# SoftUniDI Framework

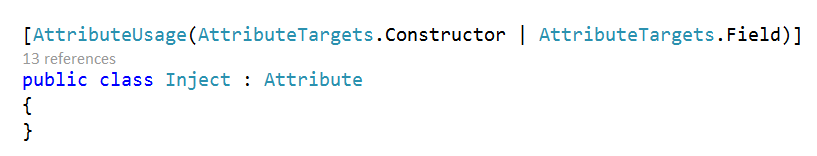


In this workshop we are going to implement our own dependency injection framework. This will help us to develop loosely coupled code. In our dependency injection framework, a dependent object is provided with the object it needs at runtime. The provided object will satisfy the dependency during program execution but would not be known at compile time. Rather than directly instantiating dependencies, or using static references, the objects a class needs in order to perform its actions are provided to the class in some abstracted form. Our framework will be very similar to [Google Guice](https://en.wikipedia.org/wiki/Google_Guice).

Create new **library .net core** project and add **Attributes**, **Injectors** and **Modules** folders.

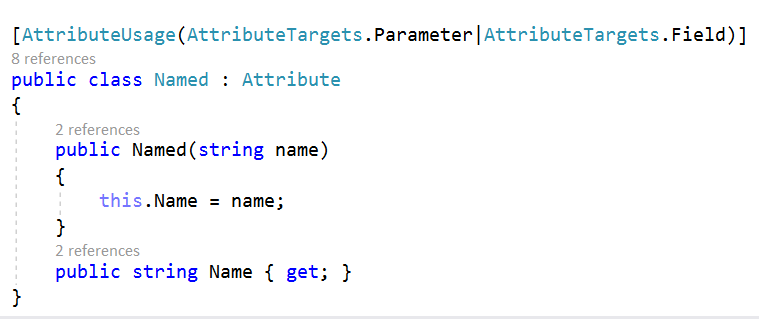
## Inject Attribute

First we need to create an attribute class **Inject** (in **Attributes** folder) which we will use to inject constructors or fields. Our attribute must inject only fields and constructors. This can happen by setting its attribute targets to constructor and field. Our inject class must inherit class Attribute.



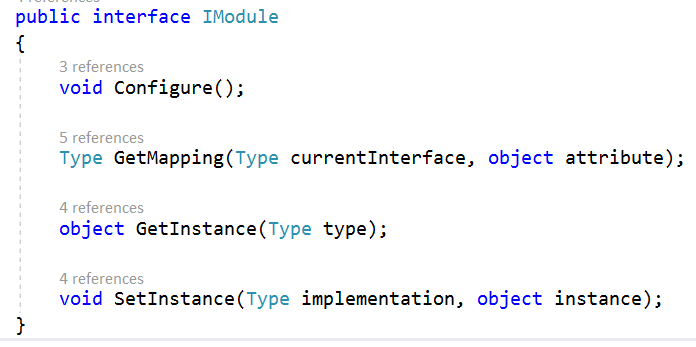
## Named Attribute

Now create new attribute and set its name to "Named". This attribute will be useful when we have classes which inherit the same Interface but execute different type of logic. This attribute could be used only for parameters in constructor and fields. It should have name property which will be the class name we want to inject with same interface as other class.

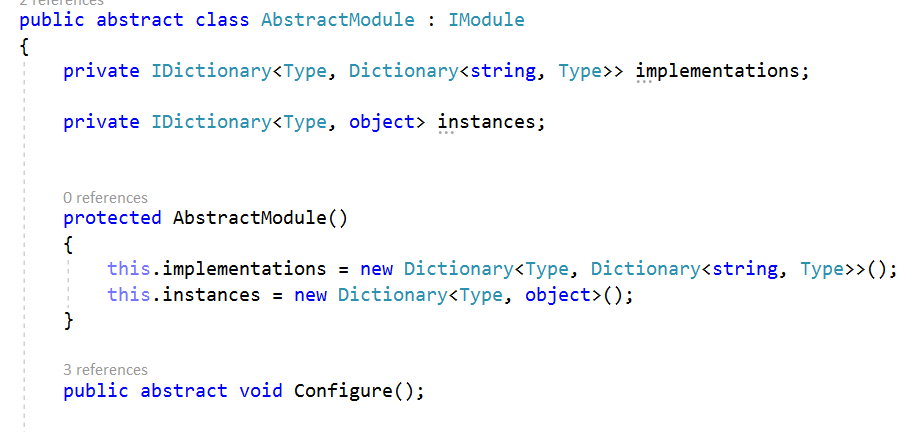


## Abstract Module

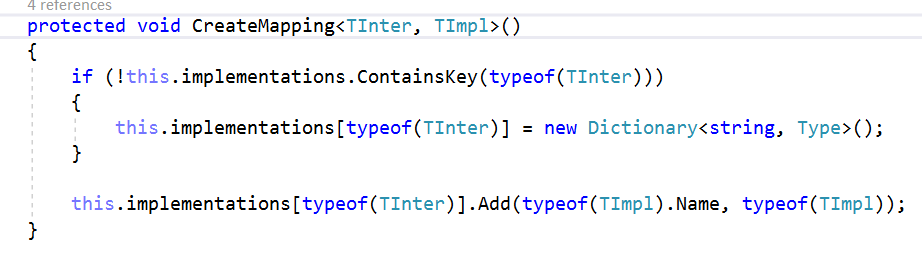
Before creating the AbsractModule class, we need an interface. Create an IModule interface with 4 methods to implement. The first method is void and is used for configuring the relation between the interface and the class which inherits it, that’s why this method we can call Configure(). When you are ready with the framework you must always configure the interfaces with their classes used for your test project. Next method is GetMapping(). It accepts the interface we want to inject and the attribute its field or parameter has (the attribute can either be Inject or Named). The method returns the class which inherits the current interface. Another method is GetInstance() which tries to get the instance of the current class. It returns the instance or null if that instance doesn’t exist. The final method is SetInstance() which accepts the current class and its instance as parameter. You should already know what this method will do.



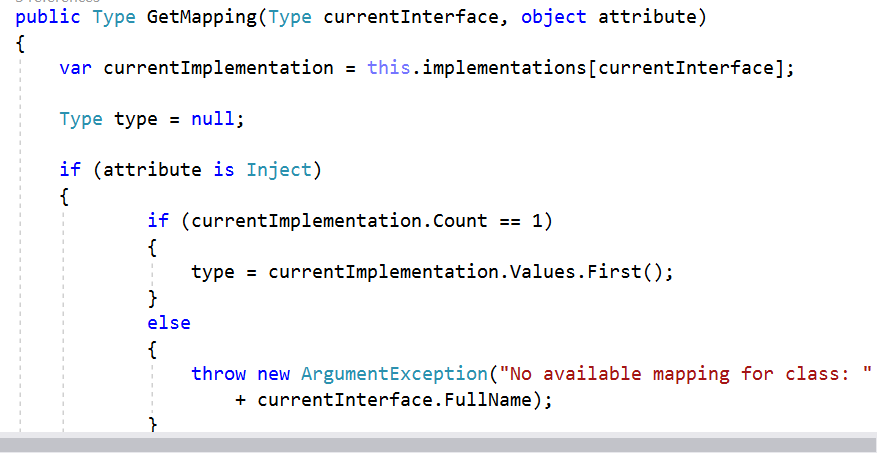
Not it’s time to create the AbstractModule class. The class should have two fields. The first one will be of type IDictionary<Type, Dictionary<string, Type>> and will take care of all the implementations we need to inject. The second one will be of type IDictionary<Type, object> and will take care for the instances. The AbstractModule constructor should initialize this two fields.



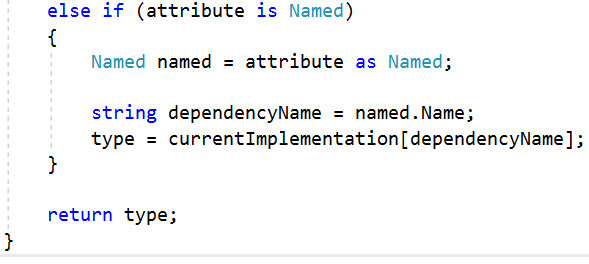
Now we need to create a protected CreateMapping() method which receives an interface and corresponding class as parameters. All this method does is to check if our implementations contain the current interface and if it doesn’t to initialize it. And in both cases to add the class name as key and the class itself as value. This method is protected because we have to use it when we test our framework and create all the mapping between the interfaces and classes we need in our application.



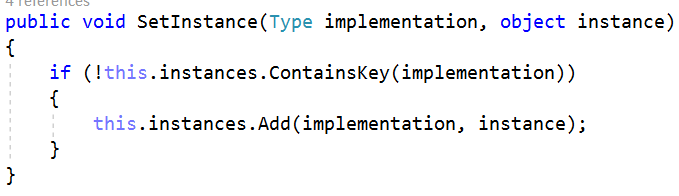
The GetMapping() method gets the current implementation using the interface parameter. After that checks for the type of attribute parameter. If the type is Inject and the implementation is only one the type it has to return is the first one implementation from the field. If there isn’t any implementation throw and exception with the message "No available mapping for class: {class name}"



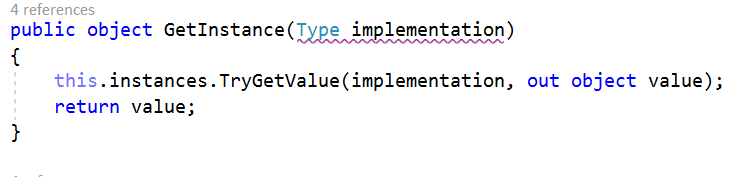
If the attribute is Named we should get its dependency name and return the type to the one with that name.



The SetInstance() method checks if our field instances contains the current implementation and if it doesn’t it adds it together with its corresponding instance.

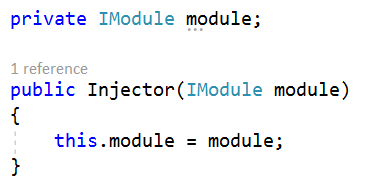


The final method is GetInstance() which tries to get the value of the current implementation and returns the instance or if it doesn’t exist it just returns null.

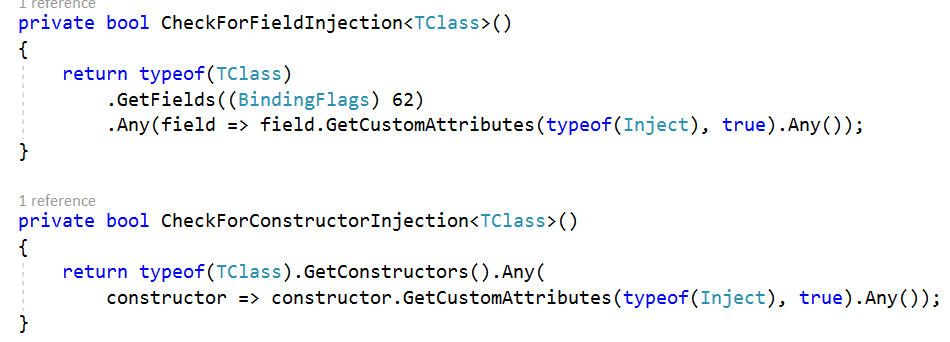


## Injector

The Injector class has field of type IModule and in its constructor takes a module as parameter.

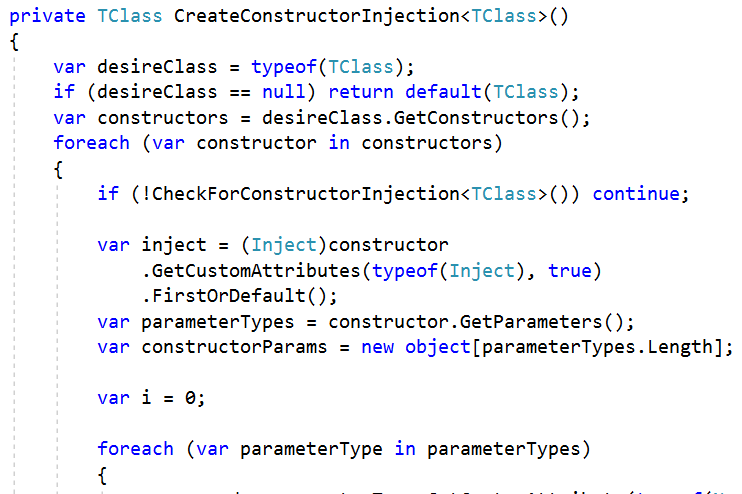


We have to check for constructor and field injection. This can happen by creating two private methods which will execute almost the same logic. We have to find the type of the current class and check If there is any Inject attribute in the constructor or in the field.

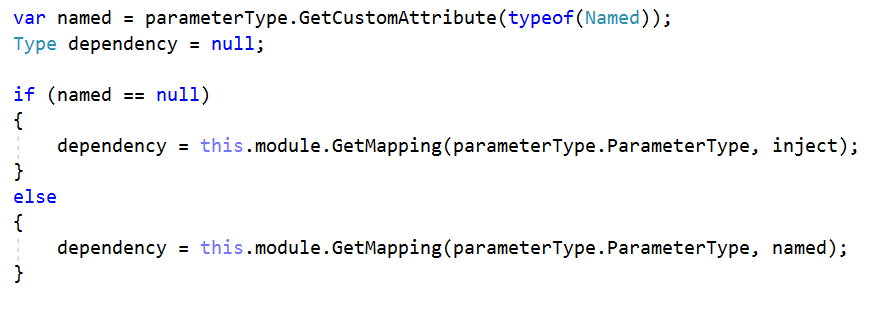


As you can see there is something new in this code and this is the BindingFlags 62. With this part of code we get all the fields apart from their access modifiers.

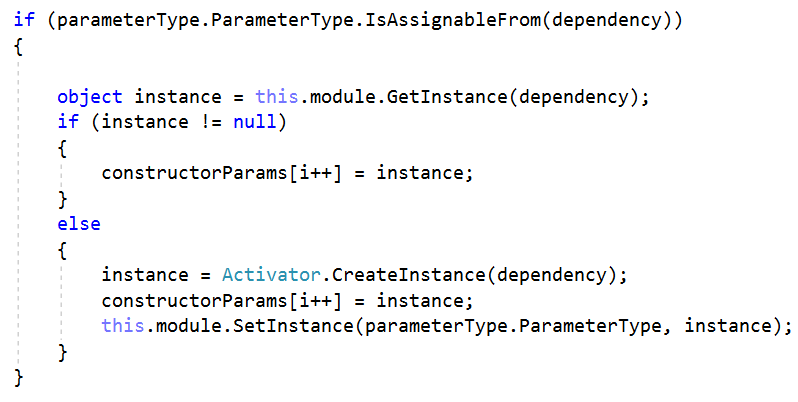
Now lets create a method for constructor injection. Get the type of the current class, if the type is null just return the default value of the generic type. After getting the type of the class, find its constructors. Check each of the constructors for constructor injection and when you find such, get its parameters.



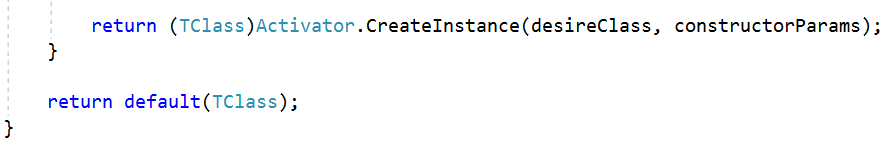
Based on the type of the attribute the parameter has get the module mapping as a dependency either with inject or named attribute as second parameter. As you can see there is an object which has the constructor parameters length. In this constructorParams object we will collect all the constructor parameters we need.



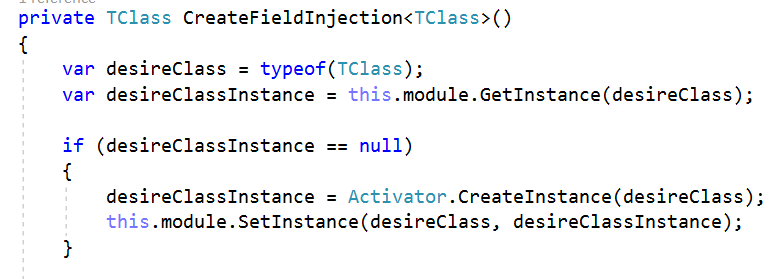
Now check if the current parameter type is assignable from the dependency type. Then get the instance of this dependency from this module. If there is no such instance create it and set it to the module.



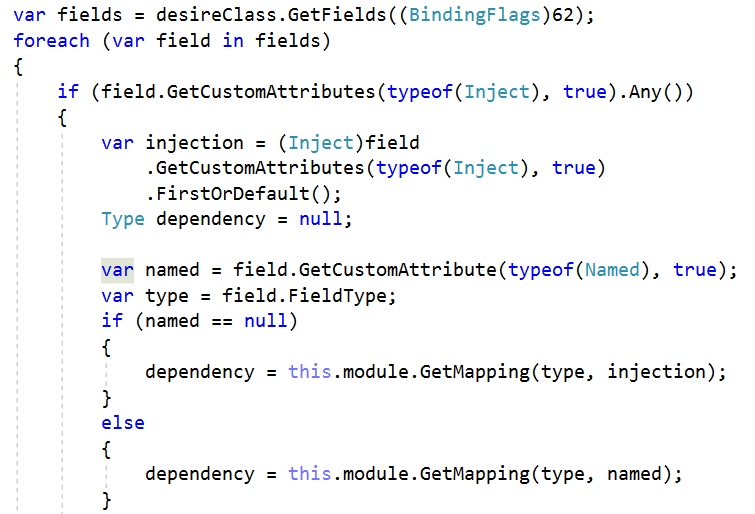
Return the instance of the desired class with the constructor parameters



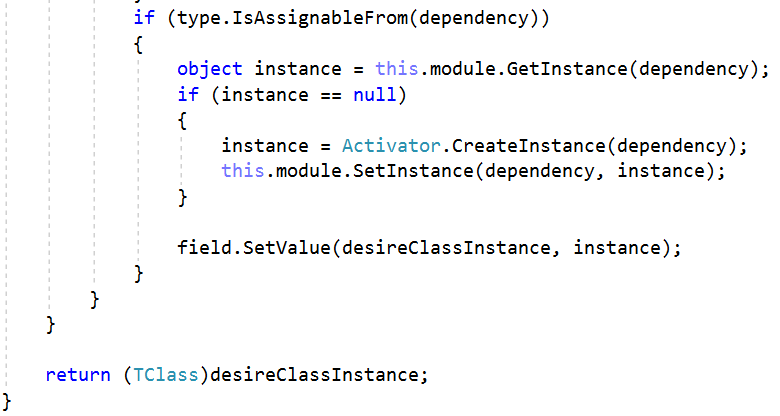
The **CreateFieldsInjection<TClass>()** method does almost the same thing but for fields. It gets the type of the **TClass**, then gets its instance from the module. If there is no such, create it and set it to the module.



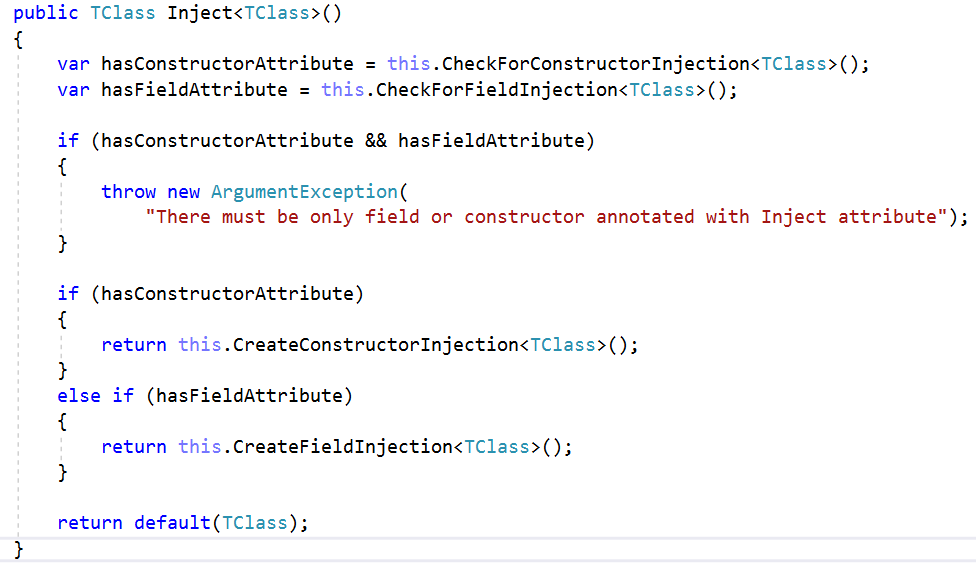
Check the current field for inject attribute or for named one. If the attribute is inject, get the mapping between the field type and the inject attribute. In the other case do the same but with the named attribute as second parameter for GetMapping() method.



Finally check if the field type is assignable from the dependency and then get the instance for this module. If there is no such create it and set it. In both cases set the instance as value of the field. Return the desired class instance.

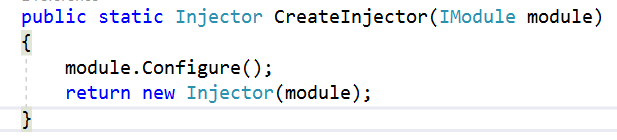


The last method in this class is for injection either for constructor or for field. This method checks for field or constructor injection. If there are both of them the method throws exception with the message "There must be only field or constructor annotated with Inject attribute". If the injection is for field, call the CreateFieldInjection<TClass>() method. In the other case call theCreateConstructorInjection<TClass>(). The method returns the default value of TClass object.



## DependencyInjector

The DependencyInjector class creates only the injection for the module provided. It has one static method which takes IModule as parameter. The method calls the module configuration and returns new Injector with the configured module.



*Source code:* [*https://github.com/A4B5TQ/TinyDIFramework*](https://github.com/A4B5TQ/TinyDIFramework)