CS 3300: Object Oriented Programming using Java

Spring 2016

Homework 2 (Individual Effort)

Exercises are partially adopted from Object-Oriented Programming in Java CMU course

Assigned: Tuesday February 23, 2016 Due: Wednesday March 9, 2016 (Midnight)

General Submission Instructions:

There are 4 exercises in this homework, each is worth 25 points. For this homework, follow the commenting and coding convention, i.e. add comments to your code to explain classes/methods definitions and use the standard java naming conventions.

Put ONLY your ''.java'' files for each exercise in their corresponding exercise number directory (e.g. excerise1, excersise2, ..etc). Bundle all your directories into a single zip file **yourname_Homework2.zip** and submit it to the corresponding turnIn in the blackboard by the deadline indicated above. Once you upload your file, make sure you can download it back to verify that it went through. Your code should be error free and ready for compilation, any code that will not compile will get a zero.

1. Write a Java program to experiment `String` and `StringBuffer` classes. See their documentation at http://java.sun.com/j2se/1.4.2/docs/api/index.html. Have a class named `StringTest` with a `main` function. The `main` function should expect exactly two command-line arguments. If the number of command-line arguments is not two, the main function should return immediately. Otherwise, it should print out the command-line arguments and continue as follows:

   - Create two `String` objects, say `s1` and `s2`, initializing them with the first and the second command-line arguments. Experiment and print out the results of the following: `s1.length()`, `s1.charAt(i)` for all `i` for `String s1`, `s1.equals(s2)`, `s1.equalsIgnoreCase(s2)`,`s1.compareTo(s2)`, `s1.regionMatches (int toffset, s2, int offset, int len)` for some offset and len, `s1.regionMatches(boolean ignoreCase, int toffset, s2, int offset, int len)` for some offset and len, `s1.indexOf(c, i)` for some c and i, `s1.concat(s2)`, `s1.replace(c1, c2)`, `s1.upperCase()`, `s1.lowerCase()`. You can prompt the user to enter the parameter values (e.g. offset, len .. etc).

- Create two `StringBuffer` objects, say `sbuf1` and `sbuf2`, initializing them with the first and the second command-line arguments. Experiment and print out the results of the following: `sbuf1.length()`, `sbuf1.delete(int start, int end)`, `sbuf1.deleteCharAt(int index)`, `sbuf1.reverse()` methods. Invoke `sbuf1.replace()` with proper arguments. Call the `sbuf1.append().append()` methods in a chain of method calls by passing primitive data types and `sbuf2` as the parameters.

2. Define a `Student` class with instance variables name, id, midterm1, midterm2 and final. Name is a string whereas others are all integers`. Also, add a static variable nextId, which is an integer and statically initialized to 1. Have some overloaded constructors. In each of them, the id should be assigned to the next available id given by nextId. The default constructor should set the name of the student object to "StudentX" where X is the next id. Add a `calculateGrade()` method which returns a string for the letter grade of the student, like "A", "B", "C", "D" or "F", based on the overall score. Overall score should be calculated as (30% of midterm1 + 30% of midterm2 + 40% of final). The letter grade should be calculated the same way as in the homework 1 exercises.
   Your test class, to be named as `TestStudents`, should create 25 student objects with default constructor and invoke the setter methods for midterm1, midterm2 and final with random numbers ranging from 50 to 100 inclusive. After that, it should print the student information. Student information should include name, midterm1, midterm2, final and the letter grade given by the `calculateGrade()` method.

3. Define a class named `ComplexNumber` in order to abstract the complex numbers. Complex numbers have a real and an imaginary part, both of which to be represented by double type in your class definition. Define `add` and `subtract` methods which both take an object of type `ComplexNumber` and return the *`this`* reference after performing the addition or subtraction. Also, define `static add` and `subtract` functions that take two `ComplexNumber` objects and return a new object. Additionally define the `equals` method that can be used to compare two complex numbers. Define at least three constructors; a default one, one taking two double parameters for the real and the imaginary parts, another one taking an object of `ComplexNumber` class as parameter to be used to initialize your current object. Have a `TestComplexNumbers` class to test each of the methods of the `ComplexNumber` class and print out the results (no need to take in user input, use any numbers for the testing purposes.

4. This exercise will get you experiment the composition relationship of "has-a" in OOP. Consider the following UML conceptual model in Figure 1 and write class definitions for each of the concepts.

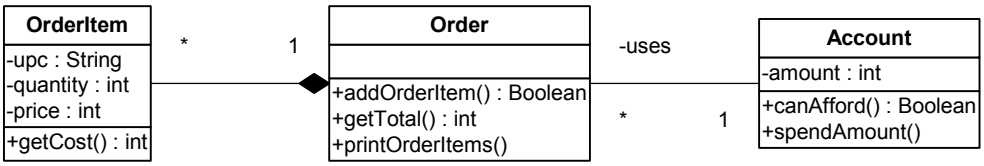

Figure 1 Conceptual UML Model for Exercise 4

`Account` class has an integer instance variable, amount, to represent the available fund. The `canAfford()` method takes an integer parameter being a cost of an order item and returns if there is enough fund in the account to afford that. The `spendAmount()` method takes also an integer parameter being a cost of an order item and deducts the amount by that cost.

`Order` class uses an object of `Account` class. It should have a list of `OrderItem` objects. The `addOrderItem()` method takes an object of `OrderItem` as the parameter. It calls the account object to check if there is enough fund to cover the cost of the order item. If so, it invokes the `spendAmount()` on the account object, stores the order item in a list and returns true. Otherwise, it returns false. The `getTotal()` method returns the total cost of all order items in the order. The `printOrderItems()` method prints information about each order item.

`OrderItem` class has a String upc, an integer quantity and an integer price. The `getCost()` method returns the multiplication of its quantity and price.

Introduce a `Homework2_4` class as a test driver. It should execute as follows:

a. Create an `Account` object with a random amount between 100 to 200 inclusive.
b. Create an `Order` object and pass to it the account reference.
    i. Prompt the user to enter the UPC code for the item or "done" to quit.
    ii. While the user enter a UPC (anything other than "done"), prompt for and read the quantity. You may assume the user enters a valid input, that is, an integer.
    iii. Generate a random number from 50 to 100 for the price.
    iv. Create an `OrderItem` object with UPC, quantity and price.
    v. Invoke the `addOrderItem()` method on the order reference with the order item just created. If it returns false, it means there is not enough fund in the account, so break the loop.
    vi. Continue the loop until either user enters "done" or account can no longer afford a new order item.
c. Print out all the order items and the total cost.