Due Sep 28 by 11:59pm **Points** 100 **Available** after Sep 18 at 6am

Module 2 — Methods for Simple Classes and Exceptions

Home

Modules

(https:/(https:/(https:/(https:/(https:/(https:/(https:/(https:/(https:/(https:/(https:/(https:/

Assignment 2: Methods, Packages and Exceptions

Goals: Practice designing the representation of data, writing getter methods and testing them.

This homework will give you experience with:

- Simple class design
- Simple "composition" of classes
- Basic exception handling (by throwing exceptions appropriately)
- Creating and using Java enums and packages
- Creating simple UML class diagrams

1 General Reminders

Pay attention to naming! Our server-side test files expect your submissions to be named a certain way. Therefore, whenever the assignment specifies:

- the exact names of classes, attributes, methods
- the names, types and order of the arguments to the constructor, or
- filenames

Remember to follow the directory structure we covered in class. Place your solution code in the src directory and your JUnit tests in the test directory.

1 Hotel Rooms

1.1 Introduction

You are tasked with writing code that will be part of a hotel's booking system. Your responsibility is to write classes to represent the various types of room the hotel has: single rooms, double rooms, and family rooms. All rooms have:

- A maximum occupancy - This is the maximum number of people that can stay in the room.
- A price - The cost of a single night's stay. **The price of a room cannot be negative.** (We may decide to give people free ("comp") rooms as a perk if they're Diamond members).

- A number of guests - The number of guests currently booked into the room. This value should be set to 0 when the room is first created in the system

The difference between each type of room (single, double, and family), is the maximum occupancy:

- Single rooms can sleep 1 person only.
- Double rooms can sleep a maximum of 2 people.
- Family rooms can sleep a maximum of 4 people.

All rooms have the same behavior (people sleep in the rooms, etc.) so the only variation is the room capacity as listed above.

1.2 What to do

Create a **Room** class that represents the functionality described above. Your class should support these features:

An **isAvailable** method. This method returns (answers) a **boolean** true if the room is available and false if the room is unavailable (currently occupied). A room is considered available if the current number of guests assigned to the room is 0. A room is unavailable if a room currently has 1 or more guests assigned to it.

An **bookRoom** method. This method takes one parameter: the number of guests that will be assigned to the room (**integer**). If the booking is valid (the room is available AND the number of guests passed to this method is greater than 0 but less-than-or-equal-to the maximum occupancy for the room type), this method sets the room's number of guests to the value of the parameter passed in.

A **getNumberOfGuests** method that returns (answers) an **integer** value representing the number of guests currently assigned to the room.

A **Room constructor** that takes two parameters: the type of room being instantiated (**RoomType**), and the price of the room (**double**). **RoomType** is a user-defined type that is described below. If the price parameter is negative, raise an **IllegalArgumentException**.

Create an enum called **RoomType** that represents the three categories of rooms described above. This is the type of the first parameter that should will be passed to your Room constructor.

Notes:

- All of these assets should be created in a package named `hw2`.

- Be sure to properly include Javadoc for your classes and methods.
- Also ensure you implement JUnit tests for your Room class.
- Include a UML class diagram for your solution (in .pdf, .png. or .jpeg format, please)

2 Locker Storage

2.1 Introduction

You are tasked with designing and implementing code that will be part of a system managing package lockers (similar to Amazon Hub lockers). Your code will need to represent mail items, lockers, and recipients (the person the mail is addressed to).

A recipient has:

- A first name
- A last name
- An email address. You do **not** need to validate the format of an email address for this assignment.

A mail item has:
- A width in inches, an integer greater than or equal to 1.
- A height in inches, an integer greater than or equal to 1
- A depth in inches, an integer greater than or equal to 1.
- A recipient.

A locker has:
- A maximum width in inches, an integer greater than or equal to 1.
- A maximum height in inches, an integer greater than or equal to 1.
- A maximum depth in inches, an integer greater than or equal to 1.
- A mail item—the item stores in the locker, if any. If there is no mail in the locker, this field should be set to null. You can assume that when a locker is first created, it is empty.

2.2 What to do

Implement the following classes (plus any other "helper" classes you might need):

- Recipient,
- MailItem, and
- Locker

`Recipient's` constructor takes three strings as parameters: `first name, last name` and `email` (in that order). If any of those parameters is null OR if any of those strings is empty (e.g.: `""`), you should raise an `IllegalArgumentException`. Your Recipient should also implement the `toString` method that represents the recipient similar to this: `FirstName LastName Email:xxx@yyyy.com`

`Again, you do NOT need to validate the email address (or format) for this assignment.`

`MailItem's` constructor takes the parameters: `width, height, depth`, and `Recipient` (in that order). If width, height or depth is less than 1, you should raise an `IllegalArgumentException`. Mail items also MUST have a recipient so you should raise an exception if no recipient is provided for a mail item. You should provide a method called `getRecipient` that returns the MailItem's recipient.

`Locker's` constructor takes the parameters `maxWidth, maxHeight,` and `maxDepth` (in that order). If any of those parameters is less than 1, you should raise an `IllegalArgumentException`. When a locker is created, the mail item should be set to null by default.

For your `Locker` class, also provide these two methods:

1. A void method called `addMail`, that takes one parameter: a mail item. This method stores the mail item in a locker with two exceptions: if the locker is occupied (it already contains a mail item) OR the mail item exceeds the dimensions of the locker (any single dimension of the mail item is bigger than the locker). If either of those situations is true, the mail item **should not be added** to the locker.
2. A method called `pickupMail`, that takes one parameter: a recipient. This method removes and returns the mail item from the locker under the following conditions: IF there is a mail item in the locker AND the recipient passed to `pickupMail` matches the recipient of the mail item. If the recipients do not match OR if there is no mail currently in the locker, this method returns `null`.

Notes:

- All of these assets should be created in the same package as you used for part 1 of this homework (package named hw2).
- Be sure to properly include Javadoc for your classes and methods.
- Also ensure you implement appropriate JUnit tests for your classes.
- Include a UML class diagram for your solution (in .pdf, .png. or .jpeg format, please)

How to submit

1. You can upload your homework to Gradescope by create a zip file that contains directly your src and test folders (and all your UML diagrams too). DO NOT simply drag your folders to the Gradescope upload since Gradescope does NOT preserve the directory structure unless you create a zip archive. Tip: Be sure that your src and test directories are at the ROOT of your archive. In other words, do NOT enclose src and test in a folder (e.g. HW2). Our test suite will look for the src and test directories at the top level of your submission. As a pre-flight check, unzip your .zip file before submission - when you unzip the file, you should see only the src and test folders.

   OR

2. You can submit to Gradescope via your GitHub account if you have one. **Your GitHub (or BitBucket) repositories must be private for this course so other students cannot simply view your code on demand**. Be sure the branch (main or sub-branch) mirrors the directory structure we specified for the assignment. In this case, src and test should be at the top level. Go to the Gradescope submission and select GitHub as the option, then specify your homework's repository/branch.