socket`. You can pass `null` to this constructor if you don't need a `SocketImpl`. However, in this case, you must override all the base class methods that depend on the underlying `SocketImpl`. This might be necessary if you were using JNI to talk to something other than the default native TCP stack.

9.3.1.7. public Socket(Proxy proxy) // Java 1.5

Java 1.5 adds this constructor, which creates an unconnected socket that will use the specified proxy server. Normally, the proxy server a socket uses is controlled by the `socksProxyHost` and `socksProxyPort` system properties, and these properties apply to all sockets in the system. However a socket created by this constructor will use the specified proxy server instead. Most notably, you can pass `Proxy.NO_PROXY` for the argument to bypass all proxy servers completely and connect directly to the remote host. Of course, if a firewall prevents such connections, there's nothing Java can do about it, and the connection will fail.

If you want to use a particular proxy server, you can specify it by its address. For example, this code fragment uses the SOCKS proxy server at myproxy.example.com to connect to the host *login.ibiblio.org*:

```
SocetAddress proxyAddress = new InetSocketAddress("myproxy.example.com", 1080);
Proxy proxy = new Proxy(Proxy.Type.SOCKS,  proxyAddress)
Socket s = new Socket(proxy);
SocketAddress remote = new InetSocketAddress("login.ibiblio.org", 25);
s.connect(remote);
```

SOCKS is the only low-level proxy type Java understands. There's also a high-level `Proxy.Type.HTTP` that works in the application layer rather than the transport layer and a `Proxy.Type.DIRECT` that represents proxyless connections.

9.3.2. Getting Information About a Socket

To the programmer, `Socket` objects appear to have several private fields that are accessible through various getter methods. Actually, sockets have only one field, a `SocketImpl`; the fields that appear to belong to the `Socket` actually reflect native code in the `SocketImpl`. This way, socket implementations can be changed without disturbing the program—for example, to support firewalls and proxy servers. The actual `SocketImpl` in use is almost completely transparent to the programmer.

9.3.2.1. public InetAddress getInetAddress( )

Given a `Socket` object, the `getInetAddress( )` method tells you which remote host the `Socket` is connected to or, if the connection is now closed, which host the `Socket` was connected to when it was connected. For example:

```
try {
  Socket theSocket = new Socket("java.sun.com", 80);
  InetAddress host = theSocket.getInetAddress( );
  System.out.println("Connected to remote host " + host);
}  // end try
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

9.3.2.2. public int getPort( )

The `getPort( )` method tells you which port the `Socket` is (or was or will be) connected to on the remote host. For example:

```
try {
  Socket theSocket = new Socket("java.sun.com", 80);
  int port = theSocket.getPort( );
  System.out.println("Connected on remote port " + port);
}  // end try
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

9.3.2.3. public int getLocalPort( )

There are two ends to a connection: the remote host and the local host. To find the port number for the local end of a connection, call `getLocalPort( )`. For example:

```
try {
  Socket theSocket = new Socket("java.sun.com", 80, true);
  int localPort = theSocket.getLocalPort( );
  System.out.println("Connecting from local port " + localPort);
}  // end try
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Unlike the remote port, which (for a client socket) is usually a "well-known port" that has been preassigned by a standards committee, the local port is usually chosen by the system at runtime from the available unused ports. This way, many different clients on a system can access the same service at the same time. The local port is embedded in outbound IP packets along with the local host's IP address, so the server can send data back to the right port on the client.

9.3.2.4. public InetAddress getLocalAddress( )

The `getLocalAddress()` method tells you which network interface a socket is bound to. You normally use this on a multihomed host, or one with multiple network interfaces. For example:

```
try {
  Socket theSocket = new Socket(hostname, 80);
  InetAddress localAddress = theSocket.getLocalAddress( );
  System.out.println("Connecting from local address  " + localAddress);
}  // end try
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Example 9-3 reads a list of hostnames from the command-line, attempts to open a socket to each one, and then uses these four methods to print the remote host, the remote port, the local address, and the local port.

Example 9-3. Get a socket's information

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

public class SocketInfo {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        Socket theSocket = new Socket(args[i], 80);
        System.out.println("Connected to " + theSocket.getInetAddress( )
         + " on port "  + theSocket.getPort( ) + " from port "
         + theSocket.getLocalPort( ) + " of "
         + theSocket.getLocalAddress( ));
      }  // end try
      catch (UnknownHostException ex) {
        System.err.println("I can't find " + args[i]);
      }
      catch (SocketException ex) {
        System.err.println("Could not connect to " + args[i]);
```

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

          } // end for

        }  // end main

      }  // end SocketInfo
```

Here's the result of a sample run. I included www.oreilly.com on the command line twice in order to demonstrate that each connection was assigned a different local port, regardless of the remote host; the local port assigned to any connection is unpredictable and depends mostly on what other ports are in use. The connection to login.ibiblio.org failed because that machine does not run any servers on port 80:

```
% java SocketInfo www.oreilly.com www.oreilly.com www.macfaq.com
  login.ibiblio.org
Connected to www.oreilly.com/208.201.239.37 on port 80 from port 49156 of
/192.168.254.25
Connected to www.oreilly.com/208.201.239.37 on port 80 from port 49157 of
/192.168.254.25
Connected to www.macfaq.com/216.254.106.198 on port 80 from port 49158 of
/192.168.254.25
Could not connect to login.ibiblio.org
```

9.3.2.5. public InputStream getInputStream( ) throws IOException

The `getInputStream( )` method returns an input stream that can read data from the socket into a program. You usually chain this `InputStream` to a filter stream or reader that offers more functionality—`DataInputStream` or `InputStreamReader`, for example—before reading input. For performance reasons, it's also a very good idea to buffer the input by chaining it to a `BufferedInputStream` and/or a `BufferedReader`.

With an input stream, we can read data from a socket and start experimenting with some actual Internet protocols. One of the simplest protocols is called daytime, and is defined in RFC 867. There's almost nothing to it. The client opens a socket to port 13 on the daytime server. In response, the server sends the time in a human-readable format and closes the connection. You can test the daytime server with Telnet like this:

```
% telnet vision.poly.edu 13
Trying 128.238.42.35...
Connected to vision.poly.edu.
Escape character is '^]'.
Wed Nov 12 23:39:15 2003
Connection closed by foreign host.
```

The line "Wed Nov 12 23:39:15 2003" is sent by the daytime server. When you read the `Socket`'s `InputStream`, this is what you will get. The other lines are produced either by the Unix shell or by the Telnet program.

Example 9-4 uses the `InputStream` returned by `getInputStream()` to read the time sent by the daytime server.

Example 9-4. A daytime protocol client

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

public class DaytimeClient {

  public static void main(String[] args) {

    String hostname;

    if (args.length > 0) {
      hostname = args[0];
    }
    else {
      hostname = "time.nist.gov";
    }

    try {
      Socket theSocket = new Socket(hostname, 13);
      InputStream timeStream = theSocket.getInputStream( );
      StringBuffer time = new StringBuffer( );
      int c;
      while ((c = timeStream.read( )) != -1) time.append((char) c);
      String timeString = time.toString( ).trim( );
      System.out.println("It is " + timeString + " at " + hostname);
    }  // end try
    catch (UnknownHostException ex) {
      System.err.println(ex);
    }
    catch (IOException ex) {
      System.err.println(ex);
    }

  }  // end main

} // end DaytimeClient
```

`DaytimeClient` reads the hostname of a daytime server from the command line and uses it to construct a new `Socket` that connects to port 13 on the server. If the hostname is omitted, the National Institute of Standards and Technology's time server at *time.nist.gov* is used. The client then calls `theSocket.getInputStream( )` to get `theSocket`'s input stream, which is stored in the variable `timeStream`. Since the daytime protocol specifies

ASCII, `DaytimeClient` doesn't bother chaining a reader to the stream. Instead, it just reads the bytes into a `StringBuffer` one at a time, breaking when the server closes the connection as the protocol requires it to do. Here's what happens:

```
% java DaytimeClient
It is 52956 03-11-13 04:45:28 00 0 0 706.3 UTC(NIST) * at time.nist.gov
% java DaytimeClient vision.poly.edu
It is Wed Nov 12 23:45:29 2003 at vision.poly.edu
```

You can see that the clocks on time.nist.gov and vision.poly.edu aren't perfectly synchronized. Differences of a few seconds can be caused by the time it takes packets to travel across the Internet. For more details about network timekeeping, see http://www.boulder.nist.gov/timefreq/service/its.htm.

On top of that problem, the time servers on these two hosts use different formats. The daytime protocol doesn't specify the format for the time it returns, other than that it be human-readable. Therefore, it is difficult to convert the character data that the server returns to a Java `Date` in a reliable fashion. If you want to create a `Date` object based on the time at the server, it's easier to use the time protocol from RFC 868 instead, because it specifies a format for the time.

When reading data from the network, it's important to keep in mind that not all protocols use ASCII or even text. For example, the time protocol specified in RFC 868 specifies that the time be sent as the number of seconds since midnight, January 1, 1900 Greenwich Mean Time. However, this is not sent as an ASCII string like "2,524,521,600" or "-1297728000". Rather, it is sent as a 32-bit, unsigned, big-endian binary number.

The RFC never actually comes out and says that this is the format used. It specifies 32 bits and assumes you know that all network protocols use big-endian numbers. The fact that the number is unsigned can be determined only by calculating the wraparound date for signed and unsigned integers and comparing it to the date given in the specification (2036). To make matters worse, the specification gives an example of a negative time that can't actually be sent by time servers that follow the protocol. Time is a fairly old protocol, standardized in the early 1980s before the IETF was as careful about such issues as it is today. Nonetheless, if you find yourself implementing a not particularly well-specified protocol, you may have to do a significant amount of testing against existing implementations to figure out what you need to do. In the worst case, different existing implementations may behave differently.

Since this isn't text, you can't easily use Telnet to test such a service, and your program can't read the server response with a `Reader` or any sort of `readLine()` method. A Java program that connects to time servers must read the raw bytes and interpret them appropriately. In this example, that job is complicated by Java's lack of a 32-bit unsigned integer type. Consequently, you have to read the bytes one at a time and manually convert them into a `long` using the bitwise operators `<<` and `|`. Example 9-5 demonstrates. When speaking other protocols, you may encounter data formats even more alien to Java. For instance, a few network protocols use 64-bit fixed-point numbers. There's no shortcut to handle all possible cases. You simply have to grit your teeth and code the math you need to handle the data in whatever format the server sends.

Example 9-5. A time protocol client

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

public class TimeClient {

  public final static int    DEFAULT_PORT = 37;
  public final static String DEFAULT_HOST = "time.nist.gov";

  public static void main(String[] args) {

    String hostname = DEFAULT_HOST ;
    int port = DEFAULT_PORT;

    if (args.length > 0) {
      hostname = args[0];
    }

    if (args.length > 1) {
      try {
        port = Integer.parseInt(args[1]);
      }
      catch (NumberFormatException ex) {
        // Stay with the default port
      }
    }

    // The time protocol sets the epoch at 1900,
    // the Java Date class at 1970. This number
```

```
        // converts between them.

        long differenceBetweenEpochs = 2208988800L;

        // If you'd rather not use the magic number, uncomment
        // the following section which calculates it directly.

        /*
        TimeZone gmt = TimeZone.getTimeZone("GMT");
        Calendar epoch1900 = Calendar.getInstance(gmt);
        epoch1900.set(1900, 01, 01, 00, 00, 00);
        long epoch1900ms = epoch1900.getTime( ).getTime( );
        Calendar epoch1970 = Calendar.getInstance(gmt);
        epoch1970.set(1970, 01, 01, 00, 00, 00);
        long epoch1970ms = epoch1970.getTime( ).getTime( );

        long differenceInMS = epoch1970ms - epoch1900ms;
        long differenceBetweenEpochs = differenceInMS/1000;
        */

        InputStream raw = null;
        try {
          Socket theSocket = new Socket(hostname, port);
          raw = theSocket.getInputStream( );

          long secondsSince1900 = 0;
          for (int i = 0; i < 4; i++) {
            secondsSince1900 = (secondsSince1900 << 8) | raw.read( );
          }

          long secondsSince1970
           = secondsSince1900 - differenceBetweenEpochs;
          long msSince1970 = secondsSince1970 * 1000;
          Date time = new Date(msSince1970);

          System.out.println("It is " + time + " at " + hostname);

        }  // end try
        catch (UnknownHostException ex) {
          System.err.println(ex);
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
        finally {
          try {
            if (raw != null) raw.close( );
          }
          catch (IOException ex) {}
        }

      }  // end main

    } // end TimeClient
```

Here's the output of this program from a couple of sample runs. Since the time protocol specifies Greenwich Mean Time, the previous differences between time zones are eliminated. Most of the difference that's left simply reflects the clock drift between the two machines:

```
% java TimeClient
It is Wed Nov 12 23:49:15 EST 2003 at time.nist.gov
% java TimeClient vision.poly.edu
It is Wed Nov 12 23:49:20 EST 2003 at vision.poly.edu
```

Like `DaytimeClient`, `TimeClient` reads the hostname of the server and an optional port from the command-line and uses it to construct a new `Socket` that connects to that server. If the user omits the hostname, `TimeClient` defaults to *time.nist.gov*. The default port is 37. The client then calls `theSocket.getInputStream( )` to get an input stream, which is stored in the variable `raw`. Four bytes are read from this stream and used to construct a long that represents the value of those four bytes interpreted as a 32-bit unsigned integer. This gives the number of seconds that have elapsed since 12:00 A.M., January 1, 1900 GMT (the time protocol's epoch); 2,208,988,800 seconds are subtracted from this number to get the number of seconds since 12:00 A.M., January 1, 1970 GMT (the Java `Date` class epoch). This number is multiplied by 1,000 to convert it into milliseconds. Finally, that number of milliseconds is converted into a `Date` object, which can be printed to show the current time and date.

9.3.2.6. public OutputStream getOutputStream( ) throws IOException

The `getOutputStream()` method returns a raw `OutputStream` for writing data from your application to the other end of the socket. You usually chain this stream to a more convenient class like `DataOutputStream` or `OutputStreamWriter` before using it. For performance reasons, it's a good idea to buffer it as well. For example:

```
Writer out;
try {
  Socket http = new Socket("www.oreilly.com", 80)
  OutputStream raw = http.getOutputStream( );
  OutputStream buffered = new BufferedOutputStream(raw);
  out = new OutputStreamWriter(buffered, "ASCII");
  out.write("GET / HTTP 1.0\r\n\r\n");
  // read the server response...
}
catch (Exception ex) {
  System.err.println(ex);
}
finally {
  try {
    out.close( );
  }
  catch (Exception ex) {}
}
```

The echo protocol, defined in RFC 862, is one of the simplest interactive TCP services. The client opens a socket to port 7 on the echo server and sends data. The server sends the data back. This continues until the client closes the connection. The echo protocol is useful for testing the network to make sure that data is not mangled by a misbehaving router or firewall. You can test echo with Telnet like this:

```
% telnet rama.poly.edu 7
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
This is a test
This is a test
This is another test
This is another test
9876543210
9876543210
^]
telnet> close
Connection closed.
```

Example 9-6 uses `getOutputStream()` and `getInputStream( )` to implement a simple echo client. The user types input on the command line which is then sent to the server. The server echoes it back. The program exits when the user types a period on a line by itself. The echo protocol does not specify a character encoding. Indeed, what it specifies is that the data sent to the server is exactly the data returned by the server. The server echoes the raw bytes, not the characters they represent. Thus, this program uses the default character encoding and line separator of the client system for reading the input from `System.in`, sending the data to the remote system, and typing the output on `System.out`. Since an echo server echoes exactly what is sent, it's as if the server dynamically adjusts itself to the client system's conventions for character encoding and line breaks. Consequently, we can use convenient classes and methods such as `PrintWriter` and `readLine( )` that would normally be too unreliable.

Example 9-6. An echo client

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

public class EchoClient {

  public static void main(String[] args) {

    String hostname = "localhost";

    if (args.length > 0) {
      hostname = args[0];
    }

    PrintWriter out = null;
```

```
        BufferedReader networkIn = null;
        try {
          Socket theSocket = new Socket(hostname, 7);
          networkIn = new BufferedReader(
           new InputStreamReader(theSocket.getInputStream( )));
          BufferedReader userIn = new BufferedReader(
           new InputStreamReader(System.in));
          out = new PrintWriter(theSocket.getOutputStream( ));
          System.out.println("Connected to echo server");

          while (true) {
            String theLine = userIn.readLine( );
            if (theLine.equals(".")) break;
            out.println(theLine);
            out.flush( );
            System.out.println(networkIn.readLine( ));
          }

        }  // end try
        catch (IOException ex) {
          System.err.println(ex);
        }
        finally {
          try {
            if (networkIn != null) networkIn.close( );
            if (out != null) out.close( );
          }
          catch (IOException ex) {}
        }

      }  // end main

    }  // end EchoClient
```

As usual, `EchoClient` reads the name of the host to connect to from the command line. This hostname is used to create a new `Socket` object on port 7, called `theSocket`. The socket's `InputStream` is returned by `getInputStream( )` and chained to an `InputStreamReader`, which is chained to a `BufferedReader` called `networkIn`. This reader reads the server responses. Since this client also needs to read input from the user, it creates a second `BufferedReader`, this one called `userIn`, which reads from `System.in`. Next, `EchoClient` calls `theSocket.getOutputStream( )` to get `theSocket`'s output stream, which is used to construct a new `PrintWriter` called `out`.

Now that the three streams have been created, it's simply a matter of reading the data from `userIn` and writing that data back out onto `out`. Once data has been sent to the echo server, `networkIn` waits for a response. When `networkIn` receives a response, it's printed on `System.out`. In theory, this client could get hung waiting for a response that never comes. However, this is unlikely if the connection can be made in the first place, since the TCP protocol checks for bad packets and automatically asks the server for replacements. When we implement a UDP echo client in Chapter 13, we will need a different approach because UDP does no error checking. Here's a sample run:

```
% java EchoClient rama.poly.edu
Connected to echo server
Hello
Hello
How are you?
How are you?
I'm fine thank you.
I'm fine thank you.
Goodbye
Goodbye
.
```

Example 9-7 is line-oriented. It reads a line of input from the console, sends it to the server, and waits to read a line of output it gets back. However, the echo protocol doesn't require this. It echoes each byte as it receives it. It doesn't really care whether those bytes represent characters in some encoding or are divided into lines. Java does not allow you to put the console into "raw" mode, where each character is read as soon as it's typed instead of waiting for the user to press the Enter key. Consequently, if you want to explore the more immediate echo responses, you must provide a nonconsole interface. You also have to separate the network input from user input and network output. This is because the connection is full duplex but may be subject to some delay. If the Internet is running slow, the user may be able to type and send several characters before the server returns the first one. Then the server may return several bytes all at once. Unlike many protocols, echo does not specify lockstep behavior in which the client sends a request but then waits for the full server response before sending any more data. The simplest way to handle such a protocol in Java is to place network input and output in separate threads.

9.3.3. Closing the Socket

That's almost everything you need to know about client-side sockets. When you're writing a client application, almost all the work goes into handling the streams and interpreting the data. The sockets themselves are very easy to work with; all the hard parts are hidden. That is one reason sockets are such a popular paradigm for network programming. After we cover a couple of remaining methods, you'll know everything you need to know to write TCP clients.

9.3.3.1. public void close( ) throws IOException

Until now, the examples have assumed that sockets close on their own; they haven't done anything to clean up after themselves. It is true that a socket closes automatically when one of its two streams closes, when the program ends, or when it's garbage collected. However, it is a bad practice to assume that the system will close sockets for you, especially for programs that may run for an indefinite period of time. In a socket-intensive program like a web browser, the system may well hit its maximum number of open sockets before the garbage collector kicks in. The port scanner programs of Example 9-1 and Example 9-2 are particularly bad offenders in this respect, since it may take a long time for the program to run through all the ports. Shortly, you'll see a new version that doesn't have this problem.

When you're through with a socket, you should call its `close( )` method to disconnect. Ideally, you put this in a `finally` block so that the socket is closed whether an exception is thrown or not. The syntax is straightforward:

```
Socket connection = null;
try {
  connection = new Socket("www.oreilly.com", 13);
  // interact with the socket...
}  // end try
catch (UnknownHostException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
finally {
  if (connection != null) connection.close( );
}
```

Once a `Socket` has been closed, its `InetAddress`, port number, local address, and local port number are still accessible through the `getInetAddress()`, `getPort( )`, `getLocalAddress( )`, and `getLocalPort( )` methods. However, although you can still call `getInputStream( )` or `getOutputStream( )`, attempting to read data from the `InputStream` or write data to the `OutputStream` throws an `IOException`.

Example 9-7 is a revision of the `PortScanner` program that closes each socket once it's through with it. It does not close sockets that fail to connect. Since these are never opened, they don't need to be closed. In fact, if the constructor failed, `connection` is actually `null`.

Example 9-7. Look for ports with socket closing

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

public class PortScanner {

  public static void main(String[] args) {
    String host = "localhost";

    if (args.length > 0) {
      host = args[0];
    }

    try {
      InetAddress theAddress = InetAddress.getByName(host);
      for (int i = 1; i < 65536; i++) {
        Socket connection = null;
        try {
          connection = new Socket(host, i);
          System.out.println("There is a server on port "
           + i + " of " + host);
        }
        catch (IOException ex) {
          // must not be a server on this port
        }
        finally {
          try {
            if (connection != null) connection.close( );
          }
          catch (IOException ex) {}
        }
      } // end for
    } // end try
    catch (UnknownHostException ex) {
      System.err.println(ex);
    }

  }  // end main

}  // end PortScanner
```

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

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

If you're uncertain about a socket's state, you can check it with this method rather than risking an `IOException`. For example,

```
if (socket.isClosed( )) {
  // do something...
}
else {
  // do something else...
}
```

However, this is not a perfect test. If the socket has never been connected in the first place, `isClosed( )` returns false, even though the socket isn't exactly open.

Java 1.4 also adds an `isConnected()` method:

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

The name is a little misleading. It does not tell you if the socket is currently connected to a remote host (that is, if it is unclosed). Instead it tells you whether the socket has ever been connected to a remote host. If the socket was able to connect to the remote host at all, then this method returns true, even after that socket has been closed. To tell if a socket is currently open, you need to check that `isConnected( )` returns true and `isClosed()` returns false. For example:

```
boolean connected = socket.isConnected( ) && ! socket.isClosed( );
```

Java 1.4 also adds an `isBound()` method:

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

Whereas `isConnected( )` refers to the remote end of the socket, `isBound( )` refers to the local end. It tells you whether the socket successfully bound to the outgoing port on the local system. This isn't very important in practice. It will become more important when we discuss server sockets in the next chapter.

9.3.3.2. Half-closed sockets // Java 1.3

The `close( )` method shuts down both input and output from the socket. On occasion, you may want to shut down only half of the connection, either input or output. Starting in Java 1.3, the `shutdownInput( )` and `shutdownOutput()` methods let you close only half of the connection:

```
public void shutdownInput( ) throws IOException  // Java 1.3
public void shutdownOutput( ) throws IOException // Java 1.3
```

This doesn't actually close the socket. However, it does adjust the stream connected to it so that it thinks it's at the end of the stream. Further reads from the input stream will return -1. Further writes to the output stream will throw an `IOException`.

Many protocols, such as finger, whois, and HTTP begin with the client sending a request to the server, then reading the response. It would be possible to shut down the output after the client has sent the request. For example, this code fragment sends a request to an HTTP server and then shuts down the output, since it won't need to write anything else over this socket:

```
Socket connection = null;
try {
  connection = new Socket("www.oreilly.com", 80);
  Writer out = new OutputStreamWriter(
   connection.getOutputStream( ), "8859_1");
  out.write("GET / HTTP 1.0\r\n\r\n");
  out.flush( );
  connection.shutdownOutput( );
  // read the response...
}
catch (IOException ex) {
}
finally {
  try {
    if (connection != null) connection.close( );
   }
   catch (IOException ex) {}
}
```

Notice that even though you shut down half or even both halves of a connection, you still need to close the socket when you're through with it. The shutdown methods simply affect the socket's streams. They don't release the resources associated with the socket such as the port it occupies.

Java 1.4 adds two methods that tell you whether the input and output streams are open or closed:

```
public boolean isInputShutdown( )   // Java 1.4
public boolean isOutputShutdown( )  // Java 1.4
```

You can use these (rather than `isConnected( )` and `isClosed( )`) to more specifically ascertain whether you can read from or write to a socket.

9.3.4. Setting Socket Options

Socket options specify how the native sockets on which the Java `Socket` class relies send and receive data. You can set four options in Java 1.1, six in Java 1.2, seven in Java 1.3, and eight in Java 1.4:

- TCP_NODELAY
- SO_BINDADDR
- SO_TIMEOUT
- SO_LINGER
- SO_SNDBUF (Java 1.2 and later)
- SO_RCVBUF (Java 1.2 and later)
- SO_KEEPALIVE (Java 1.3 and later)
- OOBINLINE (Java 1.4 and later)

The funny-looking names for these options are taken from the named constants in the C header files used in Berkeley Unix where sockets were invented. Thus they follow classic Unix C naming conventions rather than the more legible Java naming conventions. For instance, SO_SNDBUF really means "Socket Option Send Buffer Size."

9.3.4.1. TCP_NODELAY

```
public void setTcpNoDelay(boolean on) throws SocketException
public boolean getTcpNoDelay( ) throws SocketException
```

Setting TCP_NODELAY to true ensures that packets are sent as quickly as possible regardless of their size. Normally, small (one-byte) packets are combined into larger packets before being sent. Before sending another packet, the local host waits to receive acknowledgment of the previous packet from the remote system. This is known as *Nagle's algorithm*. The problem with Nagle's algorithm is that if the remote system doesn't send acknowledgments back to the local system fast enough, applications that depend on the steady transfer of small bits of information may slow down. This issue is especially problematic for GUI programs such as games or network computer applications where the server needs to track client-side mouse movement in real time. On a really slow network, even simple typing can be too slow because of the constant buffering. Setting TCP_NODELAY to true defeats this buffering scheme, so that all packets are sent as soon as they're ready.

`setTcpNoDelay(true)` turns off buffering for the socket. `setTcpNoDelay(false)` turns it back on. `getTcpNoDelay( )` returns `true` if buffering is off and `false` if buffering is on. For example, the following fragment turns off buffering (that is, it turns on TCP_NODELAY) for the socket `s` if it isn't already off:

```
if (!s.getTcpNoDelay( )) s.setTcpNoDelay(true);
```

These two methods are each declared to throw a `SocketException`. They will be thrown only if the underlying socket implementation doesn't support the TCP_ NODELAY option.

9.3.4.2. SO_LINGER

```
public void setSoLinger(boolean on, int seconds) throws SocketException
public int getSoLinger( ) throws SocketException
```

The SO_LINGER option specifies what to do with datagrams that have not yet been sent when a socket is closed. By default, the `close( )` method returns immediately; but the system still tries to send any remaining data. If the linger time is set to zero, any unsent packets are thrown away when the socket is closed. If the linger time is any positive value, the `close`

`( )` method blocks while waiting the specified number of seconds for the data to be sent and the acknowledgments to be received. When that number of seconds has passed, the socket is closed and any remaining data is not sent, acknowledgment or no.

These two methods each throw a `SocketException` if the underlying socket implementation does not support the SO_LINGER option. The `setSoLinger()` method can also throw an `IllegalArgumentException` if you try to set the linger time to a negative value. However, the `getSoLinger( )` method may return -1 to indicate that this option is disabled, and as much time as is needed is taken to deliver the remaining data; for example, to set the linger timeout for the `Socket` `s` to four minutes, if it's not already set to some other value:

```
if (s.getTcpSoLinger( ) == -1) s.setSoLinger(true, 240);
```

The maximum linger time is 65,535 seconds. Times larger than that will be reduced to 65,535 seconds. Frankly, 65,535 seconds (more than 18 hours) is much longer than you actually want to wait. Generally, the platform default value is more appropriate.

9.3.4.3. SO_TIMEOUT

```
public void setSoTimeout(int milliseconds)
 throws SocketException
publicint getSoTimeout( ) throws SocketException
```

Normally when you try to read data from a socket, the `read()` call blocks as long as necessary to get enough bytes. By setting SO_TIMEOUT, you ensure that the call will not block for more than a fixed number of milliseconds. When the timeout expires, an `InterruptedIOException` is thrown, and you should be prepared to catch it. However, the socket is still connected. Although this `read( )` call failed, you can try to read from the socket again. The next call may succeed.

Timeouts are given in milliseconds. Zero is interpreted as an infinite timeout; it is the default value. For example, to set the timeout value of the `Socket` object `s` to 3 minutes if it isn't already set, specify 180,000 milliseconds:

```
if (s.getSoTimeout( ) == 0) s.setSoTimeout(180000);
```

These two methods each throw a `SocketException` if the underlying socket implementation does not support the SO_TIMEOUT option. The `setSoTimeout()` method also throws an `IllegalArgumentException` if the specified timeout value is negative.

9.3.4.4. SO_RCVBUF

Most TCP stacks use buffers to improve network performance. Larger buffers tend to improve performance for reasonably fast (say, 10Mbps and up) connections while slower, dialup connections do better with smaller buffers. Generally, transfers of large, continuous blocks of data, which are common in file transfer protocols such as FTP and HTTP, benefit from large buffers, while the smaller transfers of interactive sessions, such as Telnet and many games, do not. Relatively old operating systems designed in the age of small files and slow networks, such as BSD 4.2, use 2-kilobyte buffers. Somewhat newer systems, such as SunOS 4.1.3, use larger 4-kilobyte buffers by default. Still newer systems, such as Solaris, use 8- or even 16-kilobyte buffers. Starting in Java 1.2, there are methods to get and set the suggested receive buffer size used for network input:

```
public void setReceiveBufferSize(int size)// Java 1.2
 throws SocketException, IllegalArgumentException
public int getReceiveBufferSize( ) throws SocketException  // Java 1.2
```

The `getReceiveBufferSize()` method returns the number of bytes in the buffer that can be used for input from this socket. It throws a `SocketException` if the underlying socket implementation does not recognize the SO_RCVBUF option. This might happen on a non-POSIX operating system.

The `setReceiveBufferSize()` method suggests a number of bytes to use for buffering output on this socket. However, the underlying implementation is free to ignore this suggestion. The `setReceiveBufferSize( )` method throws an `IllegalArgumentException` if its argument is less than or equal to zero. Although it's declared to also throw `SocketException`, it probably won't in practice since a `SocketException` is thrown for the same reason as `IllegalArgumentException` and the check for the `IllegalArgument Exception` is made first.

9.3.4.5. SO_SNDBUF

Starting in Java 1.2, there are methods to get and set the suggested send buffer size used for network output:

```
public void setSendBufferSize(int size)               // Java 1.2
 throws SocketException, IllegalArgumentException
public int getSendBufferSize( ) throws SocketException  // Java 1.2
```

The `getSendBufferSize()` method returns the number of bytes in the buffer used for output on this socket. It throws a `SocketException` if the underlying socket implementation doesn't understand the `SO_SNDBUF` option.

The `setSendBufferSize( )` method suggests a number of bytes to use for buffering output on this socket. However, again thegggg client is free to ignore this suggestion. The `setSendBufferSize( )` method also throws a `SocketException` if the underlying socket implementation doesn't understand the `SO_SNDBUF` option. However, it throws an `IllegalArgumentException` if its argument is less than or equal to zero.

9.3.4.6. SO_KEEPALIVE

If SO_KEEPALIVE is turned on, the client will occasionally send a data packet over an idle connection (most commonly once every two hours), just to make sure the server hasn't crashed. If the server fails to respond to this packet, the client keeps trying for a little more than 11 minutes until it receives a response. If it doesn't receive a response within 12 minutes, the client closes the socket. Without SO_KEEPALIVE, an inactive client could live more or less forever without noticing that the server had crashed.

Java 1.3 adds methods to turn SO_KEEPALIVE on and off and to determine its current state:

```
public void setKeepAlive(boolean on) throws SocketException // Java 1.3
public boolean getKeepAlive( ) throws SocketException // Java 1.3
```

The default for SO_KEEPALIVE is false. This code fragment turns SO_KEEPALIVE off, if it's turned on:

```
if (s.getKeepAlive( )) s.setKeepAlive(false);
```

9.3.4.7. OOBINLINE // Java 1.4

TCP includes a feature that sends a single byte of "urgent" data. This data is sent immediately. Furthermore, the receiver is notified when the urgent data is received and may elect to process the urgent data before it processes any other data that has already been received.

Java 1.4 adds support for both sending and receiving such urgent data. The sending method is named, obviously enough, `sendUrgentData( )`:

```
public void sendUrgentData(int data) throws IOException  // Java 1.4
```

This method sends the lowest order byte of its argument almost immediately. If necessary, any currently cached data is flushed first.

How the receiving end responds to urgent data is a little confused, and varies from one platform and API to the next. Some systems receive the urgent data separately from the regular data. However, the more common, more modern approach is to place the urgent data in the regular received data queue in its proper order, tell the application that urgent data is available, and let it hunt through the queue to find it.

By default, Java pretty much ignores urgent data received from a socket. However, if you want to receive urgent data inline with regular data, you need to set the OOBINLINE option to true using these methods:

```
public void setOOBInline(boolean on) throws SocketException // Java 1.3
public boolean getOOBInline( ) throws SocketException // Java 1.3
```

The default for OOBInline is false. This code fragment turns OOBInline on, if it's turned off:

```
if (s.getOOBInline( )) s.setOOBInline(true);
```

Once OOBInline is turned on, any urgent data that arrives will be placed on the socket's input stream to be read in the usual way. Java does not distinguish it from non-urgent data.

9.3.4.8. SO_REUSEADDR // Java 1.4

When a socket is closed, it may not immediately release the local address, especially if a connection was open when the socket was closed. It can sometimes wait for a small amount of time to make sure it receives any lingering packets that were addressed to the port that were still crossing the network when the socket was closed. The system won't do anything with any of the late packets it receives. It just wants to make sure they don't accidentally get fed into a new process that has bound to the same port.

This isn't a big problem on a random port, but it can be an issue if the socket has bound to a well-known port because it prevents any other socket from using that port in the meantime. If the SO_REUSEADDR is turned on (it's turned off by default), another socket is allowed to bind to the port even while data may be outstanding for the previous socket.

In Java this option is controlled by these two methods:

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

For this to work, `setReuseAddress()` must be called *before* the new socket binds to the port. This means the socket must be created in an unconnected state using the no-args constructor; then `setReuseAddress(true)` is called, and the socket is connected using the `connect( )` method. Both the socket that was previously connected and the new socket reusing the old address must set SO_REUSEADDR to true for it to take effect.

9.3.5. Class of Service

In the last few years, a lot of thought has gone into deriving different classes of service for different types of data that may be transferred across the Internet. For instance, video needs relatively high bandwidth and low latency for good performance, whereas email can be passed over low-bandwidth connections and even held up for several hours without major harm. It might be wise to price the different classes of service differentially so that people won't ask for the highest class of service automatically. After all, if sending an overnight letter cost the same as sending a package via media mail, we'd all just use Fed Ex overnight, which would quickly become congested and overwhelmed. The Internet is no different.

Currently, four traffic classes have been defined for TCP data, although not all routers and native TCP stacks support them. These classes are low cost, high reliability, maximum throughput, and minimum delay. Furthermore, they can be combined. For instance, you can request the minimum delay available at low cost. These measure are all fuzzy and relative, not hard and fast guarantees of service.

Java lets you inspect and set the class of service for a socket using these two methods:

```
public int getTrafficClass( ) throws SocketException
public void setTrafficClass(int trafficClass) throws SocketException
```

The traffic class is given as an int between 0 and 255. (Values outside this range cause `IllegalArgumentException`s.) This int is a combination of bit-flags. Specifically:

- 0x02: Low cost
- 0x04: High reliability
- 0x08: Maximum throughput
- 0x10: Minimum delay

The lowest order, ones bit must be zero. The other three high order bits are not yet used. For example, this code fragment requests a low cost connection:

```
Socket s = new Socket("www.yahoo.com", 80);
s.setTrafficClass(0x02);
```

This code fragment requests a connection with maximum throughput and minimum delay:

```
Socket s = new Socket("www.yahoo.com", 80);
s.setTrafficClass(0x08 | 0x10);
```

The underlying socket implementation is not required to respect any of these requests. They only provide a hint to the TCP stack about the desired policy. Many implementations ignore these values completely. If the TCP stack is unable to provide the requested class of service, it may but is not required to throw a `SocketException`.

Java does not provide any means to access pricing information for the different classes of service. Be aware that your ISP may charge you for faster or more reliable connections using these features. (If they make it available at all. This is all still pretty bleeding edge stuff.)

Java 1.5 adds a slightly different method to set preferences, 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 documentation even suggests using non-TCP/IP sockets, though it's not at all clear what that means.

9.3.6. The Object Methods

The `Socket` class overrides only one of the standard methods from `java.lang.Object`, `toString( )`. Since sockets are transitory objects that typically last only as long as the connection they represent, there's not much need or purpose to storing them in hash tables or comparing them to each other. Therefore, `Socket` does not override `equals( )` or `hashCode( )`, and the semantics for these methods are those of the `Object` class. Two `Socket` objects are equal to each other if and only if they are the same socket.

9.3.6.1. public String toString( )

The `toString( )` method produces a string that looks like this:

```
Socket[addr=www.oreilly.com/198.112.208.11,port=80,localport=50055]
```

This is ugly and useful primarily for debugging. Don't rely on this format; it may change in the future. All parts of this string are accessible directly through other methods (specifically `getInetAddress( )`, `getPort( )`, and `getLocalPort( )`).

9.4. Socket Exceptions

Most methods of the `Socket` class are declared to throw `IOException or its subclass, java.net.SocketException`:

```
public class SocketException extends IOException
```

However, knowing that a problem occurred is often not sufficient to deal with the problem. Did the remote host refuse the connection because it was busy? Did the remote host refuse the connection because no service was listening on the port? Did the connection attempt timeout because of network congestion or because the host was down? There are several subclasses of `SocketException` that provide more information about what went wrong and why:

```
public class BindException extends SocketException
public class ConnectException extends SocketException
public class NoRouteToHostException extends SocketException
```

A `BindException` is thrown if you try to construct a `Socket` or `ServerSocket` object on a local port that is in use or that you do not have sufficient privileges to use. A `ConnectException` is thrown when a connection is refused at the remote host, which usually happens because the host is busy or no process is listening on that port. Finally, a `NoRouteToHostException` indicates that the connection has timed out.

The `java.net` package also includes `ProtocolException`, a direct subclass of `IOException`:

```
public class ProtocolException extends IOException
```

This is thrown when data is received from the network that somehow violates the TCP/IP specification.

None of these exception classes have any special methods you wouldn't find in any other exception class, but you can take advantage of these subclasses to provide more informative error messages or to decide whether retrying the offending operation is likely to be successful.

9.5. Socket Addresses

The `SocketAddress` class introduced in Java 1.4 represents a connection endpoint. The actual `java.net.SocketAddress` class is an empty abstract class with no methods aside from a default constructor:

```
package java.net.*;

public abstract class SocketAddress {

  public SocketAddress( ) {}

}
```

At least theoretically, this class can be used for both TCP and non-TCP sockets. Subclasses of `SocketAddress` provide more detailed information appropriate for the type of socket. In practice, only TCP/IP sockets are currently supported.

The primary purpose of the `SocketAddress` class is to provide a convenient store for transient socket connection information such as the IP address and port that can be reused to create new sockets, even after the original socket is disconnected and garbage collected. To this end, the `Socket` class offers two methods that return `SocketAddress` objects: `getRemoteSocketAddress( )` returns the address of the system being connected to and `getLocalSocketAdddress( )` returns the address from which the connection is made:

```
public SocketAddress getRemoteSocketAddress( )
public SocketAddress getLocalSocketAddress( )
```

Both of these methods return null if the socket is not yet connected.

A `SocketAddress` is necessary to connect an unconnected socket via the `connect( )` method:

```
public void connect(SocketAddress endpoint) throws IOException
```

For example, first you might connect to Yahoo, then store its address:

```
Socket socket = new Socket("www.yahoo.com", 80);
SocketAddress yahoo = socket.getRemoteSocketAddress( );
socket.close( );
```

Later, you could reconnect to Yahoo using this address:

```
socket = new Socket( );
socket.connect(yahoo);
```

Not all socket implementations can use the same subclasses of `SocketAddress`. If an instance of the wrong type is passed to `connect( )`, it throws an `IllegalArgumentException`.

You can pass an `int` as the second argument to specify the number of milliseconds to wait before the connection times out:

```
public void connect(SocketAddress endpoint, int timeout) throws IOException
```

The default, 0, means wait forever.

9.6. Examples

HotJava was one of the first large-scale Java programs; it's a web browser that was easily the equal of the early versions of Mosaic. HotJava has been discontinued, but there are numerous network-aware applications written in Java, including the LimeWire Gnutella client, the Eclipse IDE, and the JBoss application server. It is completely possible to write commercial-quality applications in Java; and it is especially possible to write network-aware applications, both clients and servers. This section shows two network clients, finger and whois, to illustrate this point. I stop short of what could be done, but only in the user interface. All the necessary networking code is present. Indeed, once again we find out that network code is easy; it's user interfaces that are hard.

9.6.1. Finger

Finger is a straightforward protocol described in RFC 1288. The client makes a TCP connection to the server on port 79 and sends a one-line query; the server responds to the query and closes the connection. The format of the query is precisely defined, the format of the response somewhat less so. All data transferred should probably be pure printable ASCII text, although unfortunately, the specification contradicts itself repeatedly on this point. The specification also recommends that clients filter out any non-ASCII data they do receive, at least by default. All lines must end with a carriage return/linefeed pair (`\r\n` in Java parlance).

Failure to filter nonprintable characters allows mischievous users to configure their *.plan* files to reset people's terminals, switch them into graphics mode, or play other tricks accessible to those with intimate knowledge of VT-terminal escape sequences. While amusing to experienced users who recognize what's going on and appreciate the hack value of such *.plan* files, these tricks do confuse and terrify the uninitiated.

The simplest allowable request from the client is a bare carriage return/linefeed pair, which is usually interpreted as a request to show a list of the currently logged-in users. For example:

```
% telnet rama.poly.edu 79
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.

Login       Name            TTY    Idle    When    Where
jacola   Jane Colaginae    *pts/7       Tue 08:01  208.34.37.104
marcus   Marcus Tullius     pts/15  13d Tue 17:33  farm-dialup11.poly.e
matewan  Sepin Matewan     *pts/17  17: Thu 15:32  128.238.10.177
hengpi   Heng Pin          *pts/10      Tue 10:36  128.238.18.119
nadats   Nabeel Datsun      pts/12   56 Mon 10:38  128.238.213.227
matewan  Sepin Matewan     *pts/8     4 Sun 18:39  128.238.10.177
Connection closed by foreign host.
```

It is also possible to request information about a specific user or username by including that user or username on the query line:

```
% telnet rama.poly.edu 79
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
marcus
Login       Name            TTY    Idle    When    Where
marcus   Marcus Tullius     pts/15  13d Tue 17:33  farm-dialup11.poly.e
```

The information that finger servers return typically includes the user's full name, where he's connected from, how long he has been connected, and any other information he has chosen to make available in his *.plan* file. A few servers put finger to other uses; for example, several sites give you a list of recent earthquake activity. Vending machines connected to the Internet return a list of items available for purchase. It is possible to request information about users via their first name, last name, or login name. You can also request information about more than one user at a time like this:

```
% telnet rama.poly.edu 79
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
marcus nadats matewan
Login       Name              TTY    Idle    When    Where
marcus   Marcus Tullius    pts/15   13d Tue 17:33  farm-dialup11.poly.e
nadats   Nabeel Datsun     pts/12    59 Mon 10:38  128.238.213.227
matewan  Sepin Matewan    *pts/17   17: Thu 15:32  128.238.10.177
matewan  Sepin Matewan    *pts/8      8 Sun 18:39  128.238.10.177
Connection closed by foreign host.
```

In this section, we'll develop a Java finger client that allows users to specify a hostname on the command line, followed by zero or more usernames. For example, a typical command line will look like:

```
% java FingerClient hostname user1 user2 ...
```

`FingerClient` connects to port 79 on the specified host. The socket's `OutputStream` is chained to an `OutputStreamWriter` using the ISO 8859-1 encoding, which sends a line consisting of all the names on the command line, followed by a carriage return and a linefeed. Next, the output from the server (which is input to the program) is taken from `theSocket.getInputStream( )` and chained first to a `BufferedInputStream` for performance and then to an `InputStreamReader` so the server response can be read as text. The server's output is presented to the user on System.out. Example 9-8 shows the code.

Example 9-8. A Java command-line finger client

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

public class FingerClient {

  public final static int DEFAULT_PORT = 79;

  public static void main(String[] args) {

    String hostname = "localhost";

    try {
      hostname = args[0];
    }
    catch (ArrayIndexOutOfBoundsException ex) {
      hostname = "localhost";
    }

    Socket connection = null;
    try {
      connection = new Socket(hostname, DEFAULT_PORT);
```

```
          Writer out = new OutputStreamWriter(
           connection.getOutputStream( ), "8859_1");
          for (int i = 1; i < args.length; i++) out.write(args[i] + " ");
          out.write("\r\n");
          out.flush( );
          InputStream raw = connection.getInputStream( );
          BufferedInputStream buffer = new BufferedInputStream(raw);
          InputStreamReader in = new InputStreamReader(buffer, "8859_1");
          int c;
          while ((c = in.read( )) != -1) {
           // filter non-printable and non-ASCII as recommended by RFC 1288
            if ((c >= 32 && c < 127) || c == '\t' || c == '\r' || c == '\n')
            {
              System.out.write(c);
            }
          }
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
        finally {
          try {
            if (connection != null) connection.close( );
          }
          catch (IOException ex) {}
        }

      }

    }
```

Here are some samples of this program running:

```
D:\JAVA\JNP2\examples\10>java FingerClient rama.poly.edu
Login    Name              TTY      Idle    When      Where
jacolag  Jane Colaginae   *pts/7            Tue 08:01  208.34.37.104
hengpi   Heng Pin          pts/9    5       Tue 14:09  128.238.18.119
marcus   Marcus Tullius    pts/15   13d     Tue 17:33  farm-dialup11.
                                                       poly.e
matewan  Sepin Matewan    *pts/17   17:     Thu 15:32  128.238.10.177
hengpi   Heng Pin         *pts/10           Tue 10:36  128.238.18.119
nadats   Nabeel Datsun     pts/12   1:05    Mon 10:38  128.238.213.227
nadats   Nabeel Datsun     pts/12   1:05    Mon 10:38  128.238.213.227
matewan  Sepin Matewan    *pts/8    14      Sun 18:39  128.238.10.177

D:\JAVA\JNP2\examples\10>java FingerClient rama.poly.edu marcus
Login    Name              TTY      Idle    When       Where
Marcus   Marcus Tullius    pts/15   13d     Tue 17:33  farm-dialup11.
                                                       poly.e
```

9.6.2. Whois

Whois is a simple directory service protocol defined in RFC 954; it was originally designed to keep track of administrators responsible for Internet hosts and domains. A whois client connects to one of several central servers and requests directory information for a person or persons; it can usually give you a phone number, an email address, and a snail mail address (not necessarily current ones, though). With the explosive growth of the Internet, flaws have become apparent in the whois protocol, most notably its centralized nature. A more complex replacement called whois++ is documented in RFCs 1913 and 1914 but has not been widely implemented.

Let's begin with a simple client to connect to a whois server. The basic structure of the whois protocol is:

1. The client opens a TCP socket to port 43 on the server.
2. The client sends a search string terminated by a carriage return/linefeed pair (`\r\n`). The search string can be a name, a list of names, or a special command, as discussed below. You can also search for domain names, like oreilly.com or netscape.com, which give you information about a network.
3. The server sends an unspecified amount of human-readable information in response to the command and closes the connection.
4. The client displays this information to the user.

The search string the client sends has a fairly simple format. At its most basic, it's just the name of the person you're searching for. Here's a simple whois search for "Harold":

```
% telnet whois.internic.net 43
Trying 198.41.0.6...
Connected to whois.internic.net.
Escape character is '^]'.
Harold

Whois Server Version 1.3

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

HAROLD.NET
HAROLD.COM

To single out one record, look it up with "xxx", where xxx is one of the
of the records displayed above. If the records are the same, look them up
with "=xxx" to receive a full display for each record.

>>> Last update of whois database: Tue, 16 Dec 2003 18:36:16 EST <<<

NOTICE: The expiration date displayed in this record is the date the
registrar's sponsorship of the domain name registration in the registry is
currently set to expire. This date does not necessarily reflect the expiration
date of the domain name registrant's agreement with the sponsoring
```

```
registrar.  Users may consult the sponsoring registrar's Whois database to
view the registrar's reported date of expiration for this registration.

TERMS OF USE: You are not authorized to access or query our Whois
database through the use of electronic processes that are high-volume and
automated except as reasonably necessary to register domain names or
modify existing registrations; the Data in VeriSign Global Registry
Services' ("VeriSign") Whois database is provided by VeriSign for
information purposes only, and to assist persons in obtaining information
about or related to a domain name registration record. VeriSign does not
guarantee its accuracy. By submitting a Whois query, you agree to abide
by the following terms of use: You agree that you may use this Data only
for lawful purposes and that under no circumstances will you use this Data
to: (1) allow, enable, or otherwise support the transmission of mass
unsolicited, commercial advertising or solicitations via e-mail, telephone,
or facsimile; or (2) enable high volume, automated, electronic processes
that apply to VeriSign (or its computer systems). The compilation,
repackaging, dissemination or other use of this Data is expressly
prohibited without the prior written consent of VeriSign. You agree not to
use electronic processes that are automated and high-volume to access or
query the Whois database except as reasonably necessary to register
domain names or modify existing registrations. VeriSign reserves the right
to restrict your access to the Whois database in its sole discretion to ensure
operational stability.  VeriSign may restrict or terminate your access to the
Whois database for failure to abide by these terms of use. VeriSign
reserves the right to modify these terms at any time.

The Registry database contains ONLY .COM, .NET, .EDU domains and
Registrars.
Connection closed by foreign host.
```

Although the previous input has a pretty clear format, that format is regrettably nonstandard. Different whois servers can and do send decidedly different output. For example, here are the first couple of results from the same search at the main French whois server, *whois.nic.fr*:

```
% telnet whois.nic.fr 43
telnet whois.nic.fr 43
Trying 192.134.4.18...
Connected to winter.nic.fr.
Escape character is '^]'.
Harold

Tous droits reserves par copyright.
Voir http://www.nic.fr/outils/dbcopyright.html
Rights restricted by copyright.
See http://www.nic.fr/outils/dbcopyright.html

person:      Harold Potier
address:     ARESTE
address:     154 Avenue Du Brezet
address:     63000 Clermont-Ferrand
address:     France
phone:       +33 4 73 42 67 67
fax-no:      +33 4 73 42 67 67
nic-hdl:     HP4305-FRNIC
mnt-by:      OLEANE-NOC
changed:     hostmaster@oleane.net 20000510
```

```
changed:     migration-dbm@nic.fr 20001015
source:      FRNIC

person:      Harold Israel
address:     LE PARADIS LATIN
address:     28 rue du Cardinal Lemoine
address:     Paris, France 75005 FR
phone:       +33 1 43252828
fax-no:      +33 1 43296363
e-mail:      info@cie.fr
nic-hdl:     HI68-FRNIC
notify:      info@cie.fr
changed:     registrar@ns.il 19991011
changed:     migration-dbm@nic.fr 20001015
source:      FRNIC
```

Here each complete record is returned rather than just a list of sites. Other whois servers may use still other formats. This protocol is not at all designed for machine processing. You pretty much have to write new code to handle the output of each different whois server. However, regardless of the output format, each response likely contains a *handle*, which in the Internic output is a domain name, and in the nic.fr output is in the nic-hdl field. Handles are guaranteed to be unique, and are used to get more specific information about a person or a network. If you search for a handle, you will get at most one match. If your search only has one match, either because you're lucky or you're searching for a handle, then the server returns a more detailed record. Here's a search for *oreilly.com*. Because there is only one *oreilly.com* in the database, the server returns all the information it has on this domain:

```
% telnet whois.internic.net 43
Trying 198.41.0.6...
Connected to whois.internic.net.
Escape character is '^]'.
oreilly.com

Whois Server Version 1.3

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: OREILLY.COM
   Registrar: BULKREGISTER, LLC.
   Whois Server: whois.bulkregister.com
   Referral URL: http://www.bulkregister.com
   Name Server: NS1.SONIC.NET
   Name Server: NS.OREILLY.COM
   Status: ACTIVE
   Updated Date: 17-oct-2002
   Creation Date: 27-may-1997
   Expiration Date: 26-may-2004


>>> Last update of whois database: Tue, 16 Dec 2003 18:36:16 EST <<<
...
Connection closed by foreign host.
```

It's easy to implement a simple whois client that connects to *whois.internic.net* and searches for names entered on the command line. Example 9-9 is just such a client. The server can be changed using the WHOIS_SERVER system property, which can be set on the command line using the `-D` option. I won't claim this is an exemplary user interface, but it's simple enough to code and lets the example focus more on the interesting network parts of the problem.

Example 9-9. A command-line whois client

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

public class WhoisClient {

  public final static int DEFAULT_PORT = 43;
  public final static String DEFAULT_HOST = "whois.internic.net";

  public static void main(String[] args) {

    String serverName = System.getProperty("WHOIS_SERVER", DEFAULT_HOST);

    InetAddress server = null;
    try {
      server = InetAddress.getByName(serverName);
    }
    catch (UnknownHostException ex) {
      System.err.println("Error: Could not locate whois server "
       + server);
      System.err.println("Usage: java -DWHOIS_SERVER=hostname
                                       WhoisClient name");
      return;
    }

    try {
      Socket theSocket = new Socket(server, DEFAULT_PORT);
      Writer out = new OutputStreamWriter(theSocket.getOutputStream( ),
       "8859_1");
      for (int i = 0; i < args.length; i++) out.write(args[i] + " ");
      out.write("\r\n");
      out.flush( );
      InputStream raw = theSocket.getInputStream( );
      InputStream in  = new BufferedInputStream(theSocket.getInputStream( ));
      int c;
      while ((c = in.read( )) != -1) System.out.write(c);
    }
    catch (IOException ex) {
      System.err.println(ex);
    }

  }

}
```

The class has two `final static` fields: the `DEFAULT_PORT`, 43, and the `DEFAULT_HOST`, whois.internic.net. The host can be changed by setting the WHOISE_SERVER system property. The `main()` method begins by opening a socket to this whois server on port 43. The `Socket`'s `OutputStream` is chained to an `OutputStreamWriter`. Then each argument on the command-line is written on this stream and sent out over the socket to the whois server. A carriage return/linefeed is written and the writer is flushed.

Next, the `Socket`'s `InputStream` is stored in the variable `raw`, which is buffered using the `BufferedInputStream in`. Since whois is known to use ASCII, bytes are read from this stream with `read( )` and copied onto `System.out` until `read( )` returns -1, signaling the end of the server's response. Each character is simply copied onto `System.out`.

The whois protocol supports several flags you can use to restrict or expand your search. For example, if you know you want to search for a person named "Elliott" but you aren't sure whether he spells his name "Elliot", "Elliott", or perhaps even something as unlikely as "Elliotte", you would type:

```
% whois Person Partial Elliot
```

This tells the whois server that you want only matches for people (not domains, gateways, groups, or the like) whose names begin with the letters "Elliot". Unfortunately, you need to do a separate search if you want to find someone who spells his name "Eliot". The rules for modifying a search are summarized in Table 9-1. Each prefix should be placed before the search string on the command line.

Table 9-1. Whois prefixes

Prefix	Meaning
Domain	Find only domain records.
Gateway	Find only gateway records.
Group	Find only group records.
Host	Find only host records.
Network	Find only network records.
Organization	Find only organization records.

Prefix	Meaning
Person	Find only person records.
ASN	Find only autonomous system number records.
Handle or !	Search only for matching handles.
Mailbox or @	Search only for matching email addresses.
Name or :	Search only for matching names.
Expand or *	Search only for group records and show all individuals in that group.
Full or =	Show complete record for each match.
Partial or suffix	Match records that start with the given string.
Summary or $	Show just the summary, even if there's only one match.
SUBdisplay or %	Show the users of the specified host, the hosts on the specified network, etc.

These keywords are all useful and you could use them with the command-line client of Example 9-9, but they're way too much trouble to remember. In fact, most people don't even know that they exist. They just type "whois Harold" at the command-line and sort through the mess that comes back. A good whois client doesn't rely on users remembering arcane keywords; rather, it shows them the options. Supplying this requires a graphical user interface for end users and a better API for client programmers.

Example 9-10 is a more reusable `Whois` class. Two fields define the state of each `Whois` object: `host`, an `InetAddress` object, and `port`, an `int`. Together, these define the server that this particular `Whois` object connects to. Five constructors set these fields from various combinations of arguments. Furthermore, the host can be changed using the `setHost()` method.

The main functionality of the class is in one method, `lookUpNames( )`. The `lookUpNames ( )` method returns a `String` containing the whois response to a given query. The arguments specify the string to search for, what kind of record to search for, which database to search in, and whether an exact match is required. We could have used strings or `int` constants to specify the kind of record to search for and the database to search in, but since there are only a small number of valid values, `lookUpNames()` defines public inner classes with a fixed number of members instead. This solution provides much stricter compile-time type-checking and guarantees the `Whois` class won't have to handle an unexpected value.

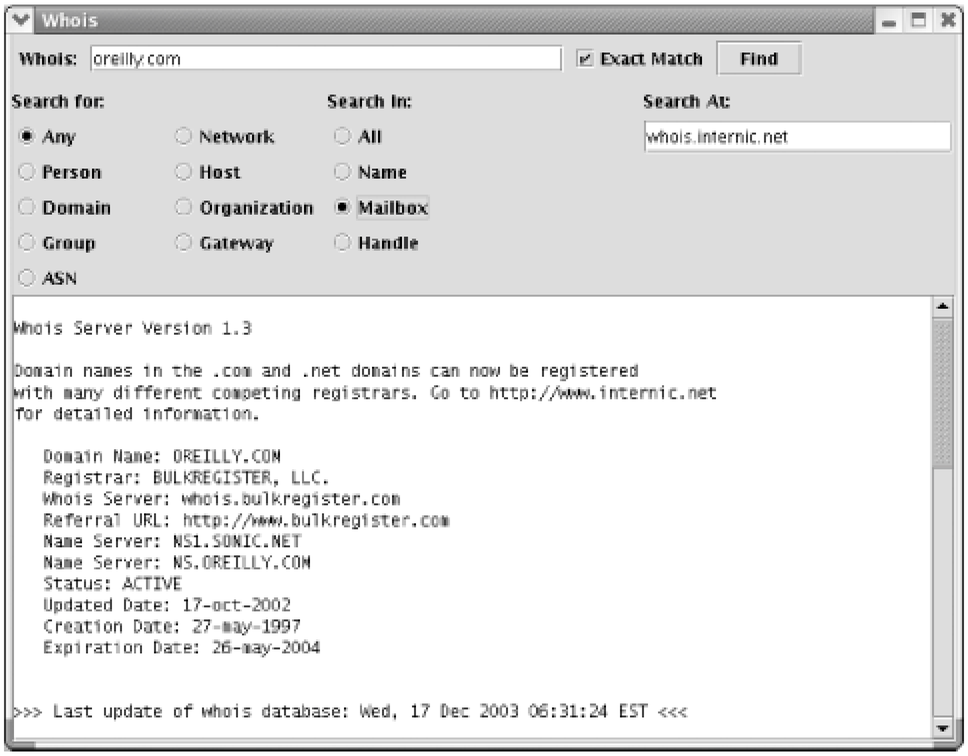
Example 9-10. The Whois class

```
import java.net.*;
import java.io.*;
import com.macfaq.io.SafeBufferedReader;


public class Whois {

  public final static int DEFAULT_PORT = 43;
  public final static String DEFAULT_HOST = "whois.internic.net";

  private int port = DEFAULT_PORT;
  private InetAddress host;

  public Whois(InetAddress host, int port) {
    this.host = host;
    this.port = port;
  }

  public Whois(InetAddress host) {
    this(host, DEFAULT_PORT);
  }

  public Whois(String hostname, int port)
   throws UnknownHostException {
    this(InetAddress.getByName(hostname), port);
  }

  public Whois(String hostname) throws UnknownHostException {
    this(InetAddress.getByName(hostname), DEFAULT_PORT);
  }

  public Whois( ) throws UnknownHostException {
    this(DEFAULT_HOST, DEFAULT_PORT);
  }

  // Items to search for
  public static class SearchFor {

    public static SearchFor ANY = new SearchFor( );
    public static SearchFor NETWORK = new SearchFor( );
    public static SearchFor PERSON = new SearchFor( );
    public static SearchFor HOST = new SearchFor( );
    public static SearchFor DOMAIN = new SearchFor( );
    public static SearchFor ORGANIZATION = new SearchFor( );
    public static SearchFor GROUP = new SearchFor( );
    public static SearchFor GATEWAY = new SearchFor( );
    public static SearchFor ASN = new SearchFor( );

    private SearchFor( ) {};

  }

  // Categories to search in
  public static class SearchIn {

    public static SearchIn ALL = new SearchIn( );
    public static SearchIn NAME = new SearchIn( );
    public static SearchIn MAILBOX = new SearchIn( );
    public static SearchIn HANDLE = new SearchIn( );

    private SearchIn( ) {};

  }

  public String lookUpNames(String target, SearchFor category,
   SearchIn group, boolean exactMatch) throws IOException {

    String suffix = "";
    if (!exactMatch) suffix = ".";

    String searchInLabel  = "";
    String searchForLabel = "";

    if (group == SearchIn.ALL) searchInLabel = "";
    else if (group == SearchIn.NAME) searchInLabel = "Name ";
    else if (group == SearchIn.MAILBOX) searchInLabel = "Mailbox ";
    else if (group == SearchIn.HANDLE) searchInLabel = "!";

    if (category == SearchFor.NETWORK) searchForLabel = "Network ";
    else if (category == SearchFor.PERSON) searchForLabel = "Person ";
    else if (category == SearchFor.HOST) searchForLabel = "Host ";
    else if (category == SearchFor.DOMAIN) searchForLabel = "Domain ";
    else if (category == SearchFor.ORGANIZATION) {
      searchForLabel = "Organization ";
    }
    else if (category == SearchFor.GROUP) searchForLabel = "Group ";
    else if (category == SearchFor.GATEWAY) {
      searchForLabel = "Gateway ";
    }
    else if (category == SearchFor.ASN) searchForLabel = "ASN ";

    String prefix = searchForLabel + searchInLabel;
    String query = prefix + target + suffix;

    Socket theSocket = new Socket(host, port);
    Writer out
     = new OutputStreamWriter(theSocket.getOutputStream( ), "ASCII");
    SafeBufferedReader in = new SafeBufferedReader(new
     InputStreamReader(theSocket.getInputStream( ), "ASCII"));
    out.write(query + "\r\n");
    out.flush( );
```

```
      StringBuffer response = new StringBuffer( );
      String theLine = null;
      while ((theLine = in.readLine( )) != null) {
        response.append(theLine);
        response.append("\r\n");
      }
      theSocket.close( );

      return response.toString( );

    }

    public InetAddress getHost( ) {
      return this.host;
    }

    public void setHost(String host)
      throws UnknownHostException {
      this.host = InetAddress.getByName(host);
    }

  }
```

Figure 9-1 shows one possible interface for a graphical whois client that depends on Example 9-11 for the actual network connections. This interface has a text field to enter the name to be searched for and a checkbox to determine whether the match should be exact or partial. A group of radio buttons lets users specify which group of records they want to search. Another group of radio buttons chooses the fields that should be searched. By default, this client searches all fields of all records for an exact match.

Figure 9-1. A graphical whois client



```
Whois Server Version 1.3

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: OREILLY.COM
   Registrar: BULKREGISTER, LLC.
   Whois Server: whois.bulkregister.com
   Referral URL: http://www.bulkregister.com
   Name Server: NS1.SONIC.NET
   Name Server: NS.OREILLY.COM
   Status: ACTIVE
   Updated Date: 17-oct-2002
   Creation Date: 27-may-1997
   Expiration Date: 26-may-2004


>>> Last update of whois database: Wed, 17 Dec 2003 06:31:24 EST <<<
```

When a user enters a string in the Whois: text field and presses the Enter or Find button, the program makes a connection to the whois server and retrieves records that match that string. These are placed in the text area in the bottom of the window. Initially, the server is set to whois.internic.net, but the user is free to change this setting. Example 9-11 is the program that produces this interface.

Example 9-11. A graphical Whois client interface

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class WhoisGUI extends JFrame {

  private JTextField searchString = new JTextField(30);
  private JTextArea names = new JTextArea(15, 80);
  private JButton findButton = new JButton("Find");;
  private ButtonGroup searchIn = new ButtonGroup( );
  private ButtonGroup searchFor = new ButtonGroup( );
  private JCheckBox exactMatch = new JCheckBox("Exact Match", true);
  private JTextField chosenServer = new JTextField( );
```

```
    private Whois server;

    public WhoisGUI(Whois whois) {

      super("Whois");
      this.server = whois;
      Container pane = this.getContentPane( );

      Font f = new Font("Monospaced", Font.PLAIN, 12);
      names.setFont(f);
      names.setEditable(false);

      JPanel centerPanel = new JPanel( );
      centerPanel.setLayout(new GridLayout(1, 1, 10, 10));
      JScrollPane jsp = new JScrollPane(names);
      centerPanel.add(jsp);
      pane.add("Center", centerPanel);

      // You don't want the buttons in the south and north
      // to fill the entire sections so add Panels there
      // and use FlowLayouts in the Panel
      JPanel northPanel = new JPanel( );
      JPanel northPanelTop = new JPanel( );
      northPanelTop.setLayout(new FlowLayout(FlowLayout.LEFT));
      northPanelTop.add(new JLabel("Whois: "));
      northPanelTop.add("North", searchString);
      northPanelTop.add(exactMatch);
      northPanelTop.add(findButton);
      northPanel.setLayout(new BorderLayout(2,1));
      northPanel.add("North", northPanelTop);
      JPanel northPanelBottom = new JPanel( );
      northPanelBottom.setLayout(new GridLayout(1,3,5,5));
      northPanelBottom.add(initRecordType( ));
      northPanelBottom.add(initSearchFields( ));
      northPanelBottom.add(initServerChoice( ));
      northPanel.add("Center", northPanelBottom);

      pane.add("North", northPanel);

      ActionListener al = new LookupNames( );
      findButton.addActionListener(al);
      searchString.addActionListener(al);

    }

    private JPanel initRecordType( ) {

      JPanel p = new JPanel( );
      p.setLayout(new GridLayout(6, 2, 5, 2));
      p.add(new JLabel("Search for:"));
      p.add(new JLabel(""));

      JRadioButton any = new JRadioButton("Any", true);
      any.setActionCommand("Any");
      searchFor.add(any);
      p.add(any);

      p.add(this.makeRadioButton("Network"));
      p.add(this.makeRadioButton("Person"));
      p.add(this.makeRadioButton("Host"));
      p.add(this.makeRadioButton("Domain"));
      p.add(this.makeRadioButton("Organization"));
      p.add(this.makeRadioButton("Group"));
      p.add(this.makeRadioButton("Gateway"));
      p.add(this.makeRadioButton("ASN"));

      return p;

    }

    private JRadioButton makeRadioButton(String label) {

      JRadioButton button = new JRadioButton(label, false);
      button.setActionCommand(label);
      searchFor.add(button);
      return button;

    }

    private JRadioButton makeSearchInRadioButton(String label) {

      JRadioButton button = new JRadioButton(label, false);
      button.setActionCommand(label);
      searchIn.add(button);
      return button;

    }

    private JPanel initSearchFields( ) {

      JPanel p = new JPanel( );
      p.setLayout(new GridLayout(6, 1, 5, 2));
      p.add(new JLabel("Search In: "));

      JRadioButton all = new JRadioButton("All", true);
      all.setActionCommand("All");
      searchIn.add(all);
      p.add(all);

      p.add(this.makeSearchInRadioButton("Name"));
      p.add(this.makeSearchInRadioButton("Mailbox"));
      p.add(this.makeSearchInRadioButton("Handle"));

      return p;

    }

    private JPanel initServerChoice( ) {

      final JPanel p = new JPanel( );
      p.setLayout(new GridLayout(6, 1, 5, 2));
      p.add(new JLabel("Search At: "));

      chosenServer.setText(server.getHost( ).getHostName( ));
      p.add(chosenServer);
      chosenServer.addActionListener( new ActionListener( ) {
        public void actionPerformed(ActionEvent evt) {
          try {
            InetAddress newHost
             = InetAddress.getByName(chosenServer.getText( ));
            Whois newServer = new Whois(newHost);
            server = newServer;
          }
          catch (Exception ex) {
             JOptionPane.showMessageDialog(p,
               ex.getMessage( ), "Alert", JOptionPane.ERROR_MESSAGE);
          }
        }
      } );

      return p;

    }

    class LookupNames implements ActionListener {

      public void actionPerformed(ActionEvent evt) {

        Whois.SearchIn group = Whois.SearchIn.ALL;
        Whois.SearchFor category = Whois.SearchFor.ANY;

        String searchForLabel = searchFor.getSelection( ).getActionCommand( );
        String searchInLabel = searchIn.getSelection( ).getActionCommand( );
        if (searchInLabel.equals("Name")) group = Whois.SearchIn.NAME;
        else if (searchInLabel.equals("Mailbox")) {
          group = Whois.SearchIn.MAILBOX;
        }
        else if (searchInLabel.equals("Handle")) {
          group = Whois.SearchIn.HANDLE;
        }

        if (searchForLabel.equals("Network")) {
          category = Whois.SearchFor.NETWORK;
        }
        else if (searchForLabel.equals("Person")) {
          category = Whois.SearchFor.PERSON;
        }
        else if (searchForLabel.equals("Host")) {
          category = Whois.SearchFor.HOST;
        }
        else if (searchForLabel.equals("Domain")) {
          category = Whois.SearchFor.DOMAIN;
        }
        else if (searchForLabel.equals("Organization")) {
          category = Whois.SearchFor.ORGANIZATION;
        }
        else if (searchForLabel.equals("Group")) {
          category = Whois.SearchFor.GROUP;
        }
        else if (searchForLabel.equals("Gateway")) {
          category = Whois.SearchFor.GATEWAY;
        }
        else if (searchForLabel.equals("ASN")) {
          category = Whois.SearchFor.ASN;
        }

        try {
          names.setText("");
          server.setHost(chosenServer.getText( ));
          String result = server.lookUpNames(searchString.getText( ),
           category, group, exactMatch.isSelected( ));
          names.setText(result);
        }
        catch (IOException ex) {
          JOptionPane.showMessageDialog(WhoisGUI.this,
            ex.getMessage( ), "Lookup Failed", JOptionPane.ERROR_MESSAGE);
        }
      }

    }

    public static void main(String[] args) {

      try {
        Whois server = new Whois( );
        WhoisGUI a = new WhoisGUI(server);
        a.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        a.pack( );
        EventQueue.invokeLater(new FrameShower(a));

      }
      catch (UnknownHostException ex) {
        JOptionPane.showMessageDialog(null, "Could not locate default host "
         + Whois.DEFAULT_HOST, "Error", JOptionPane.ERROR_MESSAGE);
      }

    }

    private static class FrameShower implements Runnable {

      private final Frame frame;

      FrameShower(Frame frame) {
        this.frame = frame;
      }

      public void run( ) {
       frame.setVisible(true);
      }

    }

  }
```

The `main( )` method is the usual block of code to start up a standalone application. It constructs a `Whois` object and, then uses that to construct a `WhoisGUI` object. Then the `WhoisGUI ( )` constructor sets up the graphical user interface. There's a lot of redundant code here, so it's broken out into the private methods `initSearchFields( )`, `initServerChoice( )`,`makeSearchInRadioButton( )`, and `makeSearchForRadioButton( )`. As usual with `LayoutManager`-based interfaces, the setup is fairly involved. Since you'd probably use a visual designer to build such an application, I won't describe it in detail here.

When the constructor returns, the `main( )` method attaches an anonymous inner class to the window that will close the application when the window is closed. (This isn't in the constructor because other programs that use this class may not want to exit the program when the window closes.) `main()` then packs and shows the window. To avoid an obscure race condition that can lead to deadlock this needs to be done in the event dispatch thread. Hence the `FrameShower` inner class that implements `Runnable` and the call to `EventQueue.invokeLater( )`. From that point on, all activity takes place in the AWT thread.

The first event this program must respond to is the user's typing a name in the Whois: text field and either pressing the Find button or hitting Enter. In this case, the `LookupNames` inner class passes the information in the text field and the various radio buttons and checkboxes to the `server.lookUpNames( )` method. This method returns a `String`, which is placed in the `names` text area.

The second event this program must respond to is the user typing a new host in the server text field. In this case, an anonymous inner class tries to construct a new `Whois` object and store it in the server field. If it fails (e.g., because the user mistyped the hostname), the old server is restored. An alert box informs the user of this event.

This is not a perfect client by any means. The most glaring omission is that it doesn't provide a way to save the data and quit the program. Less obvious until you run the program is that responsiveness suffers because the network connection is made inside the AWT thread. It would be better to place the connections to the server in their own thread and use callbacks to place the data in the GUI as the data is received. However, implementing callbacks would take us too far afield from the topic of network programming, so I leave them as exercises for the reader.