SUNY, Korea Computer Science

CSE 114 Handout 8: Problem Set 8

Due: Nov 22, 2019
Points: 20

Read This Right Away

This problem set is due at **11:59 pm on Nov 22, 2019, KST**. Don't go by the due date that you see on Blackboard because it is in EST. Go by the one given in this handout.

• To solve each problem below, you will be implementing a Java class.
• Please read carefully and follow the directions exactly for each problem. Files and classes should be named exactly as directed in the problem (including capitalization!) as this will help with grading.
• You should create your programs using emacs.
• Your programs should be formatted in a way that is readable. In other words, indent appropriately, use informative names for variables, etc. If you are uncertain about what a readable style is, see the examples from class and textbook as a starting point for a reasonable coding style.
• This problem set assumes you have installed Java and emacs on your computer. Please see the course web for installation instructions.
• You must use the command-line interface to run your programs. That is, you must use the `javac` and `java` commands to do this problem set. Do not use Eclipse yet.
• **Important!** You must use only 1 Scanner object. Assignments may require a Scanner object to be passed into other methods but do not create a new Scanner inside those methods! Use the one passed to the method. ***Creating multiple scanners causes a problem with the automated grading procedure and doing so will result in a loss of points on the assignment!***
• **Be sure to include** a comment at the top of each file submitted that gives **your name** and **email address**.
• **Remember:** We are now also assessing coding style. Be sure to use consistent indentation, good and informative variable names and write comments in your code to describe what it is doing.

What Java Features to Use

For this assignment, you are *not* allowed to use more advanced features in Java than what we have studied in class so far.

Introduction

Using Account.java in a recent lecture, we learned how to use static fields (e.g., idgenerator and sumBalance) along with their associated static methods (e.g., getNewId() and getSumBalance()). That design made sense since we wanted to keep track of that kind of information for a single bank. A similar situation is being dealt with in Problem 1 of this problem set. What if you want to keep track of that kind of information for multiple banks? Then, that design we used in Account.java would not be expressive enough. We will instead have to rely on an additional class such as Bank which has the capabilities of keeping track of that kind of information–one instance of Bank keeping track of information for one bank and another instance for another bank, etc. In that case the fields and their associated methods will be dynamic, not static, right? Well, that is the kind of design that is being dealt with in Problem 2 of this problem set (although we are using mailboxes instead of banks). So, I suggest that you try Problem 1 first and then Problem 2 in that order, while trying to understand the differences between the two situations.

Pair Programming: You are welcome to do pair programming for this problem set. I am allowing pair programming for this problem set because it will allow you to work with one other person so that you both can discuss your decisions along the way, i.e., learn from each other. By pair programming I mean you work with one other student in the class. Copying any piece of code done by your partner alone is not pair programming. In pair programming two programmers sit in front of a single keyboard/monitor and start programming, together. At any point in time during the entire programming process both are paying attention to the task at hand and whoever wants to type grabs the keyboard and types, ideally taking turns. Two people are acting as if there is only one person programming. When you submit, both partners may submit the same files. *Just add a comment at the top of each file stating whom you worked with.* If you violate any of the guidelines given here, you are cheating. If you choose to do it alone, that is fine too.

Problem 1 (10 Points)

First, read the Introduction section above before you read this problem.

Create a class named Ipad in a file named `Ipad.java`. As usual you should also create another class named UseIpad in a file named `UseIpad.java.`

In this problem, you will be using static fields and methods in addition to non-static (i.e., dynamic) fields and methods in Ipad.

To finish this problem you will do the following:

- Write a piece of code in the main of UseIpad to test each aspect of the equirements I describe below. As usual please do incremental development: build a little bit of Ipad and test that much in UseIpad before you add more. Repeat this process until you are done.

- An iPad should contain at least the following attributes: name, price, display size (diagonal length in inches), memory capacity (in giga bytes), and whether or not it has built-in cellular network capability. I want you to add at least one additional attribute that I did not mention. I want you to be careful in selecting the type of each field as you design your class: int, double, boolean, String, etc. Make reasonable decisions.

- Add appropriate constructor(s), getters, and setters. Don't blindly add a getter or a setter for each field. Think about each attribute and decide whether to provide a getter and/or setter or none.

- Each iPad that is produced, i.e., each iPad object that gets created, must also have a serial number. To assign a unique serial number to each Ipad object, you need to come up with a way to generate a unique serial number. Include that capability in your Ipad class, and use it appropriately. (Hints: see how I generated account numbers in Account.java)

- Your Ipad class should also be capable of remembering all the iPad objects that have ever been created so far, and manage them, using an array of Ipad objects. Use a static field inside Ipad class for that array. (Using a non-static field for this would not make sense, would it?) Assume that the number of iPad objects that will ever be created will never exceed a small number, say 20, to make your testing easier. Or, you may even use a smaller number like 5 for ease of testing initially and change it to a large number, say 20, after debugging is done.

  Now, add a method named *remove()* to iPad class that we can call to remove an Ipad object from that array. When you remove one iPad object from a location in the array, you don't want to leave a hole in the middle of the array somewhere. I suggest you fill the hole by shifting everything beyond the hole to the left by one position. You will need to set the location where the last one was in to null.

  In addition to the size of the array you also need to know how many of the 20 (or smaller if you used a smaller number) positions in the array have been filled so far. This could be another static field that gets updated every time an Ipad object is added to or removed from the array, right?. If you have n positions filled in so far, I assume those n iPad objects are occupying the locations 0 through n−1, and the locations from n to the last index of the array are filled with a null value.

  Should the *remove* method be static or dynamic? If it is static method, it will most likely take an argument of type Ipad. If it is a dynamic method on the other hand, it will not take any argument. Right? I like the idea of making it a static method myself in this particular situation.

  Also add a static method named *add()* that you can call to add an Ipad object to the array. Here you will need to make a decision on how this add method is going to be used. Two possibilities: (1) When you create a new Ipad object, you may add the new object being created blindly into the array, in which case you will call the add method to add it from within a constructor. Or, (2) don’t add it blindly from within a constructor, but add one only if there is an explicit add request from use code, i.e., most likely the main method of UseIpad by calling the add method after an iPad object has been created. In the second case, an object is created in the main of UseIpad. The iPad object has not been added to the array yet at this point. Now, the main calls the add method in Ipad class to add the iPad object into the array.

Depending on which design you choose, you will decide to make the add method public or private. I am inclined to suggest that you try the first option. The reason I am not insisting on one over the other is that one could argue for either one and I want you to think about pros and cons of both approaches.

- Add another attribute to Ipad class for the number of apps that have been installed in each iPad object. Also add a method called *installApp()* to your Ipad class. This method will simply update the count (the number of apps that have been installed so far) without doing anything else such as remembering what apps have been installed. So, you can call the function to install an app without actually passing an app to install. Also add a getter method for the attribute.

- Add a method to Ipad class that can be called to find out about the total number of apps that have been installed in all the iPad objects in the entire system, namely in the array of Ipad objects. Should this be a static or dynamic method? Make the right decision.

- Hints: You will need to be careful as to what needs to be static and what needs to be non-static, i.e., dynamic; what needs to be private and what needs to be public. In my specification, I intentionally left some of these things open so that you can think about them and make reasonable decisions yourself.

Note that `UseIpad.java` is given along with a sample output generated by my solution. Your solution should produce the same output. Since you will be adding at least one more attribute, your output may be slightly different. I suggest that you add another constructor with additional attribute(s) rather than changing the one I assumed. That way my code in `UseIpad.java` would still work, which is important because it will make grading much easier.

The output file is given in `UseIpad output.txt`.

Suggestion: Work incrementally. That is, comment out everything inside the main and uncomment one line at a time and make it work before you uncomment another line until you are done.

Include `Ipad.java` and your `UseIpad.java` (even though we will grade with our `UseIpad.java` file) into the zip or tar file that you hand in.

Problem 2 (10 Points)

First, read the **Introduction** section above before you read this problem. In Problem Set 7 you designed and implemented a DataMessage class. This time, let’s design and implement a Mailbox class in a file named `Mailbox.java`. This class will hold collections of data messages delivered to a 'collection station' for weather data. Do the following with this class:

- You may use the DataMessage class from PS 7. You will have fix any bugs you had in your submission and you will need to add new features to this class as you work through this problem. **I have highlighted the modifications needed to DataMessage using LucidaSans bold.**

- Suppose there are multiple mail boxes. We will create an instance (object) of Mailbox to represent each of the mail boxes, for example, an instance for the Seoul mail box, another for Incheon, and finally, one for Busan. You may create as many mail box objects as you wish in your program. Your UseMailbox class must create at least two Mailbox objects and use them as you test your implementation. I provide a UseMailbox class that you can use to test your code. On submission, we will use our UseMailbox to test and grade.

- A mail box should maintain the following information:
  - It should have a name.

  - A mail box consists of two boxes: inbox and delbox. A data message belongs to the inbox or delbox but not in both inside a given mail box object. The inbox contains all the received messages and the delbox contains all the deleted messages from the inbox. So, a mail box object should maintain two arrays: one for its inbox and the other for its delbox. Each mail box object should also remember how many messages it has in its inbox and how many in its delbox. Let’s assume that a message will forever reside in either the inbox or delbox.

  - You may assume that the capacity of an inbox is arbitrarily large, e.g., 50. Similarly for a delbox. However, you may initially set it to a small number, say 5, for ease of debugging and then set it to a large number when you feel that your program is fully debugged.

- A data message must have a unique id number represented as an integer assigned to it at the time the message is added to a mailbox. **However, you will need to embed an integer messageID field into the DataMessage object. Add the messageID to DataMessage as a private integer member. Provide a getter and setter method for it. Do NOT change the constructors for DataMessage. Simply add code to set MessageID to -1.** The id that is assigned to the message when it arrives at a Mailbox will remain with the data message forever.

- **There are a few more changes to make to your DataMessage class.**
  - **Change *toString()* to add "MsgId: " plus the message id as well as the Source: and Date: lines before the serialized data (as is done in *transmit()*.) So instead of just calling *serialize()*, it should build a String with it's message id field, name field and date field. Then add the string from *serialize()* to those items.**
  - **Add an additional public dynamic method called *transmit()*. This one will take one argument that is a target Mailbox. It should take the actual Mailbox object, not a name (i.e., String)! This *transmit()* routine will, instead of printing the message, call the Mailbox's *receive()* method with a copy of 'this'. The result will be that the Mailbox adds a copy of the DataMessage to its inbox.**

- A data message arrives at a mail box when the *receive()* method is called on the mail box object. The receive method takes a DataMessage object as an argument. When a data message is received, check the value of the message id with the data message's getter method. If it is less than 0, you will set the MessageID field with the data message's setter method. **Important note! We check this first since data messages may be submitted to more than 1 mailbox. We do not want to overwrite a previously assigned message id!** The ID you assign must be unique so the next available ID in a mailbox should be tracked with a private static integer field (this makes the number unique across ALL mailboxes. Do you understand why?). After assigning a message id, the data message goes automatically into the inbox of the mail box object. The data messages in the inbox are maintained in the order of message ids. The data message with the smallest id number must be located in the index 0 of the inbox array. The data message with the largest id number toward the other end of the array. There must not be any holes between data messages in the array. All the empty slots will be to the right of the data message with the largest id number so far.

- A message gets deleted from a mail box object when the *delete()* method is called on the mail box object with a message id as an argument. If the message is not in the inbox of the mail box object, it just prints out a warning: "Message not in inbox". If it is indeed in the inbox, the message is moved to the delbox of the same mail box object. When a message is deleted from the inbox, the hole that gets created must be filled in.

  How do you handle the hole then? If you have n messages in the array containing messages, the messages must occupy the locations in the array from index 0 to index $n - 1$ at all times. Also the locations in the array from index n to the last index of the array must contain a null value. Additionally the messages in the array (both arrays) must be kept in sorted order according to their message ids at all times.

- Given a data message, a mail box object should be able to answer if the message is in the mail box or not. To do this the Mailbox class supports a method named *searchMessage()* that takes a message id as its argument. The return value is an integer and will be one of the following three: (1) "inbox", (2) "delbox", or (3) "neither" with their obvious meanings.

- Additional methods to add to Mailbox:
  - A constructor that takes one String argument and sets the name of the Mailbox.
  - *toString()* which only returns the 'name' of the Mailbox.
  - A method called *dumpInbox()* that takes no arguments and returns void. It will print each of the DataMessages in the inbox using System.out.println on the DataMessage. This will cause DataMessage's *toString()* method to print out the contents of the DataMessage.
  - A method called *dumpDelbox()* which will dump the DataMessages in the deleted box in the same manner as *dumpInbox()* does for the inbox.

- Look over the `UseMailbox.java` file distributed with this assignment as well as `UseMailboxOutput.txt`. This will give you an idea of how your class should respond to a series of calls from a client.

- You may add any additional 'helper' methods to Mailbox in order to help you keep the inbox and delbox sorted and manage operations like receive or delete.

Include `Mailbox.java` and your revised `DataMessage.java` in your submission.

Submission Instructions

Please follow this procedure for submission:

1. Place the deliverable files (`Ipad.java, UseIpad.java, Mailbox.java, DataMessage.java`) into a folder by themselves. The folder's name should be CSE114_PS8_<yourname>_<yourid>. So if your name is Joe Cool and your id is 12345678, the folder should be named 'CSE114_PS8_JoeCool_12345678.

2. Compress the folder and submit the zip file.
   a. On windows, do this by pressing the right-mouse button while hovering over the folder. Select 'Send to -> Compressed (zipped) folder'. The compressed folder will have the same name with a .zip extension. You will upload that file to the Blackboard dropbox for PS8.
   b. On mac, move the mouse over the folder then right-click (or for single button mouse, use Control-click) and select **Compress**. There should now be a file with the same name and a .zip extension. You will upload that file to the Blackboard dropbox for PS8.
3. Navigate to the course blackboard site. Click **Assignments** in the left column menu. Click **PS8** in the content area. Under **ASSIGNMENT SUBMISSION**, click **Browse My Computer** next to **Attach Files.** Find the zip file and click it to upload it to the web site.
4. ***Important!!*** Click the **Submit** button! If you fail to click **Submit**, the assignment will not be downloadable and will not be graded.