COMP250: Project 2 – Message Buffer Queue

**DUE DATE: End of day (11:59pm) Saturday February 26th. Standard late policy applies (-10% per day late, no later than end of day March 1st).**

# Summary

In this project, you will be using the **MessageSystem** from lab 1 as the backbone of a messaging system. It allows you to act as a user sending messages to other users (all locally, no networking), with those messages passing through **MessageQueues** one character at a time. This is the type of project that isn’t too difficult (in my opinion) to work out logically, but it has a lot of small moving parts that fit together to make the finished project.

# Specifications

This program has **4** individual code files: **MessageFragment** and **MessageQueue** (from lab 1), **MessageSystem** (the actual project logic), and **Main** (used as an entry point to the program). Details of the fields/methods for each will be explained when relevant. There is also a **users.txt** supplied on that contains usernames. You can run the program by running Main.java, which already has the required code to tell the MessageSystem to start running.

**I have supplied you with MessageSystem.java and Main.java with the basic structure in place. Your job will be to fill in the details of each method through the steps below. Also, for simplicity and saving space, note that I’ll often refer to MessageQueues as Queues below.**

# Step 1: Starting the Main Program (25 points)

Open **MessageSystem** and review its fields:

* **people (ArrayList<String>)** – holds all the usernames read in from **users.txt**.
* **queues (MessageQueue[])** – array of **MessageQueues** that will hold the 3 **MessageQueues** that will be in use.
* **queueMessages (String[]) –** This array will build up each queue’s message one character at a time as it passes through.
* **keyboard (Scanner) –** standard user input using **System.in**.
* **user (String)** – stores the name of the current user that is controlling the program.

First, as always, fill in the constructor for this class. The constructor should initialize **people** to be an empty **ArrayList**, initialize **queues** to contain 3 empty **MessageQueues**, initialize **queueMessages** to contain 3 empty **Strings**, initialize **keyboard** with a **new Scanner(System.in)**, and initialize **user** to **null**.

Next, scroll down to **findFile().** This method will populate **people** with the names from **users.txt**. You don’t need to use **split()** or anything – just read in each line from **users.txt** with **nextLine()** and store them in **people**. Remember to try-catch for **FileNotFoundException**! You should use **System.exit()** to end the program if this happens.

Now, scroll to **findPerson().** This method should ask the user for their name and validate it by comparing with **people** in the class. If the name they entered exists in **people**, return it. If not, return **null**.

# Step 2: Filling In run() (10 points)

Now let’s fill in **run()**. Begin **run()** by calling **findFile()**. Then, set up a **while** loop that runs for as long as **user == null**. Inside it, use **findPerson()** to ask the user to enter their name and validate it**.** Set **user** to be the value returned from **findPerson()**. Because **findPerson()** returns **null** if no match, this loop will repeat until a valid name is entered and matched.

I’ve given you the rest of **run()** – check the code’s comments for explanation of what it’s doing.

# Step 3: Updating the Queues (25 Points)

For this next method, I’ve given you the basic skeleton of it already, to make it easier to explain. The goal of **updateQueues()** is to check whether each queue is finished processing its message. The **for** loop allows you to use **i** to refer to each queue one at a time.

For each queue, it will check if there is a message still in it, (size being non-zero). For this, you’ll want to use **poll()** to get the next **MessageFragment** from it. Then, append its letter to the message-in-progress for this queue (stored in **queueMessages**). Here’s some pseudocode of this step:

*if(queues[i].size != 0) {  
 store a copy of queues[i].poll();  
 append its letter to queueMessages[i]  
}*

The other part of this if-else is saying **if(queues[i].size == 0 && queues[i].inUse == true)**. This condition is true when the queue has no more letters in it but has *not* been marked as available yet. In this case, you’ll want to print an alert to the screen that notifies the user that a message was successfully sent and print out that information. See below for an example of my output for this step. You can get the message content from **queueMessages[i]**, and the user information from **queues[i].sendReceive.**

Text

Description automatically generated

After the alert, call **clear()** on that queue so it’s marked as available and its information is reset. Also make sure to say **queueMessages[i] = “”** to reset the message-in-progress.

# Step 4: User Input Processing (35 points)

Now we need to fill in **menuInput()**. This is where the user enters a number from the menu.

**1 (Send Message):** For this, you should use the **findQueue()** method that I’ve already provided you with (it checks for the first available queue, returning **null** if there isn’t one). Then, ask the user for the message they wish to send, as well as the username of the recipient. You need to validate the recipient’s username using **findPerson()** just like with the initial user setup. Lastly, if all of that checks out, call **enqueue()** on that queue, sending in the message as well as the sender-receiver information.

**2 (Check Queue Status):** For this option, you should use a normal **for** loop (so that you can use the tracking variable) to loop over the 3 Queues. For each Queue, call their respective **getStatus()** method that you made back in step 2.

**3 (Skip Turn):** This option requires you to do literally nothing – you can leave the code as-is.

**4 (Exit):** This option only requires you to print something like “Thanks for using the messaging system!” and then use **System.exit()** to end the program execution.

# Submission & Grading

You should submit all relevant files in a .zip file (or similar) to Project 2’s submission page on Blackboard. Please name the .zip file using your first name and first initial of last name (for example, “SeanS\_Lab1”).

Grading is as shown in the previous headings (as well as 5 points for checking if the program compiles and runs correctly). There is also a grading breakdown text file attached on Blackboard. Starting here, I will be initially setting projects to be a complete/incomplete grade on Blackboard. This allows me to mark a submission as complete and give you feedback, but without me needing to do the full grading process until after the demo.

I’m very interested in feedback on if this project was too difficult, too easy, or just right. I have never designed a project before that involved me giving you existing code and requiring you to fill in the blanks, but I felt it was necessary here to make the problem not too overbearing.