2024-04-18

Chat Server by Lazar Stanisavljevic

Chat Server with history

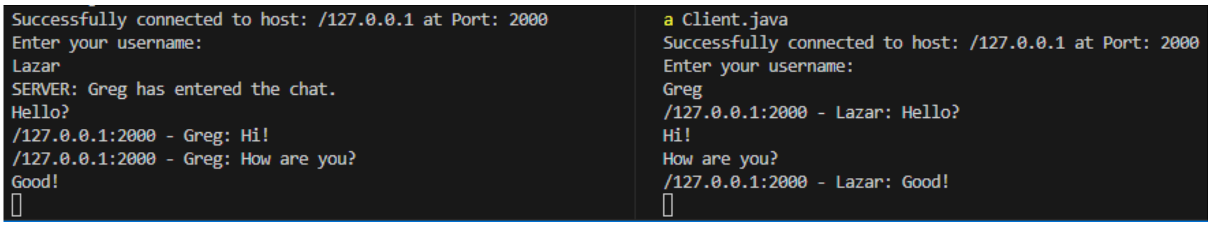
Summary 2
Development 2
Form 8
Client.java 8
Client Class 8
ChatMessage Class 11
ChatHistory Class 12
Pseudocode description 13
ClientHandler.java 17
ClientHandler Class 17
Pseudocode description 18
Server.java 19
Server Class 19
Pseudocode description 21
Function 23
Server.java 23
Error handling 24
Client.java 24
Error Handling 26

Summary

The program I have designed is a simple expansion upon a console chat application that allows users to communicate with each other in real-time over a network. Utilizing a client-server architecture, multiple clients connect to a central server, facilitating message relay between users. Through its graphical user interface, users can easily interact, sending and receiving messages via text input fields. Upon connection, users are prompted to enter a username, which serves as their identifier within the chatroom. The application stores chat history locally, allowing users to access and review previous conversations. Additionally, the chat window automatically scrolls to the bottom when new messages are received, ensuring seamless and uninterrupted communication flow. Overall, the program provides a user-friendly platform for individuals to engage in collaborative discussions and connect with others remotely.

Development

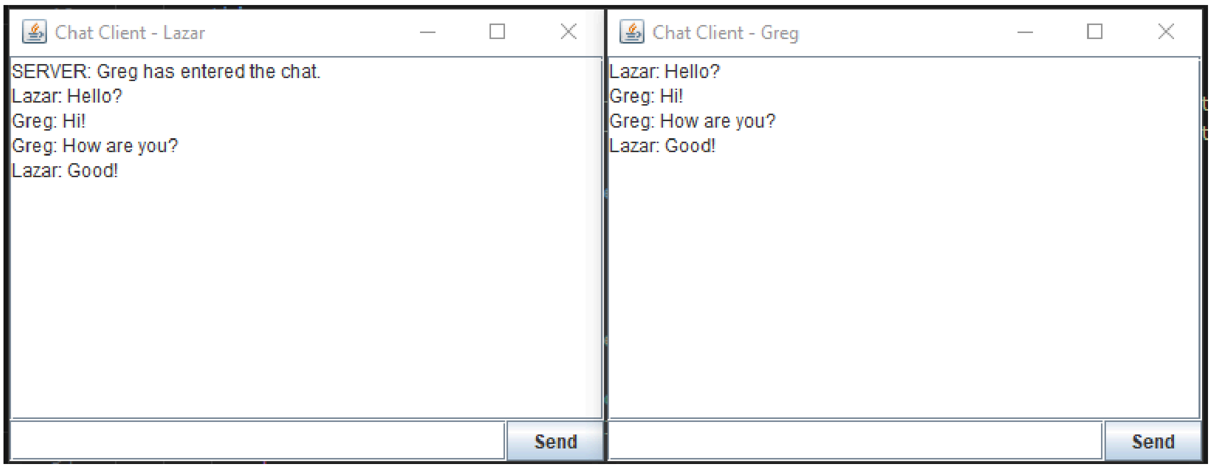
I decided to base my project off of tasks 2.1.1 (Stream sockets on Client-side) and 2.1.2 (Stream sockets on Server-side). These tasks initially intrigued me when I began and has since been at the back of my mind as to how I could expand upon this design and create something more accessible and fun to look at.
It was initially just a simple client-server connection chatroom where the user was able to connect to a chatroom through a specific host and port (has to match the host and port of the server to successfully connect). This "chatroom" was presented to the user through a console and was hence very rudimentary.
To begin with I opted to design a simple GUI to move the chatroom from the console to an independent window. This would be the first step of many to develop this idea into something more standalone and official as with software I believe it's just standard practice nowadays.



```
Successfully connected to host: /127.0.0.1 at Port: 2000
Enter your username:
Lazar
SERVER: Greg has entered the chat.
Hello?
/127.0.0.1:2000 - Greg: Hi!
/127.0.0.1:2000 - Greg: How are you?
Good!
```

```
a Client.java
Successfully connected to host: /127.0.0.1 at Port: 2000
Enter your username:
Greg
/127.0.0.1:2000 - Lazar: Hello?
Hi!
How are you?
/127.0.0.1:2000 - Lazar: Good!
```

1.1 Initial design of task 2.1.1 (Client-side)



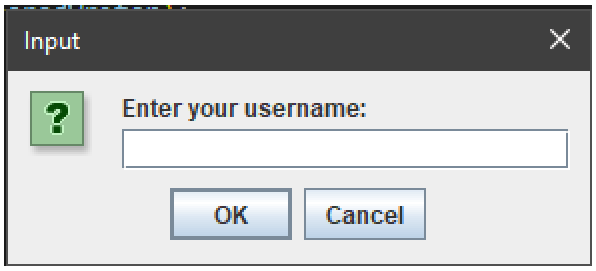
```
SERVER: Greg has entered the chat.
Lazar: Hello?
Greg: Hi!
Greg: How are you?
Lazar: Good!
```

```
Lazar: Hello?
Greg: Hi!
Greg: How are you?
Lazar: Good!
```

1.2 First iteration of design (Client-side)

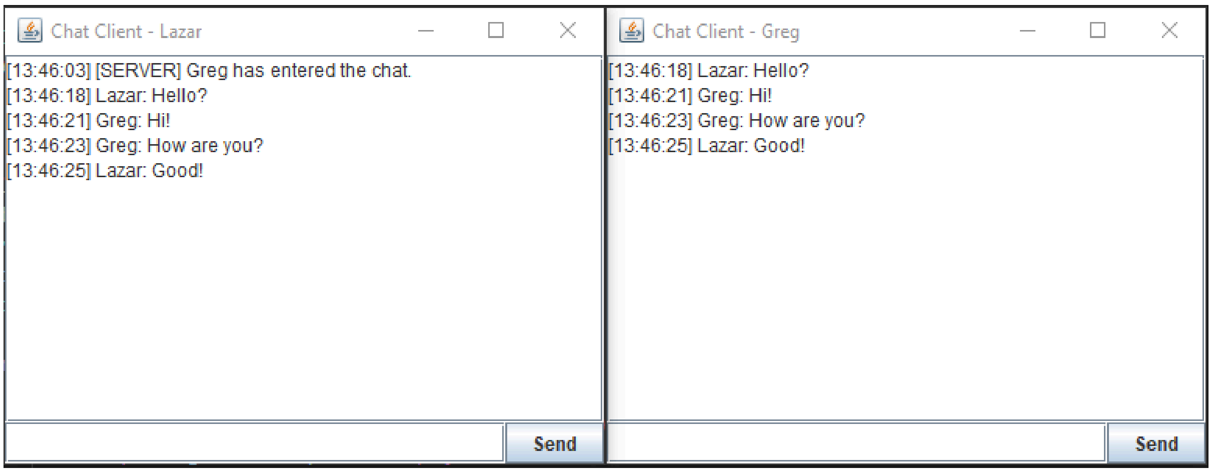
Here I have used a TextArea, TextField and a Button to create a simple, yet effective design for the chatroom. All nested within a Panel using BorderLayout. Another little detail I added that I liked a lot was setting the title of the window to "Chat Client - <username>" as this adds a bit of personality to this window for each and every user.

A problem I encountered early on was with the design of how the code was structured on the Server.java end. For task 2.1.2 I had designed the server to prompt the user with "enter your username:" upon starting the program, as seen above in picture 1.1. This, in turn, wouldn't work great unless I kept the prompt in the TextArea for the user, but as you can see in picture 1.2 I decided to remove this as I felt it wasn't proper enough. Instead I designed it so that before the Client Panel is opened, it opens a separate window using showInputDialog and requests the user to fill out their username at this stage. Because the user now filled out their username *without* sending a line of text in the console for the Server to read and register as the username, the Server.java would now await this line of text regardless, and so the first message send by the user after the Client Panel opens wouldn't be sent as a message, but in stead registered as the users username on the Server-side.
To solve this problem, I attempted to cross-reference the username variable from the Client-side to the Server-side, but was met with problems. The fix I settled with was to have the Client-side send a line of text containing the filled out username from the input prompt window and then swiftly removing it from the console. (As this text was never appended to the TextArea, it wouldn't clutter it regardless, but cleaning it from the console was just a practice I liked as it wasn't necessary for the user to see this).



2.1 Input prompt called before Panel on Client-Side is initialized

Now that the program worked as intended, I continued to develop the design of how I would like the messages to be formatted. I immediately thought of adding a time before the messages so that the users can more easily assess when a message was sent, as this can sometimes be very relevant to a conversation. To add this I simply implemented the java.util.Date package and created a new Date object every time a message is received and sent and appended this with a specific format for Hours:Minutes:Seconds using the java.text.SimpleDateFormat package.

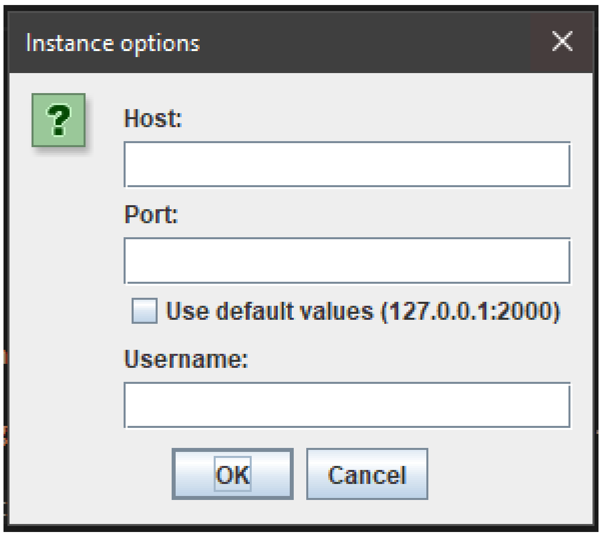


```
[13:46:03] [SERVER] Greg has entered the chat.
[13:46:18] Lazar: Hello?
[13:46:21] Greg: Hi!
[13:46:23] Greg: How are you?
[13:46:25] Lazar: Good!
```

```
[13:46:18] Lazar: Hello?
[13:46:21] Greg: Hi!
[13:46:23] Greg: How are you?
[13:46:25] Lazar: Good!
```

3.1 Text formatting iteration

I now wanted to change how much influence the user has when launching the program. At this point there were no options for the user to decide what Host or Port to connect to, it instead always defaulted to the host and port I had set up on the server end (127.0.0.1:2000). I added a new JFrame that would host the “instance options” as I called it, containing a textfield for host, port and username, as well as a checkbox that allowed the user to use the default assigned host:port as I specified above.


Instance options

Host:

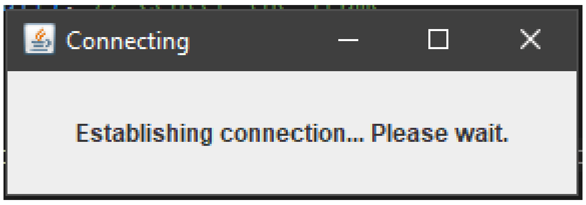
Port:

Use default values (127.0.0.1:2000)

Username:

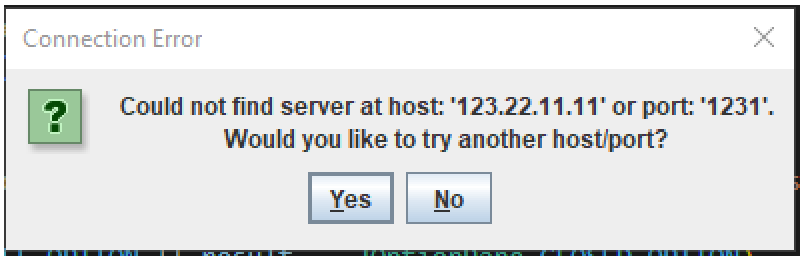
2.2 Input prompt second design iteration

Beyond this, during the establishing connection part of the code I wanted the user to have some feedback so that they understand that the program is currently running, as the Instance options window would have been closed by now and nothing else would show up until the main JFrame for the chat opened. I used a JLabel to open a temporary interface with details detailing that the connection is being established. Upon connection success, the chat window is opened, upon failure, a confirmDialog is opened requesting the user to either retry with a new host/port, or close out the program.


Connecting

Establishing connection... Please wait.

4.1 JLabel letting the user know that the client is attempting to establish a connection to server


Connection Error

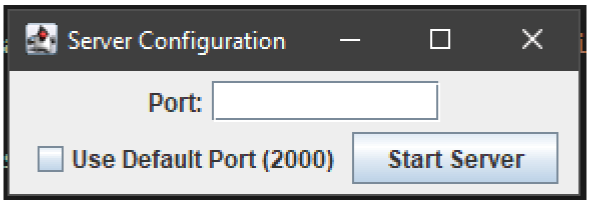
Could not find server at host: '123.22.11.11' or port: '1231'.
Would you like to try another host/port?

5.1 Error explaining that the client could not establish a connection at desired host or port.

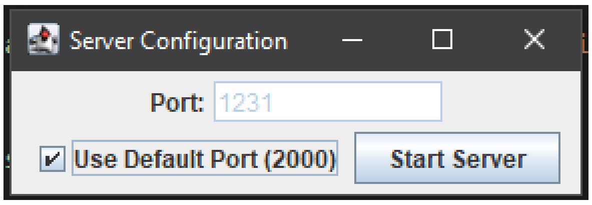
At this stage I wanted the user to be able to see the history of the chatroom from previous sessions. So to do this I converted the project to Maven on VSCode and because of this was able to reference the Google Gson library which in turn allows me to utilize .json files to store and read data. A “chat_history.json” file was added to the reference library of the Maven project and this is where the chatlogs are now stored (Client-side).

With the added chat history, the JScrollPane for the chatArea is now getting bigger and requires the user to scroll to the bottom every time they open the client, send or receive a message while the history is larger than the window itself. To fix this I decided to add a simple method that is called once upon initiation, applying the maximum value to the vertical scrollbar, as well as a separate method that is called every message sent and received *if the user is near the bottom*. The reason for the two methods is that I want the user to be able to scroll back in the history of the chat without being dragged to the bottom with every new message sent and received.

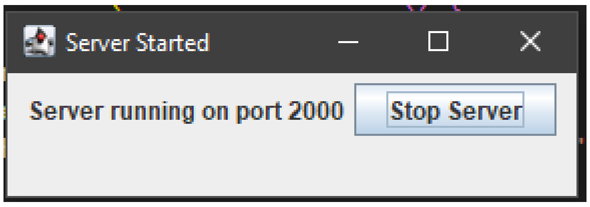
From here I opted to develop the Server-side interface, as at this stage it did not have any feedback regarding the server running apart from an initial message in the console. To develop the UI for the Server I utilised the same methods I used for the Client-side, using Javax Swing utilities. I use two frames, one for the server configuration and a separate one for the server instance information once it is running.



Server configuration UI design

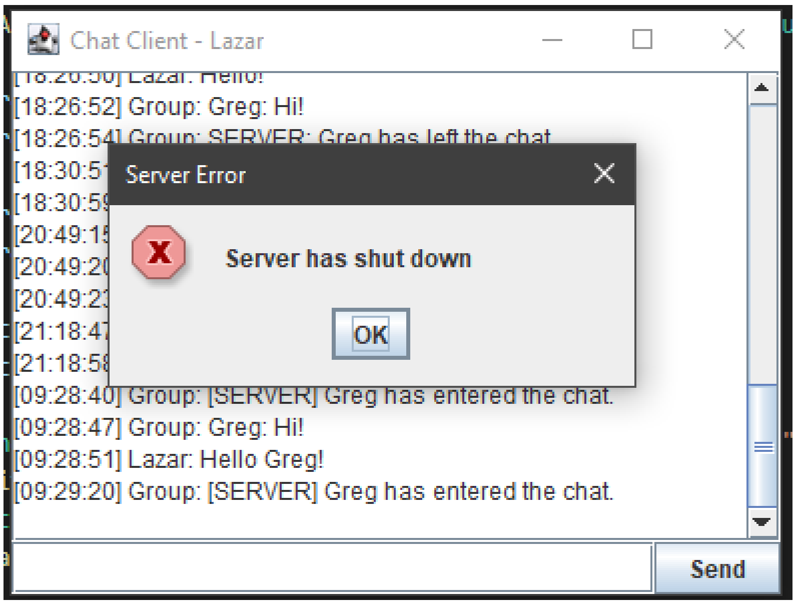


Port input field greyed out and disregarded while Default Port checkbox is checked



Server instance UI

Now that the server had its own UI, with a *Stop Server* button, I wanted to make sure that the client-side handles the server-shutdown well. Currently, when the server shuts down the client is simply unable to send any further messages. To solve this, I simply added a line to my already existing shutdown method in Client that is called upon exit. Here I call a showMessageDialog with a message "Server has shut down", that exits the program once "OK" is clicked.



```
[18:26:52] Group: Greg: Hi!
[18:26:54] Group: SERVER: Greg has left the chat
[18:30:5
[18:30:5
[20:49:1
[20:49:2
[20:49:2
[21:18:4
[21:18:5
[09:28:40] Group: [SERVER] Greg has entered the chat.
[09:28:47] Group: Greg: Hi!
[09:28:51] Lazar: Hello Greg!
[09:29:20] Group: [SERVER] Greg has entered the chat.
```

Server has shut down

Error popup on client-side when Server is shut down

Form

The program consists of three main java files. Client.java, ClientHandler.java and Server.java. Client.java is independent but requires a socket connection to Server.java. ClientHandler.java works alongside Server.java to handle the client messages between users and the server. I will structure the following form-section by initially dissecting individual code sections and presenting these with a written description as to what this section of code performs as well as include a screenshot from said code section. Furthermore, towards the end of each java file, I will include a section that will go over, using pseudocode, how the program functions as a whole.

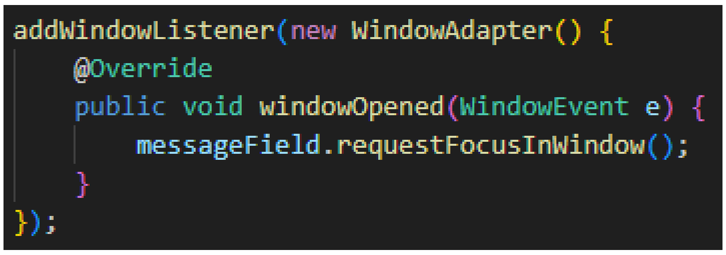
Client.java

Client.java consists of three classes in total; Client, ChatMessage and ChatHistory. The latter two are used to organise and work through the history of messages sent to a server on the client-side. This way, if a conversation was had at a previous point in time you can reconnect at another time and still have the history of the chat available to you.

Client Class

Client method

This serves as the constructor method for the Client Class. It initializes a new client instance with the provided socket and username and then sets up the chat history and UI components. I decided to add a windowListener here as well that upon a window opening event, it requests the focus in said window.

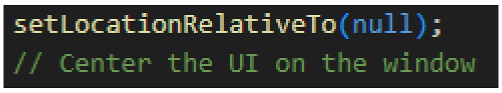


```
addWindowListener(new WindowAdapter() {
    @Override
    public void windowOpened(WindowEvent e) {
        messageField.requestFocusInWindow();
    }
});
```

WindowListener used to request the focus to the messageField element

setupUI method

Sets up the graphical user interface for the client. Configures the size, layout, and components of the client's UI, including chatArea, messageField, and sendButton. Beyond this, the method calls upon loadChatHistory and displayChatHistory methods and performs setLocationRelativeTo(null) in order to centre the UI window on the monitor/display.

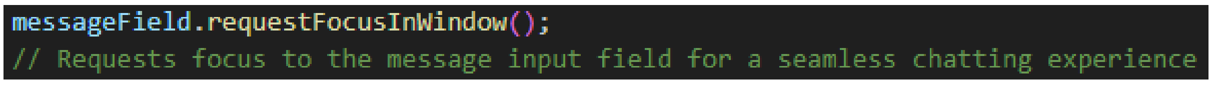


```
setLocationRelativeTo(null);
// Center the UI on the window
```

Used to centre the UI window

sendMessage method

Used to send a message to the server and update the chat interface. Retrieves the message from the message field, sends it to the server via the socket's output stream, updates the chat interface with the sent message and adds the message to the chat history. Made sure that the messageField input textbox for sending messages requests focus after sending a message for a seamless messaging experience.



```
messageField.requestFocusInWindow();
// Requests focus to the message input field for a seamless chatting experience
```

Requests focus at the input field for messages again after sending a message

listenForMessage method

Starts a new thread to continuously listen for incoming messages from the server. When a message is received, it updates the chat interface with the message and adds it to the chat history.

loadChatHistory method

Reads chat history from a JSON file and populates the chatHistory object with the loaded messages.

displayChatHistory method

Retrieves messages from the chatHistory object and displays them in the chat area with timestamps.

definiteScrollChatToBottom method

Always scrolls the chat window to the bottom to display new messages on startup. Only called on startup as it would be destructive for the user if it updated every time a new message was sent/received, as this would prevent the user from looking back at the history without being pulled down to the bottom, (the most recent message).

scrollChatToBottom method

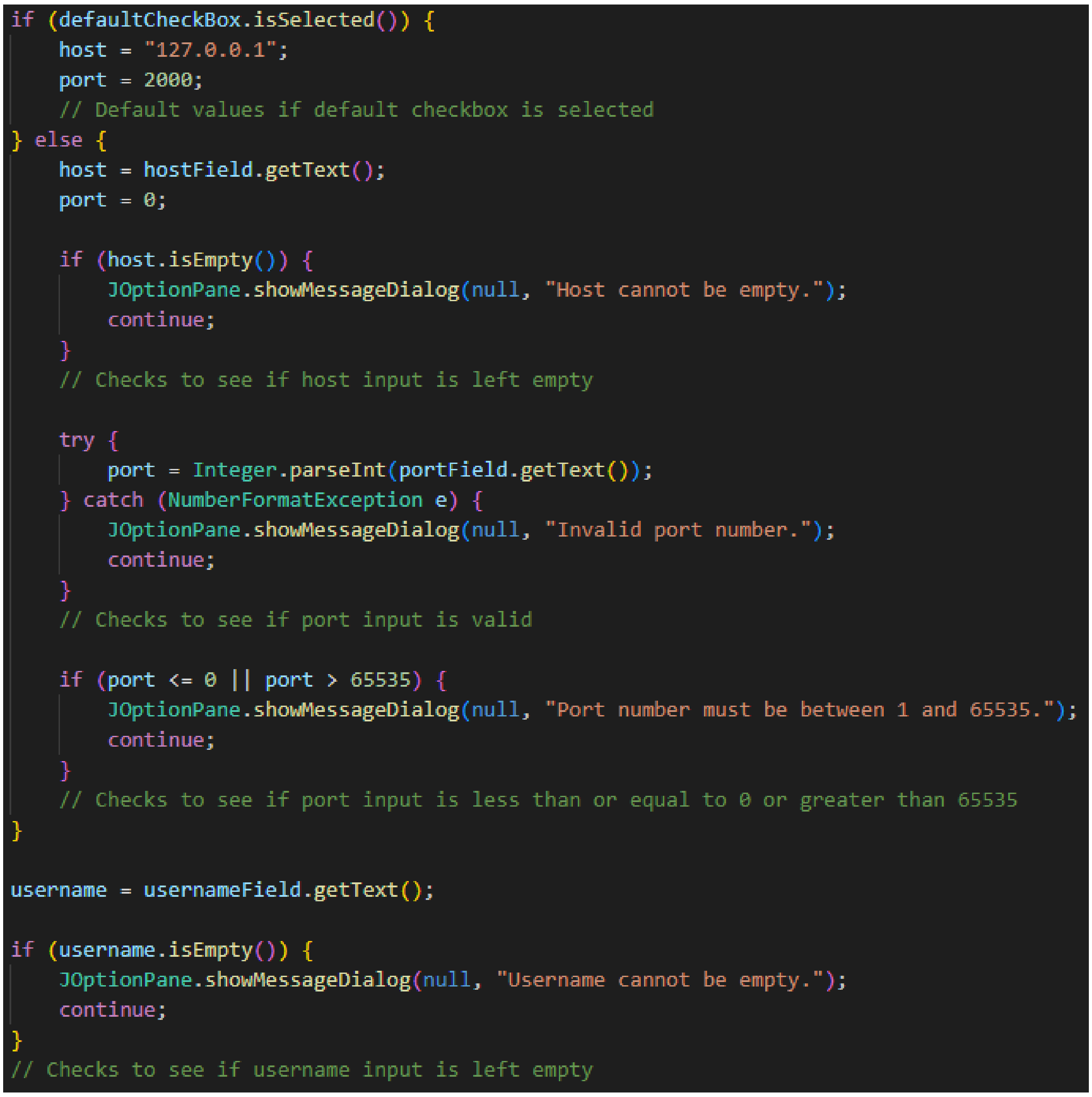
Scrolls the chat window to the bottom to display new messages if the user is already scrolled near the bottom. This was implemented as an alternative to the above method that allows the user's chat to be kept updated, (at the bottom), as long as the user is already scrolled near to the bottom. This way the user wouldn't have to keep scrolling to the bottom to keep up with the chat.

closeAll method

Used to close the socket, buffered reader, and buffered writer. Displays a message if the server has shut down and exits the program.

Main method

Sets up a GUI for the user to input connection details (host, port, username), then attempts to establish a connection with the server and starts the client application. Offers a "default" checkbox for the host and port as well as contains null/empty checkers for both host, port and username.



```
if (defaultCheckBox.isSelected()) {
    host = "127.0.0.1";
    port = 2000;
    // Default values if default checkbox is selected
} else {
    host = hostField.getText();
    port = 0;

    if (host.isEmpty()) {
        JOptionPane.showMessageDialog(null, "Host cannot be empty.");
        continue;
    }
    // Checks to see if host input is left empty

    try {
        port = Integer.parseInt(portField.getText());
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(null, "Invalid port number.");
        continue;
    }
    // Checks to see if port input is valid

    if (port <= 0 || port > 65535) {
        JOptionPane.showMessageDialog(null, "Port number must be between 1 and 65535.");
        continue;
    }
    // Checks to see if port input is less than or equal to 0 or greater than 65535
}

username = usernameField.getText();

if (username.isEmpty()) {
    JOptionPane.showMessageDialog(null, "Username cannot be empty.");
    continue;
}
// Checks to see if username input is left empty
```

Code that checks whether or not default checkbox is selected and if any user fields are left blank/empty

ChatMessage Class

ChatMessage method

This is the constructor method for the ChatMessage class. It initializes a new chat message with the provided sender, message, and current timestamp and therefore represents a message in the chat.

getSender method

A straight-forward get-method for the message sender. Returns the sender of the chat message.

getMessage method

Yet another get-method, this time for the actual message. Returns the content of the chat message.

getTimestamp method

One last get-method for the timestamp of the message. Returns the timestamp of the chat message.

ChatHistory Class

ChatHistory method

The constructor for the ChatHistory class. Initializes a new chat history instance with an empty list of messages.

addMessage method

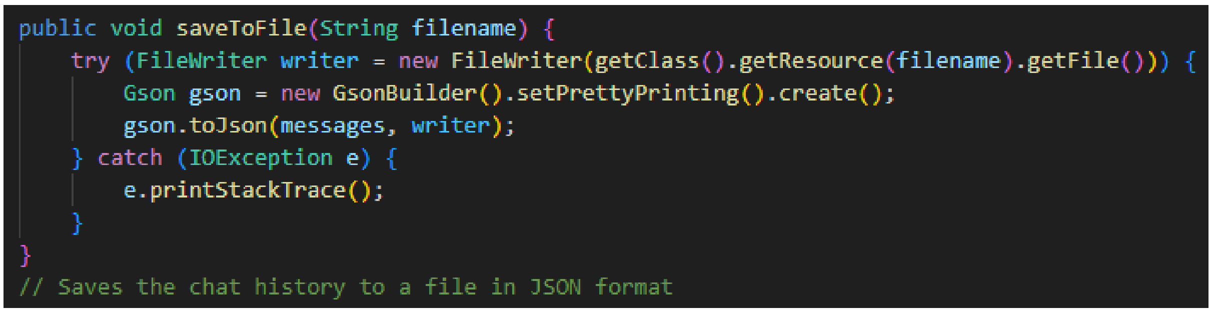
Method used to add the provided message to the list of messages in the chat history.

getMessages method

Method used to return the list of messages stored in the chat history.

saveToFile method

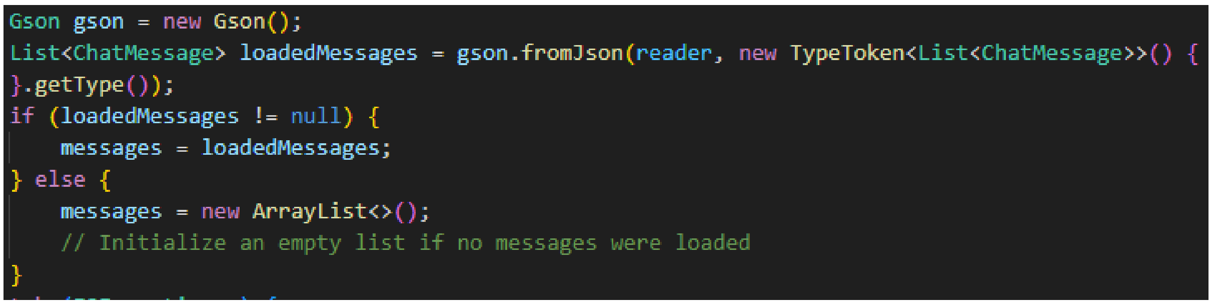
Saves the chat history to a file in JSON format by serializing the list of messages and writing it to a specific file. In my case it's "chat_history.json" located in the /resources folder of the project.



```
public void saveToFile(String filename) {
    try (FileWriter writer = new FileWriter(getClass().getResource(filename).getFile())) {
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        gson.toJson(messages, writer);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
// Saves the chat history to a file in JSON format
```

saveToFile method utilizing google.gson.Gson package to write chat history to JSON files *loadFromFile method*

Method used to read the chat history from the specified JSON file and populates the chat history instance with the loaded messages. If the file doesn't exist or an error occurs, it initializes an empty list.



```
Gson gson = new Gson();
List<ChatMessage> loadedMessages = gson.fromJson(reader, new TypeToken<List<ChatMessage>>() {
}.getType());
if (loadedMessages != null) {
    messages = loadedMessages;
} else {
    messages = new ArrayList<>();
    // Initialize an empty list if no messages were loaded
}
```

Code-segment from "loadFromFile" method that showcases how the JSON is handled to load messages into chat from history

Pseudocode description

```
Define class Client
    Declare instance variables
    Define constructor <Socket socket, String username>
        Set title of frame to "Chat Client - username"
        Initialize instance variables
        Create ChatHistory object
        Try
            Initialize buffered reader and writer
        Catch <IOException>

        Call setupUI method
        Add window listener to focus message field when window is opened

    Define setupUI method
        Set frame size and close operation
        Create UI components including chat area, message field, and send button
        Add action listeners to message field and send button
        Add components to frame
        Call methods loadChatHistory, displayChatHistory and centers UI on screen
```

```
Define sendMessage method
    Try
        Get text from messageField
        If <message is not empty> then
            Send message with timestamp and username
            Scroll chat area to the bottom
            Save chat history to file

Define listenForMessage method
    Start a new thread
        Define a Runnable that overrides the run method:
        While <socket is connected>
            Try
                Read a line from bufferedReader into
                messageFromGroup
                If <messageFromGroup is not null>
                    Send message with timestamp and username
                    Scroll chat area to the bottom
                    Save chat history to file
                Catch <IOException> and close streams and socket if
                an exception occurs

Define utility methods
    Method I ) Set scrollbar to maximum value
    Method II)  If <scrollbar is near bottom> then
            Set scrollbat to maximum value
```

```
Define main method
    While <true> loop
        Display Instance options window containing host, port and username
    fields
        If <default checkbox is selected> then
            Set host to "127.0.0.1" and port to "2000"
        Else
            Get host and port from user filled fields
            If <host or port fields are empty or null> then
                Throw error in message dialogue
        Get username from username field
        If <username is empty> then
            Throw error in message dialogue
        Set connecting dialogue to visible
        Try
            Create socket object using host and port
            Dispose connecting dialogue
            Create and show Client frame
            Call listenForMessage method for client
            Scroll chat to bottom
            Break loop
        Catch <IOException> and close connectingFrame
            Show error message, prompt to try another host/port
            Break loop

Define class ChatMessage to represent chat messages with get methods for sender
(username), message, and timestamp
```

Define class ChatHistory to manage history of chat messages including adding, saving, and loading from file

```
Define saveToFile method
    Try
        Create Gson object and write messages, (from chat history).
    Catch <IOException>

Define loadFromFile method
    Try
        Create Gson object
        Load messages from existing gson
        If <loaded messages is not null> then
            Store loaded messages into messages list
        Else
            Create new messages list
    Catch <IOException> and create a new messages list
```

ClientHandler.java

ClientHandler.java consists of the one head class, ClientHandler that carries with it from the constructor a Socket object. ClientHandler implements Runnable utilizing a run method instead of a main method to be initially executed upon start.

ClientHandler Class

ClientHandler method

In the constructor, we define a socket variable, create a buffered writer and reader, handle how the user's name is fetched, (by reading the first line sent by the user), and call a synchronised function to add the instance of a clientHandler to an ArrayList "clientHandlers". Finally a message is broadcasted to all the connected clients that a new user has entered the chat.

Run method

This method acts as the main thread operation of the ClientHandler class. Its purpose is to collect messages incoming from other clients using the "bufferedReader.readLine() method, then proceeding to broadcast said message to the client. This method got its own thread to run in as to not block the operation of the remaining code in the class.

broadcasstMessage method

Method that handles the sending of messages by the client to other clients connected to the chatserver. Initially checks whether the clienthandler instance shares the username of the client, if not, proceeds to write out the message and flushes the bufferedReader. (This is done to prevent the client sending the message from receiving the message from two sources. The user receives messages sent by users and can see their own message when they send it, though the own message is not supplied by the run method but by the broadcastMessage method.

removeClient method

This is a simple method that removes the instance of clienthandler from the clientHandlers list and broadcasts this information to the remaining users in the chat server using broadcastMessage.

closeAll method

This method calls the above removeClient method and closes the bufferedReader, bufferedWriter and the socket.

Pseudocode description

```
Define class ClientHandler
    Define ArrayList clientHandlers to keep track of all clients in session
    Define instance variables
    Define constructor <Socket socket>
        Try
            Initialize socket and bufferedWriter/Reader with socket's output/input stream
            Read the first line from bufferedReader and assign it to clientUsername
            Add ClientHandler object to clientHandlers list
            Broadcast a message announcing the new user's entry
        Catch <IOException>

    Define run method
        While <socket is connected>
            Try
                Read a line from bufferedReader and assign it to messageFromClient
                Broadcast the received message to other clients
            Catch <IOException> and close streams

    Define broadcastMessage_message method
        For <each ClientHandler clientHandler in clientHandlers>
            Try
                If <clientHandler's username is not equal to this client's username> then
                    Write the message to clientHandler's bufferedWriter
                    Flush the bufferedWriter
            Catch <IOException> and close streams
```

```
Define removeClient method:
    Remove this ClientHandler object from clientHandlers list
    Broadcast a message announcing the departure of the client

Define closeAll_socket_bufferedReader_bufferedWriter method
    Call removeClient method
        Try
            If <not null> then
                Close bufferedReader, bufferedWriter, and socket if
            they are not null
        Catch <IOException>
```

Server.java

Server.java consists of the one head class, Server that carries with it from the constructor a ServerSocket object.

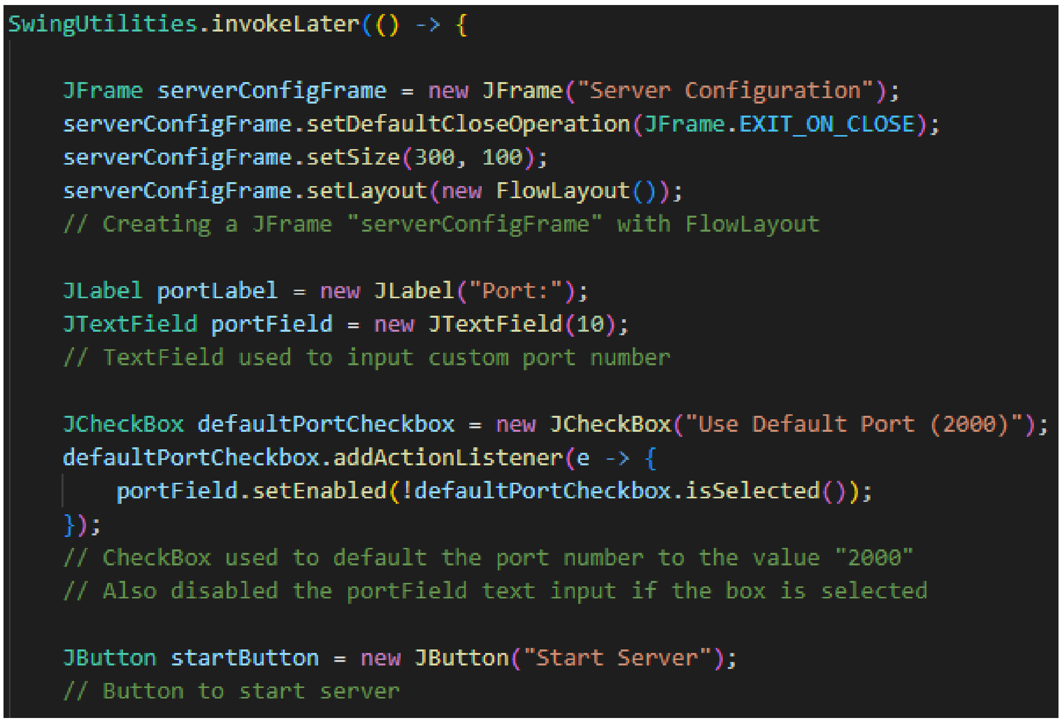
Server Class

Main method

The main method instantly calls upon SwingUtilities.invokeLater() to allow the UI to be built and executed asynchronously on the event dispatch thread. This is because, without the invokeLater method, the program would run a risk of deadlocks and potential UI freezes. Within the UI setup, we create a server configuration frame that holds the following:

- Port text input
- Default port checkbox
- Start server button

Here the user is able to input their own custom port they would like to host on, or simply default to the ordinary one I have supplied “2000”. If the user checks this default box, the port text input is both disabled and ignored.



```java
SwingUtilities.invokeLater(() -> {

    JFrame serverConfigFrame = new JFrame("Server Configuration");
    serverConfigFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    serverConfigFrame.setSize(300, 100);
    serverConfigFrame.setLayout(new FlowLayout());
    // Creating a JFrame "serverConfigFrame" with FlowLayout

    JLabel portLabel = new JLabel("Port:");
    JTextField portField = new JTextField(10);
    // TextField used to input custom port number

    JCheckBox defaultPortCheckbox = new JCheckBox("Use Default Port (2000)");
    defaultPortCheckbox.addActionListener(e -> {
        portField.setEnabled(!defaultPortCheckbox.isSelected());
    });
    // CheckBox used to default the port number to the value "2000"
    // Also disabled the portField text input if the box is selected

    JButton startButton = new JButton("Start Server");
    // Button to start server
```

Code for Server Configuration UI setup

Beyond this, still within the main method, the startButton receives an ActionListener which, upon action, performs the following:

- Checks to see if the port input text field is enabled *and* if this text field is empty. If yes to both, it alerts the user and requests an input in the text field.
- Checks to see if default port checkbox is checked, if yes, defaults port variable to 2000, else it gets the number from the port input text field.
- Attempts to create a new ServerSocket and Server using the defined port.
- Calls a method “startServer()”.
- Disposes of the server configuration UI.
- Creates a new Frame and UI to display the server status as “Server running on port <port>” alongside a “Stop Server” button that calls a method “closeServerSocket()” upon pressing.
- If a server cannot be started on the defined port an error is given to the user asking them to “check the port”.

startServer method

Within this method a thread task is created and then given a thread to run upon. The task includes awaiting a connection from a client. Upon successful connection attempt, a ClientHandler instance is created using this newly connected socket and an individual thread is created for this instance of clientHandler to handle communication with this one client on its own thread.

closeServerSocket method

This method simply sets a “serverRunning” boolean to false to halt all thread operation that originated from Server.java and proceeds to close the serverSocket created upon server creation. Finally it exits the process to fully shut down.

Pseudocode description

```
Define class Server extending JFrame
    Define instance variables
    Define constructor <ServerSocket serverSocket>
        Assign serverSocket to the instance variable

    Define startServer method
        Set serverRunning boolean to true
        Define serverTask as a lambda expression
            While <serverRunning is true> loop
                Try
                    Accept incoming connections on serverSocket and
                assign to socket
                    If <server is not running> then
                        Break loop
                    Create a new ClientHandler for the client connection
                    Start a new thread for the ClientHandler
                Catch <SocketException>  and print stack trace if serverSocket
            is not closed
            Catch <IOException>
        Create a new thread with serverTask and start it
```

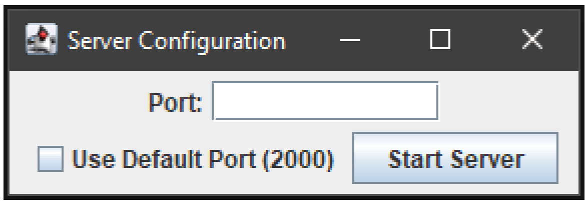
```
Define closeServerSocket method
    Set serverRunning to false
    Try
        If <serverSocket is not null> then
            Close serverSocket
        Exit the program with status code 0
    Catch <IOException>

Define main method
    Invoke SwingUtilities.invokeLater to ensure GUI operations are performed
on the Event Dispatch Thread
        Create server configuration frame/window
        If <port field is empty> then
            Throw error and request user to fill in port
        Else
            If <default port checkbox is checked> then
                Set port to 2000
            Else
                Try
                    Get port from port user field
                    If <port is invalid> then
                        Throw number format exception
                Catch <NumberFormatException> and give user an
        error message to fill in a port number between 1 and 65535
                Return
            Try
                Create ServerSocket with specified port
                Create Server object with the ServerSocket
                Start the server
                Close server configuration frame/window
                Create server running frame/window
            Catch <IOException> and show error message
```

Function

Because the program requires a server to be running in order to be able to connect to the chat server, I will go through the two processes on their own.

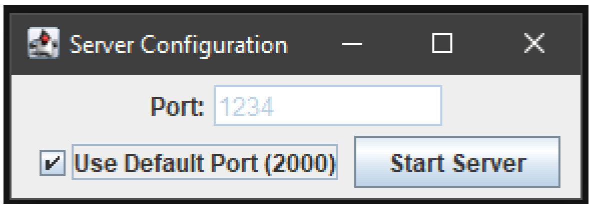
Server.java

Upon starting you are greeted with the following GUI



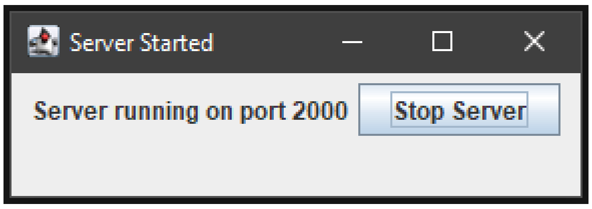
GUI for Server Configuration window

Here, you are able to fill out the port you wish to run the server on, (the host defaults to localhost). If you check the "Use Default Port (2000)" checkbox, the port inputfield is ignored and also disabled to further visualise its disregard.



Visual feedback when checking default checkbox option on Server setup

Finally, pressing the "Start Server" button will start the server, (if possible), and display the following new GUI.

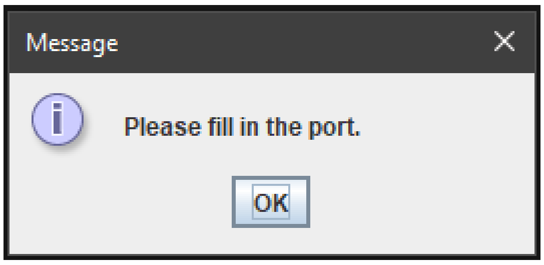
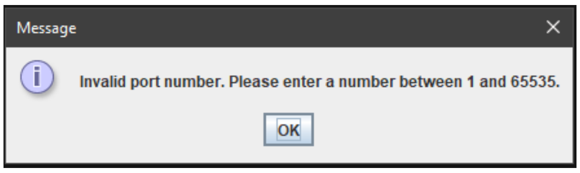


GUI visible after server has started.

This new GUI gives the user the possibility to stop the running server, but also shows the port the server is currently running on.

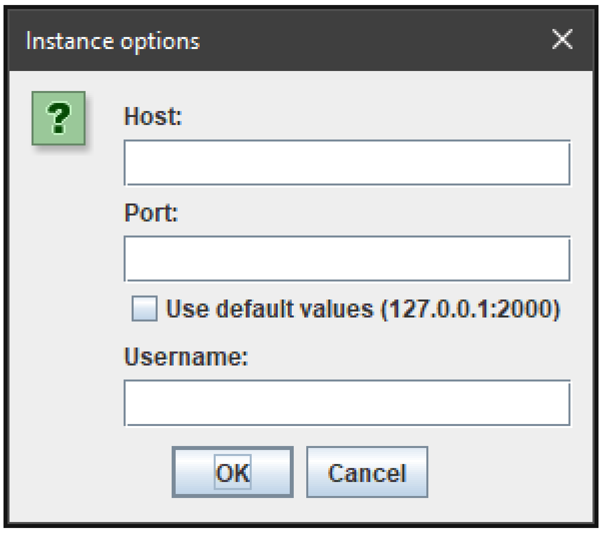
Error handling

The following are examples of how I handle errors/invalid inputs the user might come across at this stage.

 Message Please fill in the port.	 Message Invalid port number. Please enter a number between 1 and 65535.
Error displayed if user leaves port empty	*Error displayed if user's input is something other than a number between 1-65535*

Client.java

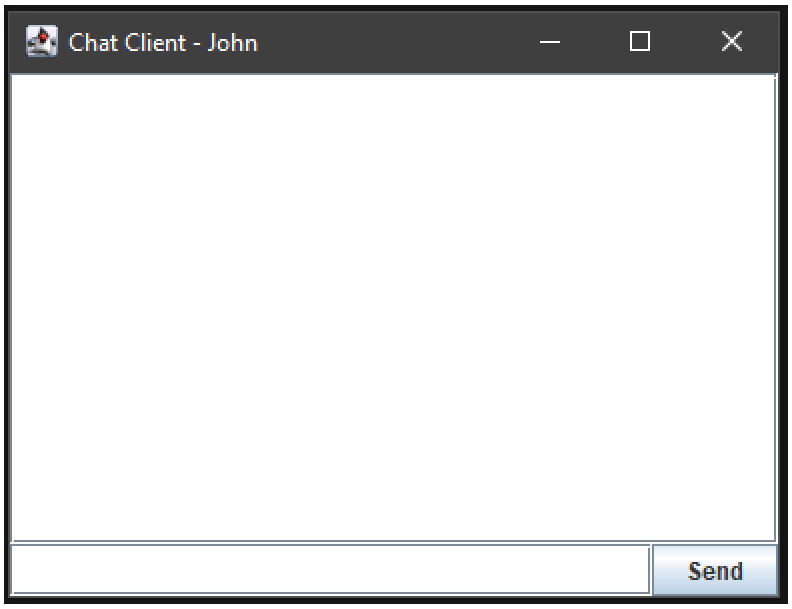
Moving on to the actual Client now, upon startup you are greeted with a configuration window.



The Client GUI that greets the user upon launching.

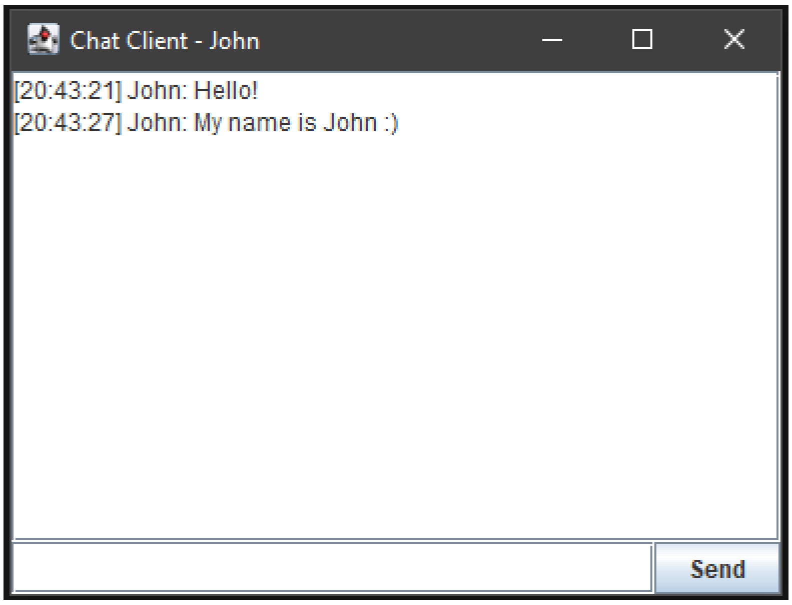
Here the user will find an input field for Host, Port and Username, as well as a default checkbox to default the Host and Port values, alongside an OK confirmation button, and a Cancel button to exit out.

Upon fulfilling the conditions for Host,Port and Username, you are once again greeted with a new UI that looks like this:



Chatroom UI

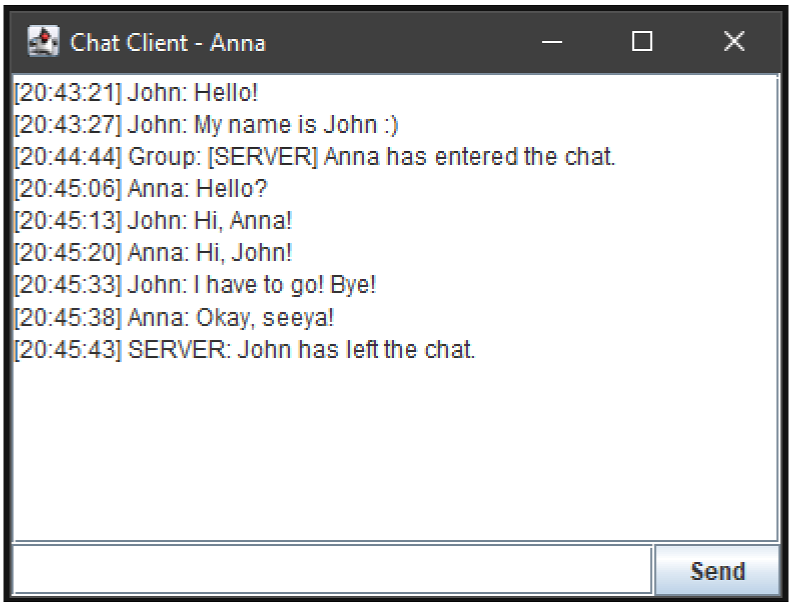
The chat-area would be filled in the case of a history for the client, though in the above example it is blank. Upon sending a few messages, the format becomes clear.



```
[20:43:21] John: Hello!
[20:43:27] John: My name is John :)
```

Chatroom UI with some chat messages

Now what's missing? Another user! If we add another user at this stage, it can help display how I handled users entering and leaving the chatroom.



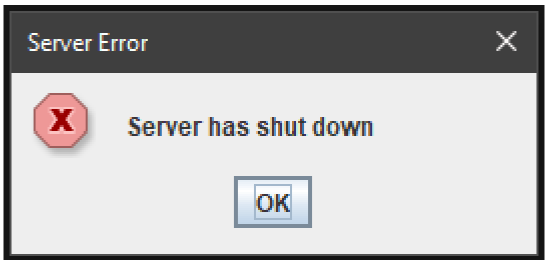
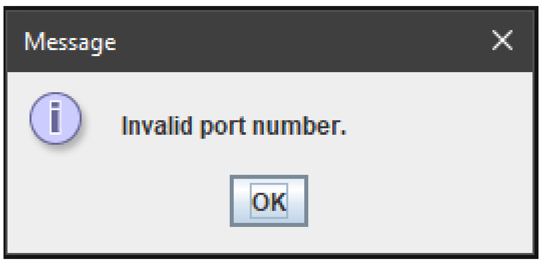
```
[20:43:21] John: Hello!
[20:43:27] John: My name is John :)
[20:44:44] Group: [SERVER] Anna has entered the chat.
[20:45:06] Anna: Hello?
[20:45:13] John: Hi, Anna!
[20:45:20] Anna: Hi, John!
[20:45:33] John: I have to go! Bye!
[20:45:38] Anna: Okay, seeya!
[20:45:43] SERVER: John has left the chat.
```

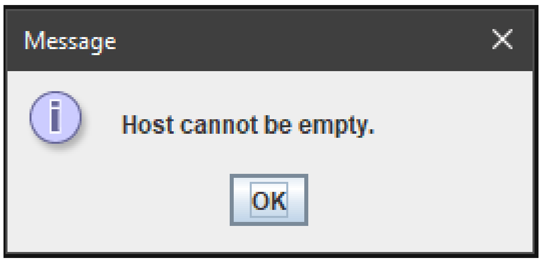
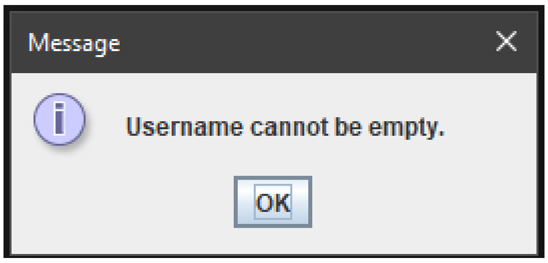
Chatroom UI with more messages, showcasing a simple conversation between two users

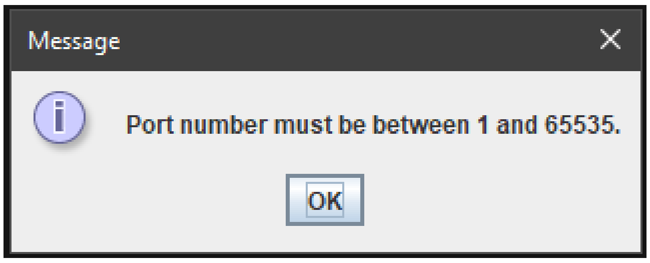
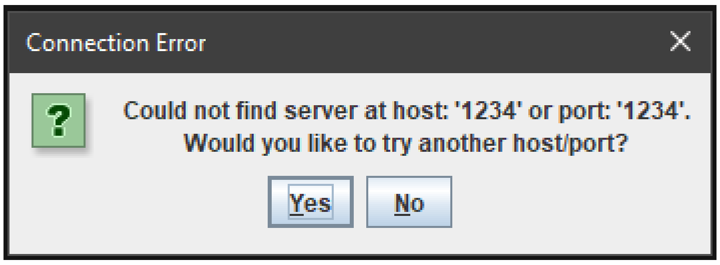
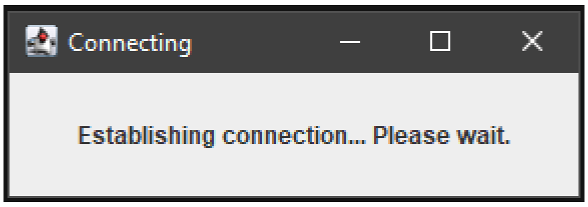
In this instance, the users 'John' and 'Anna' are seen conversating in the chatroom. Please bare in mind that since both of these clients were run off of my personal system for this showcase, the top 3 messages, timestamped "20:43:21", "20:43:27" and "20:44:44" respectively should not appear for the user 'Anna', as they would have the history loaded from *their own* client history.

Error Handling

I have taken some time to handle some errors that the user would come across at these stages as well.

 Server Error — Server has shut down — OK	 Message — Invalid port number. — OK
Error thrown when user is in a chatroom, but server is shut down	*Error given when the user inputs an invalid port number*

 Message Host cannot be empty. 	 Message Username cannot be empty.
Error given when the user leaves the Host field empty	*Error given when the user leaves the Username field empty*

 Message Port number must be between 1 and 65535.
Error given if the user inputs a number in the Port field that exceeds the required conditions
 Connection Error Could not find server at host: '1234' or port: '1234'. Would you like to try another host/port?
Prompt issued if the client is unable to find a server host at a specific Host or Port
 Connecting Establishing connection... Please wait.
Simply an information panel that gives the user feedback if the connection to the server is taking some time

That is all there is to my new and improved chatroom program! With its own sleek and simple UI, and great amount of error handling, I have attempted to create a user friendly and good-looking chat room experience that I hope is able to fulfill those expectations!