Student Name	**Pratik Manoj Dharam**
SRN No	31232438
Roll No	20
Program	Computer Engg.
Year	Third Year
Division	H
Subject	Computer Network Laboratory (BTECCE22506)
Assignment No	9

Assignment Number - 09

Title : Socket Programming for TCP Client and TCP Server.

Problem Statement : Implement a simple **TCP Client-Server** communication using **Socket Programming** in **Java**. The client sends a message to the server, and the server responds with the message prefixed by "Server received:". Both client and server should run on the same machine, using TCP as the communication protocol.

Theory :

Server: The server should listen on a specific port for incoming client connections. When a client connects, the server should accept the connection, receive a message, and respond by sending back the message prefixed with "Server received: ". The server should handle one connection at a time.

Client: The client should connect to the server on the specified IP address and port, send a message, and print the server's response.

Requirements:

- TCP connection.
- The server should be able to accept a connection, receive a message from the client, and send a response back.
- The client should be able to send a message to the server and receive the server's response.

Source code:

TCP Server code:

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

public class TCPServer {
    public static void main(String[] args) {
        try {
            // Create a server socket listening on port 65432
            ServerSocket serverSocket = new ServerSocket(65432);
            System.out.println("Server listening on port 65432...");

            // Accept incoming connection from the client
            Socket clientSocket = serverSocket.accept();
            System.out.println("Connected to client: " + clientSocket.getInetAddress());

            // Set up input and output streams
            BufferedReader in = new BufferedReader(new
```

```
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

            // Read message from the client
            String clientMessage = in.readLine();
            System.out.println("Received from client: " + clientMessage);

            // Send response back to the client
            String response = "Server received: " + clientMessage;
            out.println(response);

            // Close the connection
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

TCP Client code:

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

public class TCPClient {
   public static void main(String[] args) {
      try {
         // Create a client socket and connect to the server at localhost on port 65432
         Socket socket = new Socket("127.0.0.1", 65432);

         // Set up input and output streams
         PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
         BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

         // Send message to the server
         String message = "Hello, Server!";
         out.println(message);
         System.out.println("Sent to server: " + message);

         // Receive response from the server
         String serverResponse = in.readLine();
         System.out.println("Received from server: " + serverResponse);
```

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

Output:

Output:

Server Output:

```bash
Server listening on port 65432...
Connected to client: /127.0.0.1
Received from client: Hello, Server!
```

Client Output:

```bash
Sent to server: Hello, Server!
Received from server: Server received: Hello, Server!
```

Conclusion:

This **TCP Client-Server** communication using **Socket Programming in Java** demonstrates how the server listens for incoming connections and responds to client messages. The client connects to the server, sends a message, and receives a response. This implementation can be expanded for more advanced scenarios such as multi-threaded servers to handle multiple