CSE 216 – Homework I

Instructor: Dr. Ritwik Banerjee

This homework document consists of 5 pages. Carefully read the entire document before you start coding. You will notice that this homework may be a little more open-ended that the typical assignment in your previous coding semesters: here, you are being told *what* behavior is expected of your code, but *how* to implement it is not entirely specified. You are expected to use the concepts from the lectures and recitations to figure this out on your own.

---

1 Overview

This is a Java assignment, and you must use JDK 1.8 (or higher). Also, the standard Java compiler is `javac`, which is what you must use to compile your code and run/test it. This is the standard compiler, except for Eclipse, which by default uses a different compiler called ECJ. You should either use a different IDE, or change the compiler used by Eclipse.
In the assignment, two interface types are provided to you: `Phone` and `Computer`. Further, a class `PhoneNumber` is mentioned in one of the interfaces, but the code is not provided. In this assignment, you will have to implement a few classes in such a way that the specifications are honored.

2 Tasks

2.1 Classes and Interfaces

Your task is to fully implement the following classes:

- `PhoneNumber`
- `OldLandline`
- `Landline`
- `SmartPhone`
- `Laptop`

Your implementation must adhere to the following behavioral specifications:

1. The `PhoneNumber` class is a simple class. It is simply a wrapper containing a `final long` value, which is a 10-digit phone number. The constructor of this class must ensure that an instance of this class is always a 10-digit number. If the constructor is provided anything else, it must throw a runtime exception of the type `java.lang.IllegalArgumentException`.
2. An old landline (i.e., the `OldLandline` class type) is a phone that can only make or receive a phone call. The restrictions are:
   (a) More than one simultaneous calls cannot be made from an old landline.
   (b) More than one simultaneous calls cannot be received by an old landline.

   (c) For simplicity, you may assume that if the receiver of a call is not busy, then s/he *will* pick up their phone when it rings.
3. A landline (i.e., the `Landline` class type) "is a" old landline with the additional functionality that anyone calling it may choose to leave a message in case the intended recipient is busy on another phone call. As such, it "has a" data type(s) to store messages. The details are:
   (a) Each message is marked as `READ` or `UNREAD` using an enumerable type called `MSG_STATUS`.
   (b) A new message left on a `Landline` is initially always `UNREAD`.
   (c) `Landline` has an instance method to iterate through (i) all messages, or (ii) only unread messages. The method, called `readMessages(MSG_STATUS)` takes a `MSG_STATUS` as its parameter and iteratively prints all the relevant messages. Once a message has been read, its status must be updated to `READ` (if it was already `READ`, then no change is needed).
   (d) Adding a new message on a `Landline` must be a constant-time operation on the data type used to store the messages.
   (e) The messages must be iterable in the order in which they were received. That is, if A leaves a message m_1 for B, and later, C leaves a message m_2 for B, then the iteration through the messages must print m_1 before m_2.
4. A `SmartPhone` is a class that adds even more functionality to the `Landline`[1]. This class also obeys the behavior defined by the `Computer` interface type. It must adhere to the following:
   (a) If the parameter is not one of "on", "ON", "off', or "OFF", the `setState(String to)` method must throw a runtime `java.lang.IllegalArgumentException`.
   (b) A `SmartPhone` is able to store up to 5 games. Accessing any particular game must be a constant-time operation.
   (c) If a game is already installed on the smart phone, calling the `installGame(String gameName)` method should realize this in constant-time, and not reinstall the game. This method should silently ignore the reinstallation (i.e., you do not have to print anything if the game is already installed).
5. Finally, there is a `Laptop` class, which again "is a" Computer. But unlike the `SmartPhone`, it does not have the functionalities of any `Phone`.
   (a) A `Laptop` has two additional variables, `private String brand` and `private String hostname`. The `hostname` incoroporates the brand name, the laptop owner's name, and the type of computer. For example, if John is the owner of a Dell laptop, the `hostname` is "`John's Dell laptop`".
   (b) The behavior of the `setState(String to)` method is exactly like that of a `SmartPhone`, with the same exception handling.

2.2 Generic Interfaces and Ordering

The second part of your assignment is about ensuring that these objects can be ordered in various ways so that they can be used en masse in real-world applications. For example, we should be able to arrange phones in increasing order of the phone numbers, or computers in increase order of their screen sizes.

The classes you implemented in Sec. 2.1 are not to be *naturally* ordered, because their ordering differs according to the needs to an application (e.g., census may want to order phones by the owner's names, but marketing teams may want to order by the area codes). Therefore, your job is to implement the following:

[1] Yes, a smartphone really isn't an extension of a landline, but your professor was born in the 1980s, so he should be allowed this much creative freedom when designing assignments.

1. Order phones by their numbers (increasing).
2. Order phones by their owner names (alphabetically, case-insensitive).
3. Order computers by their screen size (increasing).
4. Order computers by their brand names (alphabetically, case-insensitive).
5. Order computers according to the amount of RAM (increasing).

Note that there may be objects that are phones as well as computers, so think carefully about how you implement the ordering. At the end of your implementation, you must be able to directly use the Java Collections library for sorting (i.e., the `java.util.Collections#sort` method), instead of writing your own sorting code.

2.3 Your own test cases

In this section, a small sample input and output is shown so that you understand what to expect from your code. Keep in mind this is just a sample. Testing the complete functionality is *your* responsibility, and we will not be providing test cases as part of the assignment.

2.3.1 Sample Input

```
public class Test {
    public static void main(String[] args) {
        Landline jack = new Landline("Jack", 6312028899L);
        Landline john = new Landline("John", 3028113434L);
        OldLandline julie = new OldLandline("Julie", 9018772324L);

        // further constructor details hidden in this sample
        SmartPhone paul = new SmartPhone("Paul", 2024449019, ...);

        jack.call(john);
        julie.call(john);
        john.end();
        julie.call(john);
        john.call(jack);
        julie.end();

        paul.installGame("fortnite");
        paul.installGame("angry birds");
        paul.installGame("fortnite");
        paul.playGame("minecraft");
        paul.playGame("fortnite");
    }
}
```

2.3.2 Output

The output of the above code is as follows (command-line user input in green italics):

> Jack is on the phone with John.
> Julie is unable to call John. Line is currently busy.
> Does Julie want to leave a message? [y/n] *y*

call me back when you can
Julie left a message for John.
John has ended the call to Jack.
Julie is on the phone with John.
Julie has ended the call to John.
Installing fornite on Paul's smart phone.
Installing angry birds on Paul's smart phone.
Cannot play minecraft on Paul's smart phone. Install it first.
Paul is playing fortnite.

As you can see from the above sample, all the actions (e.g., calling, installing, playing, leaving a message, etc.) are indicated by simple print statements that show what is going on. You must have similar indicative statements in your code so that the user knows how your code is running. For instance, if Jack is on the phone with John, and Julie calls John, the user must see some printed statement showing that Julie was not able to make a call.
Leaving a message on the `Landline` must be done through user input (because it is possible that the caller doesn't want to leave a message).
To create your own test cases, you must write a `public class Test` like above. If must include the following types of tests:

1. What happens to the ordering and the various methods like `call(), end()`, etc. when real and apparent types of the phones are different? That is, cases like:

```
OldLandline x = new Landline(...);
Phone x = new Landline(...);
Phone x = new OldLandline(...);
Phone x = new SmartPhone(...);
```

2. Similarly, what happens to the ordering and the various methods like `playGame()`, etc. when the real and apparent types of the computers are different?
3. You must also have cases in your `Test.java` where various phone call situations are tested (some examples are shown in Sec. 2.3.1).

3 Grading Scheme

`PhoneNumber`	(5 points)
`OldLandline` and its functionalities	(20 points)
`Landline` and its functionalities	(10 points)
`SmartPhone` and its functionalities	(10 points)
`Laptop` and its functionalities	(10 points)
Implementation of all the orderings	(30 points)
A test class (`Test.java`) showing what kind of tests you ran	(10 points)
Code quality	(5 points)
TOTAL	100 points

Note: Code quality has been assigned 5 points here. This is a subjective assessment, which will be based on things like whether or not you are following standard coding conventions of Java (e.g., class and interface names start with capitals, runtime exceptions have meaningful messages, variables are not unnecessarily `public`, etc.)

4 What to submit?

A single `.zip` file consisting of the following `.java` files:

- `PhoneNumber.java`
- `OldLandline.java`
- `Landline.java`
- `SmartPhone.java`
- `Laptop.java`
- `Orderings.java` [each way of implementing an ordering must be a static inner class inside `Orderings`]
- `Test.java` [a test class containing **only** a `public static void main` method, like the sample input code shown in Sec. 3.1.]

5 FAQ

Can I change the interface code?

No.

I submitted something wrong by mistake . . . what should I do now?

You are allowed multiple attempts until the deadline (at which point, submissions will no longer be accepted). If you have made a mistake, you must resubmit ALL files again (i.e., create your `.zip` file again). Partial resubmissions are not possible on Blackboard.

My code compiles on my laptop but . . .

Then you have changed some code that you were not supposed to change, or submitted something that is not in line with the guidelines provided, or used a different non-standard Java compiler or version. **If your code does not compile, it will not be graded.**

I submitted the class files by mistake.

Class files are not code, and cannot be graded. Multiple attempts are allowed on Blackboard such that everyone has several chances to correct such mistakes.

I missed the deadline by 2 minutes . . .

Do NOT continue coding until the last minute! Verifying the code, properly creating the .zip file, etc. . . . these things take some time, and you are responsible for managing all this within the deadline. **Late submissions will not be accepted under any circumstances**.

- If too many students are trying to submit at, say, 11:58 pm one minute before the deadline, Blackboard server may become very slow and you may end up missing the submission deadline! To be safe, always, ALWAYS, prepare to submit ahead of time, not exactly AT last moment!

Submission Deadline: Mar 08, 2019 (Friday), 11:59 pm