Network Programming in Java

Prof. Indira R Umarji

Assistant Professor, Department of CSE, SDMCET, Dharwad

What is Networking?

- ***Network programming*** refers to writing programs that execute across multiple devices (computers), in which the devices are connected to each other via a network.
- ***Java Networking*** is a notion of connecting two or more computing devices to share resources.
- Java program communicates over the application layer
- The package available for all networking classes and interfaces is ***java.net***

Properties of today's networking

- Today's networking environment concentrates on two main properties:
  - **Distributed nature:** computations occur on different network hosts
  - **Heterogeneous behaviour:** the hosts can operate on different operating systems.

Simple Networking terminologies

- IP address: a dotted quad unique for each network
- Protocol: set of rules to be followed for communication
- Port number: uniquely identifies various applications; associated with IP address
- MAC address: uniquehardware identification number
- Socket: end point of a 2-way communication; it is bound to aport number
- Inet address: encapsulate both numerical IP address and domain name for that address

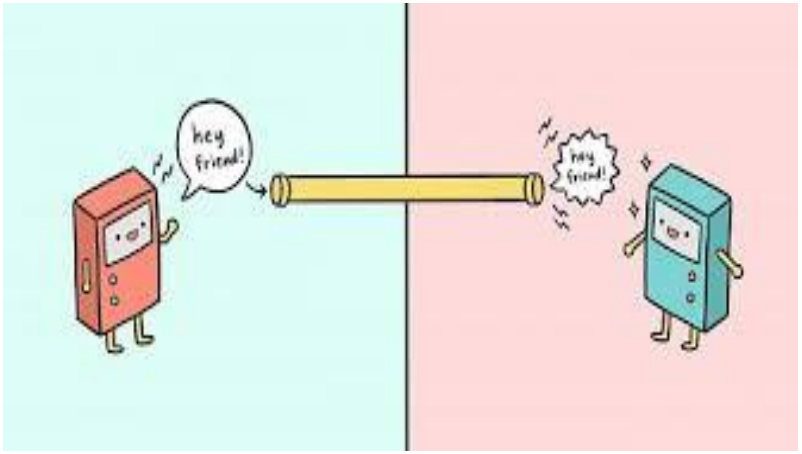
Types of communications in Java

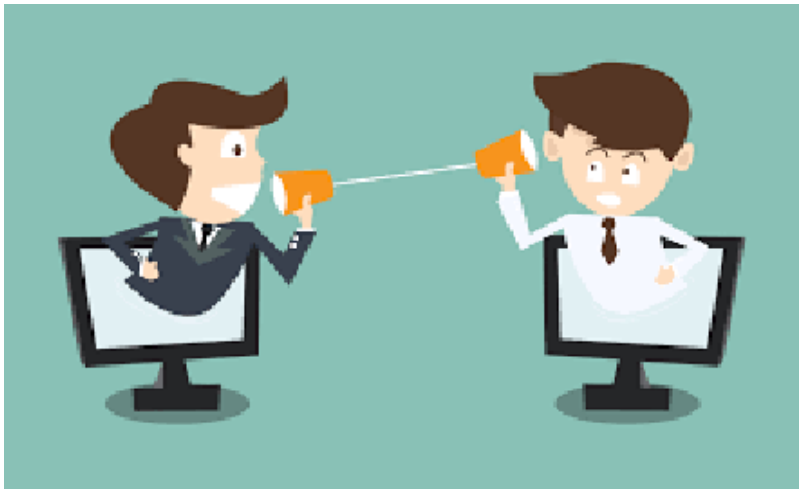
Java provides two types of distributed communications:

1. Socket-based (***java.net*** package): provides 2 types of communications
    a. TCP (Transmission Control Protocol) – Connection-oriented
    b. UDP (User Datagram Protocol) – Connectionless
2. Remote Method Invocation – RMI (***java.rmi*** package) ***Shared in 'Remote Method Invocation (RMI) in Java.pptx'***

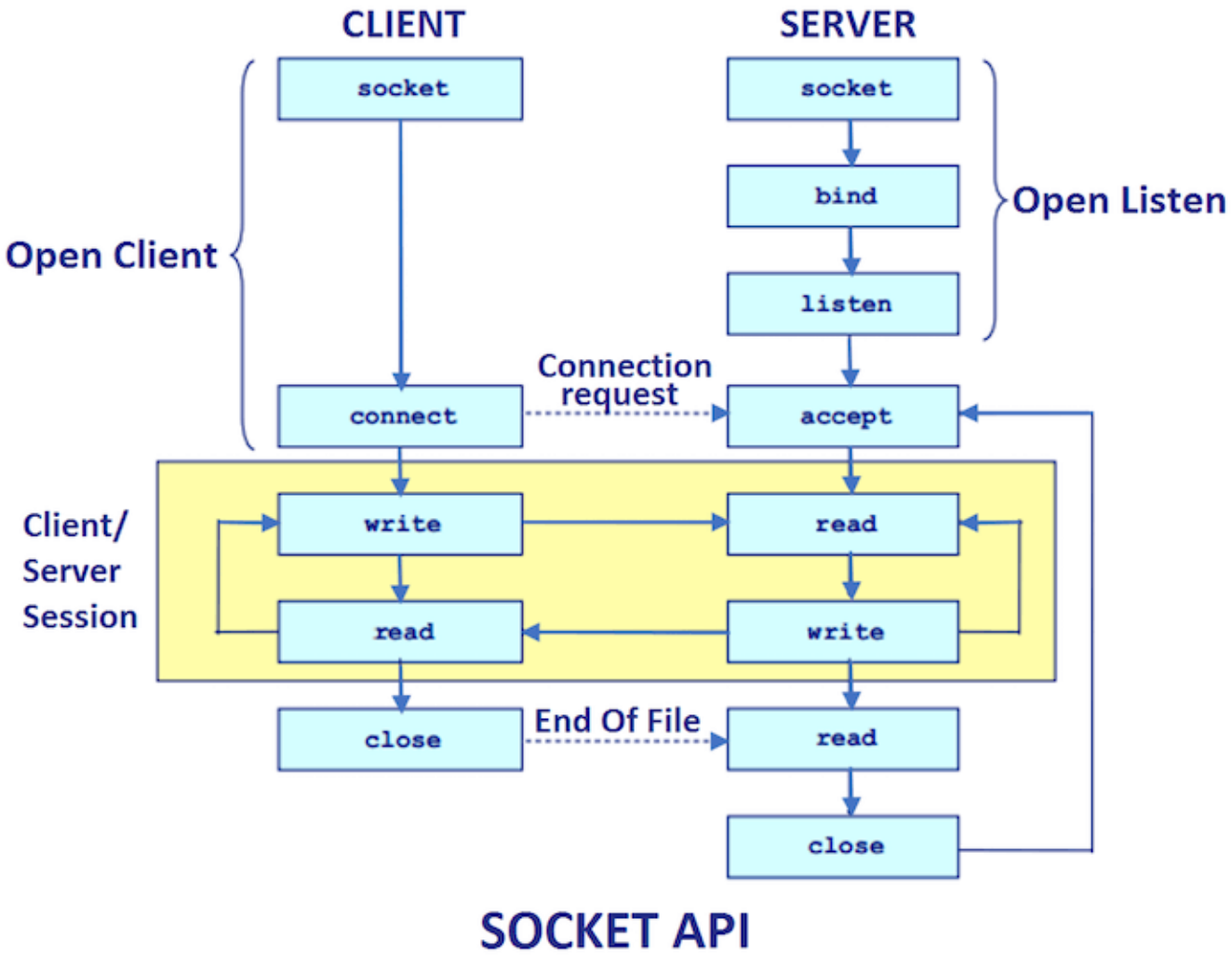
Socket-based communication

- Sockets are the end points of connections between two hosts and can be used to send and receive data.
- There are two kinds of sockets: *server sockets* and *client sockets*.
  - ✓ A server socket waits for requests from clients.
  - ✓ A client socket can be used to send and receive data.
- A server socket listens at a specific port. A port is positive integer less than or equal to 65565. The port number is necessary to distinguish different server applications running on the same host.





Typical Socket API



Important Java classes in ***java.net***

- ***Socket*** and ***ServerSocket*** classes are used for connection-oriented (TCP) programming.
- ***DatagramSocket*** and ***DatagramPacket*** classes are used for connection-less (UDP) socket programming.

Socket programming using TCP – Steps involved

- The following steps occur when establishing a TCP connection between two computers using sockets −
  - The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
  - The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
  - After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.
  - The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
  - On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

TCP Socket programming classes

- ‘ServerSocket’ class:
  - The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.
  - The different constructors of the class are mentioned in the next slide in **Table 1**.

1	**public ServerSocket(int port) throws IOException** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	**public ServerSocket(int port, int backlog) throws IOException** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.
3	**public ServerSocket(int port, int backlog, InetAddress address) throws IOException** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on.
4	**public ServerSocket() throws IOException** Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket.

Table 1

Note: If the *ServerSocket* constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

– ServerSocket's methods are:

1	**public int getLocalPort()** Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2	**public Socket accept() throws IOException** Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely.
3	**public void setSoTimeout(int timeout)** Sets the time-out value for how long the server socket waits for a client during the accept().
4	**public void bind(SocketAddress host, int backlog)** Binds the socket to the specified server and port in the SocketAddress object. Use this method if you have instantiated the ServerSocket using the no-argument constructor.

Table 2

Note: When the *ServerSocket* invokes accept(), the method does not return until a client connects. After a client does connect, the *ServerSocket* creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and the server, and communication can begin.

TCP Socket programming classes continued…

- ‘Socket’ class:
  - This class can be used to create a socket.
  - The important constructors of this class are given in the next slide in **Table 3**.

1	**public Socket(String host, int port) throws UnknownHostException, IOException.** This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2	**public Socket(InetAddress host, int port) throws IOException** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.
3	**public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.** Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String.
5	**public Socket()** Creates an unconnected socket. Use the connect() method to connect this socket to a server.

Table 3

Note: When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

– Some methods of interest in the Socket class are listed here. Notice that both the client and the server have a Socket object, so the methods can be invoked by both the client and the server and are listed in **Table 4**

1	**public void connect(SocketAddress host, int timeout) throws IOException** This method connects the socket to the specified host. This method is needed only when you instantiate the Socket using the no-argument constructor.
2	**public InetAddress getInetAddress()** This method returns the address of the other computer that this socket is connected to.
3	**public int getPort()** Returns the port the socket is bound to on the remote machine.
4	**public int getLocalPort()** Returns the port the socket is bound to on the local machine.
5	**public SocketAddress getRemoteSocketAddress()** Returns the address of the remote socket.
6	**public InputStream getInputStream() throws IOException** Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7	**public OutputStream getOutputStream() throws IOException** Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket.
8	**public void close() throws IOException** Closes the socket, which makes this Socket object no longer capable of connecting again to any server.

Table 4

Special class: InetAddress

- This class represents an Internet Protocol (IP) address.
- The different methods useful from this class in Socket programming are listed in **Table 5**.

1	**static InetAddress getByAddress(byte[] addr)** Returns an InetAddress object given the raw IP address.
2	**static InetAddress getByAddress(String host, byte[] addr)** Creates an InetAddress based on the provided host name and IP address.
3	**static InetAddress getByName(String host)** Determines the IP address of a host, given the host's name.
4	**String getHostAddress()** Returns the IP address string in textual presentation.
5	**String getHostName()** Gets the host name for this IP address.
6	**static InetAddress InetAddress getLocalHost()** Returns the local host.
7	**String toString()** Converts this IP address to a String.

Table 5

TCP Socket program - Example

```
//MyServer.java
import java.io.*;
import java.net.*;
public class MyServer {
  public static void main(String[] args){
    try{
            ServerSocket ss=new ServerSocket(6666);
            Socket s=ss.accept();                          //establishes connection

            DataInputStream dis=new DataInputStream(s.getInputStream());
            System.out.println("This is Server...");
            String  str=(String)dis.readUTF();             //read the message from client
            System.out.println("message= "+str);           //printing message received

            ss.close();                                    //close the socket
    }catch(Exception e){System.out.println(e);}
  }
}
```

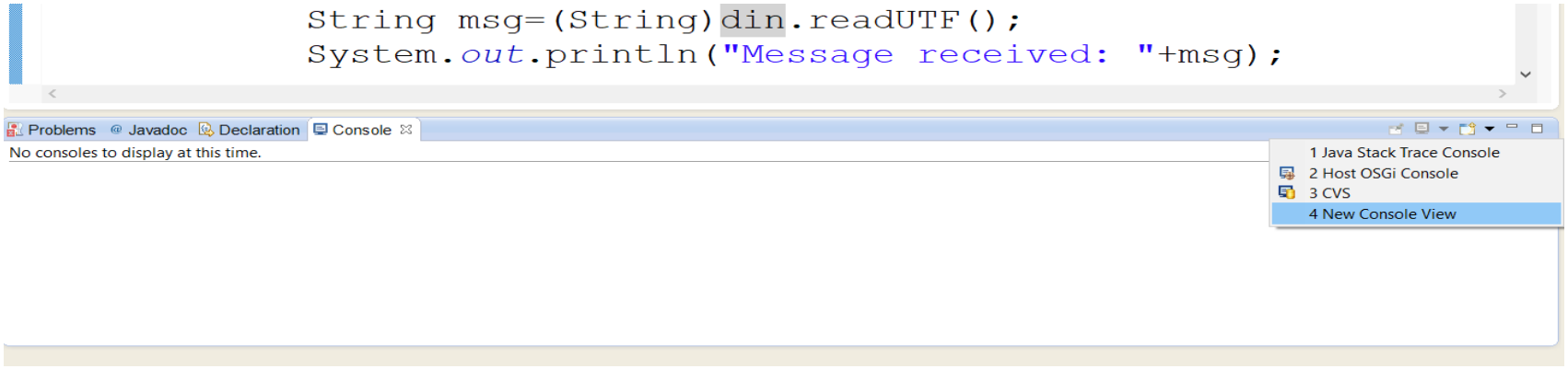
TCP Example continued…

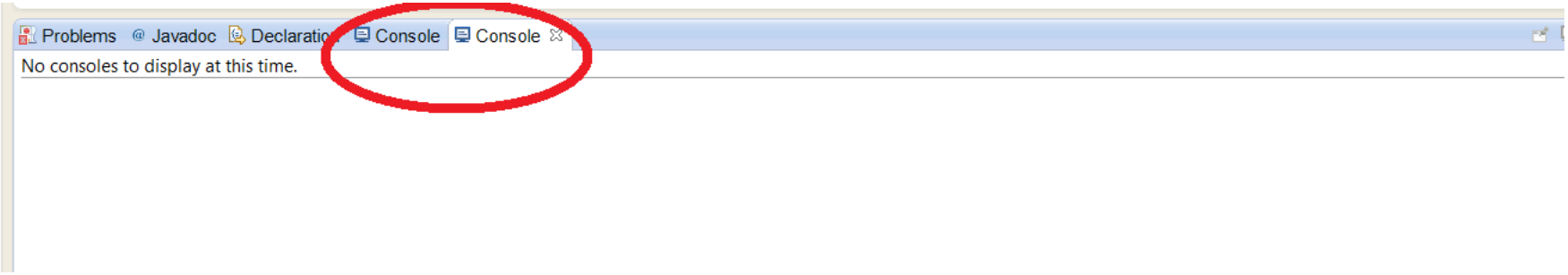
```
//MyClient.java
import java.io.*;
import java.net.*;
public class MyClient {
  try{
      Socket s=new Socket("localhost",6666);                    //create client socket

      DataOutputStream dout=new DataOutputStream(s.getOutputStream());
      System.out.println("This is Client...");
      dout.writeUTF("Hello Server");                            //send message through dout

      dout.flush();
      dout.close();
      s.close();                                                //close the client socket
  }catch(Exception e){System.out.println(e);}
   }
}
```

Run the program

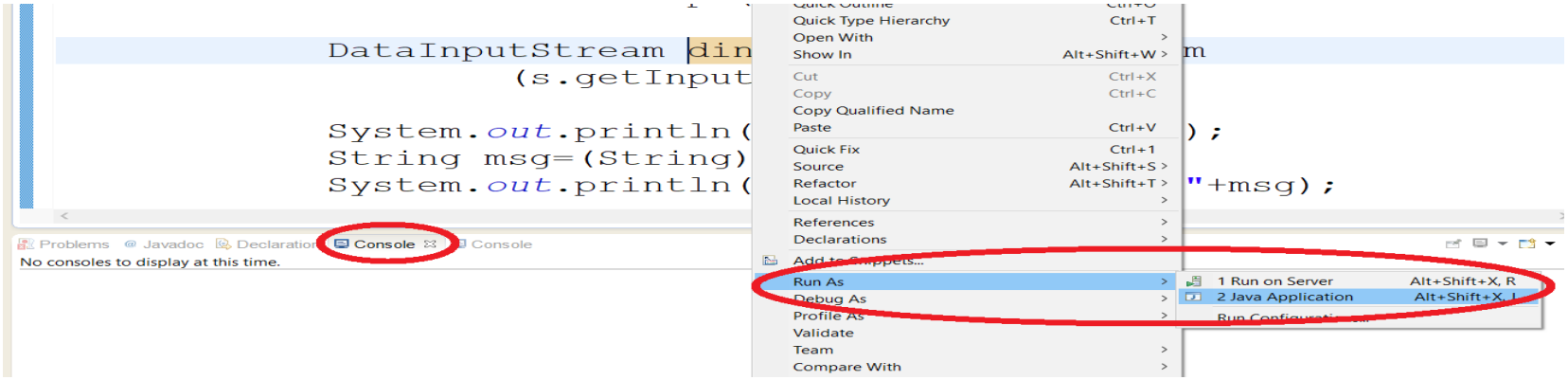
Soon after typing both client and server programs, follow these steps:

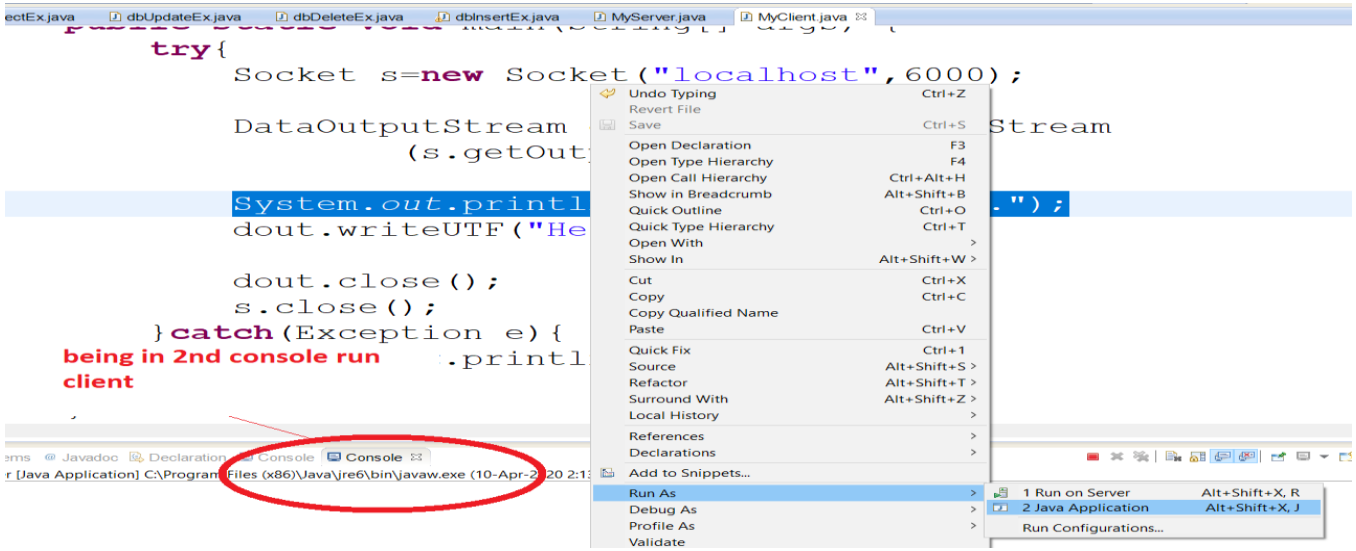
- Insert an extra console – 1st for server and 2nd for client.





- Then try running server first being in the 1st console, then click on 2nd console and run the client.





Please try-out

- If you wish to have a 2-way messaging, then include:
  - DataInputStream and DataOutputStream objects in both Client and Server.
  - Use writeUTF and readUTF in the proper places in both client as well as server.

THANK YOU