Advanced Java Programming

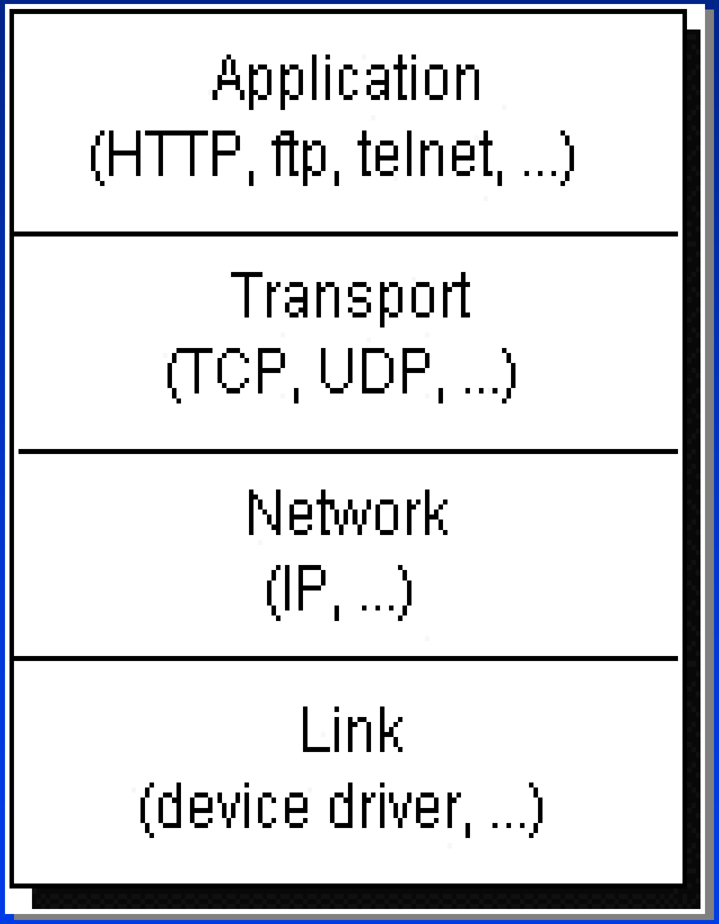
Networking

Eran Werner and Ohad Barzilay

Tel-Aviv University

Overview of networking

TCP/IP protocol suite



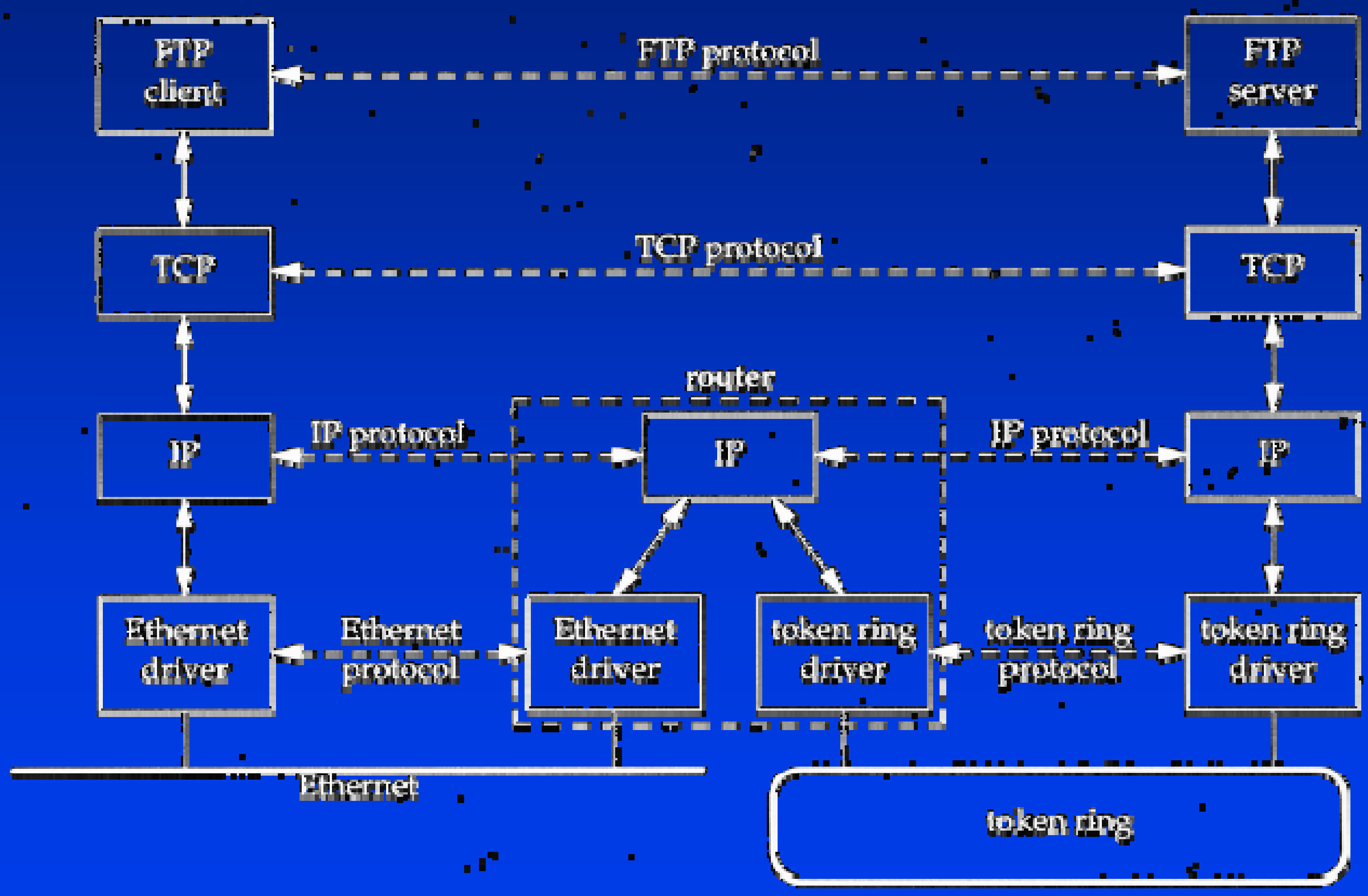
Application
(HTTP, ftp, telnet, ...)

Transport
(TCP, UDP, ...)

Network
(IP, ...)

Link
(device driver, ...)

TCP/IP protocol suite



The Network and Link layers

The link layer:

- Handles movement of data across a physical link in a network.
- Depends on the network type (Ethernet, Token-Ring).
- Also performs error checking on the arriving data

The network layer:

- Handles transmission of packets between two hosts.
- Uses the *IP protocol*.
- The packets travel in the net using *IP addresses* and *Routing Protocols*.

The Transport Layer

The two main transport protocols are *TCP* and *UDP*:

- **TCP (Transmission Control Protocol):**
  - Emulates a reliable point to point stream of data between applications, using IP and a port number.
  - Tags each packet with a number and orders the packets.
  - Retransmits lost packets.

The Transport Layer

- **UDP (Unreliable datagram protocol):**
  - Enables applications to send unreliable packets of data (datagrams), with no guarantees about order and arrival.
  - Uses underlying IP and a port number.

TCP vs. UDP

TCP is mostly used when application needs reliability (which comes at a cost). Most applications use TCP (telnet, FTP, HTTP, SMTP etc).

UDP is used by applications which do not want to pay the overhead of slowdown by TCP and don’t need reliability.

Ports

A computer has a single physical connection to the network. All arriving data goes through this connection.

The data may be intended for different applications that run on the computer.

Problem:

How does the computer know to which application to forward the data?

Ports

Ports are 16-bit numbers used to map incoming data to a process running on the computer.

When sending data in the transport layer, the port must be supplied together with the IP address.

Ports

Port numbers 0..1023 are reserved for use by well-known services:

- FTP: 21
- Telnet: 23
- HTTP: 80
- …

Application Layer Protocols

Hyper Text Transfer Protocol (HTTP)

- Used between:
  - Web clients (e.g. browsers) and
  - web-servers (or proxies)
- Text based
- Build on top of TCP
- Stateless protocol

HTTP Transaction

Client request:

- Sends request
  `GET http://www.cs.tau.ac.il/ HTTP/1.0`
- Sends header information
  `User-Agent:` *browser name*
  `Accept:` *formats recognized by the browser*
  `Host: www.cs.tau.ac.il`

  `...`
- Sends a blank line (**`\n`**)
- Can send post data

HTTP Transaction (cont.)

Server response:

- Sends status line

```
HTTP/1.0 200 OK
```

- Sends header information

```
Content-type: text/html
Content-length: 302
...
```

- Sends a blank line (**\n**)
- Sends document data (if any)

Working with URLs

Using TCP in Java

The following Java classes all use TCP:

- `URL`
- `URLConnection`
- `Socket`
- `ServerSocket`

URL (Uniform resource locator)

Each information piece on the Web has a unique identifying address which is called a URL

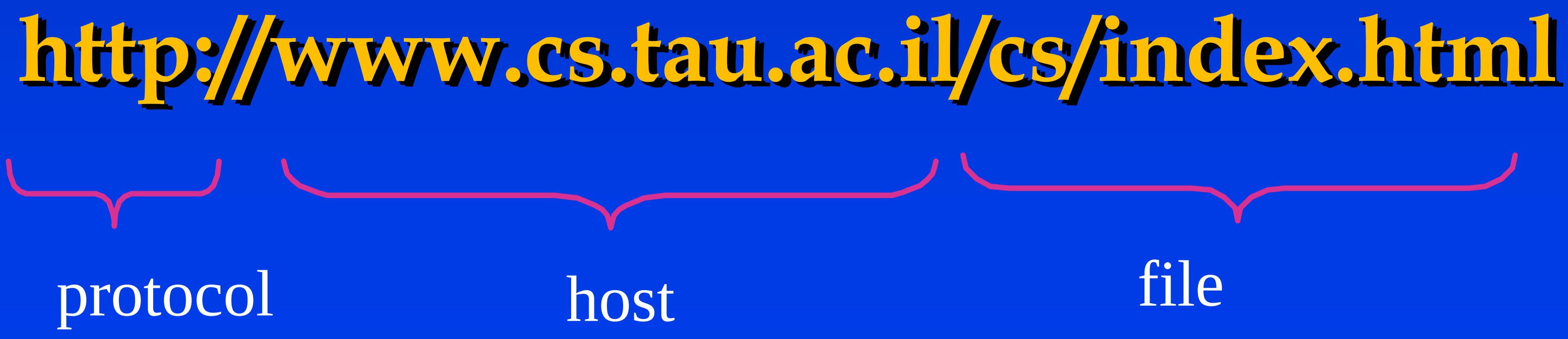
A URL has two main components :

- Protocol identifier
- Resource name

URL (Uniform resource locator)

The Resource names usually includes

- Host name
- File name
- Port number (Optional)
- Reference (Optional)



http://www.cs.tau.ac.il/cs/index.html

protocol

host

file

URL

The java.net.URL class represents a URL.

Connecting to a URL involves parsing the URL, resolving the protocol and path, connecting to the host's socket and communicating with the server through the protocol.

Java's URL object does all that for you. You need only construct a URL object, the parsing is done by the URL class

URL

Java will access the resource using the proper protocol

All common protocols are supported, treating an ftp, http, file are done the same

Using the factory pattern, different protocol handlers are slipped behind the scenes to handle communication with the server

URL

A URL object is created in one of the following ways:

- Using the full URL (string) of a resource.
- Using separate URL info:
  - Protocol name
  - Host name
  - File name
  - Port number
- Using another URL object and a name of a file relative to it.

URL

The URL details can later be obtained from the `URL` object.

A `MalformedURLException` may be thrown when creating a `URL`, if the protocol is unknown or the resource is not found.

Reading directly from a URL

The following program reads from a URL and prints its contents:

```
public class URLReader {
  public static void main(String[] args) {
    URL tau = new URL("http://www.cs.tau.ac.il/");
    BufferedReader in = new BufferedReader(
       new InputStreamReader(tau.openStream()));

    String line;
    while ((line = in.readLine()) != null)
      System.out.println(line);

    in.close();
  }
}
```

Connecting to a URL

The **openConnection()** methods initializes a communication link to the URL over the network.

The connection is represented using the **URLConnection** object.

```
try {
  URL tau = new URL("http://www.cs.tau.ac.il/");
  URLConnection idcConnection =
                       tau.openConnection();
} catch (MalformedURLException e) {
    // new URL() failed
} catch (IOException e) {
    // openConnection() failed
}
```

Reading from a URL connection

The following program reads from a URL connection and prints its contents:

```
public class URLConnectionReader {
  public static void main(String[] args) {
    URL tau = new URL("http://www.cs.tau.ac.il/");
    URLConnection tauConnection = tau.openConnection();
    BufferedReader in = new BufferedReader(
      new InputStreamReader(tauConnection.getInputStream()));

    String inputLine;
    while ((inputLine = in.readLine()) != null) {
      System.out.println(inputLine);
    }
    in.close();
  }
}
```

Writing to a URL connection

Many HTML pages use *forms* to enable the user to enter data to be sent to the server (e.g. login forms).

The browser writes the data to the URL over the network.

Writing to a URL connection

The server-side application receives the data, processes it and sends a response (usually an HTML page).

Using the `getOutputStream()` of the `URLConnection`, a Java application can write data directly to the server-side application.

Sockets

URL & URLConnections

URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet.

Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application.

Client-Server model

A server provides some service. The clients connect to the server and use this service.

When the communication channel needs to be reliable then TCP is used, as follows:

- The client and the server programs establish a connection to one another.
- Each program binds a socket to its end of the connection.

What is a Socket ?

A socket is one end-point of a two-way communication link between two programs running on the network.

Sockets are an innovation of the Berkely UNIX. The allow the programmer to treat the network connection as yet another stream of bytes that can be written to and read from.

Sockets shield the programmer from low level details of network communication by providing an abstraction at the process level.

Socket Class

Socket classes are used to represent the connection between a client program and a server program.

Sockets can perform the following operations:

- Connect to a remote host
- Send data
- Receive data
- Close a connection

Reading and Writing from Sockets

A socket can hold an input stream **and an** output stream

The streams are used to communicate between the process, once the communication is established is if they were just like any other stream.

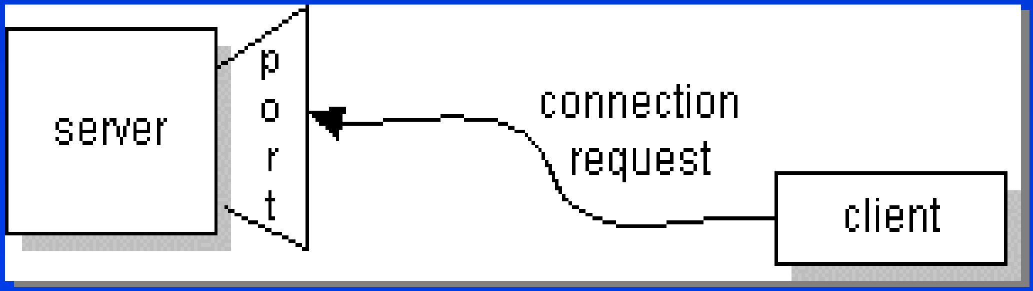
Client-Server model (cont.)

Server side:

- The server has a socket that is bound to a specific port number.
- The server listens to the socket for client requests.

Client side:

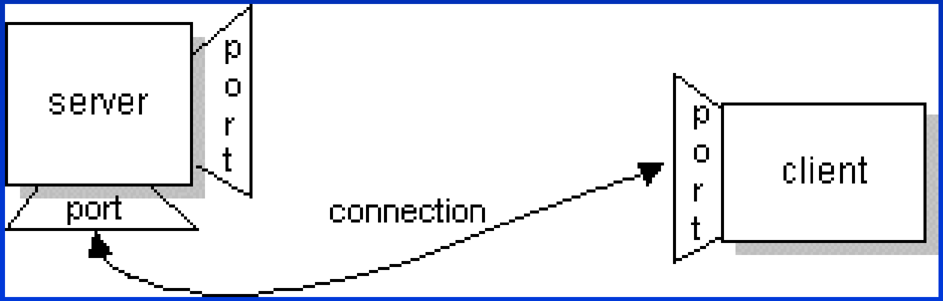
- The client knows the host name and port of the server.
- The client requests a connection from the server.



Client-Server model (cont.)

Server side:

- The server accepts the connection.
- The server bounds a new port and socket for the communication with the new client.

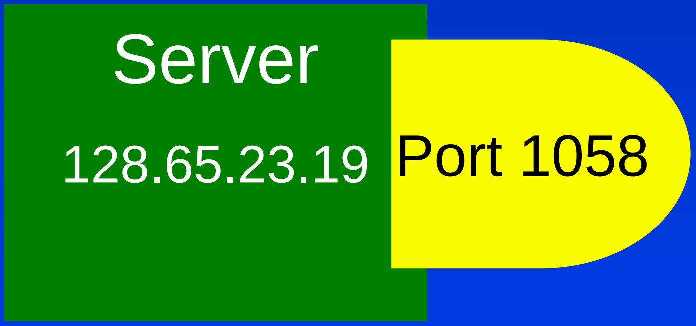


Client side:

- The client is assigned a new port in its local machine.
- A socket is bounded to this port through which the client communicates with the server.

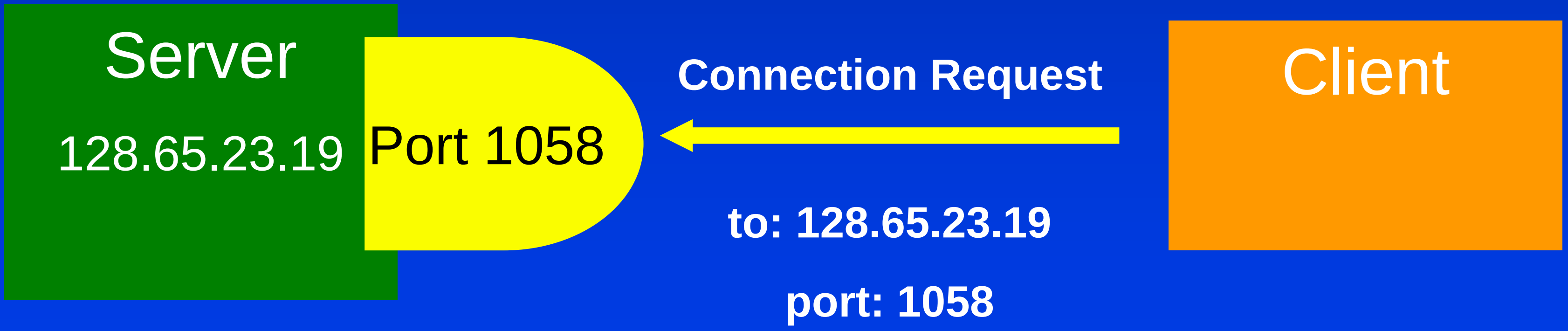
Sockets

- The server waits, listening to the socket for a client to make a connection request.



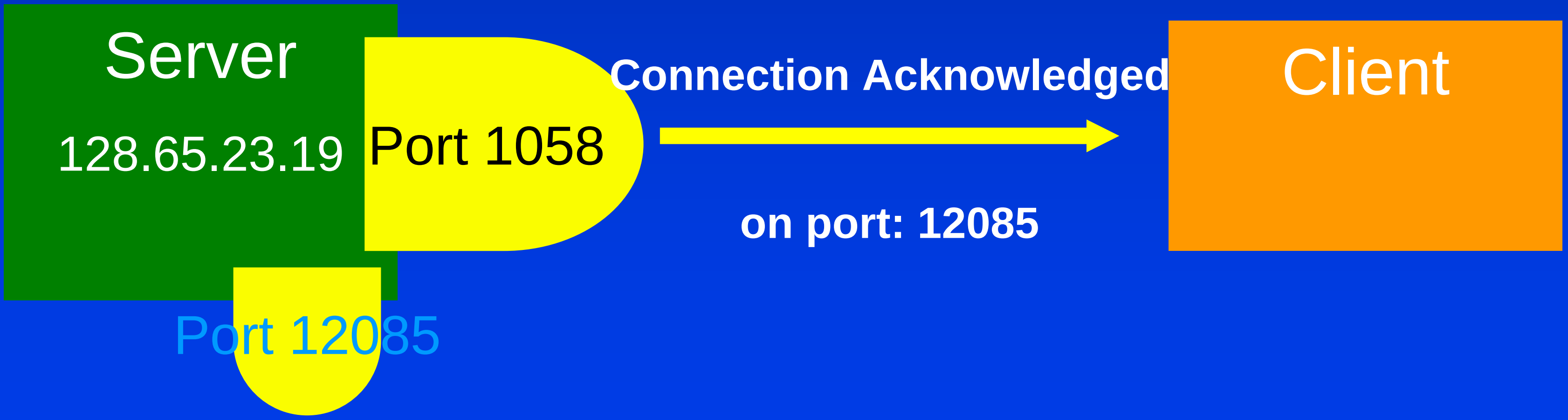
Sockets

- When a client wishes to communicate with the server, the client sends a rendezvous request to the server's address and port.



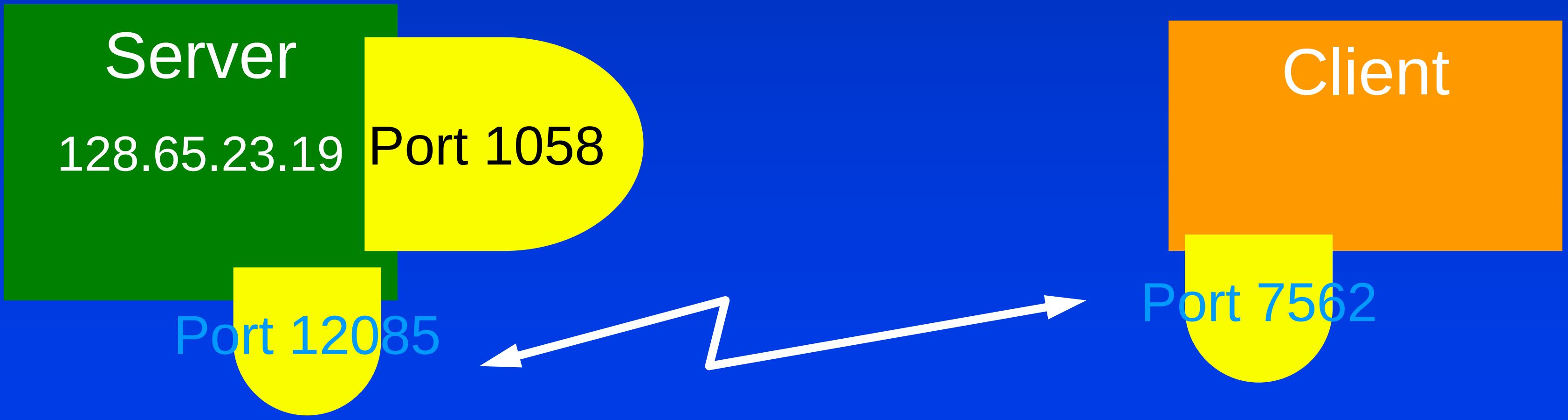
Sockets

- Upon acceptance, the server gets a new socket bound to a different port.
- This is needed in order to continue serving other clients on the original port.



Sockets

- On the client side, a new socket is created, which is bound to a local port number on the client machine.



Sockets in Java

The `java.net.Socket` class implements one side of a two-way connection between a Java program and another program on the network.

The `Socket` class hides a platform-dependent implementation. This enables Java programs to communicate in platform-independent fashion.

Sockets in Java

Remember: a socket is a communication channel between processes, therefore it is defined by the pair of host name and port number.

The `java.net.ServerSocket` class implements socket of a server.

This socket can listen for and accept connections to clients.

The Echo server example

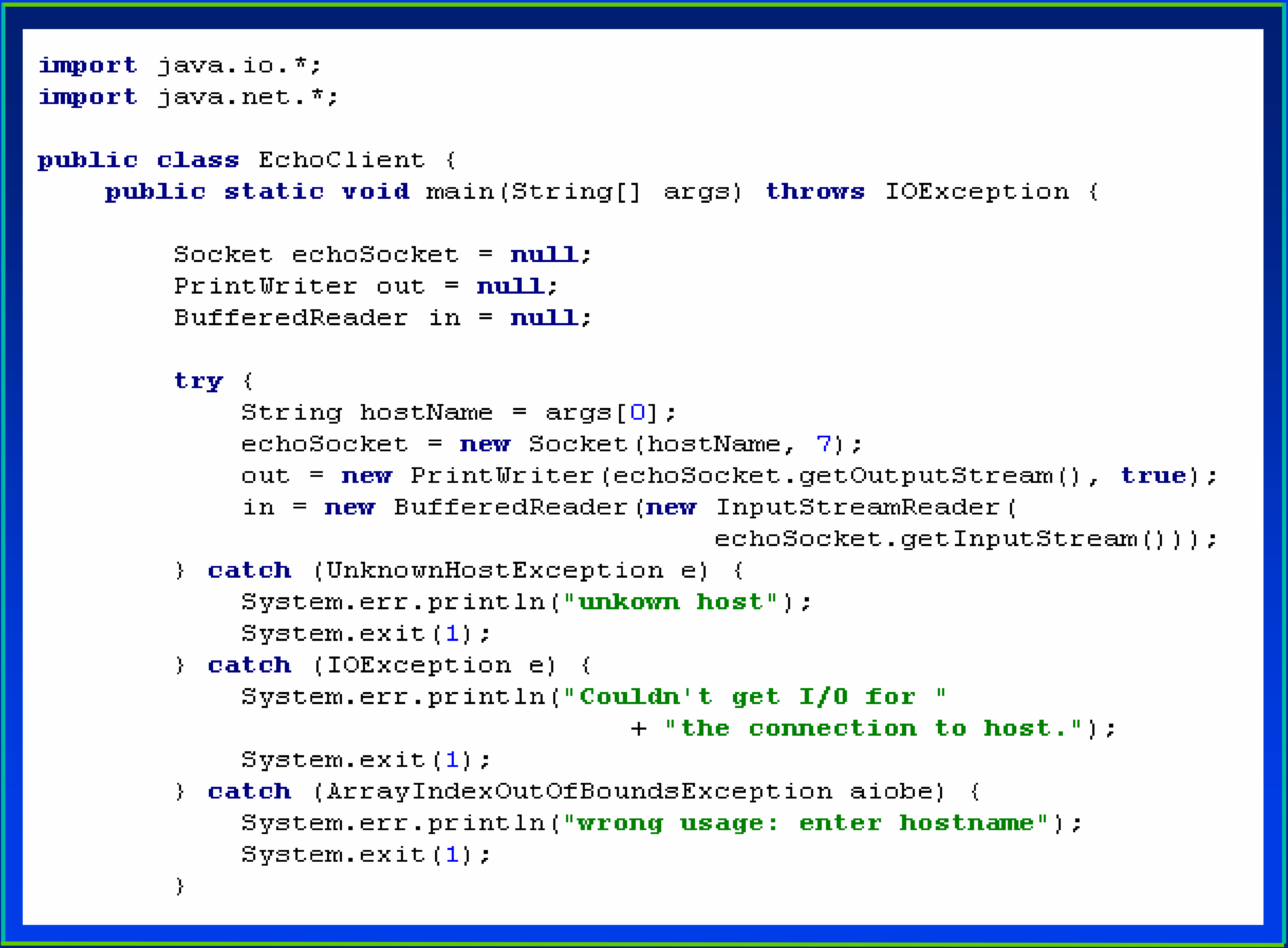
The Echo server:

- This server just echoes any data sent to it by a client.
- This is a well known service on port 7 (Both TCP and UDP).

The Echo client:

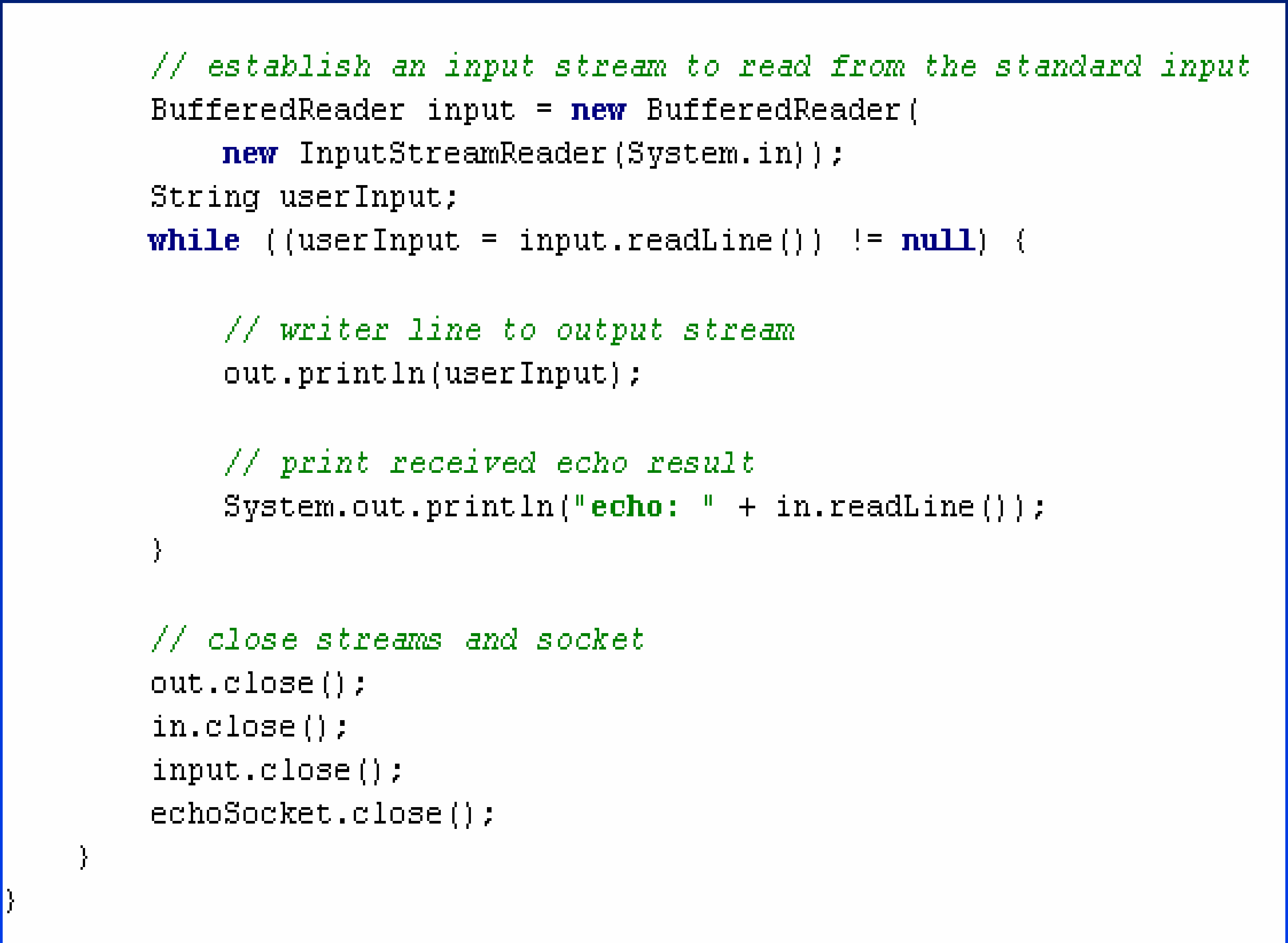
- Waits for a textual user input from the standard input.
- When a line of text is entered, it is sent to the Echo server.
- The response of the Echo server is printed.

The Echo client



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

public class EchoClient {
    public static void main(String[] args) throws IOException {

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            String hostName = args[0];
            echoSocket = new Socket(hostName, 7);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                                        echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("unkown host");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                                   + "the connection to host.");
            System.exit(1);
        } catch (ArrayIndexOutOfBoundsException aiobe) {
            System.err.println("wrong usage: enter hostname");
            System.exit(1);
        }
```

The Echo client (cont.)



```
        // establish an input stream to read from the standard input
        BufferedReader input = new BufferedReader(
            new InputStreamReader(System.in));
        String userInput;
        while ((userInput = input.readLine()) != null) {

            // writer line to output stream
            out.println(userInput);

            // print received echo result
            System.out.println("echo: " + in.readLine());
        }

        // close streams and socket
        out.close();
        in.close();
        input.close();
        echoSocket.close();
    }
}
```

Writing the Server Side

- Servers set up a port and listen for client calls
- When a client connects the server establishes input and output streams
- The Server communicates with the client according to a protocol known to both sides
- The Server can handle clients sequentially (one at a time while other clients wait)
- Or the Server can be multi-threaded and handle multiple clients in parallel.

java.net.ServerSocket

A ServerSocket can:

- Bind to a port
- Listen for incoming data
- Accept client connections on the bound port

Upon Connection the ServerSocket establishes a regular socket on a different port that will be used to communicate with the client

ServerSocket methods

```
public ServerSocket (int port)
           throws IOException,  BindException
```

- Constructs a ServerSocket bound to a port

```
public Socket accept () throws IOException
```

- Blocks and waits for a client connection on the server’s port

ServerSocket methods

```
public InetAddress getInetAddress ()
```

- Returns the Internet Address used by the server

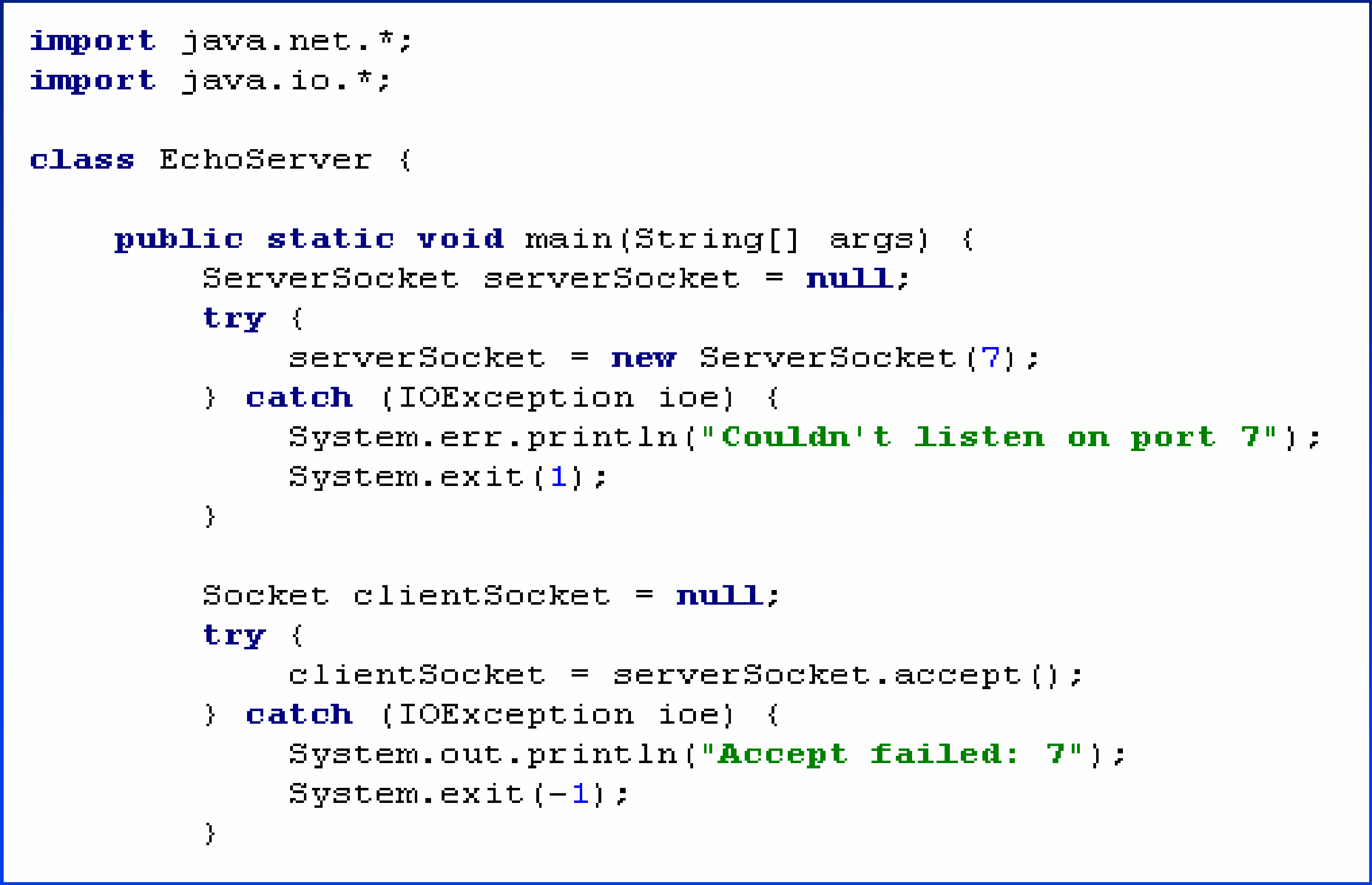
```
public int getLocalPort ()
```

- Returns the port number used by the server

```
public void close () throws IOException
```

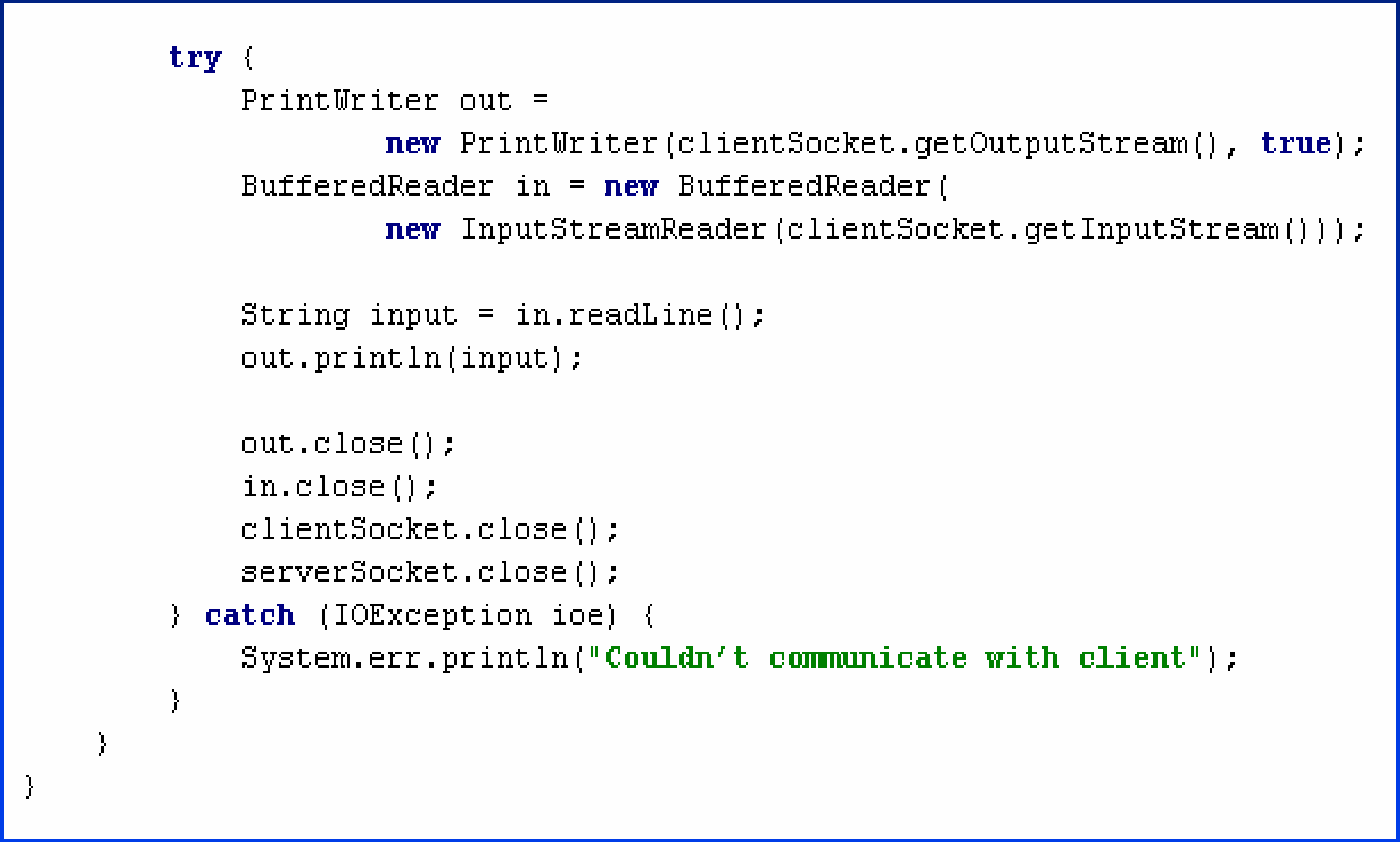
- Close the server socket (not the socket)

The Echo server



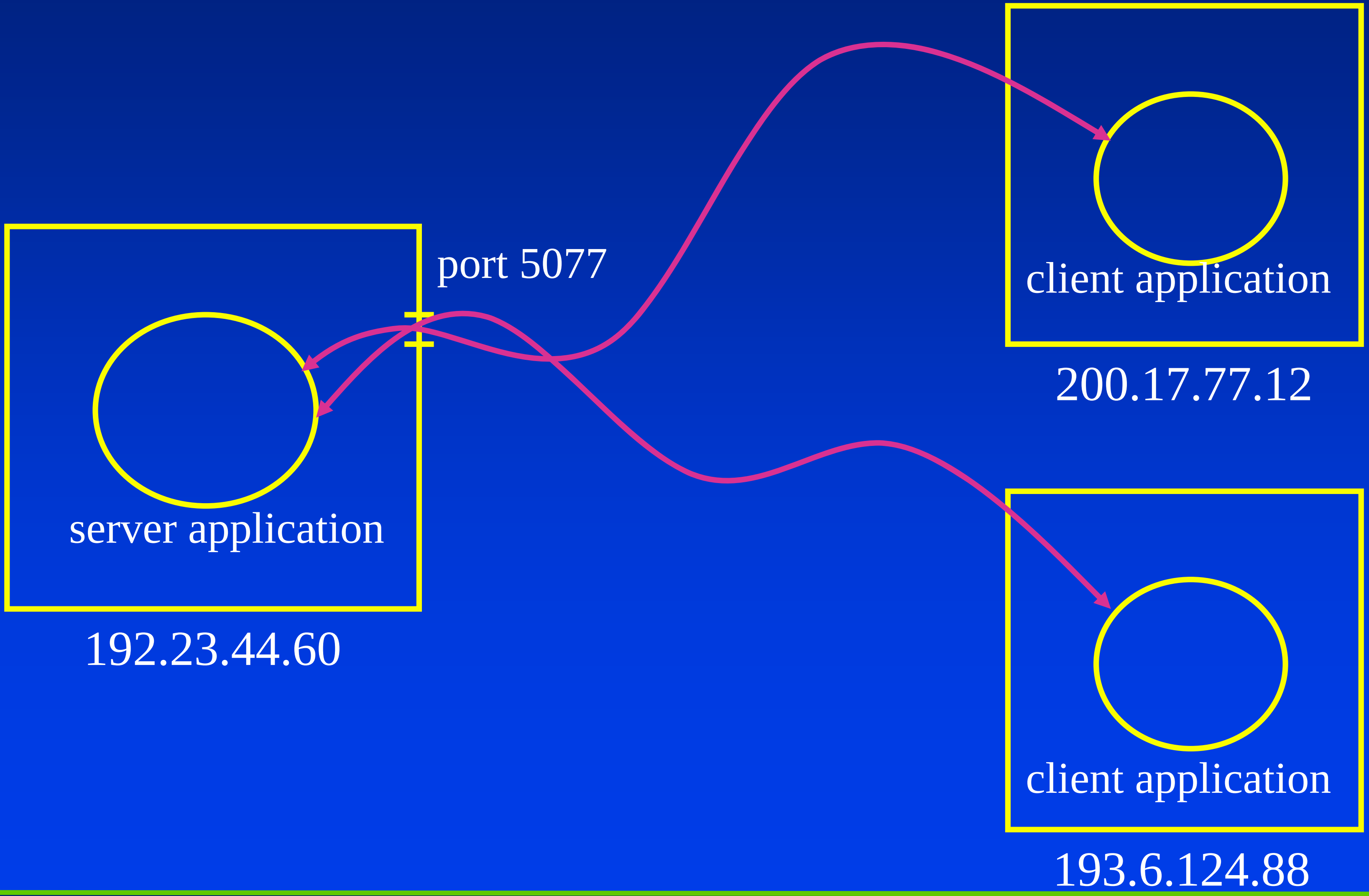
```
import java.net.*;
import java.io.*;

class EchoServer {

    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(7);
        } catch (IOException ioe) {
            System.err.println("Couldn't listen on port 7");
            System.exit(1);
        }

        Socket clientSocket = null;
        try {
            clientSocket = serverSocket.accept();
        } catch (IOException ioe) {
            System.out.println("Accept failed: 7");
            System.exit(-1);
        }
```

The Echo server (cont.)

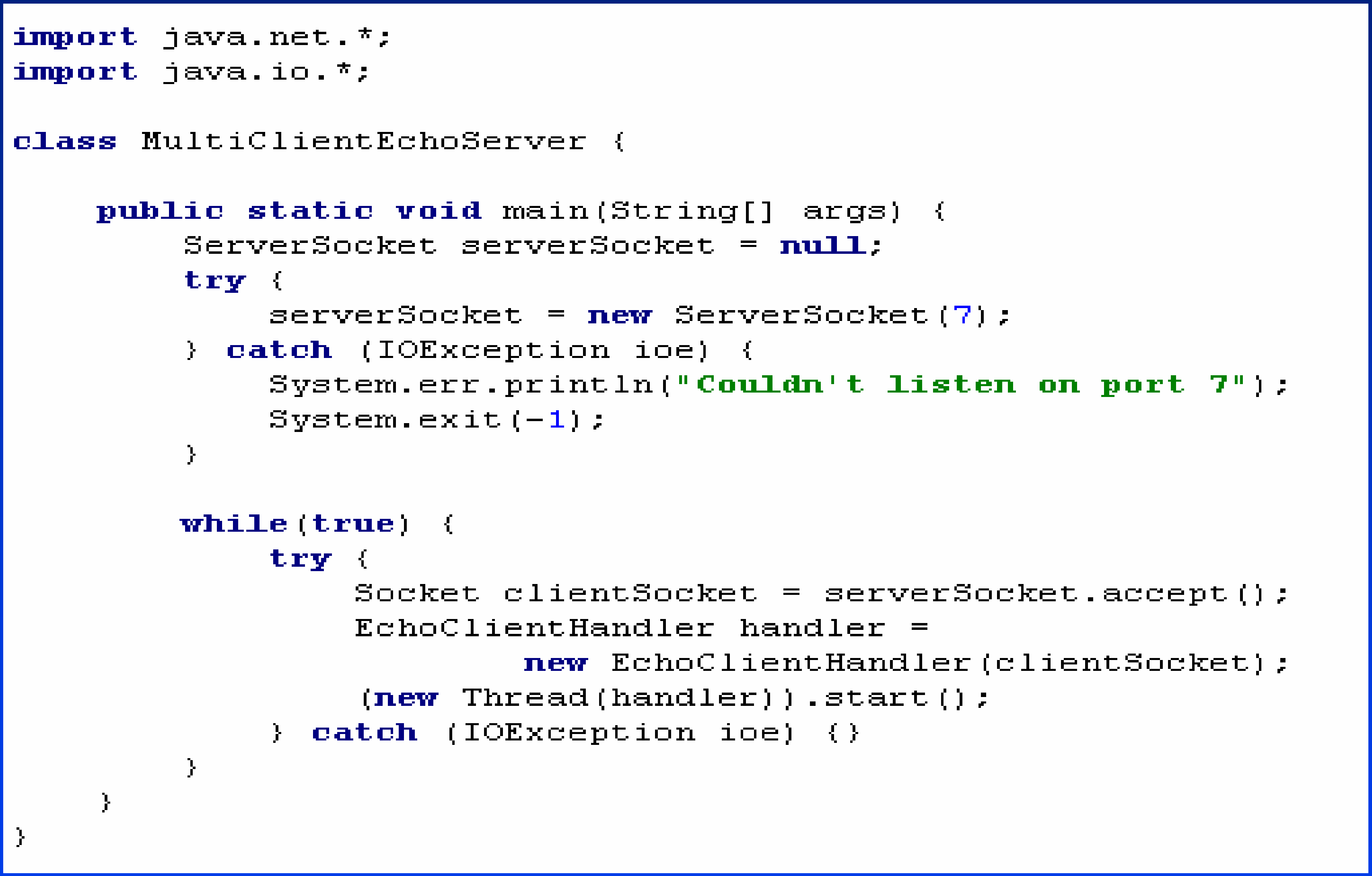


```java
        try {
            PrintWriter out =
                    new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));

            String input = in.readLine();
            out.println(input);

            out.close();
            in.close();
            clientSocket.close();
            serverSocket.close();
        } catch (IOException ioe) {
            System.err.println("Couldn't communicate with client");
        }
    }
}
```

Multiple clients



The Echo server: multiple clients



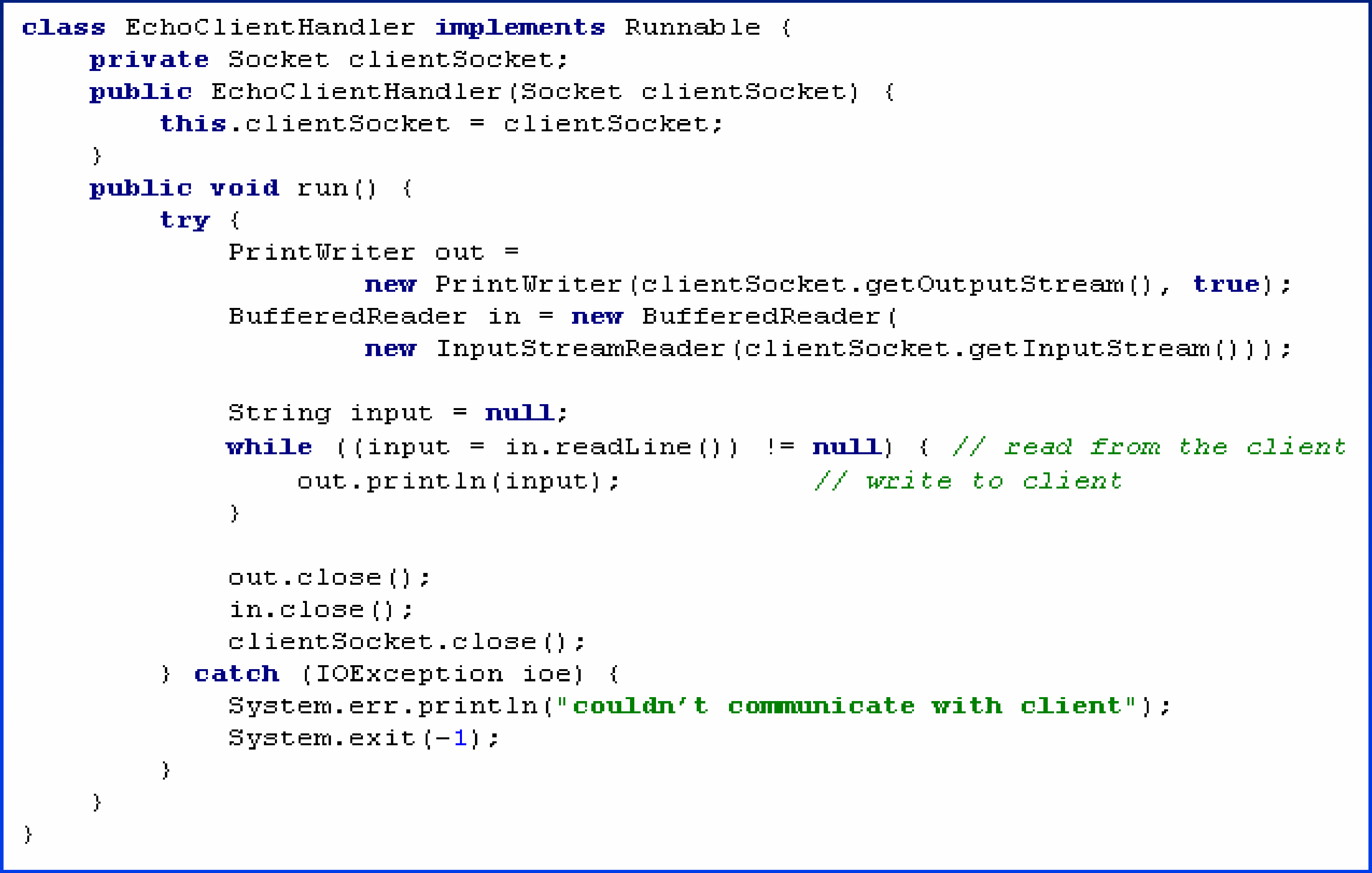
```java
import java.net.*;
import java.io.*;

class MultiClientEchoServer {

    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(7);
        } catch (IOException ioe) {
            System.err.println("Couldn't listen on port 7");
            System.exit(-1);
        }

        while(true) {
            try {
                Socket clientSocket = serverSocket.accept();
                EchoClientHandler handler =
                        new EchoClientHandler(clientSocket);
                (new Thread(handler)).start();
            } catch (IOException ioe) {}
        }
    }
}
```

The Echo server: multiple clients

We use an `EchoClientHandler` to:

- Handles a connection of an **`EchoClient`**.
- Encapsulates the task of communicating with the client in the ‘echo' protocol.

The Echo server: multiple clients



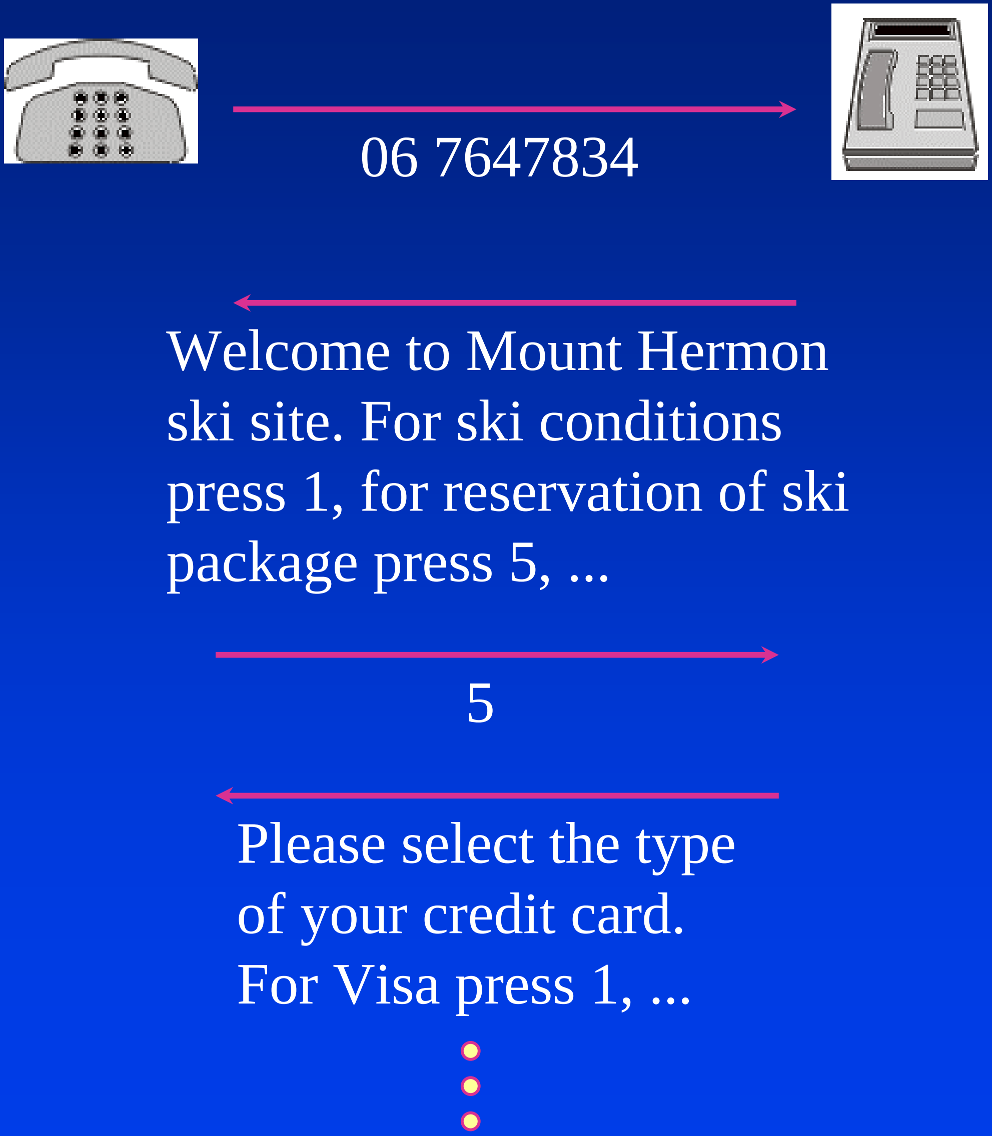
```java
class EchoClientHandler implements Runnable {
    private Socket clientSocket;
    public EchoClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }
    public void run() {
        try {
            PrintWriter out =
                    new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));

            String input = null;
            while ((input = in.readLine()) != null) { // read from the client
                out.println(input);           // write to client
            }

            out.close();
            in.close();
            clientSocket.close();
        } catch (IOException ioe) {
            System.err.println("couldn't communicate with client");
            System.exit(-1);
        }
    }
}
```

Protocols

The Client and Server should speak the same protocol.

A protocol is a specification of the syntax and interpretation of messages that can be received at any stage, and what are the possible responses at any stage (state diagram)

What is a protocol?



06 7647834

Welcome to Mount Hermon ski site. For ski conditions press 1, for reservation of ski package press 5, ...

5

Please select the type of your credit card. For Visa press 1, ...

State full vs. Stateless protocols

The HTTP is an example of a stateless protocol: when a client connects to a server, the server carries out the request, sends the reply, and does not retain any information about the request.

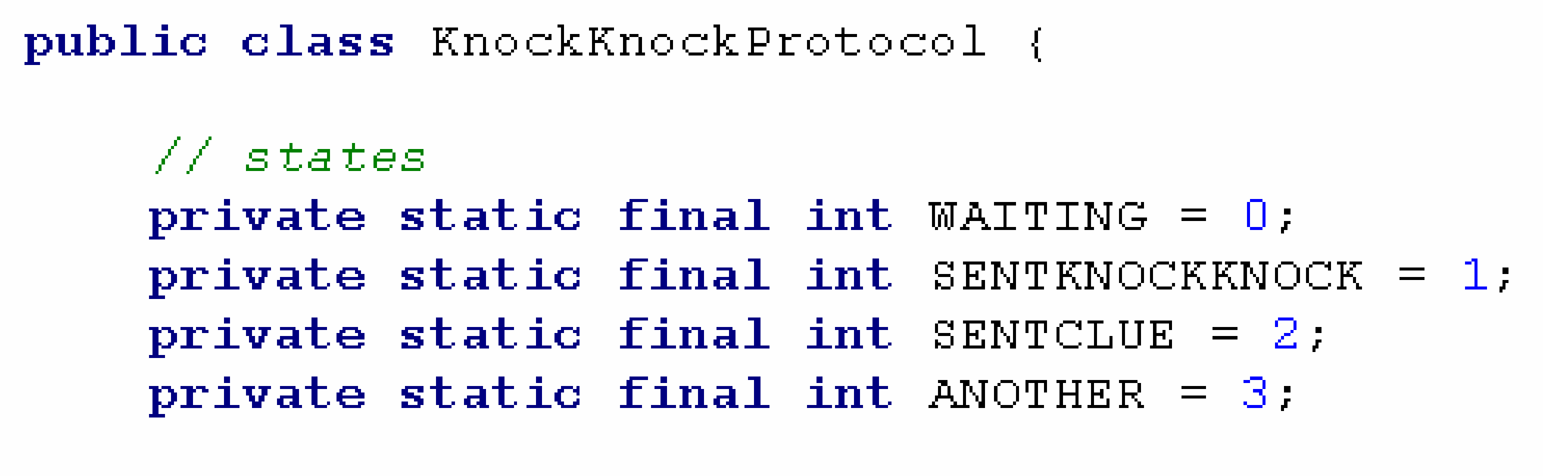
State full servers maintain information about clients between requests (e.g., shopping cart, where the client is browsing, etc.)

State full vs. Stateless Servers

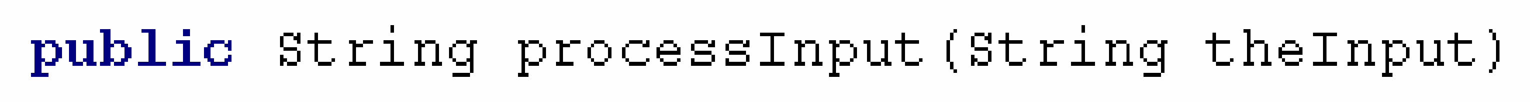
State full servers: shorter request messages, better performance, but difficult to maintain correctly (what happens if server crashes/client crashes/ a message is lost?)

Stateless servers: fault tolerant – no problem if client/server crashes, no resource constraints (memory)

Example – protocol knock knock

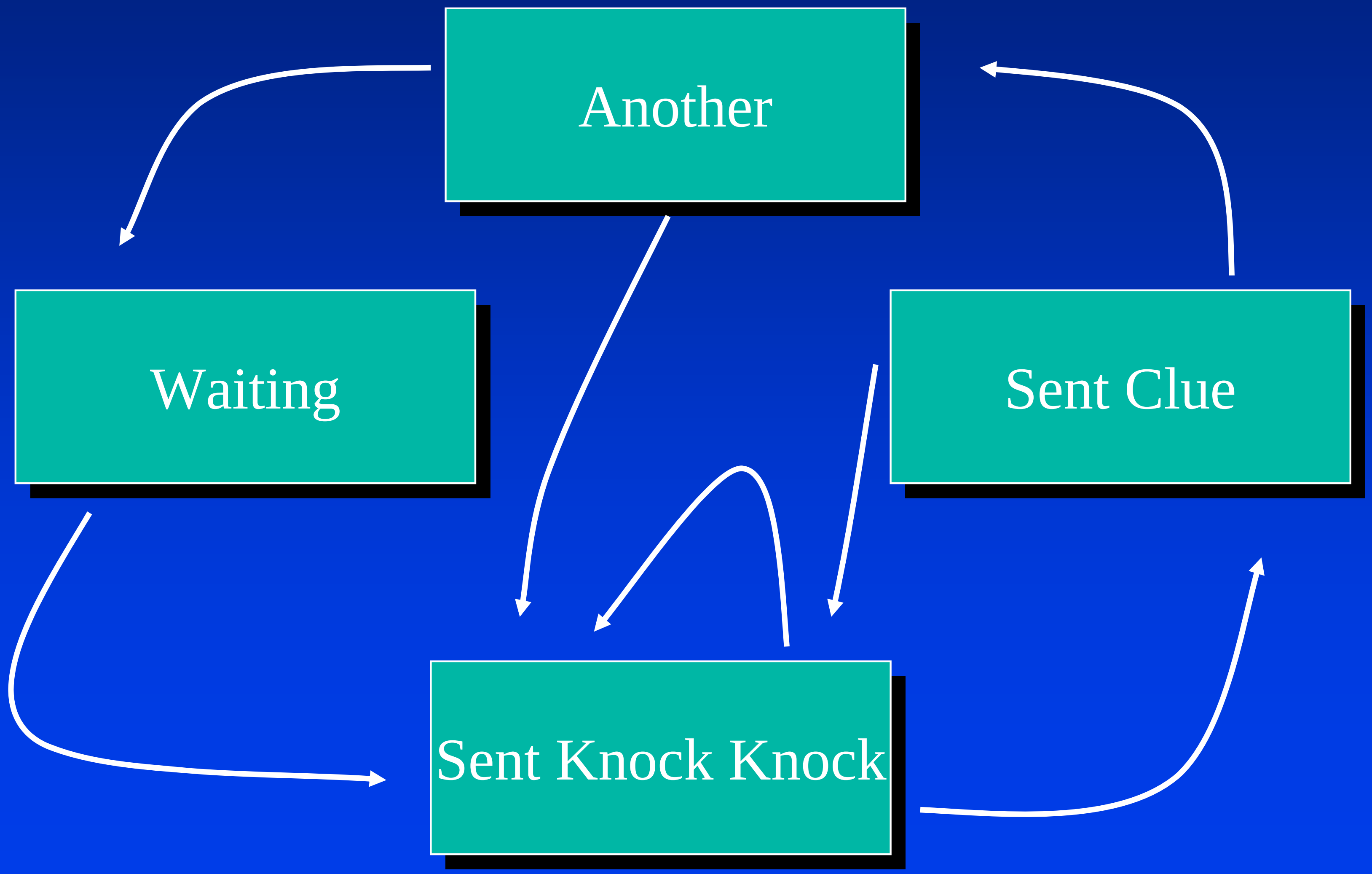


```
public class KnockKnockProtocol {

    // states
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;
```

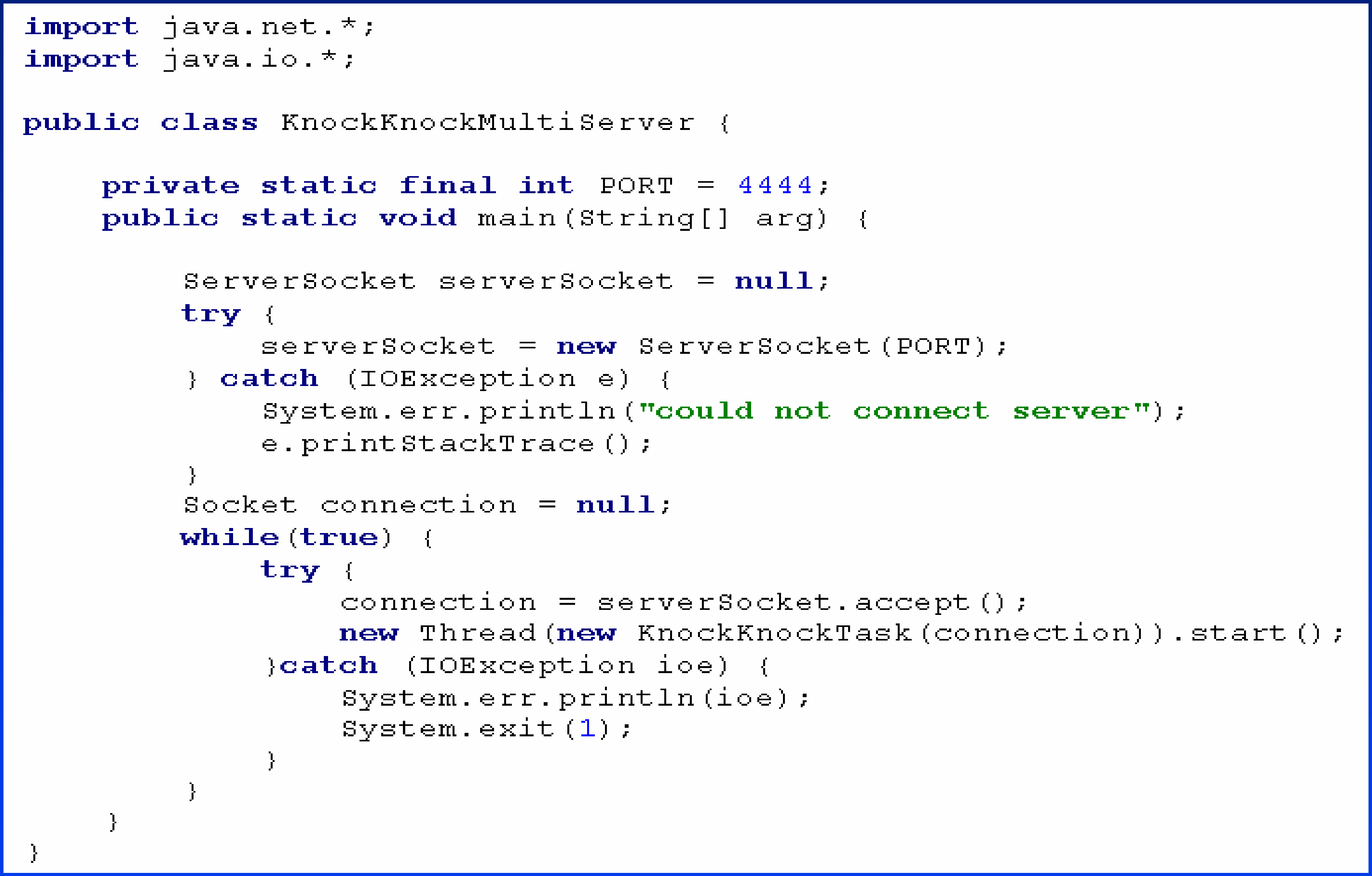


```
public String processInput(String theInput)
```

Protocol State diagram

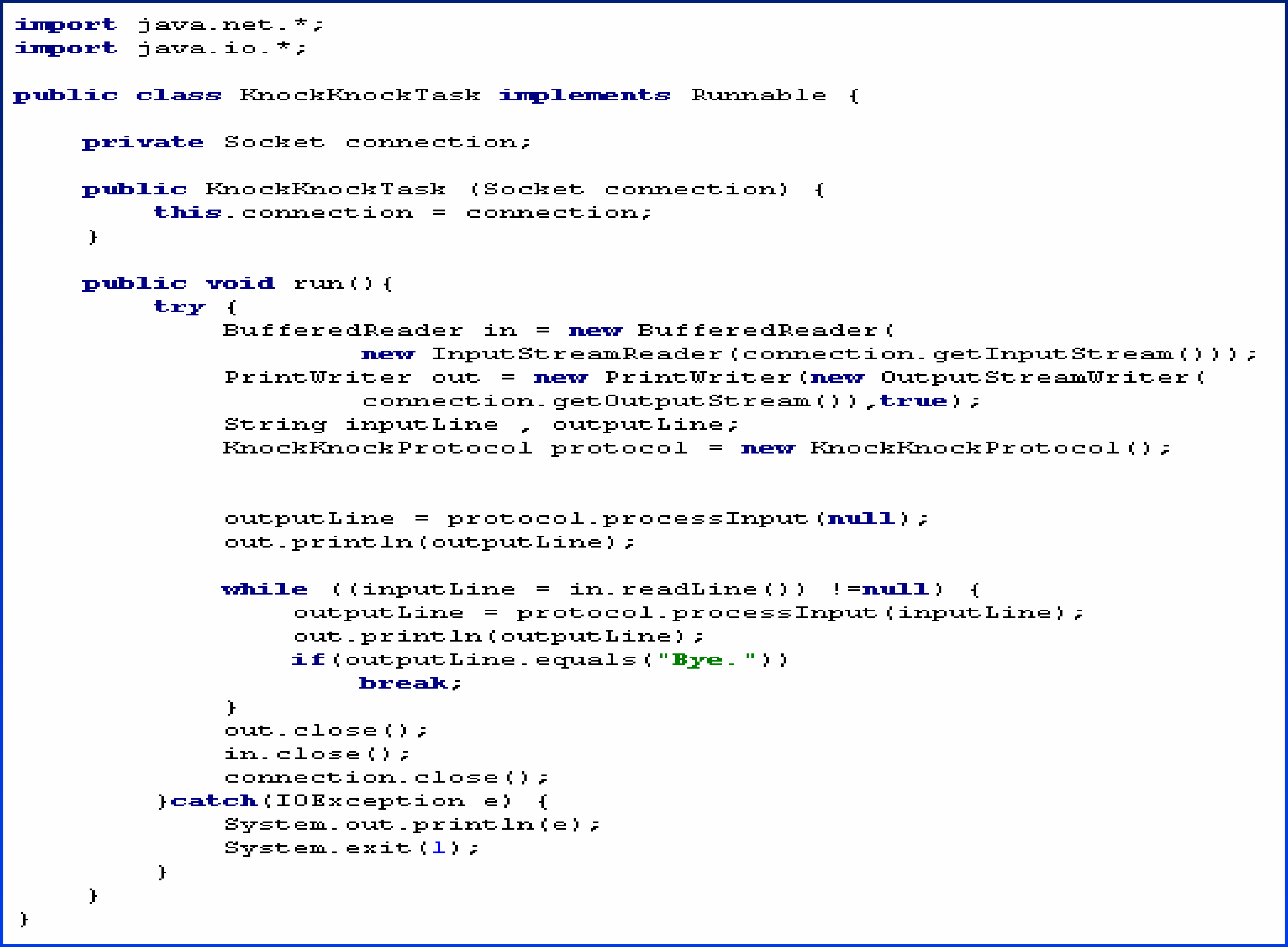


Knock knock server



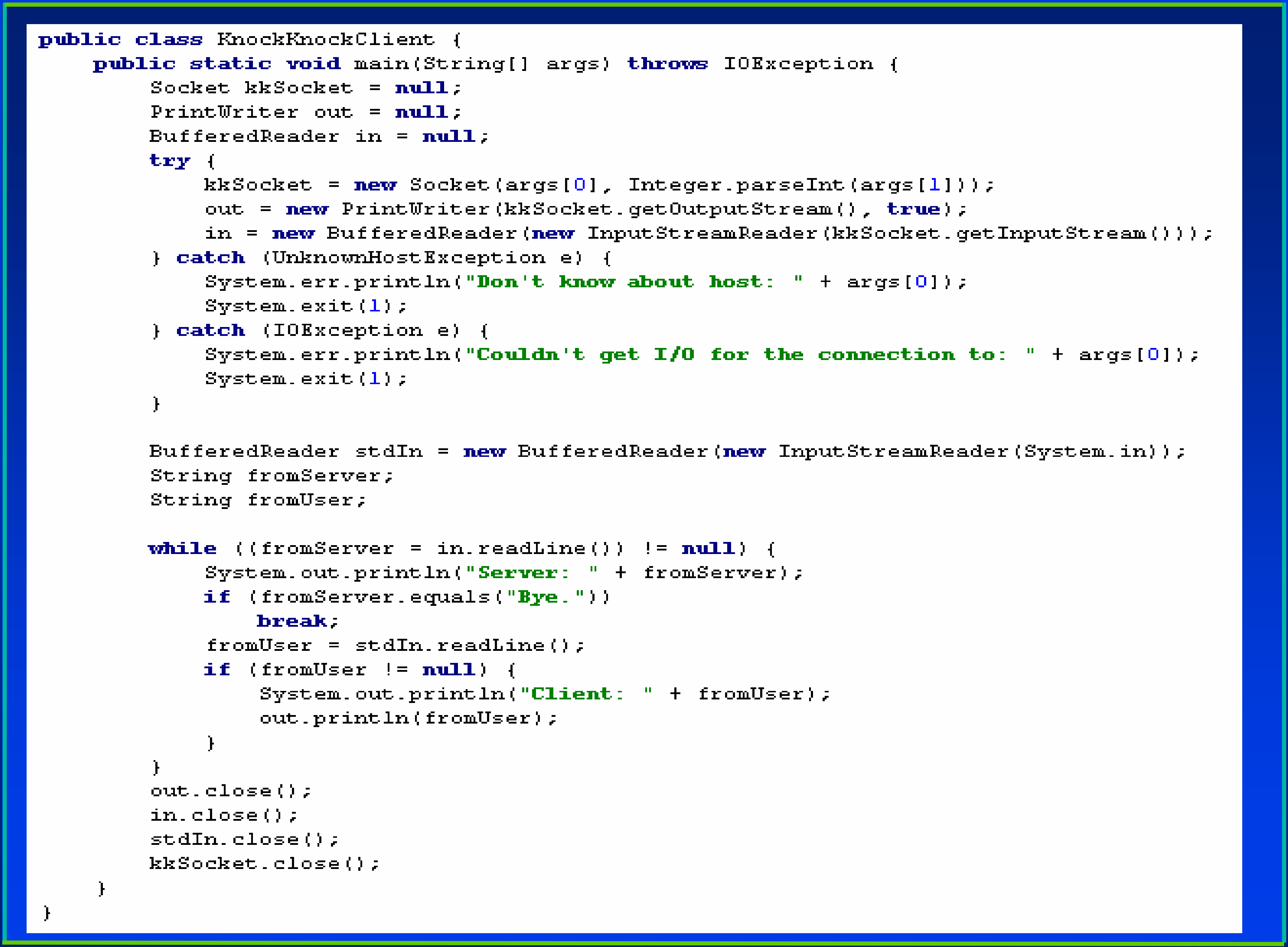
```
import java.net.*;
import java.io.*;

public class KnockKnockMultiServer {

    private static final int PORT = 4444;
    public static void main(String[] arg) {

        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(PORT);
        } catch (IOException e) {
            System.err.println("could not connect server");
            e.printStackTrace();
        }
        Socket connection = null;
        while(true) {
            try {
                connection = serverSocket.accept();
                new Thread(new KnockKnockTask(connection)).start();
            }catch (IOException ioe) {
                System.err.println(ioe);
                System.exit(1);
            }
        }
    }
}
```

Knock knock task



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

public class KnockKnockTask implements Runnable {

    private Socket connection;

    public KnockKnockTask (Socket connection) {
        this.connection = connection;
    }

    public void run(){
        try {
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(connection.getInputStream()));
            PrintWriter out = new PrintWriter(new OutputStreamWriter(
                    connection.getOutputStream()),true);
            String inputLine , outputLine;
            KnockKnockProtocol protocol = new KnockKnockProtocol();


            outputLine = protocol.processInput(null);
            out.println(outputLine);

            while ((inputLine = in.readLine()) !=null) {
                outputLine = protocol.processInput(inputLine);
                out.println(outputLine);
                if(outputLine.equals("Bye."))
                    break;
            }
            out.close();
            in.close();
            connection.close();
        }catch(IOException e) {
            System.out.println(e);
            System.exit(1);
        }
    }
}
```

Knock knock client



```java
public class KnockKnockClient {
    public static void main(String[] args) throws IOException {
        Socket kkSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;
        try {
            kkSocket = new Socket(args[0], Integer.parseInt(args[1]));
            out = new PrintWriter(kkSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(kkSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: " + args[0]);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: " + args[0]);
            System.exit(1);
        }

        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
        String fromServer;
        String fromUser;

        while ((fromServer = in.readLine()) != null) {
            System.out.println("Server: " + fromServer);
            if (fromServer.equals("Bye."))
                break;
            fromUser = stdIn.readLine();
            if (fromUser != null) {
                System.out.println("Client: " + fromUser);
                out.println(fromUser);
            }
        }
        out.close();
        in.close();
        stdIn.close();
        kkSocket.close();
    }
}
```

Datagrams

Using UDP in Java

UDP provides a fast, non-reliable communication over the internet.

In UDP:

- Applications send *datagrams* to one another.
- A connection between the applications is not maintained.

A datagram:

- An independent, self-contained message sent over the network.
- The arrival, arrival time and contents are not guaranteed.

Using UDP in Java

The following Java classes all use UDP:

- `DatagramPacket`
- `DatagramSocket`
- `MulticastSocket`

The Date Server example

The following example shows a client-server application that uses UDP.

The Date server:

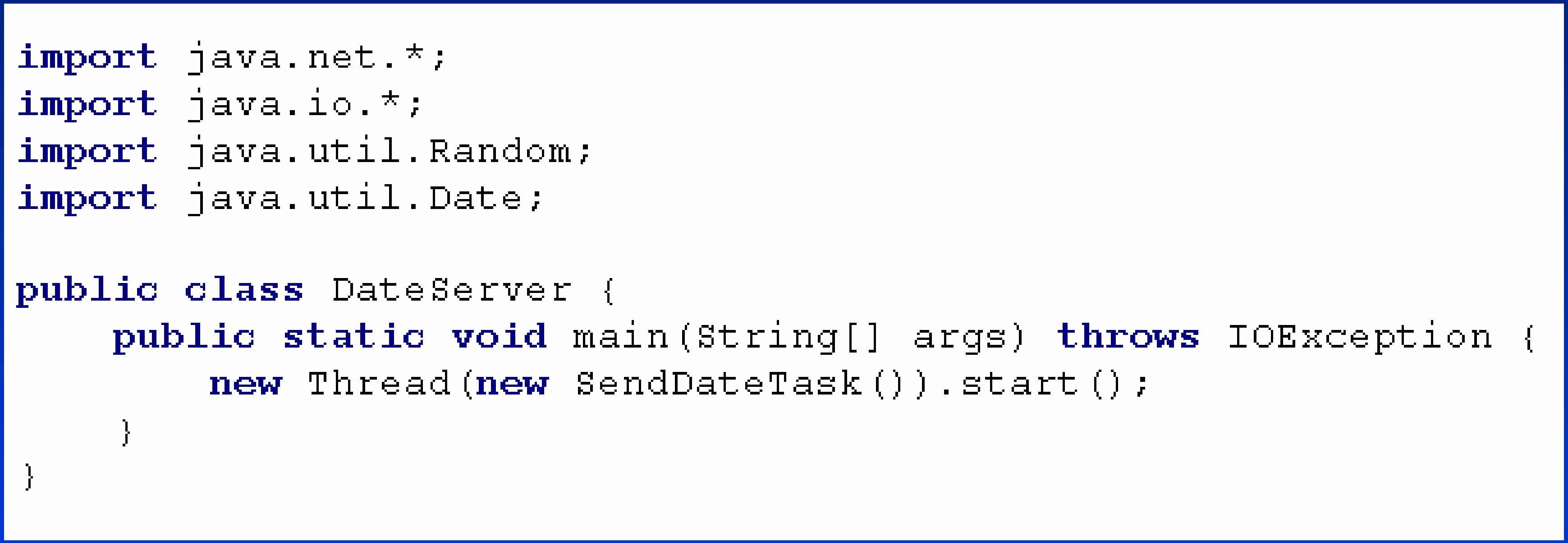
- Continuously receives datagrams over a datagram socket.
- Each received datagram indicates a client request for a the date
- The server replies by sending a datagram that contains the current date.

The Date Server example

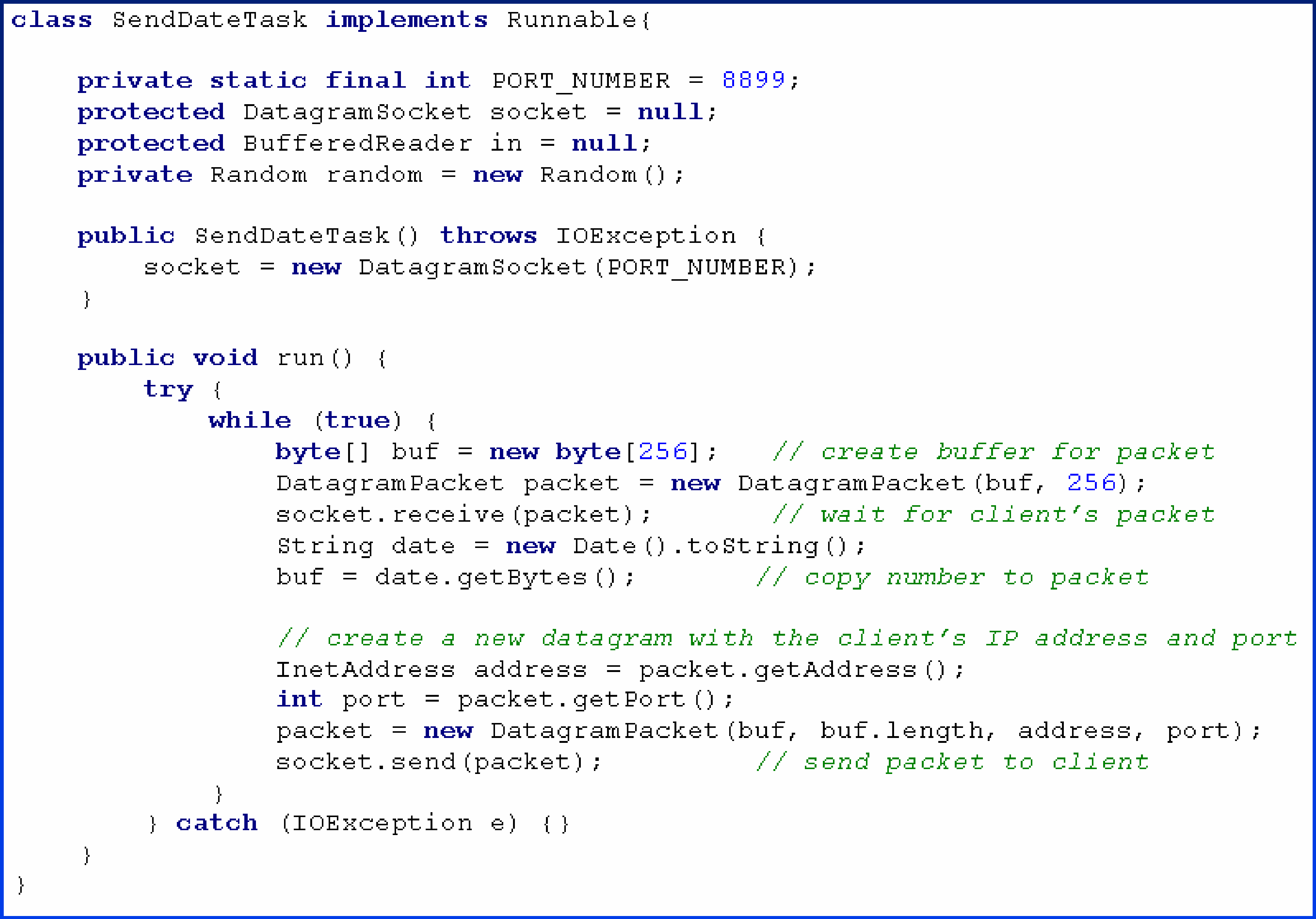
The Date client:

- Sends a single datagram requesting for the date.
- Waits for the server to send a datagram in response.
- Prints the received date.

The Date Server

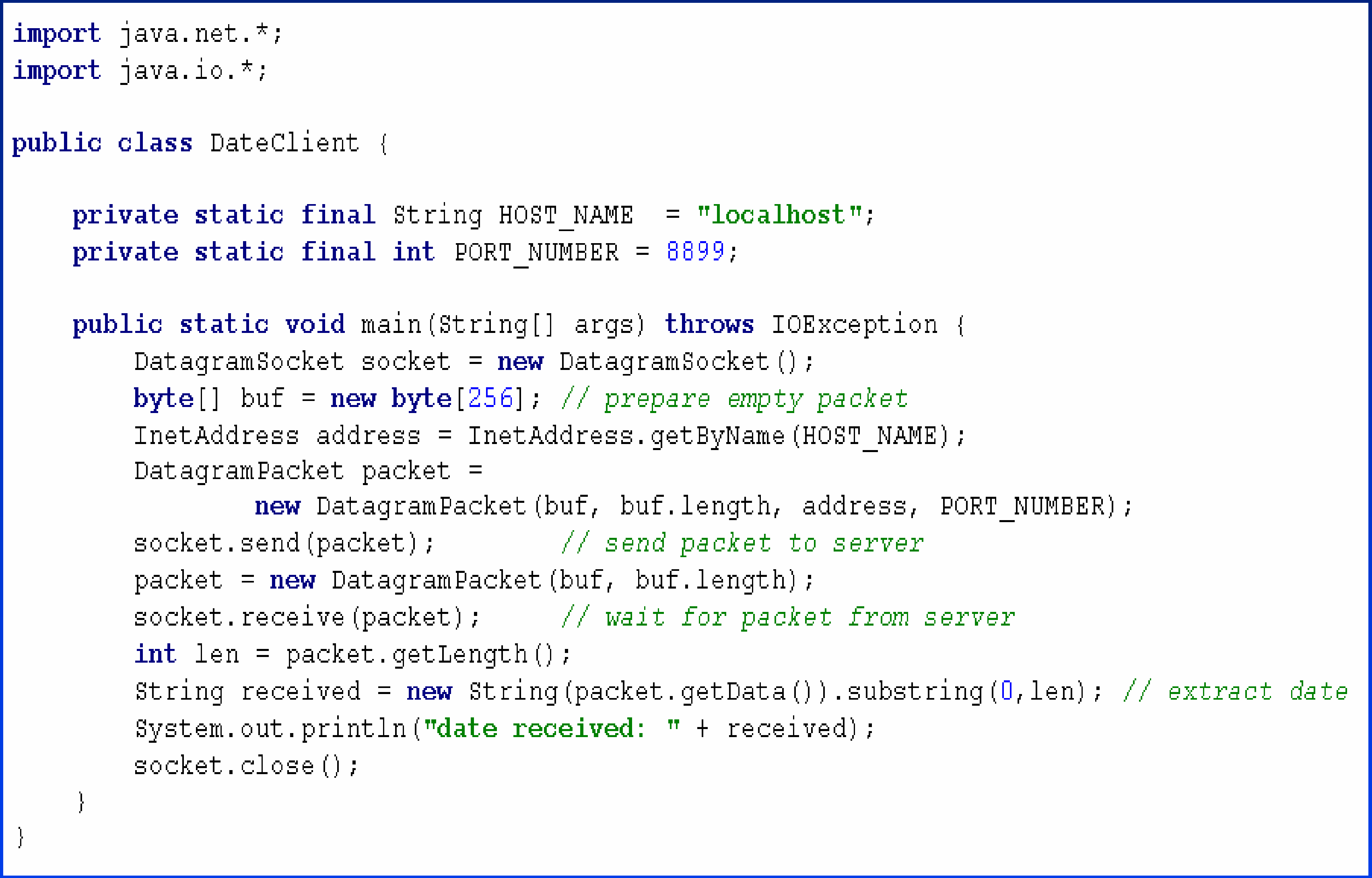


```
import java.net.*;
import java.io.*;
import java.util.Random;
import java.util.Date;

public class DateServer {
    public static void main(String[] args) throws IOException {
        new Thread(new SendDateTask()).start();
    }
}
```



```java
class SendDateTask implements Runnable{

    private static final int PORT_NUMBER = 8899;
    protected DatagramSocket socket = null;
    protected BufferedReader in = null;
    private Random random = new Random();

    public SendDateTask() throws IOException {
        socket = new DatagramSocket(PORT_NUMBER);
    }

    public void run() {
        try {
            while (true) {
                byte[] buf = new byte[256];   // create buffer for packet
                DatagramPacket packet = new DatagramPacket(buf, 256);
                socket.receive(packet);       // wait for client's packet
                String date = new Date().toString();
                buf = date.getBytes();       // copy number to packet

                // create a new datagram with the client's IP address and port
                InetAddress address = packet.getAddress();
                int port = packet.getPort();
                packet = new DatagramPacket(buf, buf.length, address, port);
                socket.send(packet);         // send packet to client
            }
        } catch (IOException e) {}
    }
}
```

The Date client



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

public class DateClient {

    private static final String HOST_NAME  = "localhost";
    private static final int PORT_NUMBER = 8899;

    public static void main(String[] args) throws IOException {
        DatagramSocket socket = new DatagramSocket();
        byte[] buf = new byte[256]; // prepare empty packet
        InetAddress address = InetAddress.getByName(HOST_NAME);
        DatagramPacket packet =
                new DatagramPacket(buf, buf.length, address, PORT_NUMBER);
        socket.send(packet);        // send packet to server
        packet = new DatagramPacket(buf, buf.length);
        socket.receive(packet);     // wait for packet from server
        int len = packet.getLength();
        String received = new String(packet.getData()).substring(0,len); // extract date
        System.out.println("date received: " + received);
        socket.close();
    }
}
```

Lost packets

Since UDP is unreliable, packets can be lost:

- On the way to the server
- On the way back to the client

This means that the client's line socket.receive(packet); **(waiting for the server's response) may wait forever.**

The client should set a timer, and retransmit if no response arrives in reasonable time.

Broadcasting to multiple recipients

Assume that we want the server to broadcast packets to multiple recipients.

Instead of sending dates to a specific client, the server should:

- Send the date to a group of clients.
- Send the dates at a regular time interval.

A group of clients is defined by an InetAddress object, constructed with a reserved IP (e.g. 230.0.0.1).

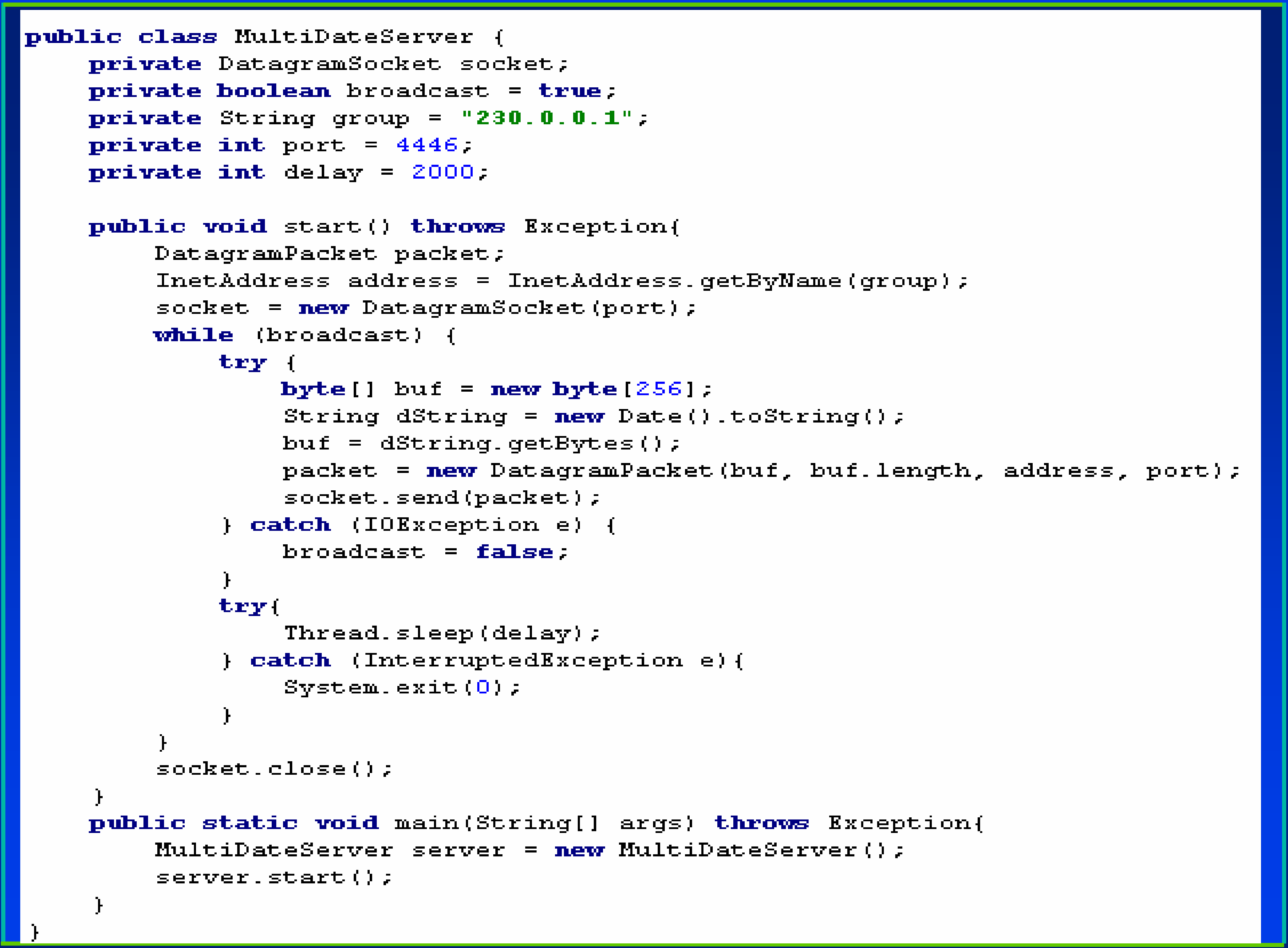
The time interval can be achieved using the Thread.sleep() method.

Broadcasting to multiple recipients

Instead of sending a request, the client should:

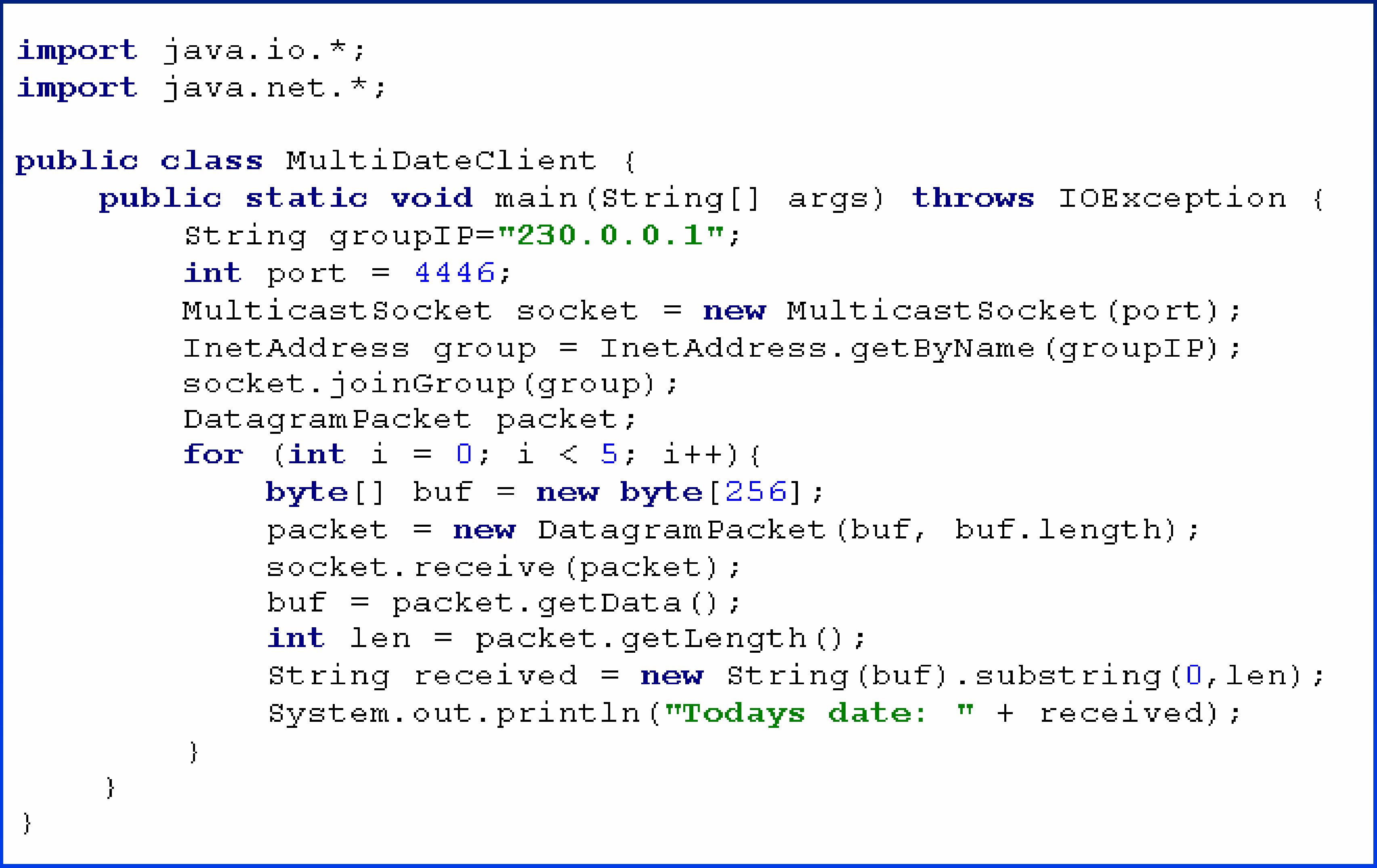
- Not send anything to the server.
- Create a **MulticastSocket** object with the server's port number.
- Register as a group member using the **joinGroup()** method of the **MulticastSocket**.
- Passively listen for the time from the server.

Multicast Server



```
public class MultiDateServer {
    private DatagramSocket socket;
    private boolean broadcast = true;
    private String group = "230.0.0.1";
    private int port = 4446;
    private int delay = 2000;

    public void start() throws Exception{
        DatagramPacket packet;
        InetAddress address = InetAddress.getByName(group);
        socket = new DatagramSocket(port);
        while (broadcast) {
            try {
                byte[] buf = new byte[256];
                String dString = new Date().toString();
                buf = dString.getBytes();
                packet = new DatagramPacket(buf, buf.length, address, port);
                socket.send(packet);
            } catch (IOException e) {
                broadcast = false;
            }
            try{
                Thread.sleep(delay);
            } catch (InterruptedException e){
                System.exit(0);
            }
        }
        socket.close();
    }
    public static void main(String[] args) throws Exception{
        MultiDateServer server = new MultiDateServer();
        server.start();
    }
}
```

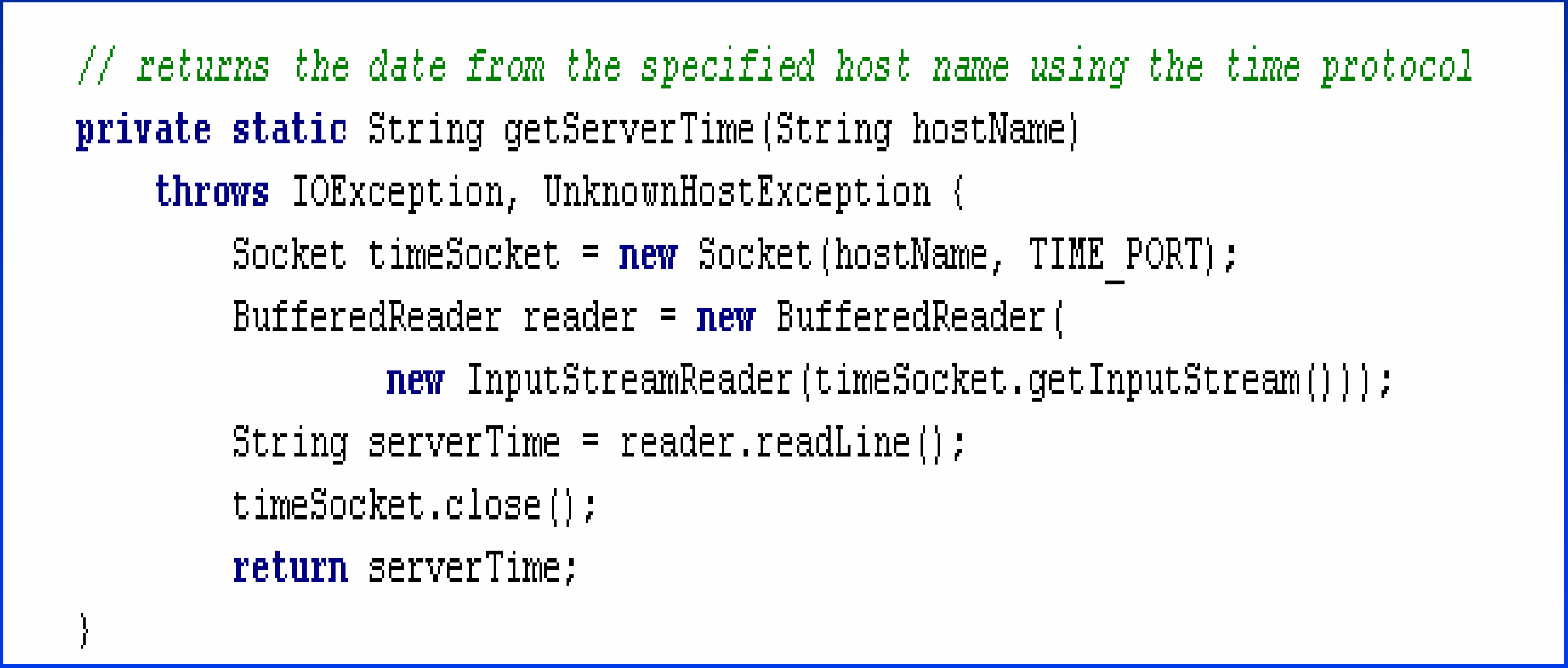
Multicast Client



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

public class MultiDateClient {
    public static void main(String[] args) throws IOException {
        String groupIP="230.0.0.1";
        int port = 4446;
        MulticastSocket socket = new MulticastSocket(port);
        InetAddress group = InetAddress.getByName(groupIP);
        socket.joinGroup(group);
        DatagramPacket packet;
        for (int i = 0; i < 5; i++){
            byte[] buf = new byte[256];
            packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            buf = packet.getData();
            int len = packet.getLength();
            String received = new String(buf).substring(0,len);
            System.out.println("Todays date: " + received);
        }
    }
}
```

Time Server - TCP

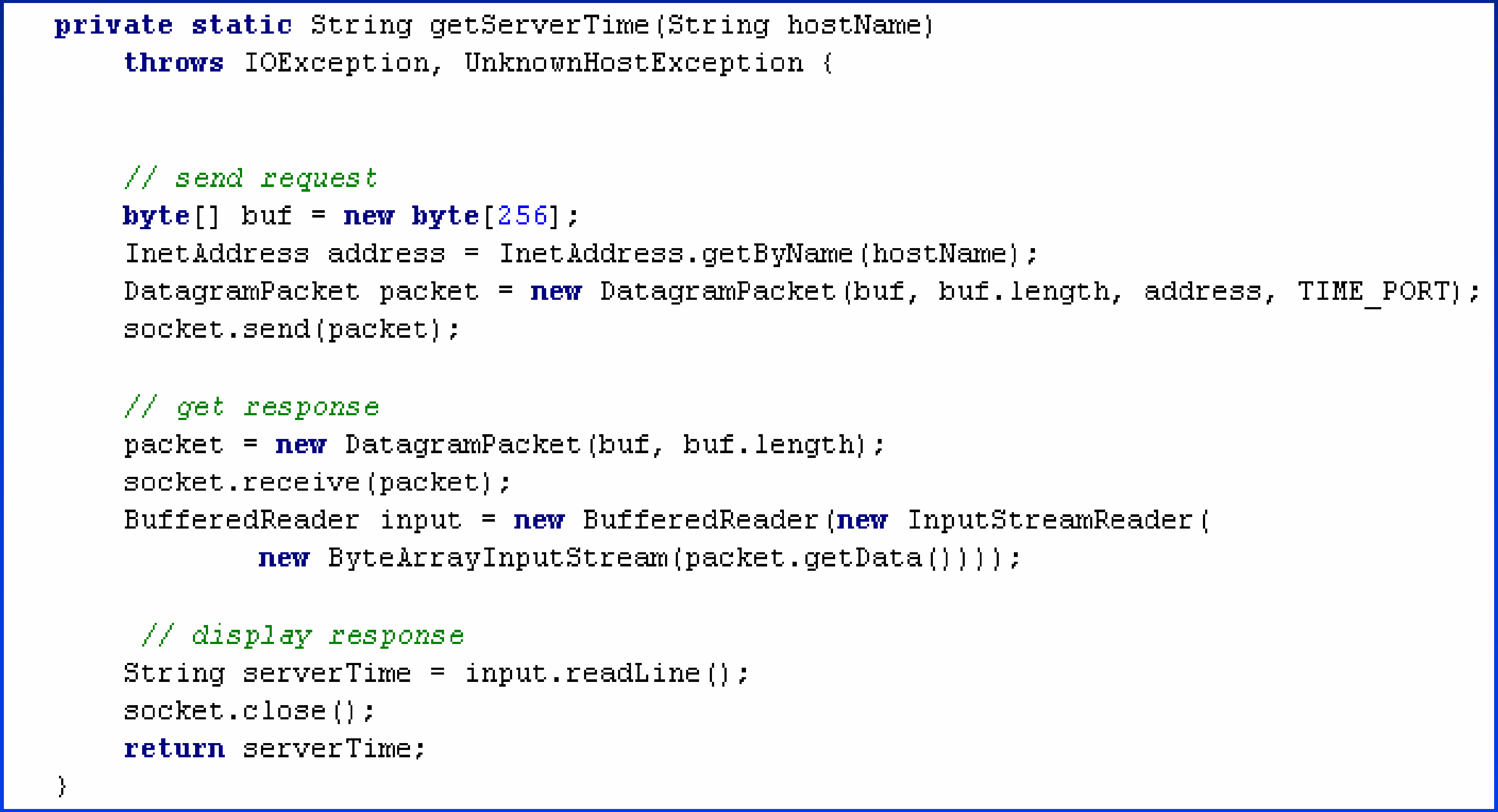
Defined in RFC 867 – listens to port 13



```java
// returns the date from the specified host name using the time protocol
private static String getServerTime(String hostName)
    throws IOException, UnknownHostException {
        Socket timeSocket = new Socket(hostName, TIME_PORT);
        BufferedReader reader = new BufferedReader(
                new InputStreamReader(timeSocket.getInputStream()));
        String serverTime = reader.readLine();
        timeSocket.close();
        return serverTime;
}
```

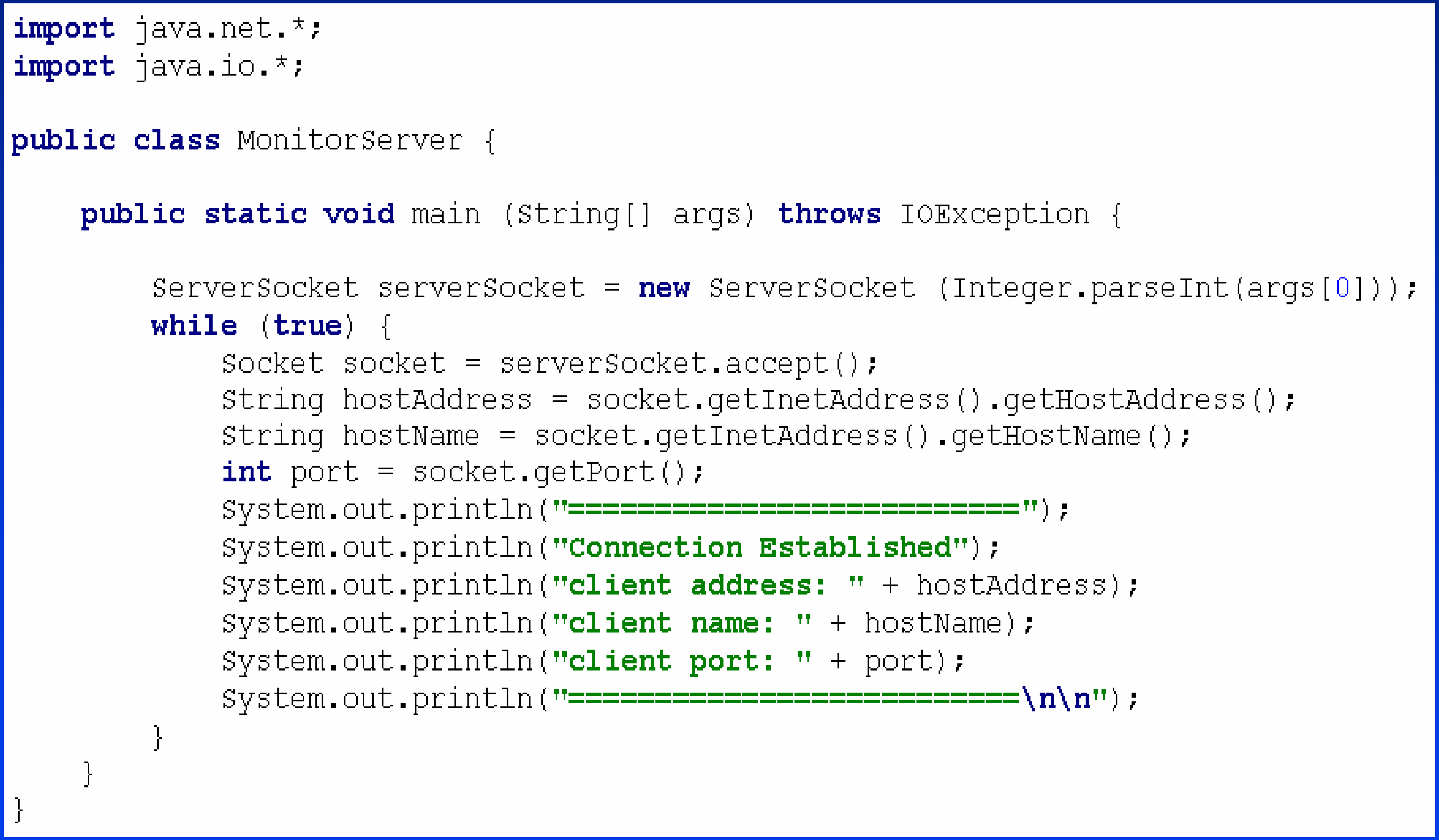
Time Server - UDP

Defined in RFC 867 – listens to port 13



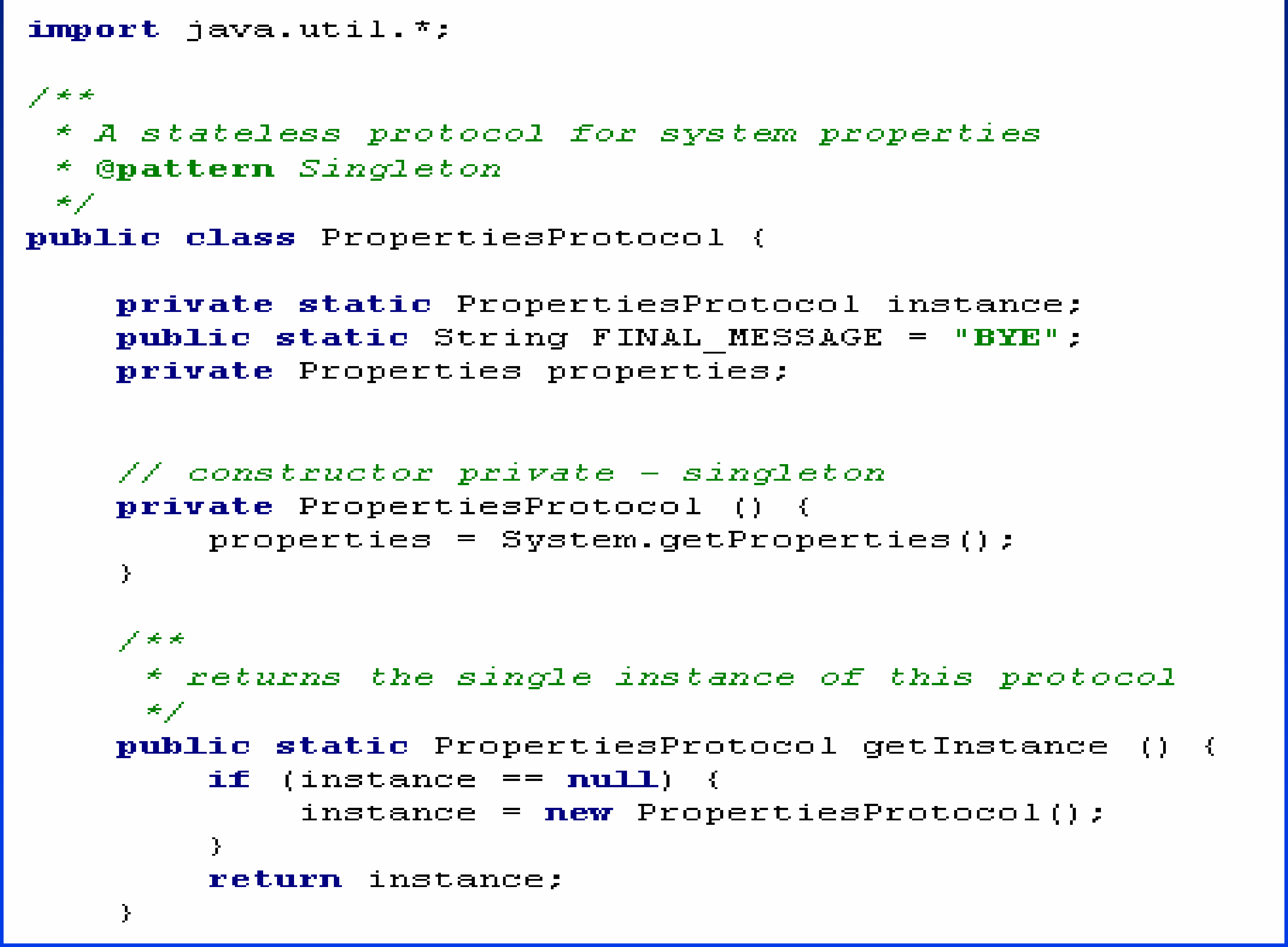
```
private static String getServerTime(String hostName)
    throws IOException, UnknownHostException {


    // send request
    byte[] buf = new byte[256];
    InetAddress address = InetAddress.getByName(hostName);
    DatagramPacket packet = new DatagramPacket(buf, buf.length, address, TIME_PORT);
    socket.send(packet);

    // get response
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);
    BufferedReader input = new BufferedReader(new InputStreamReader(
            new ByteArrayInputStream(packet.getData())));

     // display response
    String serverTime = input.readLine();
    socket.close();
    return serverTime;
}
```

Monitor Server - Prints Connecting Client Information



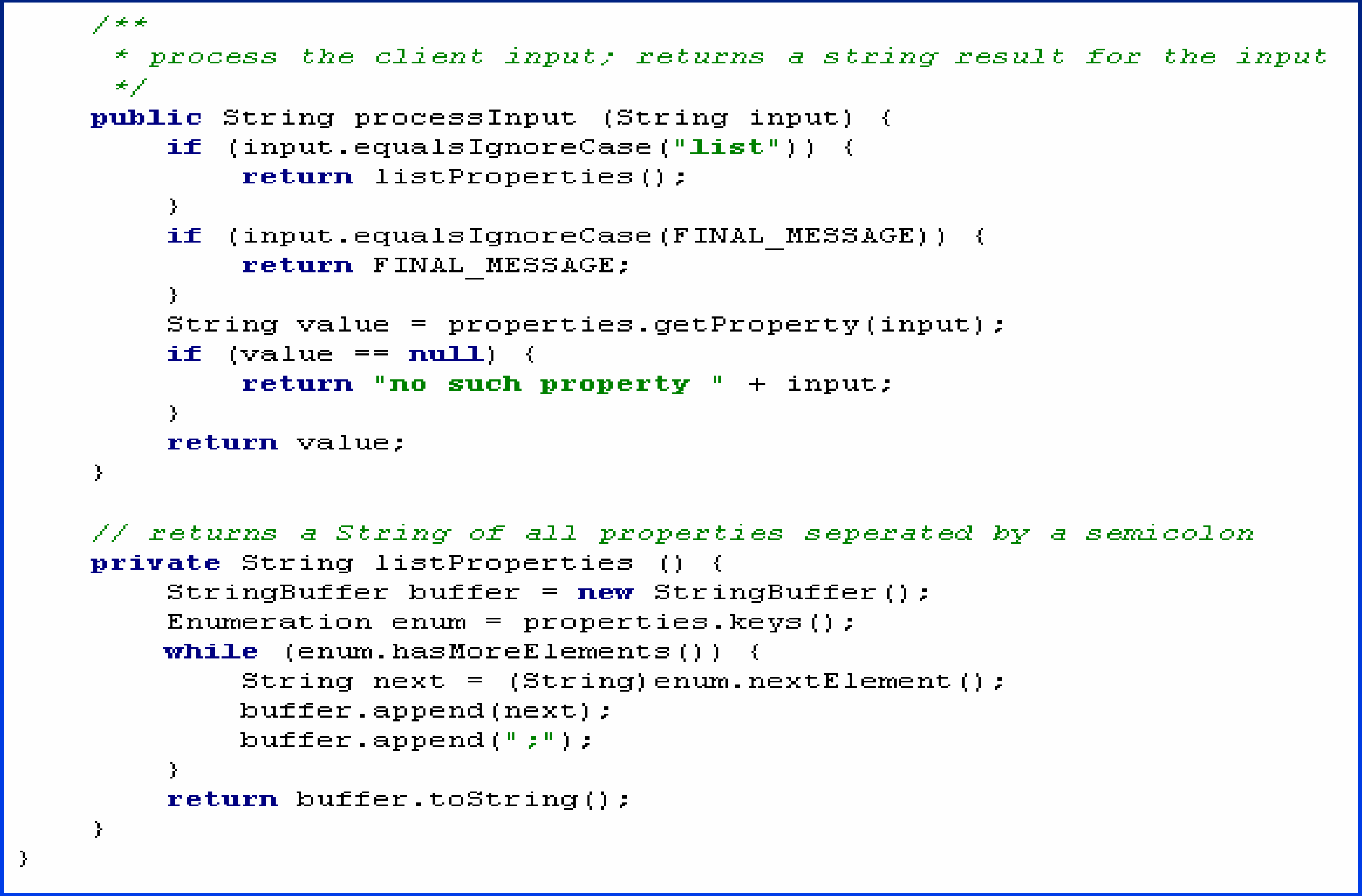
```
import java.net.*;
import java.io.*;

public class MonitorServer {

    public static void main (String[] args) throws IOException {

        ServerSocket serverSocket = new ServerSocket (Integer.parseInt(args[0]));
        while (true) {
            Socket socket = serverSocket.accept();
            String hostAddress = socket.getInetAddress().getHostAddress();
            String hostName = socket.getInetAddress().getHostName();
            int port = socket.getPort();
            System.out.println("==========================");
            System.out.println("Connection Established");
            System.out.println("client address: " + hostAddress);
            System.out.println("client name: " + hostName);
            System.out.println("client port: " + port);
            System.out.println("==========================\n\n");
        }
    }
}
```

Properties Protocol



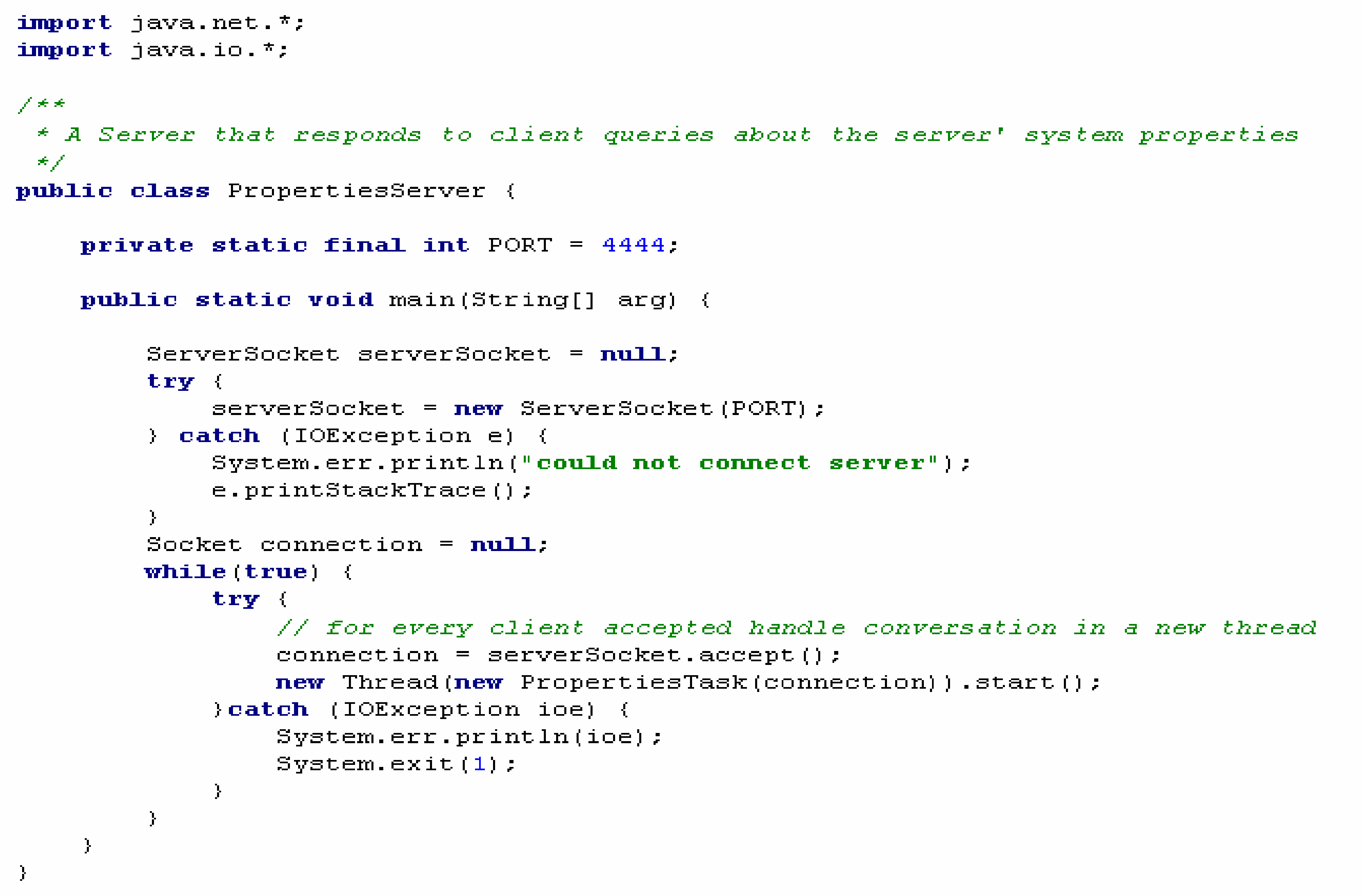
```
import java.util.*;

/**
 * A stateless protocol for system properties
 * @pattern Singleton
 */
public class PropertiesProtocol {

    private static PropertiesProtocol instance;
    public static String FINAL_MESSAGE = "BYE";
    private Properties properties;


    // constructor private - singleton
    private PropertiesProtocol () {
        properties = System.getProperties();
    }

    /**
     * returns the single instance of this protocol
     */
    public static PropertiesProtocol getInstance () {
        if (instance == null) {
            instance = new PropertiesProtocol();
        }
        return instance;
    }
```

Properties Protocol



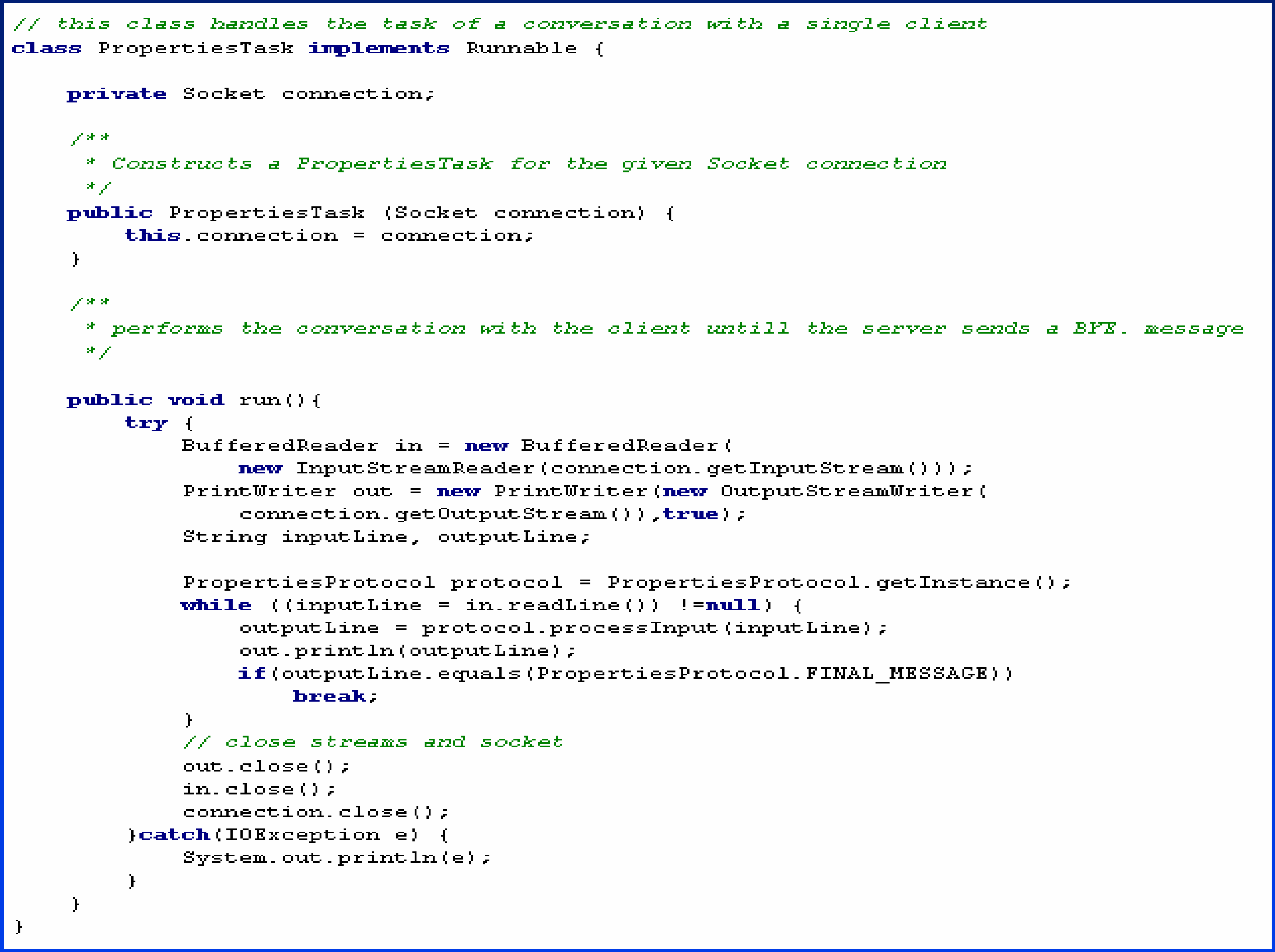
```
    /**
     * process the client input; returns a string result for the input
     */
    public String processInput (String input) {
        if (input.equalsIgnoreCase("list")) {
            return listProperties();
        }
        if (input.equalsIgnoreCase(FINAL_MESSAGE)) {
            return FINAL_MESSAGE;
        }
        String value = properties.getProperty(input);
        if (value == null) {
            return "no such property " + input;
        }
        return value;
    }

    // returns a String of all properties seperated by a semicolon
    private String listProperties () {
        StringBuffer buffer = new StringBuffer();
        Enumeration enum = properties.keys();
        while (enum.hasMoreElements()) {
            String next = (String)enum.nextElement();
            buffer.append(next);
            buffer.append(";");
        }
        return buffer.toString();
    }
}
```

Properties Server



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

/**
 * A Server that responds to client queries about the server' system properties
 */
public class PropertiesServer {

    private static final int PORT = 4444;

    public static void main(String[] arg) {

        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(PORT);
        } catch (IOException e) {
            System.err.println("could not connect server");
            e.printStackTrace();
        }
        Socket connection = null;
        while(true) {
            try {
                // for every client accepted handle conversation in a new thread
                connection = serverSocket.accept();
                new Thread(new PropertiesTask(connection)).start();
            }catch (IOException ioe) {
                System.err.println(ioe);
                System.exit(1);
            }
        }
    }
}
```

Properties Server



```java
// this class handles the task of a conversation with a single client
class PropertiesTask implements Runnable {

    private Socket connection;

    /**
     * Constructs a PropertiesTask for the given Socket connection
     */
    public PropertiesTask (Socket connection) {
        this.connection = connection;
    }

    /**
     * performs the conversation with the client untill the server sends a BYE. message
     */

    public void run(){
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(connection.getInputStream()));
            PrintWriter out = new PrintWriter(new OutputStreamWriter(
                connection.getOutputStream()),true);
            String inputLine, outputLine;

            PropertiesProtocol protocol = PropertiesProtocol.getInstance();
            while ((inputLine = in.readLine()) !=null) {
                outputLine = protocol.processInput(inputLine);
                out.println(outputLine);
                if(outputLine.equals(PropertiesProtocol.FINAL_MESSAGE))
                    break;
            }
            // close streams and socket
            out.close();
            in.close();
            connection.close();
        }catch(IOException e) {
            System.out.println(e);
        }
    }
}
```