# Lab: Reflection & Annotations

This document defines the lab overview for the ["Java OOP Advanced" course @ Software University](https://softuni.bg/java-advanced-oop). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/trainings/1376/java-advanced-oop-july-2016).

# Introduction

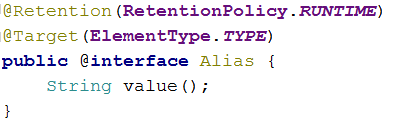
In the current lab piece we are going to make the command pattern to be implemented using the reflection API from Java. That way we are going to create instances of the commands without the usage of a switch. This makes our application more flexible, because using it, will allow us to make a new child of the Command and implement it without changing any existing classes.

## Making the Alias Annotation

Since we are not going to use the switch in our command interpreter, we will need to figure out another way to choose which command to create. This can be done by creating custom attributes and then scanning those attributes and depending on what value we’ve put in the constructor of the attribute, we will choose which our desired class to instantiate is.

We suggest you first create a new package called **annotations**.

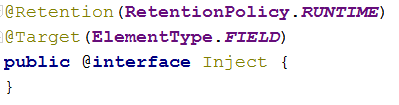
The first annotation we will create is called Alias. It will have one parameter (value). We also need to set the meta annotations for it, so it can only be applied on classes and is accessible at runtime.



## Making the Inject Annotation

This annotation will point us to the field we will set using reflection. Now we are passing all the “managers/comparers/repos” in the constructor of the chosen command, but some of the user’s wishes have changed and since we are “[Agile](https://bg.wikipedia.org/wiki/%D0%93%D1%8A%D0%B2%D0%BA%D0%B0%D0%B2%D0%B0_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F)”, we will follow the holy user’s wishes and modify the desired functionality. Also note that the pass all utility classes to all commands works for now but if we add more utility classes like say a PeshoManager we will need to add it to all the constructors. Such code is not easy to maintain and extend so we need to fix it. It also causes some commands to know about stuff they don't need nor use.

But first let's create the Inject annotation.



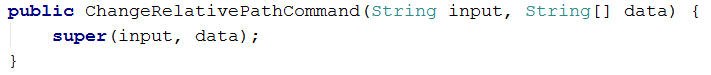
As you can see, we do not have anything in the body of the annotation. That’s because we want to use it only as a notification that the field below it needs to be injected with a value from the command interpreter. You've probably noticed that we’ve made it to be usable only on fields, and that is absolutely on purpose.

## Changing the commands to fit the new approach for instantiation

Here are the three steps you need to do for all the commands in the application:

1. Delete everything from the constructor except input and data
2. Apply alias over the class by passing to the annotation the string that is the actual command that you read from the console.
3. Leave only the fields that are used in the current scope of the class and put an inject attribute above them.

Here is how this should look:

1. 
2. 
3. 

And finally we are using this field in order to execute the current command:

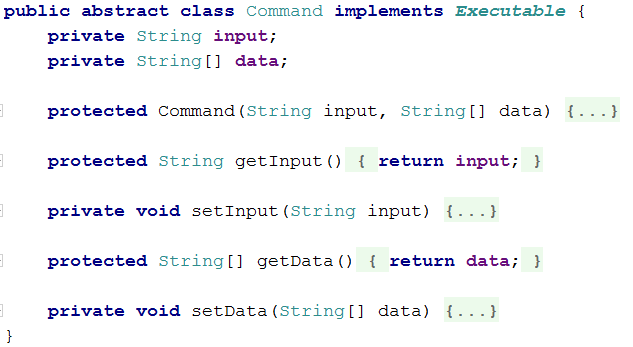
### 

Do all these steps for all commands. Note that there are some commands like the getHelp command that doesn't need any utility class as a field.

## Modify abstract command

Since you are not going to use the fields except the input and the data[], you can remove the fields, constructor parameters and properties for the others.

Here is how your class should look after you remove all the clutter:



## Modifying the Command Interpreter:

Now it’s time to make some changes in the command interpreter, to start creating the desired command using reflection and not the switch-case.

First thing you can do is delete the whole switch.

Now let’s start preparing for the creation of the objects. First thing you need to make in the **parseCommand** method is create a file object which will be the folder (package) where all our commands are. We need this because we will have to traverse the folder with IO operations in order to get all the command classes we need:

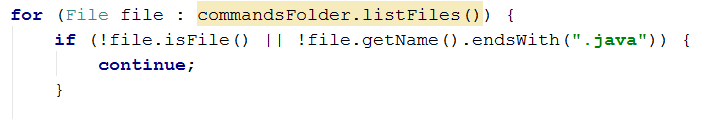


The **COMMANDS\_LOCATION** constant should hold a relative path to our project: "**src/bg/softuni/io/commands**"

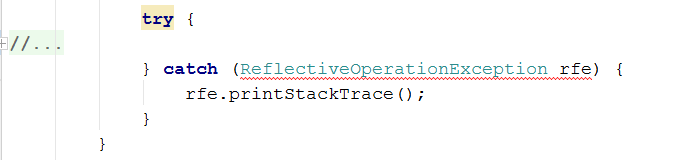
Next - declare an Executable and set it to null.



Now we need to iterate trough the files in the **commandsFolder** in order to find the class we need. In order to find only classes we need to not check files that don't end with ".java" or are not files at all:



Now make a try catch block. All our reflection code will be in the try, and because various reflective exceptions might be thrown in it we will catch a **ReflectioveOperationException**. Note that we will directly use **printStackTrace** if such an exception is caught, because it isn't meant for user info, but more for ours.



In the try block we can start with getting the name of the current class. We can extract that from the name of the current file, like so:

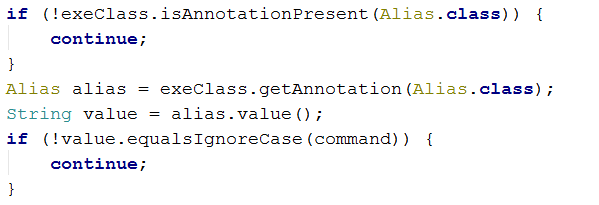


The substring is because we don't need the extension of the file for our class name. Now we are ready to create a Class object:

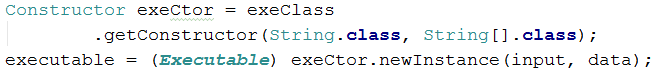


We are again using a constant with value: **"bg.softuni.io.commands."** This may seem not seem like it's the best implementation but it promotes consistency as it forces anyone who extends our code - writes a new command to put it in the correct package with the other commands.

Once we have the class we can check if it is the one we need. First check if there is a present Alias annotation and if it's value is the same as the one we need currently.



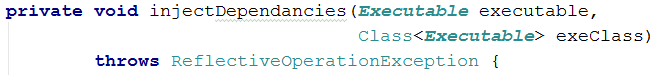
Notice that again we do the opposite checks to what we need and we put a continue statement if they are true. This approach allows our code to not get indented like mad.. we avoid the so called [hadouken code](https://i.imgur.com/BtjZedW.jpg). Once we've done these checks we're finally sure we got the class we need and can proceed to getting its constructor and using it to instantiate our executable object:



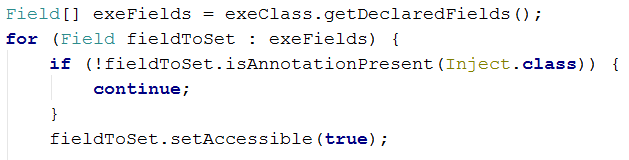
Now all that is left is for us to inject the dependency of the current wanted command. This is the field we marked with **@Inject**. This however will not be a small task so we can extract it into a separate private helper method:



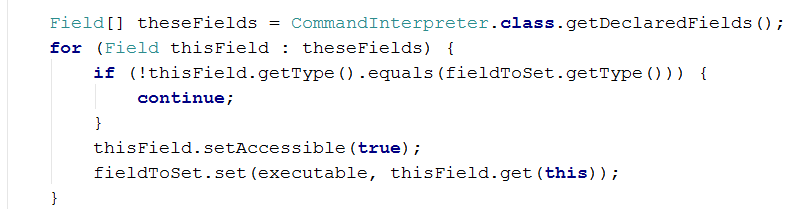
It will do some reflection too so make it through a **ReflectiveOperationException.**



Inside the method we will start by getting all declared fields of the class, iterate trough them and check if the **@Inject** annotation is present. If it is we can set the field to be accessible from the outside:



Once that is done we can get the fields of the CommandInterpreter itself, iterate trough them and if the field to set has the same type as one of the interpreters fields we can set the first ones value to the second ones.



Now we are done with all the refactoring and if you start the program, everything should be working as before. However, now if we need to implement a new command, all we have to do is inherit the abstract command class and set the corresponding attributes, but we do not need to change or add anything to the command interpreter. Not to mention this code is way cooler and shorter than the switch. However you should know that know we've reduced the performance, but this is a tradeoff we can afford.