Mail Reader

Kartikay Kini & Thomas Shields

1 Introduction

This assignment will cover JavaFX and a few assorted other topics relating to collections.

For this assignment, things will be much more open ended. There are a few high-level requirements we ask you to satisfy, but otherwise it's mostly up to you. You should employ best practices and the concepts taught thus far in the course.

Since this is the first assignment like this, you can see the Tips section for some help. The extra credit homework will be like this as well, so we want to warm you up to it.

2 Problem Description

Your boss at Company Inc. has gotten very tired of using Gmail and Yahoo and would like a much simpler desktop application that they can use to just read, delete and sort their mail. They have tasked you with creating said application, and since you took CS1331 at Georgia Tech, you're an expert in JavaFX and have decided to write this simple app using it.

2.1 Software Requirements

2.1.1 Core Logic

The core logic of your program should have at least the following concepts:

- A person, with at least a name and an email. Persons should have some sort of natural ordering.
- A message, which has a sender, multiple recipients, a subject, a date/time, and a message body. By default, messages should be sorted by date. Since several messages could exist with lots of these attributes in common, equivalence of messages should take into account every attribute to avoid similar messages being marked equal.
- A mailbox, which has a name and a set of messages.

2.1.2 Backend

You need to write some sort of "server" that provides new messages randomly. This does not have to be complicated. I repeat: **THIS DOES NOT HAVE TO BE COMPLICATED.** Just write a very simple random generator that spits back fake email messages (see the message concept below). These can be real messages from your actual inbox, they can be total gibberish, whatever. It doesn't matter. Just make sure they're not ALL the same so that we can distinguish them while grading.

For example, in my solution I have a few random subjects and senders and I mix them up randomly.

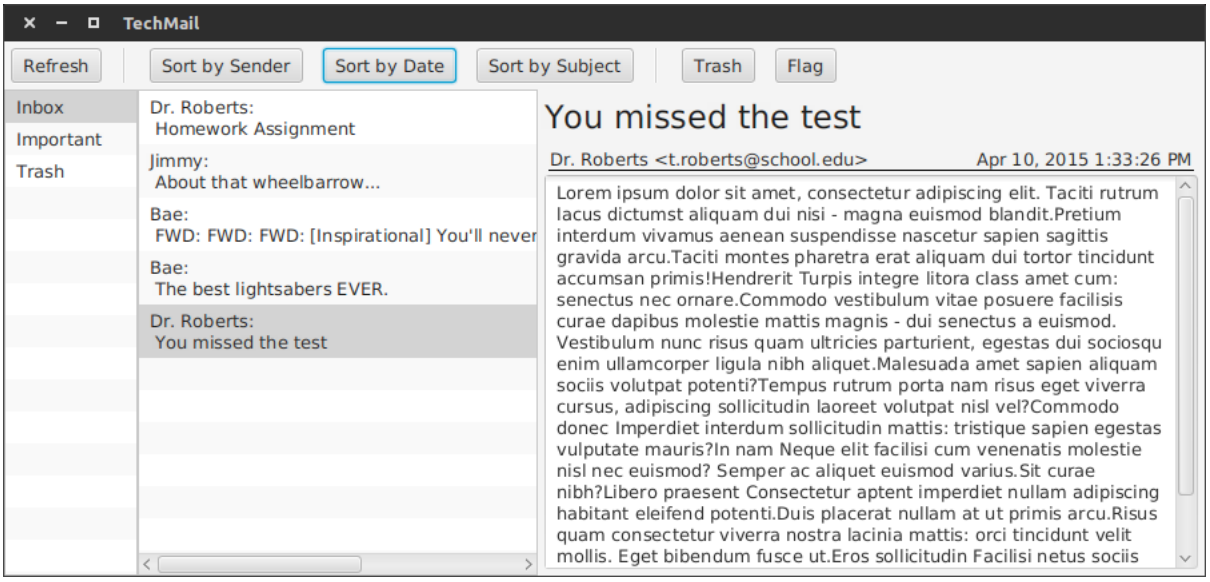
The Server should set the date/time for each message it returns to the current time.

2.1.3 MailReader functionality

- Should have at least three mailboxes: Inbox, Important, and Trash. Clicking a mailbox should display its messages.
- Refresh - should update the Inbox from the Server
- "Flag" an email to move it to the important mailbox
- "Trash" an email to move it to the trash
- Sort ANY of the mailboxes by sender, date, or subject. Obviously messages should be sorted by date by default.
- See contents of email by clicking on an individual email

2.2 GUI Design

The UI design is up to you for the most part. The only requirement is that it is able to do the above requirements and isnt so convoluted that the grading TA is unable to understand what to do. If you so desire you can include a README.txt file with your submission to explain how to access the functionality. Here is an example of what your GUI may look like (you do not have to match this perfectly):



3 Tips

Don't throw everything in one class. Use your Object-oriented programming skills to build a good, modular program.

One-liner hints:

- `LocalDateTime`
- Mailboxes: Sets, much?
- `Comparable`
- `BorderPane`
- Observable lists!

Post on Piazza if you want more hints. :)

4 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

   /**
    * Creates an awesome dog (NOT a dawg!)
    */
   public Dog(){
      ...
   }

   /**
    * This method takes in two ints and returns their sum
    * @param a first number
    * @param b second number
    * @return sum of a and b
    */
   public int add(int a, int b){
      ...
   }
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

4.1 Javadoc and Checkstyle

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

5 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **100** points.
Review the Style Guide and download the Checkstyle jar. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.1.jar *.java
Audit done. Errors (potential points off):
0
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **100** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

6 Turn-in Procedure

Submit all of the Java source files you modified and resources your program requires to run to T-Square. Do not submit any compiled bytecode (`.class` files) or the Checkstyle jar file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code through Checkstyle!

Verify the Success of Your Submission to T-Square
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.

(b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

(c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!