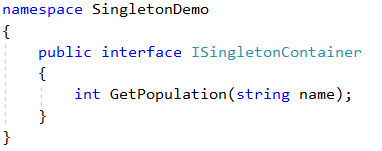
# Lab: Design Patterns

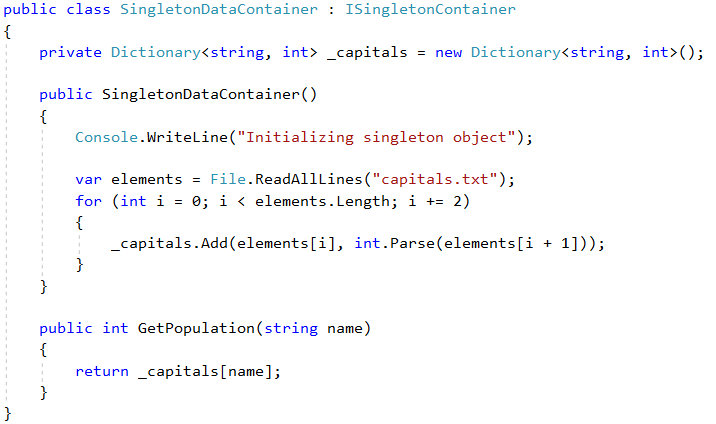
Problems for exercise and homework for the ["C# OOP" course @ SoftUni"](https://softuni.bg/trainings/3008/csharp-oop-october-2020).

## 1. Singleton

We are going to create a simple console application in which we are going to read all the data from a file (which consist of cities with their population) and then use that data. So, to start off, let’s create a single interface:

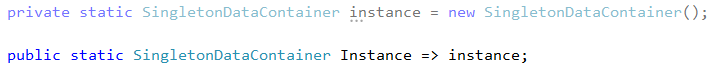


After that, we have to create a class to implement the ISingletonContainer interface. We are going to call it SingletonDataContainer:

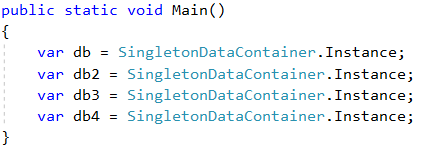


So, we have a dictionary in which we store the capital names and their population from our file. As we can see, we are reading from a file in our constructor. And that is all good. Now we are ready to use this class in any consumer by simply instantiating it. But is this really what we need to do, to instantiate the class which reads from a file which never changes (in this particular project. Population of the cities is changing daily). Of course not, so obviously using a Singleton pattern would be very useful here. Let’s implement it:

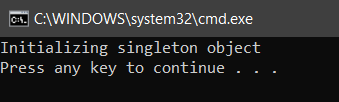
First, we will hide the constructor from the consumer classes by making it private. Then, we’ve created a single instance of our class and exposed it through the Instance property.



At this point, we can call the Instance property as many times as we want, but our object is going to be instantiated only once and shared for every other call. Check it for yourself:

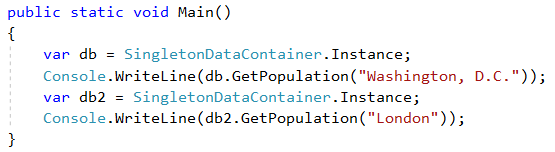


The result in out console will be the following:

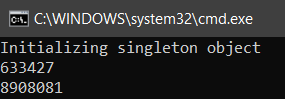


We can see that we are calling our instance four times but it is initialized only once, which is exactly what we want.

Let’s check if our console program works:



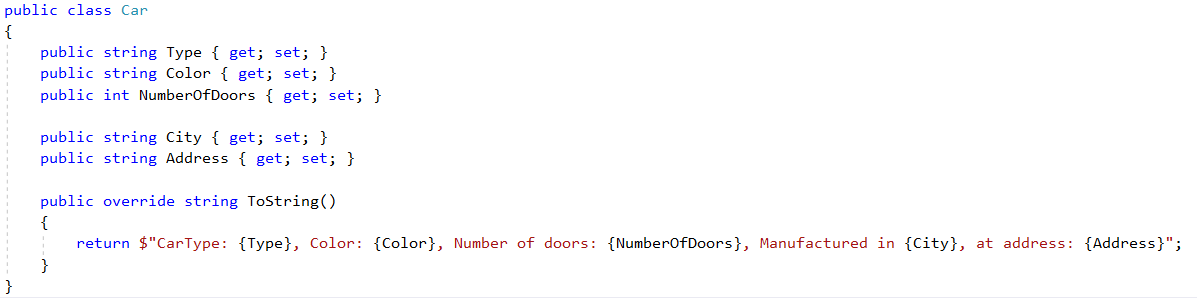
The expected output should be something like this:



## Facade

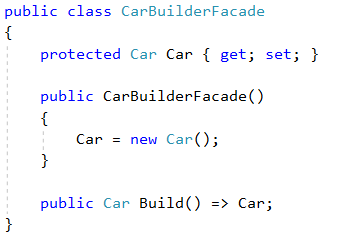
Now we will take a look at a Façade example implementation.

We will start off by creating a class to work with:



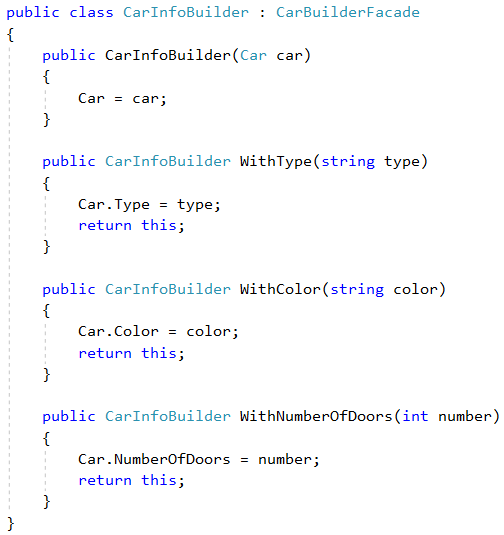
We have the info part and the address part of our object, so we are going to use two builders to create this whole object.

We need a façade, let’s create one:



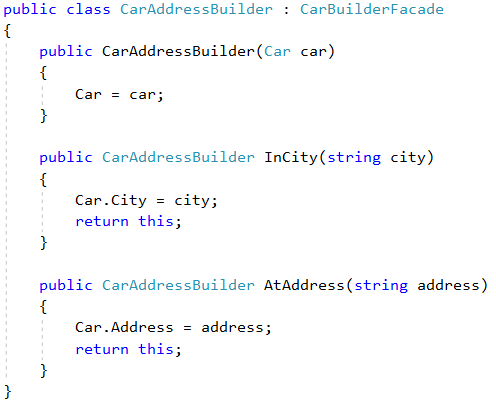
We instantiate the **Car** object, which we want to build and expose it through the Build method.

What we need now is to create concrete builders. So, let’s start with the **CarInfoBuilder** which needs to inherit from the facade class:

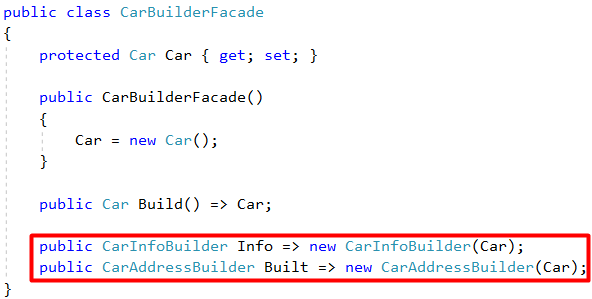


We receive, through the constructor, an object we want to build and use the fluent interface for building purpose.

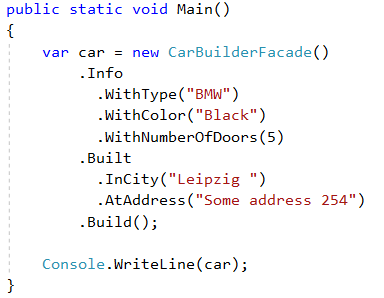
Let’s do the same for the **CarAddresBuilder** class:



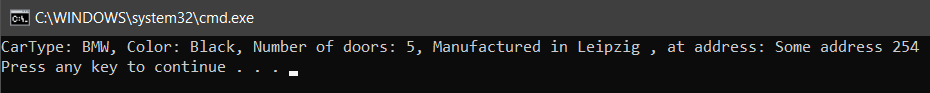
At this moment we have both builder classes, but we can’t start building our object yet because we haven’t exposed our builders inside the facade class. Well, let’s do that:



That’s it, we can start building our object:



And the output should be:

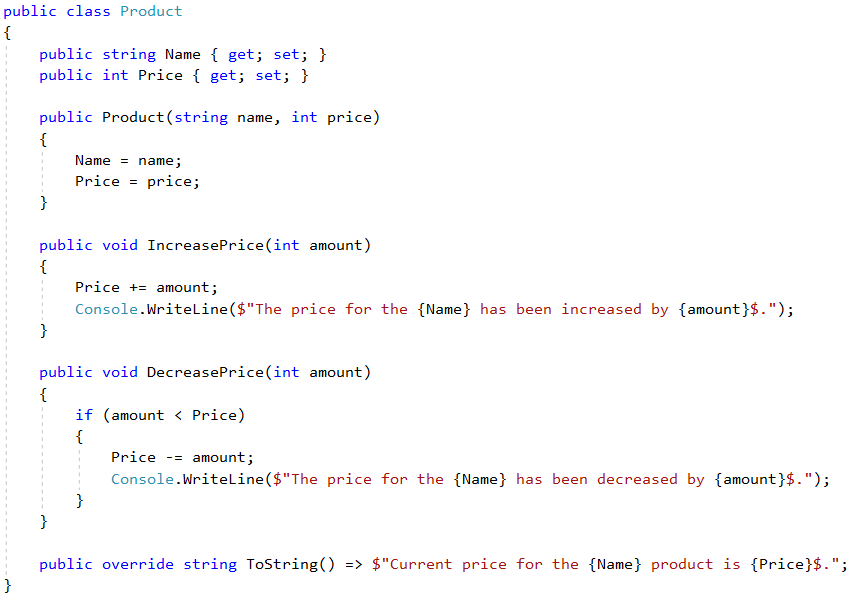


## Command Pattern

The Command design pattern consists of the Invoker class, Command class/interface, Concrete command classes and the Receiver class.  Having that in mind, in our example, we are going to follow the same design structure.

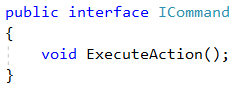
So, what we are going to do is write a simple app in which we are going to modify the price of the product that will implement the Command design pattern.

That being said, let’s start with the **Product** receiver class, which should contain the base business logic in our app:

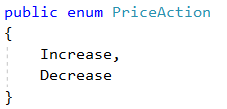


Now the Client class can instantiate the **Product** class and execute the required actions. But the Command design pattern states that we shouldn’t use receiver classes directly. Instead, we should extract all the request details into a special class - Command. Let’s do that.

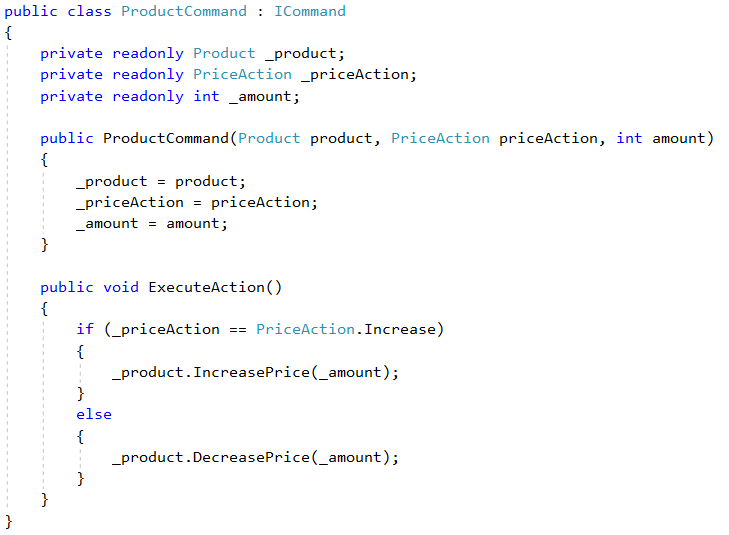
The first thing we are going to do is to add the **ICommand** interface:



Just to enumerate our price modification actions, we are going to add a simple **PriceAction** enumeration:



Finally, let’s add the **ProductCommand** class:



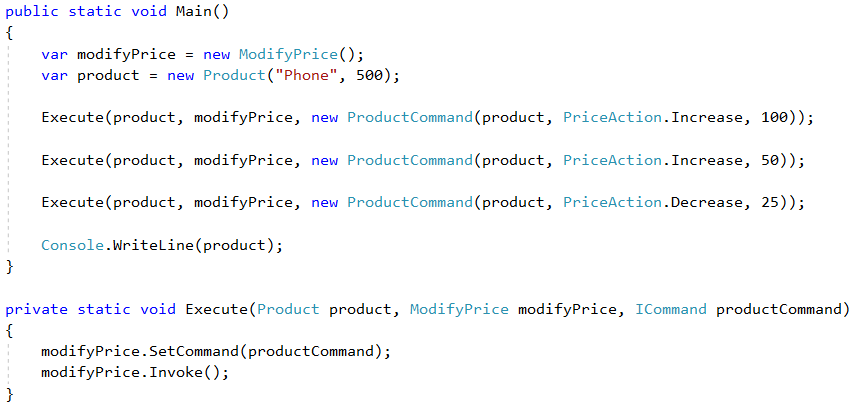
As we can see, the **ProductCommand** class has all the information about the request and based on that executes required action.

To continue on, let’s add the **ModifyPrice** class, which will act as Invoker:



This class can work with any command that implements the **ICommand** interface and store all the operations as well.

Now, we can start working with the client part:



The output should be like this:

