Object Oriented Programming (67125), 2019

Exercise 3: Testing in Java

Tests are important! Taken from Programmers Jokes on Facebook

Contents

1 Objectives

By the end of this exercise, you should:

1. Know how to write your own set of tests for upcoming exercises.
2. Have practiced *Test Driven Development* (TDD).
3. Be able to make more informed design decisions, and explain them better.

2 Importing Files

In order to write tests, we need to import code from other files. We will learn more about importing and packages later in the course. For the purpose of this exercise, we have prepared a list of files you may need to import, and the proper syntax.

> Import code you are not using is considered bad practice. In case you do, you will get appropriate warnings when trying to compile. As we have not covered this material in class yet, in this exercise, and this exercise only, this will not result in point deduction in this exercise.

- We will be providing you with a jar file called `ex3_resources.jar`. In order to use the files that are relevant for each portion of the exercise, in the IntelliJ IDE you should go to File → Project Structure → Libraries, click the plus sign, select *Java* and browse for the supplied jar. The jar file contains many resources. The ones relevant for you are:
  - `oop.ex3.spaceship`: this package contains files pertinent to Section 3.1.
  - `oop.ex3.searchengine`: this package contains files pertinent to Section 3.2
  - `org.junit`: this package contains the files you need in order to run tests in Java.

  To import a package, similarly to the import statements you used in Intro to CS, add "import ⟨*package*⟩.*;" at the beginning of your file. For example:

  ```java
  import oop.ex3.spaceship.*;
  ```

- When writing a test file, remember to import the JUnit package from the jar file using:

  ```java
  import org.junit.*;
  ```

- For the actual testing functionalities, such as verifying values, you will also need to use methods provided by the `org.junit.Assert` class. These can be imported using the following:

  ```java
  import static org.junit.Assert.*;
  ```

3 Exercise Definition

Attention!

Remember that we are working according to the TDD approach - so don't just start implementing. As usual, read through the entire exercise, and implement in the instructed order.

3.1 Starship Depository

Background

On board the starship *USS Discovery* there are multiple lockers to keep items the crew might need at any point in time.

A locker can contain different types of items. Each of these items have a unique identifying type, such as "baseball bat" or "helmet, size 3". All items of the same type take up the same amount of storage units in the locker. Storage units are positive integers.

Each locker has a **capacity**, which is the total amount of storage units it can hold. The capacity of a locker is a non-negative integer[1], and cannot be changed once it has been set. Each item can only be added in full - half an item, or any other percentage of an item, cannot be stored in a locker.

In addition to the aforementioned lockers, the USS Discovery also has a single centralized long-term storage, which has a capacity of 1000 storage units. If items of a specific type take up **more** than 50% of the storage units of a specific locker, some of them are automatically moved to the long-term storage. The remaining amount should only take **up to** 20% of the storage units of that locker (i.e. the remaining number is the maximal number which occupies up to, and including, 20% of the locker's capacity).

Given Code

Within the `oop.ex3.spaceship` package, you will find the following files:

- **`Item.java`**: an implementation of an item. See the API of the class.
- **`ItemFactory.java`**: a class to provide you with legal items. See the API of the class.

USS Discovery Locker actions

- **`public Locker(int capacity)`**. This constructor initializes a Locker object with the given capacity.
- **`public int addItem(Item item, int n)`**. This method adds `n` Items of the given type to the locker. If the action is successful, this method should return `0`. If `n` Items cannot be added to the locker at this time, no Items should be added, the method should return `-1`, and the following message should be printed to `System.out.println`: "*Error: Your request cannot be completed at this time. Problem: no room for* ***`n`*** *items of type* ***`type`***". If this action causes Items to be moved to long-term storage, the method should return `1`, and the following message should be printed to `System.out.println`: "*Warning: Action successful, but has caused items to be moved to storage*". In case the long-term storage is full, no Items should be added, the method should return `-1`, and the following message should be printed to `System.out.println`: "*Error: Your request cannot be completed at this time. Problem: no room for* ***`n`*** *Items of type* ***`type`*** *in the long-term storage.*

[1] this is a working assumption, meaning you can assume that Lockers won't be initialized with a non-positive capacity. However, you may enforce it if you wish to.

- **`public int removeItem(Item item, int n)`**. This method removes `n` Items of the type `type` from the locker. If the action is successful, this method should return `0`. In case there are less than `n` Items of this type in the locker, no Items should be removed, the method should return `-1`, and the following message should be printed to `System.out.println`: "*Error: Your request cannot be completed at this time. Problem: the locker does not contain `n` items of type `type`*". In case `n` is negative, no Items should be removed, the method should return `-1`, and the following message should be printed to `System.out.println`: "*Error: Your request cannot be completed at this time. Problem: cannot remove a negative number of items of type `type`*".

- **`public int getItemCount(String type)`**. Returns the number of Items of type `type` the locker contains.

- **`public Map<String, Integer> getInventory()`**. Returns a map of all the item types contained in the locker, and their respective quantities. For example: {"Baseball bat" : `1`, "helmet, size 3" : `5`}.

- **`public int getCapacity()`**. Returns the locker's total capacity.

- **`public int getAvailableCapacity()`**. Returns the locker's available capacity, i.e. how many storage units are unoccupied by Items.

USS Discovery Long-Term Storage actions

- **`public LongTermStorage()`**. This constructor initializes a Long-Term Storage object.

- **`public int addItem(Item item, int n)`**. This method adds `n` Items of the given type to the long-term storage unit. If the action is successful, this method should return `0`. If `n` Items cannot be added to the locker at this time, no Items should be added, the method should return `-1`, and the following message should be printed to `System.out.println`: "*Error: Your request cannot be completed at this time. Problem: no room for `n` Items of type `type`*".

- **`public int getItemCount(String type)`**. Returns the number of Items of type `type` the long-term storage contains.

- **`public Map<String, Integer> getInventory()`**. Returns a map of all the Items contained in the long-term storage unit, and their respective quantities.

Note: Maps in Java

A `Map` is an object that maps keys to values, similar to Python's Dictionaries. It is used to store key/value pairs, where all keys must be of a specific type and all values have the same type (as usual in Java). Note that there's no guaranteed order between pairs. Also, while a `Map` **cannot** hold duplicate keys, it can hold duplicate values.

Java's `Map` is an interface, implemented by classes that represent concrete data structures. An example for creating a `Map` is as follows: `Map<Key, Value> map = new HashMap();`, where `HashMap` is a popular implementation of `Map`. Key and Value are any non-primitive types, and represent the type of keys and values stored in the `Map`, respectively.

In this exercise, we will use `String`s as our keys, and `Integer`s as our values. According to the `Map` interface, to add a pair use the **`put(String key, Integer value)`** method. You may use the **`remove(String key)`** method to remove a pair, and **`get(String key)`** to get a pair.

We will learn more about the different types of `Map`s available in Java, as well as other data structures, in the following weeks.

- **`public int getCapacity()`**. Returns the long-term storage's total capacity.
- **`public int getAvailableCapacity()`**. Returns the long-term storage's available capacity, i.e. how many storage units are unoccupied by Items.

Section A - Writing the Tests (and the tests <u>only</u>)

1. Write a class of tests in a file called `LockerTest.java`. This file should be able to test any possible implementation a `Locker` class (to be provided in a `Locker.java` file) which implements the functionalities described in the Locker actions section of this exercise.
2. Write a class of tests for each of the long-term storage actions listed above. All of these tests should be included in a file named `LongTermTest.java`. This test file should be able to detect all possible bugs to a possible implementation of a long-term storage class.
3. Write a test suite for the entirety of your code in a file called `SpaceshipDepositoryTest.java`.

Section B - Implementing Storage

4. Implement a locker in a file called `Locker.java`. It, of course, should include all the functionalities described above. In the README file, explain your design choices. How did you choose to store the information? Why did you prefer it to other methods?
5. Implement the long-term storage in a file called `LongTermStorage.java`. It, of course, should include all the functionalities described above. In the README file, explain your design choices. How did you choose to store the information? How is it different from `Locker.java`?

Section C - Plot Twist!

A new type of item was introduced by a crew member of the starship: a `football`. A football takes up 4 storage units. The problem is, a `football` and a `baseball bat` cannot reside in the same locker. When a crew member wishes to store an Item in a locker, this constraint should be checked **as a first step**. In case the constraint is violated, no Items should be added, the **`addItem`** method of the `locker` should return `-2`, and the following message should be printed to `System.out.println`: "*Error: Your request cannot be completed at this time. Problem: the locker cannot contain items of type* ***`type`****, as it contains a contradicting item*".

Note that footballs and baseball bats **can** both reside in the **long-term storage**.

6. Add a test to `LockerTest.java` for this new functionality.

7. Implement this new functionality in your existing code.

3.2 Hotel Search Engine

Preface

Comparison between objects can be complicated, especially when dealing with complex objects involving multiple members. In this exercise, we will be using one of the most popular tools for this purpose - the `Comparable<T>` and `Comparator<T>` interfaces.

> **Note: Generics in Java**
> What should we do if we want to write an interface that could apply to multiple types of objects? We use *generics*!
> A generic class defines one or more type parameters. When **creating** a class which **implements** a generic interface, you should replace the parameter(s) with an actual (non primitive) Java type(s). For example, let's take a look at a class you already know:
> `public final class String extends Object implements Serializable, Comparable<String>, CharSequence`
> `String` implements the interface `Comparable<T>`, where the type parameter `T` has been replaced with the class `String`. This allows comparison between different String objects, based on their "natural" (in this case, lexicographic) order.
> You can also create a class which is comparable with other matching types, for example: `class Student implements Comparable<Person>` where the `Student` class extends `Person`.
> As you can imagine, the more we delve into inheritance and polymorphism, the more we'll be able to use this mechanism. We will see more about generics in the following weeks.

The `Comparable<T>` interface contains a single method, namely `int compareTo(T o)`. This interface is used to create a single order, which is often referred to as the "natural" order - the default way to compare between objects of this type. The `Comparator<T>` interface is used to define other order relations, which may be very different from the default order. Using comparators, we can enforce other order relations on specific data structures.

While objects should have a single "natural" order, programmers might want to, when needed, save items in a different order, better suited for their current needs. The Comparator interface allows us to create many different ways to sort an object.

Background

Booping.com is a new hotel booking site, that allows for personalized search methodologies. In this part of the exercise, you will implement a portion of the website's actions. Specifically, you will provide the users with the ability to get a list of hotels based on different parameters. The hotels you are provided with are a part of a larger hotel dataset.

> A *dataset* is a collection of data, commonly used for specific tasks. An entry to a dataset can be an image, text, a table row (also known as a *tuple*), an audio spectogram, etc.. All entries in a specific dataset have the same structure.
> In the context of this exercise, the dataset is tabular, where each entry has many *attributes*, each providing a different kind of information.

Provided Files

The `oop.ex3.searchengine` package (located in the provided jar file) contains the following files:

- `Hotel.java` - contains an implementation of a Hotel object, with all the attributes defined in the dataset. You can assume that all textual fields contain text in lower-case. See the API of the class.
- `HotelDataset.java` - a selection of utility functions that allows you to read the dataset and generate an array of Hotel objects out of it. Specifically, the only function you must use is `getHotels(String fileName)`. See the API of the class.

In addition, the jar includes the following files:

- `hotels_dataset.txt`, which constitutes as your dataset. The file contains more than 3000 records of hotels and other types of guest houses in India. Each entry has a lot of information, including the hotel's name, location, star-rating, facilities, surrounding points-of-interest (POI) and so on.
- `hotels_dataset1.txt` and `hotels_dataset2.txt`, which are shorter versions of the full dataset (contain all the attributes but only some of the entries). You are encouraged to use these for testing purposes.

BoopingSite actions

- **`public BoopingSite(String name)`**. This constructor receives as parameter a string, which is the name of the dataset (e.g. `"hotels_dataset.txt"`). This parameter can later be passed to the **`HotelDataset.getHotels(String fileName)`** function.
- **`public Hotel[] getHotelsInCityByRating(String city)`**. This method returns an array of hotels located in the given city, sorted from the highest star-rating to the lowest. Hotels that have the same rating will be organized according to the alphabet order of the hotel's (property) name. In case there are no hotels in the given city, this method returns an empty array.
- **`public Hotel[] getHotelsByProximity(double latitude, double longitude)`**. This method returns an array of hotels, sorted according to their (euclidean) distance from the given geographic location, in ascending order. Hotels that are at the same distance from the given location are organized according to the number of points-of-interest for which they are close to (in a decreasing order). In case of illegal input, this method returns an empty array.

- **`public Hotel[] getHotelsInCityByProximity(String city, double latitude, double longitude)`**. This method returns an array of hotels in the given city, sorted according to their (euclidean) distance from the given geographic location, in ascending order. Hotels that are at the same distance from the given location are organized according to the number of points-of-interest for which they are close to (in a decreasing order). In case of illegal input, this method returns an empty array.

> **Attention!**
>
> You can assume that the coordinates are given in the same format of the dataset's coordinates, in WGS-84 Geo, and are parsed as **double** values. This means that latitudes below the equator, and longitudes left of the prime meridian are described by negative coordinates.

Section A - Testing

1. Implement a test class in a file `BoopingSiteTest.java`. This class should test all the functionalities mentioned in the description above.

Section B - Compare by Rating

2. Implement the method **`getHotelsInCityByRating`** in the file `BoopingSite.java`. Explain your design decisions in the README file.

Section C - Compare by Location

3. Implement the method **`getHotelsByProximity`** in the file `BoopingSite.java`.

4. Implement the method **`getHotelsInCityByProximity`** in the file `BoopingSite.java`.

5. Explain your design decisions in the README file. What were your options? Why did you prefer one over another?

4 Further Guidelines

- This exercise includes parsing data from json file (don't worry, it's part of the code that is provided in the jar file). After configuring it as instructed in Section 2 your IDE should recognize imports such as *import com.google.gson.Gson*.

- A test file will be given full points **only** if it manages to catch all bugs. Note that this also means that false negatives, i.e., a scenario that should pass but fails, also constitutes a wrong implementation and will be treated as such (as in the real world). With this being said, you are **not** expected to test properties that you have not learned yet, such as printouts and thrown exceptions. Note that our tests nevertheless will test for those.

- You may **NOT** force any execution order on the tests within a test class. This is generally a bad practice.

- You may choose to use any `Map` implementation that is provided to you by Java (i.e. appears as part of the `java.util` library). Those are: HashMap, LinkedHashMap and TreeMap. You do **not** have to explain your choice in the README file, choose whichever implementation is appropriate for your needs and easier for you to work with.

- Your submission should at the very least include all the files listed in Section 3. You are free to add as many other java files you deem necessary for your design, but be sure to document these decisions.
- Whenever you encounter a scenario that is not covered by the description, but could potentially take place, the following message should be printed to `System.out.println`, and the program should promptly exit with an error code (-1). Later in the course, we'll learn how to deal nicely with bugs.

5 General Instructions

- **Start early**. In fact, start **today**. We know we keep saying this, but trust us, we really mean it.
- Read this document, write yourself notes, and repeat at least 3 times. Exercises descriptions, like *Product Requirements Documents* (PRDs) you'll see when you start working, are hard to understand and easy to misinterpret.
- Document your code using the Javadoc documentation style, as described in the coding style guidelines and exampled in the code. You should check yourself by invoking: `javadoc -private -d doc *.java` within your project directory or by running the designated tool in your IDE.
- Remember encapsulation.
- Write your code according to the styling guidelines.
- Notice that, similarly to the previous exercises (and even though this one contains two unrelated tasks), you are required to write all your code in the source directory of your project (the *src* folder).

6 Submission

1. Deadline: **Thursday, April 18 2019, 23:55**
2. Make sure you do not submit any .class, Javadoc or the provided files by mistake.
3. Create a JAR file named ex3.jar containing these files.
4. Make sure that the JAR file you submit passes the presubmit script by **carefully** reading the response file generated by your submission. **Exercises failing this presubmit script will get an automatic 0!**