Exercises: Unit Testing

This document defines the exercise for the "Java Advanced" course @ Software University.

Problem 1. Database

You are given a simple class - **Database**. It **stores Integers**. Your task is to **test the class**. W**rite tests**, so you are sure its methods are working as intended.

Constraints

- Storing array's **capacity** must be **exactly 16 integers**.
  - If the size of the array is not 16 integers long, throw **OperationNotSupportedException**.
- **Add** operation, should **add an element at the next free cell.** (just like a stack)
  - If a passed element is null throw **OperationNotSupportedException**.
- **Remove** operation, should support only removing an element **at the last index**. (just like a stack)
  - If you try to remove an element from an empty Database throw **OperationNotSupportedException**
- **Constructors** should take integers only, and store them in an **array.**
- **The fetch method** should return the elements as an **array.**

Hint

Do not forget to **test the constructor(s)**. They are methods too!

Problem 2. Extended Database

You already have a class - Database. This time it is extended. There are the following provided methods: adding, removing, and finding People. In other words, it should store People. There should be two types of finding methods - first: findById (long id) and the second one: findByUsername (String username). As you may guess, each person should have its own unique id and unique username. Your task is to test these functions.

Constraints

The database should have methods:

- add
  - If there are multiple users with this id, throw **OperationNotSupportedException.**
  - If negative nor null ids are present, throw **OperationNotSupportedException**.
- remove
- findByUsername
  - If no user is present by this username, throw **OperationNotSupportedException**.
  - If the username parameter is null, throw **OperationNotSupportedException.**
  - Arguments are all CaseSensitive!
- findById
  - If no user is present by this id, throw **OperationNotSupportedException**.

Hint

Do not forget to test the constructor(s). They are methods too!

Problem 3. Iterator Test

You are given a simple class "ListIterator", it should receive the collection (Strings) which it will iterate, through its constructor. You should store the elements in a List and get them initially through its constructor. If there is a null passed to the constructor, throw a new **OperationNotSupportedException**. The class should have three main functions:

- **Move** - should move an internal index position to the next index in the list, the method should return true if it successfully moved and false if there is no next index.
- **HasNext** - should return true if there is a next index and false if the index is already at the last element of the list.
- **Print** - should print the element at the current internal index, calling Print on a collection without elements should throw an appropriate exception with the message "**Invalid Operation!**".

Command	Return Type	Description
Move	boolean	This command should move the internal index to the next index.
Print	void	This command should return the element at the current internal index.
HasNext	boolean	Returns whether the collection has a next element.

Test

Create tests, so you are sure all methods in the class ListIterator are working as intended.

Constraints

- There will always be only **1 Create** command and it will always be the first command passed.
- The last command will always be the only **END** command.

Examples

Input	Output
Create Print END	Invalid Operation!
Create Stefan George HasNext Print Move Print END	true Stefan true George
Create 1 2 3 HasNext Move HasNext HasNext	true true true true true

Move HasNext END	false

Problem 4. **Bubble Sort Test

There is a sorting algorithm - Bubble Sort. You could read this article, to get a better idea: Bubble Sort.

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items, and swaps them if they are in the wrong order. The pass of the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

You are given a simple class "Bubble". Create a Test class and test, is it working as intended. Think about the border cases.

Problem 5. Custom Linked List

You are given a data structure that needs to be tested. Use the Java file **CustomLinkedList.java** and:

- Create Test Class and Test Methods for **all public members** that need testing.
- Create tests that ensure all methods, getters and setters **work correctly**.
- Use the **`@Test(expected = ExpectedException.class)`** annotation for methods that are expected to throw an exception in case wrong input is entered (those tests don't need to assert messages).
- Give **meaningful assert messages** for failed tests.

Problem 6. Tire Pressure Monitoring System

You are given a small project for a system that monitors the pressure in car tires. Your task is to write unit tests for the system. You will need to use mocking to pass dependencies. Think about the corner cases of the project.