# Lab: Interfaces and Abstraction

This document defines the lab overview for the ["C# OOP Advanced" course @ Software University](https://softuni.bg/trainings/1637/c-sharp-oop-advanced-july-2017). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/).

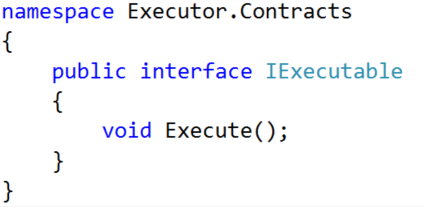
# Introduction

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

# Creating Interfaces for the IO package

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

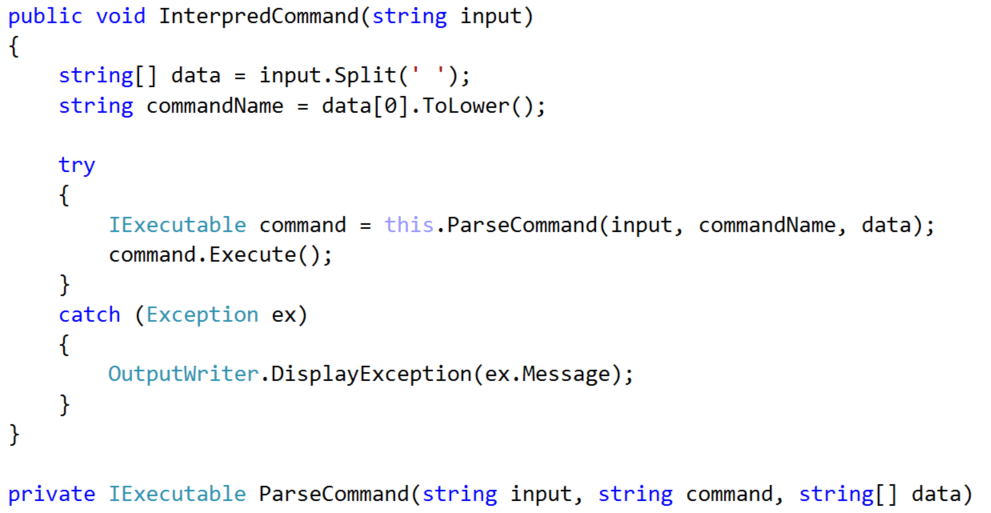
Let's start with creating our first **interface** that will define what a command should have. Create a new package called "**Contracts"** and inside of it - an **interface** called "". Then define the proper method signature for the execute method our commands have:



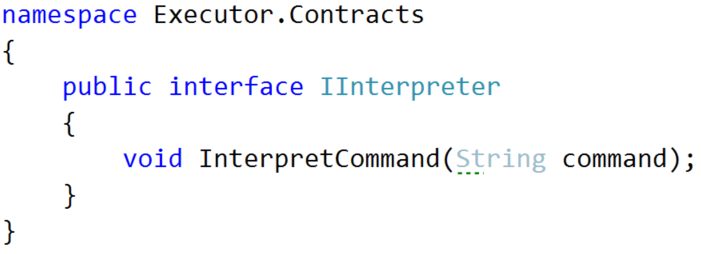
Now implement the method in our abstract **Command** class:



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

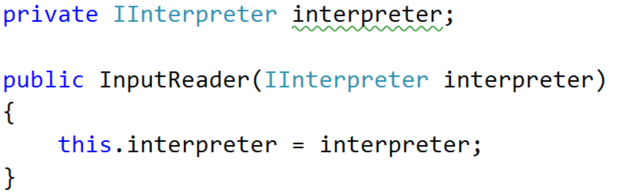


Our next **Interface** will be for the **CommandInterpreter**. You can call it "**IInterpreter**" and define in it the public method of our class:



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

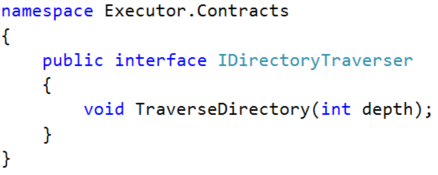
In the **InputReader:**



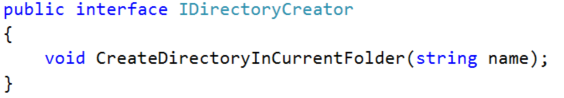
In the **main** method:



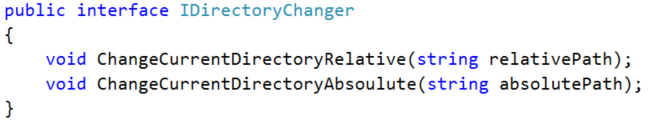
Next up is the **IOManager -** it is a good place to employ some **interface segregation.** First create an **interface** called **IDirectoryTraverser** - it will hold the **TraverseDirectory** method:



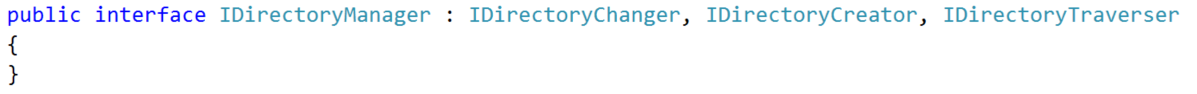
Create an **interface** called **IDirectoryCreator** for the **CreateDirectoryInCurrentFolder** method:



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



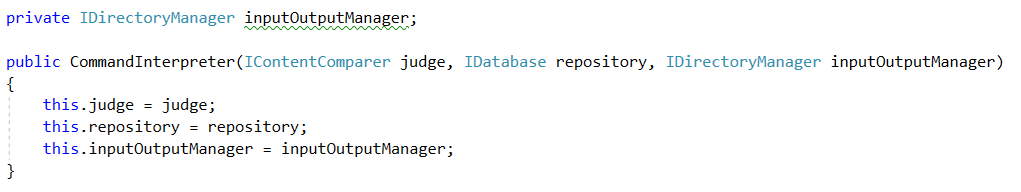
Creating these three **interfaces** would allow us to oblige different classes to implement **only** the interfaces that are **related** to their actions. However, in the case of our **IOManager** we will need all three of them, so we will combine the three interfaces in a single one:

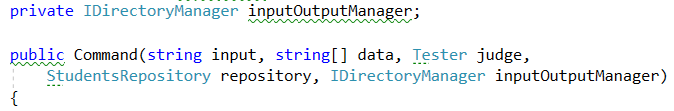


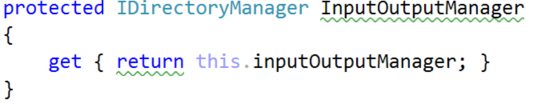
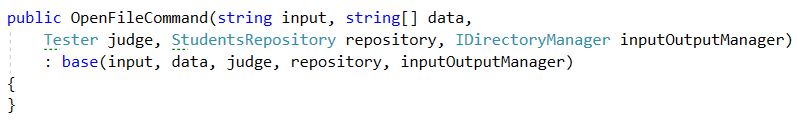
Implement it like this:



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





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

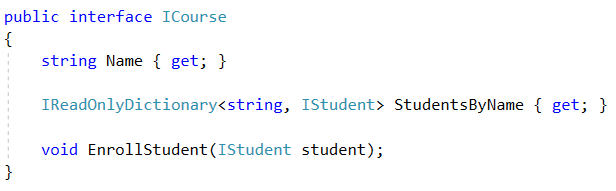
Make an interface for the **InputReader** by yourself. You can call it **IReader** with method **StartReadingCommands.** Don't forget to implement it and replace **InputReader** with its interface where possible… (psst - the main method).

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

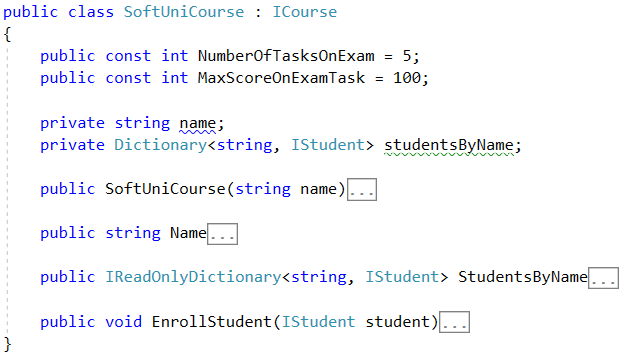
# Creating Interfaces for the Models package

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

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



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



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

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



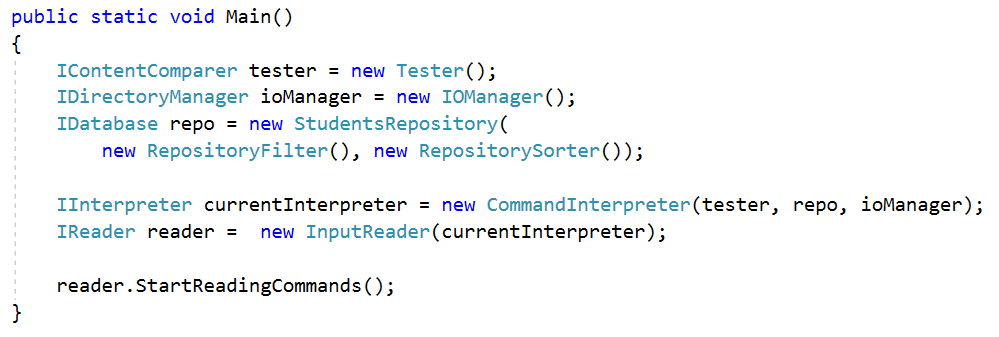
It is a good idea to make sure that the code compiles before proceeding to the final task.

# Creating the remaining Interfaces

Create the following interfaces by yourself:

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

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



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