

## 2 Starting Network Programming in Java

### Learning Objectives

After reading this chapter, you should :

- know how to determine the host machine's IP address via a Java program;
- know how to use TCP sockets in both client programs and server programs;
- know how to use UDP sockets in both client programs and server programs;
- appreciate the convenience of Java's stream classes and the consistency of the interface afforded by them;
- appreciate the ease with which GUIs can be added to network programs;
- know how to check whether ports on a specified machine are running services;
- know how to use Java to render Web pages.

Having covered fundamental network protocols and techniques in a generic fashion in Chapter 1, it is now time to consider how those protocols may be used and the techniques implemented in Java. Core package *java.net* contains a number of very useful classes that allow programmers to carry out network programming very easily. Package *javax.net*, introduced in J2SE 1.4, contains factory classes for creating sockets in an implementation-independent fashion. Using classes from these packages (primarily from the former), the network programmer can communicate with any server on the Internet or implement his/her own Internet server.

### 2.1 The *InetAddress* Class

One of the classes within package *java.net* is called *InetAddress*, which handles Internet addresses both as host names and as IP addresses. Static method *getByName* of this class uses DNS (Domain Name System) to return the Internet address of a specified host name as an *InetAddress* object. In order to display the IP address from this object, we can simply use method *println* (which will cause the object's *toString* method to be executed). Since method *getByName* throws the checked exception *UnknownHostException* if the host name is not recognised, we must either throw this exception or (preferably) handle it with a `catch` clause. The following example illustrates this.

### Example

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

public class IPFinder
{
    public static void main(String[] args)
    {
        String host;
        Scanner input = new Scanner(System.in);

        System.out.print("\n\nEnter host name: ");
        host = input.next();
        try
        {
            InetAddress address =
                InetAddress.getByName(host);
            System.out.println("IP address: "
                               + address.toString());
        }
        catch (UnknownHostException uhEx)
        {
            System.out.println("Could not find " + host);
        }
    }
}
```

The output from a test run of this program is shown in Figure 2.1.



Figure 2.1 Using method *getByName* to retrieve IP address of a specified host.

It is sometimes useful for Java programs to be able to retrieve the IP address of the current machine. The example below shows how to do this.

### Example

```
import java.net.*;

public class MyLocalIPAddress
{
    public static void main(String[] args)
    {
        try
        {
            InetAddress address =
                InetAddress.getLocalHost();
            System.out.println(address);
        }
        catch (UnknownHostException uhEx)
        {
            System.out.println(
                "Could not find local address!");
        }
    }
}
```

Output from this program when run on the author's office machine is shown in Figure 2.2.

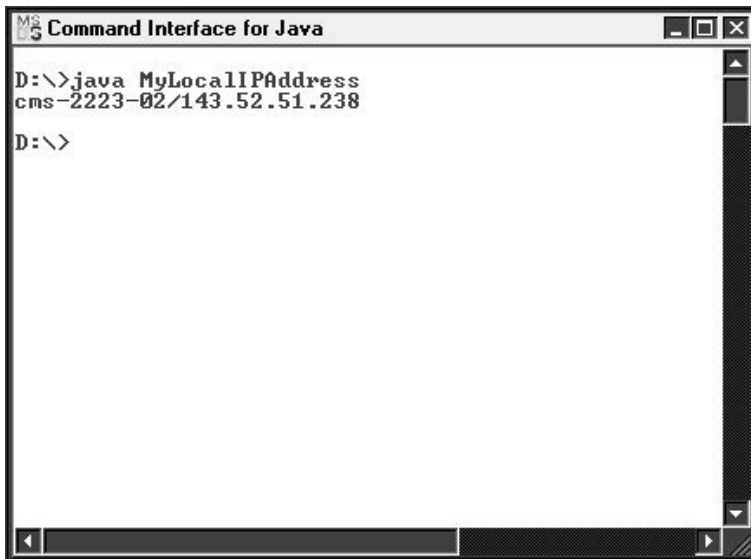


Figure 2.2 Retrieving the current machine's IP address.

## 2.2 Using Sockets

As described in Chapter 1, different processes (programs) can communicate with each other across networks by means of sockets. Java implements both **TCP/IP** sockets and **datagram** sockets (UDP sockets). Very often, the two communicating processes will have a *client/server* relationship. The steps required to create client/server programs via each of these methods are very similar and are outlined in the following two sub-sections.

### 2.2.1 TCP Sockets

A communication link created via TCP/IP sockets is a **connection-orientated** link. This means that the connection between server and client remains open throughout the duration of the dialogue between the two and is only broken (under normal circumstances) when one end of the dialogue formally terminates the exchanges (via an agreed protocol). Since there are two separate types of process involved (client and server), we shall examine them separately, taking the server first. Setting up a server process requires five steps...

#### 1. *Create a ServerSocket object.*

The *ServerSocket* constructor requires a port number (1024-65535, for non-reserved ones) as an argument. For example:

```
ServerSocket servSock = new ServerSocket(1234);
```

In this example, the server will await ('listen for') a connection from a client on port 1234.

#### 2. *Put the server into a waiting state.*

The server waits indefinitely ('blocks') for a client to connect. It does this by calling method *accept* of class *ServerSocket*, which returns a *Socket* object when a connection is made. For example:

```
Socket link = servSock.accept();
```

#### 3. *Set up input and output streams.*

Methods *getInputStream* and *getOutputStream* of class *Socket* are used to get references to streams associated with the socket returned in step 2. These streams will be used for communication with the client that has just made connection. For a non-GUI application, we can wrap a *Scanner* object around the *InputStream* object returned by method *getInputStream*, in order to obtain string-orientated input (just as we would do with input from the standard input stream, *System.in*). For example:

```
Scanner input = new Scanner(link.getInputStream());
```

Similarly, we can wrap a *PrintWriter* object around the *OutputStream* object returned by method *getOutputStream*. Supplying the *PrintWriter* constructor with a second argument of `true` will cause the output buffer to be flushed for every call of *println* (which is usually desirable). For example:

```
PrintWriter output =  
    new PrintWriter(link.getOutputStream(), true);
```

#### 4. *Send and receive data.*

Having set up our *Scanner* and *PrintWriter* objects, sending and receiving data is very straightforward. We simply use method *nextLine* for receiving data and method *println* for sending data, just as we might do for console I/O. For example:

```
output.println("Awaiting data...");  
String input = input.nextLine();
```

#### 5. *Close the connection (after completion of the dialogue).*

This is achieved via method *close* of class *Socket*. For example:

```
link.close();
```

The following example program is used to illustrate the use of these steps.

#### Example

In this simple example, the server will accept messages from the client and will keep count of those messages, echoing back each (numbered) message. The main protocol for this service is that client and server must alternate between sending and receiving (with the client initiating the process with its opening message, of course). The only details that remain to be determined are the means of indicating when the dialogue is to cease and what final data (if any) should be sent by the server. For this simple example, the string `***CLOSE***` will be sent by the client when it wishes to close down the connection. When the server receives this message, it will confirm the number of preceding messages received and then close its connection to this client. The client, of course, must wait for the final message from the server before closing the connection at its own end.

Since an *IOException* may be generated by any of the socket operations, one or more `try` blocks must be used. Rather than have one large `try` block (with no variation in the error message produced and, consequently, no indication of precisely what operation caused the problem), it is probably good practice to have the opening of the port and the dialogue with the client in separate `try` blocks. It is also good practice to place the closing of the socket in a `finally` clause, so that, whether an exception occurs or not, the socket will be closed (unless, of course, the exception is generated when actually closing the socket, but there is nothing we can do about that). Since the `finally` clause will need to know about the *Socket*

object, we shall have to declare this object within a scope that covers both the `try` block handling the dialogue and the `finally` block. Thus, step 2 shown above will be broken up into separate declaration and assignment. In our example program, this will also mean that the *Socket* object will have to be explicitly initialised to `null` (as it will not be a global variable).

Since a server offering a public service would keep running indefinitely, the call to method *handleClient* in our example has been placed inside an ‘infinite’ loop, thus:

```
do
{
    handleClient();
}while (true);
```

In the code that follows (and in later examples), port 1234 has been chosen for the service, but it could just as well have been any integer in the range 1024-65535. Note that the lines of code corresponding to each of the above steps have been clearly marked with emboldened comments.

```
//Server that echoes back client's messages.
//At end of dialogue, sends message indicating number of
//messages received. Uses TCP.
```

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

public class TCPEchoServer
{
    private static ServerSocket servSock;
    private static final int PORT = 1234;

    public static void main(String[] args)
    {
        System.out.println("Opening port...\n");
        try
        {
            servSock = new ServerSocket(PORT);    //Step 1.
        }
        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to attach to port!");
            System.exit(1);
        }
        do
        {
            handleClient();
```

```

    }while (true);
}

private static void handleClient()
{
    Socket link = null;                                //Step 2.

    try
    {
        link = servSock.accept();                      //Step 2.

        Scanner input =
            new Scanner(link.getInputStream()); //Step 3.
        PrintWriter output =
            new PrintWriter(
                link.getOutputStream(),true); //Step 3.

        int numMessages = 0;
        String message = input.nextLine();             //Step 4.
        while (!message.equals("***CLOSE***"))
        {
            System.out.println("Message received.");
            numMessages++;
            output.println("Message " + numMessages
                + ": " + message); //Step 4.
            message = input.nextLine();
        }
        output.println(numMessages
            + " messages received."); //Step 4.
    }
    catch(IOException ioEx)
    {
        ioEx.printStackTrace();
    }

    finally
    {
        try
        {
            System.out.println(
                "\n* Closing connection... *");
            link.close();                        //Step 5.
        }
        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to disconnect!");
            System.exit(1);
        }
    }
}

```

```

    }
}
}

```

Setting up the corresponding client involves four steps...

### 1. *Establish a connection to the server.*

We create a *Socket* object, supplying its constructor with the following two arguments:

- the server's IP address (of type *InetAddress*);
- the appropriate port number for the service.

(The port number for server and client programs must be the same, of course!)

For simplicity's sake, we shall place client and server on the same host, which will allow us to retrieve the IP address by calling static method *getLocalHost* of class *InetAddress*. For example:

```

Socket link =
    new Socket(InetAddress.getLocalHost(), 1234);

```

### 2. *Set up input and output streams.*

These are set up in exactly the same way as the server streams were set up (by calling methods *getInputStream* and *getOutputStream* of the *Socket* object that was created in step 2).

### 3. *Send and receive data.*

The *Scanner* object at the client end will receive messages sent by the *PrintWriter* object at the server end, while the *PrintWriter* object at the client end will send messages that are received by the *Scanner* object at the server end (using methods *nextLine* and *println* respectively).

### 4. *Close the connection.*

This is exactly the same as for the server process (using method *close* of class *Socket*).

The code below shows the client program for our example. In addition to an input stream to accept messages from the server, our client program will need to set up an input stream (as another *Scanner* object) to accept user messages from the keyboard. As for the server, the lines of code corresponding to each of the above steps have been clearly marked with emboldened comments.



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

public class TCPEchoClient
{
    private static InetAddress host;
    private static final int PORT = 1234;

    public static void main(String[] args)
    {
        try
        {
            host = InetAddress.getLocalHost();
        }
        catch(UnknownHostException uhEx)
        {
            System.out.println("Host ID not found!");
            System.exit(1);
        }
        accessServer();
    }

    private static void accessServer()
    {
        Socket link = null;                                //Step 1.

        try
        {
            link = new Socket(host,PORT);                  //Step 1.

            Scanner input =
                new Scanner(link.getInputStream()); //Step 2.

            PrintWriter output =
                new PrintWriter(
                    link.getOutputStream(),true); //Step 2.

            //Set up stream for keyboard entry...
            Scanner userEntry = new Scanner(System.in);

            String message, response;
            do
            {
                System.out.print("Enter message: ");
                message = userEntry.nextLine();
                output.println(message);                //Step 3.
                response = input.nextLine();              //Step 3.
            }
        }
    }
}
```

```

        System.out.println("\nSERVER> "+response);
    }while (!message.equals("***CLOSE***"));
}
catch(IOException ioEx)
{
    ioEx.printStackTrace();
}

finally
{
    try
    {
        System.out.println(
            "\n* Closing connection... *");
        link.close(); //Step 4.
    }
    catch(IOException ioEx)
    {
        System.out.println(
            "Unable to disconnect!");
        System.exit(1);
    }
}
}
}

```

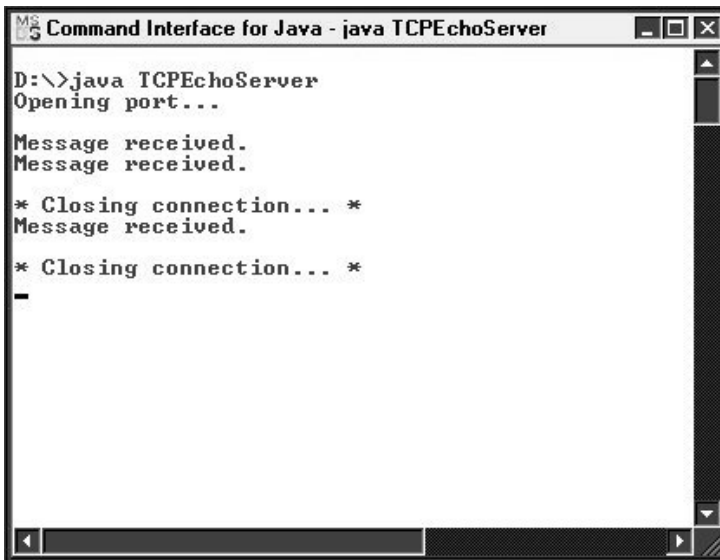
For the preceding client-server application to work, TCP/IP must be installed and working. How are you to know whether this is the case for your machine? Well, if there is a working Internet connection on your machine, then TCP/IP **is** running. In order to start the application, first open two command windows and then start the server running in one window and the client in the other. (Make sure that the server is running first, in order to avoid having the client program crash!) The example screenshots in Figures 2.3 and 2.4 show the dialogues between the server and two consecutive clients for this application. Note that, in order to stop the TCPEchoServer program, Ctrl-C has to be entered from the keyboard.

### 2.2.2 Datagram (UDP) Sockets

Unlike TCP/IP sockets, datagram sockets are **connectionless**. That is to say, the connection between client and server is **not** maintained throughout the duration of the dialogue. Instead, each datagram packet is sent as an isolated transmission whenever necessary. As noted in Chapter 1, datagram (UDP) sockets provide a faster means of transmitting data than TCP/IP sockets, but they are unreliable.

Since the connection is not maintained between transmissions, the server does not create an individual *Socket* object for each client, as it did in our TCP/IP example. A further difference from TCP/IP sockets is that, instead of a *ServerSocket* object, the server creates a *DatagramSocket* object, as does each client when it wants to send datagram(s) to the server. The final and most significant difference is that

*DatagramPacket* objects are created and sent at both ends, rather than simple strings.



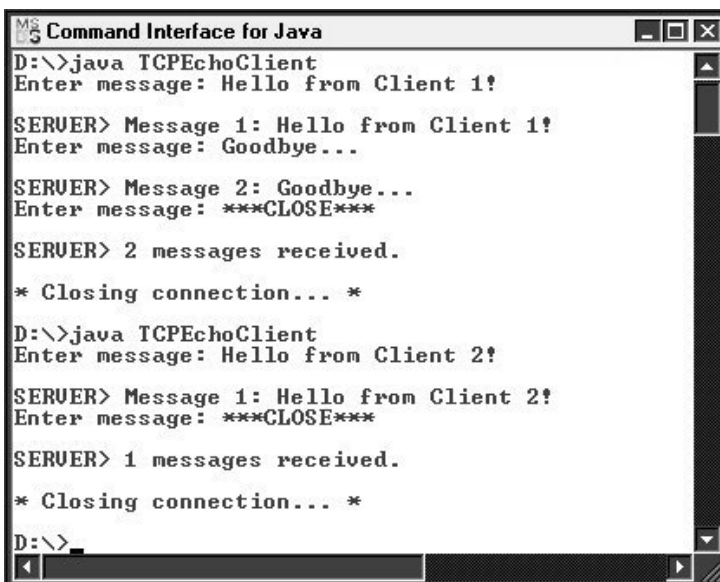
```
MS-DOS Command Interface for Java - java TCPEchoServer
D:\>java TCPEchoServer
Opening port...

Message received.
Message received.

* Closing connection... *
Message received.

* Closing connection... *
-
```

Figure 2.3 Example output from the TCPEchoServer program.



```
MS-DOS Command Interface for Java
D:\>java TCPEchoClient
Enter message: Hello from Client 1!

SERVER> Message 1: Hello from Client 1!
Enter message: Goodbye...

SERVER> Message 2: Goodbye...
Enter message: ***CLOSE***

SERVER> 2 messages received.

* Closing connection... *

D:\>java TCPEchoClient
Enter message: Hello from Client 2!

SERVER> Message 1: Hello from Client 2!
Enter message: ***CLOSE***

SERVER> 1 messages received.

* Closing connection... *

D:\>
```

Figure 2.4 Example output from the TCPEchoClient program.

Following the style of coverage for TCP client/server applications, the detailed steps required for client and server will be described separately, with the server process being covered first. This process involves the following nine steps, though only the first eight steps will be executed under normal circumstances...

### **1. *Create a DatagramSocket object.***

Just as for the creation of a *ServerSocket* object, this means supplying the object's constructor with the port number. For example:

```
DatagramSocket datagramSocket =  
                                new DatagramSocket(1234);
```

### **2. *Create a buffer for incoming datagrams.***

This is achieved by creating an array of bytes. For example:

```
byte[] buffer = new byte[256];
```

### **3. *Create a DatagramPacket object for the incoming datagrams.***

The constructor for this object requires two arguments:

- the previously-created byte array;
- the size of this array.

For example:

```
DatagramPacket inPacket =  
    new DatagramPacket(buffer, buffer.length);
```

### **4. *Accept an incoming datagram.***

This is effected via the *receive* method of our *DatagramSocket* object, using our *DatagramPacket* object as the receptacle. For example:

```
datagramSocket.receive(inPacket);
```

### **5. *Accept the sender's address and port from the packet.***

Methods *getAddress* and *getPort* of our *DatagramPacket* object are used for this. For example:

```
InetAddress clientAddress = inPacket.getAddress();  
int clientPort = inPacket.getPort();
```

## 6. *Retrieve the data from the buffer.*

For convenience of handling, the data will be retrieved as a string, using an overloaded form of the *String* constructor that takes three arguments:

- a byte array;
- the start position within the array (= 0 here);
- the number of bytes (= full size of buffer here).

For example:

```
String message = new String(inPacket.getData(),
                           0, inPacket.getLength());
```

## 7. *Create the response datagram.*

Create a *DatagramPacket* object, using an overloaded form of the constructor that takes four arguments:

- the byte array containing the response message;
- the size of the response;
- the client's address;
- the client's port number.

The first of these arguments is returned by the *getBytes* method of the *String* class (acting on the desired *String* response). For example:

```
DatagramPacket outPacket =
    new DatagramPacket(response.getBytes(),
                      response.length(), clientAddress, clientPort);
(Here, response is a String variable holding the return message.)
```

## 8. *Send the response datagram.*

This is achieved by calling method *send* of our *DatagramSocket* object, supplying our outgoing *DatagramPacket* object as an argument. For example:

```
datagramSocket.send(outPacket);
```

Steps 4-8 may be executed indefinitely (within a loop).

Under normal circumstances, the server would probably not be closed down at all. However, if an exception occurs, then the associated *DatagramSocket* should be closed, as shown in step 9 below.

## 9. *Close the DatagramSocket.*

This is effected simply by calling method *close* of our *DatagramSocket* object. For example:

```
datagramSocket.close();
```

To illustrate the above procedure and to allow easy comparison with the equivalent TCP/IP code, the example from Section 2.2.1 will be employed again. As before, the lines of code corresponding to each of the above steps are indicated via emboldened comments. Note that the *numMessages* part of the message that is returned by the server is somewhat artificial, since, in a real-world application, many clients could be making connection and the overall message numbers would not mean a great deal to individual clients. However, the cumulative message-numbering will serve to emphasise that there are no separate sockets for individual clients.

There are two other differences from the equivalent TCP/IP code that are worth noting, both concerning the possible exceptions that may be generated:

- the *IOException* in *main* is replaced with a *SocketException*;
- there is no checked exception generated by the *close* method in the *finally* clause, so there is no *try* block.

Now for the code...

```
//Server that echoes back client's messages.
//At end of dialogue, sends message indicating number of
//messages received. Uses datagrams.

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

public class UDPEchoServer
{
    private static final int PORT = 1234;
    private static DatagramSocket datagramSocket;
    private static DatagramPacket inPacket, outPacket;
    private static byte[] buffer;

    public static void main(String[] args)
    {
        System.out.println("Opening port...\n");
        try
        {
            datagramSocket =
                new DatagramSocket(PORT);           //Step 1.
        }
    }
}
```

```

        catch(SocketException sockEx)
        {
            System.out.println(
                "Unable to attach to port!");
            System.exit(1);
        }
        handleClient();
    }

    private static void handleClient()
    {
        try
        {
            String messageIn,messageOut;
            int numMessages = 0;

            do
            {
                buffer = new byte[256];                //Step 2.
                inPacket =
                    new DatagramPacket(
                        buffer, buffer.length);    //Step 3.
                datagramSocket.receive(inPacket); //Step 4.

                InetAddress clientAddress =
                    inPacket.getAddress(); //Step 5.
                int clientPort =
                    inPacket.getPort();    //Step 5.

                messageIn =
                    new String(inPacket.getData(),
                        0,inPacket.getLength()); //Step 6.

                System.out.println("Message received.");
                numMessages++;
                messageOut = "Message " + numMessages
                    + ": " + messageIn;

                outPacket =
                    new DatagramPacket(messageOut.getBytes(),
                        messageOut.length(),clientAddress,
                        clientPort);                //Step 7.
                datagramSocket.send(outPacket);    //Step 8.
            }while (true);
        }
        catch(IOException ioEx)
        {
            ioEx.printStackTrace();
        }
    }

```

```

        finally //If exception thrown, close connection.
        {
            System.out.println(
                "\n* Closing connection... *");
            datagramSocket.close(); //Step 9.
        }
    }
}

```

Setting up the corresponding client requires the eight steps listed below.

### **1. *Create a DatagramSocket object.***

This is similar to the creation of a `DatagramSocket` object in the server program, but with the important difference that the constructor here requires no argument, since a default port (at the client end) will be used. For example:

```
DatagramSocket datagramSocket = new DatagramSocket();
```

### **2. *Create the outgoing datagram.***

This step is exactly as for step 7 of the server program. For example:

```
DatagramPacket outPacket =
    new DatagramPacket(message.getBytes(),
        message.length(), host, PORT);
```

### **8. *Send the datagram message.***

Just as for the server, this is achieved by calling method *send* of the *DatagramSocket* object, supplying our outgoing *DatagramPacket* object as an argument. For example:

```
datagramSocket.send(outPacket);
```

Steps 4-6 below are exactly the same as steps 2-4 of the server procedure.

### **4. *Create a buffer for incoming datagrams.***

For example:

```
byte[] buffer = new byte[256];
```

### **5. *Create a DatagramPacket object for the incoming datagrams.***

For example:

```
DatagramPacket inPacket =
    new DatagramPacket(buffer, buffer.length);
```



**6. *Accept an incoming datagram.***

For example:

```
datagramSocket.receive(inPacket);
```

**7. *Retrieve the data from the buffer.***

This is the same as step 6 in the server program. For example:

```
String response =
    new String(inPacket.getData(), 0,
               inPacket.getLength());
```

Steps 2-7 may then be repeated as many times as required.

**8. *Close the DatagramSocket.***

This is the same as step 9 in the server program. For example:

```
datagramSocket.close();
```

As was the case in the server code, there is no checked exception generated by the above *close* method in the *finally* clause of the client program, so there will be no *try* block. In addition, since there is no inter-message connection maintained between client and server, there is no protocol required for closing down the dialogue. This means that we do not wish to send the final '\*\*\*CLOSE\*\*\*' string (though we shall continue to accept this from the user, since we need to know when to stop sending messages at the client end). The line of code (singular, this time) corresponding to each of the above steps will be indicated via an emboldened comment.

Now for the code itself...

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

public class UDPEchoClient
{
    private static InetAddress host;
    private static final int PORT = 1234;
    private static DatagramSocket datagramSocket;
    private static DatagramPacket inPacket, outPacket;
    private static byte[] buffer;
```

```

public static void main(String[] args)
{
    try
    {
        host = InetAddress.getLocalHost();
    }
    catch(UnknownHostException uhEx)
    {
        System.out.println("Host ID not found!");
        System.exit(1);
    }
    accessServer();
}

private static void accessServer()
{
    try
    {
        //Step 1...
        datagramSocket = new DatagramSocket();

        //Set up stream for keyboard entry...
        Scanner userEntry = new Scanner(System.in);

        String message="", response="";
        do
        {
            System.out.print("Enter message: ");
            message = userEntry.nextLine();
            if (!message.equals("***CLOSE***"))
            {
                outPacket = new DatagramPacket(
                    message.getBytes(),
                    message.length(),
                    host,PORT); //Step 2.

                //Step 3...
                datagramSocket.send(outPacket);
                buffer = new byte[256]; //Step 4.
                inPacket =
                    new DatagramPacket(
                        buffer, buffer.length); //Step 5.
                //Step 6...
                datagramSocket.receive(inPacket);
                response =
                    new String(inPacket.getData(),
                        0, inPacket.getLength()); //Step 7.
                System.out.println(
                    "\nSERVER> "+response);
            }
        }
    }
}

```

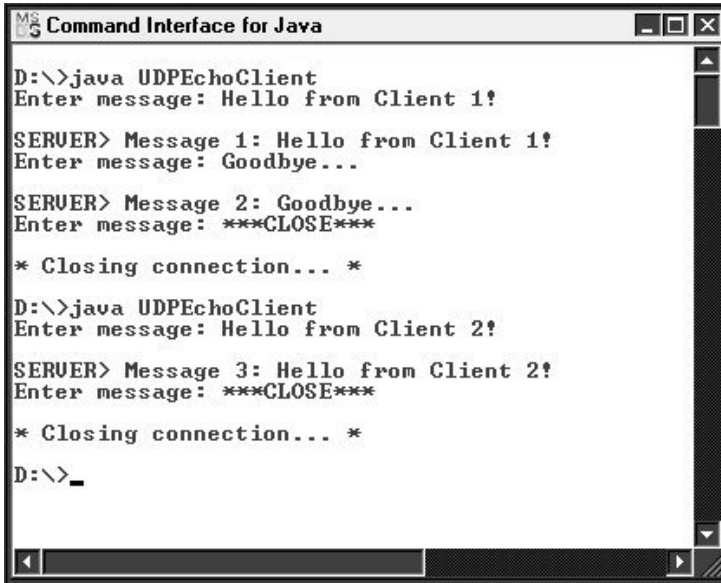
```
        }
    }while (!message.equals("***CLOSE***"));
}
catch(IOException ioEx)
{
    ioEx.printStackTrace();
}

finally
{
    System.out.println(
        "\n* Closing connection... *");
    datagramSocket.close();           //Step 8.
}
}
```

For the preceding application to work, UDP must be installed and working on the host machine. As for TCP/IP, if there is a working Internet connection on the machine, then UDP **is** running. Once again, in order to start the application, first open two command windows and then start the server running in one window and the client in the other. (Start the server *before* the client!) As before, the example screenshots in Figures 2.5 and 2.6 show the dialogues between the server and two clients. Observe the differences in output between this example and the corresponding TCP/IP example. (Note that the change at the client end is simply the rather subtle one of cumulative message-numbering.)



Figure 2.5 Example output from the UDPEchoServer program



```
D:\>java UDPEchoClient
Enter message: Hello from Client 1!

SERVER> Message 1: Hello from Client 1!
Enter message: Goodbye...

SERVER> Message 2: Goodbye...
Enter message: ***CLOSE***

* Closing connection... *

D:\>java UDPEchoClient
Enter message: Hello from Client 2!

SERVER> Message 3: Hello from Client 2!
Enter message: ***CLOSE***

* Closing connection... *

D:\>_
```

Figure 2.6 Example output from the UDPEchoClient program (with two clients connecting).

## 2.3 Network Programming with GUIs

Now that the basics of socket programming in Java have been covered, we can add some sophistication to our programs by providing them with graphical user interfaces (GUIs), which users have come to expect most software nowadays to provide. In order to concentrate upon the interface to each program, rather than upon the details of that program's processing, the examples used will simply provide access to some of the standard services, available via 'well known' ports. Some of these standard services were listed in Figure 1.1.

### Example

The following program uses the *Daytime* protocol to obtain the date and time from port 13 of user-specified host(s). It provides a text field for input of the host name by the user and a text area for output of the host's response. There are also two buttons, one that the user presses after entry of the host name and the other that closes down the program. The text area is 'wrapped' in a *JScrollPane*, to cater for long lines of output, while the buttons are laid out on a separate panel. The application frame itself will handle the processing of button presses, and so implements the *ActionListener* interface. The window-closing code (encapsulated in an anonymous *WindowAdapter* object) ensures that any socket that has been opened is closed before exit from the program.

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

public class GetRemoteTime extends JFrame
    implements ActionListener
{
    private JTextField hostInput;
    private JTextArea display;
    private JButton timeButton;
    private JButton exitButton;
    private JPanel buttonPanel;
    private static Socket socket = null;

    public static void main(String[] args)
    {
        GetRemoteTime frame = new GetRemoteTime();
        frame.setSize(400,300);
        frame.setVisible(true);

        frame.addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing(
                    WindowEvent event)
                {
                    //Check whether a socket is open...
                    if (socket != null)
                    {
                        try
                        {
                            socket.close();
                        }
                        catch (IOException ioEx)
                        {
                            System.out.println(
                                "\nUnable to close link!\n");
                            System.exit(1);
                        }
                    }
                    System.exit(0);
                }
            }
        );
    }
}
```

```
public GetRemoteTime()
{
    hostInput = new JTextField(20);
    add(hostInput, BorderLayout.NORTH);

    display = new JTextArea(10,15);

    //Following two lines ensure that word-wrapping
    //occurs within the JTextArea...
    display.setWrapStyleWord(true);
    display.setLineWrap(true);

    add(new JScrollPane(display),
        BorderLayout.CENTER);

    buttonPanel = new JPanel();

    timeButton = new JButton("Get date and time ");
    timeButton.addActionListener(this);
    buttonPanel.add(timeButton);

    exitButton = new JButton("Exit");
    exitButton.addActionListener(this);
    buttonPanel.add(exitButton);

    add(buttonPanel, BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == exitButton)
        System.exit(0);

    String theTime;

    //Accept host name from the user...
    String host = hostInput.getText();
    final int DAYTIME_PORT = 13;

    try
    {
        //Create a Socket object to connect to the
        //specified host on the relevant port...
        socket = new Socket(host, DAYTIME_PORT);

        //Create an input stream for the above Socket
        //and add string-reading functionality...
```

```

        Scanner input =
            new Scanner(socket.getInputStream());

        //Accept the host's response via the
        //above stream...
        theTime = input.nextLine();

        //Add the host's response to the text in
        //the JTextArea...
        display.append("The date/time at " + host
            + " is " + theTime + "\n");
        hostInput.setText("");
    }
    catch (UnknownHostException uhEx)
    {
        display.append("No such host!\n");
        hostInput.setText("");
    }
    catch (IOException ioEx)
    {
        display.append(ioEx.toString() + "\n");
    }

    finally
    {
        try
        {
            if (socket!=null)
                socket.close(); //Close link to host.
        }
        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to disconnect!");
            System.exit(1);
        }
    }
}

```

If we run this program and enter *ivy.shu.ac.uk* as our host name in the client's GUI, the result will look something like that shown in Figure 2.7.

Unfortunately, it is rather difficult nowadays to find a host that is running the *Daytime* protocol. Even if one does find such a host, it may be that the user's own firewall blocks the output from the remote server. If this is the case, then the user will be unaware of this until the connection times out – which may take some time! The user is advised to terminate the program (with Ctrl-C) if the waiting time

appears to be excessive. One possible way round this problem is to write one's own 'daytime server'...

To illustrate just how easy it is to provide a server that implements the *Daytime* protocol, example code for such a server is shown below. The program makes use of class *Date* from package *java.util* to create a *Date* object that will automatically hold the current day, date and time on the server's host machine. To output the date held in the *Date* object, we can simply use *println* on the object and its *toString* method will be executed implicitly (though we could specify *toString* explicitly, if we wished).

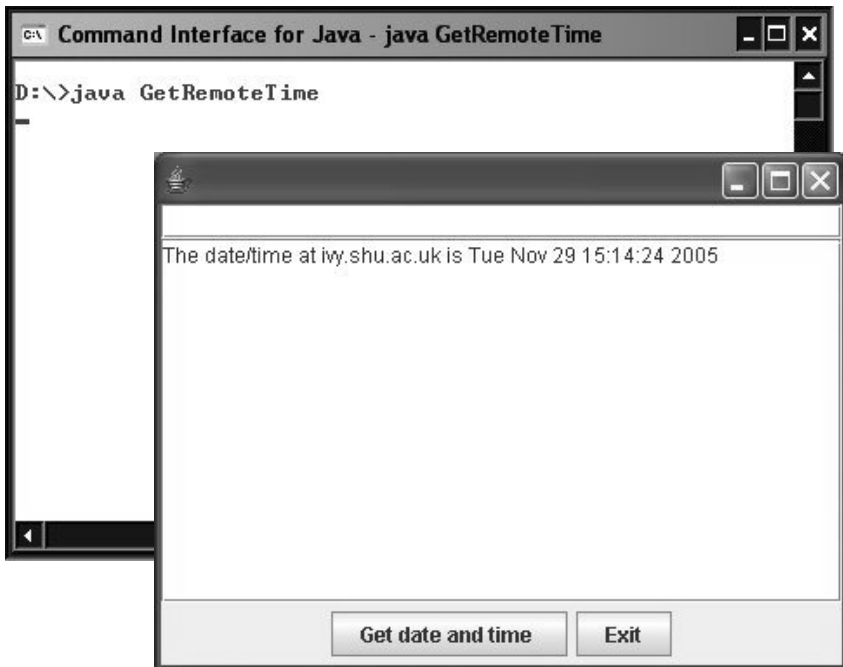


Figure 2.7 Example output from the GetRemoteTime program.

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

public class DaytimeServer
{
    public static void main(String[] args)
    {
        ServerSocket server;
        final int DAYTIME_PORT = 13;
        Socket socket;
```



```

try
{
    server = new ServerSocket(DAYTIME_PORT);

    do
    {
        socket = server.accept();
        PrintWriter output =
            new PrintWriter(
                socket.getOutputStream(), true);
        Date date = new Date();
        output.println(date);
        //Method toString executed in line above.

        socket.close();
    }while (true);
}
catch (IOException ioEx)
{
    System.out.println(ioEx);
}
}

```

The server simply sends the date and time as a string and then closes the connection. If we run the client and server in separate command windows and enter *localhost* as our host name in the client's GUI, the result should look similar to that shown in Figure 2.7. Unfortunately, there is still a potential problem on some systems: since a low-numbered port (i.e., below 1024) is being used, the user may not have sufficient system rights to make use of the port. The solution in such circumstances is simple: change the port number (in both server and client) to a value above 1024. (E.g., change the value of *DAYTIME\_PORT* from 13 to 1300.)

Now for an example that checks a range of ports on a specified host and reports on those ports that are providing a service. This works by the program trying to create a socket on each port number in turn. If a socket is created successfully, then there is an open port; otherwise, an *IOException* is thrown (and ignored by the program, which simply provides an empty *catch* clause). The program creates a text field for acceptance of the required URL(s) and sets this to an initial default value. It also provides a text area for the program's output and buttons for checking the ports and for exiting the program.

```

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

```

```
public class PortScanner extends JFrame
                                implements ActionListener
{
    private JLabel prompt;
    private JTextField hostInput;
    private JTextArea report;
    private JButton seekButton, exitButton;
    private JPanel hostPanel, buttonPanel;
    private static Socket socket = null;

    public static void main(String[] args)
    {
        PortScanner frame = new PortScanner();
        frame.setSize(400,300);
        frame.setVisible(true);

        frame.addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing(
                    WindowEvent event)
                {
                    //Check whether a socket is open...
                    if (socket != null)
                    {
                        try
                        {
                            socket.close();
                        }
                        catch (IOException ioEx)
                        {
                            System.out.println(
                                "\nUnable to close link!\n");
                            System.exit(1);
                        }
                    }
                    System.exit(0);
                }
            }
        );
    }

    public PortScanner()
    {
        hostPanel = new JPanel();

        prompt = new JLabel("Host name: ");
```

```
hostInput = new JTextField("ivy.shu.ac.uk", 25);
hostPanel.add(prompt);
hostPanel.add(hostInput);
add(hostPanel, BorderLayout.NORTH);

report = new JTextArea(10, 25);
add(report, BorderLayout.CENTER);

buttonPanel = new JPanel();

seekButton = new JButton("Seek server ports ");
seekButton.addActionListener(this);
buttonPanel.add(seekButton);

exitButton = new JButton("Exit");
exitButton.addActionListener(this);
buttonPanel.add(exitButton);

add(buttonPanel, BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == exitButton)
        System.exit(0);
    //Must have been the 'seek' button that was
    //pressed, so clear the output area of any
    //previous output...
    report.setText("");

    //Retrieve the URL from the input text field...
    String host = hostInput.getText();

    try
    {
        //Convert the URL string into an InetAddress
        //object...
        InetAddress theAddress =
            InetAddress.getByName(host);
        report.append("IP address: "
            + theAddress + "\n");

        for (int i = 0; i < 25; i++)
        {
            try
            {
                //Attempt to establish a socket on
                //port i...
```

```

        socket = new Socket(host, i);

        //If no IOException thrown, there must
        //be a service running on the port...
        report.append(
            "There is a server on port "
                + i + ".\n");

        socket.close();
    }
    catch (IOException ioEx)
    {} // No server on this port
}
catch (UnknownHostException uhEx)
{
    report.setText("Unknown host!");
}
}
}

```

When the above program was run for the default server (which is on the author's local network), the output from the GUI was as shown in Figure 2.8. Unfortunately, remote users' firewalls may block output from most of the ports for this default server (or any other remote server), causing the program to wait for each of these port accesses to time out. This is likely to take a **very** long time indeed! The reader is strongly advised to use a local server for the testing of this program (and to get clearance from your system administrator for port scanning, to be on the safe side). Even when running the program with a suitable local server, **be patient** when waiting for output, since this may take a minute or so, depending upon your system.

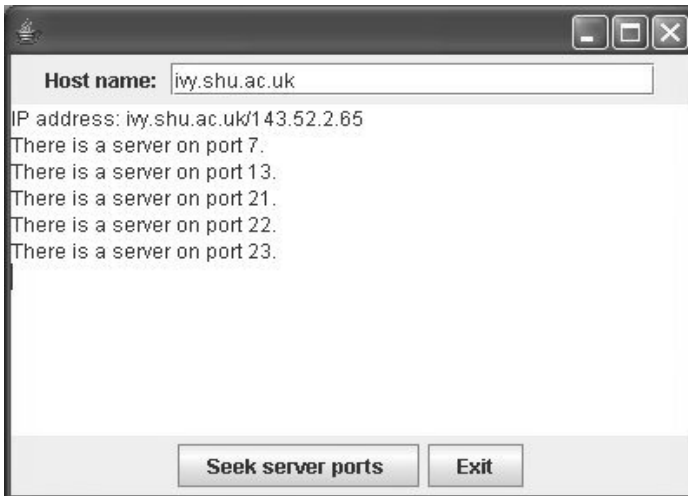


Figure 2.8 Example output from the PortScanner program.

## 2.4 Downloading Web Pages

Just one of the multitude of useful features of Java is its ability to render HTML pages as a browser would do, including the correct handling of hyperlinks contained within those pages. The class used to hold a Web page in a Java program is *JEditorPane*, which automatically renders HTML formatted text for any Web page that is downloaded via the *setPage* method of the *JEditorPane* object. (It also supports plain text and Rich Text Format, but attention will be devoted solely to HTML formatted text here.) The handling of hyperlinks requires only a modest amount of extra coding on the part of the Java programmer, as described in the following paragraphs.

If hyperlinks are contained within a downloaded page, a *HyperlinkEvent* is generated when the user clicks on one of these and must be handled by a *HyperlinkListener* (i.e., an object that implements the *HyperlinkListener* interface). A *HyperlinkEvent* is also generated when the user's mouse either moves over the hyperlink or moves away from it. Both of these actions may also cause processing activity to take place, if the application requires this (and may be ignored if it doesn't). In order to implement the *HyperlinkListener* interface, the listener object must provide a definition for method *hyperlinkUpdate*, which takes the *HyperlinkEvent* that occurred as its single argument. Method *hyperlinkUpdate*, of course, will specify the action that is to take place when a *HyperlinkEvent* occurs.

The first thing that method *hyperlinkUpdate* needs to ascertain is just which of the three possible *HyperlinkEvents* has just occurred. Class *HyperlinkEvent* contains a public inner class *EventType* that defines three constants for possible hyperlink event types:

- *ACTIVATED* (user clicked a hyperlink);
- *ENTERED* (mouse moved over a hyperlink);
- *EXITED* (mouse moved away from a hyperlink).

Method *getEventType* (of class *HyperlinkEvent*) returns one of the above three constants.

### Example

The following program displays the contents of a file at a user-specified URL, effectively acting as a simple browser. A text field is used to accept the user's URL string and a *JEditorPane* object is used to render the Web page at the specified URL. Since the *JEditorPane*'s size may very well be inadequate to display the full page, the pane is 'wrapped' in a *JScrollPane* object that will allow the user to scroll both vertically and horizontally on the page. The application frame states that it implements the *ActionListener* interface, thereby undertaking to provide a definition for the *actionPerformed* method. This method will specify the action to be carried out when the user presses <Enter> in the URL text field (both for the initial entry and for any subsequent, non-hyperlink changes of URL). Since this action is the same as that to be carried out when any hyperlink is clicked, this code has been placed inside a separate method called *showPage* (to avoid code duplication).

As for the *HyperlinkListener* interface, this is implemented by a private inner class called *LinkListener* (which means, of course, that it supplies a definition for method *hyperlinkUpdate*). If the *HyperlinkEvent* object indicates that the user clicked upon a hyperlink, then method *showPage* is called to display the page at the other end of the hyperlink. If, on the other hand, either of the other two possible hyperlink events occurred, then no action is taken in this application.

Method *showPage* renders the new page by calling method *setPage* of the *JEditorPane* object and then displays the URL of the page in the text field. Note that, if this latter step is not carried out, a runtime error will occur!

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

public class GetWebPage extends JFrame
    implements ActionListener
{
    private JLabel prompt;    //Cues user to enter a URL.
    private JTextField sourceName;    //Holds URL string.
    private JPanel requestPanel;    //Contains prompt
                                //and URL string.
    private JEditorPane contents;    //Holds page.

    public static void main(String[] args)
    {
        GetWebPage frame = new GetWebPage();
        frame.setSize(700,500);
        frame.setVisible(true);

        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public GetWebPage()
    {
        setTitle("Simple Browser");

        requestPanel = new JPanel();
        prompt = new JLabel("Required URL: ");
        sourceName = new JTextField(25);
        sourceName.addActionListener(this);
        requestPanel.add(prompt);
        requestPanel.add(sourceName);
        add(requestPanel, BorderLayout.NORTH);
        contents = new JEditorPane();
```

```
//We don't want the user to be able to alter the
//contents of the Web page display area, so...
contents.setEditable(false);

//Create object that implements HyperlinkListener
//interface...
LinkListener linkHandler = new LinkListener();

//Make the above object a HyperlinkListener for
//our JEditorPane object...
contents.addHyperlinkListener(linkHandler);

//'Wrap' the JEditorPane object inside a
//JScrollPane, to provide scroll bars...
add(new JScrollPane(contents),
    BorderLayout.CENTER);
}

public void actionPerformed(ActionEvent event)
//Called when the user presses <Enter>
//after keying a URL into the text field
//and also when a hyperlink is clicked.
{
    showPage(sourceName.getText());
}

private class LinkListener
    implements HyperlinkListener
{
    public void hyperlinkUpdate(HyperlinkEvent event)
    {
        if (event.getEventType() ==
            HyperlinkEvent.EventType.ACTIVATED)
            showPage(event.getURL().toString());
        //Other hyperlink event types ignored.
    }
}

private void showPage(String location)
{
    try
    {
        //Reset page displayed on JEditorPane...
        contents.setPage(location);

        //Reset URL string in text field...
        sourceName.setText(location);
    }
}
```

```

catch(IOException ioEx)
{
    JOptionPane.showMessageDialog(this,
        "Unable to retrieve URL",
        "Invalid URL",
        JOptionPane.ERROR_MESSAGE);
}
}
}

```

Figure 2.9 below shows some example output from running the above program.

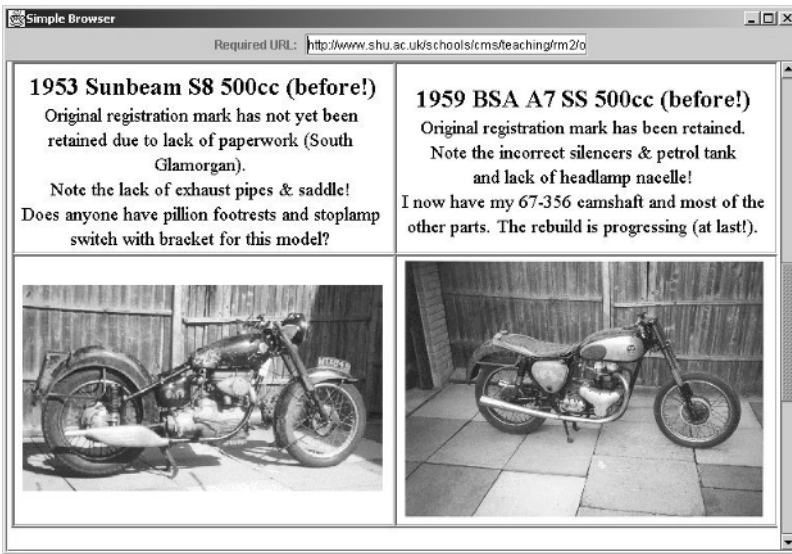


Figure 2.9 Example output from the GetWebPage program.  
 (Screenshot by kind permission of Dr Ray McLaughlin.)

Before closing this chapter, it is worth mentioning another class from the *java.net* package: the *URL* class. This class has six possible constructors, but the only one that is commonly used is the one that takes a *String* object as its single argument. Method *setPage* is also overloaded, allowing the user to specify the target page as either a *String* or an object of class *URL*. It was the first of these options that was used in the preceding example, since it would have been pointless to create a *URL* object from the string simply so that we could then pass the newly-created *URL* object to the other version of *setPage*. Indeed, it is often the case that we can use the *URL* string directly in Java, without having to create a *URL* object. However, creating such an object is occasionally unavoidable. We shall encounter such a case when we consider applets in Chapter 13.



## Exercises

- 2.1 If you haven't already done so, compile programs *TCPEchoServer* and *TCPEchoClient* from Section 2.2.1 and then run them as described at the end of that section.
- 2.2 This exercise converts the above files into a simple email server and email client respectively. The server conversion has been done for you and is contained in file *EmailServer.java*, a printed version of which appears on the following pages for ease of reference. Some of the code for the client has also been provided for you and is held in file *EmailClient.java*, a printed version of which is also provided. You are to complete the coding for the client and then run the server program in one command window and the client program in each of two further command windows (so that there are two clients communicating with the server at the same time). The details of this simplified client-server application are given below.
- The server recognises only two users, called 'Dave' and 'Karen'.
  - Each of the above users has a message box on the server that can accept a maximum of 10 messages.
  - Each user may either send a one-line message to the other or read his/her own messages.
  - A count is kept of the number of messages in each mailbox. As another message is received, the appropriate count is incremented (if the maximum has not been reached). When messages are read, the appropriate count is reduced to zero.
  - When sending a message, the client sends three things: the user's name, the word 'send' and the message itself.
  - When requesting reading of mail, the client sends two things: the user's name and the word 'read'.
  - As each message is received by the server, it is added to the appropriate mailbox (if there is room). If the mailbox is full, the message is ignored.
  - When a read request is received, the server first sends an integer indicating the number of messages (possibly 0) that will be sent and then transmits the messages themselves (after which it reduces the appropriate message count to 0).
  - Each user is to be allowed to 'send' and/or 'read' as many times as he/she wishes, until he/she decides to quit.
  - When the user selects the 'quit' option, the client sends two things: the user's name and the word 'quit'.
- 2.3 If you haven't already done so, compile and run the server program *DayTimeServer* and its associated client, *GetRemoteTime*, from Section 2.3.

2.4 Program *Echo* is similar to program *TCPEchoClient* from Section 2.2.1, but has a GUI front-end similar to that of program *GetRemoteTime* from Section 2.3. It provides an implementation of the **echo** protocol (on port 7). This implementation sends one-line messages to a server and uses the following components:

- a text field for input of messages (in addition to the text field for input of host name);
- a text area for the (cumulative) echoed responses from the server;
- a button to close the connection to the host.

Some of the code for this program has been provided for you in file *Echo.java*, a printed copy of which appears at the end of this chapter. Examine this code and make the necessary additions in the places indicated by the commented lines. When you have completed the program, run it and supply the name *holly.shu.ac.uk* (or that of any other convenient server) when prompted for a server name.

//For use with exercise 2.2.

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

public class EmailServer
{
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;
    private static final String client1 = "Dave";
    private static final String client2 = "Karen";
    private static final int MAX_MESSAGES = 10;
    private static String[] mailbox1 =
        new String[MAX_MESSAGES];
    private static String[] mailbox2 =
        new String[MAX_MESSAGES];
    private static int messagesInBox1 = 0;
    private static int messagesInBox2 = 0;

    public static void main(String[] args)
    {
        System.out.println("Opening connection...\n");
        try
        {
            serverSocket = new ServerSocket(PORT);
        }
        catch(IOException ioEx)
        {
```

```

        System.out.println(
            "Unable to attach to port!");
        System.exit(1);
    }
    do
    {
        try
        {
            runService();
        }
        catch (InvalidClientException icException)
        {
            System.out.println("Error: " + icException);
        }
        catch (InvalidRequestException irException)
        {
            System.out.println("Error: " + irException);
        }
    }while (true);
}

private static void runService()
    throws InvalidClientException,
           InvalidRequestException
{
    try
    {
        Socket link = serverSocket.accept();

        Scanner input =
            new Scanner(link.getInputStream());
        PrintWriter output =
            new PrintWriter(
                link.getOutputStream(),true);

        String name = input.nextLine();
        String sendRead = input.nextLine();
        if (!name.equals(client1) &&
            !name.equals(client2))
            throw new InvalidClientException();
        if (!sendRead.equals("send") &&
            !sendRead.equals("read"))
            throw new InvalidRequestException();

        System.out.println("\n" + name + " "
            + sendRead + "ing mail...");

        if (name.equals(client1))

```

```

    {
        if (sendRead.equals("send"))
        {
            doSend(mailbox2,messagesInBox2,input);
            if (messagesInBox2<MAX_MESSAGES)
                messagesInBox2++;
        }
        else
        {
            doRead(mailbox1,messagesInBox1,output);
            messagesInBox1 = 0;
        }
    }
    else    //From client2.
    {
        if (sendRead.equals("send"))
        {
            doSend(mailbox1,messagesInBox1,input);
            if (messagesInBox1<MAX_MESSAGES)
                messagesInBox1++;
        }
        else
        {
            doRead(mailbox2,messagesInBox2,output);
            messagesInBox2 = 0;
        }
    }

    link.close();
}
catch(IOException ioEx)
{
    ioEx.printStackTrace();
}
}

private static void doSend(String[] mailbox,
                           int messagesInBox, Scanner input)
{
    /*
    Client has requested 'sending', so server must
    read message from this client and then place
    message into message box for other client (if
    there is room).
    */
    String message = input.nextLine();
    if (messagesInBox == MAX_MESSAGES)
        System.out.println("\nMessage box full!");
}

```

```
        else
            mailbox[messagesInBox] = message;
    }

    private static void doRead(String[] mailbox,
                               int messagesInBox, PrintWriter output)
    {
        /*
        Client has requested 'reading', so server must
        read messages from other client's message box and
        then send those messages to the first client.
        */
        System.out.println("\nSending " + messagesInBox
                           + " message(s).\n");
        output.println(messagesInBox);
        for (int i=0; i<messagesInBox; i++)
            output.println(mailbox[i]);
    }
}

class InvalidClientException extends Exception
{
    public InvalidClientException()
    {
        super("Invalid client name!");
    }
    public InvalidClientException(String message)
    {
        super(message);
    }
}

class InvalidRequestException extends Exception
{
    public InvalidRequestException()
    {
        super("Invalid request!");
    }
    public InvalidRequestException(String message)
    {
        super(message);
    }
}
```

---

```
//For use with exercise 2.2.
```

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

public class EmailClient
{
    private static InetAddress host;
    private static final int PORT = 1234;
    private static String name;
    private static Scanner networkInput, userEntry;
    private static PrintWriter networkOutput;

    public static void main(String[] args)
                                throws IOException
    {
        try
        {
            host = InetAddress.getLocalHost();
        }
        catch(UnknownHostException uhEx)
        {
            System.out.println("Host ID not found!");
            System.exit(1);
        }

        userEntry = new Scanner(System.in);
        do
        {
            System.out.print(
                "\nEnter name ('Dave' or 'Karen'): ");
            name = userEntry.nextLine();
        }while (!name.equals("Dave")
                && !name.equals("Karen"));

        talkToServer();
    }

    private static void talkToServer() throws IOException
    {
        String option, message, response;

        do
        {
```

```

/*****
    CREATE A SOCKET, SET UP INPUT AND OUTPUT STREAMS,
    ACCEPT THE USER'S REQUEST, CALL UP THE APPROPRIATE
    METHOD (doSend OR doRead), CLOSE THE LINK AND THEN
    ASK IF USER WANTS TO DO ANOTHER READ/SEND.
*****/

        }while (!option.equals("n"));

    }

    private static void doSend()
    {
        System.out.println("\nEnter 1-line message: ");
        String message = userEntry.nextLine();
        networkOutput.println(name);
        networkOutput.println("send");
        networkOutput.println(message);
    }

    private static void doRead() throws IOException
    {
        /*****
        BODY OF THIS METHOD REQUIRED
        *****/

    }
}

```

---

```

//For use with exercise 2.4.

```

```

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

public class Echo extends JFrame
                        implements ActionListener
{
    private JTextField hostInput,lineToSend;
    private JLabel hostPrompt,messagePrompt;
    private JTextArea received;

```

```

private JButton closeConnection;
private JPanel hostPanel,entryPanel;
private final int ECHO = 7;
private static Socket socket = null;
private Scanner input;
private PrintWriter output;

public static void main(String[] args)
{
    Echo frame = new Echo();
    frame.setSize(600,400);
    frame.setVisible(true);

    frame.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                if (socket != null)
                {
                    try
                    {
                        socket.close();
                    }
                    catch (IOException ioEx)
                    {
                        System.out.println(
                            "\n* Unable to close link! *\n");
                        System.exit(1);
                    }
                    System.exit(0);
                }
            }
        }
    );
}

public Echo()
{
    hostPanel = new JPanel();

    hostPrompt = new JLabel("Enter host name:");
    hostInput = new JTextField(20);
    hostInput.addActionListener(this);
    hostPanel.add(hostPrompt);
    hostPanel.add(hostInput);
    add(hostPanel, BorderLayout.NORTH);
}

```



```

    entryPanel = new JPanel();

    messagePrompt = new JLabel("Enter text:");
    lineToSend = new JTextField(15);

    //Change field to editable when
    // host name entered...
    lineToSend.setEditable(false);

    lineToSend.addActionListener(this);

    /*****
    * ADD COMPONENTS TO PANEL AND APPLICATION FRAME *
    *****/

    /*****
    * NOW SET UP TEXT AREA AND THE CLOSE BUTTON *
    *****/
}

public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == closeConnection)
    {
        if (socket != null)
        {
            try
            {
                socket.close();
            }
            catch(IOException ioEx)
            {
                System.out.println(
                    "\n* Unable to close link!*\n");
                System.exit(1);
            }
            lineToSend.setEditable(false);
            hostInput.grabFocus();
        }
        return;
    }

    if (event.getSource() == lineToSend)
    {
        /*****/
        * SUPPLY CODE HERE *
        /*****/
    }
}

```

```
}

//Must have been entry into host field...
String host = hostInput.getText();
try
{
    /**
     * SUPPLY CODE HERE *
     */

}
catch (UnknownHostException uhEx)
{
    received.append("\n*** No such host! ***\n");
    hostInput.setText("");
}
catch (IOException ioEx)
{
    received.append("\n*** " + ioEx.toString()
                    + " ***\n");
}
}
```