CMPT 270 – 2016-2017 Term 01

Assignment 3:
Inheritance and Data Structures

Submission Deadline: Friday October 21 2016 at 11:55pm – Submit via Moodle!

Description

The objective of this assignment is to learn about inheritance in Java and get familiar with the basic data structures of Java. In the process, four classes, that extend the flight reservation application of Assignment 2, will be built and tested.

The four classes that you write should extend and use the sample solutions provided for Assignment 2 (to be posted Oct. 11; can get started with own files beforehand). These will be posted on Moodle.

In particular, you must create the three classes: a `Person` class, a `Booking` class, and a `BasicFlight` class. Each class should have a constructor, and the fields and methods specified below. For each of the classes, all the fields should be declared as private. Be sure to follow the specification given below, since the classes will be used in subsequent assignments to build a simple flight reservation system. In addition, each class is to have a static main method that should be used to test the class following the principles of regression testing as discussed in class.

Deliverables

***** Upload .java files and .txt files for documentation to Moodle. *****

You are expected to do a good job of testing all routines.

We may run your classes to verify that they run as you state. You should also include .txt files for:

(i) the external documentation for the assignment

(ii) a listing of all the classes,

(iii) a listing of the output from running your classes

Flight Reservation Application Extended

Each of the classes is now discussed in more detail:

`Passenger` class:

The first class is to be a `Passenger` class that extends the `Person` class. The `Passenger` class should have a linked list to store all the bookings of the `Passenger` . This class will need a method to add a booking for the `Passenger` , and toString() should be overridden to include the main information for the bookings of the `Passenger` . If you output all the information for a flight of a passenger, you might end up with an infinite loop. Also, include methods in class `Passenger` with a `Flight` parameter to determine whether the passenger has a booking on the flight, and whether the passenger has a seat reserved on the flight.

`FirstClassBooking` class:

The second class is to be a `FirstClassBooking` that is a descendant of Booking, i.e., it extends Booking. What distinguishes a first-class booking from a regular booking is that a first-class booking has a meal plan. The meal plan will be a String that is initially set via the constructor. There should be provisions to access the meal plan and to change the meal plan. A null meal plan is allowed.

`Flight` class:

A `Flight` class that extends `BasicFlight` should also be included. The main new part of the `Flight` class is to use a linked list to store the bookings for the flight that have not yet been assigned a seat. From the last assignment, the `BasicFlight` class has a method to set a booking into a seat. However, it is not clear which class is actually creating the booking. The `Flight` class will be given this responsibility. The `Flight` class needs two ways (methods) to make a booking: a regular booking for a passenger with no seat or meal plan, and a first-class booking for a passenger with both a seat and meal plan specified. In both cases, the method should create the booking object, place the booking in the passengers list of bookings, and either place the booking in the appropriate seat or place the booking in the list of bookings that have not yet been assigned a seat. In addition, a method is needed to assign a seat to a `Passenger` that already has a booking. As well as putting the booking into the array, this method will need to remove the booking from the list of those bookings without a seat. The class should also have two new accessors: a method to return the remaining capacity of the flight (capacity - total number of bookings), and a method to return a string that lists the empty seats of the flight. The latter string should be formatted so that if the string is printed, the empty seats will be laid out in a fashion similar to the seats of the flight. Note that each empty seat should be displayed using the seat string, and the display of a booked seat showing up as blank. Because the new system will only place persons of type `Passenger` in a `Flight` , when getPerson is invoked on a `Flight` object it would be nice to have the return type `Passenger` . Starting with version 5 of Java, a method can be overridden to return a type that is a descendant of the original return type. You should do this for getPerson. Of course, toString() should be overridden to include list of people booked on the flight but without a seat.

`FlightReservationSystem` class:

The `FlightReservationSystem` class is one to run a simple airline reservation system. It should have two containers: one for all the people (passengers) known to the system, and the other for all the flights known to the system. Both containers should be keyed dictionaries, with the key for a person being their name, and the key for a flight being its number. The system should display a message to the user for the user to select an operation. When an operation is selected, it should be carried out, and then another operation selected. It is easiest to handle operation selection by numbering the operations and having the user enter an integer. Note that when a value is read using Scanner, other than by nextLine(), none the characters after the value are read. Thus, a subsequent nextLine() read (say to read a name)will read those

characters up to and including the next end-of-line characters, rather than the characters on the next line (which is probably what was wanted). Such a read often just obtains the characters to mark the end of the line. Thus, after reading an int value, you might want to invoke a nextLine call to read the rest of the line.

The operations to be supported are the following:

1. quit
2. add a new passenger to the system
3. add a new flight to the system
4. display the empty seats of a specified flight
5. book a passenger on a flight
6. book a first-class passenger on a flight
7. assign a seat to a passenger on a flight where the passenger already has a booking
8. display all the passengers in the system
9. display all the flights in the system

When the user quits, the system should print out the people and flights that are currently in the system. The output of the passengers and flights can simply use the default format obtained by invoking toString on their container. The parameters for these operations should be read from the console. In most cases, the parameters for the operation should be obvious. For example, to book a passenger on a flight, the persons name and the number of the flight need to be entered. This operation assumes that both the passenger and the flight are already in the system. The assign a seat operation needs the persons name, the flight number, and the seat. Although they are easy to do, to save a little time, operation 3 (display empty seats) and operation 5 (book first-class passenger) are optional, and you need not do them. The last two commands are included to help you check your system by outputting the passengers and flights to ensure that the correct information is stored.

Additional Guidelines

Internal Documentation

When writing these classes, be sure to properly document each method, including @param and @return comments. Also, if a method has a precondition, specify the precondition in a @precond comment, and throw a runtime exception if it is not satisfied. When appropriate, exceptions should be caught and handled. In particular, if any operation of the system fails and as a result throws an exception, it is reasonable to catch the exception, print the message of the exception, and then go on to the next operation. Note that these additional comments and precondition checks have already been added to the classes in the solution for Assignment 2.

External Documentation

For external documentation, include the following:

(i) A description of how to execute your test classes and system. This should be very short.

(ii) The status of your assignment. What is working and what is not working? What is tested and what is not tested? If it is only partially working, the previous point should have described how to run that part or parts that work. For the part or parts not working, describe how close they are to be working. For example, some of the alternatives for how close to working are (i) nothing done; (ii) designed but no code; (iii) designed and part of the code; (iv) designed and all the code but anticipate many faults; or (v) designed and all the code but with a few faults.

Code Structure

- Recall that in developing a system, one of the worst things is to reach the deadline for completion but not have anything working. Be sure to have at least part of your system working.
- Also, a characteristic of good system is that I/O is not scattered throughout the system. It should be concentrated in one place, a class or subsystem, so that if the I/O interface is to be changed, only the one place needs to be changed. For this assignment, all the I/O should be in the system class, except for the main in each class that tests that class. Tests could be developed to test the methods of the system class, but for this assignment, the main of the system class can simply create an instance of the class and run the interactive application.
- Make sure that you do not have long methods. In particular, in your system class, you will probably have a method with a switch statement to determine which operation was selected by the user. When there are many cases, this method can get long. The key to keeping it from getting too long is to have as many tasks as possible (other than the actual switch statement that is determining the choice made) abstracted into other methods of the class. Thus, if something is a self-contained task, separate the task into a separate method. In particular, include a method to read the next integer. You can base your method on the code to read an integer towards the end of the slides on exceptions.
- In keeping with the principle of information hiding, the fields of a class should be private unless there is a very good reason to make them visible. When appropriate, methods should be supplied to access and set the fields.

This is an individual assignment. You are encouraged to discuss the general concepts of classes, types, containers, etc. with you classmates, but the specific details of the Flight Reservation System in this assignment should be done completely individually. Students that copy / share work will be penalized.