HW7: Email Client 2

CSE1102 Spring 2015
Jeffrey A. Meunier
University of Connecticut

1. Introduction
In this assignment you will add file I/O to the email client project from last time. This will allow the program to save its state (meaning address book, inbox, and outbox) to three disk files, and if it is shut down and restarted later, it can load it state from the disk files.

This assignment does not describe how to get your email client to interact with a real mail server. Having you do that is sort of beyond the scope of this course (you'd be capable of doing it, but it's not a topic we need to cover). However, what I'll do is try to get that running and then just send you the code that you can plug right into your program to get it working.

2. Due Date
This project is due by midnight at the end of the day on **Friday, May 1, 2015**, but you may submit it by midnight at the end of the day on **Sunday, May 3, 2015** without a penalty. A penalty of 20% per day will be deducted from your grade starting at 12:00:01am on Monday May 4. 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
This assignment will give you experience using file I/O and handing exceptions.

5. Background
Exceptions are covered in chapter 9 in the book starting on page 651.

Reading data from a file is shown in chapter 9. There's a simple example shown on pages 655 - 656.

Writing to a file using the **PrintWriter** class is not covered in the book. Refer to the **52-Exceptions.pptx** lecture slides.

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 client.cmd package

Summary

You will take the command-handling methods that you wrote in the **CmdLoop** class and turn each of them into a separate class. These command classes will be placed into a **client.cmd** package. (This means that the **cmd** package is a sub-package of the **client** package. This is a good way to organize your projects, placing packages within packages.)

Do this

Right-click on the **client** package in your project, and select **New → Package**. For the package name, type **client.cmd**, then click **Finish**.

6.2. Create an ICommand interface

Summary

Each command that you place in the **client.cmd** package will contain a single **run** method that the **CmdLoop.run** method will call in order to invoke the command. This method will be specified by an ICommand interface.

It is common in Java to name interface files with an initial letter "I".

Do this

Create an interface called **ICommand** and place it in the **client.cmd** package.

In this class place this method declaration:

```
public void run(MailClient client);
```

6.3. Create a help command in the cmd package

Summary

This is the simplest command to add to the group of commands.

Do this

Create a new **Help** class in the **client.cmd** package. Have it implement the **ICommand** interface. Copy all the **System.out.print{ln}** statements that you used in the **CmdLoop** class to display the help information, and paste those lines in the **Help.run** method.

6.4. Create a command hashtable

Summary

You will change the **CmdLoop.run** method so that instead of using an **if/else** ladder to check for each command, you will place the commands in a hash table. When the user enters a command, the hash table will be checked for that command string, which, if found, will give you the correct command instance to run.

Do this

Create an instance variable in the **CmdLoop** class that is a hash table that maps **String**s to **ICommand**s. Here's the statement that I used:

```
Hashtable<String, ICommand> _commands = new Hashtable<String, ICommand>();
```

Make that a private member variable.

6.5. Add the help command to the hashtable

Summary

Here I will describe how to add the help command to the command hash table that you just created. You will need to do this to other commands, so you can refer to this section if you need to.

Do this

In the CmdLoop constructor, add a new table entry that maps the user command "h" to the Help command class. Here's the statement that I used:

```
_commands.put("h", new client.cmd.Help());
```

Notice that I used the fully-qualified class name **client.cmd.Help** instead of importing **client.cmd.Help** and then writing **new Help()**. It works the same either way. I just wanted to show you that you could do this.

6.6. Check for the user command

Summary

The **CmdLoop.run** method must now check the hash table to see if the user entered one of the commands in the hash table. Right now there's only one, the "h" command.

Do this

Remove the test for the "h" command in the **if/else** ladder and delete the **_help** method.

At the end of the ladder, you should have a clause that looks something like this:

```
else if(!cmdString.equals("")) {
  System.out.println(cmd + " not understood, type h for help");
}
```

Change it so that instead of just doing a **System.out.println**, it does this instead:

1. Retrieve the **ICommand** instance from the **_commands** hash table based on the command string that the user entered.
2. If that **ICommand** instance is **null**, then display the *command not understood* message.
3. Otherwise if it is not null, call the **run** method on that **ICommand** instance.

6.7. Test the help command

Run the program and test it to be sure that the "h" command still shows the help contents.

6.8. Move the remaining commands

Move the remaining commands except for the "q" (quit) command into the command hash table. Remove the tests for those commands in the **if/else** ladder, and delete all those command methods from the **CmdLoop** class. The final **if/else** statement should look something like this:

```
if(cmdString.equals("q")) {
  // causes loop to quit
}
else if(!cmdString.equals("")) {
  // bunch of stuff
}
```

6.9. Add a Contact.save method

Summary

In order for the program to save all its lists (contacts, inbox, outbox) to files, the instances in the lists must be saved to files. I usually like to work from the bottom up, meaning first I get the simplest part to work, and then I build on top of that.

This method will save some of the contact information to a **PrintStream**, either the nick name or the email address. This will be useful for when **Message** instances are stored into a disk file: this is because a **Message** is created from either a nick name or an email address, not a complete **Contact** instance.

Do this

This method returns nothing, and takes a **PrintStream** as an argument. This is the stream to which the contact information will be written.

Check to see if the nick name for this contact is in the address book. If it is, then write just the nick name to the file. Otherwise, write just the email address to the file.

Since a **Contact** does not normally have access to an address book, you will need to make this a parameter of the method.

6.10. Add a Contact.saveFull method

Summary

This method will actually write all the information for a contact to a file. This method is the one the **AddressBook** will use to store contacts to a file.

Do this

Write this method. It returns nothing and takes a **PrintStream** as an argument. It prints each field of the **Contact** to the print stream, one field per line.

6.11. Add an AddressBook.save method

Summary

This method saves every **Contact** in an **AddressBook** to a disk file in such a way that it can easily be read back by a **load** method that you'll write later.

Do this

This method returns nothing and has a **String** file name as a parameter.

Create a new **File** instance using the file name. Then create a new **PrintStream** instance using the **File** instance. Save each **Contact** instance (use **Contact.saveFull**) to the print stream.

Close the print stream when the loop completes.

An exception might be thrown when creating the **PrintStream**. Enclose all the statements in the body of this method in a **try** clause, and **catch** the exception that may be thrown. If an exception is caught, display the message "AddressBook unable to save to file " followed by the file name.

6.12. Create a save command

Summary

The **save** command will save all the mail client's information, the inbox, outbox, and address book, into a file. The program can then be exited and restarted, and the information can then be loaded back in using a **load** command that you will write later.

Do this
Create this command. Be sure to add the "sv" command to the command hash table.

Before you write statements in the run method, create a new symbolic constant in this class that represents the name of the file to save the contacts into. I did it like this:

```
public static final String CONTACTS = "Contacts.txt";
```

Now instead of using the string **"Contacts.txt"** when you need to refer to the name of the contacts file, use the word **CONTACTS**. This will become more useful when you create the **load** command later.

This command simply calls the **save** method on the address book. Use **CONTACTS** as the name of the file. Later you'll have it also save the inbox and outbox.

Test it
Check to see that the save command works. Run the program and then type "sv", then quit the program. Click once on the project folder in Eclipse and hit F5 to refresh the file list. You should see Contacts.txt file appear. Open it. You should see three lines with your contact information. The file probably has four lines in it, though, where the last line is empty.

6.13. Add a Message.save method

Summary
This method saves a single **Message** instance to a disk file in such a way that it can easily be read back by a **load** method that you'll write later.

Do this
This method takes a **PrintStream** as an argument and returns nothing.

Save the *from & to* contacts to the **PrintStream**. Use the **Contact.save** method to do this (remember that it's an instance method in the **Contact** class, even though I've written ***Contact.save***). Calling that method requires that you give it an address book as an argument, but a **Message** instance doesn't know anything about an address book. Add the address book as a parameter to this method so that you can use it when you save the contacts.

Then print the three remaining fields of the message to the print stream.

6.14. Add a Mailbox.save method

Summary

This method saves each message in a **Mailbox** to a disk file in such a way that it can easily be read back by a **load** method that you'll write later.

Do this

This method returns nothing and has a **String** file name as a parameter.

Create a new **File** instance using the file name. Then create a new **PrintStream** instance using the **File** instance. Save each **Message** instance to the print stream. You will need to tell the message what address book to use, so make the address book a parameter to this method because a **Mailbox** instance does not know anything about an address book.

Close the print stream when the loop completes.

An exception might be thrown when creating the **PrintStream**. Enclose all the statements in the body of this method in a **try** clause, and **catch** the exception that may be thrown. If an exception is caught, display the message "Mailbox unable to save to file " followed by the file name.

6.15. Modify the save command

Have the save command save the inbox and outbox. Create named constants for the file names of the inbox and outbox, and use these constants just like you did for the contacts file name.

6.16. Test the save command

Run the program: add a new contact or two, compose a new message, do a send/receive to move the message into the inbox, then compose another message so that the outbox has a message in it, then save everything.

In Eclipse, click on the project folder once and then hit F5 to refresh the file list. You should see the **Contacts.txt**, **Inbox.txt**, **Outbox.txt** (or whatever you named them) files appear near the bottom of the file list.

6.17. Add a Contact.loadFull method

Summary

This method will load data for a single contact from a scanner. It is assumed that this scanner is already attached to an open file instead of to the keyboard. Reading a contact's information from a file is identical to that for a keyboard, except you don't need to prompt the user for what to type in next because the information is already in the file, waiting to be read.

Each contact takes up three lines of the file, like this:

```
jeffm@engr.uconn.edu
Jeff Meunier
jeff
```

One complication with this method is that you don't want to create a new **Contact** instance until after you know that you have read all three lines from the file. That means that the **Contact** instance associated with the data in the file won't be created until the reading from the file completes, which means that this can't be an instance method.

Do this
Write this method. It should be a **static** method. It returns a **Contact**. Its parameter is the **Scanner** from which to read the three lines of data.

The method must work like this:
1. Read three lines from the file: one for email address, one for full name, and one for nick name.
2. Create a new instance from those three strings.
3. Return that new instance.

Note that this method creates a new instance even though it's not an instance method.

Put all those statements inside a **try** block. Below that, **catch** the exception that might happen (let Eclipse tell you what the exception is called, but don't let Eclipse correct the problem). If the exception is caught, return **null**.

6.18. Add an AddressBook.load method

Summary
This method loads all the contacts from the contacts file, creating a new **Contact** instance for each one.

Do this
Write this method. It returns nothing. Use only those parameters that are necessary. You can determine this as you write the method.

When reading contacts from the file, add all the contacts to a new **ArrayList**, not to the one stored in the member variable. If all the contacts load successfully, then replace the **ArrayList** in the member variable with this new one.

You will need to create a new **File** instance, then create a new **Scanner** from that file. Use that scanner when you call **Contact.loadFull**.

This method must read contacts from the file by calling the **Contact.loadFull** method until that method returns **null**. Each time it reads a contact that is *not* **null**, it must put that contact in the new contact list.

An exception might be thrown when creating the **Scanner**. Enclose all the statements in the body of this method in a **try** clause, and **catch** the exception that may be thrown. If an exception is caught, display the message "AddressBook unable to load from file " followed by the file name.

If the complete list of contacts was loaded without error, then store the new array list in the **_addresses** member variable.

6.19. Create a load command

Summary
The **load** command will load all the mail client's previously saved information.

Do This
Add a "ld" command. This command will cause the program to load the address book, outbox, and inbox.

Right now only the address book can be loaded (you haven't written methods to load a mailbox yet). Have it load the address book by calling the **AddressBook.load** method on the existing address book instance. You will need to tell it the file name to use, which is simply **Save.CONTACTS**. (Literally: type **Save.CONTACTS** for the file name. It'll work. You'll see.) Even though you're loading and not saving, the file name is the same, and this way the file name is stored in a single place. If you want to change the file name later, you need only change it in the **Save** class.

Test this method to be sure it works: delete an address from the address book, then use the load command. The address should be restored, as long as it has already been saved to the file.

6.20. Add a Contact.load method

Summary
Note that a message is stored in the disk file in this way (I've labeled each line in brackets, but the bracketed parts don't exist in the file):

[*from*]	`jeff`
[*to*]	`someone@somewhere.com`
[*subj*]	`hi`
[*text*]	`this is a test`
[*time*]	`Fri Apr 24 14:03:57 EDT 2015`

Thus, the **Contact.load** method will be used to convert the *from* and *to* lines into **Contact** instances.

Do this
Since a **Contact** instance can't be created until after the line is read from the file, and only if the line is read successfully from the file, then that means that this method can't be an instance method. It's the same issue that there was for the **Contact.loadFull** method.

This is a static method that returns a **Contact** and has a **Scanner** parameter.

Inside this method, read a single line from the scanner. If the line is not **null** and the length of the line is > 0, then do this:

- Determine if the line contains an email address, the same way you did for the compose message command. If it does, then just create a new **Contact** instance with that email address and return it from this method.
- If the line does not contain an email address, then assume that it's a nick name that can be found in the address book. Search the address book for that nick name (add it as a parameter to this method), and return that contact *even if it is null*.
- Otherwise if the line is **null** or the string has length 0, then just return **null**.

You will need to catch an exception. If an exception is caught, just return **null** without displaying a message.

I was able to use just a single **return** statement in this method. Can you figure out how to do that, too?

6.21. Add a Message.load method

Summary
This method loads an entire message from a disk file.

Do this
Have this method load the two contacts, the subject, the text, and the time from a scanner. This method is **static** and it returns an instance of **Message**. You'll need to determine what parameters to use.

Recall again how a message is stored in the file:

[*from*]	`jeff`

[*to*]	`someone@somewhere.com`
[*subj*]	`hi`
[*text*]	`this is a test`
[*time*]	`Fri Apr 24 14:03:57 EDT 2015`

After you load each contact, check to see if it's **null**. If it is, then immediately return **null** from this method.

Load the next three lines (subject, text, and time) as strings. The time will need to be converted into an actual **Date** instance. These next lines will do that conversion. Copy & paste into your method.

```
SimpleDateFormat sdf = new SimpleDateFormat("EEE MMM dd HH:mm:ss zzz yyyy");
Date date = null;
try {
  date = sdf.parse(dateString);
}
catch(ParseException e)
{}
```

After that, create a new **Message** instance from the information that you now have. Notice that the message's time stamp will be wrong: it will be set to the current time and not the time that's stored in the file. Now you must set the message's time stamp to be the **Date** instance that you just parsed. Do that by assigning the date directly to the message's member variable.

Don't forget to return the message.

6.22. Add a Mailbox.load method

Summary

This will work a lot like the **AddressBook.load** method works. This method loads all the messages from a file, creating a new **Message** instance for each one.

Do this

Write this method. It returns nothing. Use only those parameters that are necessary. You can determine this as you write the method.

When reading messages from the file, add all the messages to a new **ArrayList**, not to the one stored in the member variable. If all the messages load successfully, then replace the **ArrayList** in the member variable with this new one.

You will need to create a new **File** instance, then create a new **Scanner** from that file. Use that scanner when you call **Message.load**.

This method must read messages from the file by calling the **Message.load** method until that method returns **null**. Each time it reads a message that is *not* **null**, it must put that message in the new message list.

An exception might be thrown when creating the **Scanner**. Enclose all the statements in the body of this method in a **try** clause, and **catch** the exception that may be thrown. If an exception is caught, display the message "Mailbox unable to load from file " followed by the file name.

If the complete list of contacts was loaded without error, then store the new array list in the **_messages** member variable.

6.23. Modify the load command

Have the load command load the inbox and outbox after it loads the address book.

6.24. Test everything

Make sure all the commands work.

6.25. Export to runnable Jar file

Summary

Several of you have asked me if there's a way to distribute your Java projects as stand-alone programs without having to run them from Eclipse. This is how you do it.

Do this

1. Right click on the project folder in Eclipse.
2. Expand the **Java** folder in the window that pops up.
3. Choose **Runnable JAR file**.
4. Click the **Next** button.
5. For **Launch configuration**, expand the list and choose **Main - CSE1102-MailClient_2**, or whatever you named yours that looks similar to that. Be sure you export this second email client project and not the first one.
6. For **Export destination**, click the **Browse…** button and choose the desktop, and name the file **emailClient.jar**.
7. Click the **Finish** button.

You may be able to just double click on the jar file on your desktop to start up the program.

If that doesn't work, then do this:
Mac:

1. Open **Terminal**.
2. Type **cd Desktop**
3. Type **java -jar emailClient.jar**

Windows:

1. Open **Command Prompt**.
2. Type **cd Desktop**
3. Type **java -jar emailClient.jar**

Note that if you try to load the address book, inbox, and outbox files, it will look for them on the desktop. If you save files, it will put them on the desktop.

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. (but not the runnable jar file, if you created one)
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.