# BashSoft – OOP – Advanced

This document defines the project overview for the ["C# OOP Basics" course @ Software University”](https://softuni.bg/courses/csharp-oop-advanced-high-quality-code). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/courses/csharp-oop-advanced-high-quality-code).

# Interfaces and Abstraction

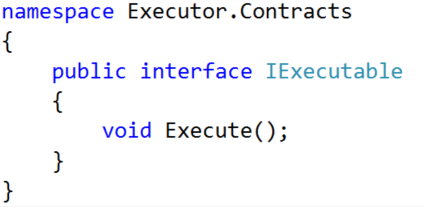
## Introduction

So far, we've learned a lot about **classes**, their **members** and the first three principles of OOP - **Encapsulation**, **Inheritance** and **Polymophism**. We've employed all of that and even more in refactoring our **BashSoft** application. Now it is time to go one step further and employ the fourth OOP principle in our code - namely **Abstraction**. We will do this by creating several **interfaces** that will correspond to the classes we have. We will also try to **segregate** them somewhat into more than one **interface** per class at the times we notice a possibility.

## Creating Interfaces for the IO Package

As you've learned during the lecture the purpose of an **interface** is to define what methods a certain class needs to have in order to be sufficient for a certain task. Thus, all methods in an interface are considered public and abstract.

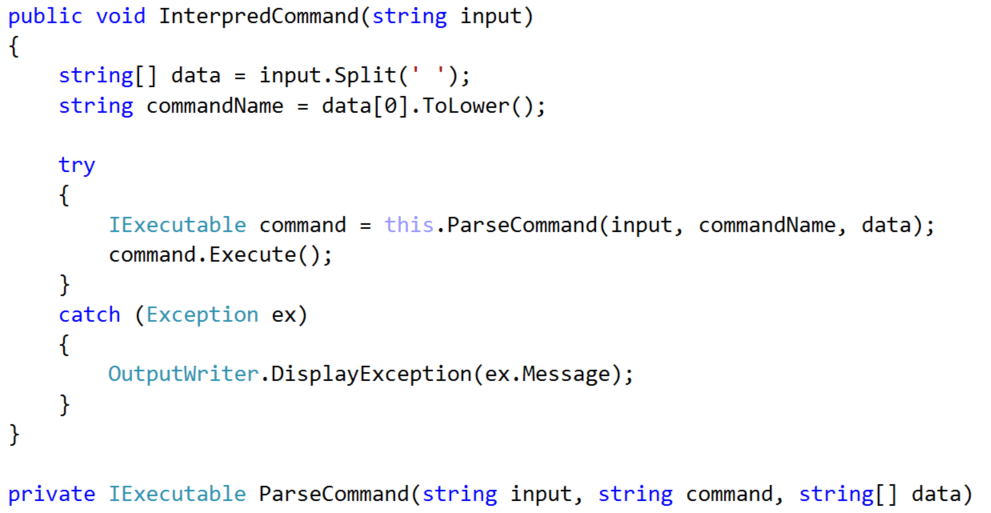
Let's start with creating our first **interface** that will define what a command should have. Create a new package called "**Contracts"** and inside of it - an **interface** called "**IExecutable**". Then define the proper method signature for the execute method our commands have:



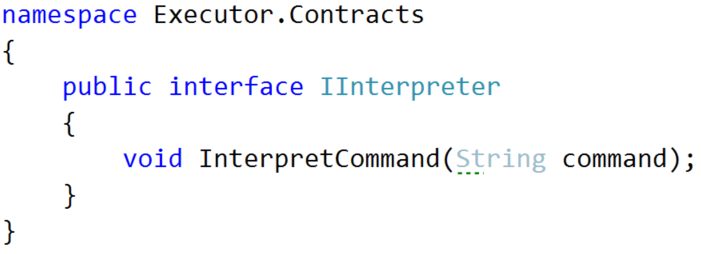
Now implement the method in our abstract **Command** class:



If you have defined the **interface** correctly there shouldn't be any issues. You could also implement the interface in the derived commands. Although this would be redundant, the code will be considered more readable. Finally replace the abstract **Command** in every **field**, **method** or **constructor** **return type / parameter**. In this way we will raise the **abstraction** level of our application:

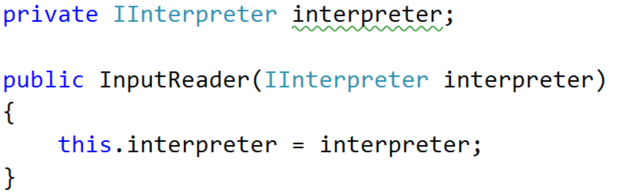


Our next **Interface** will be for the **CommandInterpreter**. You can call it "**IInterpreter**" and define in it the public method of our class:



Don't forget to **implement** the **interface** in the **CommandInterpreter** and raise the abstraction level in **fields**, **methods**, **constructors** and **return types / parameters**.

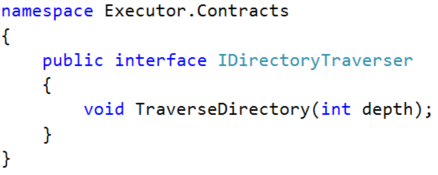
In the **InputReader:**



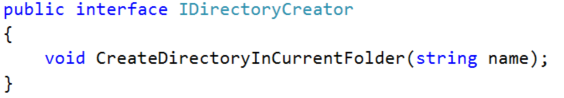
In the **main** method:



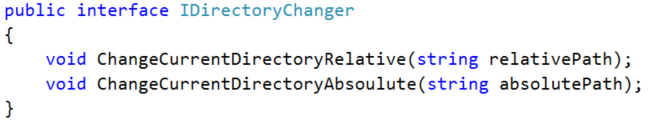
Next up is the **IOManager -** it is a good place to employ some **interface segregation.** First create an **interface** called **IDirectoryTraverser** - it will hold the **TraverseDirectory** method:



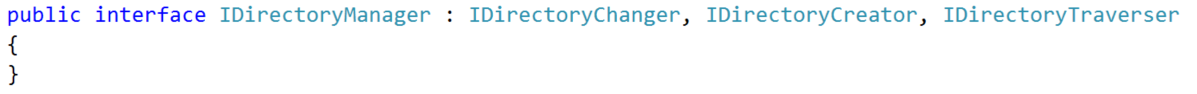
Create an **interface** called **IDirectoryCreator** for the **CreateDirectoryInCurrentFolder** method:



Finally, create an **interface** called **IDirectoryChanger** for the remaining two methods:



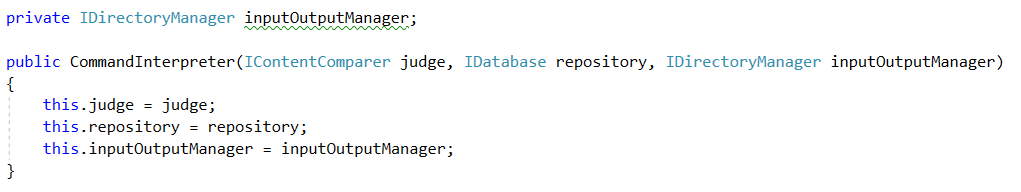
Creating these three **interfaces** would allow us to oblige different classes to implement **only** the interfaces that are **related** to their actions. However, in the case of our **IOManager** we will need all three of them, so we will combine the three interfaces in a single one:

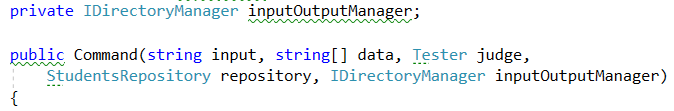


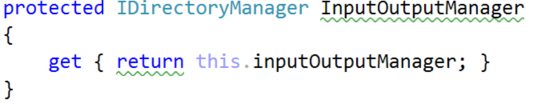
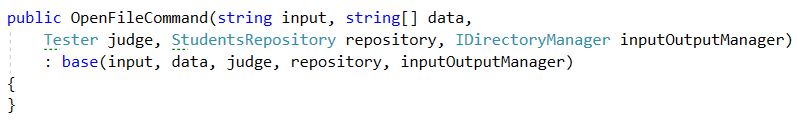
Implement it like this:



Finally replace **IOManager** with I**DirectoryManager** in all **fields**, **methods** or **constructor** **return types / parameters** in the **CommandInterpreter** and all the **Command** classes:





Finish the rest of the command classes yourself. (*Hint* *->* use the VS search tool)

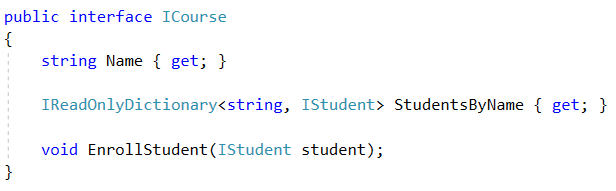
Make an interface for the **InputReader** by yourself. You can call it **IReader** with method **StartReadingCommands**. Don't forget to implement it and replace **InputReader** with its interface where possible… (*psst* - the main method).

We will not make an interface for our **OutputWriter** because it is not meant to be an instance class - it only has static methods and those don't belong to interfaces.

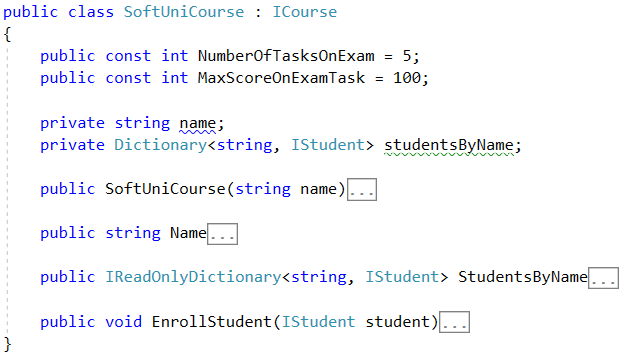
## Creating Interfaces for the Models Package

Let's begin by changing the name of both the classes to **SoftUniStudent** and **SoftUniCourse**.

We are doing this because we need the distinguish them from the more **abstract** names for the **underlying interfaces** we are about to create. Now let's do just that - create interfaces called **ICourse** and **IStudent.** They will have all the public methods of our **SoftUni** classes. This is how the **ICourse** interface should look like (notice how we use the **IStudent** interface instead of the **SoftUniStudent** class in the method signatures):



And the **SoftUniCourse** class should now look like this:



Do the **IStudent** interface and **SoftUniStudent** classes by yourself, following the pattern:

1. Put all **public** method signatures of the **class** in the **interface**.
2. Implement the interface in the class
3. Where needed change **Course** with **ICourse**
4. Where needed change **Student** with **IStudent**
5. Fix the **StudentRepository** to work through both the **IStudent** and **ICourse** **interfaces -** you can start off with the fields:



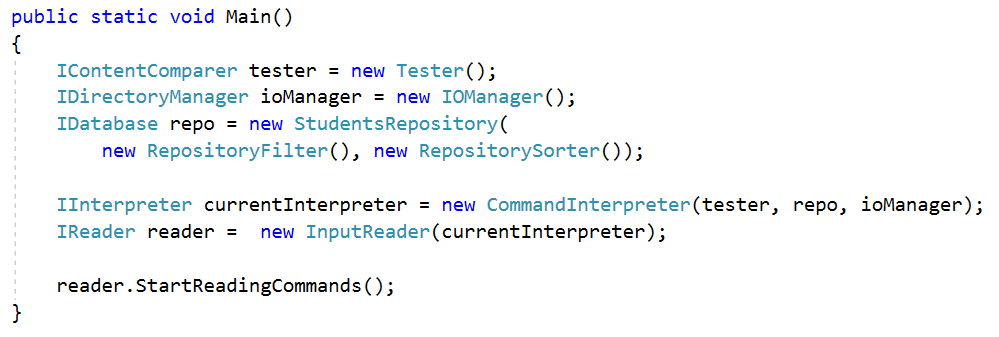
It is a good idea to make sure that the code compiles before proceeding to the final task.

## Creating the Remaining Interfaces

Create the following interfaces by yourself:

* For the **Tester** class
  + **IContentComparer** with method **CompareContent**
* For the **StudentsRepository**
  + **IDatabase** with methods: **LoadData, UnloadData** and extending sub-interfaces:
    - **IRequester** with methods: **GetStudentMarkInCourse, GetStudentsByCourse**
    - **IFilteredTaker** with methods: **FilterAndTake**
    - **IOrderedTaker** with methods: **OrderAndTake**
* For the **RepositorySorter**
  + **IDataSorter** with methods: **OrderAndTake**
* For the **RepositoryFilter**
  + **IDataFilter** with methods: **FilterAndTake**

Once you are done your main method should look something like this:



If the application still compiles at the end - congratulations you have completed the lab for **Interfaces** **and Abstraction**!

# Generics, Iterators and Comparators

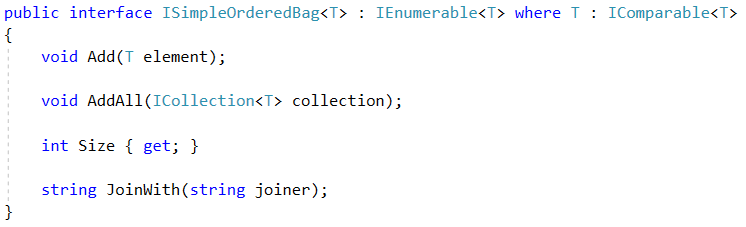
## Introduction

In the last few parts all we've been doing is refactoring the **BashSoft** application with all kinds of good practices of **OOP** and **OOD** (object oriented design). We must admit that although necessary, that has been kind of boring. Now that we've learned about **Generics**, **Iterators** and **Comparators** it's a good moment to finally add some new functionality to our project. We will create our own data structure called **SimpleSortedList.** It will have some normal methods for a list but it will differ slightly because the elements inside it will always be sorted. Kind of like a **SortedSet** with repeating elements. Such an abstract data structure is called an **OrderedBag**.

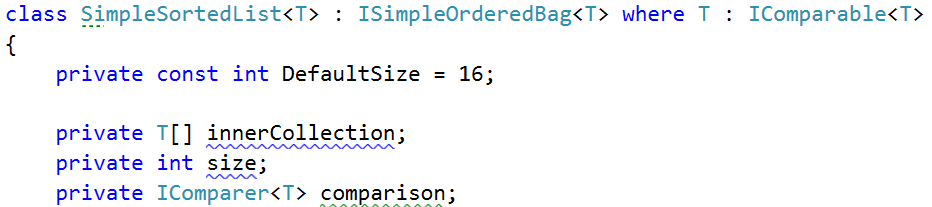
**Note that our implementation will be very slow in terms of performance because that is not the focus of the current material. If you are interested in making it fast you may learn about that and much more in the** [**Data Structures**](https://softuni.bg/courses/data-structures) **course at SoftUni.**

## Implementing the SimpleSortedList Data Structure

Let's start with the interface so we know what features our data structure should have in case we (or someone else) decide to do a different implementation of it. Call the interface **ISimpleOrderedBag.** It will be simple because we don't plan to add much functionality to it for now and it will not have the best implementation as we mentioned. If we decide we need more the interface can be extended. Our interface should have the methods: A**dd**, **AddAll**, **Size** and **JoinWith** (a method which will assist in printing). It should also keep a **generic type** that is **comparable** and the bag itself should be **iterable.** This is how it should look (don't forget about good formatting of empty lines and spaces):

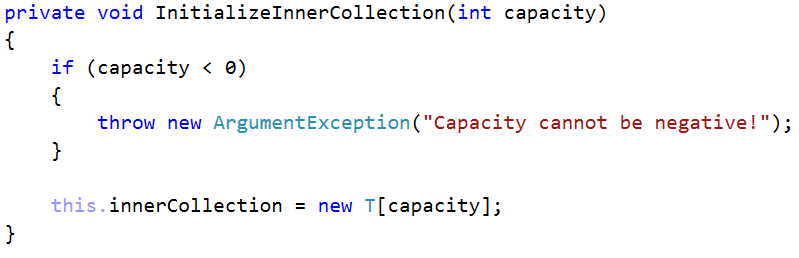


Now that we've created the interface it's time for the class itself. Let's start with creating a new folder called **DataStructures.** In it make a new class called **SimpleSortedList**.We will need three fields inside - one for keeping the **internal collection** (a generic array), one for holding the size of our list, and one for the comparator of our sorted list. Don't forget to also write the correct class signature. We will also need a constant field to keep a default field for our list.

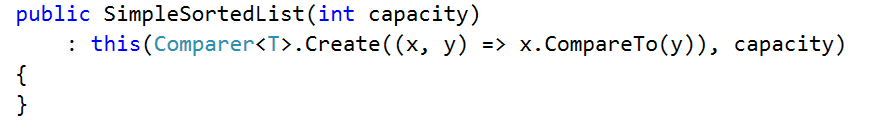


Notice how we have unimplemented methods?

Well we will implement them later because we should do the constructors first. The first one will accept **the most** **parameters** - a comparer and capacity. Don't forget to **validate** the capacity!

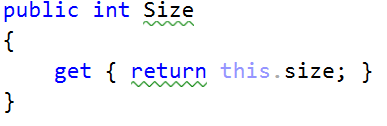


The **second** constructor will have a default comparer:

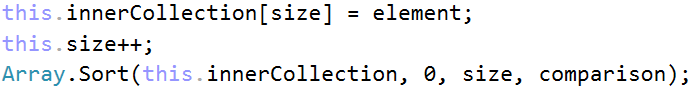


The **third** will have a **default capacity** and the **fourth** one will have both **capacity and comparer to be default**. Implement them by yourself.

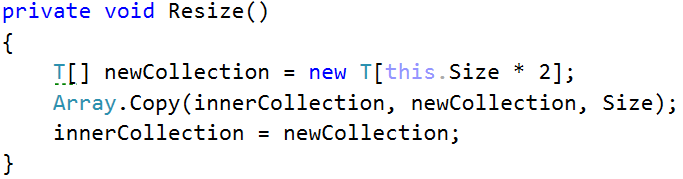
OK, now let's start implementing the interface. The **Size** property should just return our size field as is:



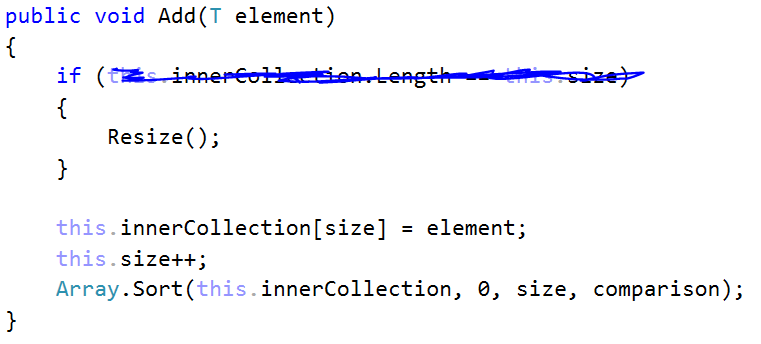
The **Add** method should set the element at the current **size index of our inner collection** to the generic element passed to it. Then **increment the size** and finally **sort the inner collection**, because after all we're creating a **sorted list.**



Well as Nakov says this should work like a dude, except… not always. What happens when our inner array is full? The answer is - **IndexOutOfRangeException.** To prevent this, we need to resize our array. We basically need to copy our array into a new one that is twice as big and leave the empty values to be null. The **Array.Copy** method can do this for us but let's wrap it in our own private method.

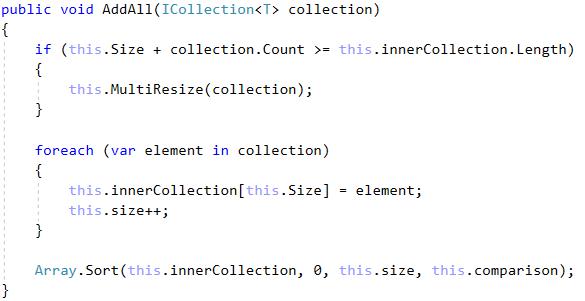


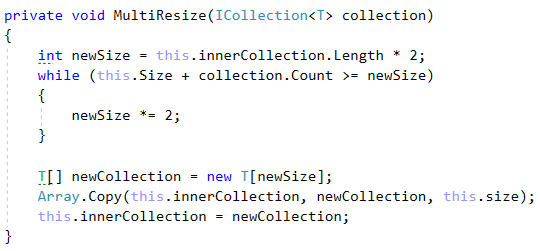
Now all that is left is to call this method when our **size** is bigger or equal to the inner array's length. This is how the **Add** **method** should look in the end:



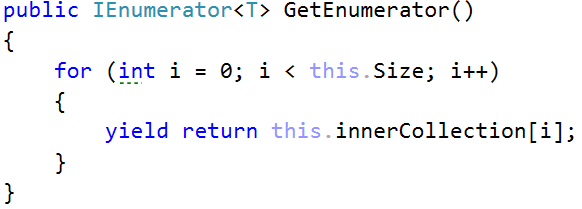
The **AddAll** method will work in a similar fashion as the **add** method. We could even implement it by calling **add** but that would trigger sorting at each element to be added, so a better approach would be to add all the elements and only sort once at the end.

However, resizing in this case might not be so simple because our current elements + the ones we want to add might be more than the inner collection's length \* 2. Thus, we will have a slightly different resize approach (think about the logic behind this):

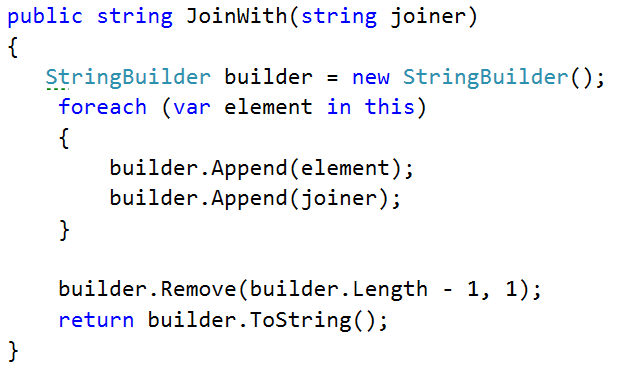




Now that we're done with the main functionality it's a good time to override the **IEnumerator<T> GetEnumerator()** method. Inside we should create a loop through the inner collection and yield return the current element.



The last thing we need to implement is the **JoinWith** method. It will connect all the elements in our structure with the given **joiner** string. Since now we have the iterator, we can reuse it.

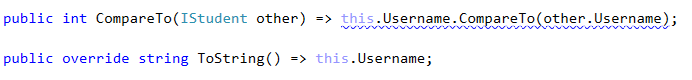


## Making Students and Courses Comparable

As always start by altering the interfaces:



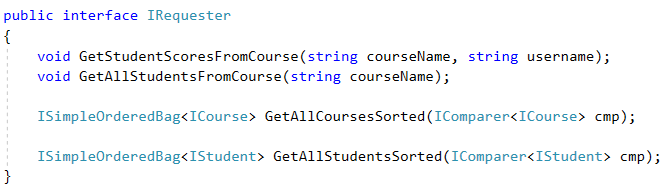
Do the same for the Student interface. Then go to the classes. This would be a good time to **override** the **ToString** method of our SoftUniStudent/Course classes too:



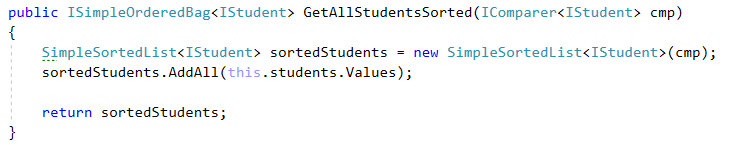
Do the same for the **SoftUniCourse** class, comparing them by **Name**.

## Adding Functionality to the StudentsRepository

The methods we are going to add are most akin to the **Requester** interface which the **Database** interface extends. Thus, we will add our two new method signatures here:



Now implement them in the **StudentRepository:**



Notice how we can add the **values** from our **Dictionary** with the **AddAll** method of our **SimpleSortedList** because they are a **Collection**. The other one is exactly the same but with **Students,** implement it **by yourself**.

## Adding the new DisplayCommand

Start off by adding a case to our **CommandInterpreter's** **parseCommand** method:



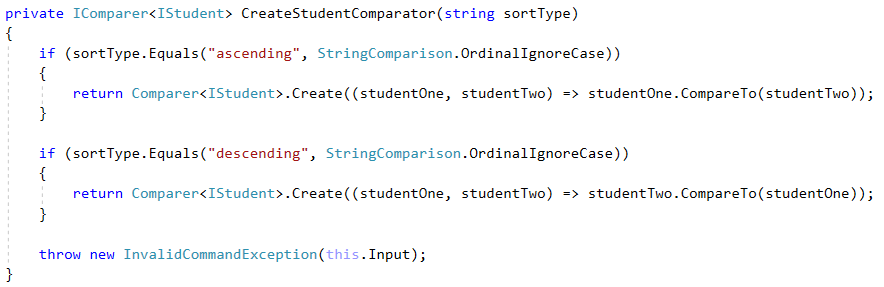
Then create the matching class with the **usual constructor**. You can try to figure out an implementation for its execute command on your own. The command should receive two parameters in its input data - **entity to display** (students / courses) and **the** **order** in which to display the data (ascending/descending).

Try to reuse as much code as you can and use our SimpleSortedList's **JoinWith** command for printing.

Alternatively, you can look at our implementation. We won't go into much detail about it because it is not a subject of this lab:



This is the **CreateStudentComparator** command, implement the other by **yourself:**



Finally go to the **GetHelpCommand** and add some more help info about our new command:



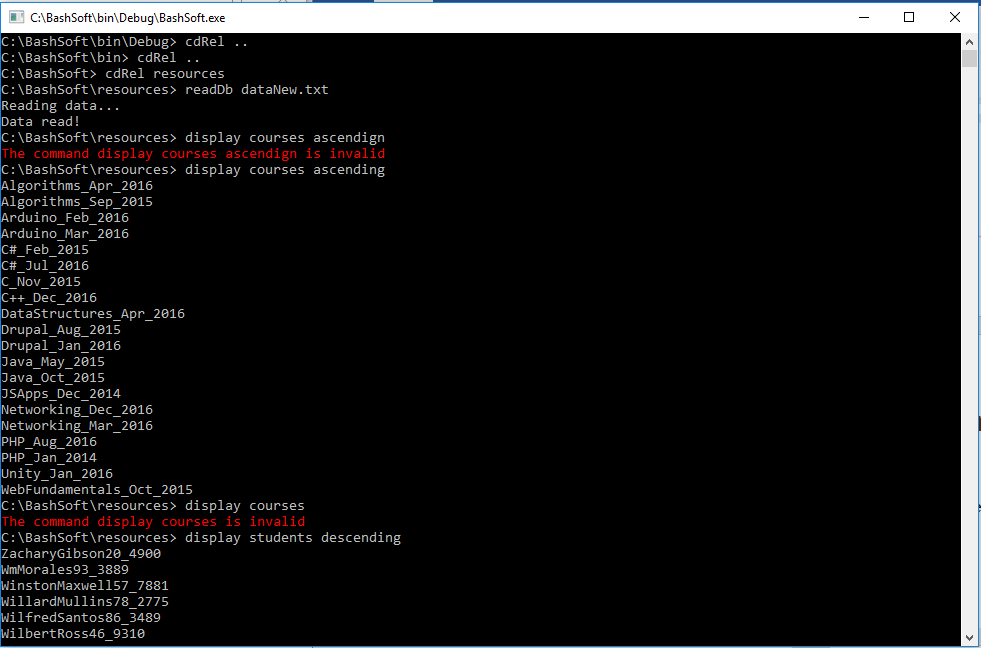
## Testing our new Functionality

If you've implemented all the new functionality correctly you should get such a result:

cdrel resources

readDb dataNew.txt

display courses descending



## \* BONUS TASK: Implement Your Own Sorting Algorithm for the SimpleSortedList

Instead of using the state **Arrays.Sort()** method you can make your own **generic** sorting method inside our class. Use one of the following algorithms:

* **BubbleSort (easy difficulty)**
* **SelectionSort (easy difficulty)**
* **InsertionSort (medium difficulty)**
* **QuickSort (hard difficulty)**

You can research about them all around the internet, but here are some more interesting sources:

<http://visualgo.net/sorting> - great site with visualization of all the algorithms and more + pseudo code. A good programmer should never be **limited** to using just one language and should always be able to read pseudo code.

<https://softuni.bg/trainings/1331/algorithms-april-2016> - check out the lecture by **Atanas Rusenov** about Sorting and Searching Algorithms - 12 April 2016.

# Reflection & Attributes

## Introduction

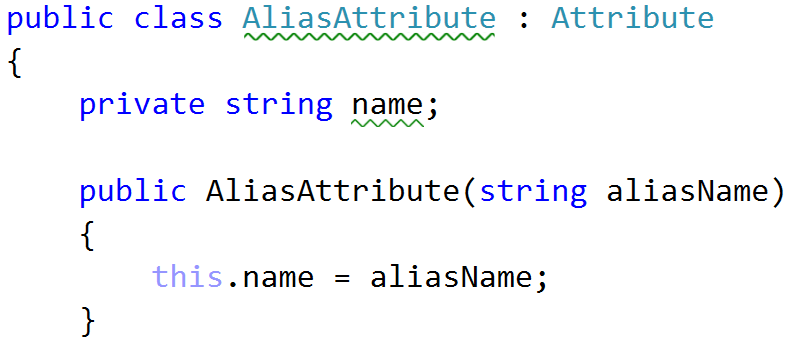
In the current парт, we are going to make the command pattern to be implemented using the reflection API from C#. That way we are going to create instances of the commands without the usage of a switch. This makes our application more flexible, as it allows us to create a new child of the Command class and implement it without changing any existing classes.

## Making the Alias Attribute

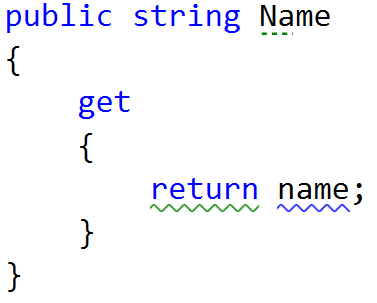
Since we are **not** going to use the switch in our command interpreter, we will need to figure out another way to choose of which command to return an instance. This can be done by creating custom attributes and then scanning those attributes and depending on what value we’ve put in the constructor of the attribute, we will choose which is the desired class to instantiate.

We suggest first creating a new folder called **Attributes.**

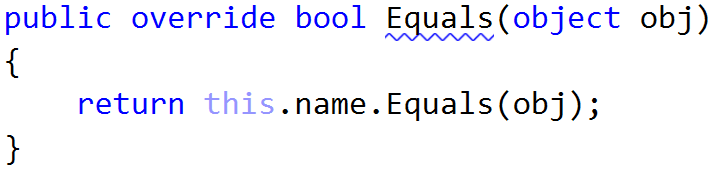
The first attribute we will create is called Alias. It will have one property (name) which will be set by receiving the value for the string in the constructor.



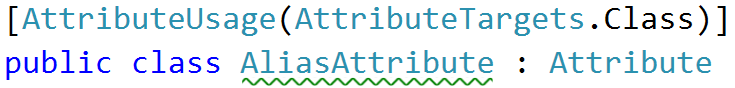
Now we need a getter for the name:



And finally, for our ease we can override the Equals method, by comparing the name of the current instance with the given object.



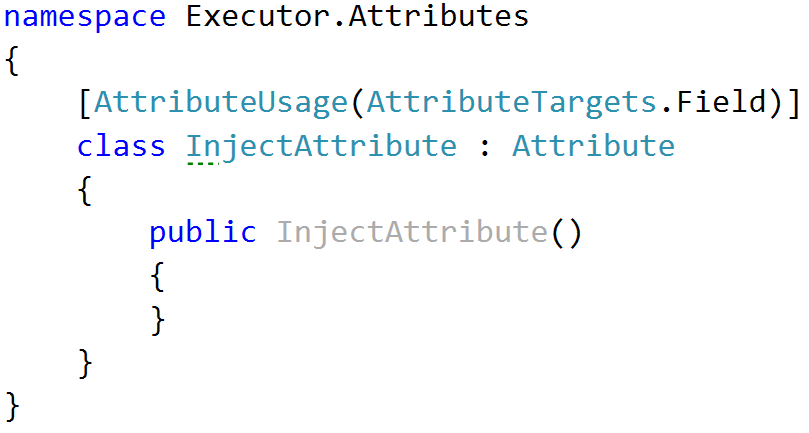
Finally, since we will be using this attribute for classes you might want to tell the compiler just that.



## Making the Inject Attribute

This attribute will point us which field we will set using the reflection. Now we are passing all the “managers/comparers/repos” in the constructor of the chosen command, but some of the users wishes have changed and since we are “[Agile](https://bg.wikipedia.org/wiki/%D0%93%D1%8A%D0%B2%D0%BA%D0%B0%D0%B2%D0%B0_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F)”, we will follow the holy user’s wishes and modify the desired functionality. Also note that passing all utility classes to all commands works for now but if we add more utility classes like say a PeshoManager we will need to add it to all the constructors. Such code is not easy to maintain and extend so we need to fix it. It also causes some commands to know about stuff they don't need nor use.

But first let's create the Inject attribute class.

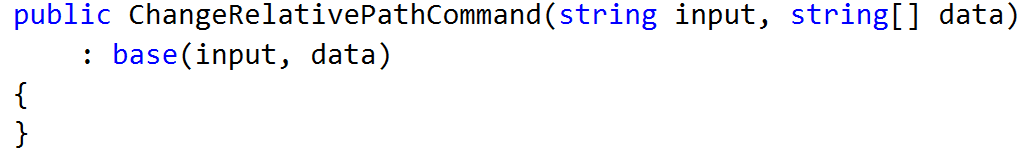


As you can see, we do not have any fields to set in the current attribute. That’s because we want to use it only as a notification that the field below it needs to be injected with a value from the command interpreter. Probably you notice that we’ve put a usage for the field only, and that is absolutely **on purpose**.

### Changing the Commands to Fit the New Approach for Instantiation

Here are the three steps you need to do for **all the commands** in the application.

##### Delete everything from the constructor except input and data.

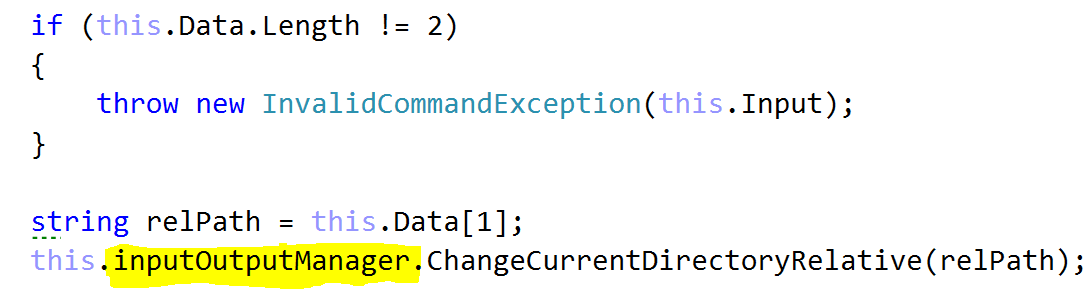


##### Apply alias over the class by passing to the constructor of the attribute the string that is the actual command that you read from the console.



##### Leave only the fields that are used in the current scope of the class and put an inject attribute above them.

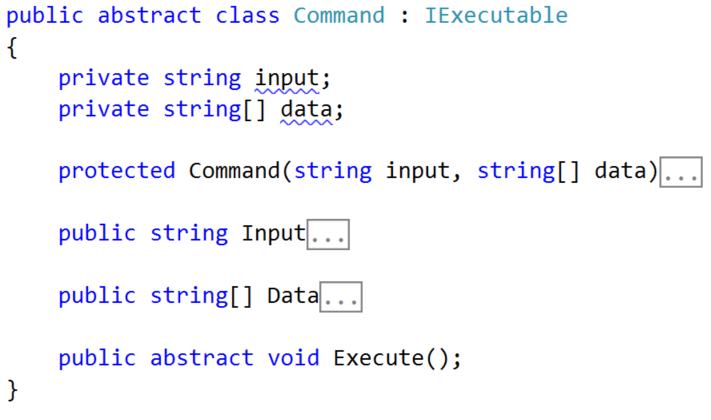


And finally, we are using this field in order to execute the current command:

## Modify Abstract Command

Since you are not going to use the fields except the input and the data[], you can remove the fields, constructor parameters and properties of all the other members.

Here is how your class should now look:

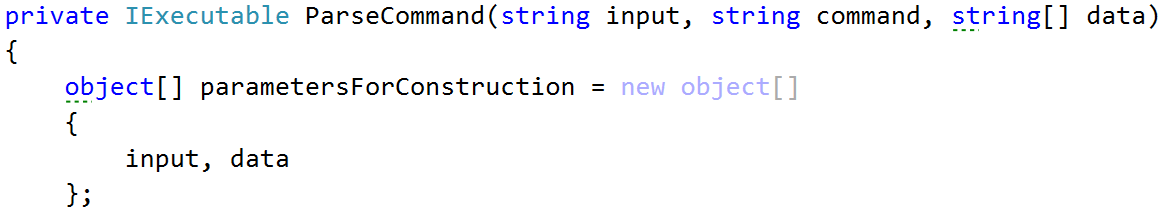


## Modifying the Command Interpreter:

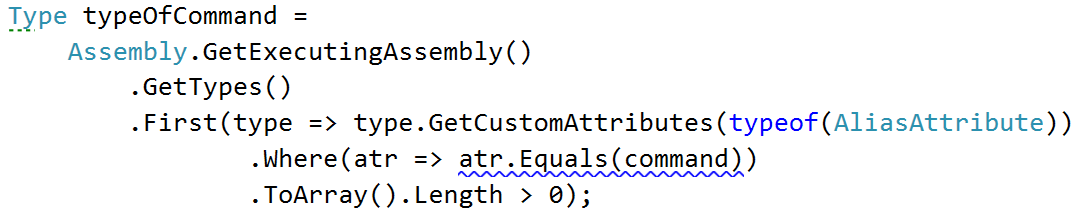
Now it’s time to make some changes in the command interpreter, to start creating the desired command using reflection and not the switch-case.

First thing you can do is delete the whole switch.

Now let’s start preparing for the creation of the objects. First thing you need to make in the ParseCommand method is create an object array filling it with the input and the data:



Now we need to get the type of the command that we want to create. Here is the time to profit from the alias attribute. We want to take all types from the current assembly -> take the first one that has an Alias attribute with a name equal to the wanted command. Here is how this should look:



Another thing we will need is the type of the command interpreter, but it could be easily taken.

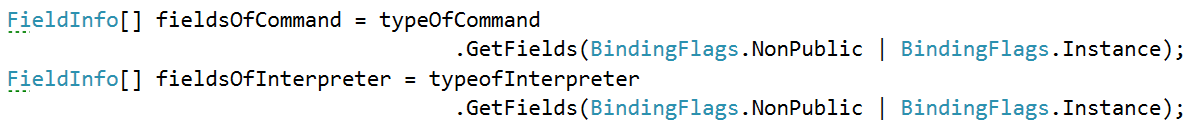


Now it’s time to create the desired command and we are going to do this by using the activator, and passing it the type of command and the parameters for the construction:

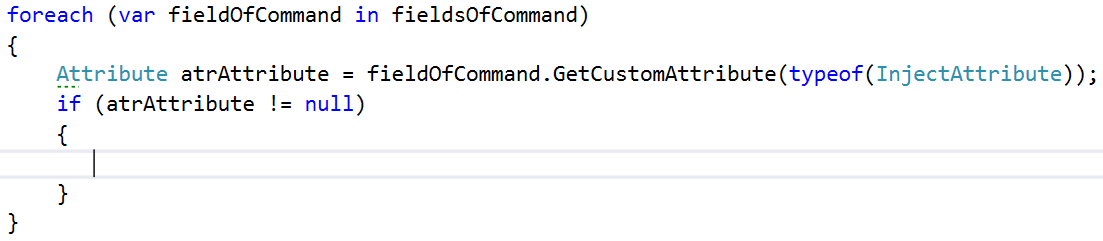


Now that we’ve created the wanted command, we might want to inject the needed field in the command, by passing it the instance from the command interpreter. This is why we will need to get the fields of both of the types of classes.

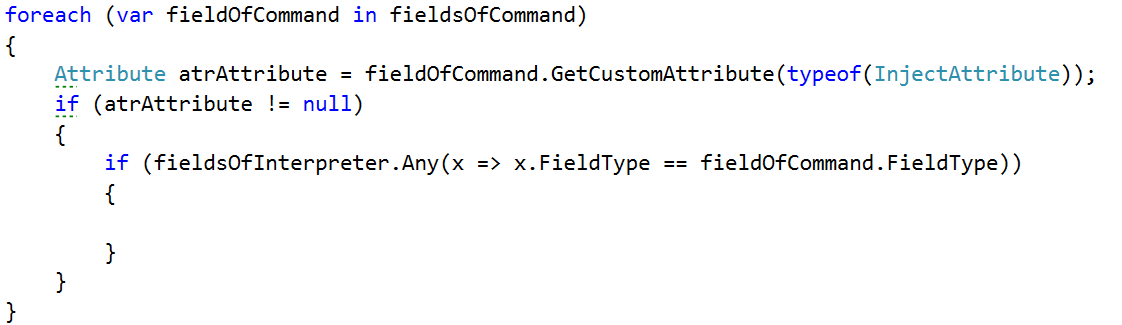
Here is how we do this:



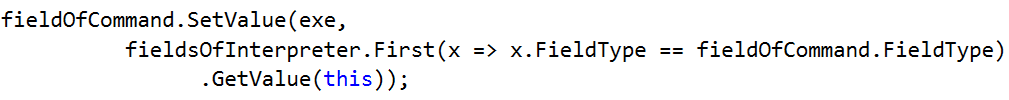
Now we would like to iterate through the fields of the command. Inside, take the custom attribute of type **InjectAttribute** and check whether it is not null:



Next thing we need to check is whether there are any fields in the array of fields of the interpreter, that have the same type as the current one from the command:



Finally, in the inner most if, we should set the value of the current field, by passing the exe(command) and for a value, you can get it from the fields of the interpreter, by taking the first that has the same type and taking its value from the current object (this).



Finally, after the **foreach**, we should return the current created command (exe).

Now we are done with all the refactoring and if you start the program, everything should be working as before. However, from now on if we need to implement a new command, all we have to do is inherit the abstract command class and set the corresponding attributes, but we do not need to change or add anything to the command interpreter. Not to mention this code is way cooler and shorter than the switch. However, you should know that now we've reduced the performance, but this is a tradeoff we can afford.

Here is a cool article which you might want to check out:

<http://www.codeproject.com/Articles/615139/An-Absolute-Beginners-Tutorial-on-Dependency-Inver>

# Unit Testing

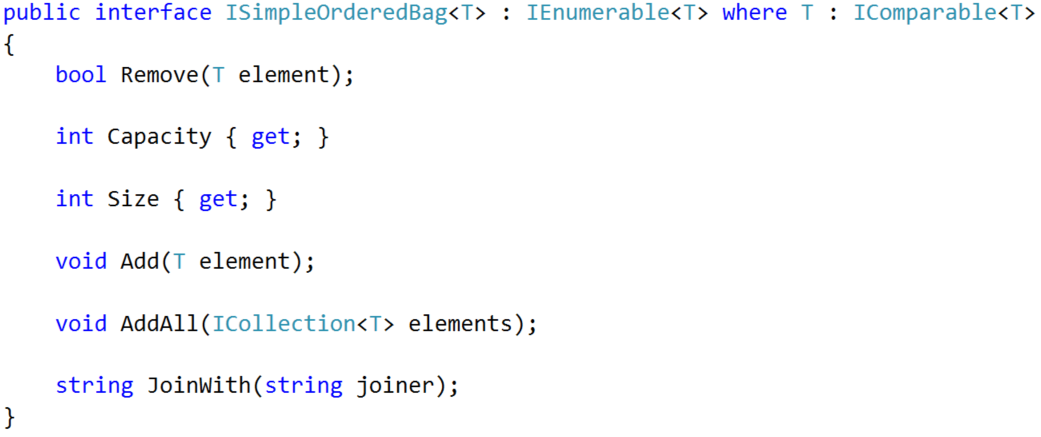
## Introduction

Next, we are going to extend the sorted data structure a little bit and then test if the whole functionality works correctly. If we work through the interface, in a later stage of our project, we may change the concrete implementation of the data structure, but the behavior will stay the same, so the only thing we will need to modify is the actual instantiation in the unit test class.

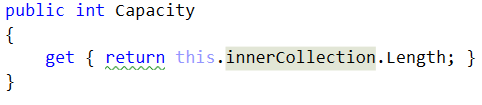
## Modify the ISimpleOrderedBag and SimpleSortedList

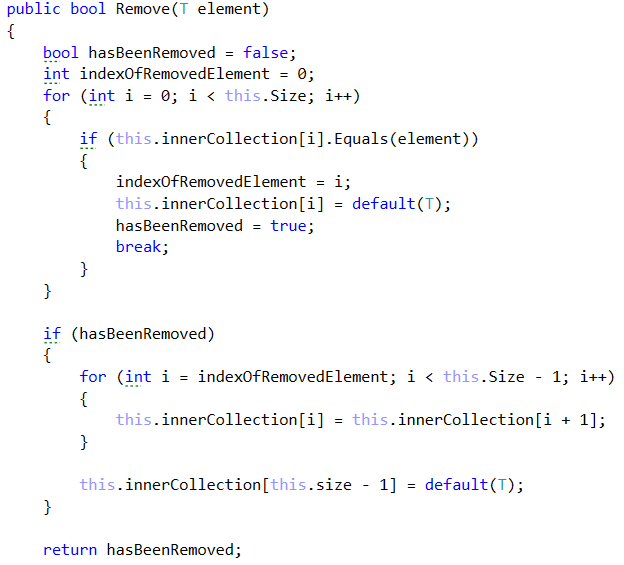
There are **two** things that we will add in the interface. The first one is a **Remove** method that receives a **<T> element** and returns **true** or **false** - whether it has been removed or not.

The second thing - a property for the **Capacity** of the data structure. Here is how the interface should look like after the modification:



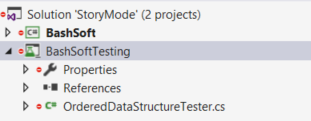
Now implement the missing functionality in the SimpleSortedList. Here is how it should look:





## Writing Unit Tests

Start by creating a new Unit testing project in the current Solution. You can call it BashSoftTesting and rename the class inside it to OrderedDataStructureTester. Here is how your structure should look by now:

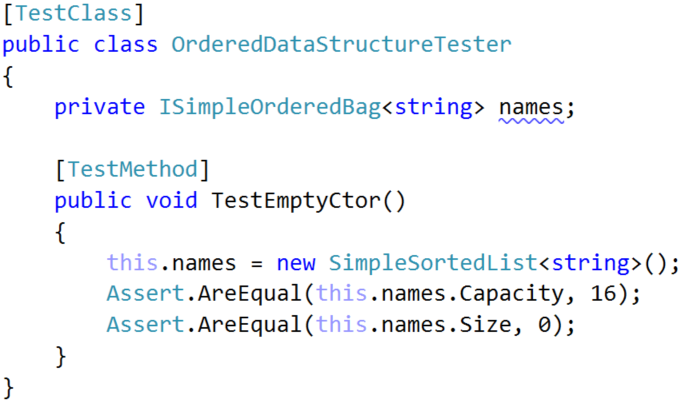


Now it’s time to add a reference to the BashSoft in the current testing project.

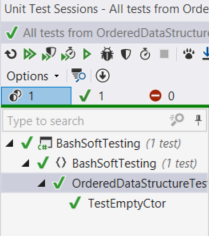
**Right click References -> Add reference -> Select Projects -> Select BashSoft -> Click OK**

Make the simple sorted list public, so that you can see it in the testing app.

Let’s first start by testing the constructors of the class. Name the first test method **TestEmptyCtor** and in it we will make a new instance of the **SimpleSortedList** and check the capacity and size values:

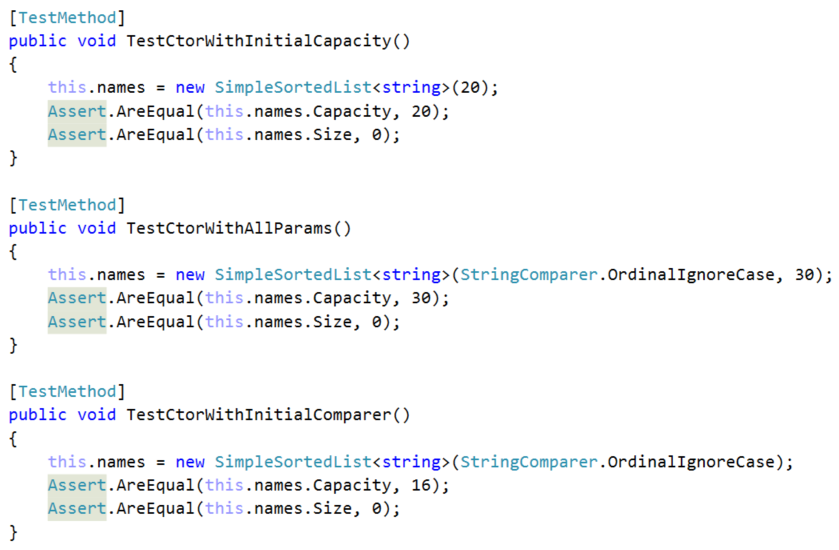


If you run the unit test everything should be according to plan and the result should be such:

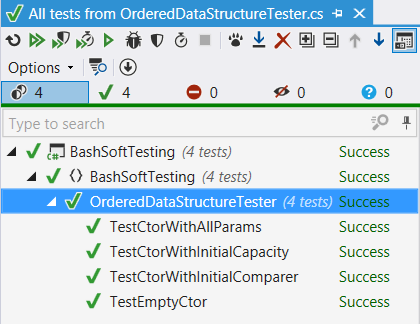


Now let’s make methods that test the other 3 constructors:

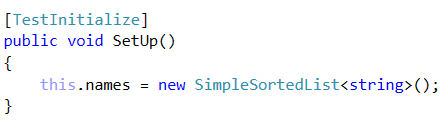
Here are their checks:



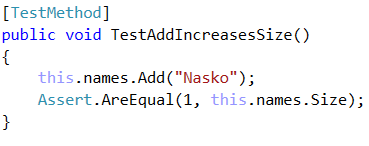
Resulting in:



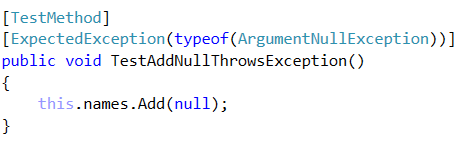
Now that we are done with testing the constructors, it’s time to make a method that initializes an empty collection, to test the rest of the object behavior and it is initialized for each test:



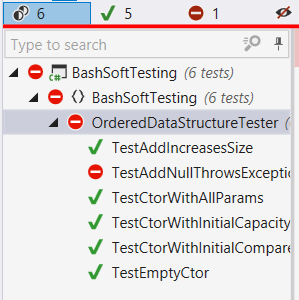
Since we are done with that, let’s check that the add method works as expected and increases the size (count) of the collection:



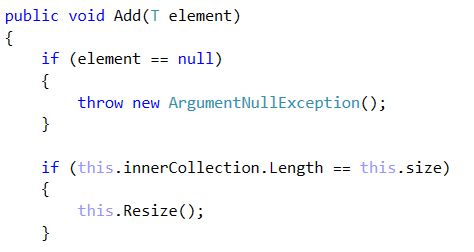
Now we should start thinking like QAs and think what should happen if someone adds **null**. We think it makes sense to throw a new argument null exception, because that does not match the conditions for what values can be held in our data structure:



Now let’s run the unit tests and see the results:



If we look at the implementation of the add method we will see, that we do not have any check for null and so we do not throw argument null exception, anywhere. Let’s implement the functionality now:



If your **Add** method begins like the picture above, everything is great.

Now we’ve shown you how to make unit tests. Try doing unit tests for the following functionality on your own:

**TestAddUnsortedDataIsHeldSorted** – adds three strings (“Rosen”, “Georgi”, “Balkan”) and checks by **iterating** the collection if it holds them in a sorted order: **"Balkan"**, **"Georgi"**, **"Rosen"**

**TestAddingMoreThanInitialCapacity** – adds 17 elements and checks whether the Size is 17 and Capacity is NOT 16.

**TestAddingAllFromCollectionIncreasesSize** – adds 2 elements to a List<string> and adds this list to the names using AddAll. Then check whether the size of the collection equals 2.

**TestAddingAllFromNullThrowsException** – adds a null value to the AddAll and expects it to throw an ArgumentNullException. Since this functionality was not considered in the implementation of the DS, we should now implement it.

**TestAddAllKeepsSorted** – adds a collection with a few elements and after that check if the elements in the SimpleSortedList are sorted.

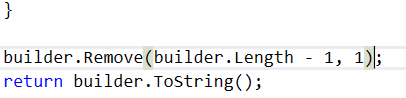
**TestRemoveValidElementDecreasesSize** – adds an element, and removes the same element and then checks if the size has decreased. As you can see, this test should not pass, because we are not decreasing the size anywhere. Fix that and everything should be fine.

**TestRemoveValidElementRemovesSelectedOne** – adds two elements (ivan, nasko) and then removes “ivan” from the collection and asserts it is not there.

**TestRemovingNullThrowsException** – tries to remove null from the collection, which should result in an exception being thrown. Since we haven’t thought of that earlier, it’s now time to implement it.

**TestJoinWithNull** – adds a few elements and tries to join them with null joiner. This should throw an argument null exception; however, the current implementation does not do so. Fix the implementation and the unit test should pass.

**TestJoinWorksFine** – adds a few elements and tries to join them with “, ”. This should not result in a passing test. Let’s see why:

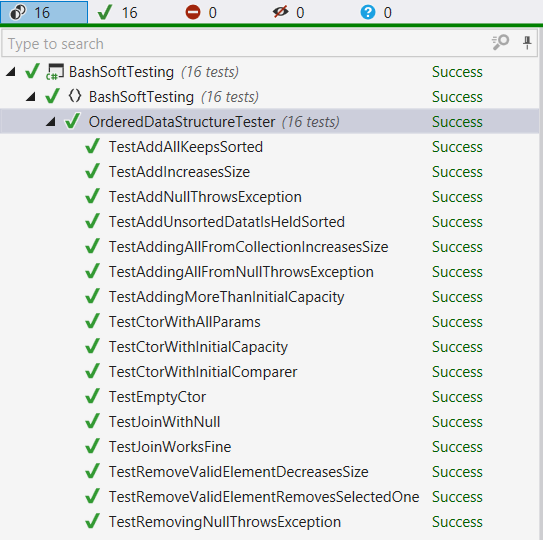


As we can see, we’ve assumed that joiner will be with length 1 which is not a valid case. Let’s fix it.



These are only the common unit tests. You may want to make some more using the other constructors, to check if there are any problems with the comparators.

Final result:



This is your last lab for the current course and for the whole C# Fundamentals Module. Hope you’ve found it interesting and helpful. You can add more functionality on your own, or extend and refactor the project even further. You can even upload it to your GitHub profile as it could be a part of your portfolio.