# Lab: Encapsulation

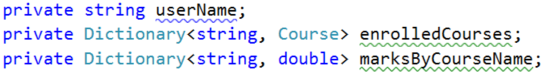
This document defines the lab overview for the ["C# OOP Basics" course @ Software University](https://softuni.bg/trainings/1636/c-sharp-oop-basics-june-2017). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/trainings/1373/c-sharp-basics-oop-june-2016).

# Introduction

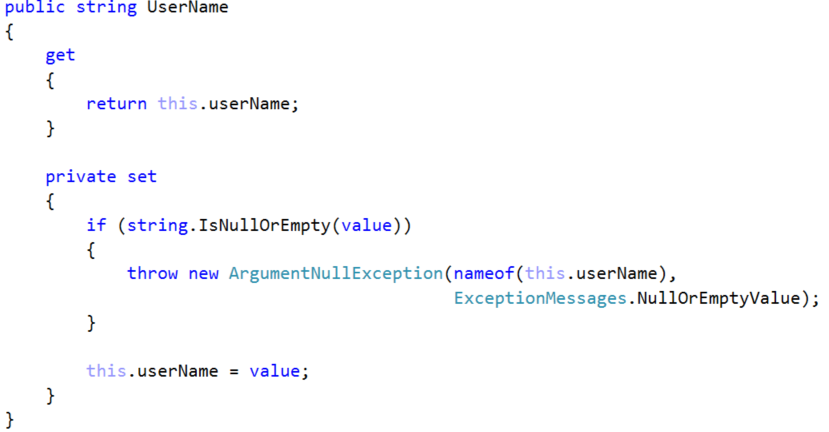
We don’t know if you’ve noticed, but we’ve pretty much used **encapsulation** and **defensive programming** a lot so far. For that reason, there are going to be only a few changes that we’ll need to make here in the current BashSoft lab. Until now we’ve used many access modifiers even though you haven’t 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.

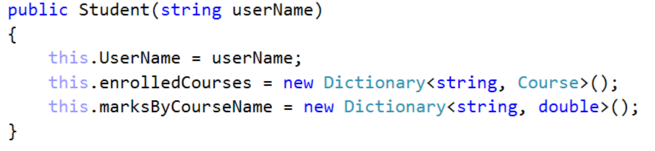


Then **encapsulate** them through **getters** and **setters**. You can put validation in the **setter**:



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**.



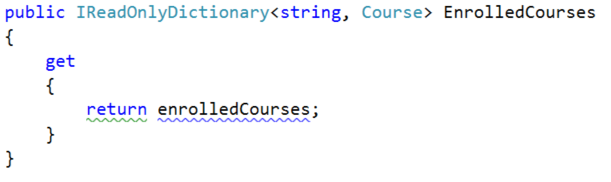
Now everywhere where you used the public field for getting the userName (studentName) you will get errors because it is now private. **Fix that by using the getter instead.**

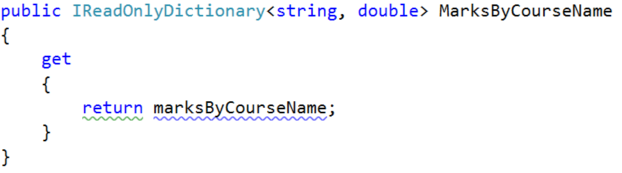
In order to **encapsulate** the **Dictionaries,** we will make **only** **getters** (**read only** properties). There is a problem though. Someone from outside of this class may not be able to set the dictionary to a new one but they could modify its internal contents. We can avoid that in two ways:

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.

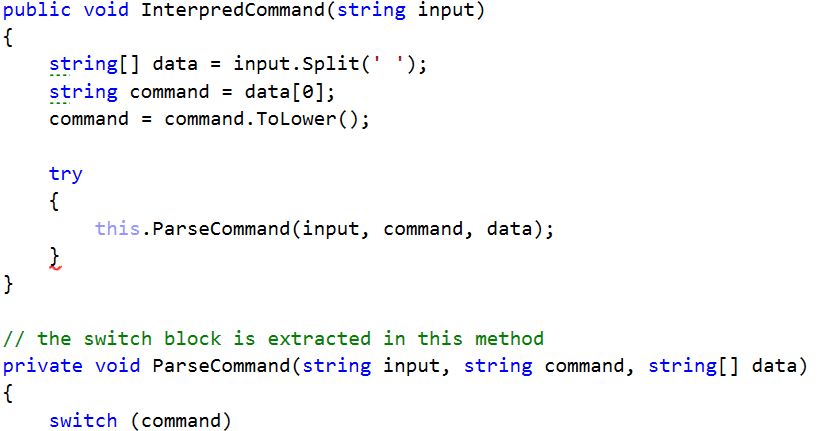
# Encapsulating fields in the Course class

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.

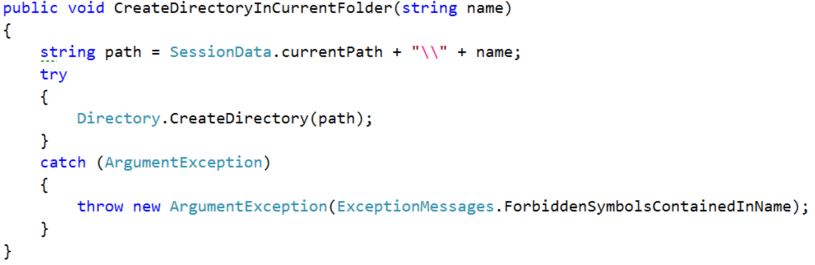
Let's start with the **CommandInterpreter** itself. In order to get it ready for all the exceptions we are going to throw at it we need to surround the switch statement in the **InterpretCommand** method with a try block. But first extract the whole switch block in a new private method, so the code is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



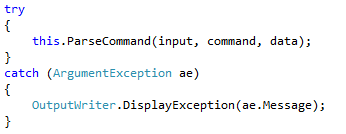
# Throwing exceptions in the IOManager

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.**

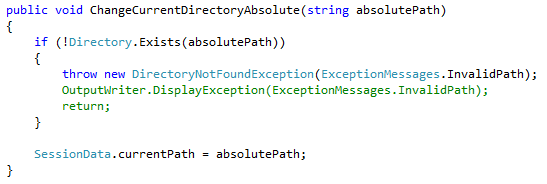
Next up is **CreateDirectoryInCurrentFolder** here we can **re-throw** the argument exception by catching it and throwing a new one:



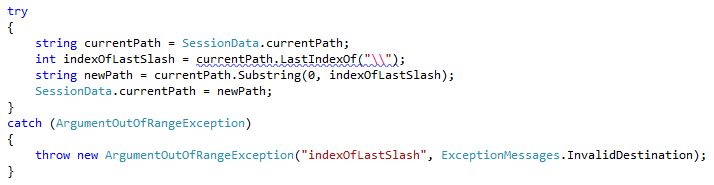
Now that we've thrown an exception we must return to the **CommandInterpreter** and add a **catch block** after the **try block** we made there:



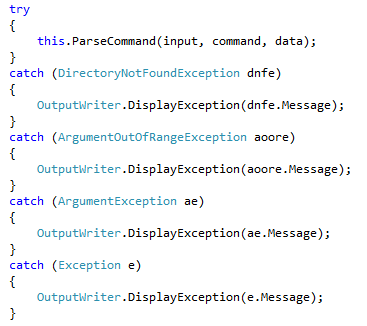
Next is the **ChangeCurrentDirAbsolute** here we can **throw** an **IOException** with the same message that we used to display directly:



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



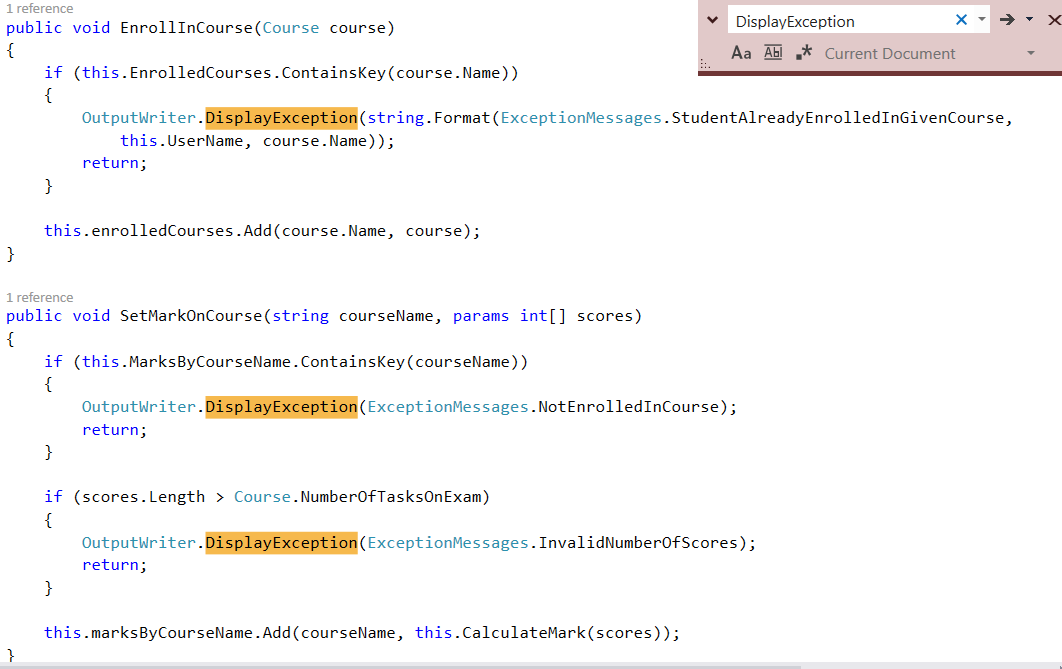
We are done with this class, so it's time to return to the **CommandInterpreter** and catch everything in the correct order:



# 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

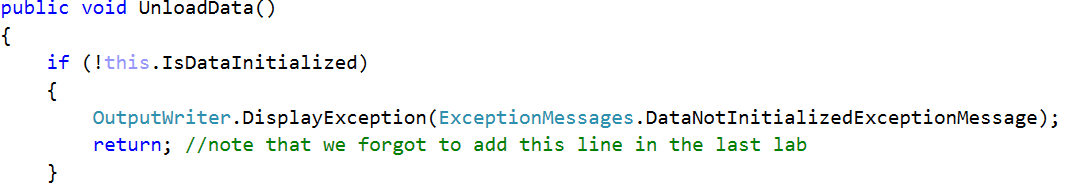
Use the find tool in **Visual Studio** to find every **DisplayException** inthe **Student** and **Course** classes and replace it with a corresponding new exception throw. Carefully consider what exception to throw (use google if you can't think of the right exception fast). Don't forget to pass the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

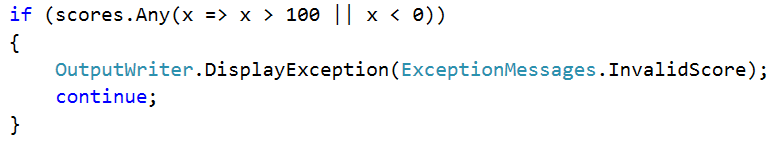


To these:



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 didn't break 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** lab exercise!