# Practical-1

**Aim:**

The objective of this assignment is to design and implement a simple client-server application in Java, where the server echoes the client's messages. Additionally, both the server and the client should be able to exchange messages. When the client types exit(), both the client and server should stop gracefully.

**Theory:**

Client-server architecture is a model in which multiple clients connect to a central server. The server typically handles requests and responses, acting as the central communication hub. In this particular implementation, we focus on a basic form of communication where the server simply echoes messages back to the client and can also send messages to the client.

**Client-Server Communication:**

1. **Client**: Initiates communication by connecting to the server using a socket. It sends and receives messages through input/output streams.
2. **Server**: Listens for incoming connections using a ServerSocket. Once a client is connected, the server can send and receive messages through input/output streams.

**Graceful Shutdown:**

A graceful shutdown ensures that all resources (sockets, input/output streams) are closed properly, avoiding potential data loss or corruption. In this implementation, the client sends a special command (exit()) to the server, signaling both parties to terminate their connections and exit.

**Functions & Modules Used:**

**Server Code (EchoServer.java):**

- **Modules/Classes:**
  - ServerSocket: Listens for incoming client connections.
  - Socket: Represents the connection between the server and a specific client.
  - PrintWriter: Used to send messages to the client.
  - BufferedReader: Used to receive messages from the client.
  - Thread: For handling server-side message sending and listening concurrently.
- **Functions:**
  - start(int port): Starts the server on the specified port, waits for client connections, and handles communication.
  - stop(): Gracefully shuts down the server, closing all sockets and streams.

**Client Code (EchoClient.java):**

- **Modules/Classes:**
  - Socket: Represents the connection between the client and the server.
  - PrintWriter: Used to send messages to the server.
  - BufferedReader: Used to receive messages from the server and get user input from the console.
  - Thread: For handling client-side message listening concurrently.
- **Functions:**
  - startConnection(String ip, int port): Connects to the server at the specified IP address and port.
  - stopConnection(): Gracefully shuts down the client, closing all sockets and streams.

**Analysis:**

The implementation consists of a basic client-server application where the server echoes messages received from the client and can also send messages to the client. The main

challenge was to ensure that both the client and the server shut down gracefully when the exit() command is issued by the client.

**Communication Flow:**

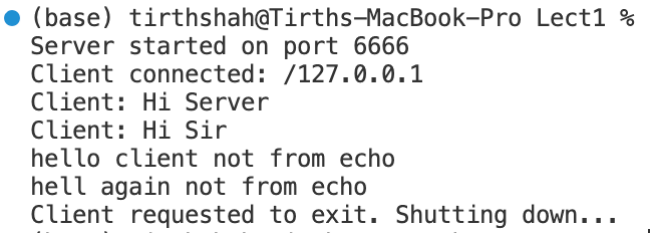
1. The client initiates a connection to the server and receives a confirmation message indicating successful connection.
2. The client can send messages to the server, which the server echoes back.
3. The server can also send messages to the client.
4. Upon entering exit(), the client sends this command to the server, prompting both the client and the server to terminate their connections and exit.

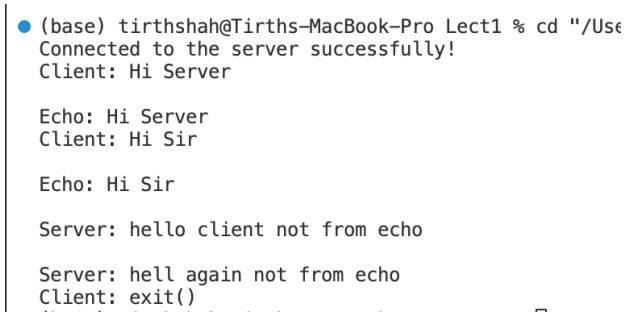
**Concurrency Considerations:**

- Separate threads were used on both the client and the server to handle concurrent reading and writing of messages, ensuring that the server can send messages to the client even while the client is typing a message, and vice versa.

**Test Case and Output:**

1. Client first will talk to server and show echoing abilities.
2. Server will type a message not received from client showing the second ability.
3. Client will show the exit() capability.



**Conclusion:**

The assignment successfully implements a simple client-server communication system in Java. The server can echo messages from the client and also send messages back to the client. Additionally, a graceful shutdown is achieved when the client sends the exit() command, ensuring that both the client and the server close their connections and terminate without errors.

The use of multithreading allows for concurrent message handling, providing a seamless communication experience between the client and the server. The solution meets the specified requirements, making it a solid foundation for more complex client-server applications.