The State Universtiy of New York, Korea
Computer Science

CSE 114 Handout 12: PS 8 May 3, 2019

This problem set is due **Sunday, May 12 at 11:59pm, KST**. Note that the due date that you see on Blackboard is not accurate since it shows the time in EST. You should go by the due date in this handout.

- 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 thats 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.
- **Be sure to include** a comment at the top of each file submitted that gives **your name** and **email address**.

What to Submit

Submit the following files as a **single zip or tar file** on **Blackboard**. (Mac users usually create a tar file.) Multiple submissions are allowed before the due date. Please do **not** submit `.class` files or any I did not ask for.

```
Ipad.java
UseIpad.java
Message.java
Mailbox.java
UseMailbox.java
(And any others if you added more)
```

What Java Features to Use

For this assignment, you are allowed to use anything we have covered so far.

Partial vs. Complete Solutions

Please read what I said in PS 1 on this issue.

Naming Conventions In Java And Programming Style In General

Refer to the ones given in PS 1.

Introduction

Using `Account.java` in Lecture 15 (May 1) 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` in Lecture 15 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

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 or what you do will satisfy the following:

- Write a piece of code in the `main` of `UseIpad` to test each aspect of the requirements 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 `UseIpad.java` into the zip or tar file that you hand in.

Problem 2

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

In Problem Set 6 you designed and implemented a `Message` class. This time, let's design and implement a `Mailbox` class in a file named `Mailbox.java`. Do the following with this class:

- You may use the `Message` class from PS 6. You will have to add new features to the `Message` class from PS 6 as you work through this problem. You are welcome to start with my sample solution if you wish.
- 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 Kay's mail box and another for Jim's mail box, etc. 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.
- A mail box should maintain the following information:
  - It should have a name.
  - A mail box consists of two boxes: inbox and delbox. A message belongs to the inbox or delbox but not in both given a mail box. 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 message must have a *unique* id number represented as an integer assigned to it at the time the message is created. The id associated with a message will remain with the message forever.
- A message arrives at a mail box when the `receive` method is called on the mail box object with a message as an argument. When a message is received, it goes automatically into the inbox of the mail box object. The messages in the inbox is maintained in the order of message ids. The message with the smallest id number must be located in the index 0 of the inbox array. The message with the largest id number toward the other end of the array. There must not be any holes between messages in the array. All the empty slots will be to the right of the 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 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.
- A mail box object may get called with the `undelete` method. This method undeletes only the very last one that was deleted. That is, it moves the very last message that was moved from the inbox to the delbox back to the inbox again maintaining the order of messages. This `undelete` feature is optional, i.e., not required. In the given `UseMailbox.java` I included calls to `undelete`. If you don't have this feature implemented in yours, comment out the calls to it in `UseMailbox.java` as you test your code.
- Given a 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 will be one of the following three: (1) "inbox", (2) "delbox", or (3) "neither" with their obvious meanings.

Now that we have a class representing messages and a class representing mail boxes, let us build a class that tests these classes. We will call the new class `UseMailbox` in a file named `UseMailbox.java`. The `UseMailbox` class includes only one `static` method in it named `main` as usual. Ideally the `main` must test all aspects of the `Mailbox` class. A good starting point would be to create an instance of `Mailbox` to represent a mail box for, say, Kay. And create some number of messages and add them to Kay's mail box by calling the `receive` method. Then perform various operations in some well-engineered order so that the various capabilities of the class is adequately tested. Add enough so that we will be convinced that your testing is adequate. In this problem, however, I decided to give a `main` in `UseMailbox` to make our grading task easier. So, use the one I give you. You don't need to create the `UseMessage` class of your own. Use mine instead.

Discussion 1: As you think about the design of this problem, you will realize that almost everything in `Mailbox` will be dynamic (non-static). The id generating scheme will rely on a dynamic field (e.g., `nextID`) with a dynamic method associated with the field (e.g., `getNewId()`). All the public methods such as `receive`, `delete`, `undelete`, etc. will be dynamic. The `main` in in `UseMailbox` will be static, as usual. You may add some helper functions that you find useful in `Mailbox` which can be static if needed.

Discussion 2: How many arrays would you use in a mail box object to maintain the messages in a mailbox? There are at least two possibilities I can think of:

- A single array containing all the messages in the mail box, deleted or not. In this design you will have to add a flag to each message to remember whether the message has been deleted or not at any given time. So the boolean flag will remember the status of the message.
- Two arrays: one containing all the messages in the inbox and another containing all the messages in the delbox. This is the one I described in the problem description above.
- You may even think of other possibilities.

Although I described the one with two arrays in the main part of the problem description, if you want to use the single array approach, you are welcome to try. Even in that case you must maintain the messages in the sorted order according to their message ids at all times.

Note that `UseMailbox.java` is given along with a sample output generated by my solution. Your solution should produce the same output. Here again try to keep `UseMailbox.java` as close to what I give you as possible so that grading will be easier. My sample output file is given in `UseMailbox_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 `Message.java`, `Mailbox.java`, and `UseMailbox.java` to the zip or tar file that you submit.