CSL303 IOOM

Programming Format Guidelines

Writing code:

- For every class write constructors (default as well as parameterized), copy constructor, getter & setter methods and destructor.
- Write driver program to demonstrate the working of the code wherever required.

Grading criteria

- *Does it work?* Your programs should compile without warnings or errors. They should give the correct answer, both on the example inputs given in the assignment (if provided) and possibly on arbitrary new examples we may create during testing. You can get at most 30% of the maximum score if the program does not run.
- Is it found correct also after reading the code? (The test set may not have revealed all bugs.)
- The quality of the solution: e.g., is it unduly complex?
- Comments and structuring.

Here are some details of the last point.

Comments

- A program header containing the name of the name, enrollment, assignment number and date, and a very short description (shorter than the external documentation) of what the program is about.
- Brief comments before the definition of each major procedure or group of procedures detailing:
    - What does it do?
    - The use of each parameter.
    - Preconditions that must be met when the procedure is called.
    - Output, return values and side effects (if any).
- Similar explanation is expected for each non-obvious variable, structure, or class declaration.
- Brief comments at any point in the program at which the reader needs help in understanding. Comments which merely rephrase code are unnecessary, distracting, and a potential maintenance hazard (often code is changed, but the comments are not). Comments should edify the code, not repeat it. Use of good variable names and function names can help keep code self-commenting.

Structuring

- Make the program largely self-explanatory by arranging it in an intelligent way and giving informative names to your variables and procedures. Then fewer comments are needed since you must only comment the non-obvious.
- Maintain consistent indentation as it helps to discover many syntactic errors early.

Naming conventions

- Names of variables, constants and functions must clearly indicate the function and type of the named object.
- Follow the convention that constants should be all capital letters, with multiple words separated by underscore "_".
- Within variable and function names, multiple words can be separated by either capitalizing or an underscore.
- The names must be self-explanatory not only for you but also for others reading your code.

Global variables

Try to avoid global variables using methods for structuring programs taught in the course.

CSL303- Introduction to Object Oriented Methodology

Lab Assignment – 3 – Java Programming

Submission Date: Tuesday 1 December, 2020 Time 11.50 PM

Marks : 20

Que. 1)

Problem Statement

The Department of Transport & Highways of a particular state is installing a toll collection system on one of its major roads. Trucks pulling up to a tollbooth are required to pay a toll of Rs. 5 per axle plus Rs.10 per half-ton of the truck's total weight. A display in the booth shows the toll receipts and the number of truck arrivals since the last collection.

You will design an object oriented program in Java that simulates the operation of the tollbooth.

Sample Scenario

To aid understand the problem statement let us imagine how such a toll collection system might work. A toll agent sits in a tollbooth that is equipped with a computer screen and a bar code reader. When a truck arrives at the booth, the agent scans a bar code on the windshield of the truck; it contains information about the truck, including its number of axles. The weight of the truck is obtained by scanning a bar code contained on the bill of lading presented by the driver. The truck information and toll due are then displayed on the computer screen:

Truck arrival - Axles: 5 Total weight: 12500 Toll due: Rs.275

When a button on the side of the screen is pressed, the booth's totals are displayed:

Totals since last collection - Receipts: Rs.305 Trucks: 2

When the cash drawer is removed from its cradle, the following is displayed on the screen. The totals are displayed and then reset to zero:

****** Collecting receipts ******

Totals since the last collection - Receipts: Rs.353 Trucks: 5

This scenario will be simulated by the object oriented system that you will build.

Primary Objects in the Problem Statement

The primary objects of the tollbooth model problem are the relevant noun phrases of the problem statement. In this case we list: *trucks, tollbooth, axle, weight, receipts.* Of these, tollbooth and truck are the most important. Axles and weight are properties of a truck, and receipts are secondary to the tollbooth. Therefore, truck and tollbooth are primary types.

Trucks Behaviors and Attributes

Toll depends on the number of axles and weight of a truck. Tollbooths need to get both of these from a truck. Therefore, a truck should keep track of the number of *axles* it has, its *total weight*, and allow other objects to get access to these but not modify these attributes. In addition a truck is of a particular *make*.

Tollbooth Behaviors and Attributes

The main behavior required of a tollbooth is the ability to *calculate the toll* due. We also want to be able to *display the data* for the total receipts and number of trucks since the last receipt collection. This implies that these totals can be reset by the tollbooth *on receipt of collection.*

Directions to complete the assignment

1. Create and compile the two interfaces (*Truck* and *TollBooth)*

2. Create two classes that implement the specified interface Truck. Each class should be named after a truck make of your choice (obviously the two truck makes must be different).
   These classes should have the member variables (attributes) identified earlier. Carefully choose the access level modifier and make sure that you take into account the fact that some variables change over time and others don't.

3. Create a class that implements the specified interface TollBooth. For example: **AirportTollBooth**
   This class keeps track of a) the total number of trucks that have gone through the tollbooth and b) total receipts since collection.
4. Have a clock of 24 hrs for collection and store this data on file system, so as to retrieve the same for future use.
5. The methods of your class specified in the interface will at least perform the following:
   a) display the booth's totals simply by printing those values to System.out and b) display the totals maintained by the booth and then reset them to zero because receipts and truck. Remember that totals are maintained only since the most recent collection. This corresponds to the supervisor of the tollbooth, emptying the cash box in the booth and resetting the meters.
   c) Read and display the collection datewise for specific dates or range of the dates for the specific TollBooth.

Using the tollbooth

The following represents sample usage code for the classes above. You may use it to test your overall implementation.

Use the class names that you have defined for your implementations of the truck interface. Also, the method names on your tollbooth may be called differently. You may also have different types of constructors depending on how much initialization of variables you choose when instantiating a class.

```
Class TestTollBooth {

      Public static void main(String [] args){

            TollBooth booth = new AirportTollBooth();

            Truck Tata = new TataTruck(5, 12000); // 5 axles and 12000 kilograms

            Truck LandT = new LandTruck(2, 5000); // 2 axles and 5000kg
            Truck AshokLeyland = new AshokLeylandTruck(6, 17000); // ….



            booth.calculateToll(LandT);

            booth.displayData();
            booth.calculateToll(AshokLeyland);

            booth.datewiseCollection(d1, d2);// d1 & d2 are dates


      }

}
```

Que. 2)

A Course Coordinator wishes to create a program that lists his / her students sorted by the number of lab assignments they have completed. The listing should be greatest number of assignments first, sub-sorted by name in lexicographical order (A to Z).

A class **`StudentInfo`** stores the name and number of assignments completed for a student. Amongst other methods, it contains a **`void setCompleted(int n)`** method that allows changes to the number of completed assignments.

(a) Provide a definition of **`StudentInfo`** with an **`equals()`** method and a natural ordering that matches the given requirement.

(b) A **`TreeSet`** is used to maintain the **`StudentInfo`** objects in appropriate order. When **`setCompleted(...)`** is called on a **`StudentInfo`** object it is necessary to remove the object from the set, change the value and then reinsert it to ensure the correct ordering. This is to be automated with the classes **`UpdatableTreeSet`** and **`SubscribableStudentInfo.`**

A partial definition of **`UpdatableTreeSet`** is provided below.

```
public class UpdatableTreeSet extends
TreeSet<SubscribableStudentInfo> {
// To be called just before the StudentInfo object is updated
      public void beforeUpdate(SubscribableStudentInfo s) {
```

```
            remove(s);
      }
// To be called just after the StudentInfo object is updated
      public void afterUpdate(SubscribableStudentInfo s) {
            add(s);
      }
}
```

(i) Extend **`StudentInfo`** to create **`SubscribableStudentInfo`** such that: multiple **`UpdatableTreeSet`** objects can subscribe and unsubscribe to receive updates from it; and the **`beforeUpdate(...)`** and **`afterUpdate(...)`** methods are called appropriately on the subscribed **`UpdatableTreeSet`** objects whenever **`setCompleted(...)`** is called.

(ii) Give a complete definition of **`UpdatableTreeSet`** that overrides the inherited methods **`Boolean add(SubscribableStudentInfo)`** and **`Boolean remove(Object)`** to automatically subscribe and unsubscribe to their arguments, as appropriate. You may ignore all other methods inherited from **`TreeSet.`**