East London Science School

JarChat

OCR A-Level Computer Science Project

Adam Tazul

2022-05-07

adam_tazul@outlook.com

Signed by: adam_tazul@outlook.com

Date: 2022-05-07 14:50:16 BST

Contents

Contents 1
1: Analysis 2
1.1: Aims of JarChat 2
1.2: Target Audience 2
1.3: Research 2
1.3.1: Existing IRC Clients 2
1.3.2: Features of JarChat 4
1.3.3: Limitations of JarChat 4
1.4: Community input 4
1.5: Requirements 5
1.5.1: Software 5
1.5.2: Hardware 5
1.6: Success Criteria 5
2: Design 5
2.1: UI Design 6
2.1.1: Start-up Screen 6
3: Development 6
3.1: Connection to IRC 6
3.2: The code used in the library 6
3.3: Sending messages, joining/leaving channels 7
4: Testing, Review and Evaluation 8
4.1: Achieved Success Criteria 8
4.1.1: Proof of achieved Success Criteria 8
4.2: Limitations 8
4.2.1: Avoiding these limitations & Maintenance 9
5: Final Code 10
5.1: IRCMessageLoop.java 10
5.2: JarChat.java 12

1: Analysis

1.1: Aims of JarChat

JarChat will aim to be a cross-platform IRC client which will support Windows, MacOS, Linux, and maybe more Operating Systems not already listed. This will have the benefit of native cross-platform support as it is built-in to Java. Java's JDK version 8 (Java version 1.8) will be used to compile all binaries as most computers still run JRE version 8, even though Oracle has moved their LTS (Long Term Support) version to JRE 11.

1.2: Target Audience

I expect everyone who uses a computer to be able to take advantage of JarChat. While most services have moved their IM services to other platforms (Microsoft Teams, Skype, Discord, TeamSpeak, etc.), various FOSS (Free and Open-Sourced Software) still use IRC as their support protocol, taking advantage of IRC's decentralized nature.

JarChat can also expect to be popular with developers in any programming language, as there are still some programming help channels/servers hosted using IRC. The same will be true for new/existing users of most Linux distributions. Most of the time, if somebody has a question regarding Linux which can't be answered by Google, they can ask in the appropriate channel on Libera.chat.

1.3: Research

1.3.1: Existing IRC Clients

1.3.1.1: Windows

Existing Windows clients include mIRC and XChat.

1.3.1.1.1: mIRC

mIRC is a paid and proprietary software which costs £17.94 (though it offers a 30-day free trial). It provides a slightly dated User Interface and was compiled in 32-bit form. This makes it compatible with 32-and-64-bit versions of Windows from Windows XP up to Windows 11. The UI makes use of a template to store different tabs as windows on a canvas. These windows cannot be moved or used in other parts of Windows. The settings menu leaves much to be desired and leaves few customization options. There is no option for IRC's VOIP feature included with the software.

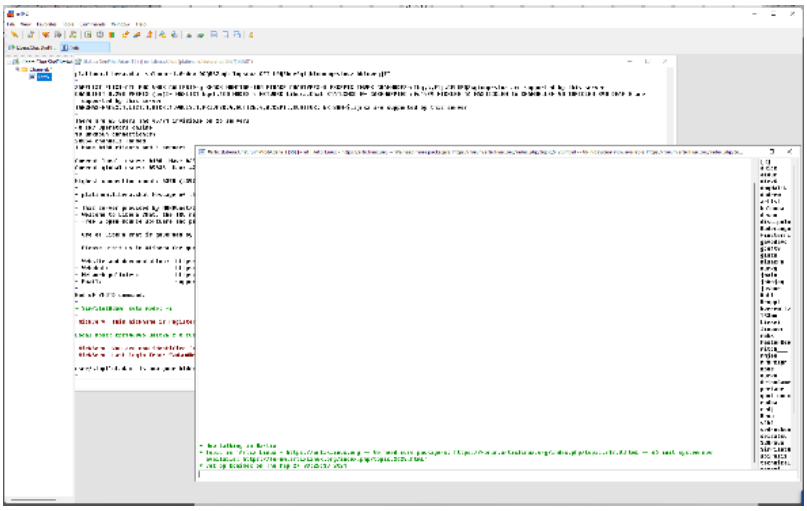


Figure 1: Default look of mIRC, running on Windows 11 build 22449.1000 (Dev Insider Preview).

I intend to take inspiration from mIRC's raw functionality as it works very well (albeit only on Windows). I do plan, however, to expand this functionality to take advantage of all of the features that are included on IRC's feature set.

1.3.1.1.2: XChat

XChat is another IRC client. Written in C, the Linux version is free and open-sourced under the GNU GPLv2 license. The client is precompiled for Windows and Fedora GNU/Linux, with forks available for Arch Linux (in the AUR), and with a precompiled *.deb file in the official Debian and Ubuntu repositories. The Windows version of the software is closed-sourced and proprietary, costing users US$19.99 (equivalent to £14.42 in September of 2021), though it offers a 30-day free trial. The Windows version officially supports all Windows versions from Windows 2000 up to Windows 10 (all in 32-bit).

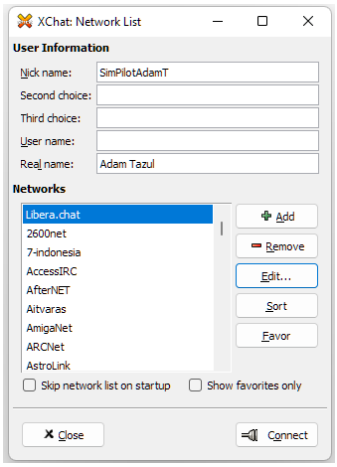


Figure 2: The first screen of the Windows version of XChat, running on Windows 11 build 22449.1000 (Dev Insider Preview).

I tried to use XChat to connect to my favorite IRC server, but the Windows version failed to do so on Windows Vista, 7, 8, 8.1, 10, and 11. This lack of ease of use is an issue that I would like to resolve with JarChat.

1.3.1.2: Linux

Existing Linux IRC clients include Konversation, WeeChat, and HexChat.

1.3.1.2.1: Konversation

Konversation is a Linux IRC client which is tightly integrated with the KDE Plasma Desktop Environment. It is fully free & open-sourced under the GNU GPLv2 license. As a part of the KDE Applications suite. Konversation is precompiled for all major distributions of Linux & GNU/Linux, with source code hosted on the official KDE git repository (https:/invent.kde.org/network/konversation).

I use Konversation myself frequently, as my main desktop setup is Artix Linux with KDE Plasma, and its tight integration means accurate and appropriate theming in line with the global QT5 theme, no matter what theme is used. It also uses Kcolorchooser to allow for the user to create a custom colour for Konversation's theme, even if the user is not using a QT5-based theming engine.

Using Konversation, I found that it is a fully-featured client with support for every part of IRC, displayed in a user-friendly layout that is easy-to-use. I intend to take inspiration from this and remake these features in JarChat.

1.3.1.2.2: WeeChat

WeeChat is a Terminal User Interface IRC client for Linux which can be used on any Linux Distribution without the need for any Desktop Environment or Window Manager. It is also FOSS under the GNU GPLv3 license.

While strictly TUI and Keyboard Interaction only (unless the setting is changed), WeeChat still provides a fully-featured experience with support for every part and situation possible in IRC, with every customization as needed by anyone who would use it. Since WeeChat is TUI, it is aimed at people who are willing to learn how to use it, which can take some time. For this very reason, there are a lot of people who have installed it, tried to use it, and then immediately uninstalled it as they found it extremely difficult to get started in (even with the existing documentation provided by the developers). This is completely understandable as WeeChat was designed with minimalism at the forefront of the developers' minds.

1.3.1.2.3: HexChat

HexChat is a cross-platform IRC client which supports all Linux Distributions via a Flatpak, its source code, .deb file, the official Arch Linux repositories, and more. It is also FOSS under the GNU GPLv2 license and features full support for Windows 7, 8, 8.1, 10, and 11. While it has Windows support, I am testing it on my install of Artix Linux (my setup will allow for testing with tiling window managers, floating window managers, Xorg, Wayland, Pulseaudio, and Pipewire).

1.3.1.3: MacOS

Even though there are one or two IRC clients for MacOS, I do not have the means to test this. Creating virtual machines with MacOS installed on them have proved to create bugs at best, and hackintoshing (installing MacOS on computers not made by Apple) is also problematic, with stringent hardware requirements I do not meet. In any case, trying to do either will result in me breaking Apple's EULA, which is legally binding.

1.3.2: Features of JarChat

JarChat is intended to include:

- Basic functionality as an IRC Client
- Cross-compatibility between various different operating systems without compromising on features
- Vast configurability through an easy-to-use UI
- GNU GPLv3 Open-Sourced license to allow for the community to better help shape the future of JarChat

1.3.3: Limitations of JarChat

The main limitation of JarChat is the fact that it will be written in Java, which means that JarChat will require and expect the user to have a JVM compatible with Java version 1.8.0 already installed on their system. While Java is generally installed on millions of computer systems globally, it cannot be guaranteed that everyone will be able to use JarChat for lack of the ability to install the software.

1.4: Community input

I am already taking input from the community about the project. Since all source code and development will remain public under the GNU GPLv3 license, it is relatively easy to get community input at every stage in the development cycle. To start things off, I posted an anonymous survey in the *#libera* chat in the most popular IRC server, libera.chat.

The questions in this survey include:

1. Roughly how many hours per week do you spend actively logged onto an IRC server?
2. Which servers do you frequent?
3. Which channels do you frequent in those servers?
4. What do you mainly do when logged into IRC?
5. What do you look for in an IRC client?
6. Which IRC client do you use currently?

The most popular answers are as follows:

1. Between 30 and 40 hours
2. Freenode (before the change of ownership), Libera.chat
3. Various Linux Distribution/software development support channels
4. Ask for/provide support from/to other users of the channel
5. Usability, theming, FOSS

6. A mix of Konversation, HexChat, and WeeChat

1.5: Requirements

1.5.1: Software

- Windows:
  - Vista SP2/Server 2008 R2 SP1 (or newer)
  - At least Internet Explorer 9, or Firefox (a dependency of Java)
- Linux:
  - Any distribution which has Java JRE version 8 in its repositories
  - Firefox (a dependency of Java)
- MacOS
  - MacOS X 10.8.3 or newer (not compatible with Apple Silicon)
  - Any 64-bit web browser (a dependency of Java)
- Java version 1.8 (OpenJDK JRE version 8 in most Linux repositories)

1.5.2: Hardware

- Windows & Linux:
  - Pentium 2 266MHz CPU or better
  - 128MB RAM or better
  - 124MB free secondary storage or better
- MacOS
  - An Intel CPU
  - 124MB free secondary storage or better

1.6: Success Criteria

In order to be successful, JarChat will need to:

- Function as a fully working IRC client
- Be user-friendly
- Be able to use SSL encryption for servers which use it
- Run on any OS that Java version 1.8.0 can be installed on

In order to check that each criterion is met, JarChat will need to

- Be tested in every aspect that is needed of any IRC client
- Be distributed to various users with various skill levels in computer usage
- Be tested on every OS JarChat is likely to be run on
- Be tested in connections with servers that use SSL encryption

2: Design

Most design will be happening after the core code of the program is written, as UI design takes time.

2.1: UI Design

2.1.1: Start-up Screen

On startup, I intend for JarChat to show the main window where server output will go, with a message in there if the user has not added any IRC servers. Alongside the main text area, there will be two other areas holding lists of connected servers/channels as well as active users in each channel.

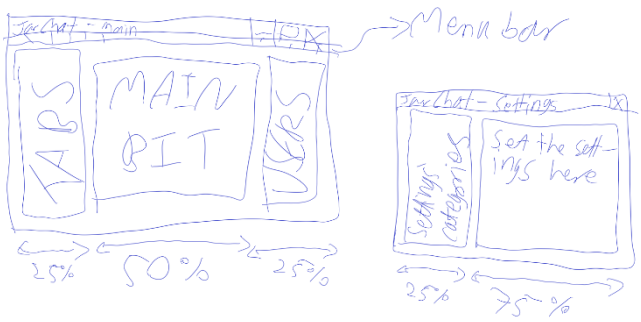


Figure 3: First (rough) sketch of how the UI may to look like.

3: Development

3.1: Connection to IRC

The actual connection to IRC is going to be handled by code I found on a GitHub Gist by Kaecy, which seems to handle the connection decently and more stable than others I have found. I have made some edits to the code provided by Kaecy, ensuring it is well-commented, has static methods in line with my main class (called JarChat), has support for secure TLS connections, and has unnecessary parts removed. The code in the provided gist was originally created to be used in a bot, so did not need many of these features I deem required by JarChat.

The IRCMessageLoop class provided by Kaecy is an abstract superclass which JarChat needs to slightly extend on in order to be able to set up any kind of connection. To test the connection, I started to write the program in CLI form, in order to quickly set the server/user information and establish the connection. Lines 38 and 39 of the source code (figure 10) as of the state of the code on 2022-05-02 at 23:17 currently create an instance of an object using the JarChat class, and then starts the connection. Using methods already declared in IRCMessageLoop, I was able to test and validate that the connection to the server was successul in this method.

```
1  /*
2   * JarChat IRC Client Source Code
3   * Free and Open-sourced under the GNU GPL v3 Licence
4   *
5   * Build using the latest JDK 8 to ensure compatibility with all
6   * modern devices. Will change JDK once more devices use JRE 11.
7   *
8   * Last Edited: 2022-05-05 16:47 by SimPilotAdamT
9   */
10
11 package com.AdamT;
12
13 // Imports
14 import java.net.InetAddress;
15 import java.net.UnknownHostException;
16 import java.util.Scanner;
17
18 // Main class
19 public class JarChat extends IRCMessageLoop {
20
21     JarChat(String server, int port) { super(server, port); }
```

Figure 4: The beginning of the main class file, showing the constructor of the JarChat class, as well as the necessary libraries being imported.

3.2: The code used in the library

The "library" used, as made by Kaecy, has several parts which have been edited by myself. For example, the extra code between lines 27 and 35 of IRCMessageLoop.java (figure 5) includes a custom implementation of the javax.net.ssl.SSLSocket and javax.net.ssl.SSLSocketFactory classes, in order to allow for most servers' main connection method. Lines 45 through 57 (Figure 6) were made compact to reduce the overall size of the source code file.

```
23     IRCMessageLoop(String serverName, int port) {
24         // Both outcomes of this if statement can throw exceptions, so need to be encased in a try-catch statement
25         try {
26             // These ports are exclusively used by encrypted TLS connections   |
27             if (port == 6697 || port == 7000 || port == 7070){
28                 // Initialise and start the secure socket
29                 SSLSocketFactory factory = (SSLSocketFactory)SSLSocketFactory.getDefault();
30                 SSLSocket server = (SSLSocket)factory.createSocket(serverName, port);
31                 server.startHandshake();
32                 // Allow the program to read everything being received from the socket, as well as to send info back to the server
33                 out = server.getOutputStream();
34                 stream = server.getInputStream();
35             }
36             // Any other ports are going to be plaintext connections, so do not need SSL Sockets
37             else {
38                 Socket server = new Socket(serverName, port); // Initialise plaintext connection (this is automatically started)
39                 // Allow the program to read everything being received from the socket, as well as to send info back to the server
40                 out = server.getOutputStream();
41                 stream = server.getInputStream();
42             }
43         } catch (Exception info) { info.printStackTrace(); } // Print information about the error to the terminal for debugging in case it's needed
44     }
```

Figure 5: The code which handles the actual Socket which connects to the IRC Server, with my own edits applied.

Figure 6: The refactored code of the IRCMessageLoop Class.

The processMessage() method in the IRCMessgeLoop class take in all messages being sent in through the socket and calls the appropriate method from figure 6 in order to process the input. The IRCMessageLoop class implements the Thread class and rewrites the run() method (figure 7) in order to allow for it to read all outputs from the IRC server, and then calls the processMessage() method for each message.

Figure 7: The rewritten run() method.

The Message and MessageBuffer classes (figure 8) are used so that there is a buffer to store messages in and to ensure that each message can be stored in its own object, setting each attribute. No constructor method is needed in the Message class because Java automatically handles this on creation of each object of type Message.

Figure 8: The Message and MessageBuffer classes.

The MessageParser class (figure 9, called by IRCMessageLoop.processMessage()) parses each message received from the server into a way better readable by IRCMessageLoop.processMessage(). It also slightly beautifies each message for when it gets put into STDOUT.

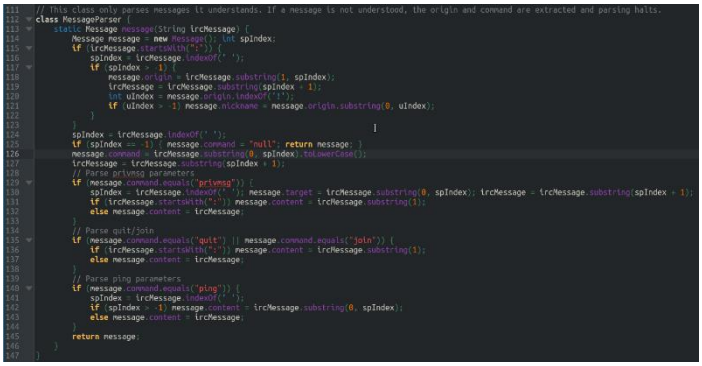


Figure 9: The MessageParser class.

3.3: Sending messages, joining/leaving channels

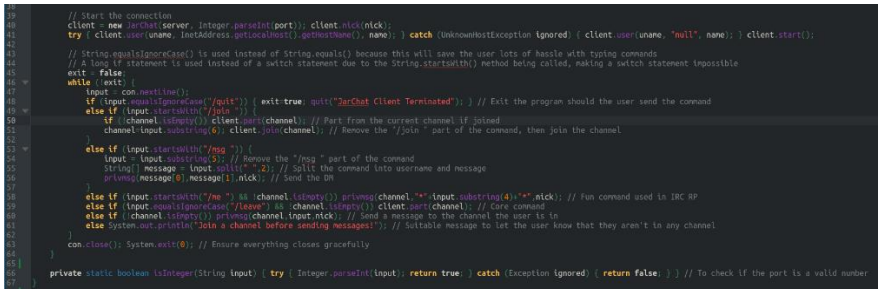


Figure 10: The part of the JarChat class which handles the connection and the user's inputs to STDIN

Lines 41 thru 60 of my code (figure 10) as of 2022-05-02 at 23:23 show a while loop which keeps going unless the program is exited (either by issuing a /quit command to STDIN, or by running into some kind of connection error). For every thing sent to STDIN, the long if statement (used instead of a switch statement due to a quirk in the way switch statements work) checks for valid messages or commands, with an appropriate error message for anything invalid. So far everything can only take place in a single IRC text channel. The user is able to send direct messages to several users at a time, but can only be connected to a single channel at any given time. I hope to be able to make use of the Thread class within a javax.swing GUI in order to allow for the user to send/receive messages to/from multiple channels at once.

4: Testing, Review and Evaluation

4.1: Achieved Success Criteria

So far, as of 2022-05-05 at 22:30, JarChat has achieved the following:

- Function as at least a semi-workable IRC Client
- Ability to use SSL encrypted connections
- Ability to be run on any given OS that has a JVM
- Ability to chat with multiple users directly at the same time

4.1.1: Proof of achieved Success Criteria

I have been able to connect to Libera.Chat's IRC server using JarChat in CLI (figures 11-12). The test shown here uses the TLS encrypted connection method and shows its success.

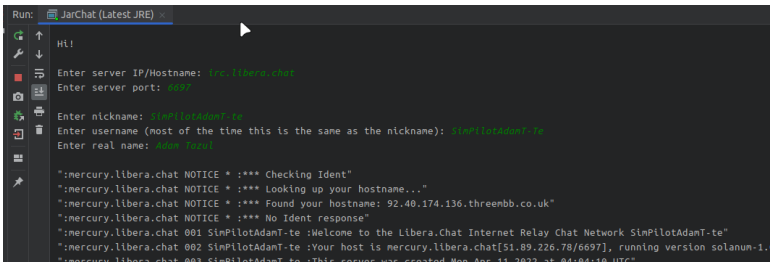


Figure 11: Collecting server/user info and connecting to the server. An IP address is shown but it is a mobile network one, not a DSL Broadband one.

Figure 12 shows the rest of the successful connection, with the login as managed by NickServ (a popular login bot used in IRC servers), with the password censored out.

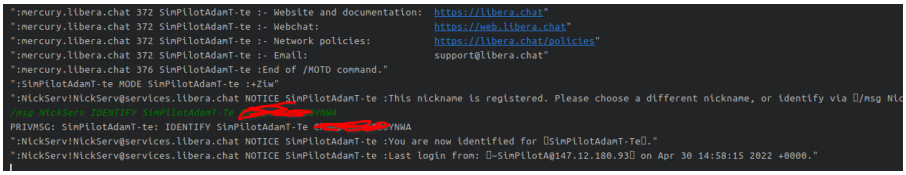
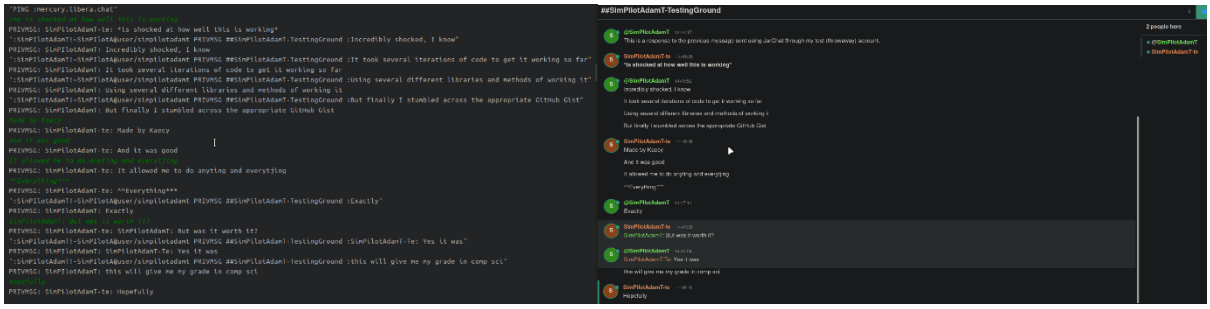


Figure 12: Successful connection, with login. Password is censored for security reasons.

Figures 13 & 14 show several messages being sent back and forth between my account on Jar Chat and my main account. Included there is a use of the /me command, which seems to fully work as intended.



Figures 13 & 14: Messages being sent back and forth in the test channel I created on Libera.Chat

4.2: Limitations

The most important parts of this project that are so far missing are the following:

- OP/Administrative commands
- javax.swing GUI
- Ability to join multiple channels
- Ability to join multiple servers

The lack of a GUI is a major limitation, as this is one of the goals I had in mind when I first thought of JarChat. This significantly reduces ease of use as JarChat in its current state expects the user to be able to open the command line to run the appropriate command.

The lack of OP/Administrator commands is also a fairly major limitation, since part of the IRC specification is being able to use these commands to moderate chats in channels where you have the rights to. This also allows server administrators to do everything they may need to do when signed in to the server to keep it running smoothly.

4.2.1: Avoiding these limitations & Maintenance

There is no surefire way accessible to the user to avoid them. JarChat is open-sourced and hosted on GitHub (https://github.com/SimPilotAdamT/JarChat/), under the GNU GPLv3 license, so anyone can have a look at the code.

The codebase for the project itself is very modular. In order to add a GUI all that there is needed to do is import the swing classes and build the GUI by hand. This will take tremendous amounts of time, and would ideally need a large team of people working on the project together, which is something which can be seen with Konversation (discussed in section 1.3.1.2.1), as well as HexChat (discussed in section 1.3.1.2.3).

Future maintenance should be simple to carry on, even if I abandon the project. The GPLv3 license and GitHub allow for other users to fork the project and make updates to the codebase to keep it up-to-date, fix any bugs that slip through the cracks, and add any new features. The codebase has been divided into two parts, each with their own source code file (see section 5).

I have annotated the code fairly well and anything that has no annotation is code which I believe is self-explanatory. I have kept all subroutine and variable names reasonable so anybody new looking at the codebase who understands Java's syntax should be able to decipher how the code functions.

Future versions will most likely finally incorporate the GUI I originally set out to create, but will likely incorporate any missing commands first. I do intend to keep maintaining this project for as long as I can, so all references to code in this document which have links to code in the GitHub repository by commit, so that references in here stay valid.

5: Final Code

5.1: IRCMessageLoop.java

```java
// All of these classes are taken from Kaecy's gist at https://gist.github.com/kaecy/286f8ad334aec3fcb588516feb727772,
// with my own edits to ensure they better suited for use as an actual client, as well as to add comments to the code.

package com.AdamT;

// Imports
import com.sun.istack.internal.Nullable;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import java.util.ArrayList;
import java.util.Arrays;

abstract class IRCMessageLoop extends Thread {
    // Variables (local to the class as many methods require them)
    static OutputStream out;
    ArrayList<String> channelList = new ArrayList<>();
    boolean initial_setup_status;
    InputStream stream;
    IRCMessageLoop(String serverName, int port) {
        // Both outcomes of this if statement can throw exceptions, so need to be encased in a try-catch statement
        try {
            // These ports are exclusively used by encrypted TLS connections
            if (port == 6697 || port == 7000 || port == 7070){
                // Initialise and start the secure socket
                SSLSocketFactory factory = (SSLSocketFactory)SSLSocketFactory.getDefault();
                SSLSocket server = (SSLSocket)factory.createSocket(serverName, port);
                server.startHandshake();
                // Allow the program to read everything being received from the socket, as well as to send info back to the server
                out = server.getOutputStream();
                stream = server.getInputStream();
            }
            // Any other ports are going to be plaintext connections, so do not need SSL Sockets
            else {
                Socket server = new Socket(serverName, port); // Initialise plaintext connection (this is automatically started)
                // Allow the program to read everything being received from the socket, as well as to send info back to the server
                out = server.getOutputStream();
                stream = server.getInputStream();
            }
```

~\JarChat\Client\src\com\AdamT\IRCMessageLoop.java [FORMAT=dos] [TYPE=JAVA] [POS=1,1][0%] [BUFFER=1] 06/05/2022 17:29:23

```java
        } catch (Exception info) { info.printStackTrace(); } // Print information about the error to the terminal for debugging in case it's needed
    }
    static void send(String text) {
        byte[] bytes = (text + "\r\n").getBytes(); // Ensure the message being sent ends with a CRLF line break and not na LF line break
        try { out.write(bytes); } catch (IOException info) { info.printStackTrace(); } // Try to send the message, and print out error info if it fails (without exiting the program)
    }
    void nick(String nickname) { String msg = "NICK " + nickname; send(msg); } // Set the nickname as seen by the server and other users
    // Set the rest of the user's info
    void user(String username, String hostname, String real_name) { String msg = "USER " + username + " " + hostname + " " + "null" +  " :" + real_name; send(msg); }
    void join(String channel) { if (!initial_setup_status) { channelList.add(channel); return; } String msg = "JOIN " + channel; send(msg); } // Join the channel as requested by the user
    void part(String channel) { String msg = "PART " + channel; send(msg); } // Leave the channel as requested by the user, without disconnecting from the server
    // Send messages, either directly or as a DM
    static void privmsg(String to, String text, @Nullable String from) { String msg = "PRIVMSG " + to + " :" + text; send(msg); System.out.println("PRIVMSG: " + from + ": " + text); }
    void pong(String server) { String msg = "PONG " + server; send(msg); } // Respond back to the server every few minutes to ensure the connection isn't forcibly removed
    static void quit(String reason) { String msg = "QUIT :Quit: " + reason; send(msg); } // Disconnect from the server as requested by the user
    void initial_setup() {
        initial_setup_status = true;
        for (String channel: channelList) { join(channel); } // Now you can join the channels. You need to wait for message 001 before you join a channel.
    }
    // Method to call the other methods here as required depending on the message received from the server or other users.
    void processMessage(String ircMessage) {
        Message msg = MessageParser.message(ircMessage);
        switch (msg.command) {
            case "privmsg": if (msg.content.equals("\001VERSION\001")) { privmsg(msg.nickname, "JarChat",null); return; } // Reflect this client's name in the response
            System.out.println("PRIVMSG: " + msg.nickname + ": " + msg.content); break; // Show the received message in the console log
            case "001": initial_setup(); break; // Initial message as sent by the server
            case "ping": pong(msg.content);break; // see comment on line 53
        }
    }

    public void run() {
        try {
            MessageBuffer messageBuffer = new MessageBuffer();
            byte[] buffer = new byte[512];
            int count;
            while (true) {
                count = stream.read(buffer);
                if (count == -1) break; // There is nothing being sent by the server, so no need to continue the loop
                messageBuffer.append(Arrays.copyOfRange(buffer, 0, count));
```

~\JarChat\Client\src\com\AdamT\IRCMessageLoop.java [FORMAT=dos] [TYPE=JAVA] [POS=76,1][51%] [BUFFER=1] 06/05/2022 17:30:32

```
                // Process every complete IRC message in the message buffer
                while (messageBuffer.hasCompleteMessage()) {
                    String ircMessage = messageBuffer.getNextMessage();
                    System.out.println("\"" + ircMessage + "\"");
                    processMessage(ircMessage);
                }
            }
        }
        catch (IOException info) { quit("error in IRCMessageLoop"); info.printStackTrace(); System.exit(1); } // We cannot continue execution, so we disconnect and halt execution
    }
}

// This class just sets all the attributes assigned to each message
class Message { public String origin; public String nickname; public String command; @SuppressWarnings("unused") public String target; public String content; }

// This class ensures the program can deal with every message one at a time
class MessageBuffer {
    String buffer;
    public MessageBuffer() { buffer = ""; } // At the start, there should be nothing in the buffer, data will be appended to this as the execution progresses
    public void append(byte[] bytes) { buffer += new String(bytes); }
    public boolean hasCompleteMessage() { return buffer.contains("\r\n"); } // The end of every complete message always has a CRLF line ending
    public String getNextMessage() {
        int index = buffer.indexOf("\r\n");
        String message = "";
        if (index > -1) { message = buffer.substring(0, index); buffer = buffer.substring(index + 2); }
        return message;
    }
}

// This class only parses messages it understands. If a message is not understood, the origin and command are extracted and parsing halts.
class MessageParser {
    static Message message(String ircMessage) {
        Message message = new Message(); int spIndex;
        if (ircMessage.startsWith(":")) {
            spIndex = ircMessage.indexOf(' ');
            if (spIndex > -1) {
                message.origin = ircMessage.substring(1, spIndex);
                ircMessage = ircMessage.substring(spIndex + 1);
                int uIndex = message.origin.indexOf('!');
                if (uIndex > -1) message.nickname = message.origin.substring(0, uIndex);
            }
~\JarChat\Client\src\com\AdamT\IRCMessageLoop.java [FORMAT=dos] [TYPE=JAVA] [POS=117,1][79%] [BUFFER=1] 06/05/2022 17:32:26
```

```
        if (index > -1) { message = buffer.substring(0, index); buffer = buffer.substring(index + 2); }
        return message;
    }
}

// This class only parses messages it understands. If a message is not understood, the origin and command are extracted and parsing halts.
class MessageParser {
    static Message message(String ircMessage) {
        Message message = new Message(); int spIndex;
        if (ircMessage.startsWith(":")) {
            spIndex = ircMessage.indexOf(' ');
            if (spIndex > -1) {
                message.origin = ircMessage.substring(1, spIndex);
                ircMessage = ircMessage.substring(spIndex + 1);
                int uIndex = message.origin.indexOf('!');
                if (uIndex > -1) message.nickname = message.origin.substring(0, uIndex);
            }
        }
        spIndex = ircMessage.indexOf(' ');
        if (spIndex == -1) { message.command = "null"; return message; }
        message.command = ircMessage.substring(0, spIndex).toLowerCase();
        ircMessage = ircMessage.substring(spIndex + 1);
        // Parse privmsg parameters
        if (message.command.equals("privmsg")) {
            spIndex = ircMessage.indexOf(' '); message.target = ircMessage.substring(0, spIndex); ircMessage = ircMessage.substring(spIndex + 1);
            if (ircMessage.startsWith(":")) message.content = ircMessage.substring(1);
            else message.content = ircMessage;
        }
        // Parse quit/join
        if (message.command.equals("quit") || message.command.equals("join")) {
            if (ircMessage.startsWith(":")) message.content = ircMessage.substring(1);
            else message.content = ircMessage;
        }
        // Parse ping parameters
        if (message.command.equals("ping")) {
            spIndex = ircMessage.indexOf(' ');
            if (spIndex > -1) message.content = ircMessage.substring(0, spIndex);
            else message.content = ircMessage;
        }
        return message;
    }
}
~\JarChat\Client\src\com\AdamT\IRCMessageLoop.java [FORMAT=dos] [TYPE=JAVA] [POS=146,1][99%] [BUFFER=1] 06/05/2022 17:33:23
```

5.2: JarChat.java

```java
/*
 * JarChat IRC Client Source Code
 * Free and Open-sourced under the GNU GPL v3 Licence
 *
 * Build using the latest JDK 8 to ensure compatibility with all
 * modern devices. Will change JDK once more devices use JRE 11.
 *
 * Last Edited: 2022-05-05 16:47 by SimPilotAdamT
 */

package com.AdamT;

// Imports
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Scanner;

// Main class
public class JarChat extends IRCMessageLoop {

    JarChat(String server, int port) { super(server, port); }

    public static void main(String[] args) {
        // Variables (local to this method instead of the class in case any variables with these names or similar are required elsewhere)
        boolean exit; String input; String channel = ""; JarChat client; boolean valid; Scanner con; String server; String port; String nick; String uname; String name;

        // Welcome and entering server details
        System.out.println("\nHi!"); con = new Scanner(System.in); System.out.print("\nEnter server IP/Hostname: ");
        server = con.nextLine(); System.out.print("Enter server port: "); port = con.nextLine();

        // Check the port is valid (most servers use a port with 4 digits
        valid = false;
        while(!valid) { if (isInteger(port) && port.length() == 4) valid=true; else { System.out.print("Error! Invalid port!\nEnter server port: "); port=con.nextLine(); } }

        // Get user's details
        System.out.print("\nEnter nickname: "); nick = con.nextLine(); System.out.print("Enter username (most of the time this is the same as the nickname): ");
        uname = con.nextLine(); System.out.print("Enter real name: "); name = con.nextLine(); System.out.print("\n");

        // Start the connection
        client = new JarChat(server, Integer.parseInt(port)); client.nick(nick);
        try { client.user(uname, InetAddress.getLocalHost().getHostName(), name); } catch (UnknownHostException ignored) { client.user(uname, "null", name); } client.start();

"~\JarChat\Client\src\com\AdamT\JarChat.java [FORMAT=dos] [TYPE=JAVA] [POS=1,1][1%] [BUFFER=1] 06/05/2022 17:35:24"
        // Welcome and entering server details
        System.out.println("\nHi!"); con = new Scanner(System.in); System.out.print("\nEnter server IP/Hostname: ");
        server = con.nextLine(); System.out.print("Enter server port: "); port = con.nextLine();

        // Check the port is valid (most servers use a port with 4 digits
        valid = false;
        while(!valid) { if (isInteger(port) && port.length() == 4) valid=true; else { System.out.print("Error! Invalid port!\nEnter server port: "); port=con.nextLine(); } }

        // Get user's details
        System.out.print("\nEnter nickname: "); nick = con.nextLine(); System.out.print("Enter username (most of the time this is the same as the nickname): ");
        uname = con.nextLine(); System.out.print("Enter real name: "); name = con.nextLine(); System.out.print("\n");

        // Start the connection
        client = new JarChat(server, Integer.parseInt(port)); client.nick(nick);
        try { client.user(uname, InetAddress.getLocalHost().getHostName(), name); } catch (UnknownHostException ignored) { client.user(uname, "null", name); } client.start();

        // String.equalsIgnoreCase() is used instead of String.equals() because this will save the user lots of hassle with typing commands
        // A long if statement is used instead of a switch statement due to the String.startsWith() method being called, making a switch statement impossible
        exit = false;
        while (!exit) {
            input = con.nextLine();
            if (input.equalsIgnoreCase("/quit")) { exit=true; quit("JarChat Client Terminated"); } // Exit the program should the user send the command
            else if (input.startsWith("/join ")) {
                if (!channel.isEmpty()) client.part(channel); // Part from the current channel if joined
                channel=input.substring(6); client.join(channel); // Remove the "/join " part of the command, then join the channel
            }
            else if (input.startsWith("/msg ")) {
                input = input.substring(5); // Remove the "/msg " part of the command
                String[] message = input.split(" ",2); // Split the command into username and message
                privmsg(message[0],message[1],nick); // Send the DM
            }
            else if (input.startsWith("/me ") && !channel.isEmpty()) privmsg(channel,"*"+input.substring(4)+"*",nick); // Fun command used in IRC RP
            else if (input.equalsIgnoreCase("/leave") && !channel.isEmpty()) client.part(channel); // Core command
            else if (!channel.isEmpty()) privmsg(channel,input,nick); // Send a message to the channel the user is in
            else System.out.println("Join a channel before sending messages!"); // Suitable message to let the user know that they aren't in any channel
        }
        con.close(); System.exit(0); // Ensure everything closes gracefully
    }

    private static boolean isInteger(String input) { try { Integer.parseInt(input); return true; } catch (Exception ignored) { return false; } } // To check if the port is a valid number
}
"~\JarChat\Client\src\com\AdamT\JarChat.java [FORMAT=dos] [TYPE=JAVA] [POS=55,1][82%] [BUFFER=1] 06/05/2022 17:36:39"
```