HW6: Email Client
CSE1102 Spring 2014
Jeffrey A. Meunier
University of Connecticut

1. Introduction

In this assignment you will write a Java program to model an email client. This is another way of saying an "email program". This email client will run from the console window, and it will not actually send or receive email.

Later we'll add a GUI to it and have it interact with a real mail server. Hypothetically, this email program will be able to send and receive email from your actual UConn / GMail email account.

2. Due Date

This project is due by midnight at the end of the day on **Sunday, April 19, 2015**. A penalty of 20% per day will be deducted from your grade, starting at 12:00:01am. See this link for additional information.

3. Value

This program is worth a maximum of 20 points. See the grading rubric at the end for a breakdown of the values of different parts of this project.

4. Objectives

The purpose of this assignment is to give you experience writing classes from a specification that's more vague than usual. You will be left to fill in more of the details.

Also you will get experience using the ArrayList class.

5. Background

5.1. ArrayList

Read about the **ArrayList** class in section 6.13, starting on page 496 in the Pearson Custom book.

An **ArrayList** is like an array, except that it is a class from which you make instances and call methods. The biggest benefit to using an **ArrayList** instead of a normal array is that an **ArrayList** instance will adjust its size automatically as you add elements to it.

There is some new syntax associated with using the **ArrayList** class. This is a *collection* or *container* class in Java, meaning that it's used to store other instances. Each collection class is parameterized over the type of instance you can store in it. Just like the type of an array that can store strings is **String[]**, the type of **ArrayList** that can store strings is **ArrayList<String>**. Whatever class type you want to store in the **ArrayList**, that type must appear within the angle brackets.

Here are some examples of the similarities between an array and an **ArrayList**.

Array	**ArrayList**
`String[] a;`	`ArrayList<String> a;`
`a = new String[5];`	`a = new ArrayList<String>();`
`a[0] = "Hello";`	`a.add("Hello");`
`a[1] = "World";`	`a.add("World");`
`String s = a[0];`	`String s = a.get(0);`

You may also use an **ArrayList** instance in an enhanced **for** loop:

```
ArrayList<String> a = new ArrayList<String>();
a.add(whatever); // do a few of these
for(String s: a)
  System.out.println(a);
```

I've used the example of an **ArrayList<String>**, but you may use any class type within the angle brackets like **ArrayList<Message>** or **ArrayList<Contact>**.

6. Assignment

Read through all of the subsections in this section before you start, to make sure you understand how you must proceed with the assignment.

6.1. Create a new project

Create a **Main** class with a **main** method. You don't have to put anything in the **main** method yet. I just want you to do this so that you can have a class in the default package before you do the next step.

In this assignment, the remainder of the classes that you create will go into a specific package called **client**. Create that package now. Right-click on the project in Eclipse, select **New**, then **Package**, and name it **client**.

6.2. Contact class

Summary

The **Contact** class represents people to whom you can send email. Contact instances will be stored in an address book.

Do this

Create this class in the **client** package. This class should have three fields (member variables):

1. An email address
2. A full name
3. A nick name

All of these should be **Strings**.

Provide two constructors: one full constructor (parameters for all three member variables), and one that has a parameter only for the email address. This second constructor will leave the other two fields null.

Add a **toString** method (and use the @Override annotation) that works like this:

```
Contact c1 = new Contact("jeffm@engr.uconn.edu");
```
`c2.toString()` should return `"<jeffm@engr.uconn.edu>"`.

```
Contact c2 = new Contact("jeffm@engr.uconn.edu", "Jeff Meunier", "jeff");
```
`c2.toString()` should return `"Jeff Meunier (jeff) <jeffm@engr.uconn.edu>"`.

Ensure that there are spaces between the sections in that string (i.e., after the full name and after the nick name).

6.3. AddressBook class

Summary

This class stores a collection of **Contact** instances.

Do this

1. Create this class and have a single member variable of type **ArrayList<Contact>**. Define the variable, don't just declare it (i.e., assign a new instance to it). I will refer to this as the *contact list*.

You do not need a constructor.

2. Write an **add** method that has a **Contact** parameter and adds the contact to the contact

list.

3. Write a **remove** method that has a **String** parameter, which is the nick name of the contact to remove. Remove that contact from the contact list. Have it return the contact that was removed, or **null** if there was no contact found.

Hint: Use the **search** method that you already wrote in order to find the contact, then remove that contact from the contact list. See the online documentation for **ArrayList** for how the **remove** method works in the **ArrayList** class.

Be sure that the **remove** method does not crash if you give it a nick name that does not exist in the list. Display a message that the nick name can't be found.

4. Write a **search** method that has a **String** parameter, which is a nick name to search for. The method must iterate over the contact list. If the nick name is found (use **.equals**), return that contact. If no contact is found, return **null**.

5. Write a **show** method that displays each **Contact** instance. It has no parameters and returns nothing. Display one contact per line, and number each line. Like this:

```
1. Jeff Meunier (jeff) <jeffm@engr.uconn.edu>
2. Bill Gates (money) <billg@microsoft.com>
3. Vladimir Putin (vman) <vlad@kremlin.ru>
```

Make sure that the numbers shown start at 1, not at 0.

[*Note: Some of you may understand that using an ArrayList here may not be the best way to implement an address book. Yes, you are correct, but we'll get to that later. We need to go one step at a time*.]

Testing
Add some statements to the **main** method that create two or three **Contact** instances and add them to an **AddressBook**. Then search for each address, and search for one that doesn't exist. Display the result of each search to see if each was retrieved correctly. Then remove one of them and show the list to be sure that it was removed correctly.

6.4. Message class

Summary
This class represents email messages.

Do this
1. Create this class. This class must have the following fields: two contacts representing

whom the message is from and whom it's to, two strings representing the subject and body of the message, and a **Date** (in **java.util**) that's the time that the message was created (you'll also use this field for the time that a message was *received* — that can be considered to be its creation time).

2. Write a constructor that has a parameter for each member variable except the **Date** variable. To initialize the **Date** variable, just create a new instance of **Date** without any arguments (that represents the exact time of day that the **Date** instance was created).

3. Write a **show** method that displays the message. It has no parameters and returns nothing. Here's an example of a correctly formatted message:

```
Date: 2015/04/12 17:16:33
From: Jeff Meunier (jeff) <jeffm@engr.uconn.edu>
Subj: a test message
This is a test of the Message class.
```

You will need to use a **SimpleDateFormat** instance in order to display the date the way that I show it. Google it.

4. Write a **toString** method that *returns* a one-line summary of the message, everything but the body, like this:

```
"FROM: Jeff Meunier (jeff) <jeffm@engr.uconn.edu>, TO: Bill Gates (money)
<billg@microsoft.com>, SUBJ: wasssup, DATE: 2015/04/12 17:16:33"
```

Testing
Add some statements to the **main** method to test all the methods in this class.

6.5. MailBox class

Summary
A mail box is where a group of related messages is stored. In this application you'll use a mail box to hold messages received, and another one for out-going messages that have not yet been delivered.

Do this
1. Create this class. Also give it a member variable that can hold a list of **Message** instances. When I say *list* I mean for you to use an **ArrayList**. (Yes, Java has linked lists, but we're not using them here.)

You do not need to write a constructor for this class.

2. Write an **add** method that lets you add a message to this mail box, where the message is a parameter. The method returns nothing. Just add the message to the list of messages.

3. Write a **count** method that returns the number of messages in this mail box. Have a look at the documentation for the **ArrayList** class to determine what method to use.

4. Write a **getMessage** method that has an integer parameter and returns the message at that index from the list, or returns **null** if that message does not exist. Recall that messages are numbered starting at 0.

5. Write a **remove** method that deletes a message from this mail box. There should be one parameter, which is the index number of the message to delete. Have it return the message that was removed, or null if there was no message at that location.

6. Write a **show** method that displays each message. It should work a lot like the show method, in that it shows each message on one line (or as close to it as possible), and numbers each message. Use the **Message.toString** method.

Make sure that the numbers shown start at 1, not at 0.

```
1. FROM: Jeff Meunier (jeff) <jeffm@engr.uconn.edu>, TO: Bill Gates (money)
<billg@microsoft.com>, SUBJ: wasssup, DATE: 2015/04/12 17:16:33
2. FROM: Bill Gates (money) <billg@microsoft.com>, TO: Jeff Meunier (jeff)
<jeffm@engr.uconn.edu>, SUBJ: hey dude, DATE: 2015/04/12 17:19:41
```

Testing
Add some statements to the **main** method to test all the methods in this class.

6.6. MailClient class

Summary
This class will hold the address book, an inbox, and an outbox. It also contains the contact information for the sender, meaning it keeps track of your account information.

Do this
1. Create this class with the following properties: a **Contact** instance that is for the user (meaning, *you*), an **AddressBook**, and two **Mailbox**es, one for in-coming mail and one for out-going mail.

2. Create a constructor that has a **Contact** parameter. This will be the contact information for the user. Make sure you create new instances of the address book and two mailboxes and store them in the member variables. You can do that either here in the constructor or where you declare the member variables.

Add the user's contact instance to the address book.

3. Write an **addToInBox** method that provides the ability to add a **Message** (parameter) to the inbox.

4. Write an **addToOutBox** method that provides the ability to add a **Message** (parameter) to the outbox.

5. Write **getInBox**, **getOutBox**, and **getMyAddress** accessors. The **getMyAddress** method returns the contact instance that was provided when the constructor was called.

6. Write a **searchForContact** method that takes a nick name (**String**) and returns that contact from the address book, or null if the nick name is not found. This is the same behavior as the **AddressBook.search** method, so you'd better just call that method.

Testing
Add some statements to the **main** method to test all the methods in this class.

6.7. CmdLoop class

Summary
This class will allow a user to interact with this mail client. The user will be able to enter simple commands to manage an address book, compose and read messages, and manage the inbox and outbox.

Do this
1. Create this class, and give it a constructor that has a **MailClient** parameter. Store this in an instance member variable.

2. Write a run method that repeats these steps indefinitely:
   1. It displays the prompt "Mail: ".
   2. On the same line, it lets the user enter a command that consists of a few characters.
   3. After the user enters the command, compare the command string to a few built-in commands (use the **.equals** method):
      - "q": This quits the loop, exiting the method.
      - "h": This displays a list of commands that are available to the user. Have this command call a method that displays all the commands — this method is not needed outside this class, so make it private.
   4. If the command that the user entered is not recognized, display a message.
   5. If no command at all was entered (i.e., empty string), then do not display anything.

Example:

```
Mail: x
x not understood, type h for help
```

```
Mail: h
h   Show this help menu
q   Quit
Mail:    ← I just hit Enter here without typing anything
Mail:    ← Same here
Mail: q
```

You may change the spacing between commands (like adding a blank line), or change the prompt style a bit if you want to.

6.8. Test the CmdLoop

Delete the statements in the **main** method and then add statements to do this:

1. Create a new **Contact** instance with your information in it. For example, my contact instance looks like this:
   `new Contact("jeffm@engr.uconn.edu", "Jeff Meunier", "jeff")`
2. Create a new **MailClient** instance. Give it your **Contact** instance.
3. Create a new **CmdLoop** instance.
4. Run it.

Make sure it works the same as what I show you in the previous section.

6.9. Add commands

For each of the following commands, add the check for the command string inside the **run** method, then have it call a specific private method in order to handle the command, just like you did for the *help* command.

Please use exactly the same command names as I've shown you here. Your TAs will be testing these commands.

Command: la
Description: List address book

If the user enters the string "la", then show all the entries in the address book. This should be an easy method to write. You already have a reference to a **MailClient** instance, and the **MailClient** instance contains an address book. You'll probably need to add a **getAddressBook** method to the **MailClient** class. Then be sure to call the **show** method that you wrote for the **AddressBook** class.

There should already be one **Contact** in the address book when you run the program.

Command: li
Description: List inbox

Works similar to the "la" command. You may need to add a getter to the **MailClient** class.

Command: lo
Description: List outbox

Almost the same as the "li" command.

Command: aa
Description: Add to address book

This method prompts the user to enter an email address, full name, and nick name. Then it creates a new **Contact** instance and stores it in the **AddressBook**. Like this (I've underlined the stuff that the user has typed):

```
Email: jeffm@engr.uconn.edu
Full name: Jeff Meunier
Nick name: jeff
```

The user will be expected to type all the information correctly the first time.

Command: da
Description: Delete from address book

Ask the user which nick name to delete, then delete that contact if it exists. If it does not exist, display a message indicating that the nick name can't be found.

You'll probably need to write a loop to do this: iterate through the contacts until you find the right one, then delete it.

Command: cm
Description: Compose message

This lets the user enter the following things:

1. Either an email address or a nick name — determine which it is by checking for an "at" sign @ in the string

2. A subject
3. A message (one line is sufficient)

If the user enters an email address, then just create a new **Contact** with just that email address. An email address is any string that contains an "at" sign: @ (yes, it's actually more complex than that, but keep this simple for now).

If the user did not enter a proper email address, consider it to be a nick name. Find that **Contact** instance in the address book. If the contact is found, display it and use it, otherwise ask the user all over again to enter a new name or email address.

The subject and message can just each be a single line.

After all the information is collected from the user, create a new **Message** instance from that information and place the message in the outbox. Don't forget you also need to use your own contact instance as the "from" contact in the message. You may need to add a getter to the **MailClient** class to help you with this.

Then use the "lo" command to be sure the message was placed in the outbox correctly.

Here's an example:

```
      Mail: cm
      To: mom
      Unknown recipient mom
      To: jeff ← yours will have your nickname, not mine
      Found Jeff Meunier (jeff) <jeffm@engr.uconn.edu>
      Subject: hi
      Message: This is the message.
      Mail: lo
      1. FROM:Jeff Meunier (jeff) <jeffm@engr.uconn.edu>, TO:Jeff Meunier
(jeff) <jeffm@engr.uconn.edu>, SUBJ:hi, DATE: 2015/04/14 19:30:57
      Mail:
```

Command: ro
Description: Read outbox message

If the outbox contains 1 or more messages, prompt the user for a message number. Display that message in full. Note that the user will enter a number 1 or greater, but the messages are numbered 0 and greater. You will need to handle this. If the user enters an incorrect number, display the message "Message number *n* can't be found." If there are no messages in the outbox, display the message "Outbox empty."

Command: do
Description: Delete from outbox

Ask the user which message number to delete, then delete that message. Handle the number in the same way as for the "ro" command: empty mailbox, or incorrect number.

Command: sr
Description: Send and receive

This command will eventually(™) connect to a real (or simulated) email server, send all messages in the outbox, and download new messages into the inbox.

Right now have it do this: Move each message from the outbox into the inbox.

Command: ri
Description: Read inbox message

This should work the same way as for the "ro" command, except for the inbox instead of the outbox.

Command: di
Description: Delete from inbox

Just like "do", except for the inbox.

6.10. Update the h command

Make sure you add all the commands and descriptions to the *help* command method.

7. Report

Create a Microsoft Word or OpenOffice Word file (or some other format that your TA agrees on -- ask him or her if you are not sure). Save the file with a .doc or .docx format.

At the beginning of the document include this information:

Project title goes here
CSE1102 Project *project-number*, *semester*
Your name goes here

The current date goes here
TA: *Your TA's name goes here*
Section: *Your section number goes here*
Instructor: Jeffrey A. Meunier

Be sure to replace the parts that are underlined above.

Now create the following three sections in your document.

1. **Introduction**
   In this section copy & paste the text from the introduction section of this assignment. (It's not plagiarism if you have permission to copy something. I give you permission.
2. **Output**
   Run your program and copy & paste the output from the command window here. Demonstrate that each of the commands works. You do not need to write anything.
3. **Source code**
   Copy & paste the contents of your Java file(s) here. You do not need to write anything.

8. Submission

Submit the following things things on HuskyCT:

1. Your exported Eclipse project.
2. The report document.

If for some reason you are not able to submit your files on HuskyCT, email your TA before the deadline. Attach your files to the email.

9. Grading Rubric

Your TA will grade your assignment based on these criteria:

- (2 points) The comment block at the beginning of the Main class file is correct.
- (10 points) The program displays the correct output and uses the correct calculations to generate the answers.
- (4 points) The program is formatted neatly.
- (4 points) The document contains all the correct information and is formatted neatly.

10. Getting help

Start your project early, because you will probably not be able to get help in the last few hours before the project is due.

- If you need help, send e-mail to your TA immediately.
- Go to the office hours for (preferably) your TA or any other TA. I suggest you seeing

your own TA first because your TA will know you better. However, don't let that stop you from seeing any TA for help.

- Send e-mail to Jeff.
- Go to Jeff's office hours.