# Lab: Functional Programming

In the current bashsoft piece we are going to **make** **some** **filters** and **implement** some **sort** **algorithm** so we can **see** **how** **functional** **programming** **can** **be** **helpful** here. The **filters** and **sort** **types** are **described** **in** the **piece** **for** the **strings**, however let’s revise them again. We said that we are going to make a filter for a given course in order to **extract** **some**/**all** **poor**/**average**/**excellent** **students** and print them on the current output in the output writer. After that we are going to **sort** the **wanted** **data** **by** **given** **criteria** (**ascending**/**descending**) and again take some or all the students from the query.

Let’s first stat by **making** two **new** **public** **static** **classes** called **RepositoryFilters** and **RepositorySorters**.

# Part I: Filtered Students Query

## Implement Filters

The **first** **method** we need **in** the **filters** **repository** class **is** the **public** **API** we are going to give to the world to use. It’s going to be a **public** **static** **void** **method** called **FilterAndTake**. Since we are going to **filter** **students** **from** a given **course**, we need to **receive** the **dictionary** that corresponds to the **students** **with** their **scores** **from** the **wanted** **course**. Another thing the **method** **has** **to** **receive** is which **filter** **to** **use**. Since we are **reading** **strings**, **from** the **InputReader** , we can **pass** **them** **to** this **method** **as** a **string** **and** here **in** the **RepositoryFilters** class we can now **decide** **which** **filter** **to** **apply** **to** the **data**. The **final** **parameter** that the method needs to receive is the number of **students** to **take**. Since **we** **can** **parse** it **in** the **checking** **of** the **data**, that we do in **the** **command** **interpreter**, the data type of the variable can be an integer.

By now the method signature of **FilterAndTake** should look like this:

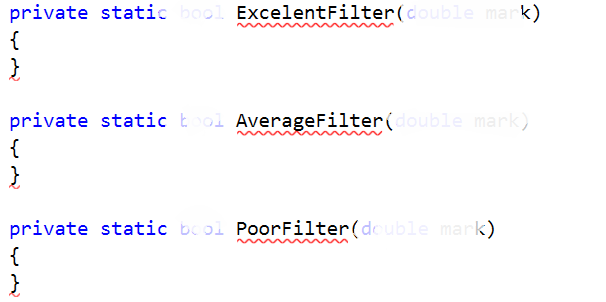


Since the **public** **method** **receives** the **wanted** **filter** **as** a **string**, it’s his job to decide how to **decide** **which** **method** **for** **filtering** **to** **use**. Since it’s going to be another method that is actually going to do the filtering, let’s **make** it too. It can be called **FilterAndTakeagain**, however it’s going to be **private** **static** **void** and **with** a **change** **in** the **parameters**. The **new** **FilterAndTakeis** going to **receive** the **same** **wantedData**, and the **same** **variable** **studentsToTake**, **but** the **wantedFilter** is **now** a **Predicate** (method that **returns** a **bool**) that **receives** a **double**. The description above should look like this:

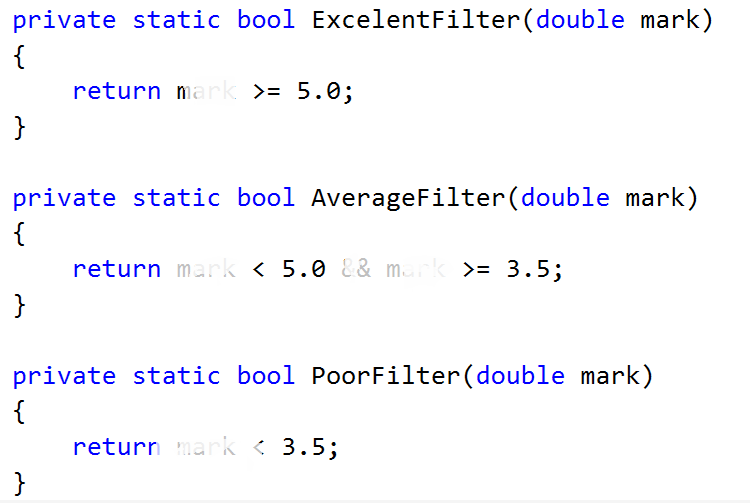


As you can see, the things are a bit coupled, but in the same time quite detached, because we can easily extend it. Now the things we need to **implement**, in order to figure out **how** the **FilterAndTake method** is going to **work**, are the methods that we are going to pass as **predicates** that are actually **going** **to** **be** **our** **filters**.

There are going to be **three** **methods** of such type since we wanted the three type of students (**excellent**/**poor**/**average**). This is how the **initialization** **of** the **methods** **should** **look** **like**:

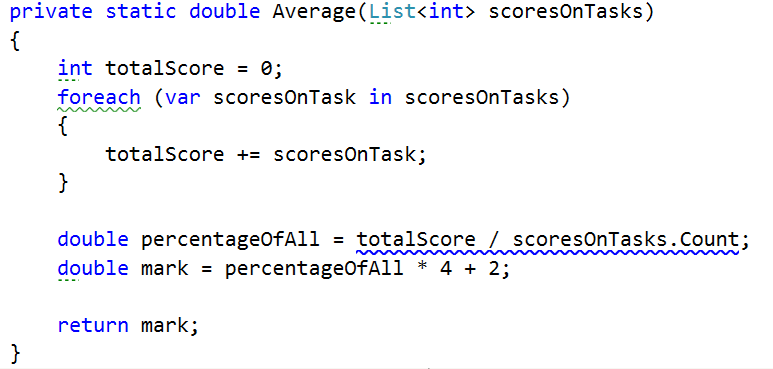


The three of them are underlined with red, because the **method** **has** a **return** **type** and by now we are not returning anything. Well maybe it’s time to start writing, in order to change that. Since we are receiving a mark as a parameter, it’s in the range from 2 to 6, so it’s up to use, which mark is excellent, which is average and which is poor. We suggest that you return true for an excellent mark if it is more than or equal to 5.00, return true for an average mark if It is more than or equal to 3.50 and less than 5.00 and finally return true for a poor mark if it is less than 3.50. If you’ve followed the instructions, by now you should have something like this:



## Implement Average Mark

There is **one** **more** helper **method** we need **to** **make** in order to do the job. It’s called **Average** and **receives** a **list** of **scores**. It should be **private** and **static** and since it’s going to **return** the **average** **mark**, we leave it up to you to decide what’s the good return type of the method.

After we’ve implemented the signature of the method it’s time for the implementation. First we’ll need a **variable** to **hold** the **total** **score** **of** **all** **the** **tasks**. Next thing we should do is **iterate** **through** the **list** and **add** **each** **value** **to** the **total** **score**. Finally **after** the **foreach** we should **take** the **percentage** **of** the **total** **possible** **result** and since we have 5 tasks on exam, we **divide** by the number of tasks multiplied by 100. Now we have the percentage of all the possible points and we can easily calculate our mark by the formula **mark = percentageOfAll \* 4 + 2.** If you’ve done everything correct, by now your implementation of the method for the calculation of the average mark should be something pretty close to this piece of code:   
  


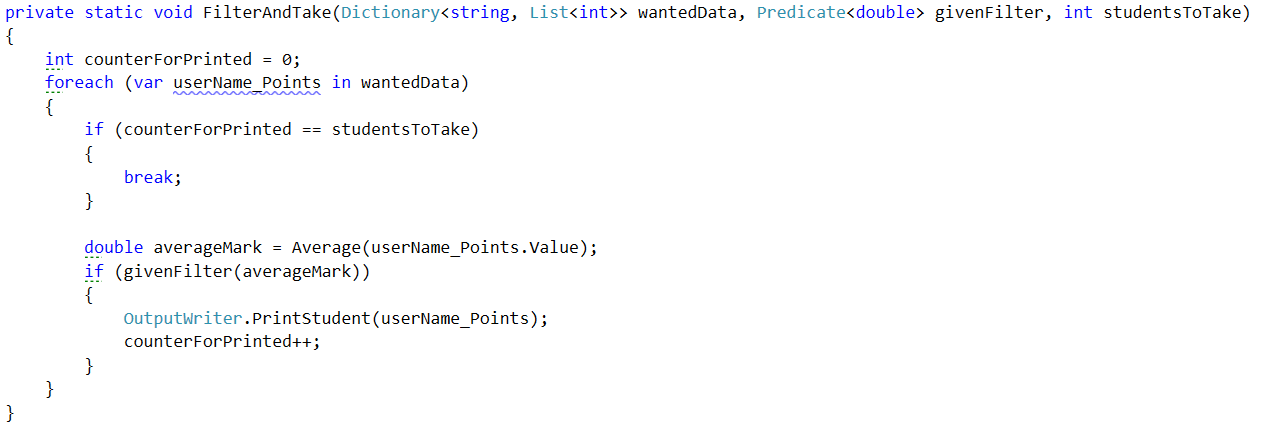
## Implement private FilterAndTake method

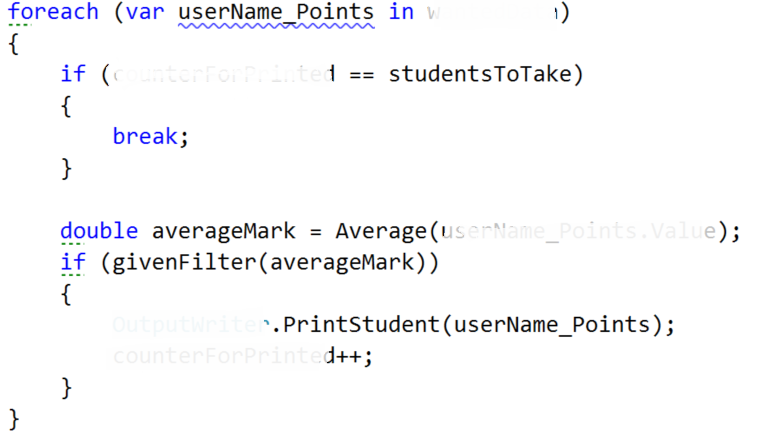
Now that we are done with the helper methods, I suggest it’s time to **move** **to** the **actual** place where the **filtering** is done and that is the **private** **FilterAndTake** **method**.

First thing we are going to need in the method is a **variable** to **hold** the **number** **of** **printed** **students** that **match** the **given** **filter** and therefor are **printed** **on** the **output** **writer**.

The next thing we do is **iterate through all** the **entries in** the **dictionary** called **wanted data** and **for each student**, we **calculate** it’s **average mark** using the method we implemented above, as we pass to it, the value of the key-value pair that give us the current iteration of the dictionary.

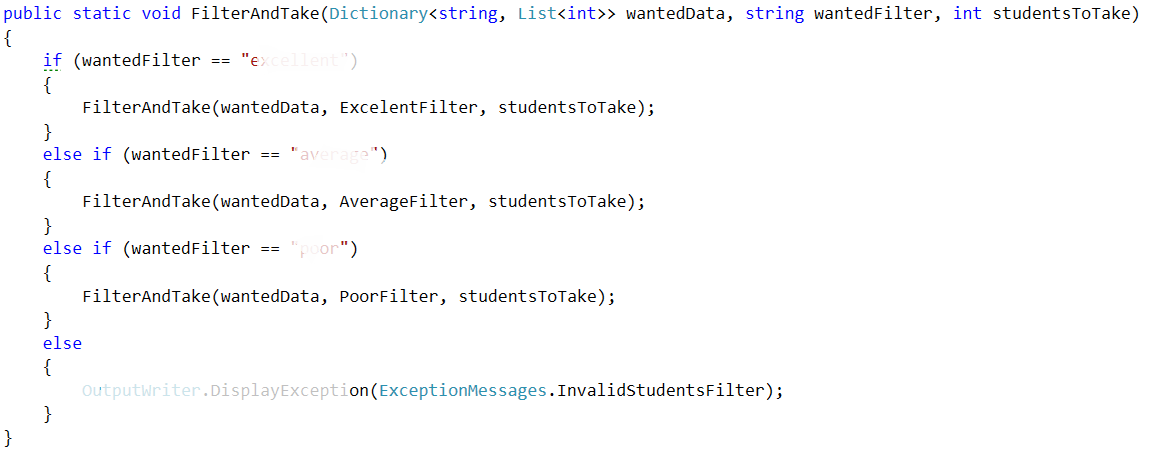
Finally we **check** **if** the **average** **mark**, **passed** **to** the given **filter**, **returns** **true**. And if that is so. We print the student on the output writer using print student method and increment the counter for printed students. By now the implementation of the method we are talking about should look like this:

   
  
There is just one little problem here and it is the fact that we don’t have the implementation for taking only the wanted quantity of students matching the filter and not all of them. So now we have to **add** a **block** of code **that** **breaks** the **loop** **if** we’ve **printed** **enough** **students** and it should be **first** **in** the **foreach** **loop**. By doing this, our foreach loop now look like the following:



## Implement Public FilterAndTake Method

Now we are only **left** **with** the **public** **FilterAndTake** **method** which is actually going to be the method that the other world is going to use, in order to filter the given data. It’s implementation is very straightforward. All we do is **check** **if** the **wanted** **filter** **corresponds** **to** **one** of the **possible** **categories** (**excellent**/**average**/**poor**) and if it is one of them, we **call** the **private** **FilterAndTake** **method**, **with** an **input** **parameter** for the **Predicate**, the **function** **that** **corresponds** **to** the **category**. **If** the **given** **word** **does** **not** **match** **any** of the **categories**, we **display** an **exception** called **InvalidStudentFilter**, which we **first** need to **add** **to** the **ExceptionMessages** **with** a **message** of: “The given filter is not one of the following: excellent/average/poor”. So our implementation of the public method should look likes this:



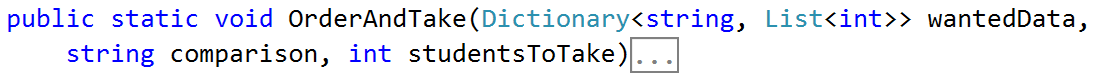
Finally we should be ready with the filtering repositories class and it’s time to move on to the sorting repos’ class.

# Part II: Sorted Students Query

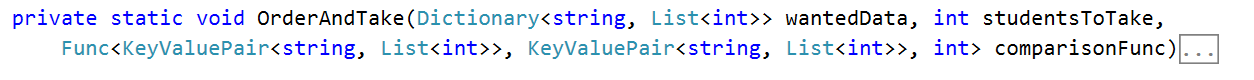
## Implement Sorters

The **first** **method** we need **in** the **sorter** **repository** class **is** the **public** **API** we are going to give to the world to use. It’s going to be a **public** **static** **void** **method** called **OrderAndTake**. Since we are going to **sort** **students** **from** a given **course**, we need to **receive** the **dictionary** that corresponds to the **students** **with** their **scores** **from** the **wanted** **course**. Another thing the **method** **has** **to** **receive** is which **sorter** **to** **use**. Since we are **reading** **strings**, **from** the **InputReader** , we can **pass** **them** **to** this **method** **as** a **string** **and** here **in** the RepositorySorters class we can now **decide** **which** **sorter** **to** **apply** **to** the **data**. The **final** **parameter** that the method needs to receive is the number of **students** to **take**. Since **we** **can** **parse** it **in** the **checking** **of** the **data**, that we do in **the** **command** **interpreter**, the data type of the variable can be an integer.

By now the method signature of **OrderAndTake** should look like this:

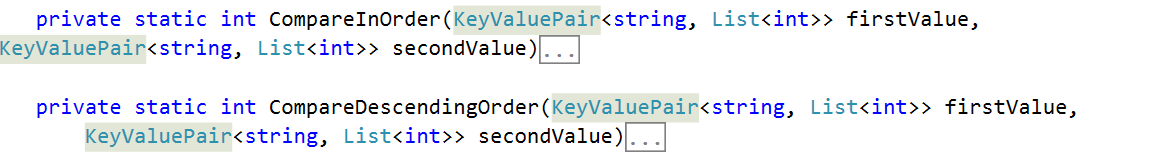


Since the **public** **method** **receives** the **wanted** **sorter** **as** a **string**, it’s his job to decide how to **decide** **which** **method** **for** **sorting** **to** **use**. Since it’s going to be another method that is actually going to do the sorting, let’s **make** it too. It can be called **OrderAndTake again**, however it’s going to be **private** **static** **void** and **with** a **change** **in** the **parameters**. The **new** **OrderAndTake is** going to **receive** the **same** **wantedData**, and the **same** **variable** **studentsToTake**, **but** the **comparison type (sorter)** is **now** a **Func** that **receives** a **two key value pairs (students with marks) and returns an int which is the result of the comparison**. The description above should look like this:

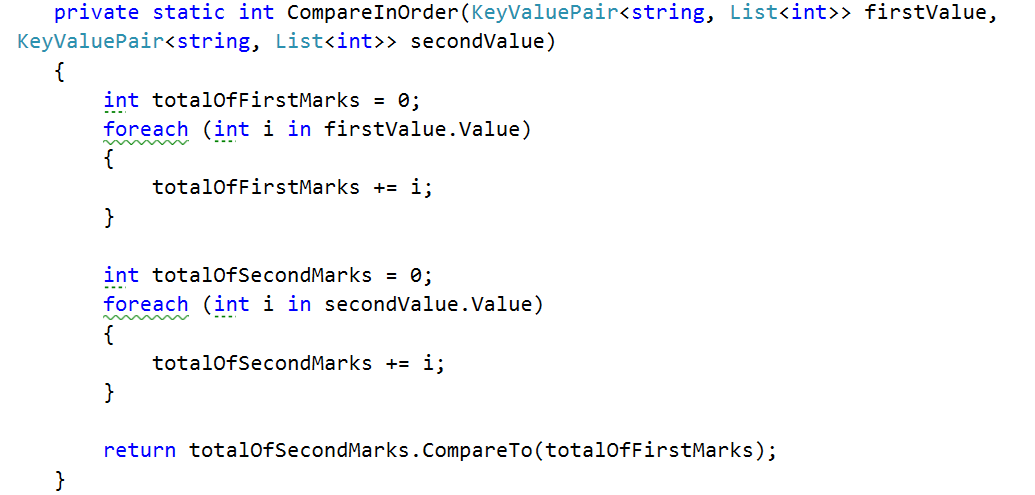


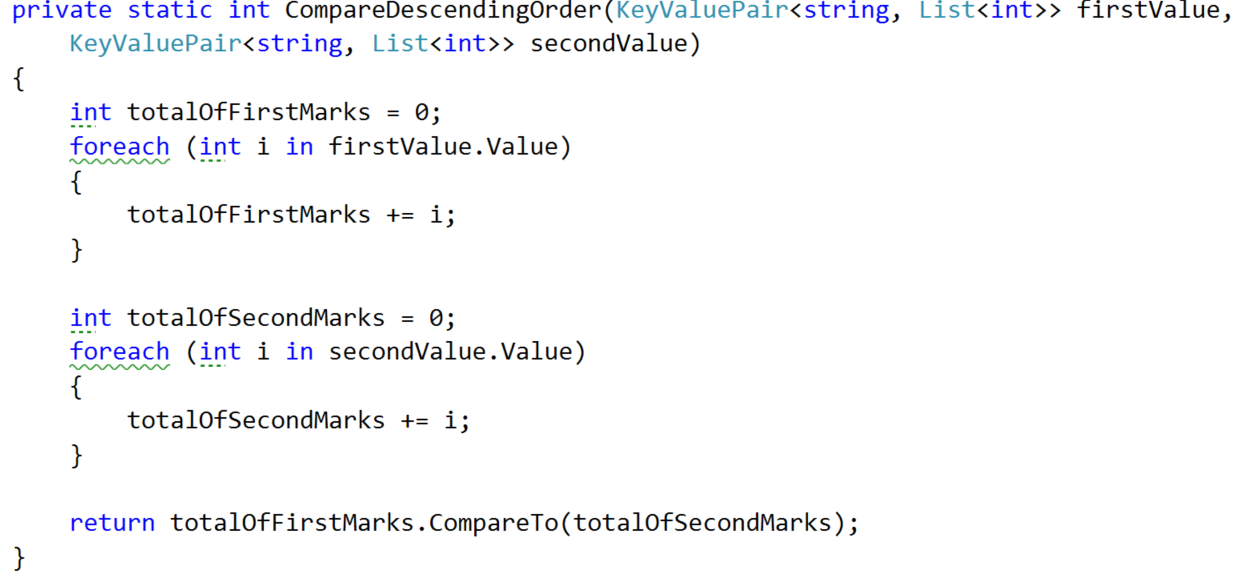
Now the things we need to **implement**, in order to figure out **how** the **OrderAndTake method** is going to **work**, are the methods that we are going to pass as **functions** that are actually **going** **to** **be** **our** **comparison types**.

There are going to be **two** **methods** of such type since we wanted the **two** **type** of **comparisons** (**ascending**/**descending**) . This is how the **initialization** **of** the **methods** **should** **look** **like**:



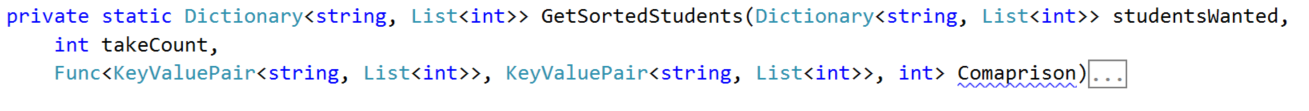
Since we are receiving a two students, we have to compare them in by a given way and return 1, 0 or -1 depending on which one is greater/smaller. To compare them in order, we compare the sum of the scores of all tasks and return the result of the second compared to the first. For the other one we do the same thing, but we compare them in the opposite way. The way the implementation should look is like the following:





## Implement Private OrderAndTake Method

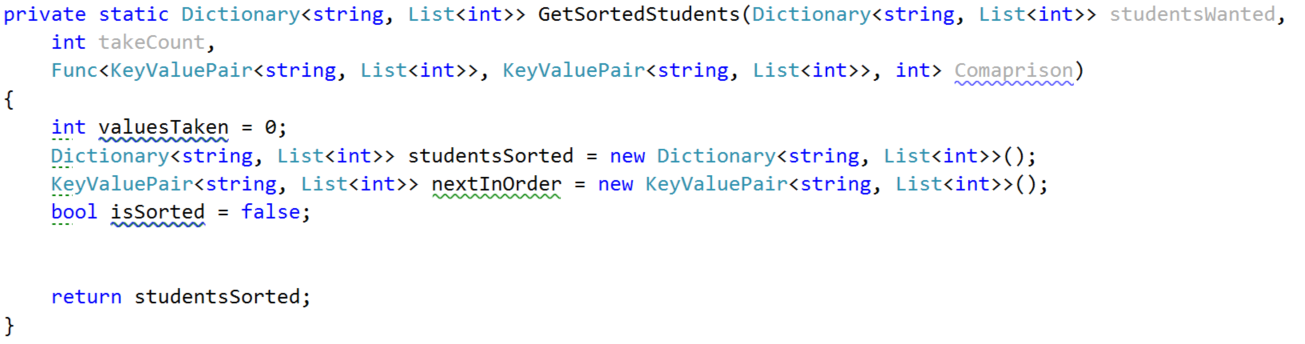
Now that we are done with the helper methods, I suggest it’s time to **move** **to** the **actual** place where the **sorting** is printed and that is the **private** **OrderAndTake** **method**. We simply make a new dictionary of string and list of ints called studentsSorted that is equal to the GetSortedStudents method, which we haven’t talked about, but it’s signature look like this:

After we’ve gotten the sorted student in a dictionary, we simply print it on the output writer using the print student method.

## Implement Private GetSortedStudents

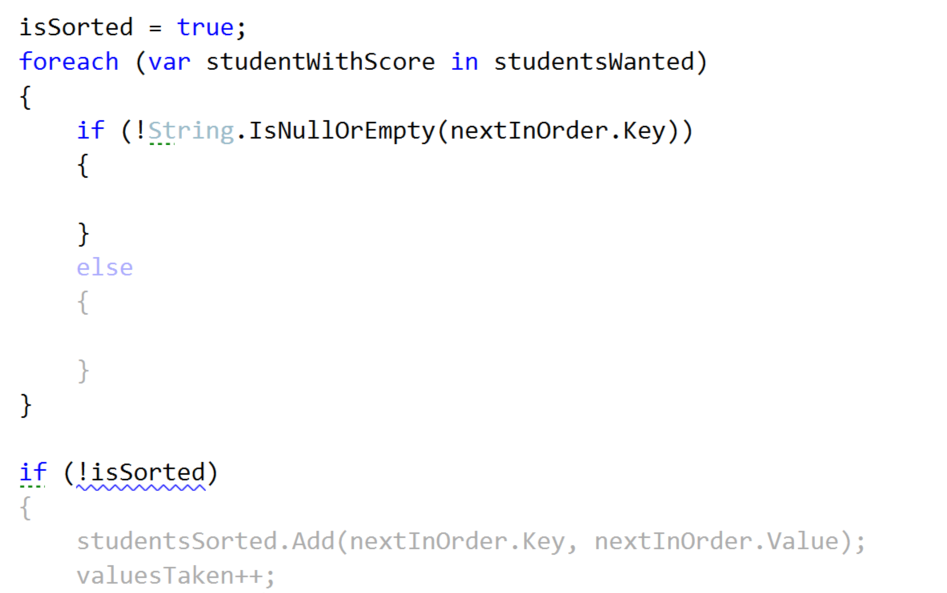
The first thing we do in this method is to **make** a **variable** **for** the **number** of **values** **taken** and **set** it **to** **zero**, because as with the filters, we do not want to take all the students from the sorting, but only the requested amount. Next thing in order is to **make** a **new** **dictionary** **for** the **sorted** **students**. Finally we should make **one** **more** **helper** **variable** to **hold** the **next** **value** that is in the requested order.

Now it’s time to **implement** an easily **understandable** **sorting** **algorithm** and for that reason we’ve chosen **bubble** **sort**. For the job you need to **add** **one** **final** **helper** **variable** of **Boolean** **type** that is **called** **isSorted**, because you should all know that the **bubble** **sort** **needs** **such** a **variable** **for** the **condition** of the **loop**. By now your method should look like this:

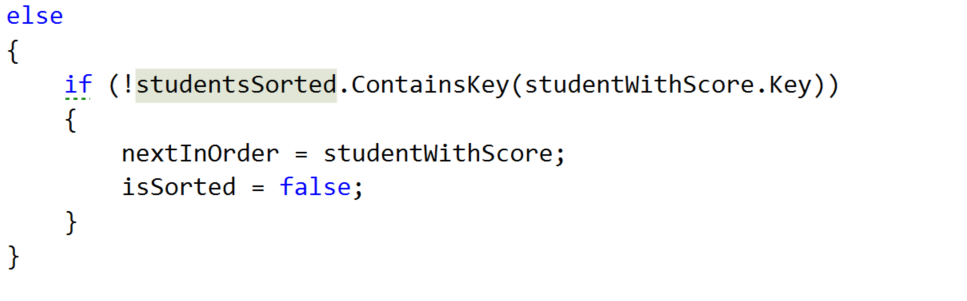


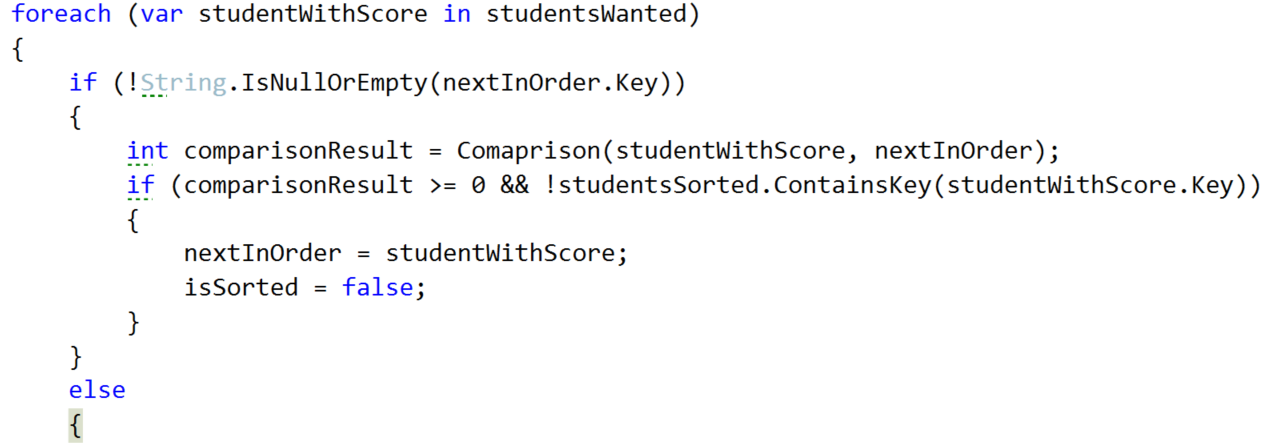
From now on we **place** the **while** **loop** **of** the **bubble** **sort** and on each iteration we first **set** the **is** **sorted** **to** **true**. **At** the **end** **of** the **loop** we **check** **if** the **isSorted** **bool** is **not** **true** and **if** **so**, **add** the **data** **from** the **nextInOrder** **to** the **studentsSorted**. After that **increment** the **valuesTaken** and **finally** **set** the **nextInOrder** **to** a **new** **KeyValuePair**:



**Next** **thing** in the queue with the things **to** **implement** **is** the **inner** **loop** that **finds** the **current** **min**/**max** **element**. For that reason we **make** a **new** **foreach** **over** the **studentsWanted**. Since we have two possibilities for the **keyvalue** **pair** **nextInOrder**. It’s value **is** **either** **set** **or** **not** **set** **so** we have a **null** **key** and a **null** **value**. So we can check **if** the **nextInOrder’s** **key** is **not** **null** or **empty** and **do** **one** **thing** and **if** **not** **do** **another** thing:  
  
 

Let’s first **implement** the **else** **clause**. In it we have to **check** **whether** the **new** **sorted** **dictionary** **does** **NOT** **contain** as a **key** the **current studentWithScore’s key**. **If** **so**, we **set** the **nextInOrder** **to** the **studentWithScore** and **set** the **isSorted** to **false**.

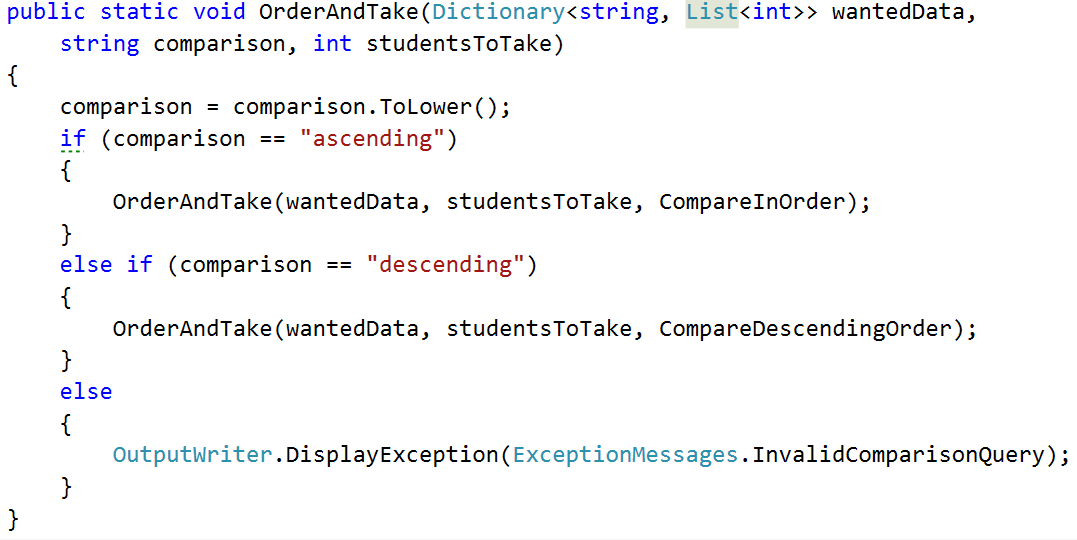


Waiting up next is the if clause. We **take** the **int** that our **Comparison** function **returns**, **by** **passing** it the **nextInOrder** **and** the **studentWithScore** **If** the **comparison** **result** is **greater** **than** **or** **equal** **to** **0** and the **dictionary** that we use **for** the **sorted students does NOT contain** the **key of** the **studentsWithScore’s key**, we **set** the **nextInOrder** **to** the **studentWithScore** **and** the **isSorted** to **false**.  
  


Now that we are ready with the get sorted students, we hope that the private OrderAndTake will also work correctly. So one last thing is left in the current class and it is to implement the public OrderAndTake.

## Implement Public OrderAndTake Method

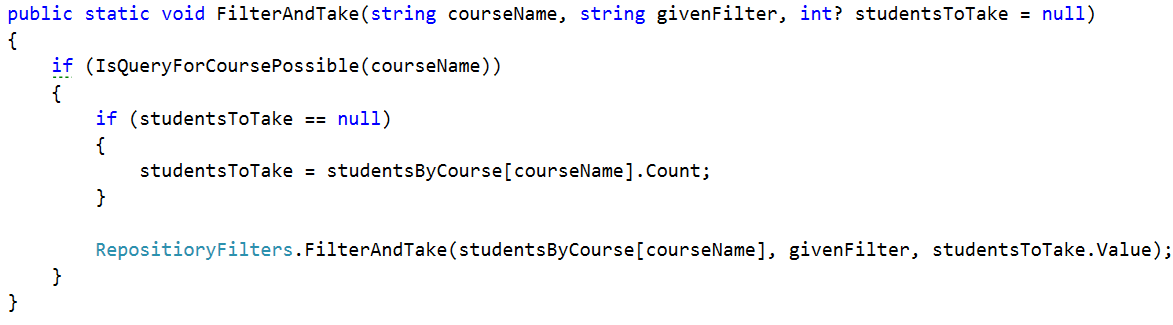
Here our only job is to decide how to **choose** **which** **comparison** **type** **to** **use**. That is why we do pretty much the same thing as in the public FilterAndTake. First we check **if** the **comparisonType** string is **ascending** and if so, **call** the private **OrderAndTake**, **passing** the **in** **order** **comparison** **Func**. **If** **descending** is **chosen**, **call** the same **method** **with** the **descending** **order** **comparison** **Func**. **If** **none** of the comparisons is chosen we **display** a new **Exception** **message**, which we should **first** **add** **to** the **ExceptionMessages** called **InvalidComparisonQuery** with a **message** “The comparison query you want, does not exist in the context of the current program!”



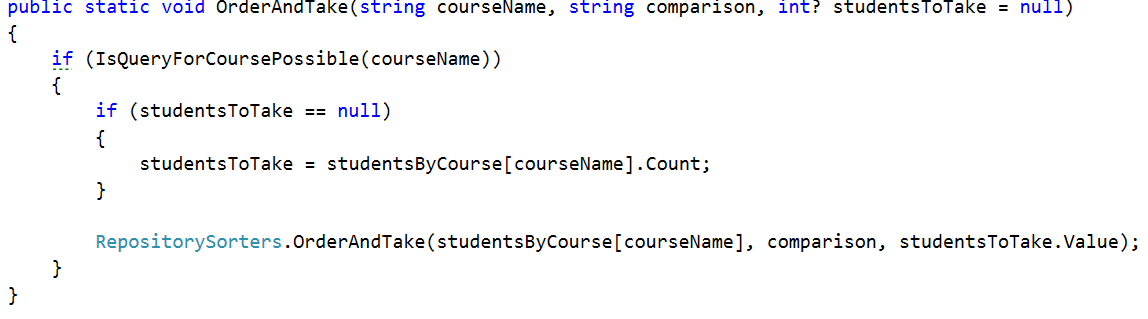
# Student Repository Implementation Part of Filters and Sorters

Since we are going to use the **dictionary** **from** the **StudentsRepository** **class** and it is **private**, we can **easily** **take** **all** that **we** **need** **from** the **StudentsRepository** **by** **using** **it** as a **mediator** **between** the **command** **interpreter** **and** the **filters**/**sorters**. So what we are going to **make** are **two** **methods** in **this** **class**. **One** that **called** **FilterAndTake** **and** **one** **OrderAndTake**. The **filter** **follows** the **following** **signature**:



If you’re wondering why the students to take is **nullable** **with** a **default** **value** of **null** it’s because we want to call the method with giving it the parsed value and if it hasn’t parsed (in the command interpreter – we’ll get there soon) for example if the user has inputted “**all**”, we want to **make** **sure** we **take** the **number** **of** **students** **in** the **current** **course** and that **is** only **possible** one we’re **in** the **StudentRepository** **class**. If you are confused, don’t worry it’s harder to explain that to see it in code.   
  


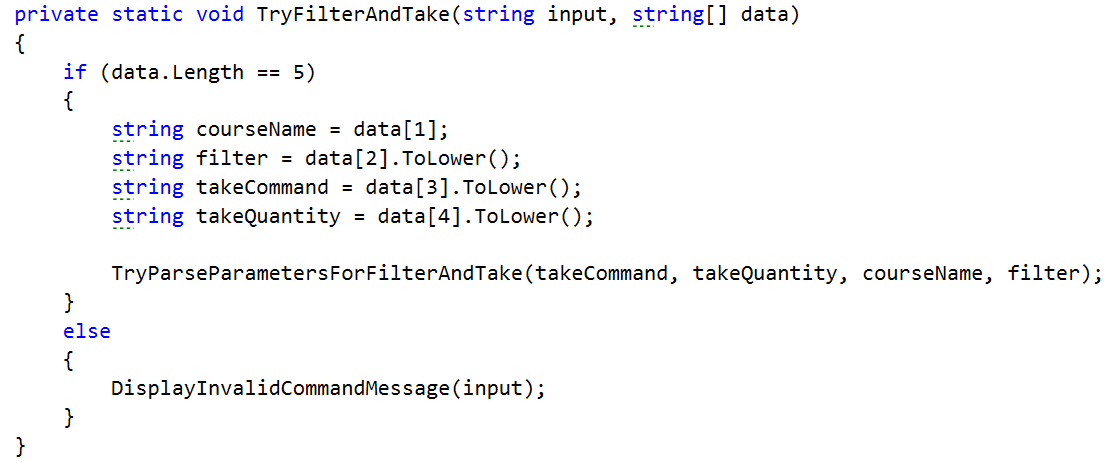
This situation with the OrderAndTake is pretty much the same as you can see :

  
Now that we have these methods we can easily **communicate** **with** the **RepositoeryFilters** **indirectly** **using** the **StudentsRepository**.

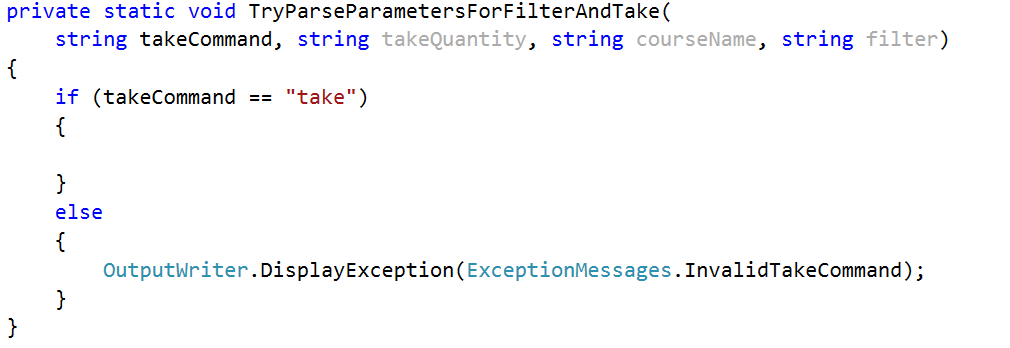
# Part III Command Interpreter Implementation Part of Filters and Sorters.

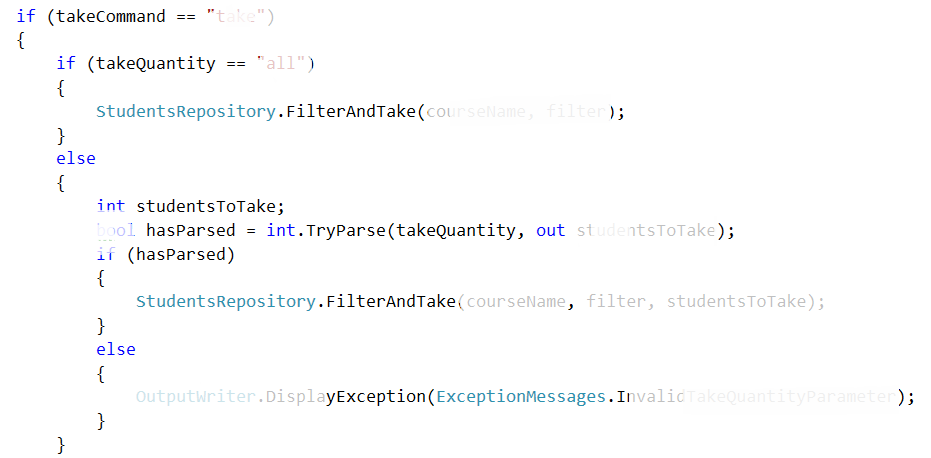
**In** the **command** **interpreter** we should **make** **two** **methods** **called** **TryFilterAndTake** and **TryOrderAndTake** that **take** **input** **parameters**, the **same** **as** **all** the **other** **try** **methods** in this class. After making them we should **call** **them** **in** the **InterpredCommand** **method** **in** the **appropriate** **place**.

## Implement Filtering Data Parsing in Command Interpreter

Let’s first **look at** the **implementation** of the **TryFilterAndTake** method. All we have to do there is **check** **if** the **number** of **input** **parameters** **are** **5** and **if** **not**, **DisplayInvalidCommandMessage**. **If** they **are**, we **take** the **course** **name** which is **at** **index** **1,** the **filter** in **lower** **case** at **index** **2**, the take **command** in **lower** **case** at **index** **3** and finally the take **quantity** in **lower** **case** at **index** **4.** Finally we should **pass** all those **parameters** **to** a **new** **method** **TryParseParametersForFilterAndTake**.   
  


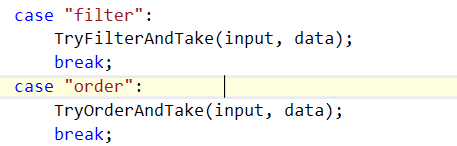
Actually the method we mentioned above does almost all of the validation of the parameters so let’s look at it’s implementation.

First we **check** **if** the **take** **command** **is** actually **equal** **to** the word “**take**” and **if** **not** we **print** an **exception** **message** on the **output** **writer**, which of course we should first **add**, called **InvalidTakeQuantityParameter** with a **message** “The take command expected does not match the format wanted!”  
  


If this is the actual command we now have to **check** **if** the **take** **quantity** **is** “**all**” and **if** **so**, **call** **FilterAndTake** **from** the **StudentsRepository** **without** the **last** **parameter** **for** the **quantity** and therefor it is **null** **by** **default**, because we set it to a nullable int. However **if** that is **not** **the** **case**, we have to **check** **if** it is a **number** that **can** **be** **parsed**. **If** the number **can** **be** **parsed**, we **get** the **result** **from** the **parse** and **call** the **FilterAndTake** but **including** the **last** **parameter**. In the case where the **number** **hasn’t** **been** **parsed** we should **display** an **exception** for **InvalidTakeQuantityParameter**. All of the above should look something like this:  
  


The situation with the **TryParseParametersForOrderAndTake** is the **same** so leave the **implementation** of this **method** **to** **you**.

Now if you’ve done everything and the situation in the switch case in the **InterpredCommand** method is the following :



..everything should be ok and we are **ready** **to** **start** **reading** **from** the **input**.

Next thing to do is read the **dataNew.txt** from where you’ve saved it and **apply** one **sorting** and one **filtering**.

