

Design Patterns - Lab

Lab problems for the ["C# Advanced" course @ Software University](#).

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:

```
namespace SingletonDemo
{
    public interface ISingletonContainer
    {
        int GetPopulation(string name);
    }
}
```

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

```
public class SingletonDataContainer : ISingletonContainer
{
    private Dictionary<string, int> _capitals = new Dictionary<string, int>();

    public SingletonDataContainer()
    {
        Console.WriteLine("Initializing singleton object");

        var elements = File.ReadAllLines("capitals.txt");
        for (int i = 0; i < elements.Length; i += 2)
        {
            _capitals.Add(elements[i], int.Parse(elements[i + 1]));
        }
    }

    public int GetPopulation(string name)
    {
        return _capitals[name];
    }
}
```

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.

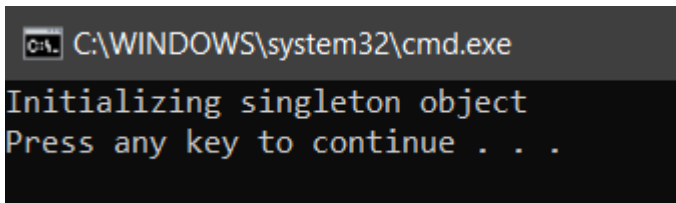
```
private static SingletonDataContainer instance = new SingletonDataContainer();

public static SingletonDataContainer Instance => instance;
```

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:

```
public static void Main()
{
    var db = SingletonDataContainer.Instance;
    var db2 = SingletonDataContainer.Instance;
    var db3 = SingletonDataContainer.Instance;
    var db4 = SingletonDataContainer.Instance;
}
```

The result in our console will be the following:



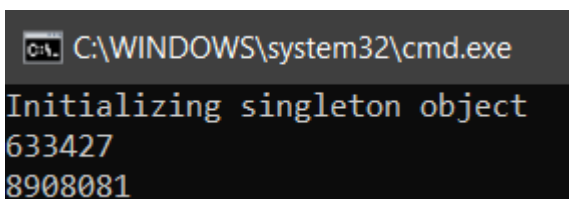
```
C:\WINDOWS\system32\cmd.exe
Initializing singleton object
Press any key to continue . . .
```

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:

```
public static void Main()
{
    var db = SingletonDataContainer.Instance;
    Console.WriteLine(db.GetPopulation("Washington, D.C.));
    var db2 = SingletonDataContainer.Instance;
    Console.WriteLine(db2.GetPopulation("London"));
}
```

The expected output should be something like this:



```
C:\WINDOWS\system32\cmd.exe
Initializing singleton object
633427
8908081
```

2. Façade

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

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

```

public class Car
{
    public string Type { get; set; }
    public string Color { get; set; }
    public int NumberOfDoors { get; set; }

    public string City { get; set; }
    public string Address { get; set; }

    public override string ToString()
    {
        return $"CarType: {Type}, Color: {Color}, Number of doors: {NumberOfDoors}, Manufactured in {City}, at address: {Address}";
    }
}

```

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:

```

public class CarBuilderFacade
{
    protected Car Car { get; set; }

    public CarBuilderFacade()
    {
        Car = new Car();
    }

    public Car Build() => Car;
}

```

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:

```

public class CarInfoBuilder : CarBuilderFacade
{
    public CarInfoBuilder(Car car)
    {
        Car = car;
    }

    public CarInfoBuilder WithType(string type)
    {
        Car.Type = type;
        return this;
    }

    public CarInfoBuilder WithColor(string color)
    {
        Car.Color = color;
        return this;
    }

    public CarInfoBuilder WithNumberOfDoors(int number)
    {
        Car.NumberOfDoors = number;
        return this;
    }
}

```

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 **CarAddressBuilder** class:

```
public class CarAddressBuilder : CarBuilderFacade
{
    public CarAddressBuilder(Car car)
    {
        Car = car;
    }

    public CarAddressBuilder InCity(string city)
    {
        Car.City = city;
        return this;
    }

    public CarAddressBuilder AtAddress(string address)
    {
        Car.Address = address;
        return this;
    }
}
```

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:

```
public class CarBuilderFacade
{
    protected Car Car { get; set; }

    public CarBuilderFacade()
    {
        Car = new Car();
    }

    public Car Build() => Car;

    public CarInfoBuilder Info => new CarInfoBuilder(Car);
    public CarAddressBuilder Built => new CarAddressBuilder(Car);
}
```

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

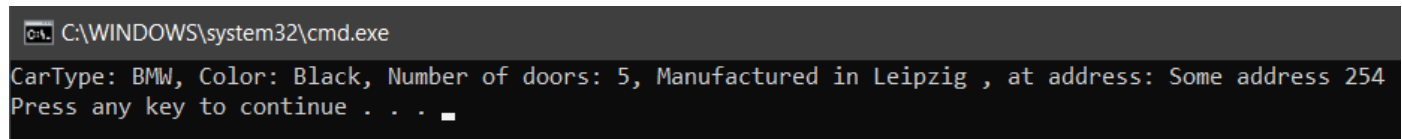
```

public static void Main()
{
    var car = new CarBuilderFacade()
        .Info
        .WithType("BMW")
        .WithColor("Black")
        .WithNumberOfDoors(5)
        .Built
        .InCity("Leipzig ")
        .AtAddress("Some address 254")
        .Build();

    Console.WriteLine(car);
}

```

And the output should be:



```

C:\WINDOWS\system32\cmd.exe
CarType: BMW, Color: Black, Number of doors: 5, Manufactured in Leipzig , at address: Some address 254
Press any key to continue . . .

```

3. 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:

```

public class Product
{
    public string Name { get; set; }
    public int Price { get; set; }

    public Product(string name, int price)
    {
        Name = name;
        Price = price;
    }

    public void IncreasePrice(int amount)
    {
        Price += amount;
        Console.WriteLine($"The price for the {Name} has been increased by {amount}$.");
    }

    public void DecreasePrice(int amount)
    {
        if (amount < Price)
        {
            Price -= amount;
            Console.WriteLine($"The price for the {Name} has been decreased by {amount}$.");
        }
    }

    public override string ToString() => $"Current price for the {Name} product is {Price}$.";
}

```

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:

```
public interface ICommand
{
    void ExecuteAction();
}
```

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

```
public enum PriceAction
{
    Increase,
    Decrease
}
```

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

```
public class ProductCommand : ICommand
{
    private readonly Product _product;
    private readonly PriceAction _priceAction;
    private readonly int _amount;

    public ProductCommand(Product product, PriceAction priceAction, int amount)
    {
        _product = product;
        _priceAction = priceAction;
        _amount = amount;
    }

    public void ExecuteAction()
    {
        if (_priceAction == PriceAction.Increase)
        {
            _product.IncreasePrice(_amount);
        }
        else
        {
            _product.DecreasePrice(_amount);
        }
    }
}
```

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:

```
public class ModifyPrice
{
    private readonly List<ICommand> _commands;
    private ICommand _command;

    public ModifyPrice()
    {
        _commands = new List<ICommand>();
    }

    public void SetCommand(ICommand command) => _command = command;

    public void Invoke()
    {
        _commands.Add(_command);
        _command.ExecuteAction();
    }
}
```

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:

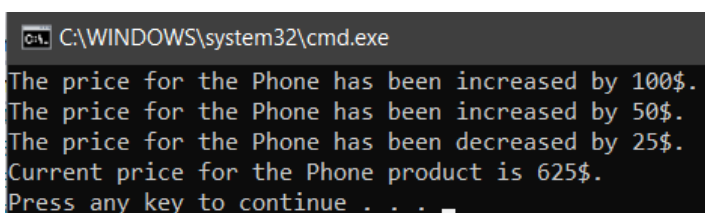
```
public static void Main()
{
    var modifyPrice = new ModifyPrice();
    var product = new Product("Phone", 500);

    Execute(product, modifyPrice, new ProductCommand(product, PriceAction.Increase, 100));
    Execute(product, modifyPrice, new ProductCommand(product, PriceAction.Increase, 50));
    Execute(product, modifyPrice, new ProductCommand(product, PriceAction.Decrease, 25));

    Console.WriteLine(product);
}

private static void Execute(Product product, ModifyPrice modifyPrice, ICommand productCommand)
{
    modifyPrice.SetCommand(productCommand);
    modifyPrice.Invoke();
}
```

The output should be like this:



```
C:\WINDOWS\system32\cmd.exe
The price for the Phone has been increased by 100$.
The price for the Phone has been increased by 50$.
The price for the Phone has been decreased by 25$.
Current price for the Phone product is 625$.
Press any key to continue . . .
```