Registration No. :

CIIT/FA23-BCS-209/LHR (A-section)

Submitted By :

Umair Ameer Hussain

Submitted To :

Muhammad Shahid bhatti

Course Name and Code :

Object oriented programming

➢ Demo Class (Client-Server Application)

This Java application demonstrates a simple client-server communication system using socket programming and multithreading. The program allows a server to wait for a connection from a client, handle message exchanges, and perform basic operations like sending and receiving messages.

Key Components:

- **ServerSocket**: The server listens for incoming client connections on a specific port (8080).
- **PrintWriter and Scanner**: These are used for sending and receiving data between the client and server.
- **Runnable Interface**: Implements multithreading, allowing the server to handle client communication in a separate thread.

Classes Used:

1. **Demo**:
    - **Main Method**: The `main` method is the entry point of the program. It creates an instance of the `MsgSystem` class (presumably used for managing contacts and messages) and a `DefaultMsgs` instance (used to initialize default messages). The method presents a simple text-based menu to the user for interacting with the system. The available options are:
        - **1**: View the contact list.
        - **2**: Send a message to a contact.
        - **3**: View received messages.
        - **4**: View status-based messages.
        - **5**: Add a contact.
        - **6**: Delete a contact. The user can interact with the system by selecting an option, and the loop continues until the user exits.
    - **run Method** (from `Runnable`): The `run` method is executed in a separate thread and handles the server-side functionality:
        - A `ServerSocket` listens for a client connection on port 8080.
        - When a client (identified as "Umair") connects, the server prints a confirmation message and begins reading and writing messages.
        - It reads messages from the client and displays them. If the client sends the message "bye", the server will terminate the communication and close the connection.
        - The server prompts the user to input messages to send back to the client. If the server sends "bye", the communication ends.

Networking:

- **ServerSocket**: Creates a socket that listens for incoming client connections.
- **Socket**: Once a client connects, a socket object is created for communication.
- **PrintWriter**: Sends messages to the client over the socket connection.
- **Scanner**: Reads messages received from the client.

Menu System (in the `main` method):

The program provides a text-based menu system for the user to interact with the messaging system. This includes the following options:

- **Option 1**: View the contact list. Displays all available contacts.
- **Option 2**: Send a message. Allows the user to send a message to a specific contact.
- **Option 3**: View messages. Displays received messages.
- **Option 4**: View status-based messages. Presumably, this shows messages related to the user's status (e.g., online/offline).
- **Option 5**: Add a contact. Allows the user to add a new contact to the system.
- **Option 6**: Delete a contact. Allows the user to remove a contact from the list.

Networking Flow:

1. **Client Connection**: When a client connects, the server will print `Umair Connected`.
2. **Message Exchange**: After the connection, the server can send and receive messages. It reads incoming messages from the client and sends responses back.
3. **Disconnection**: If the client sends a "bye" message, the server will end the connection and print "Umair disconnected". Similarly, if the server sends "bye", the communication ends.

Code Structure:

- **MsgSystem**: This class appears to handle the management of contacts and messages. It is assumed that it includes methods for displaying the contact list, sending and receiving messages, and managing contacts.
- **DefaultMsgs**: This class seems to define default messages that are used when sending messages. The method `originalMessages(m1)` likely initializes the default messages for the messaging system.
- **Multithreading**:
  - The `Demo` class implements the `Runnable` interface, which allows for concurrent execution. The `run()` method handles the server-side communication in a separate thread to avoid blocking the main application loop.

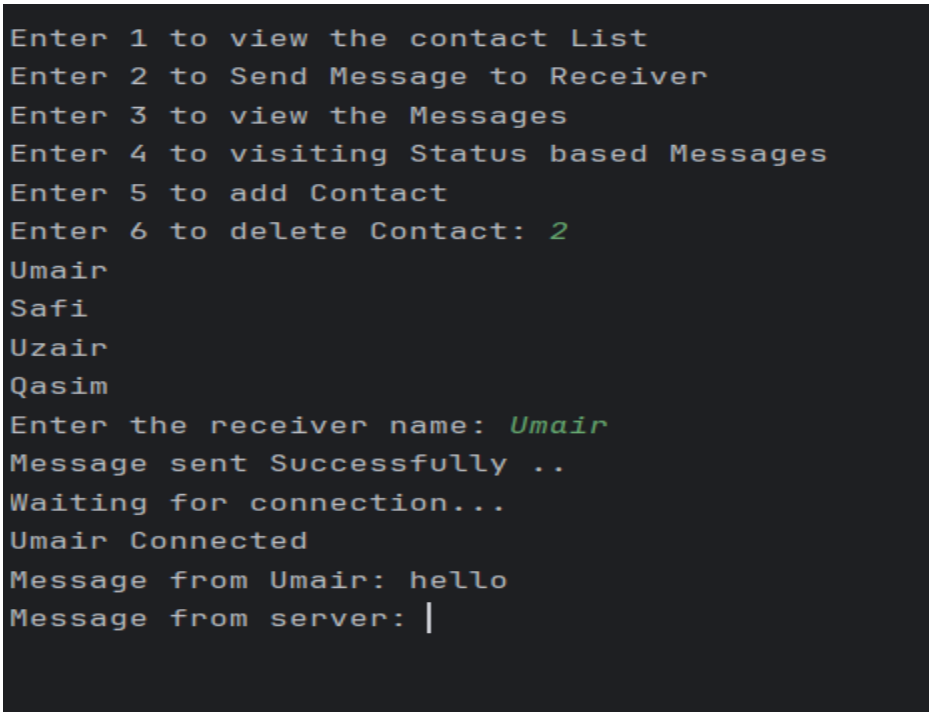
Error Handling:

The program uses basic exception handling with `try-catch` blocks to handle `IOException` that may occur during networking operations (e.g., issues with socket connections).

Output box

Sent connection:

Connected to Client:



Communication Ends:

```
Enter 6 to delete Contact: 2
Umair
Safi
Uzair
Qasim
Enter the receiver name: Umair
Message sent Successfully ..
Waiting for connection...
Umair Connected
Message from Umair: hello
Message from server: how are you
Message from Umair: bye
Umair disconnected.
```

- **Client Class (Client-Side Implementation)**

This Java program is the client-side of a client-server communication system. It connects to a server and allows the user (identified as "Umair" in this case) to send and receive messages interactively. Below is a breakdown of the functionality:

Key Components:

1. **Socket**:
    - Establishes the connection to the server using the server's IP address (`localhost`) and port number (8080).

2. **PrintWriter**:
    - Used to send messages to the server.
3. **Scanner**:
    - **`in`**: Reads incoming messages from the server.
    - **`sc`**: Reads the user's input (messages to be sent to the server).

Code Breakdown:

Initialization:

- **IP and PORT**:
    - The client connects to the server at `localhost` on port `8080`.
- **Socket Connection**:
    - The client establishes a socket connection to the server using the `Socket` class.
    - The `out` variable is used to send messages to the server via the `PrintWriter` object.
    - The `in` variable listens for incoming messages from the server using a `Scanner`.
    - The `sc` variable allows the user to type messages that will be sent to the server.

Communication Loop:

The `while (true)` loop is the heart of the communication between the client and server. It continuously:

1. **Sending Messages**:
    - The client prompts the user (Umair) to type a message.
    - The message is sent to the server using the `out.println()` method.
    - If the user types "bye", the communication ends and the client terminates the connection.
2. **Receiving Messages**:
    - The client listens for any messages from the server using `in.hasNextLine()`. If the server sends a message, it is displayed on the client side.
    - If the server sends the message "bye", the client recognizes this and ends the communication.

Handling Termination:

- The communication ends if either the client sends "bye" or the server sends "bye". Both cases result in the following:
    - The client prints the message `"Communication is ended."` and terminates the loop.
    - If the server sends "bye", the client prints `"Server disconnected."` and exits the communication.

Error Handling:

- The `try-catch` block is used to handle potential input/output errors related to the socket connection. If there is an issue establishing the connection or during the communication process, an `IOException` is caught, and the stack trace is printed.

Example Flow:

1. **Client Sends Message**:
    - The client sends a message like "Hello server!" to the server.
2. **Server Responds**:
    - The client waits for a response. The server might send back a message such as "Hello Umair! How can I help you today?"
3. **Repeat**:
    - The client can continue sending and receiving messages until either party sends "bye".

Code Structure:

- **Main Method**: The main method in the `Client` class starts by establishing the socket connection to the server. Once the connection is established, it enters the communication loop, allowing for interaction with the server.
- **Communication**: The program allows for bi-directional communication (sending and receiving messages). The client can input messages, which will be sent to the server, and it will display any incoming messages from the server.

Output box :

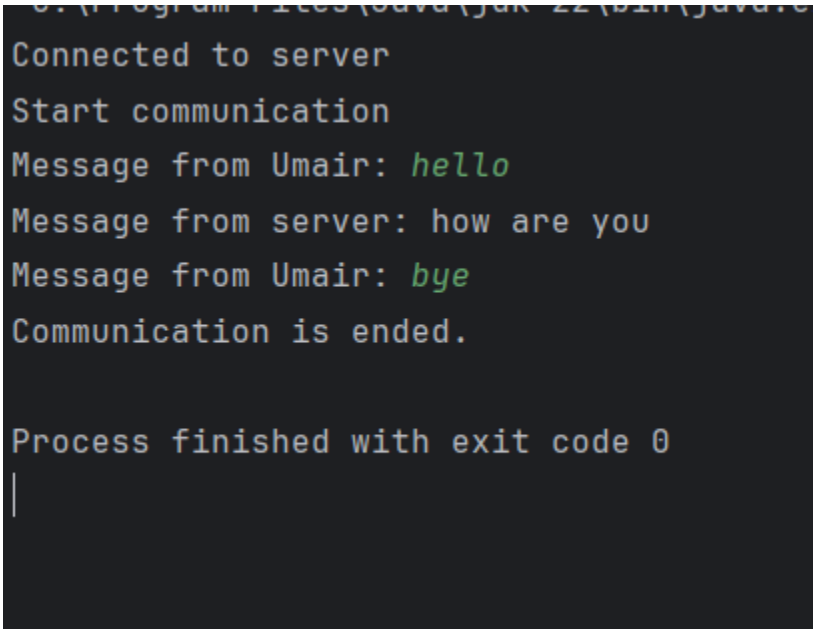
Connection with server:

```
Connected to server
Start communication
Message from Umair: hello
```

Communication with server:

```
Connected to server
Start communication
Message from Umair: hello
Message from server: how are you
Message from Umair:
```

Connection ends :



----------The End----------