**Federal TVET Institute**

**Department of Information Communication Technology**

**Master of ICT Teachers Education**

**Learning Guide 3: Internetworking Programming in Java**

**Information Sheet 3:**

**Basic of Java TCP Networking Programming**

**Socket Programming**

**Dr. Ana Romina A. Miguel (Associate Professor)**

At the end of the chapter the students will be able to:

* Learn the foundation of Networking Programming
* Revisit the OSI Model and the common well-known ports
* Understand the TCP Programming requirements in Java
* Classes and Libraries needed in TCP networking Programming

INTRODUCTION:

 Internet and WWW have emerged as global ubiquitous media for communication and changed the way we conduct science, engineering, and commerce. They are also changing the way we learn, live, enjoy, communicate, interact, engage, etc. The modern life activities are getting completely centered around or driven by the Internet. To take advantage of opportunities presented by the Internet, businesses are continuously seeking new and innovative ways and means for offering their services via the Internet. This created a huge demand for software designers and engineers with skills in creating new Internet-enabled applications or porting existing/legacy applications to the Internet platform. The key elements for developing Internet-enabled applications are a good understanding of the issues involved in implementing distributed applications and sound knowledge of the fundamental network programming models.

 Client/Server Communication At a basic level, network-based systems consist of a server, client , and a media for communication as shown in Fig. 13.1. A computer running a program that makes a request for services is called client machine. A computer running a program that offers requested services from one or more clients is called server machine. The media for communication can be wired or wireless network.



Generally, programs running on client machines make requests to a program (often called as server program) running on a server machine. They involve networking services provided by the transport layer, which is part of the Internet software stack, often called TCP/IP (Transport Control Protocol/Internet Protocol) stack, shown in Fig. 13.2. The transport layer comprises two types of protocols, TCP (Transport Control Protocol) and UDP (User Datagram Protocol). The most widely used programming interfaces for these protocols are sockets.



 TCP is a connection-oriented protocol that provides a reliable fl ow of data between two computers. Example applications that use such services are HTTP, FTP, and Telnet.

 UDP is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival and sequencing. Example applications that use such services include Clock server and Ping.

 The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer. Port is represented by a positive (16-bit) integer value. Some ports have been reserved to support common/well known services:

* ftp 21/tcp
* telnet 23/tcp
* smtp 25/tcp
* login 513/tcp
* http 80/tcp,udp
* https 443/tcp,udp

User-level process/services generally use port number value >= 1024.



Object-oriented Java technologies—Sockets, threads, RMI, clustering, Web services—have emerged as leading solutions for creating portable, efficient, and maintainable large and complex Internet applications.

**Hosts Identification and Service Ports**

 Every computer on the Internet is identified by a unique, 4-byte IP address . This is typically written in dotted quad format like 128.250.25.158 where each byte is an unsigned value between 0 and 255. This representation is clearly not user-friendly because it does not tell us anything about the content and then it is difficult to remember. Hence, IP addresses are mapped to names like www.buyya.com or www.google.com, which are easier to remember. Internet supports name servers that translate these names to IP addresses.

 In general, each computer only has one Internet address. However, computers often need to communicate and provide more than one type of service or to talk to multiple hosts/computers at a time. For example, there may be multiple ftp sessions, web connections, and chat programs all running at the same time. To distinguish these services, a concept of port s, a logical access point, represented by a 16-bit integer number is used. That means, each service offered by a computer is uniquely identified by a port number. Each Internet packet contains both the destination host address and the port number on that host to which the message/request has to be delivered. The host computer dispatches the packets it receives to programs by looking at the port numbers specified within the packets. That is, IP address can be thought of as a house address when a letter is sent via post/snail mail and port number as the name of a specified individual to whom the letter has to be delivered.

Java Networking

Java Networking is a concept of connecting two or more computing devices together so that we can share resources.

Java socket programming provides facility to share data between different computing devices.

Advantage of Java Networking

1. sharing resources
2. centralize software management

Java Networking Terminology

The widely used java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket

1) IP Address

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

It is a logical address that can be changed.

2) Protocol

A protocol is a set of rules basically that is followed for communication. For example:

* TCP
* FTP
* Telnet
* SMTP
* POP etc.

3) Port Number

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

4) MAC Address

MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.

5) Connection-oriented and connection-less protocol

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.

6) Socket

A socket is an endpoint between two way communication.

**Sockets and Socket-based Communication**



Sockets provide an interface for programming networks at the transport layer. Network communication using Sockets is very much similar to performing fi le I/O. In fact, socket handle is treated like file handle. The streams used in file I/O operation are also applicable to socket-based I/O. Socket-based communication is independent of a programming language used for implementing it. That means, a socket program written in Java language can communicate to a program written in non-Java (say C or C++) socket program.

 A server (program) runs on a specific computer and has a socket that is bound to a specific port. The server listens to the socket for a client to make a connection request (see Fig. 13.4a). If everything goes well, the server accepts the connection (see Fig. 13.4b). Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (consequently a different port number) so that it can continue to listen to the original socket for connection requests while serving the connected client.



**SOCKET PROGRAMMING AND JAVA.NET CLASS**

 A socket is an endpoint of a two-way communication link between two programs running on the network. Socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent. Java provides a set of classes, defined in a package called java.net, to enable the rapid development of network applications. Key classes, interfaces, and exceptions in java.net package simplifying the complexity involved in creating client and server programs are:





**TCP/IP SOCKET PROGRAMMING**

 The two key classes from the java.net package used in creation of server and client programs are:

1. ServerSocket
2. Socket

 A server program creates a specific type of socket that is used to listen for client requests (server socket), In the case of a connection request, the program creates a new socket through which it will exchange data with the client using input and output streams. The socket abstraction is very similar to the file concept: developers have to open a socket, perform I/O, and close it. Figure 13.5 illustrates key steps involved in creating socket-based server and client programs.



Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

Important methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public InputStream getInputStream() | returns the InputStream attached with this socket. |
| 2) public OutputStream getOutputStream() | returns the OutputStream attached with this socket. |
| 3) public synchronized void close() | closes this socket |

ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public Socket accept() | returns the socket and establish a connection between server and client. |
| 2) public synchronized void close() | closes the server socket. |

A simple Server Program in Java: The steps for creating a simple server program are:

1. Open the Server Socket:

 ServerSocket server = new ServerSocket( PORT );

2. Wait for the Client Request:

 Socket client = server.accept();

3. Create I/O streams for communicating to the client

 DataInputStream is = new DataInputStream(client.getInputStream()); DataOutputStream os = new DataOutputStream(client.getOutputStream());

4. Perform communication with client

 Receive from client: String line = is.readLine();

 Send to client: os.writeBytes(“Hello\n”);

5. Close socket:

 client.close();

An example program illustrating creation of a server socket, waiting for client request, and then responding to a client that requested for connection by greeting it is given below:

Program 13.1 // SimpleServer.java: A simple server program.

import java.net.\*;

import java.io.\*;

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

// Register service on port 1254

 ServerSocket s = new ServerSocket(1254);

 Socket s1=s.accept(); // Wait and accept a connection

 // Get a communication stream associated with the socket

 OutputStream s1out = s1.getOutputStream();

 DataOutputStream dos = new DataOutputStream (s1out);

 // Send a string!

 dos.writeUTF(“Hi there”);

 // Close the connection, but not the server socket

 dos.close();

 s1out.close();

 s1.close();

 }

 }

**A simple Client Program in Java -** The steps for creating a simple client program are:

1. Create a Socket Object:

 Socket client = new Socket(server, port\_id);

2. Create I/O streams for communicating with the server.

 is = new DataInputStream(client.getInputStream());

 os = new DataOutputStream(client.getOutputStream());

3. Perform I/O or communication with the server:

 Receive data from the server: String line = is.readLine();

 Send data to the server: os.writeBytes(“Hello\n”);

4. Close the socket when done:

 client.close();

 An example program illustrating establishment of connection to a server and then leading a message sent by the server and displaying it on the console is given below:

Program 13.2

 // SimpleClient.java: A simple client program.

import java.net.\*;

import java.io.\*;

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

 // Open your connection to a server, at port 1254

 Socket s1 = new Socket(“localhost”,1254);

 // Get an input file handle from the socket and read the input

 InputStream s1In = s1.getInputStream();

 DataInputStream dis = new DataInputStream(s1In);

 String st = new String (dis.readUTF());

 System.out.println(st);

 // When done, just close the connection and exit

 dis.close();

 s1In.close();

 s1.close();

 }

}

**Running Socket Programs**

 Compile both server and client programs and then deploy server program code on a machine which is going to act as a server and client program, which is going to act as a client. If required, both client and server programs can run on the same machine. To illustrate execution of server and client programs, let us assume that a machine called mundroo.csse.unimelb.edu.au on which we want to run a server program as indicated below:

 [raj@mundroo] java SimpleServer

 The client program can run on any computer in the network (LAN, WAN, or Internet) as long as there is no fi rewall between them that blocks communication. Let us say we want to run our client program on a machine called gridbus.csse.unimelb.edu.au as follows:

 [raj@gridbus] java SimpleClient

 The client program is just establishing a connection with the server and then waits for a message. On receiving a response message, it prints the same to the console. The output in this case is:

 Hi there

which is sent by the server program in response to a client connection request.

 It should be noted that once the server program execution is started, it is not possible for any other server program to run on the same port until the first program which is successful using it is terminated. Port numbers are a mutually exclusive resource. They cannot be shared among different processes at the same time.

Another Example of Java Socket Programming

Let's see a simple of java socket programming in which client sends a text and server receives it.

*File: MyServer.java*

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** MyServer {
4. **public** **static** **void** main(String[] args){
5. **try**{
6. ServerSocket ss=**new** ServerSocket(6666);
7. Socket s=ss.accept();//establishes connection
8. DataInputStream dis=**new** DataInputStream(s.getInputStream());
9. String  str=(String)dis.readUTF();
10. System.out.println("message= "+str);
11. ss.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

*File: MyClient.java*

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** MyClient {
4. **public** **static** **void** main(String[] args) {
5. **try**{
6. Socket s=**new** Socket("localhost",6666);
7. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
8. dout.writeUTF("Hello Server");
9. dout.flush();
10. dout.close();
11. s.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



Another Example of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

*File: MyServer.java*

1. **import** java.net.\*;
2. **import** java.io.\*;
3. **class** MyServer{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. ServerSocket ss=**new** ServerSocket(3333);
6. Socket s=ss.accept();
7. DataInputStream din=**new** DataInputStream(s.getInputStream());
8. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
9. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
10.
11. String str="",str2="";
12. **while**(!str.equals("stop")){
13. str=din.readUTF();
14. System.out.println("client says: "+str);
15. str2=br.readLine();
16. dout.writeUTF(str2);
17. dout.flush();
18. }
19. din.close();
20. s.close();
21. ss.close();
22. }
23. }

*File: MyClient.java*

1. **import** java.net.\*;
2. **import** java.io.\*;
3. **class** MyClient{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. Socket s=**new** Socket("localhost",3333);
6. DataInputStream din=**new** DataInputStream(s.getInputStream());
7. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
8. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
9.
10. String str="",str2="";
11. **while**(!str.equals("stop")){
12. str=br.readLine();
13. dout.writeUTF(str);
14. dout.flush();
15. str2=din.readUTF();
16. System.out.println("Server says: "+str2);
17. }
18.
19. dout.close();
20. s.close();
21. }
22. }

Java InetAddress class

**Java InetAddress** class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name *for example* www.javatpoint.com, www.google.com, www.facebook.com etc.

Commonly used methods of InetAddress class

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static InetAddress getByName(String host) throws UnknownHostException | it returns the instance of InetAddress containing LocalHost IP and name. |
| public static InetAddress getLocalHost() throws UnknownHostException | it returns the instance of InetAdddress containing local host name and address. |
| public String getHostName() | it returns the host name of the IP address. |
| public String getHostAddress() | it returns the IP address in string format. |

Example of Java InetAddress class

Let's see a simple example of InetAddress class to get ip address of www.javatpoint.com website.

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public** **class** InetDemo{
4. **public** **static** **void** main(String[] args){
5. **try**{
6. InetAddress ip=InetAddress.getByName("www.javatpoint.com");
7.
8. System.out.println("Host Name: "+ip.getHostName());
9. System.out.println("IP Address: "+ip.getHostAddress());
10. }**catch**(Exception e){System.out.println(e);}
11. }
12. }

Output:

Host Name: www.javatpoint.com

IP Address: 206.51.231.148

**UDP SOCKET PROGRAMMING**

 The previous two example programs used the TCP sockets. As already said, TCP guarantees the delivery of packets and preserves their order on destination. Sometimes these features are not required and since they do not come without performance costs, it would be better to use a lighter transport protocol. This kind of service is accomplished by the UDP protocol which conveys datagram packets.

 Datagram packets are used to implement a connectionless packet delivery service supported by the UDP protocol. Each message is transferred from source machine to destination based on information contained within that packet. That means, each packet needs to have destination address and each packet might be routed differently, and might arrive in any order. Packet delivery is not guaranteed.

 The format of datagram packet is:



Java DatagramSocket and DatagramPacket

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

Java supports datagram communication through the following classes:

1. DatagramPacket

Java DatagramPacket class

**Java DatagramPacket** is a message that can be sent or received. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

Commonly used Constructors of DatagramPacket class

* **DatagramPacket(byte[] barr, int length):**it creates a datagram packet. This constructor is used to receive the packets.
* **DatagramPacket(byte[] barr, int length, InetAddress address, int port):**it creates a datagram packet. This constructor is used to send the packets.

 The class DatagramPacket contains several constructors that can be used for creating packet object. One of them is:

 DatagramPacket(byte[ ] buf, int length, InetAddress address, int port);

 This constructor is used for creating a datagram packet for sending packets of length length to the specifi ed port number on the specified host. The message to be transmitted is indicated in the fi rst argument. The key methods of DatagramPacket class are:

 byte[] getData() - Returns the data buffer.

 int getLength() - Returns the length of the data to be sent or the length of the data received.

 void setData(byte[ ] buf) - Sets the data buffer for this packet.

 void setLength(int length) - Sets the length for this packet.

2. DatagramSocket

Java DatagramSocket class

**Java DatagramSocket** class represents a connection-less socket for sending and receiving datagram packets.

A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.

Commonly used Constructors of DatagramSocket class

* **DatagramSocket() throws SocketEeption:**it creates a datagram socket and binds it with the available Port Number on the localhost machine.
* **DatagramSocket(int port) throws SocketEeption:**it creates a datagram socket and binds it with the given Port Number.
* **DatagramSocket(int port, InetAddress address) throws SocketEeption:**it creates a datagram socket and binds it with the specified port number and host address.

 The class DatagramSocket supports various methods that can be used for transmitting or receiving data a datagram over the network. The two key methods are:

 void send(DatagramPacket p) - Sends a datagram packet from this socket.

 void receive(DatagramPacket p) - Receives a datagram packet from this socket.

 A simple UDP server program that waits for client’s requests and then accepts the message (datagram) and sends back the same message is given below. Of course, an extended server program can manipulate client’s messages/request and send a new message as a response.

// UDPServer.java: A simple UDP server program.

import java.net.\*;

import java.io.\*;

 **public** **class** UDPServer {

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

 DatagramSocket aSocket = **null**;

 **if** (args.length < 1) { System.out.println(“Usage:

 java UDPServer ”); System.exit(1);

 } **try** { **int** socket\_no = Integer.valueOf(args[0]).intValue();

 aSocket = **new** DatagramSocket(socket\_no);

 **byte**[] buffer = **new** **byte**[1000];

 **while**(**true**) { DatagramPacket request = **new** DatagramPacket(buffer,

 buffer.length);

 aSocket.receive(request);

 DatagramPacket reply = **new** DatagramPacket(request.getData(), request.getLength(),request.getAddress(), request.getPort());

 aSocket.send(reply);

 }

 } **catch** (SocketException e) { System.out.println(“Socket: ” + e.getMessage());

 } **catch** (IOException e) { System.out.println(“IO: ” + e.getMessage());

 } **finally** { **if** (aSocket != **null**) aSocket.close();

 }

 }

}

 A corresponding client program for creating a datagram and then sending it to the above server and then accepting a response is listed below.

// UDPClient.java: A simple UDP client program.

**import** java.net.\*;

**import** java.io.\*;

**public** **class** UDPClient {

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

 // args give message contents and server hostname

 DatagramSocket aSocket = **null**;

 **if** (args.length < 3) {

 System.***out***.println( "Usage: java UDPClient <message> <Host name> <Port number>");

 System.*exit*(1);

 **try** {

 aSocket = **new** DatagramSocket();

 **byte** [] m = args[0].getBytes();

 InetAddress aHost = InetAddress.*getByName*(args[1]);

 **int** serverPort = Integer.*valueOf*(args[2]).intValue();

 DatagramPacket request = **new** DatagramPacket(m, args[0].length(),

 aHost, serverPort);

 aSocket.send(request);

 **byte**[] buffer = **new** **byte**[1000];

 DatagramPacket reply = **new** DatagramPacket(buffer, buffer.length);

 aSocket.receive(reply);

 System.***out***.println("Reply: " + **new** String(reply.getData()));

 }

 **catch** (SocketException e) {

 System.***out***.println("Socket: " + e.getMessage());

 }

 **catch** (IOException e) {

 System.***out***.println("IO:" + e.getMessage());

 }

 **finally** {

 **if** (aSocket != **null**)

 aSocket.close();

 }

 }

 }

}

Another Example of Sending DatagramPacket by DatagramSocket

1. //DSender.java
2. **import** java.net.\*;
3. **public** **class** DSender{
4. **public** **static** **void** main(String[] args) **throws** Exception {
5. DatagramSocket ds = **new** DatagramSocket();
6. String str = "Welcome java";
7. InetAddress ip = InetAddress.getByName("127.0.0.1");
8.
9. DatagramPacket dp = **new** DatagramPacket(str.getBytes(), str.length(), ip, 3000);
10. ds.send(dp);
11. ds.close();
12. }
13. }

Another Example of Receiving DatagramPacket by DatagramSocket

1. //DReceiver.java
2. **import** java.net.\*;
3. **public** **class** DReceiver{
4. **public** **static** **void** main(String[] args) **throws** Exception {
5. DatagramSocket ds = **new** DatagramSocket(3000);
6. **byte**[] buf = **new** **byte**[1024];
7. DatagramPacket dp = **new** DatagramPacket(buf, 1024);
8. ds.receive(dp);
9. String str = **new** String(dp.getData(), 0, dp.getLength());
10. System.out.println(str);
11. ds.close();
12. }
13. }

CONCLUSION

**TCP/IP and UDP/IP communications**

There are two communication protocols that one can use for socket programming: datagram communication and stream communication.

**Datagram communication:**

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, additional data must be sent each time a communication is made.

**Stream communication:**

The stream communication protocol is known as TCP (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

Now, you might ask what protocol you should use -- UDP or TCP? This depends on the client/server application you are writing. The following discussion shows the differences between the UDP and TCP protocols; this might help you decide which protocol you should use.

In UDP, as you have read above, every time you send a datagram, you have to send the local descriptor and the socket address of the receiving socket along with it. Since TCP is a connection-oriented protocol, on the other hand, a connection must be established before communications between the pair of sockets start. So there is a connection setup time in TCP.

In UDP, there is a size limit of 64 kilobytes on datagrams you can send to a specified location, while in TCP there is no limit. Once a connection is established, the pair of sockets behaves like streams: All available data are read immediately in the same order in which they are received.

UDP is an unreliable protocol -- there is no guarantee that the datagrams you have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you send will be received in the order in which they were sent.

In short, TCP is useful for implementing network services -- such as remote login (rlogin, telnet) and file transfer (FTP) -- which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. It is often used in implementing client/server applications in distributed systems built over local area networks.

**Programming sockets in Java**

*How do I open a socket?*

If you are programming a client, then you would open a socket like this:

 **Socket** **MyClient**;

 **MyClient** = **new** **Socket**("Machine name", **PortNumber**);

Where Machine name is the machine you are trying to open a connection to, and PortNumber is the port (a number) on which the server you are trying to connect to is running. When selecting a port number, you should note that port numbers between 0 and 1,023 are reserved for privileged users (that is, super user or root). These port numbers are reserved for standard services, such as email, FTP, and HTTP. When selecting a port number for your server, select one that is greater than 1,023!

In the example above, we didn't make use of exception handling, however, it is a good idea to handle exceptions. The above can be written as:

 **Socket** **MyClient**;

 **try** {

 **MyClient** = **new** **Socket**("Machine name", **PortNumber**);

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

If you are programming a server, then this is how you open a socket:

 **ServerSocket** **MyService**;

 **try** {

 **MyServerice** = **new** **ServerSocket**(**PortNumber**);

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

When implementing a server you also need to create a socket object from the ServerSocket in order to listen for and accept connections from clients.

 **Socket** clientSocket = **null**;

 **try** {

 serviceSocket = **MyService**.accept();

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

*How do I create an input stream?*

On the client side, you can use the DataInputStream class to create an input stream to receive response from the server:

 **DataInputStream** input;

 **try** {

 input = **new** **DataInputStream**(**MyClient**.getInputStream());

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

The class DataInputStream allows you to read lines of text and Java primitive data types in a portable way. It has methods such as read, readChar, readInt, readDouble, and readLine,. Use whichever function you think suits your needs depending on the type of data that you receive from the server.

On the server side, you can use DataInputStream to receive input from the client:

 **DataInputStream** input;

 **try** {

 input = **new** **DataInputStream**(serviceSocket.getInputStream());

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

*How do I create an output stream?*

On the client side, you can create an output stream to send information to the server socket using the class PrintStream or DataOutputStream of java.io:

 **PrintStream** output;

 **try** {

 output = **new** **PrintStream**(**MyClient**.getOutputStream());

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

The class PrintStream has methods for displaying textual representation of Java primitive data types. Its Write and println methods are important here. Also, you may want to use the DataOutputStream:

 **DataOutputStream** output;

 **try** {

 output = **new** **DataOutputStream**(**MyClient**.getOutputStream());

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

The class DataOutputStream allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method writeBytes is a useful one.

On the server side, you can use the class PrintStream to send information to the client.

 **PrintStream** output;

 **try** {

 output = **new** **PrintStream**(serviceSocket.getOutputStream());

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

*Note: You can use the class DataOutputStream as mentioned above.*

*How do I close sockets?*

You should always close the output and input stream before you close the socket.

On the client side:

 **try** {

 output.close();

 input.close();

 **MyClient**.close();

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

On the server side:

 **try** {

 output.close();

 input.close();

 serviceSocket.close();

 **MyService**.close();

 }

 **catch** (**IOException** e) {

 **System**.**out**.println(e);

 }

**Socket programming with Java: TCP and UDP**

Socket programming boils down to two systems communicating with one another. Network communication comes in two flavors: Transport Control Protocol (TCP) and User Datagram Protocol (UDP). TCP and UDP are used for different purposes and both have unique constraints:

* TCP is relatively simple and reliable protocol that enables a client to make a connection to a server and the two systems to communicate. In TCP, each entity knows that its communication payloads have been received.
* UDP is a *connectionless protocol* and is good for scenarios where you do not necessarily need every packet to arrive at its destination, such as media streaming.

To appreciate the difference between TCP and UDP, consider what would happen if you were streaming video from your favorite website and it dropped frames. Would you prefer that the client slow down your movie to receive the missing frames or would you prefer that the video continue playing? Video streaming protocols typically leverage UDP. Because TCP guarantees delivery, it is the protocol of choice for HTTP, FTP, SMTP, POP3, and so forth.

Java TCP client socket code is handled by the java.net.Socket class. The following code opens a connection to a server:

 **Socket** socket = **new** **Socket**( server, port );

Once our socket instance is connected to the server we can start obtaining input and output streams to the sever. Input streams are used to read data from the server while output streams are used to write data to the server. We can execute the following methods to obtain input and output streams:

 **InputStream** **in** = socket.getInputStream();

 **OutputStream** **out** = socket.getOutputStream();

Because these are ordinary streams, the same streams that we would use to read from and write to a file, we can convert them to the form that best serves our use case. For example, we could wrap the OutputStream with a PrintStream so that we can easily write text with methods like println(). For another example, we could wrap the InputStream with a BufferedReader, via an InputStreamReader, in order to easily read text with methods like readLine().