1. **What is TCP Socket?**
    - TCP (Transmission Control Protocol) socket is a communication endpoint used to establish a connection between two nodes over a network. It provides reliable, ordered, and error-checked delivery of data between applications running on hosts connected via an IP network. TCP sockets operate on a client-server model, where one side initiates the connection (client) and the other side accepts it (server).

2. **Steps in Sender Side (Client Side):**
    - Create a socket: Create a TCP socket using the `Socket` class, specifying the IP address and port number of the server.
    - Establish Connection: Use the `connect()` method to establish a connection to the server.
    - Send Data: Use the output stream obtained from the socket to send data to the server.
    - Close Connection: Close the socket to release resources once the communication is complete.

3. **Steps in Receiver Side (Server Side):**
    - Create a Server Socket: Create a TCP server socket using the `ServerSocket` class, specifying the port number.
    - Accept Connection: Use the `accept()` method to accept incoming client connections.
    - Receive Data: Use the input stream obtained from the client socket to receive data sent by the client.
    - Close Connection: Close the client socket once communication is complete. Optionally, the server socket can remain open to accept further connections.

SERVER SIDE PROGRAM:

```
import java.io.*;

import java.net.*;


public class Client {

  public static void main(String[] args) {
        try {
            // Create a socket to connect to the server
            Socket socket = new Socket("server_ip", server_port);

            // Get the output stream of the socket
            OutputStream outputStream = socket.getOutputStream();

            // Send data to the server
            String message = "Hello, Server!";
            outputStream.write(message.getBytes());

            // Close the socket
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

RECEIVER SIDE PROGRAM

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

public class Server {
    public static void main(String[] args) {
        try {
            // Step 1: Create a server socket
            ServerSocket serverSocket = new ServerSocket(server_port);
            // Step 2: Accept client connection
            Socket clientSocket = serverSocket.accept();

            // Step 3: Get the input stream of the client socket
            InputStream inputStream = clientSocket.getInputStream();

            // Step 4: Receive data from the client
            byte[] buffer = new byte[1024];
            int bytesRead = inputStream.read(buffer);
            String message = new String(buffer, 0, bytesRead);
            System.out.println("Message from client: " + message);

            // Step 5: Close the client socket
            clientSocket.close();

            // Step 6: Close the server socket
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();  }}}
```

1. What is UDP Socket?

- UDP (User Datagram Protocol) socket is a communication endpoint used for sending and receiving data between two hosts over a network. Unlike TCP, UDP is connectionless and does not guarantee delivery or order of packets. It's often used for applications where real-time communication and low overhead are more important than reliability, such as streaming media or online gaming.

2. Steps in Sender Side (Client Side):

1. **Create a DatagramSocket**: Create a UDP socket to send data.
2. **Create DatagramPacket**: Create a packet containing the data to be sent and the recipient's address and port.
3. **Send DatagramPacket**: Send the packet over the network using the socket.
4. **Close DatagramSocket**: Close the socket once communication is complete.

3. Steps in Receiver Side (Server Side):

1. **Create DatagramSocket**: Create a UDP socket to receive data.
2. **Create DatagramPacket**: Create a packet to store the received data.
3. **Receive DatagramPacket**: Receive the packet over the network using the socket.
4. **Process Data**: Extract and process the data from the received packet.
5. **Close DatagramSocket**: Close the socket once communication is complete.

SERVER SIDE PROGRAM:

```
import java.io.*;

import java.net.*;


public class UDPClient {

    public static void main(String[] args) {

        try {

            // Step 1: Create a DatagramSocket

            DatagramSocket socket = new DatagramSocket();


            // Step 2: Create DatagramPacket with data, IP address, and port of the server

            byte[] data = "Hello, Server!".getBytes();

            InetAddress serverAddress = InetAddress.getByName("server_ip");

            int serverPort = 12345; // Replace with the server's port number

            DatagramPacket packet = new DatagramPacket(data, data.length, serverAddress,

           serverPort);
```

```
            // Step 3: Send DatagramPacket
            socket.send(packet);

            // Step 4: Close DatagramSocket
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }}}
```

RECEIVER SIDE PROGRAM:

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

public class UDPServer {
    public static void main(String[] args) {
        try {
            // Step 1: Create a DatagramSocket
            DatagramSocket socket = new DatagramSocket(12345); // Listen on port 12345

            // Step 2: Create DatagramPacket to store received data
            byte[] buffer = new byte[1024];
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

            // Step 3: Receive DatagramPacket
            socket.receive(packet);

            // Step 4: Process Data
            String message = new String(packet.getData(), 0, packet.getLength());
            System.out.println("Message from client: " + message);


            // Step 5: Close DatagramSocket
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }}}
```

JAVA BEAN

1. What is JavaBean?

- JavaBean is a reusable software component model for Java, designed to be easily manipulated in visual development environments. It is essentially a Java class that follows specific conventions, allowing it to be easily integrated into various Java development environments.

Visual Development Environments (VDEs) are software tools that allow developers to create graphical user interfaces (GUIs) and develop software applications using visual components rather than writing code manually. These environments provide a visual interface for designing and constructing software applications, often with drag-and-drop functionality and graphical editors. Some common features of visual development environments include: **Graphical User Interface (GUI) Builders, Component Libraries**, **Event Handlers**, **Data Binding.**

2. Why JavaBean?

- JavaBean provides a standardized way to create reusable software components in Java. These components are self-contained, easily manageable, and can be visually manipulated in development tools like IDEs (Integrated Development Environments) or visual design tools. JavaBeans simplify the development process by promoting modular, reusable code.

Let's create a simple JavaBean representing a **`Person`** with properties such as **`name`** and **`age`**

```
public class Person {
  private String name;
```

```
    private int age;

    // Default constructor (required for JavaBean)

    public Person() {

    } // Getter method for name property

    public String getName() {

        return name;

    }

    // Setter method for name property

    public void setName(String name) {

        this.name = name;

    }

   // Getter method for age property

    public int getAge() {

        return age;

    }

     // Setter method for age property

    public void setAge(int age) {

        this.age = age;

    }

}
```