Shop - Functional design

The shop is the backend system of an online store like eBay or Amazon. The part of the store we are creating gets instructions from a customer that wants information from the database and gives back the information requested.

I have started by creating the database, exceptions and necessary classes. I've also created the add, remove and update methods + tests for you so you can see the workflow.

I recommend you install the Lombok plugin in IntelliJ (you can install via IntelliJ plugins menu) so you can use snacks like checkNotNull() or annotations if you should choose to use them.
SonarLint is also a useful code quality plugin, but quite aggressive.

The point of this exercise is to use test driven development and functional programming in Java 8 to improve performance, create clean code and generally make life a little easier for the jvm and whoever takes over the code after you.

1. Start by creating the methods you think you will need by looking at the test requirements and workflow. Make sure all methods return null unless they are void.
2. Create a test (move the caret to the class name and press ALT+ENTER and "create test"). Finish the tests and run them. Almost all should fail.
3. Implement the body of the methods in the actual classes, remember to use the Stream api where applicable.

Follow the steps above for both components, you should only need to work in the **ShopRepository** and **ShopService** classes, however, test driven development promotes the use of more methods and classes, so create as many classes and methods as you think you will need (for example an input validator class?/method? Or maybe it's logical to have more methods in the repository layer?).

Good luck!

Shop - Repository

Functional rules

1. Receive instructions from the service layer
2. Perform CRUD operations
3. Return an object or list based on the instructions
    a. If there are no results, proceed with the error handling.

Test requirements

The repository should return

1. An item by id
2. A list of all items
3. A list of items by ids from range **x** to **y**
4. A list per item for location **x**
5. A list per item for type **x**
6. A list per item for producer **x** (eg hugo_boss)
    a. Assume the input will be “hugo boss” (no underscore)

The repository should be able to

7. Remove an item (using the itemsID)
8. Add an item
9. Update an item

Error handling

No items match the given criteria	Return null or empty list

Shop - Service (or Core)

Functional rules

1. Validate search values to ensure they are not null or empty
    a. If a value is missing, throw an **InvalidCriteriaException** with message “Input was null, empty or lower than 0.”.
2. Proceed with the call
3. Validate the object returned from the repository.
    a. If the reply is null, throw a **NoItemsFoundForCriteriaException** with message “No items were found for the given search criteria.”.
    b. If not null, return the object.
4. Return the information requested

Test requirements

The service should have methods for

1. Getting a Map<ItemLocation, List<Item>> per Location
2. Getting a Map<ItemType, List<Item>> per Type
3. Getting a Map<String, List<Item>> per producer
4. Getting a Map<Boolean, List<Item>> where one list has items under 1500 in stock and one over.
5. Getting an item by id (hint: Optional)
6. Getting a String of all the producers separated by **x**
7. Getting a list of all locations with more than **x** in stock
8. Getting a list of all locations with less than **x** in stock
9. Getting a list of items for **x** location with more than **y** in stock
10. Getting a list of items for **x** location with less than **y** in stock
11. Getting a list of all items where the name (not producer) starts with **x**
12. Finding the average item stock for location **x**
13. Finding which item is most in stock
14. Finding which item is least in stock
15. Finding all items for location **x** that have more than **y** in stock
16. Finding all items sorted alphabetically by **producer**
17. Finding all items sorted alphabetically by **name** (not producer)
18. Finding all items sorted by stock from **high** to **low**
19. Finding all items without any duplicates
20. Creating a single list from two lists where the first list gets items from range **a** to **b** and the second list gets items from range **x** to **y** (hint, see Stream.of, use the repository methods you created earlier)
21. Creating a single list from three lists where the first list is of item location **a**, the second list is of item type **b**, the third list is of item producer **c**, no duplicates
22. Get the total stock for all items
    a. If empty, return 0

Optional

1. Create more methods in the service layer, experiment with the Stream api and perhaps even parallel streams.
2. Create a provider (or app) layer that has a class with a main method which can run the methods in the service class on the larger database in the common area.