Design Doc (Lab Statement)

30 October, 2018

Introduction

This lab is based on exception handling in Java.

In the last lab (Lab 7), you had created a few classes that simulated various types of rooms using classes, interfaces and abstract classes. While the code works fine when an "*ideal*" user is using it, there are a couple of places and situations that need improvement, when the user is "*non-ideal*".

In this lab, the objective is to deal with those mistakes/exception cases using exception handling. We have a "non-ideal" user Raj, who is having a trouble working around the code for his simple Hotel Management suite. Your objective is to understand his issues and figure out what is wrong and how to fix it.

You have been provided with a working code from the last lab. You need to identify places where you would need exception handling and write the necessary code for it. In addition to that, you will create two Custom Exception classes, **InvalidDimensionException** and **InvalidUsageException** that deal with custom errors that are specific to our Hotel Management Suite, and two simple classes **HotelSuite** and **Hotel,** that oversee the operations of a hotel suite and the hotel respectively.

The javadoc provided contains details about this additional classes and the previous classes. However, **details about the exceptions have to be determined by you**.

Class Design

We first explain the class design. This is quite similar to the last lab apart from a few changes/additions. We then list the problems faced by Raj.

Room Class

- The Room abstract class has attributes *roomLength, roomWidth, roomHeight*, which store the length, width, and height of the room respectively. In addition to this, it also provides methods which return the volume and surface area of the room. It also has an abstract method to get the amount of the bill of the room.

LivingRoom Class

- The living room has a TV whose status is controlled by *tvStatus* variable. Depending on the TV usage the electricity bill of the room increases. We have also added a bookshelf in the living room which will store books. These books will be stored as integers, and the bookshelf has a capacity of **10** books.
- The bookshelf is filled using the method fillBookshelf(). The method fillBookshelf() takes an integer array books as an argument, and returns ***true*** if it is filled with books successfully, and ***false*** otherwise.
- To successfully fill the bookshelf the user must be able to fit in all his books in the bookshelf. This must be done by copying the books received as argument to the bookshelf array.
- **Note**: The user cannot use any other appliance apart from TV in the living room.

WashRoom Class

- The washroom consists of a Wash Basin and a Shower. Depending on their usage, the water bill of the wash room increases.
- Note: The user cannot use any other appliance/plumbing fixture apart from Wash Basin and Shower in the WashRoom.

Bed Class

This class represents a bed. It has a bed material property, which determines its cost.

BedRoom Class

- In the last lab, we had a BedRoom with a single bed. But Raj requested to have the option to have *multiple beds in the room*. Hence, we now have an **array of Beds**, denoting multiple beds in the room. Consequently, the maintenance cost of the room was updated to reflect the *average cost of the beds* in a bedroom. The code for the same has been given.
- In case the average cost cannot be calculated, say if the room has 0 beds, the maintenance cost is 0.

New Classes to be created

1. InvalidDimensionException Class

This class handles the Exceptions when dimensions of a room are not valid. For example, it is not possible for a room to have zero height. It stores the *length, width, and height* of the room in consideration, which it receives as arguments in the constructor.

This class will override the getMessage() method of Exception class, and provide us with meaningful custom exception messages.

For example, if the length of the room is less than or equal to zero, getMessage() should return “*Length value is invalid.*”

Similarly, the messages for invalid width and height should be, “*Width value is invalid.*”, and “*Height value is invalid.*”.

Note: Only one invalid value should be present in the exception message returned by getMessage(). Incase, multiple values are invalid, we will follow the priority order, *length > width > height* i.e. say for example, width and height are invalid, the message should be “*Width value is invalid.*”.
In case, length and width are invalid, the message should be “*Length value is invalid.*”.

2. InvalidUsageException Class

This class handles the exceptions when objects (appliances and plumbing fixtures) are tried to be used in invalid places. For example, a user cannot use a TV in the washroom or a shower in the living room.

In this class, we will use the base class constructor to initialise the message attribute of the Exception class, with the message format "**<RoomType> cannot have <Object>**".

For example, if TV is tried to being used in the Wash Room, the exception message should be “*WashRoom cannot have TV*”. These two string values of the room type and object are passed to the constructor of the InvalidUsageException class as arguments.

3. HotelSuite Class

Raj requested that we have a suite room facility available in his Hotel. So, we design a HotelSuite class to fulfil this requirement. The uniqueness of a hotel suite lies in the fact that a hotel suite should have *one living room*, *one wash room*, and can have *multiple bed rooms.* Instances of the living room, wash room, and multiple bed rooms will have to be provided in the constructor of HotelSuite class. We will have a single constructor that can take care of this requirement using variable arguments or varargs feature of Java.

Along with this, we have a method calculateCost() that calculates the total cost of a hotel suite by incorporating the maintenance costs and bills (electricity and water) of all the rooms in it. You can use the getBill() and maintenaceCost() methods of all the rooms for the same.

4. Hotel Class

To represent Raj's Hotel, we have a Hotel class, which stores a collection of hotel suites in it. The hotel can have a *variable number of hotel suites*, which will be provided as arguments to the constructor of the Hotel class. You will have to ensure that only one constructor is used for this purpose using the variable arguments or varargs feature of java.

Along with this, we have a method getHotelCost() that calculates the total cost of the hotel by incorporating the total cost of each hotel suite in it. You can use the calculateCost() of the all hotel suites for the same.

Issues:

Raj has hired Stuart to make the entry of all the rooms in Raj's Hotel into this new Hotel Management Suite. Unlike you, Stuart is not so smart and is prone to mistakes on a regular basis. Also, Stuart cannot identify what he has done wrong from the Exceptions thrown by Java. Hence, to help him, we will make sure we deal with all the exception cases that we can think of Stuart will face. A list of problems is given below, and you are expected to stick to the same. Do not try to add any more exception cases that you can think of.

1. While creating a Room, the room dimensions can be zero or less than zero, due to a typo by Stuart. In this case, we will make use of our **InvalidDimensionException**, to display the custom error message as described in the class design above. **Note**: The object of a room with invalid dimensions should not be created. i.e. *the constructor should throw this error*.

2. In the LivingRoom, while filling the bookshelf one book at a time, if the number of books given to fill the bookshelf is more than the the size of the bookshelf (10), while accessing the bookshelf array for elements with index 10 or more, an exception **ArrayIndexOutOfBounds** is thrown.
   This needs to be caught in the method itself, and we print the exception message using the *toString()* method of the exception instance, on the output stream using System.out.println(), and return **false** in this case.

3. In the BedRoom, while calculating the maintenance cost, if the number of beds is zero, i.e. if the room has no beds, we will try to *divide the sum with 0*.
   This will be a divide by 0 case, which will throw an **ArithmeticException**. This needs to be caught in the method itself, and we print the exception message using the toString() method of the exception instance, on the output stream using

System.out.println(), and return **0** in this case.

4. While using the use() method in any of the rooms, if the user tries to use an object that cannot be used in that room, an **InvalidUsageException** has to be thrown, with the appropriate custom message as described in the class design.
   There is also one more case to consider here. There can be a case where use() is called with a "null" string, i.e. the user tried to use nothing!! This can result in a NullPointerException, which has to be caught.
   Note: The room should not be affected in any way if such a case should happen. i.e. the cost/bill of the room should not increase, appliances should remain in the same state as before, etc.

Additional Notes:

1. The document is complete and correct. If any changes are required, you would find it uploaded on Gitlab immediately.
2. You are strongly advised to use **Eclipse**.
3. Take care of the string format. Stick to the format specified, with the spaces and orientation of lowercase and uppercase characters.
4. All calculations should be of integer type. Do not use any other data type for the same.
5. Make sure you do not violate any inheritance principles while making the modifications.
6. Make sure the existing code works. Do not break the existing functionality while making the changes.
7. TAs will not be helping you to identify any of the exceptions.