# BashSoft – OOP

This document defines the project overview for the ["C# OOP Basics" course @ Software University”](https://softuni.bg/courses/csharp-oop-basics). 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-basics).

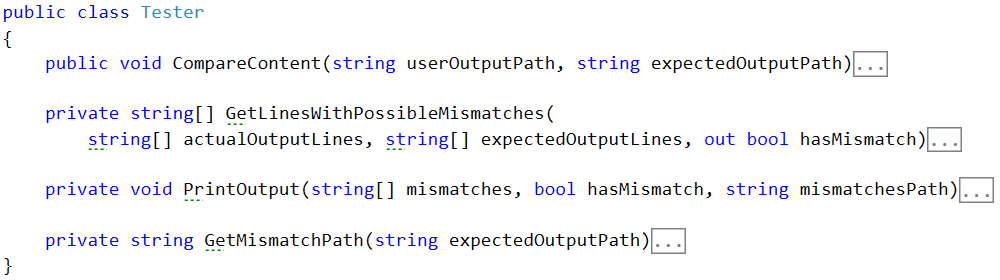
# Defining Classes and Methods

## Introduction

We added a lot of functionality during the last course. However, the code we wrote was using only static methods. Now the time has come for us to start following the principles of writing good OOP code. We are going to start by replacing some of the static members with instance ones. Note that we should start from the classes that **don't depend on any others**.

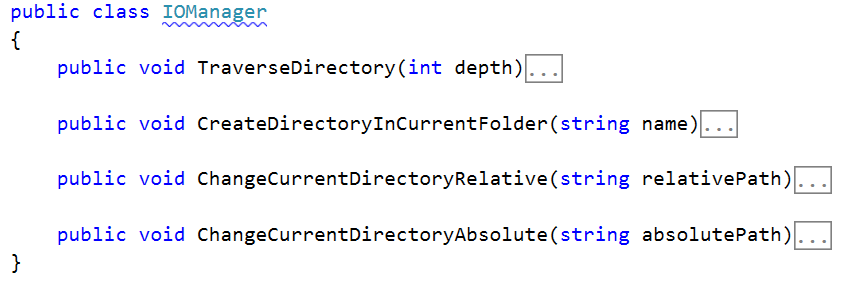
## Refactoring the Tester

In this class there is no need of a constructor so the only thing we have to do is make everything **non-static**.



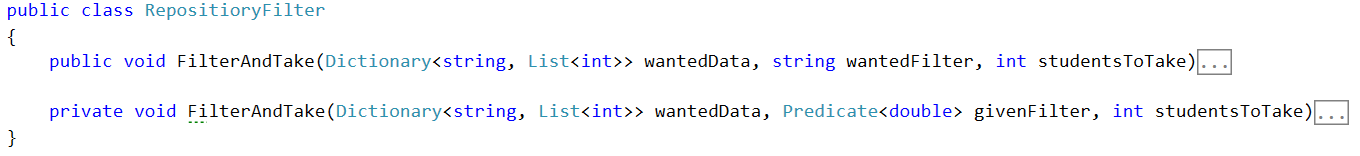
## Refactoring the IOManager

Same deal as the **Tester** class:



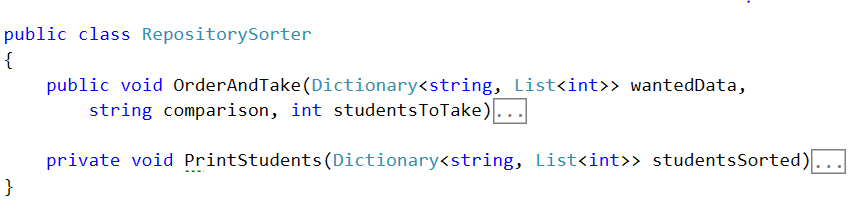
## Refactoring the RepositoryFilters

First rename the class to **RepositoyFilter** (without the s). We do this because now it is an instance class, instead of a static one. Also remove the static keyword everywhere again.



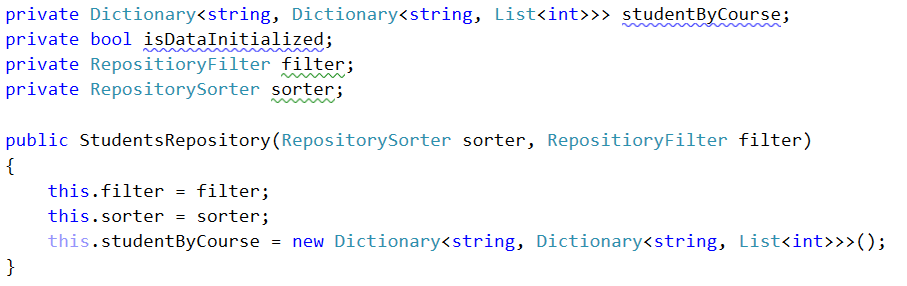
## Refactoring the RepositorySorters

Repeat the same with the **RepositoryFilter** class.

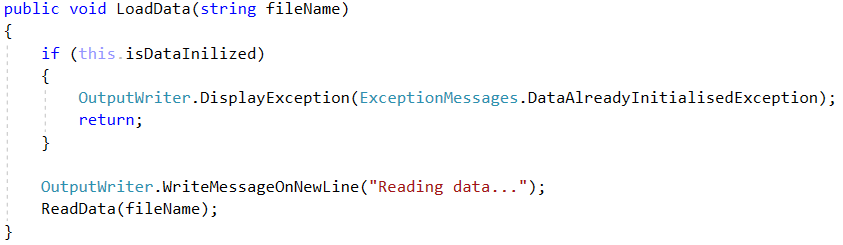


## Refactoring the StudentRepository

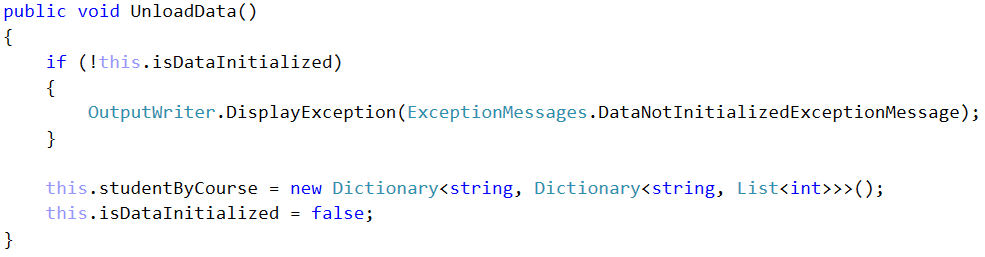
First remove every static word you see and set the present fields to private. Since we've changed the **RepositoryFilter** and **Sorter** we now have to make fields of these classes in the **StudentRepository.** And we can make instances of them in the constructor:



As you can see we have moved the initialization of the data structure to the constructor. So now we need to rename the **InitializeData** method to **LoadData** and remove the initialization from there, otherwise its implementation stays the same.



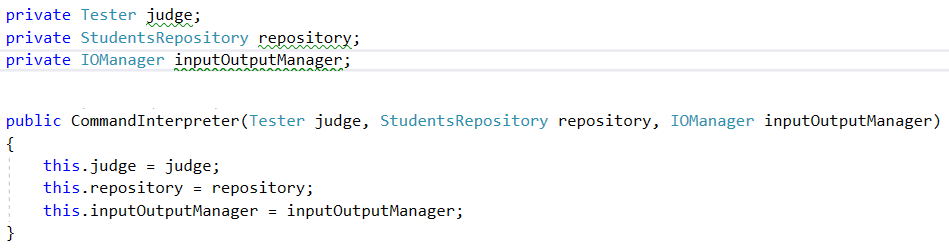
We will also create an **UnloadData** method which will do the exactly what is says:



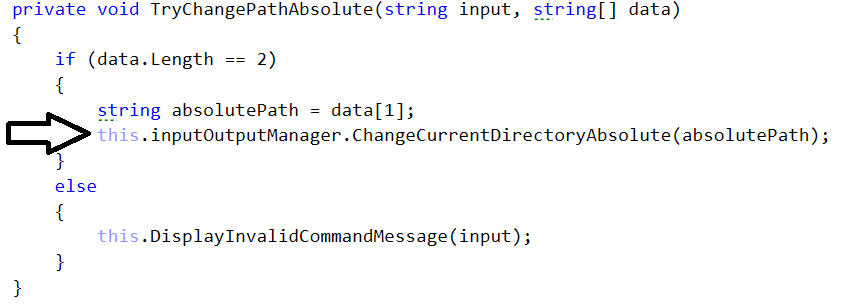
## Refactoring the CommandInterpreter

As always - start with removing every static word you find *(hint - use your IDEs find and replace all functionality and don't do it by hand)*. As you can see many errors showed up in the error list of this file, so let’s fix them all. They appear because until now we've been using only static classes and now we need to replace them with instances of these classes.

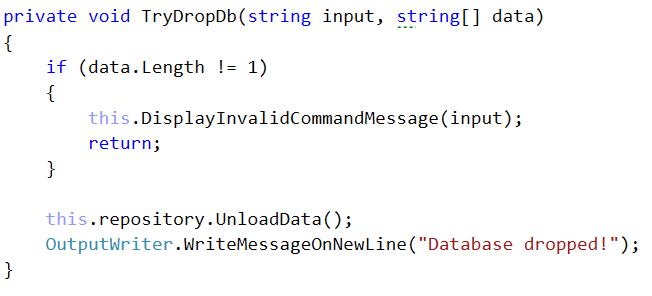
To do this we will create **fields** in the **CommandInterpreter** and **set** them in its **constructor**:



So now fix the method calls. Here is an example, you would have to do the others by **yourself**:



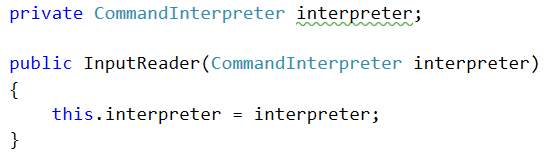
We also need to create a new command called "**dropdb**", so add such a case to the switch with the according method call. Then create the according method.



## Refactoring the Input Reader and the Main method

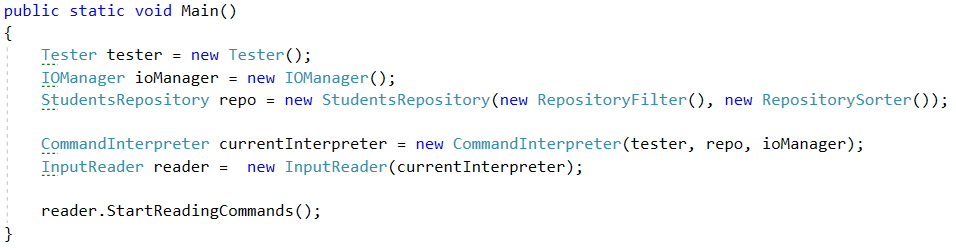
Start with the thing we always start with (psst… static).

Also we have to make a **constructor** for the class. It will receive as parameters a **CommandInterpreter**. We also have to make an appropriate field and replace the old calling with it.



By now you should not have any errors related to our classes.

In the Main method of our application we need to initialize everything trough theirs constructor.



## Creating a class Student

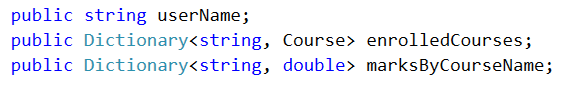
Before we start with the class - create a Folder which will hold all our models.

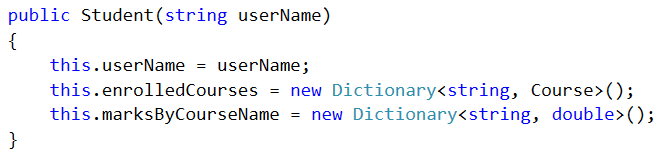
C:\Users\tilchev\Desktop\image_2016-06-22_13-56-07.png

It will have fields (for now you can make them public, we will fix that in the encapsulation part):

* userName
* enrolledCourses
* marksByCourseName

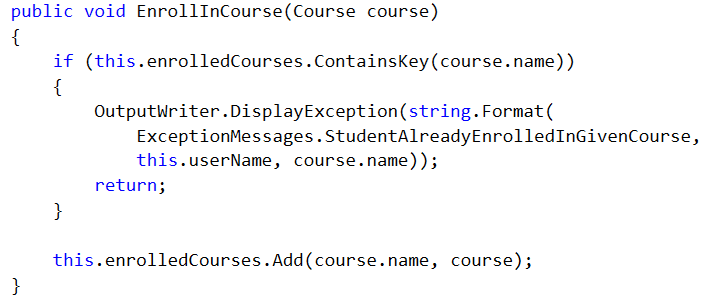
**! Try to make them by yourself before looking at the screenshot!**



Make a **Constructor** which **initializes** all the **fields**.

Our class has the following **Methods**:

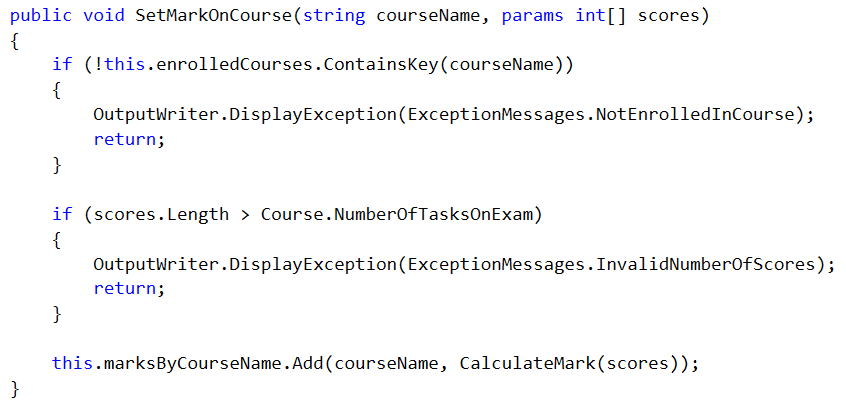
* **EnrollInCourse** - used for enrolling the current student in a certain course. Notice in the code bellow how we first check for the exceptional case(s). This is part of a programming approach called "Defensive programming". The message should be "The {0} already exists in {1}.".



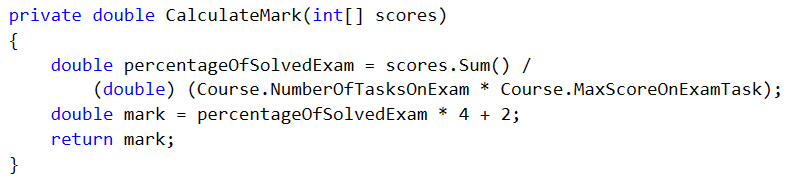
* **SetMarksInCourse** - used for setting the current students' average mark in a certain course. Notice how we use a **params** array to pass as many scores as we want to the method.

The message for **NotEnrolledInCourse** is: "Student must be enrolled in a course before you set his mark.".

The message for **InvalidNumberOfScores** is: "The number of scores for the given course is greater than the possible.".



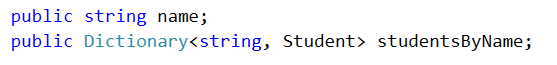
* **CalculateMark** - this is only a helper method to calculate the average mark from all the scores we get. As such we can leave it to be private. We use a certain formula to do this:



## Creating a class Course

It will have fields:

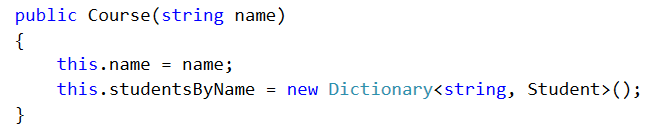
* name
* studentsByName



**Constant fields**:

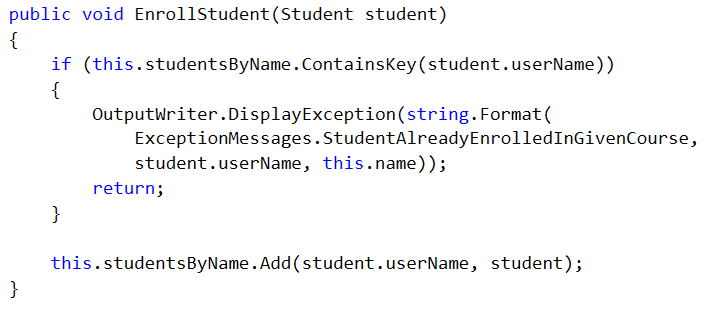


**Constructor**:



**Methods**:

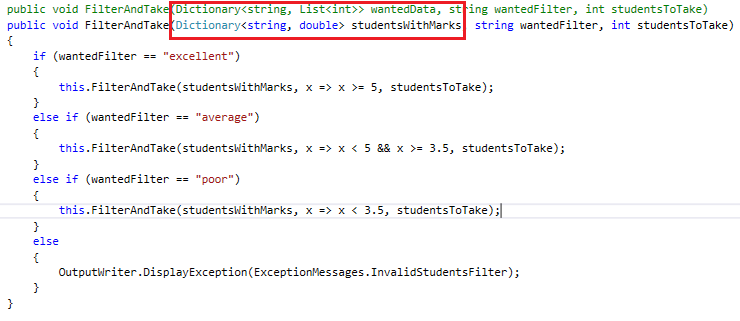
* **EnrollStudent** - this method will enroll a certain student in the current course.



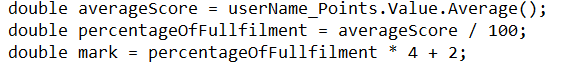
## Reworking the RepositoryFilter class

Now that we've made ourselves some pleasent classes for **Students** and **Courses** we can use them to simplify all the classes in the Repository folder. Lets start with the **RepositoryFilter**:

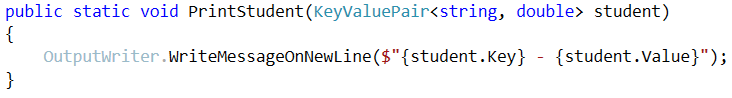
First off we need to change the Dictionary that we receive as a parameter in the **FilterAndTake** methods (both public and private ones). The value of the dictionary will now be simply a **double** which will represent the final mark of the student in the wanted course. We will also make the name of the dictionary more descriptive:



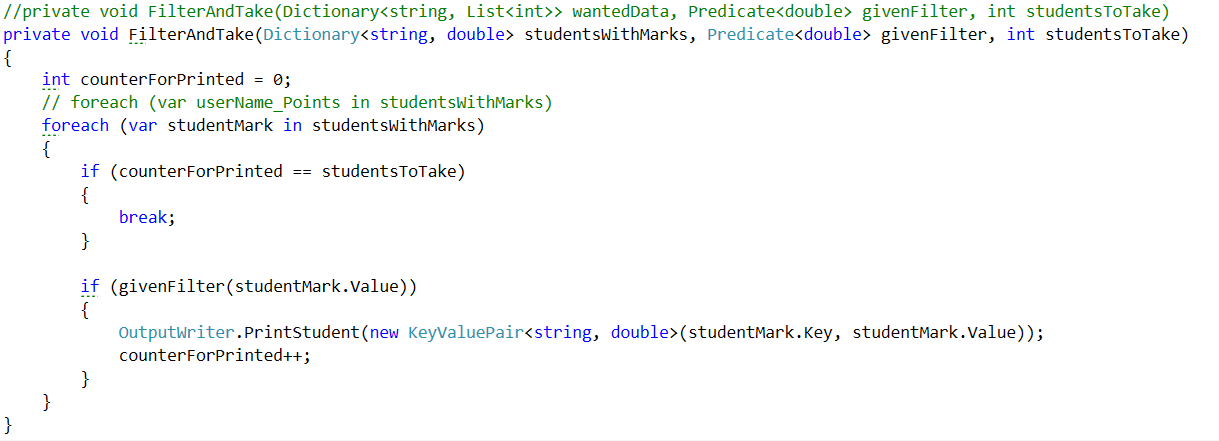
Do the same with the private method. Since we are going to receive a student with his mark we don't need to calculate the mark here. So we remove that code:



After you've deleted them you will notice that the **PrintStudent** command gives an error. That's because it waits for a **KeyValuePair<String, List<int>>** so we need to change that to **KeyValuePair<String, Double>:**

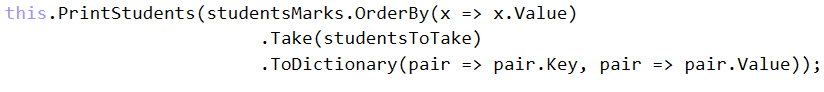


Now we need to return to the **RepositoryFilter** and give the **PrintStudent** method its corresponding arguments. Finally, the method should look like this:



## Reworking the RepositorySorter class

First change the dictionary parameter in the methods the same way as in the **Filter**. Next we want to change the lambda expression for ordering to match our new **KeyValuePair**:



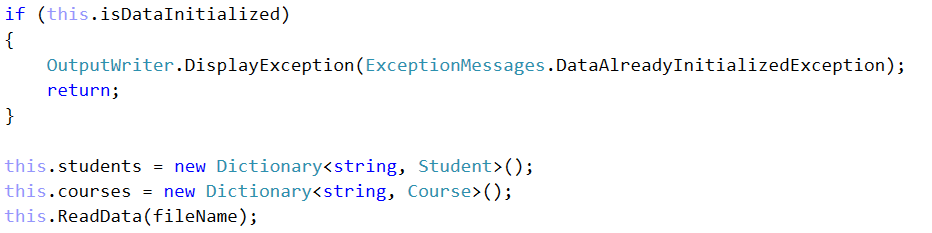
Fix the rest of the errors in the class by yourself.

## Reworking the StudentsRepository class

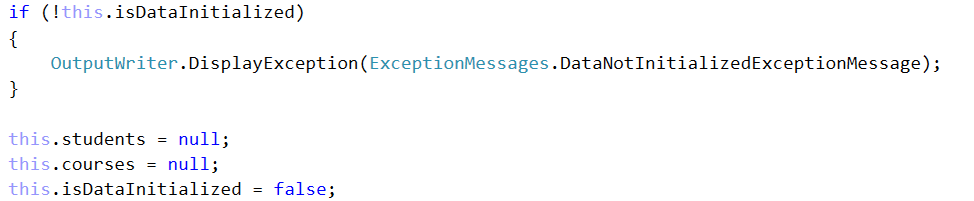
First off we need to delete the old structure where we kept our data, because we will use our new shiny classes instead. The new data structure will consist of two Dictionaries: one for the courses (**courseName -> Course**) and the other for the students (**studentName -> Student**).

C:\Users\tilchev\Desktop\image_2016-06-22_14-33-14.png

Next thing we need to do is initialize the two structures in the **LoadData** method. Here is an example:

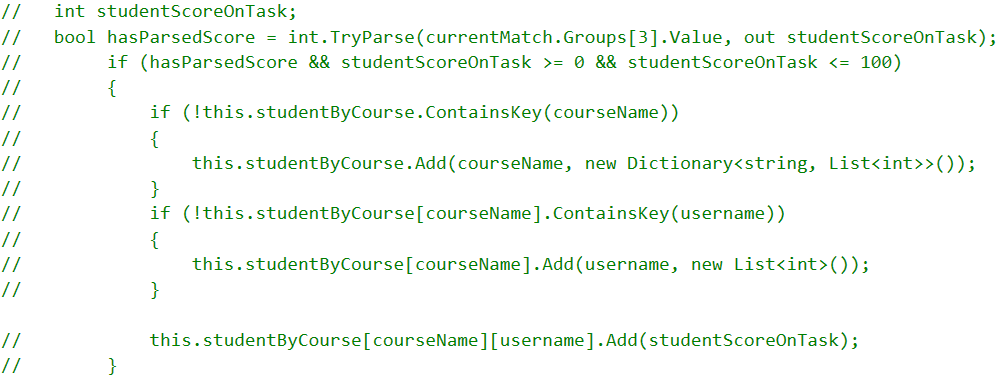


In the UnloadData method we will do the exact opposite - set them to null.



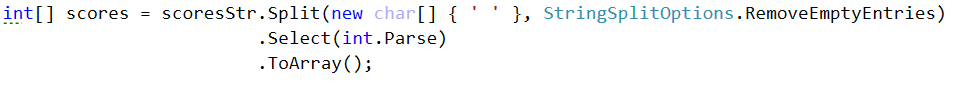
Next stop is the **ReadData** method. First change the regular expression because we've made some changes to the database. Now we will have students with both first and last name. Also, they can have from 1 to 5 scores from tasks. Here is the new regex: ([A-Z][a-zA-Z#\++]\*\_[A-Z][a-z]{2}\_\d{4})\s+([A-Za-z]+\d{2}\_\d{2,4})\s([\s0-9]+)

Now since we're changing the way we get the scores for each task we need to handle the parsing of the third group of the match. Until now we had a boolean that has been telling us whether the score has passed. There was also an Integer that held the actual value after the parsing. You can delete the following code block:s

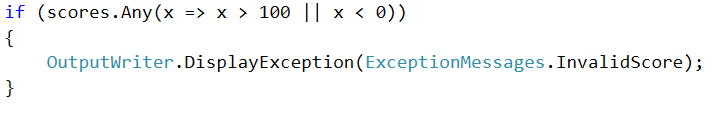


Now we have to take the third groups' value and save it in a string called **scoresStr**. Since we are going to make some unprotected parsing you are going to make a new try/catch block. The code in the try block will do the following:

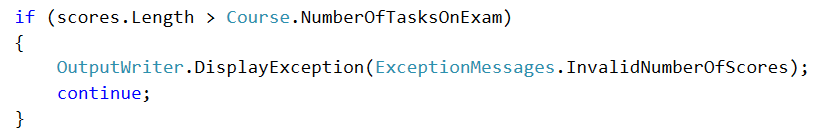
* Try to split, parse and collect the string in an integer array.



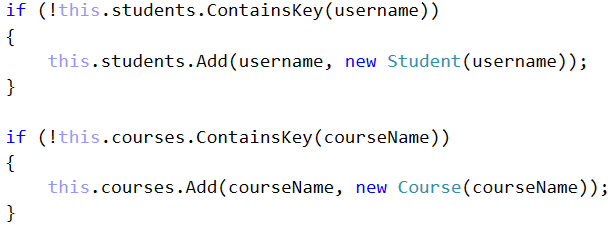
* Check whether any score is above 100 or below 0 and if so display the exception message: "The number for the score you've entered is not in the range of 0 - 100".



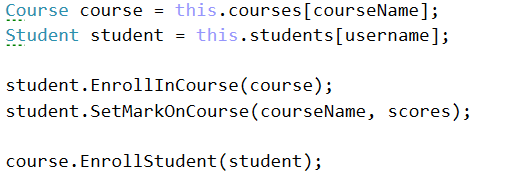
* Check whether the scores are more than the maximum **NumberOfTasksOnExam**



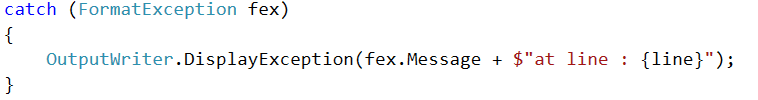
* Check if both of our data structures contain their corresponding key and if not add it with according new object:



* Finally call the methods we made in the **Course** and **Student** classes so that our base is up.



In the catch block we will display the **FormatException** we might throw.



Now we have to change the **IsQueryForCoursePossible** method. Change the data structure in which we have to check whether such a course exists:



In the **IsQueryForStudentPossible** we should basically do the same, but first we should take the wanted course and from it the studentsByName structure. Then we check whether it contains the wanted student:



In the **GetStudentScoresFromCourse** method we need to change only the printing according to the new **KeyValuePair** we use for that. The tricky part is getting the value because we need to go deep.

**wanted course** -> **wanted student inside course** -> **chosen students marks for course**



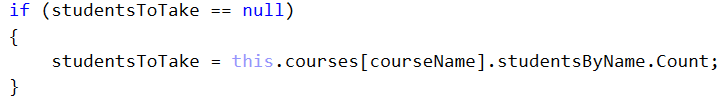
In the **GetAllStudentsFromCourse** **-** here the foreach needs to iterate over the students by name in the corresponding course:



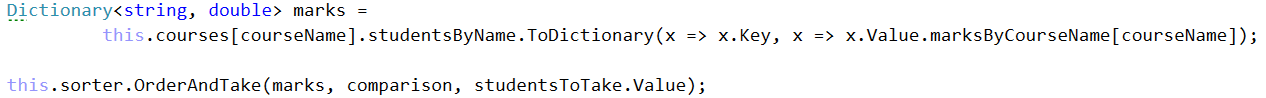
And in the foreach we can delete the print student method call because we can re-use the previous method.



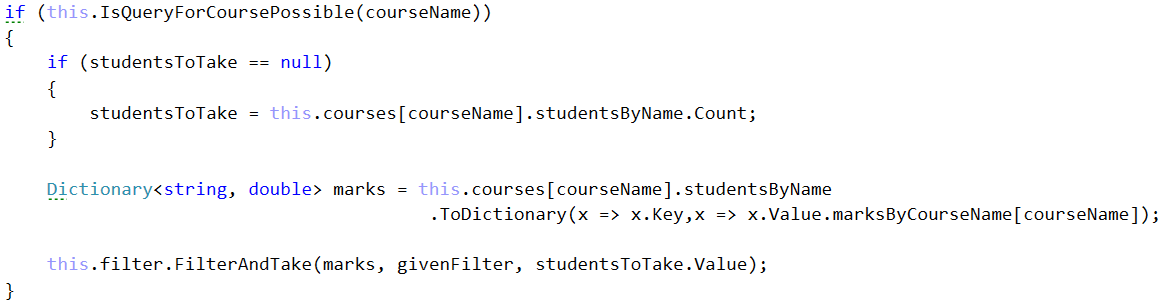
In the **OrderAndTake** method in its if-block where we check whether the students to take is null we should take the count of the students by name from the corresponding course.



In order to pass the dictionary with the marks in the order and take method in the sorter we first need to extract the marks. Here is how we do that - you must figure out why it's done like that on your own.

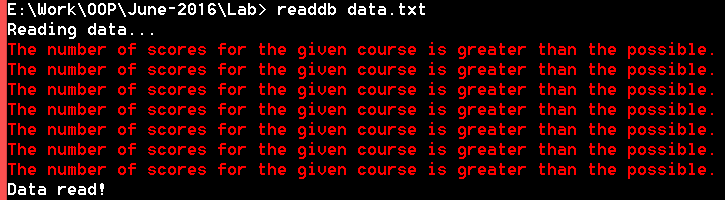


The last method we need to change in this class is the **FilterAndTake** Method. We make the same changes as the previous method.



Finally, we shouldn’t have any more errors and we should easily be able to run the project.

The result of reading the database should look something like this:



The errors here are normal to appear. They are because there are some entries in the end of the file which contain more than 5 scores on tasks. If you like, you can delete them. They are present just for the sake of testing.

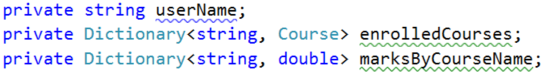
# Encapsulation

## Introduction

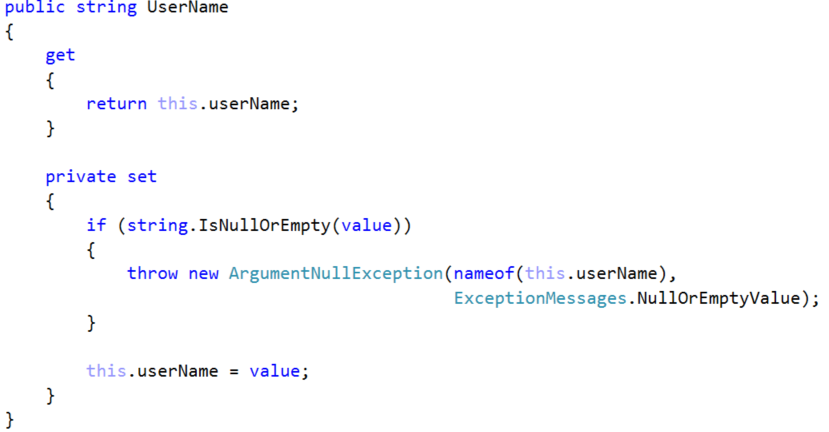
We don’t know if you’ve noticed, but we’ve pretty much used **encapsulation** and **defensive programming** a lot so far. For that reason, there are going to be only a few changes that we’ll need to make here in the current BashSoft part. Until now we’ve used many access modifiers even though you haven’t studied them. Now that you know how they work, you probably are aware why we’ve used **private** in many places and why we’ve used **public** in others. Another thing that is a concern of the current lecture is the act of not giving the user/programmer the possibility to break the program easily. Or in other words make it "durak-proof". So far everywhere we wanted to display an exception we used the **OutputWriter**'s **DisplayException** method. This approach however makes all our classes coupled to the **OutputWriter** and coupled classes (ones that depend on each other too much) are a sign of bad **Object Oriented Design**.

## Encapsulating fields in the Student class

Make all the class **fields** **private**, so they aren't accessible outside of the class.

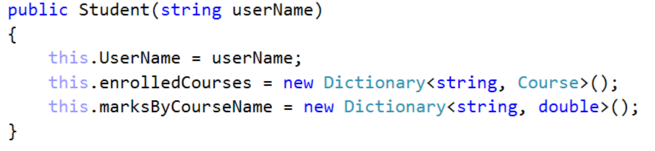


Then **encapsulate** them through **getters** and **setters**. You can put validation in the **setter**:



The first parameter of the **ArgumentNullException** is the variable that causes the exception. The second one is a new message that can be something like: "**The value of the variable CANNOT be null or empty!**"

Use the **setter** in the **constructor** in order to go through the **validation**.



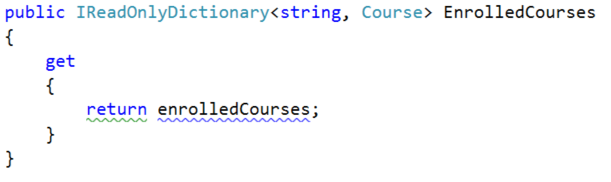
Now everywhere where you used the public field for getting the userName (studentName) you will get errors because it is now private. **Fix that by using the getter instead.**

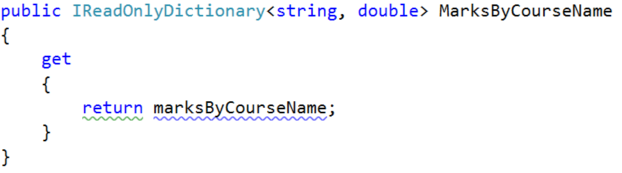
In order to **encapsulate** the **Dictionaries,** we will make **only** **getters** (**read only** properties). There is a problem though. Someone from outside of this class may not be able to set the dictionary to a new one but they could modify its internal contents. We can avoid that in two ways:

The first one is for the **getter** to return a **new collection** which is a copy of the internal field one. This will work but it is a **very memory inefficient** approach because if we do it many times over a large collection it will **allocate new memory each time**.

The second approach is to **mask** our **Dictionary** behind an **interface** which doesn't allow internal modification. This would work better because there isn't **any additional memory allocation**. However, the downside of this is that someone can break that trough a cast.

After carefully considering the trade-offs we decided that the second option is better after all. You however can use the other one if you like. Here is how to do the second one:





Note that we don't need a setter for these fields because they are **read only**. Don't forget to fix everything that that change broke because of making the fields private.

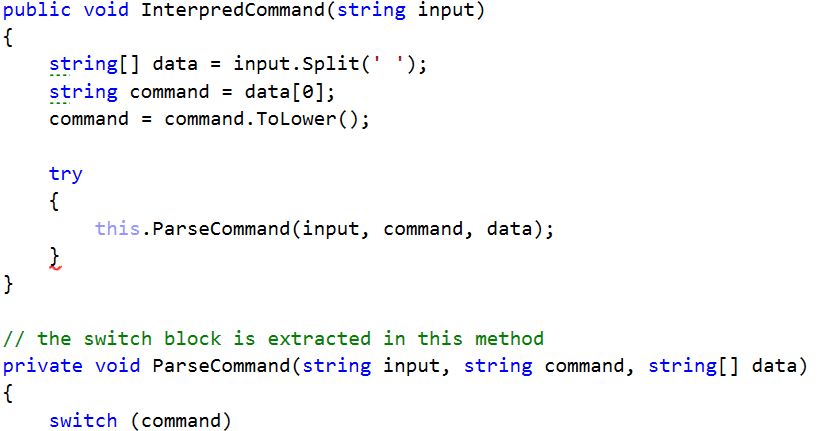
## Encapsulating fields in the Course class

Encapsulate **name** and **studentsByName**. There is nothing new here just do repeat the same steps from Problem 1. **No screenshots, this time - do it yourself.**

## Catching exceptions in a common class

As we mentioned in the introduction we have to replace almost every **OutputWriter.DisplayException** with throwing of a new exception. After that we have to catch them in a common class where we will finally display their messages. This class could be the **CommandInterpreter** because it connects all the other classes.

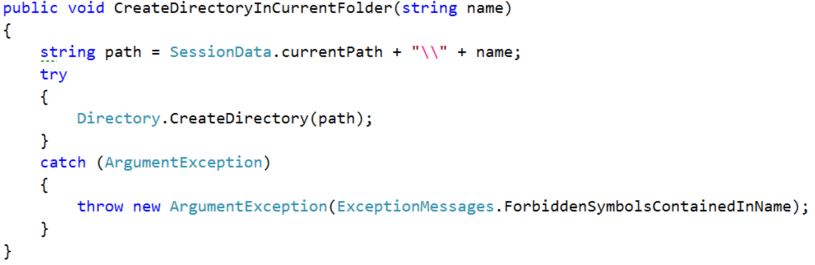
Let's start with the **CommandInterpreter** itself. In order to get it ready for all the exceptions we are going to throw at it we need to surround the switch statement in the **InterpretCommand** method with a try block. But first extract the whole switch block in a new private method, so the code is more **readable.**



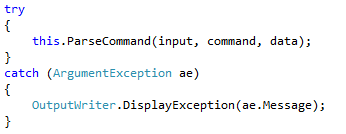
## Throwing Exceptions in the IOManager

Now we are going to refactor the **IOManager.** Let's look at the first method called **TraverseDirectory**. In it we can see that we had a **try-catch block** where we catch a **UnauthorizedAccessException**. In most cases we should re-throw this exception to raise it higher in the hierarchy. However, the case is not such here because if we throw it we are going to get out of the while loop and that’s not what we want because it would break the traversal. So **don't change anything here.**

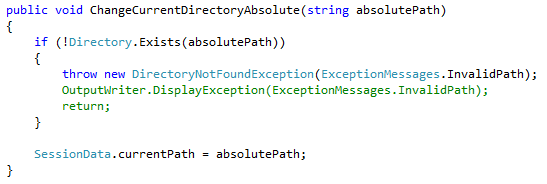
Next up is **CreateDirectoryInCurrentFolder** here we can **re-throw** the argument exception by catching it and throwing a new one:



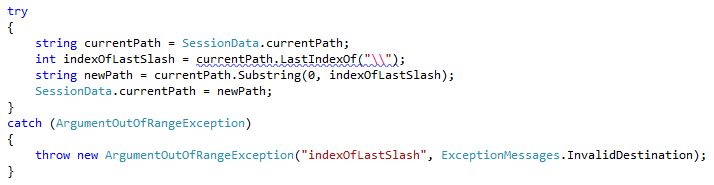
Now that we've thrown an exception we must return to the **CommandInterpreter** and add a **catch block** after the **try block** we made there:



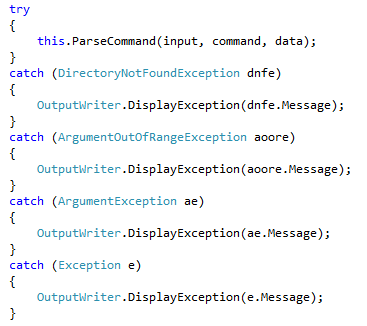
Next is the **ChangeCurrentDirAbsolute** here we can **throw** an **IOException** with the same message that we used to display directly:



**ChangeCurrentDirRelativePath -** here we can **throw** an **ArgumentOutOfRangeException** with the same message that we used to display directly:



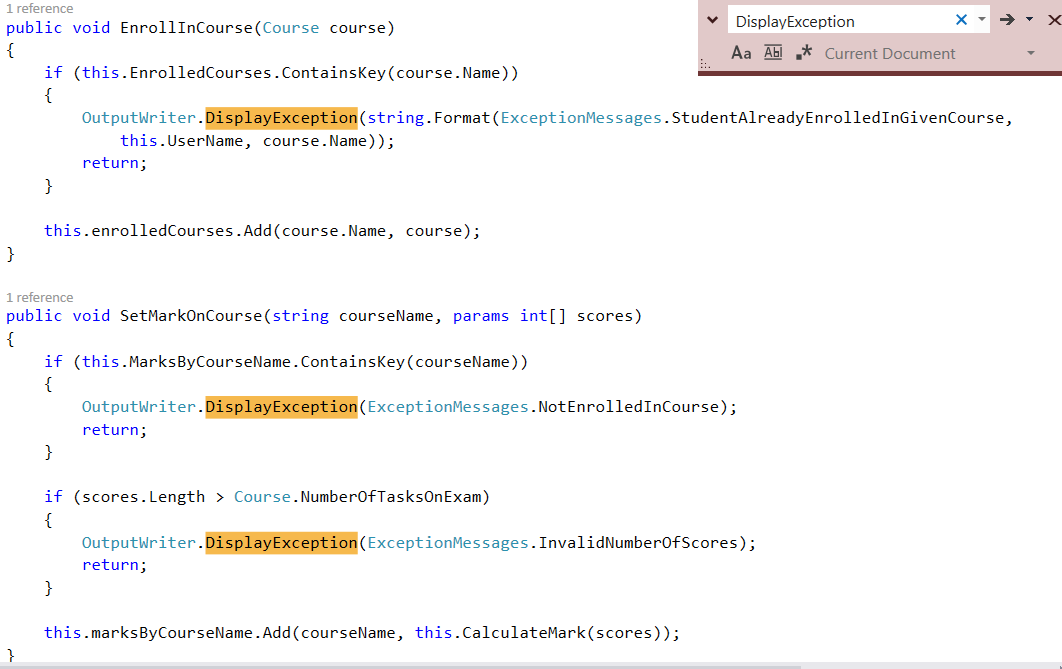
We are done with this class, so it's time to return to the **CommandInterpreter** and catch everything in the correct order:



## Throwing Exceptions in the Tester In the Tester class you may remove everything try-catches in the private methods and leave only the one in the public method CompareContent, by refactoring the catch block. You should have figured out how by now.

## Throwing Exceptions in the Models Folder

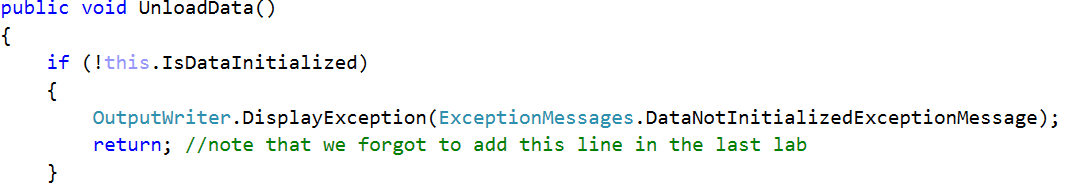
Use the find tool in **Visual Studio** to find every **DisplayException** inthe **Student** and **Course** classes and replace it with a corresponding new exception throw. Carefully consider what exception to throw (use google if you can't think of the right exception fast). Don't forget to pass the **same message** you used to print until now to the exceptions **constructor**.



## Throwing Exceptions in the Repository Folder

There is nothing special about the **RepositorySorter** or **Filter**, so do what we did everywhere else **on your own**.

In the **StudentRepository** class only throw exceptions where we display an exception and then return. For instance, you should change these kinds of blocks:

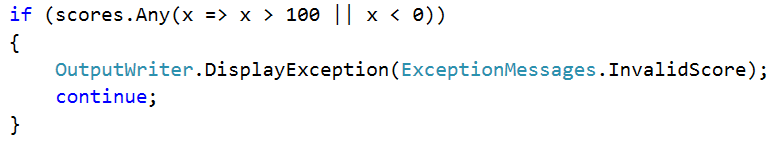


To these:



But you **must NOT change** blocks without a return after displaying an exception, because that would break our functionality.

Example:



Leave these as they are.

## Test if you Broke Anything

If you did all the refactoring **correct** everything should run as before but now your code is one step closer to not being "**smelly**".

Congratulations you completed the **Encapsulation** part!

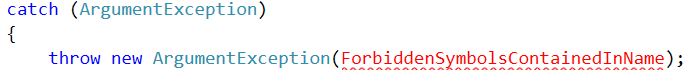
# Inheritance

## Introduction

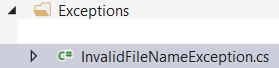
After we replaced "**Display exception**" with **throw new Exception** everywhere in the **Encapsulation** lab, now it's time to make our own **Exceptions** and throw them instead of the ones we have by default. This way we can set the messages of these Exceptions in the body of the class.

## Making Exceptions for the IOManager and Tester

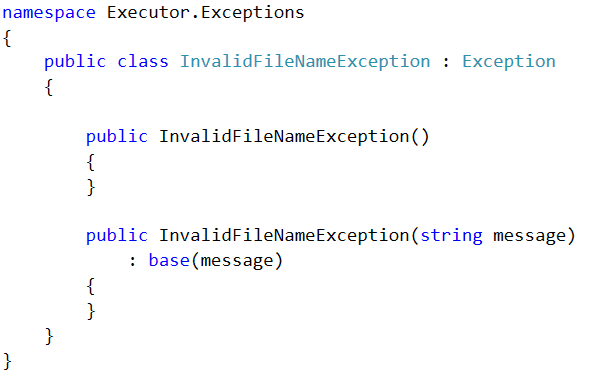
Instead of catching an ArgumentException and re-throwing it, with the corresponding message in the **CreateDirectoryInCurrentFolder** method we can make our own.



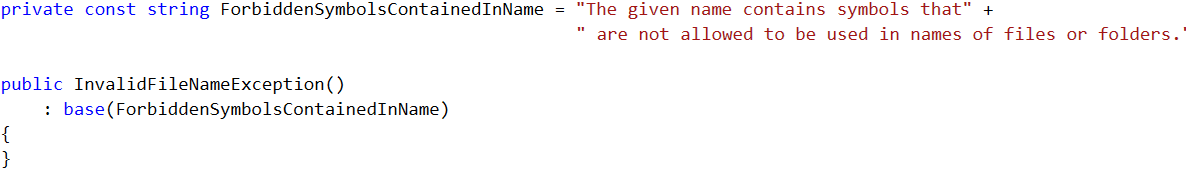
First create a folder "**Exceptions"** and a class in it called "**InvalidFileNameException"**:



Make the class **inherits the class** **Exception** and create two constructors – the first one should be parameterless and the other one with only one string parameter:

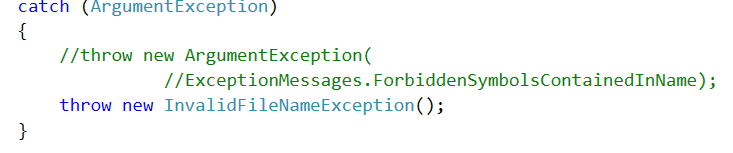


The first one should call the base class' constructor and pass it a default message. You can just use the message in the **ExceptionMessages** class. You can also move the constant field to this class so it is decoupled (independent from other classes).



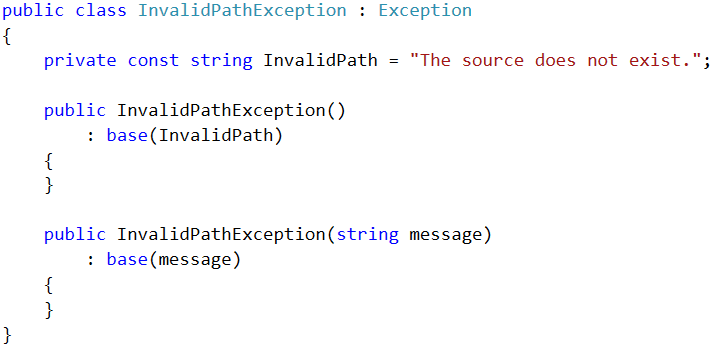
Leave the second constructor as it is. It can be used if we want to pass a different message when creating a new **InvalidFileNameException.**

Now replace the throwing of ArgumentException with our own more specific InvalidFileNameException**.**



As you can see there is no need to pass a message to the constructor because we know the default message we automatically set is just what we need.

For the **ChangeCurrentDirectoryRelative** and **ChangeCurrentDirectoryAbsolute** methods - we can make one common exception - **InvalidPathException**. First delete the **InvalidDestination** constant from the **ExceptionMessages** class. You will see only one compile time error in the **IOManager**. We will fix it by creating a common exception.

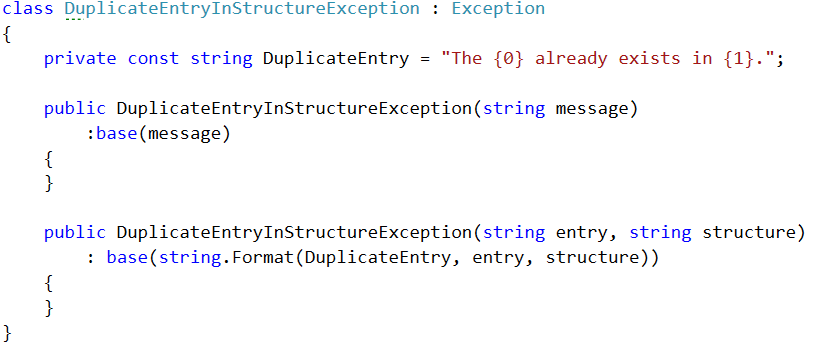


Now throw this new exception in the two methods mentioned above and **everywhere else** where we used to throw an Exception with the **InvalidPath** or **InvalidDestination** messages.

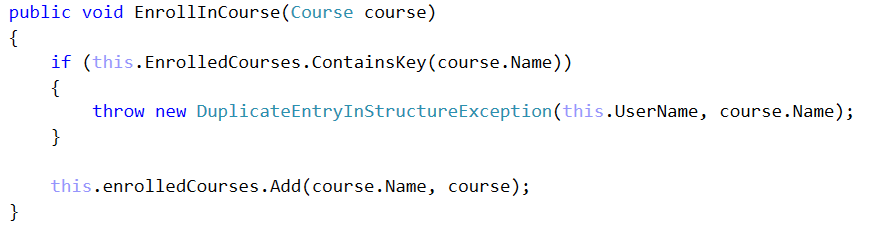
## Making Exceptions for the Models package

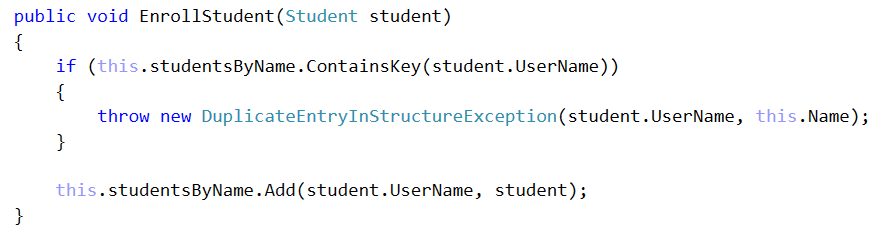
Let’s move to the **Course** class - here we can set up a more specific exception for the field **name's setter.** We can call it **InvalidStringException**. Now just repeat what we did in Problem 1 and put as default message the **NullOrEmptyValue**. We can use this exception for the **Student** class' **name setter** as well.

Next we can make an exception for the **EnrollStudent** and **EnrollInCourse** respectively in the **Course** and **Student** classes. This will be a more complex exception because it will accept 2 parameters - **entryName** and **structureName,** thus we will call it **DuplicateEntryInStructureException.** For the message use the same **StudentAlreadyEnrolledInGivenCourse,** but rename the constant to **DuplicateEntry**:



Now just throw the exception in both of the methods but be careful for the order of the parameters.





The last method we need to make an exception for this package is the **SetMarkOnCourse.** Make a **CourseNotFoundException** following the same pattern and then throw it in the correct place.

## Making Exceptions for the Repository Package

In this package the exceptions are mostly okay but you can make some of your own if you want.

Congratulations you are done with the part for **Inheritance**.

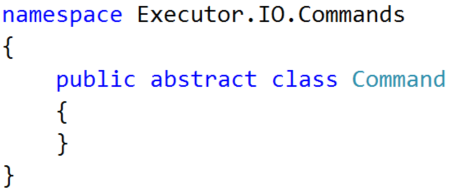
# Polymorphism

## Introduction

After learning about **Inheritance** and **Polymorphism** the time has come to do some more substantial refactoring to our project. In this lab we will employ a **design pattern** called "[**Command pattern**](https://en.wikipedia.org/wiki/Command_pattern)". As you can probably guess from the name it is mainly related to the **CommandInterpreter** and how it parses commands from user input. Our goal in the end is to provide a source code which is a lot more extensible and readable than at the moment. The whole idea of the **Command pattern** is to replace simple method calls in the **ParseCommand** method with creation of different **Command** objects. This will make the **CommandInterpreter** **much less bulky** as every command will be in a different class. Currently the **CommandInterpreter** is around 300 lines of code and is on the big side for our small project. In order to do this we need to solve several problems:

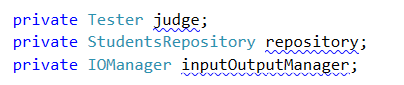
## Making an abstract class called Command

Start by making a **sub-folder** in the **IO** namespace called **Commands.** This is the place where we will store our first class **i,** which we will define **abstract.**



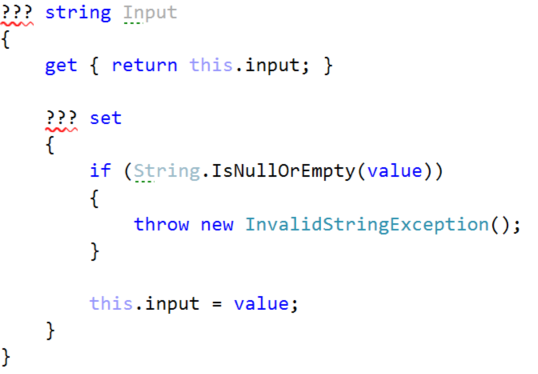
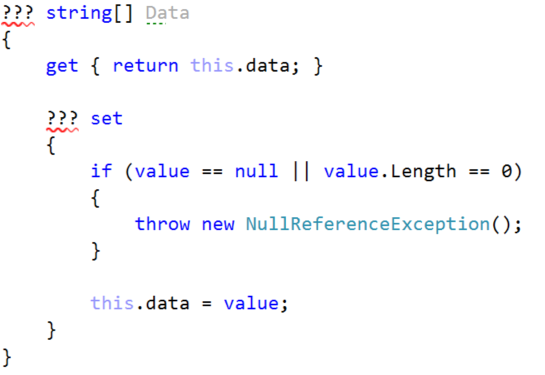
It will have **two fields** corresponding to the parameters we used to pass to each command so far – **input** and **data**:

And another **three fields** that will represent all the utility classes we have:

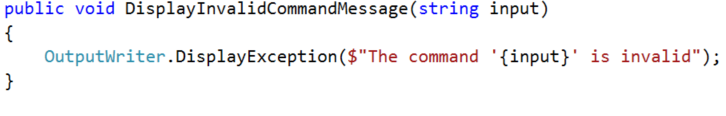


As you can see all command methods have these exact two parameters so make such fields.

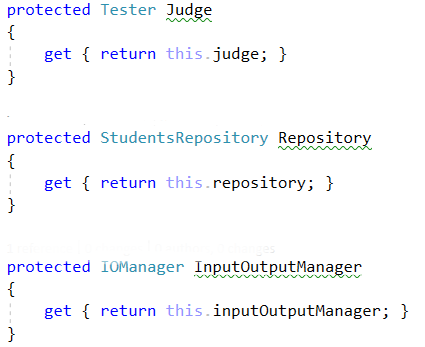
Now **encapsulate** the fields trough getters and setters. Think about the appropriate access modifiers - weather the **getters** will be used in the rest of Command classes which will **extend** our **abstract** Command and **setters** that will only be used in the current class.

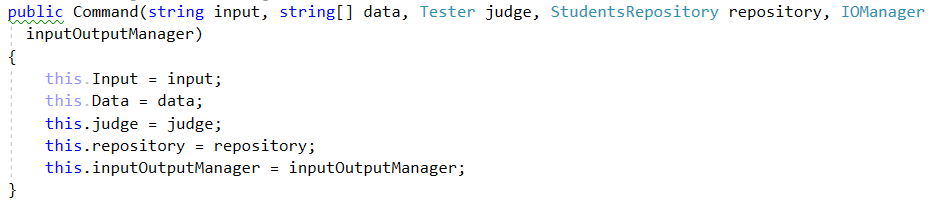
As for the **InvalidCommandException –** create one to replace the purpose of the **DisplayInvalidCommandMessage** method. It should extend the class **Exception** once again:



Also provide **getters** for the utility classes. They should be useable **only by classes that extend our abstract** class:



Don't forget to make a constructor that will set all the fields. Use the setters of the fields where possible:



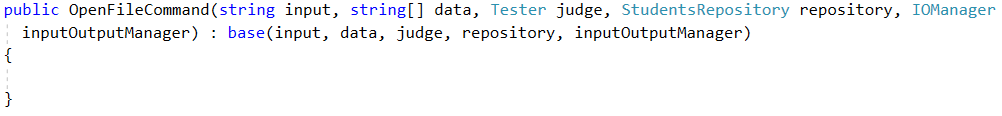
As you can see the constructor is quite big and it receives utility objects that not every command really needs. Still we are leave it like this because we need all the commands to have the same parameters in their constructors. You will find out why that is so in the **Reflection** **lab** in the **next course**, so stay tuned with **BashSoft!**

The final thing we will have in our abstract class is an abstract method called **Execute** it will be abstract because we want to force classes that inherit this one to **override** it. It will also throw and **Exception**.

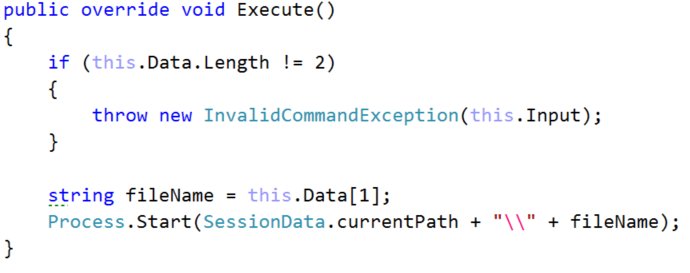


## Making the OpenFileCommand class

Create a class with the name above in the same package as our **Command** class. Add a constructor corresponding to the base class:



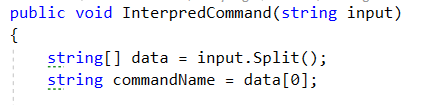
Then **override** the abstract method from the base class. Inside copy the code from the **TryOpenFile** method with a few small changes: throw a new **InvalidInputException** instead of using the **DisplayInvalidCommandMessage** method; replace data with its corresponding field getter.



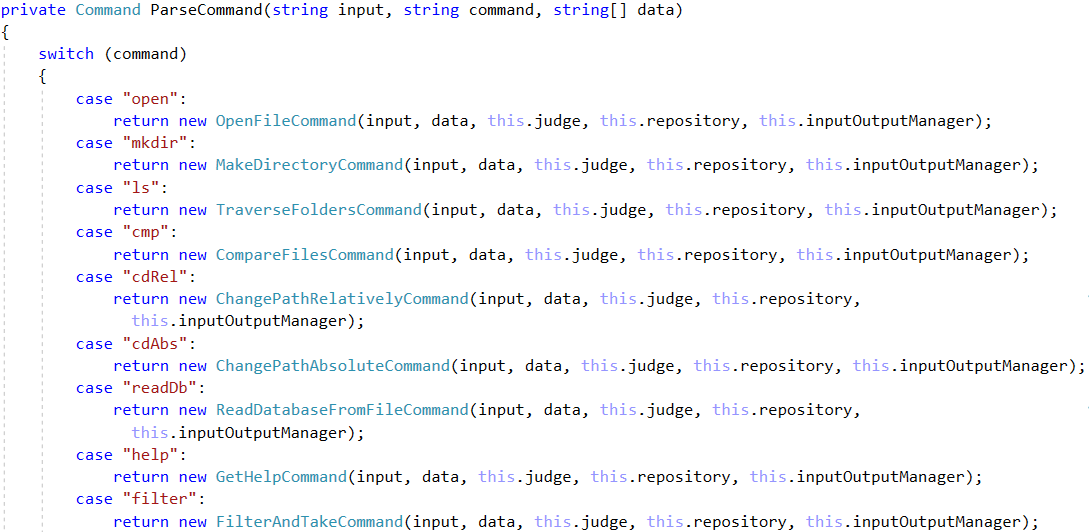
## Accommodate the CommandInterpreter to work with our new commands.

Now that we have the **OpenFileCommand** class we can make the **CommandInterpreter** use it instead of the corresponding method.

Let's start in the **InterpretCommand** method. First - rename the String **command** to **commandName,** we will need that name for the **Command** object.



Now change the return type of the **ParseCommand** method to **Command.** In the **switch-case block** instead of calling methods in every case **return** new **Commands** of the corresponding type:



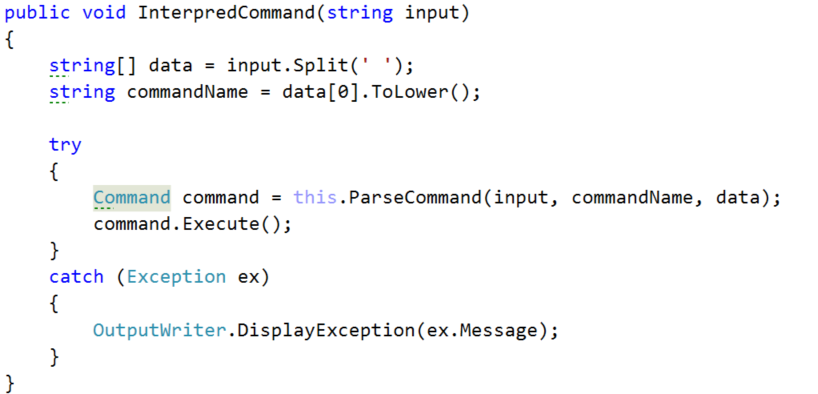
You will later create classes for all the other commands. The **default case** throws a new **InvalidCommandException**:



The command classes' names are the following:

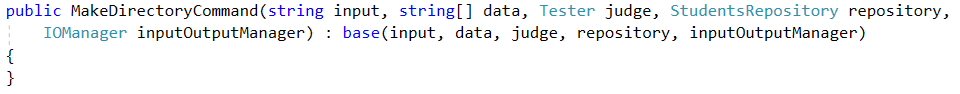
* “**open**” -> OpenFileCommand
* “**mkdir**” -> MakeDirectoryCommand
* “**ls**” -> TraverseFoldersCommand
* “**cmp**” -> CompareFilesCommand
* “**cdrel**” -> ChangeRelativePathCommand
* “**cdabs**” -> ChangeAbsolutePathCommand
* “**readdb**” -> ReadDatabaseCommand
* “**help**” -> GetHelpCommand
* “**show**” -> ShowCourseCommand
* “**filter**” -> PrintFilteredStudentsCommand
* “**order**” -> PrintOrderedStudentsCommand
* “**dropdb**” -> DropDatabaseCommand

Finally in the try block of the InterpedCommand method creat a Command object and set it to **ParseCommand,** then call the **command.Execute**() method. We can also collapse all our catch blocks to a single catch of type **Exception**. In this way we are sure that everything will be caught and we will still print the right message thus we reduce code clutter. Here is the final look of the **interpretCommand** method:

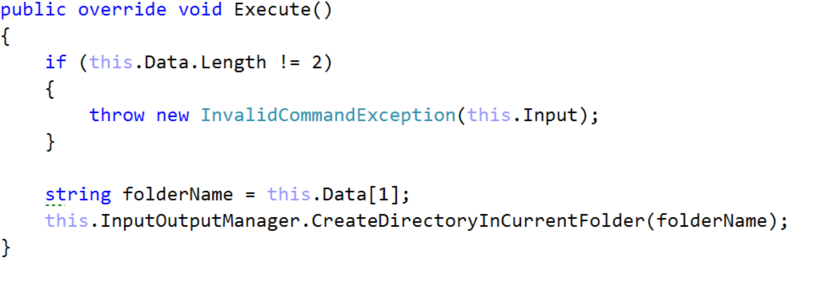


## Creating the MakeDirectoryCommand

Following the same pattern as with creating the **OpenFileCommand** make a constructor matching the base one.

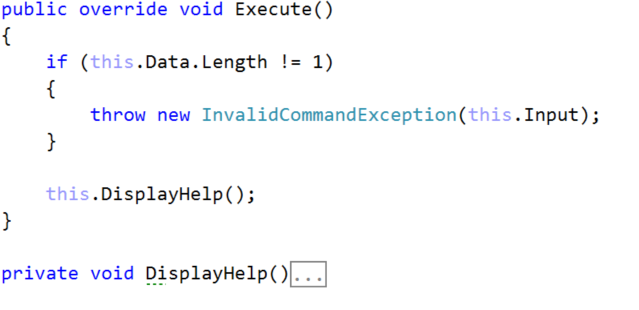


When copying the code from **TryCreateDirectory** to the **Execute** methoddon't forget to change the usage of the **inputOutputManager** field with the corresponding **getter:**



## Creating the GetHelpCommand

In order to **encapsulate** the **main logic** behind some commands as this one, we use **private helper methods**. Just copy the whole **DisplayHelp()** **method** alongside the code you move from the **TryGetHelp** method. Here is how it looks in the **GetHelpCommand,** you must do the others by yourself:



The other Commands that have such helper methods are: **PrintFilteredStudentsCommand** and **PrintOrderedStudentsCommand.**

## Finish all the other Command classes by yourself

We went through the basic logic behind the transition from a **Try<DoSomething>** method to a **<DoSomthing>Command** class. Now it is your turn - finish refactoring the other commands. Once you are done you can delete (or comment if you prefer) all the methods in the **CommandInterpreter** except **InterpretCommand** and P**arseCommand.**

Before we consider the job done, do some testing in order to confirm all the changes. If everything works as previously - congratulations you have completed the part for **Polymorphism!**