CPSC 1181 - Assignment 3: Part 2

I will post a video discussing this assignment soon. You need to watch it to see the program
https://youtu.be/ULFVSzJEeJU

Submission:

- **This is the second part of a multi-part assignment.**
  - It builds upon what was asked for in Part 1.
  - The Part 2 submission will be graded as normal.
  - Penalties based on the first submission may be applied
  - Penalties for submitting late, not submitting in a ZIP file/etc. will be applied to your final submission
- Submit a **zip file** containing **only** the .java files to Brightspace prior to the **due date set in Brightspace**. Do not include class or other files
- Submissions that are less than 24 hours late receive a 1% per hour late penalty. Submission that are more than 24 hours late will not be accepted.
- You can use one of your 2 day extensions.
  - If you submit more than 24 hours late, one will be used automatically
  - **Add a comment to your submission in Brightspace IF**
    - You want to use an extension for a submission less than 24 hours late
    - You are submitting more than 24 hours late, but **don't** want to use an extension…you just want feedback.
- Submissions that are unzipped or that contain .class or other unneeded files will be penalized.
- **IMPORTANT: If you are unsure of your submission for any reason, submit it AND email it to me.**

Working With a Partner

- You must tell me **in-person** with your partner that you are working as partners to have permission. This must be done before the assignment is posted.
- If one partner no longer has an extension, neither partner can use an extension.
- Only one group submission is required: submitted by either partner
  - If both submit, we will mark whichever we open first
- **Add a second @author to your JavaDoc**

Notes

- **You don't need any new JUnit classes. Just include the one for Message**
  - **You don't need to add any test to it for getPreview**

Exercise 1

In the Message class add the method String getPreview(). It should return a String containing the status, the sender, and the beginning of the text for a message on one line. To do this, I'd suggest using a copy of the text and replace newline characters ("\n") with spaces. Then, if the result is too long, shorten it so that it appears on a single line when printed to the console.

Exercise 2

Create a class called SmileMessage that extends Message. It only needs a constructor that takes the sender and recipient usernames. It should always set the text so that it shows a smile. Something like…

```
" @  @ \n" +
"      \n" +
"@    @\n" +
" @  @ \n" +
"  @@  \n";
```

The Message constructor contains all of the needed logic, so this SmileMessage just needs to call it.

Override the getPreview method from Message. Instead of using the text (which will look like a mess on one line), replace it with an emoticon smile such as ":)"

Exercise 3
Create a class called Reply that extends Message. It needs one new data member of type Message which will store the message that this Reply is answering.

Create one constructor that takes the three strings the Message constructor needs **and** a parameter for the Message this one is replying to.

Override the toString method. It should start by using the Message class's toString. Then add a line similar to "----------Replying To----------". Finally, add on the toString for the Message data member so that the original message is included in the printout. When working this will allow a chain of replies to be created and have all included when toString is called.

Exercise 4
Add two new methods to your Messenger class.

Create `void sendSmile(String from, String to)` which creates a SmileMessage and adds it the the message ArrayList.

Create `void sendReply(Message original, String text)` which creates a Reply using the supplied information. The needed sender/recipient information is contained in the original message object. Add the Reply to the message ArrayList.

Create a getter for the user ArrayList in the normal way.

Exercise 6
Create a MessengerProgram class. This will be the main program. Start with this code...

```
4 public class MessengerProgram {
5
6     public static void main(String[] args) {
7         MessengerProgram prog = new MessengerProgram();
8         prog.execute();
9     }
10
11    public MessengerProgram() {
12    }
13
14    public void execute() {
15    }
16
17
18 }
```

Instead of creating a whole series of static methods, the main method creates a MessengerProgram object and calls the execute method. The execute method then does everything needed.

The constructor should initialize data members. Your data members may vary some from this, but I used

- A Scanner

- A Messenger (add some users, maybe send some messages)
- A String representing the active user

The execute method first has the user choose which user from the Messenger they are. (All options are presented in a menu, we're not worried about passwords/security here).

Then display a menu with the options seen in my example below. Create a basic program loop to make all of those options work. **See the video for a demo of the expected behavior/program flow.**
https://youtu.be/ULFVSzJEeJU

```
1: See All Messages
2: See Unread Messages
3: Send Message
4: Send Smile
5: Switch Active User
6: See Messenger Stats
7: Exit
Enter a number from the choices above: 9
Enter a number from the choices above: 9
Enter a number from the choices above: 7
User Chose: Exit
```

In all menus, if the user types a bad value, keep asking for another.

Organize your code with extra methods instead of dumping all of it into the constructor/execute methods.

Files to Turn In
Message.java MessageTest.java Messenger.java SmileMessage.java Reply.java MessengerProgram.java
MessengerTester.java from part 1 is not necessary.

Marking Rubric:

75 Marks

Style and Documentation: 10 marks
Message.java: 10 marks
MessageTest.java: 5 marks
SmileMessage.java: 3 marks
Reply.java: 5 marks
Messenger.java: 12 marks

MessengerProgram.java: 25 marks
Code Organized Into Methods: 5 marks