# Lab: Functional Programming

In the current Bashsoft piece we are going to **add** **some** **filters** and **implement** some **sorting** **algorithms** so that we may **see** **how** **functional** **programming** **could** **be** **helpful** here. The **filters** and **sort** **types** are **described** **in the** String’s part but let’s do a short revision. We said that we are going to add 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 OutputWriter. After that we are going to **sort** the **filtered** **data** **by** **a given** **criteria** (**ascending**/**descending**) and again take some or all the students from the query.

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

# Part I: Filtered Students Query

## Implement Filters

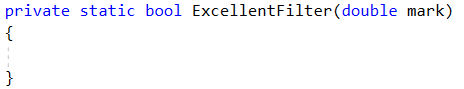
The **first** **method** we need **in** the **filters** **repository** class **is** the **public** **API** we are going to give to the world to use. It’s going to be a **public** **static** **void** **method** called **FilterAndTake**. Since we are going to **filter** **students** **from** a given **course**, we need to **receive** the **dictionary** that corresponds to the **students** **with** their **scores** **from** the **seeked** **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 **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** **parse** it in **the** **command** **interpreter** when we check the input data, the type of the variable will be an integer. By now the method signature of **FilterAndTake** should look like this:

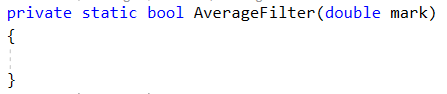


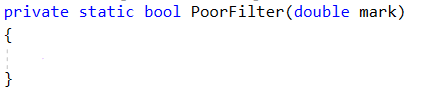
Since the **public** **method** **receives** the **wanted** **filter** **as** a **string**, it’s his job to decide **which** **method** **for** **filtering** **to** **use**. The method which will actually do the filtration can be named **FilterAndTake again**, however it’s going to be **private** **static** **void** and with a **change in the parameters**. The **new** **FilterAndTake is** 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:



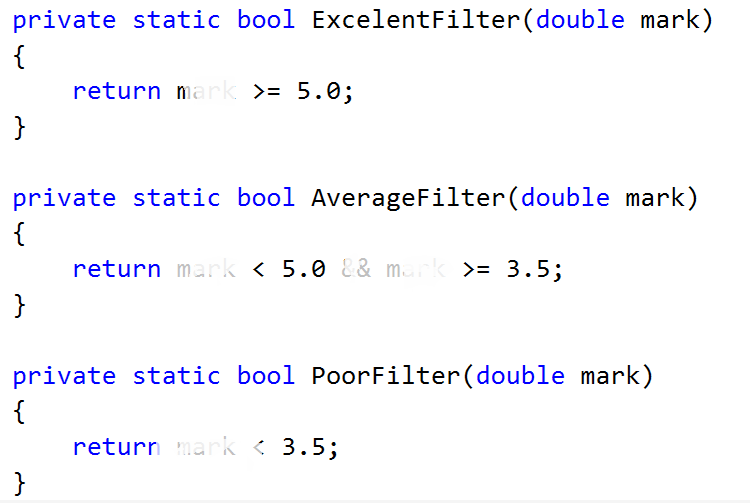
As you can see things are a bit coupled but in the same time quite detached because we can easily extend the methods. Now we need to **implement** the methods that we are going to be passed as **predicates** which are actually **the real** **filters**. We are going to **have three** of them since we have three types of students (**excellent**/**poor**/**average**). This is how the **initialization** **of** the **methods** **should** **look** **like**:





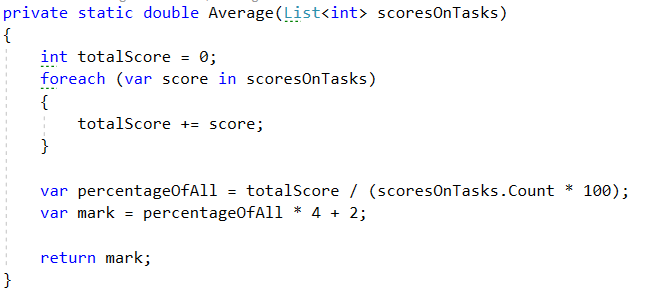


The parameter representing the mark should be in the range from 2 to 6, so it’s up to us to decide which mark corresponds to excellent, average and poor. We suggest that you return true for an excellent mark if it is higher or equal to 5.00, return true for an average mark if it is higher or equal to 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** **implement** 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.

Let’s create this method. First we’ll need a **variable** to **hold** the **total** **score** **of** **all** **the** **tasks**. Then we should **iterate** **through** the **list** and **add** **each** **value** **to** the **total** **score**. Finally **after** the **foreach** we should **take** the **percentage** **of** the **total** **success rate** which can be obtained by **dividing** the average score by the **number of tasks** **multiplied** **by 100** to get percents. Now we have the percentage of total success 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 calculating the average mark should be something pretty close to this piece of code:   
  


## Implement private FilterAndTake method

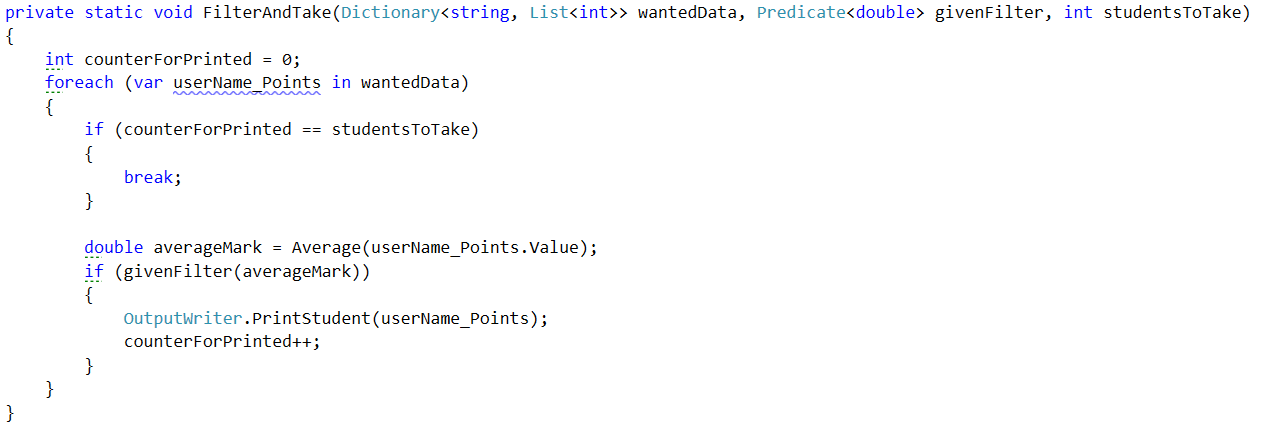
Now that we are done with the helper methods it’s time to **move** **to** the **actual** place where the **filtering** is done - the **private** **FilterAndTake** **method**.

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** in order **not to exceed the limited number** of students we are asked to return.

Next we’ll **iterate through all** the **entries in** the **dictionary** called **wantedData** and **for each student**, we **calculate** it’s **average mark** using the method we implemented just before.

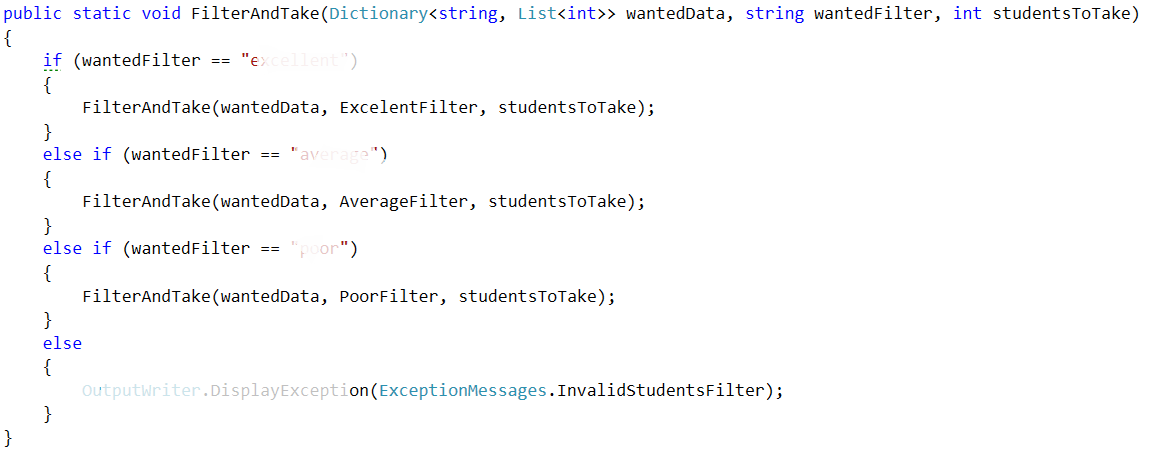
Finally we **check** **if** the **average** **mark**, **passed** **to** the **given** **filter**, **returns** **true**. And if that is so we print the student on the OutputWriter using PrintStudent method and increment the counter for printed students.

There is just one more little problem. We have no condition to stop printing students when the limit is reached. So 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 outer world is going to use in order to filter the given data. It’s implementation is very straightforward. All we do is to **check** **if** the **wanted** **filter** **corresponds** **to** **one** of the **possible** **categories** (**excellent**/**average**/**poor**) and if it is so, 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:



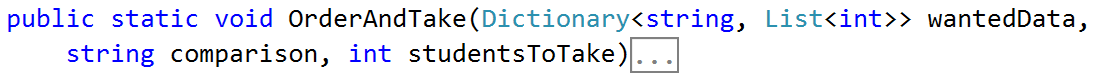
We should be ready with the filtering repositories class and it’s time to move on to the sorting repos’ class.

# Part II: Sorted Students Query

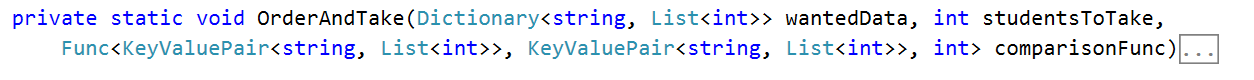
## Implement Sorters

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** **it** **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:

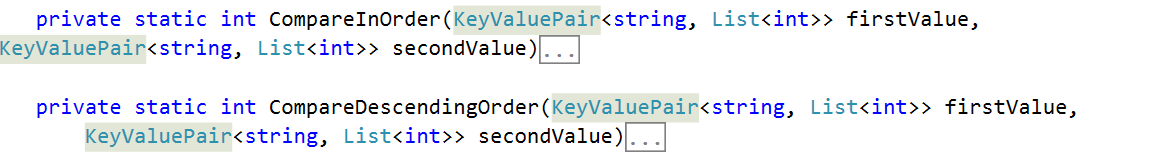


Since the **public** **method** **receives** the **wanted** **sorter** **as** a **string** here we’ll decide which sorting method to use. Again we will put the real sorting in another method. Similarly it can be called **OrderAndTake**, however it’s going to be again a **private** **static** **void** 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:

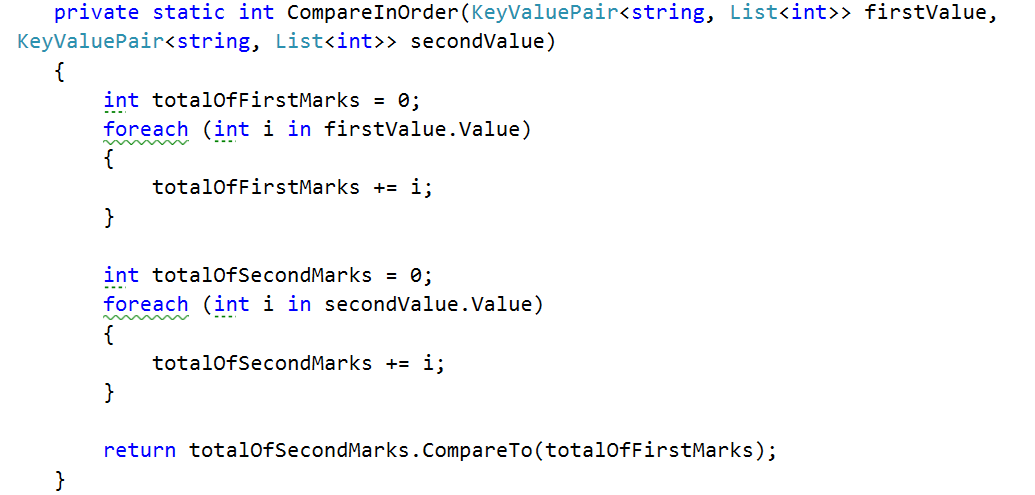


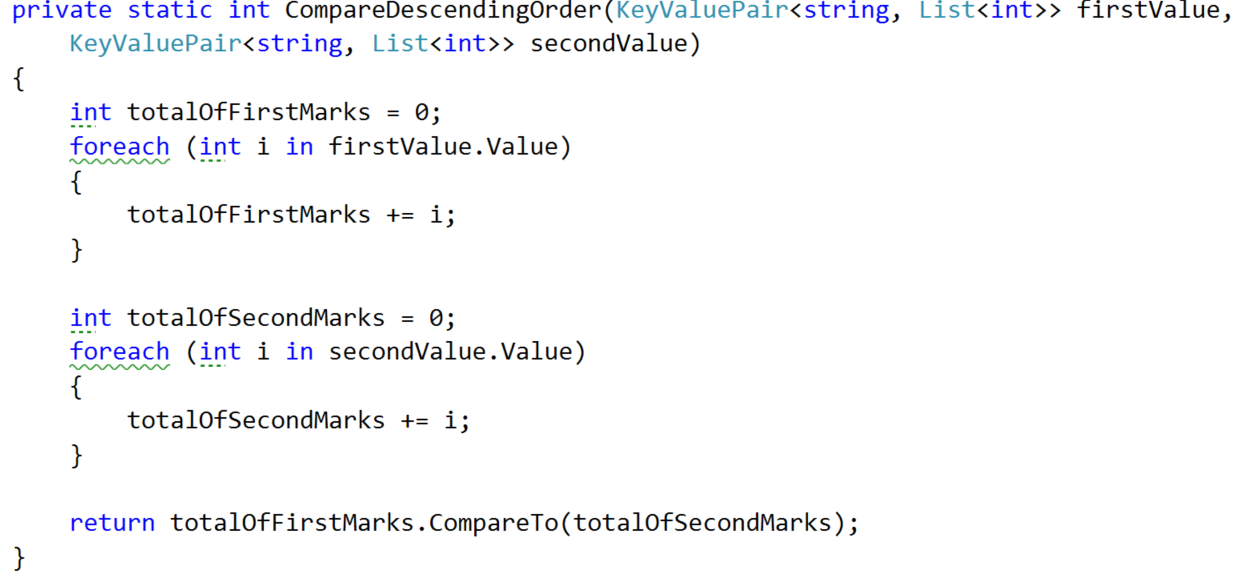
Now we need to **implement** the **functions** that are actually **going** **to** **be** **our** **comparison types**, in order to figure out **how** the **OrderAndTake method** is going to **work.**

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



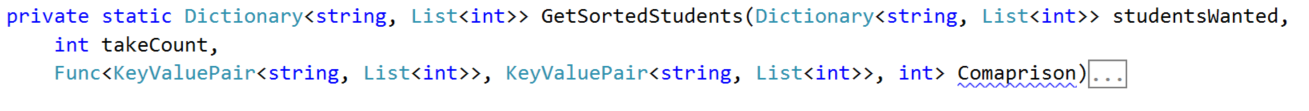
We have to **compare the two students** by a given way and **return 1, 0 or -1** depending on which one is greater/smaller. To compare them **in order**, we compare the sum of the scores of all tasks and return the result of the **second compared to the first**. For the **other one** we do the **same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





## Implement Private OrderAndTake Method

Now that we are done with the helper methods 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 which should contains a string and a list of integers called **studentsSorted** that is equal to the GetSortedStudents method. We haven’t talked about it yet but it’s signature should look like this:

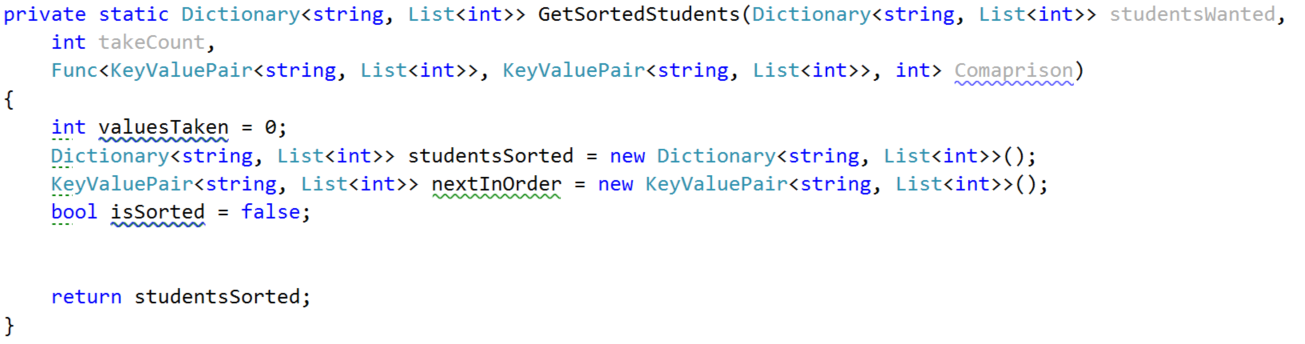


After we’ve gotten the sorted student in a dictionary, we simply print it on the output writer using the PrintStudent method.

## Implement Private GetSortedStudents

The first thing we do in this method is to **set up** a **variable** **for** the **number** of **values** **taken** and **set** it **to** **zero** to help us return only the requested amount of students. Next **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.

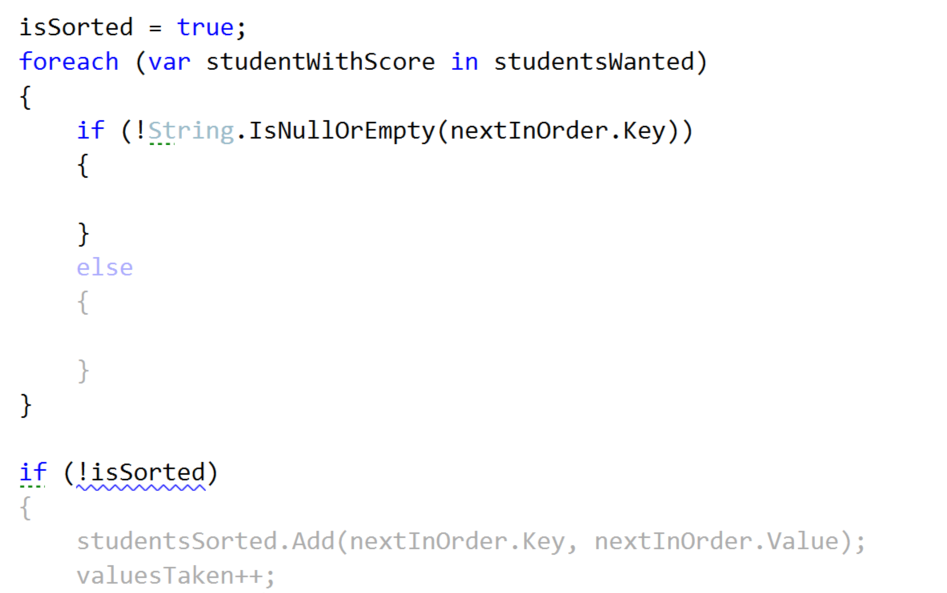
Now it’s time to **implement** an easily **understandable** **sorting** **algorithm** and for that reason we’ve chosen the **bubble** **sort algorithm**. For the job you need to **add** **one** **final** **helper** **variable** of **Boolean** **type** that is **called** **isSorted**, because the **bubble**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



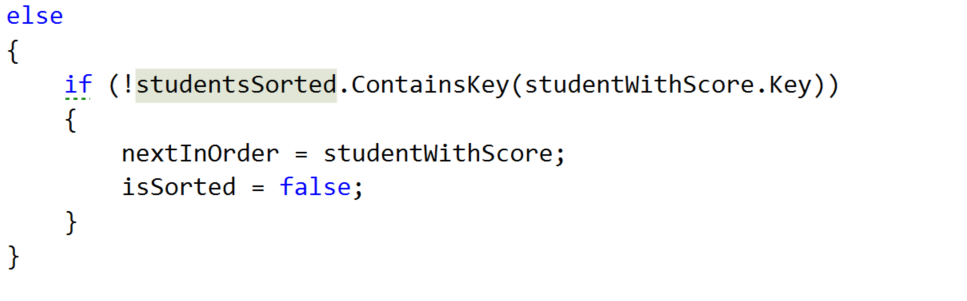
From now on we **place** the **while** **loop** **of** the **bubble** **sort** and on each iteration we first **set** the **is** **sorted** **to** **true**. **At** the **end** **of** the **loop** we **check** **if** the **isSorted** **bool** is **not** **true** and **if** **so**, **add** the **data**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

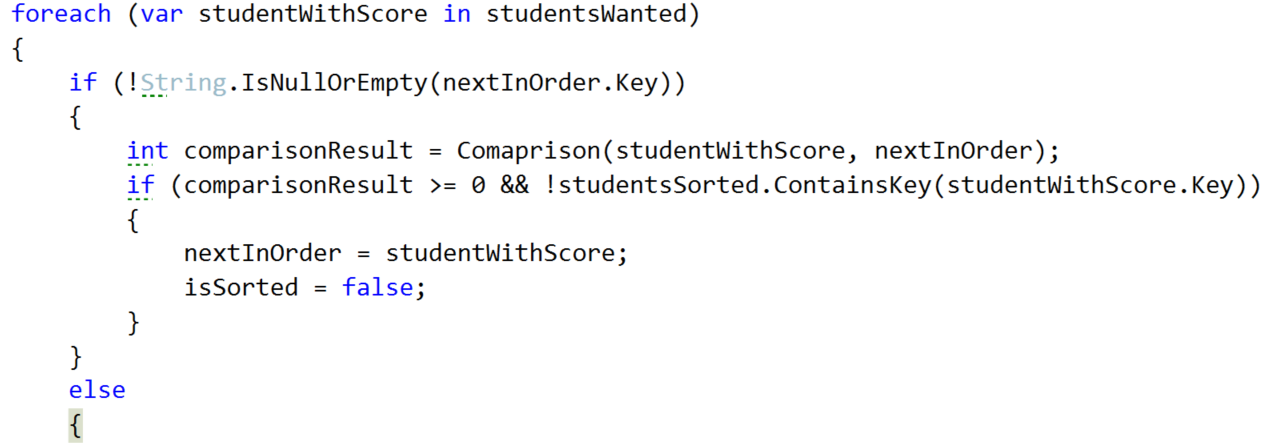


**Next** **thing** in the queue for **implementation** **is** the **inner** **loop** that **finds** the **current** **min**/**max** **element**. For that reason we **make** a **new** **foreach** **over** the **studentsWanted**. The keyvalue pair **nextInOrder** could be **set** **but it could also null** **so** we may have a **null** **key** and a **null** **value**. So we can check **if** the **nextInOrder’s** **key** is **not** **null** or **empty** and **do** something and **if** **not** **do** **another** thing:

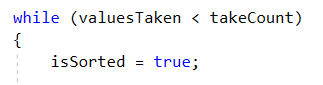


Let’s first **implement** the **else** **clause**. In there we have to **check** **whether** the **new** **sorted** **dictionary** **does** **NOT** **contain** as a **key** the **current studentWithScore’s key**. **If** **so**, we **set** the **nextInOrder** **to** the **studentWithScore** and **set**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



Waiting up next is the if clause above. We **take** the **int** that our **Comparison** function **returns**, **by** **passing** to 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**.  
  


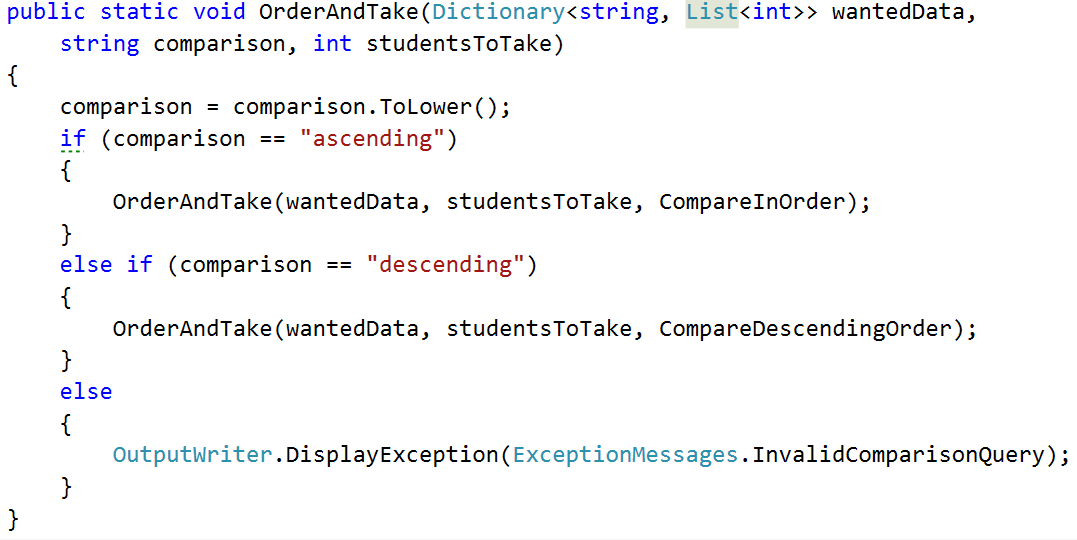
Don’t forget to set the condition of the while loop to stop when we gather the needed students or else you are going to end up with an endless cycle.



Now that we are ready with the **GetSortedStudents**, 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

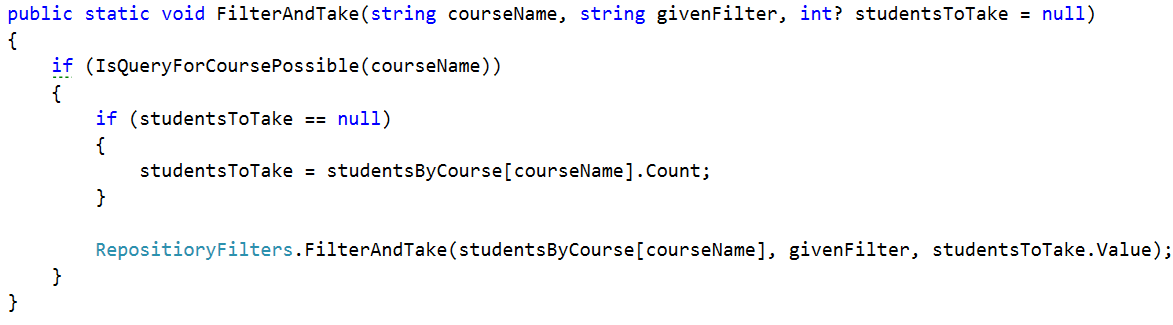
Here our only job is to decide how to **choose** **which** **comparison** **type** **to** **use**. That is why we do pretty much the same thing as in the public FilterAndTake. First we check **if** the **comparisonType** string is **ascending** and if so, **call** the private **OrderAndTake**, **passing** 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!”



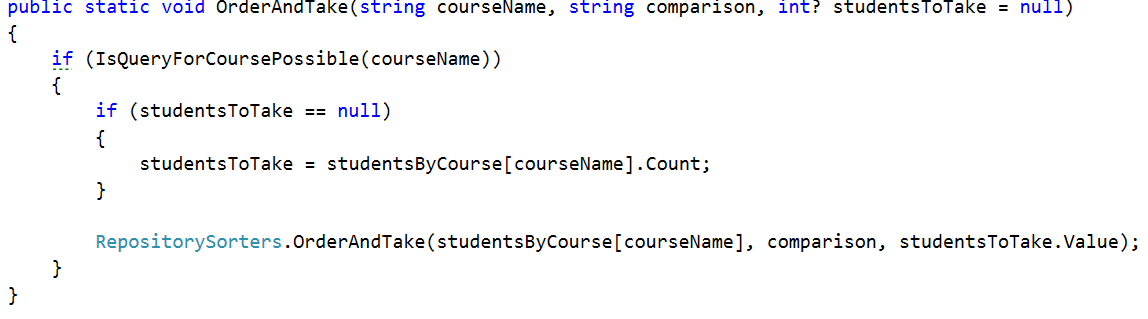
# Student Repository Implementation Part of Filters and Sorters

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 we are going to **add** **two** **methods** in **this** **class**. **One** that is **called** **FilterAndTake** and **another** **one** **OrderAndTake**. The **filter** **follows** the **following** **signature**:



If you’re wondering why the studentsToTake 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 been parsed (this happens 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** once we’re **in** the **StudentRepository** **class**. If you are confused, don’t worry it’s harder to explain that to see it in code.   
  


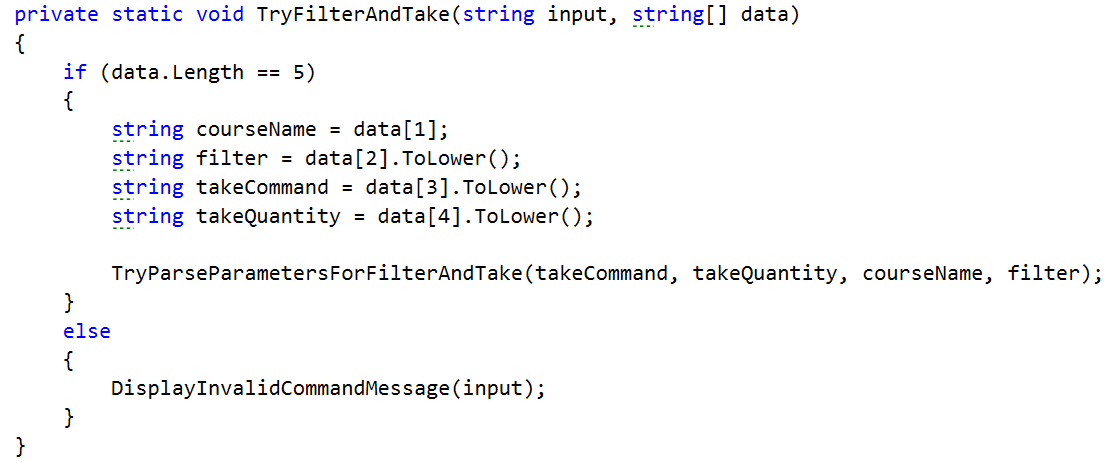
The situation in the OrderAndTake method is pretty much the same as you can see:

  
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.

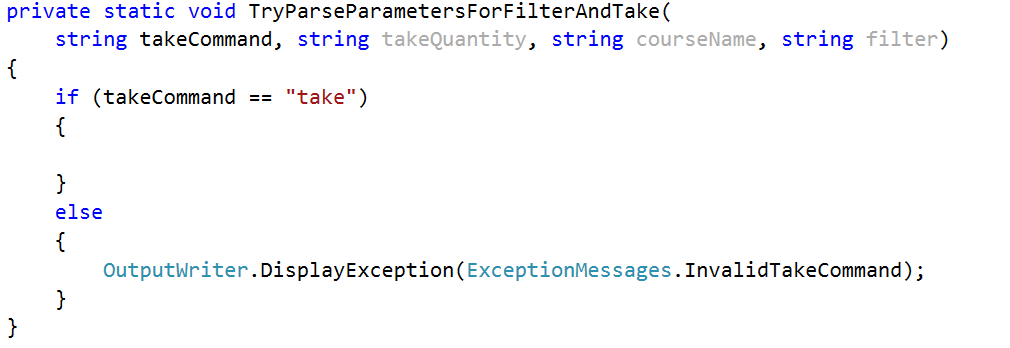
**In** the **command** **interpreter** we should **make** **two** **methods** **called** **TryFilterAndTake** and **TryOrderAndTake** that **take** **input** **parameters**, the **same** **as** **all** the **other** **try** **methods** in this class. After making them we should **call** **them** **in**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

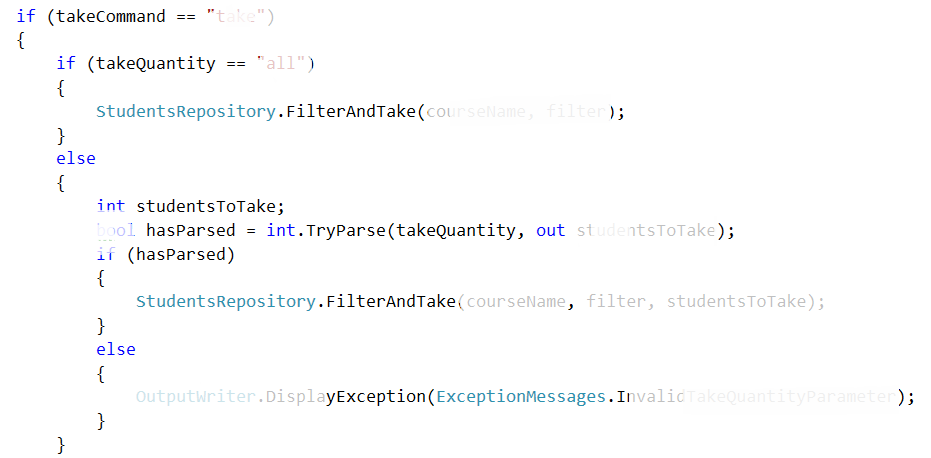
## Implement Filtering Data Parsing in Command Interpreter

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** **is** **5** and **if** **not**, **DisplayInvalidCommandMessage**. **If** it is, 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 its implementation.

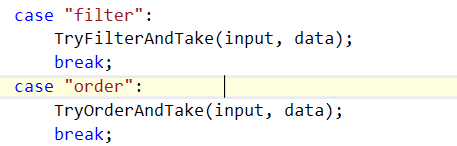
First we **check** **if** the **take** **command** **is** actually **equal** **to** the word “**take**” and **if** **not** we **print** an **exception** **message** on the **output** **writer**, which of course we should first **add**, called **InvalidTakeQuantityParameter** with a **message** “The take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



If this is the actual command then we have to **check** **if** the **take** **quantity** **is** “**all**”. I**f** **so**, **call** **FilterAndTake** **from** the **StudentsRepository** **without** the **last** **parameter** **for** the **quantity** and therefore 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 implementation of the **TryParseParametersForOrderAndTake** is the **same** so we 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**.

