* [Logout](https://review.udacity.com/#!/reviews/3696700)

[Return to Classroom](https://review.udacity.com/#!/reviews/3696700)

**Hotel Reservation Application**

* [Review](https://review.udacity.com/#!/reviews/3696700)
* [Code Review](https://review.udacity.com/#!/reviews/3696700)
* [History](https://review.udacity.com/#!/reviews/3696700)

**Requires Changes**

**3 specifications require changes**

Dear student,

You are almost there. You have done a wonderful job so far. I was able to successfully run your application and execute most of the actions.

However, you did not meet the recommended rooms requirements.

After meeting the Conflicting reservation requirement above, please make sure to meet the recommended rooms requirement.

**Search for recommended rooms**. If there are no available rooms for the customer's date range, a search will be performed that displays recommended rooms on alternative dates. The recommended room search will add seven days to the original checkin and checkout dates to see if the hotel has any availabilities, and then display the recommended rooms/dates to the customer.

Example: If the customers date range search is 1/1/2020 – 1/5/2020 and all rooms are booked, the system will search again for recommended rooms using the date range 1/8/2020 - 1/12/2020. If there are no recommended rooms, the system will not return any rooms.

You can read more about this requirement [here](https://classroom.udacity.com/nanodegrees/nd079-BETA/parts/d74d0f2c-11e0-43bf-b3ba-e24c8fb931d6/modules/749c72fb-f622-4d19-961a-05e6e733b883/lessons/213a5242-2808-4823-a931-e53b7ec837eb/concepts/660f3c55-24dd-46d2-8b0d-ca3b000e005b)

**I was able to see recommended dates but I was not able to book a recommended rooms**

After making these changes, please, make sure to **test your work** before resubmitting.

**Object-Oriented Programming**

The hotel reservation application contains the IRoom interface , which is implemented by the Room class.

Great job meeting this rubric item. It is a good design principle to **program to an interface**  
In your Reservation model, you can add any room as long as it is an IRoom

Imagine that in the future the hotel adds a PremiumRoom. No problems will arise with your Reservation domain as long as this new room implements the IRoom interface.

The FreeRoom class extends the Room class.

There is at least one example of the model classes (Room, Customer, Reservation`) using data encapsulation.

Data in your models are *private* and only exposed and modified via accessors and mutators. This closeness is a good way to avoid bugs and security concerns that may arise as a result of unnecessarily exposing variables and methods of certain classes.

There is at least one example of the model classes (Room, Customer, Reservation) overriding the toString method.

Good job meeting this requirement 

@Override

public String toString() {

return "Customer:\n" + customer

+ "\nRoom:\n" + room

+ "\nCheckInInformation:\n" + checkInDate

+ "\nCheckOutInformation:\n" + checkOutDate;

}

There is at least one example of the model classes (Room, Customer, Reservation) overriding the equals and hashcode methods.

It is necessary to **correctly** override the equals and hashcode methods to provide better identity to your objects. Without **properly** overriding these methods, working with certain collections like sets and collection methods like contains may be impossible.

[Check out this article for more information on how and why you need to do this](https://www.baeldung.com/java-equals-hashcode-contracts)

[Check how to do this with IntelliJ here](https://www.jetbrains.com/help/idea/generating-code.html)

The application contains at least one example of using each of the following access modifiers: ‘public’, ‘private’ and ‘final’.

Good job implementing data encapsulation in your model classes. However, you should also use the final keyword here to enforce immutability.

Note that the final variables are immutable because they cannot be reassigned. Final methods cannot be overridden.

**How is this important?**

In concurrent environments where access over a shared resource can conflict (ie two threads modifying the same shared *mutable\** resource) immutable variables and methods really shine. So, when possible, always use the final keyword.

Your model variables should be final since they are never reassigned. You can read more about the final keyword in Java [here](https://www.educative.io/edpresso/what-is-the-final-keyword-in-java)

**Notice that:**  
-> I have removed the unused setter methods. Don't add setter methods unless absolutely necessary because they make your classes more mutable, something you will want to avoid in concurrent environments  
-> I have also removed some getter methods. Do not return data that is not needed for other classes to function. Always expose as minimum as possible from your classes.

See the changes below:

package model.customer;

import java.util.regex.Pattern;

public class Customer {

//Creating the variables

private final String firstName;

private final String lastName;

private final String email;

//Setting the email Regex with a wildcard to stabilise input format

private final String emailRegex = "(?)^(.+)@(.+).com$" ;

private final Pattern pattern = Pattern.compile(emailRegex);

public Customer(String firstName, String lastName, String email) {

super();

if(!pattern.matcher(email).matches()){

throw new IllegalArgumentException("Please use acceptable email format\n"+

"Email should be in lower case");

}

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

//Overriding the class

@Override

public String toString() {

return "Customer{" +

"firstName='" + firstName + '\'' +

", lastName='" + lastName + '\'' +

", email='" + email + '\'' +

'}';

}

}

**Processing and Storing Data**

Collections are used to store data for:

* Room
* Customer
* Reservation

The collection type chosen for rooms ensures that two rooms cannot be booked at the same time.

[Text

Description automatically generated](https://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/570120/1662903153/Screenshot_2022-09-11_at_14.32.27.png)

Good job duplicate reservations are not allowed

The ReservationService contains for or while loops that are used to iterate over and process data in order to do the following:

* Search for available rooms
* Search for recommended rooms

[Text

Description automatically generated](https://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/570120/1662904454/Screenshot_2022-09-11_at_14.53.47.png)

Recommended rooms are being displayed but Im not able to book one. **please, use the alternative dates to book the recommended rooms**

All of the service classes use static references to create singleton objects.

Good job using static references to create singleton objects in your application. You can read more about singletons [here](https://www.baeldung.com/java-singleton#:~:text=For%20example%2C%20singletons%20are%20often,configuration%20for%20the%20test%20environment)

The ReservationService contains at least one example of using each of the following method access modifiers:

* public
* private
* default

Good job using private constructors in your singleton classes.

**Core Java Concepts**

The Customer class should contain at least one example of validating a String to ensure that it has valid email address syntax.

The application contains the enumeration class RoomType.

Enums are a good way to group compile-time constants in our application. Good job meeting this requirement. You can read more about enums in Java [here](https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html)

The Reservation class uses Date objects for check-in date and check-out date.

The application contains at least one example of using Exceptions to validate input and try and catch blocks to handle error flow without crashing the application.

Exception handling is very important. Sometimes, we need to know when errors occur and trigger necessary actions/alerts. Oher times, we just need to make sure that exceptions/errors don't block other tasks from being executed. There are many reasons why we need to provide good exception handling mechanisms.

You can read more about best practices for handling exceptions [here](https://dzone.com/articles/9-best-practices-to-handle-exceptions-in-java)

The application uses different Java types (String, Double and Dates) to store data on objects.

The application UI uses a switch statement to handle the user input flow.

[Download Project](https://review-api.udacity.com/api/v1/submissions/3696700/archive)

A person with the arms crossed

Description automatically generated with medium confidence

**Best practices for your project resubmission**

Ben shares 5 helpful tips to get you through revising and resubmitting your project.

[Watch Video](https://review.udacity.com/#!/reviews/3696700) (3:01)

Rate this review