Description

This project involves building a **basic client-server chat application** in Java where the client connects to a server over a network to send and receive messages. The functionality is to allow real-time text-based communication between the client and server, with the client capable of sending messages to the server and receiving responses in a loop.

The application consists of two main components:

- **Chat Client**: This part is responsible for interacting with the user, sending messages to the server, and receiving responses from it.
- **Chat Server**: The server listens for incoming client connections, processes messages from clients, and responds appropriately.

Client-Server Communication: The client and server communicate using a **TCP/IP** socket connection. The client initiates a connection to the server at a specified hostname (local host) and port number (13 used). After the connection is established, both the client and server can send and receive messages in real-time.
Single Threaded: The current design is single threaded for simplicity. The client sends messages to the server and waits for responses in a loop, until a message is received.

Used Java, the programming language for both the client and server-side code.
Used Java's `java.net` package is used to establish a network connection between the client and server using sockets.

Used **I/O Streams**: Input and Output streams (`BufferedReader`, `PrintWriter`) are used for reading from and writing to the socket.

Design

Client-Side:

- **Socket Initialization**: The client creates a `Socket` object to connect to the server on a specified address and port.
- **Message Sending/Receiving**: The client uses `PrintWriter` to send messages to the server and `BufferedReader` to read responses from the server.
- **User Interaction**: The client continuously prompts the user for input. It checks if the user types a special exit command (`\q`) to terminate the chat.

- **Error Handling**: If the client cannot connect to the server, or if the connection is lost during the session, the client catches exceptions (e.g., `UnknownHostException`, `IOException`) and informs the user about the error.

Server-Side

- **Socket Listening**: The server listens on a specific port for incoming client connections using **ServerSocket**.
- **Message Processing**: After a client connects, the server processes the messages sent by the client and can send responses back to the client.
- **Continuous Communication**: The server continuously waits for new messages from the client and responds, allowing a text-based chat until either client or server terminates the connection.
- **Error Handling**: Like the client, the server handles connection errors (e.g., lost connection) and gracefully shuts down when necessary.

Instructions for Running the Application

Make sure **Java Development Kit (JDK)** installed on your system

Text editor or IDE (Eclipse)

Instructions

1. Create the files ChatClient.java and ChatServer.java in a new project **NetworkProject** and in the **src** directory created a package **ie.atu.sw** there I created the java files.
2. Compile the files using javac Command
   Open a new terminal window (or command prompt) to compile.
   \Network Project\src\ie\atu\sw> javac ChatServer.java ChatClient.java
   This will generate two .class files: ChatClient.class and ChatServer.class.
3. Start the server.
   Open the terminal to run the server.
   First, you need to run the server. The server listens for incoming client connections.
   \Network Project\src> java ie.atu.sw.ChatServer
   The server should now be waiting for a connection on the specified port (default: port 13)
4. Start the client.

Open a new terminal window (or command prompt) to run the client.
Run the following command to start the **client**:
\Network Project\src> java ie.atu.sw.ChatClient
The client will attempt to connect to the server on `localhost` (which means the same machine) and port `13.`

5. Interaction Between Client and Server
   After connecting, the client can send messages to the server, and the server will respond accordingly.
   Client-side:

- The client will prompt the user for input, and the user can send messages to the server.
- To exit the chat, type `\q` and press **Enter**.
- **Server-side**:
- The server will receive messages from the client and send responses back to the client.
- To exit the server, you can manually stop it using `Ctrl+C` in the terminal where the server is running.
- **Client Exit**:
- The client will exit the chat if the user types `\q`, or if the connection is lost during the session (e.g., if the server crashes or the network is disrupted).
- **Server Exit**:
- The server can be stopped by manually terminating the server process (`Ctrl+C` in the terminal).

To run the application on different machine, find the IP address of the server machine and replace localhost with this address and run

Additional Credit

1. How does the client know what address to find the server at?

The IP address and port **hardcoded** in my code
`int port = 13;`
`hostname = "localhost";` which resolves to `127.0.0.1` by default, meaning the local machine.
The address and port are fixed in the program, which makes it simple but not flexible. If you want to change the server's address or port, you'd need to modify the code itself and recompile it.

We can provide **command line parameters**.
Instead of hardcoding the server's address and port in the code, you could pass them as **command-line arguments** when you start the client. This makes the client more flexible because it doesn't require code modification to connect to different servers. The user provides the address and port at runtime.
I have included ChatClientCommandLine.java code file to implement this.
Run the following command to start the **client**:
\Network Project\src> java ie.atu.sw.ChatClientCommandLine localhost 13
Another way is to store it in a **configuration file**. This way, the client can read the configuration at runtime, making it more flexible and easier to modify the server settings without changing the code.
I have a file **config.properties** with port and IP address and a class ChatClientConfigFile.java to implement the client side.
Run the following command to start the **client**:
java ie.atu.sw.ChatClientConfigFile

- **Hardcoding** is fine for very simple, one-off applications or internal tools.
- **Command-line parameters** provide good flexibility without changing the source code.
- **Configuration files** are useful for applications that need to be easily configurable without recompiling the code, especially in production environments.

2. What happens if the client can't reach the server when it starts up?

If the client cannot reach the server when it starts up, it will encounter an exception, and the connection attempt will fail.
The type of exceptions in this code:
UnknownHostException: when the client is unable to resolve the hostname to an IP address.
This could be due to: The provided hostname is incorrect or unavailable, the server is down or not reachable on the network

```
catch (UnknownHostException ex) {
    System.err.println("Server not found: " +
    ex.getMessage());
    }
```

IOException: This occurs when the client cannot establish a connection to the server usually due to

The server is not running or listening on the expected port, and there are some network issues.

```
catch (IOException ex) {
    System.err.println("I/O error: " +
    ex.getMessage());
    }
```

3. What happens if the client and server connect initially but the connection is lost during the chat session?

   In this client- server communication model, if the connection is lost, the client will usually encounter an exception or error when trying to read from or write to the socket. The connection might be lost due to reasons like **Network issues, Server crashes or shutdowns, Closing the socket on either the client or server side**.
   In this code the issues arise when

   1. **Lost Connection During Read**:
      Reading from the server: If the client tries to read data from the server after the connection is lost, it will encounter an `IOException`. This can happen if the server crashes or disconnects unexpectedly.
      ```
      serverMessage = serverInput.readLine();
      ```
      This could throw an exception
   2. **Lost Connection During Send (Write)**:
      if the client tries to send data, it might get an exception if the connection is no longer active:
      ```
      clientOutput.println(clientMessage);
      ```
      This could throw an exception

```
catch (IOException ex) {
    System.err.println("Error during chat session:
    " + ex.getMessage());
    }
```

the `IOException` would be thrown when the client attempts to read or write to the socket after the connection is lost. The exception handling will take this.
This `IOException` would capture cases like: If trying to write to the server after it disconnected and if trying to read from the server after it closed the connection.
To handle the connection loss, improve the client code like this
exiting the chat if the connection is lost, with a user-intractive message.

```
     catch (IOException ex) {
     System.err.println("Error during chat session: " +
     ex.getMessage());
     System.out.println("Connection lost. Exiting
chat."); break;
     }
```

Reference

https://vlegalwaymayo.atu.ie/course/view.php?id=11188&section=10#tabs-tree-start
Socket Programming Example

https://docs.oracle.com/en/java/javase/11/docs/api/
https://docs.oracle.com/en/java/javase/11/

Socket Class, Server Socket class, IO exceptions class, Java Documentation, Java IO streams, Command line arguments documentation.