# Exercises: Unit Testing

This document defines the exercises for ["C# OOP Advanced" course @ Software University](https://softuni.bg/courses/csharp-oop-advanced-high-quality-code).

## Database

Create a simple class - **Database**. It should **store integers**. You should **set the initial integers by constructor**. Store them **in array**. Your Database should have a functionality to **add**, **remove** and **fetch all stored items**. Your task is to **test the class**. In other words, create the class and **write 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 **InvalidOperationException**
* **Add** operation, should **add an element at the next free cell** (just like a stack)
  + If there are 16 elements in the Database and try to add 17th, throw an **InvalidOperationException**
* **Remove** operation, should support only removing an element **at the last index** (just like a stack)
  + If you try to remove element from empty Database throw **InvalidOperationException**
* **Constructors** should take integers only, and store them in **array**
* **Fetch method** should return the elements as **array**

### Hint

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

## Extended Database

You already have a class - **Database**. Now your task is to modify and extend it. It should support, **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 implement these functions and test them.

### Constraints

Database should have methods:

* Add
  + If there are already users with this username, throw **InvalidOperationException**
  + If there are already users with this id, throw **InvalidOperationException**
* Remove
* FindByUsername
  + If no user is present by this username, throw **InvalidOperationException**
  + If username parameter is null, throw **ArgumentNullException**
  + Arguments are all **CaseSensitive**
* FindById
  + If no user is present by this id, throw **InvalidOperationException**
  + If negative ids are found, throw **ArgumentOutOfRangeException**

### Hint

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

## Iterator Test

Create a 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 null passed to the constructor, throw new ArgumentNullException(). 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!**".

By default, the internal index should be pointing to the **0th index** of the List. Your program should support the following commands:

|  |  |  |
| --- | --- | --- |
| **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 Stefcho Goshky  HasNext  Print  Move  Print  END | True  Stefcho  True  Goshky |
| Create 1 2 3  HasNext  Move  HasNext  HasNext  Move  HasNext  END | True  True  True  True  True  False |

## \*\*Bubble Sort Test

There is a sorting algorithm - Bubble Sort. You could read this article, to get better the idea: [Bubble Sort](https://en.wikipedia.org/wiki/Bubble_sort#Analysis).

**Bubble sort**, sometimes referred to as **sinking sort**, is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that repeatedly steps through the list to be sorted, compares each pair of adjacent items and [swaps](https://en.wikipedia.org/wiki/Swap_(computer_science)) them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a [comparison sort](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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.

Create a class "Bubble", and try to implement the sorting method yourslef. Create a Test class and test, is it working as intended. Think about the border cases.