CS-4333/6333-Project-1

Overview

For this project you will have to implement a bidirectional Talk program, whose entry point is the Java class **Talk.java**. The program is intended to support two-party communication.

Your program should accept command-line options and run in any of the following modes (as well as report errors for invalid invocations).

`Talk –h [hostname | IPaddress] [–p portnumber]`

The program behaves as a client connecting to `[hostname | IPaddress]` on port `portnumber`. If a server is not available your program should exit with the message **"Client unable to communicate with server"**. Note: portnumber in this case refers to the server and not to the client.

`Talk –s [–p portnumber]`

The program behaves as a server listening for connections on port `portnumber`. If the port is not available for use, your program should exit with the message **"Server unable to listen on specified port"**.

`Talk –a [hostname | IPaddress] [–p portnumber]`

The program enters *auto* mode. When in auto mode, your program should start as a client attempting to communicate with `[hostname | IPaddress]` on port portnumber. If a server is not found, your program should detect this condition and start behaving as a server listening for connections on port `portnumber`.

`Talk –help`

The program prints your name and instructions on how to use your program.

Once a two-party connection has been established, messages received through the socket should be prepended with **[remote]** when displaying the message on the screen to differentiate it from messages that are typed locally.

In addition, the string `STATUS` is considered to be a keyword and will not be transmitted. If a user types STATUS your program should print information about the state of the connection (IP numbers, and remote and local ports).

e.g., `[STATUS] Client: 127.0.0.1:8080; Server: 127.0.0.1:12987`

Arguments in brackets are optional. In case a `hostname` or `IPaddress` is not provided their default value should be **localhost**. If a `portnumber` is not provided its default value should be **12987**. Once a connection is established you will have to obtain suitable stream objects such as a `BufferedReader` and a `PrintWriter` (from the socket) as your input and output streams respectively. The unit of data sent/received must be a line of text as returned by the `readLine()` method. Users of your program will be using the keyboard to type their messages and their screens to display incoming and outgoing messages.

Note: the `readLine()` method is a blocking operation and the call does not return until something is read from the input stream. This may result in undesired effects on your program such as not being able to

receive and display remote messages on your screen while you are typing. Your program must not suffer from this problem and your solution should be described in your report.

Assignment

Complete the implementation of **Talk.java**, **TalkClient.java**, and **TalkServer.java** according to the specification above.

Report

A brief report should be typed and turned in by the due date at class time. The report does not need to be long and should only describe your approach (and challenges) in solving the problem and any methods or variables of relevance in your class. The report should also include a **feedback** section about this project. **Please submit your report as a .DOCX or a .PDF to Harvey.**

Submission

The project is due by the end-of-day on the last Friday of the month.

*To submit your work, please review the **Commits** section below.*

Compiling and Testing Code

Your IDE should provide tools to compile your code. If you're unfamiliar with that process, you can research it online or ask. Most developers compile their code from command line using a shell script, such as a **Makefile** or build script (**build.sh**). I've provided build scripts for you in both *Powershell* and *Bash*. Refer to the following directions on how to use these scripts based on the terminal that you're using. If you're on Windows, please use Windows Subsystem for Linux (WSL), Git Bash, or Powershell, not Command Prompt.

Windows Users (Powershell)

- To compile and run your code: `./build.ps1`
- To test your code: `./build.ps1 test`
- To format your code: `./build.ps1 fmt`
- To sync your code: `./build.ps1 sync`
- To submit your code: `./build.ps1 submit`
- To remove class files: `./build.ps1 clean`

Windows Users (WSL, Git Bash), Mac and Linux Users

- To compile and run your code: `./build.sh`
- To test your code: `./build.sh test`
- To format your code: `./build.sh fmt`
- To sync your code: `./build.sh sync`
- To submit your code: `./build.sh submit`
- To remove class files: `./build.sh clean`

These scripts use the following commands.

- To compile a Java file: `javac -d target -cp lib/junit.jar:src <filepath>.java`

- To execute a Java file: `java -cp lib/junit.jar:target <package-path>.<filename>`
- To format a Java file: `java -jar lib/google-java-format.jar --replace --skip-javadoc-formatting <filepath>.java`
- To remove class files: `rm -r target/*`

Code Style

All code should follow the Google Java style guidelines. If you find anything in the code that does not follow the style guidelines, feel free to fix it, but you are not required to do so. Only your handwritten code will be evaluated for its style. You do not need to follow the style guidelines to the letter but egregious deviations from the style guidelines will be penalized. A submission that passes all test cases but does not use an appropriate style will not receive an A for the assignment.

For those using an IDE, such as Eclipse or VS Code, the IDE should provide a formatting tool. I've included the XML specification of the Google Java Style Guidelines at `.vscode/java-google-style.xml`. You can configure your IDE to use the provided XML as its formatting rules to format your code to the Google Java Style Guidelines, which are the industry standard for most languages.

If you're working from command-line, google-java-format is an open-source formatting tool that you can use to format your files. You can use the following commands to format your code depending on your terminal.

- `./build.ps1 fmt`
- `./build.sh fmt`

Commits

Commits should be made incrementally. Many commits are always better than few, and commits can always be squashed together later if there are too many. You should try to make a commit every time you've made tangible progress in the development of your code. At the very least, you should make a commit each time you've passed a new test case.

Every commit should have a commit message. The standard format for commit messages is to start with a verb in present-tense followed by a concise message (typically less than 50 characters) that summarizes the change that the commit is introducing to the repo. For example, "Updates README", "Implements Array", "Passes testGet".

Popular IDEs, such as Eclipse and VS Code, provide integrated Git tools. If you're on Windows, you can install Git Bash or Windows subsystem for Linux. If you're on Mac or Linux, you already have git installed.

To sync changes made from another device, use the following command.

- `git fetch origin main`

To push commits from command line, use the following commands.

- `git fetch origin main`
- `git add -A`
- `git commit -m "<your message goes here>"`
- `git push origin main`

You can also sync all changes and submit with the following commands depending on your terminal.

- `./build.ps1 submit`
- `./build.sh submit`