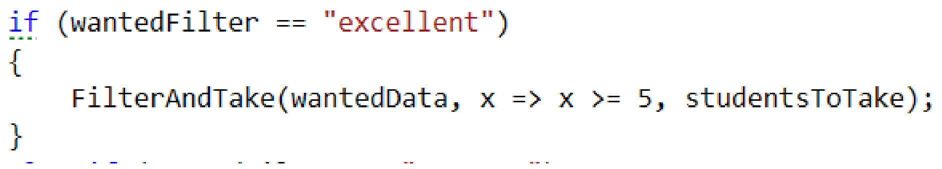
Lab: LINQ

In the current piece we are going to jump from one place to another in our code and see if we can do some of the things with less code. This is our goal, because that can improve the readability of the project and that's something you are obliged to do.

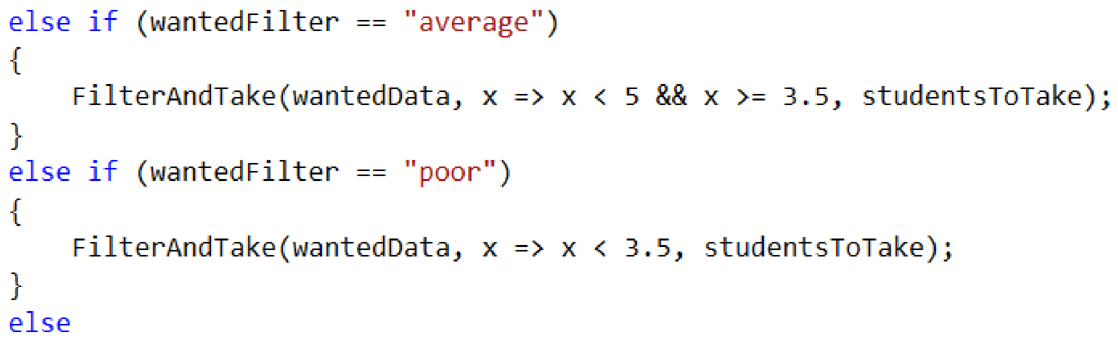
Problem 1. Change Predicate Methods with Lambda Expressions

The first thing we are going to change is actually not connected to the LINQ. All we want to change here is in the RepositoryFilters in the public FilterAndTake. In the previous piece we made 3 methods that filter the given data. However, we have the possibility to delete the 15 rows that the methods take in the code. In the first method, where the wantedFilter is excellent we can easily express a predicate using lambda. We have only one input parameter so our statement in the call of the private FilterAndTake should look something like this:



```
if (wantedFilter == "excellent")
{
    FilterAndTake(wantedData, x => x >= 5, studentsToTake);
}
```

Next up is the average filter and after it the poor filter, however they are pretty much the same as the one above. For that reason, we are not going to discuss them any further and just change the calls of the filter methods with the appropriate lambda expression:



```
else if (wantedFilter == "average")
{
    FilterAndTake(wantedData, x => x < 5 && x >= 3.5, studentsToTake);
}
else if (wantedFilter == "poor")
{
    FilterAndTake(wantedData, x => x < 3.5, studentsToTake);
}
else
```

Now we can easily delete the three methods from the repository filters class.

Problem 2. Use LINQ Average Instead of a Custom One

What you can do now is delete the Average method so that we can replace it with the one that comes from the LINQ. In the private FilterAndTake we should change the averageMark's name to averageScore and its value equal to the the one we just said above. Next thing we need to make is a new variable called percentageOfFullfillments equal to the average score devided by the maximal score on a task which is 100. Finally, we should make one last variable that is the actual mark and it is equal to the percentage of fulfillment multiplied by 4 and summed with 2 after that. Here is how the situation in the private FilterAndTake method should look:

```
double averageScore = userName_Points.Value.Average();
double percentageOfFullfilment = averageScore / 100;
double mark = percentageOfFullfilment * 4 + 2;

if (givenFilter(mark))
{
```

Problem 3. Changing Structure in Repository Sorters

I know you may not like that but now we've done the sorting the hard way, we can easily replace it with the easy and more readable way. For that reason, we are going to extract only one method that we can reuse for our functionality. From the private method OrderAndTake we should extract a new method for printing the sorted students. It's declaration and implementation should look like this:

```
private static void PrintStudents(Dictionary<string, List<int>> studentsSorted)
{
    foreach (KeyValuePair<string, List<int>> keyValuePair in studentsSorted)
    {
        OutputWriter.PrintStudent(keyValuePair);
    }
}
```

Now you can easily delete all the methods except the public OrderAndTake and the one we just extracted. So after the deletions of all occurrences, the class should look like this:

```
public static class RepositorySorters
{
    public static void OrderAndTake(Dictionary<string, List<int>> wantedData,
        string comparison, int studentsToTake)
    {
        comparison = comparison.ToLower();
        if (comparison == "ascending")
        {
        }
        else if (comparison == "descending")
        {
        }
        else
        {
            OutputWriter.DisplayException(ExceptionMessages.InvalidComparisonQuery);
        }
    }

    private static void PrintStudents(Dictionary<string, List<int>> studentsSorted)
    {
        foreach (KeyValuePair<string, List<int>> keyValuePair in studentsSorted)
        {
            OutputWriter.PrintStudent(keyValuePair);
        }
    }
}
```

No matter whether the wanted filter I ascending or descending, we'll want to call the PrintStudents method passing it the sorted Dictionary.

```
public static void OrderAndTake(Dictionary<string, List<int>> wantedData,
    string comparison, int studentsToTake)
{
    comparison = comparison.ToLower();
    if (comparison == "ascending")
    {
        PrintStudents();
    }
    else if (comparison == "descending")
    {
        PrintStudents();
    }
    else
    {
        OutputWriter.DisplayException(ExceptionMessages.InvalidComparisonQuery);
    }
}
```

First we are going to implement the ascending comparison and then we'll need to only change one word and copy the sorting in order for it to work for descending.

Since we have a method for ordering a collection from LINQ, we are going to use it out of the box.

```
if (comparison == "ascending")
{
    PrintStudents(wantedData.OrderBy());
}
```

What the function wants is the criteria which to use to sort, so we'll pass it the sum of the scores on the tasks.

```
if (comparison == "ascending")
{
    PrintStudents(wantedData.OrderBy(x => x.Value.Sum()));
}
```

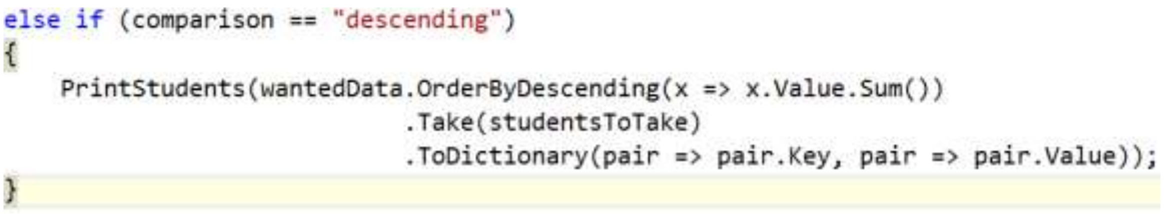
Now we should take the number of wanted results:

```
if (comparison == "ascending")
{
    PrintStudents(wantedData.OrderBy(x => x.Value.Sum()).Take(studentsToTake));
}
```

Finally we should make it to a dictionary since now it's a collection of different type:

```
if (comparison == "ascending")
{
    PrintStudents(wantedData.OrderBy(x => x.Value.Sum())
                            .Take(studentsToTake)
                            .ToDictionary(pair => pair.Key, pair => pair.Value));
}
```

Now we can do the same thing but only change the OdrerBy with a OrderByDescending:



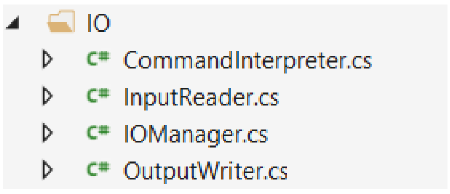
```
else if (comparison == "descending")
{
    PrintStudents(wantedData.OrderByDescending(x => x.Value.Sum())
                            .Take(studentsToTake)
                            .ToDictionary(pair => pair.Key, pair => pair.Value));
}
```

Now you can test the functionality of the filters and sorters and see if they still work.

Problem 4. Creating folder structure

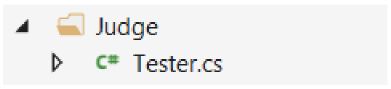
There are no more things we can change using LINQ so I suggest we use the current piece so that we can at least order the structure of our project at least a little bit.

We can make 4 folders in the current project called IO, Judge, Repository, Static data. In the IO folder we can put all the following things:



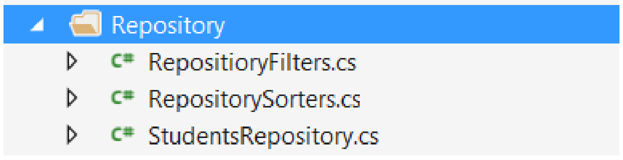
```
◢ IO
    ▷ C# CommandInterpreter.cs
    ▷ C# InputReader.cs
    ▷ C# IOManager.cs
    ▷ C# OutputWriter.cs
```

In the Judge folder as you can imagine, we'll put the Tester class:



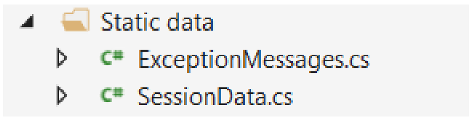
```
◢ Judge
    ▷ C# Tester.cs
```

In the repositories folder we'll put everything related to the repository:



```
◢ Repository
    ▷ C# RepositioryFilters.cs
    ▷ C# RepositorySorters.cs
    ▷ C# StudentsRepository.cs
```

And finally in the static data we should put the ExceptionMessages and the SessionData.



```
◢ Static data
    ▷ C# ExceptionMessages.cs
    ▷ C# SessionData.cs
```

This should be everything to change in the current exercise. Next time we are going to learn how to download files from the Internet and that is going to be the last thing to implement in the BashSoft.