Problem Solving Session

- The remainder of today's class will comprise the ***problem solving session*** (***PSS***).
- Your instructor will divide you into ***teams of 3 or 4 students***.
- Each team will ***work together*** to solve the following problems over the course of ***20-30 minutes***.
  - You may work on paper, a white board, or digitally as determined by your instructor.
  - You will submit your solution by pushing it to GitHub before the end of class.
- Your instructor will go over the solution before the end of class.
- If there is any time remaining, you will begin work on your homework assignment.

Class participation is a significant part of your grade (20%). This includes in class activities and the problem solving session.

Your Course Assistants will grade your participation by verifying that you pushed your solutions before the end of the class period each day.

Problem Solving Team Members

Record the name of each of your problem solving team members here.

Do not forget to ***add every team member's name***! Your instructor (or course assistant) may or may not use this to determine whether or not you participated in the problem solving session.

Luke Demi
Logan Nickerson

tinychat

For this lab you will implement an instant messaging program that consists of a server to which multiple clients may connect simultaneously. Connected clients can execute a few simple commands including:

- Announce yourself to the server.
- Broadcast a chat message to all currently connected users.
- Quit/disconnect from the server.

Problem Solving 1

Design and implement a protocol using plain text (ASCII characters). For each message described, consider some or all of the following:

- Which user is the message from?
- Messages may have multiple parts; how will you tell where one part ends and the next begins?
- How will you know when you have reached the end of a message?

Client-to-Server Messages

Description	Message Format
User 'Jodi' connects to the server	“CONNECT Jodi\n”
User 'Joe' sends broadcast "Hi!" to server	“MESSAGE Joe Hi!\n”
User 'Janet' disconnects from the server	“DISCONNECT Janet\n”

Server-to-Client Messages

Description	Message Format
Server response to Jodi's connection request	“CONNECTED Jodi\n”
Server sends Joe's message to any connected client	“SENT Joe Hi!\n”
Server responds to Janet's disconnection	“DISCONNECTED Janet\n”

```
// extend Duplexer
public void userConnect(String username) {
      send(“CONNECT ” + username);
      receive();
}
```

```
String response = receive();
String[] tokens = response.split(“ “);
switch(tokens[0]) {
      case “CONNECT”:
            chat.userConnect(username);
            nextSlide(tokens[1]);
            break;
      default: //would be more
            send(“ERROR”);
            break;
}
```

Problem Solving 2

- Choose one of the messages from the ***Client-to-Server*** table of messages that you created in the last question and write the code required to send the message from the client to the server. You may assume that a socket between the client and server has already been established.
- For the same message, write the code required to read and parse the message on the server

Problem Solving 3

- Choose one message from the ***Server-to-Client*** table of messages that you created in the last question and write the code required to send the message from the server to the client.
- For the same message, write the code required to read and parse the message on the client

```
private void nextSlide(String username) {
      send(“CONNECTED “ + username);
}
```

```
String message = receive();
String[] tokens = message.split(“ “);
switch(tokens[0]) {
      case “CONNECTED”:
            user.showConnection(token[1]);
            break;
      default:
            break;
}
```

```
@Override
public void run() {

}

// naive approach: 4 different classes with 4
different run methods for: client sending
messages, client responding to the server,
server sending messages, server responding to
client
```

Problem Solving 4

Consider the fact that:

- The client sends messages to the server whenever a command is typed by the user, which may occur at any time.
- The server sends messages to the client in response to chat messages sent by other clients, which may also occur at any time.

Because of this, both ends of the connection between client and server must be prepared to receive messages asynchronously at any time without blocking other parts of the application. When those messages do arrive they will be handled differently by the client and server.

Together with your group, write the code that the client would use to continuously read incoming messages from the server. Consider the following questions while your team brainstorms:

- What kind of class will you need to create?
- How would you modify this class so that you could reuse as much of the code as possible on the server?
- What parts of the class would remain the same on the server, and what parts would change?