Factory Design Pattern in C#

Back to: Design Patterns in C# With Real-Time Examples

Factory Design Pattern in C# with Real-Time Example

In this article, I am going to discuss the **Factory Design Pattern in C#** with examples. The Factory Design Pattern is one of the most frequently used design patterns in real-time applications. The Factory Design Pattern in C# falls under the category of Creational Design Pattern. As part of this article, we are going to discuss the following pointers related to the Factory Design Pattern.

1. **What is Factory Design Pattern?**
2. **Understanding the Factory Design Pattern.**
3. **Example without using the Factory Pattern.**
4. **Understanding the Problem of not using the factory design pattern**
5. **Implementing the Factory Design Pattern in C#?**
6. **When to use Factory Design Pattern in Real-time Application?**
7. **Problems of the Factory Design Pattern.**

What is Factory Design Pattern in C#?

Let us first try to understand the definitions of the factory design pattern. If you are not understanding the following definitions then don’t worry, we will explain the same with real-time examples.

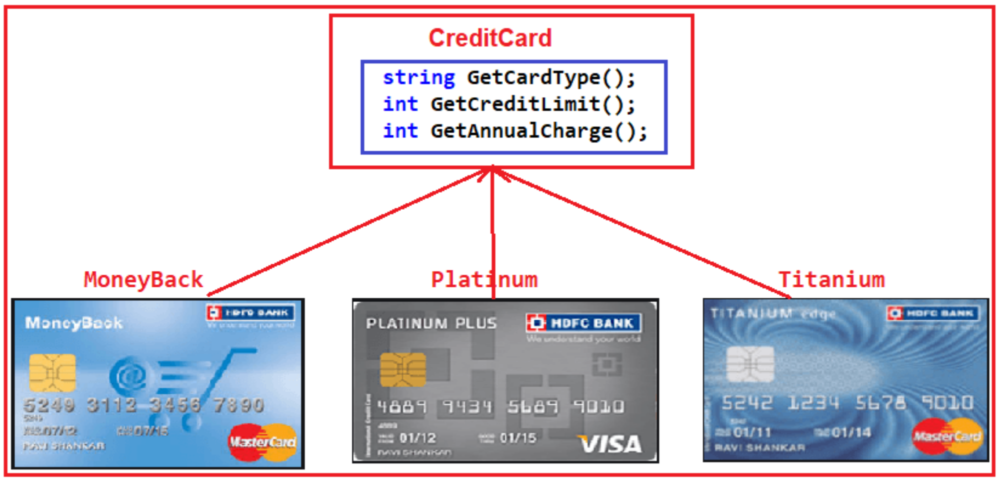
According to Gang of Four, the Factory Design Pattern states that **“A factory is an object which is used for creating other objects”**. In technical terms, we can say that a factory is a class with a method. That method will create and return different types of objects based on the input parameter, it received. In simple words, if we have a superclass and n number of subclasses, and based on the data provided, if we have to create and return the object of one of the subclasses, then we need to use the Factory Design Pattern in C#.

In the Factory Design pattern, we create an object without exposing the object creation logic to the client and the client will refer to the newly created object using a common interface. The basic principle behind the factory design pattern is that, at run time, we get an object of a similar type based on the parameter we pass.

Understanding the Factory Design Pattern in C# with one real-time example

Let us understand the Factory Design Pattern with one simple example. We are going to develop an application for showing the credit card details.

Please have a look at the following diagram. Here, as you can see we have three credit card classes i.e. MoneyBack, Titanium, and Platinum and these three classes are the subclasses of CreditCard superclass or super interface. The CreditCard superclass or super interface has three methods i.e. GetCardType, GetCreditLimit, and GetAnnualCharge. The subclasses i.e. MoneyBack, Titanium, and Platinum have implemented the above three methods.



Our requirement is, we will ask the user to select the credit card. Once the user selects the credit card then we need to display the required information of that selected card. Let us first discuss how to achieve this without using the Factory Design Pattern in C#. Then we will discuss the problems and finally, we will create the same application using the Factory Design Pattern in C#.

Without using the Factory Design Pattern in C#

Step1: Create the Product Interface

Here we need to create either an interface or an abstract class that will expose the operations a credit card should have. So, create a class file with the name **CreditCard.cs** and then copy and paste the following code in it. As you can see, we created the interface with three methods.

```
namespace FactoryDesignPattern
{
    public interface CreditCard
    {
        string GetCardType();
        int GetCreditLimit();
        int GetAnnualCharge();
    }
}
```

Now we need to create three Product classes that will implement the above interface.

Step2: Creating Product Classes

In our example, we have three Credit cards. So, we need to create three classes. First, create a class file with the name **MoneyBack.cs** and then copy and paste the following code into it.

```
namespace FactoryDesignPattern
{
    class MoneyBack : CreditCard
    {
        public string GetCardType()
        {
            return "MoneyBack";
        }

        public int GetCreditLimit()
        {
            return 15000;
        }

        public int GetAnnualCharge()
        {
            return 500;
        }
    }
}
```

As you can see this class implements the **CreditCard** interface and provide implementation to all three methods. Similarly, we need to do the same thing for the other two credit card classes.

Titanium.cs:

Create a class file with the name **Titanium.cs** and then copy and paste the following code into it.

```
namespace FactoryDesignPattern
{
    public class Titanium : CreditCard
    {
        public string GetCardType()
        {
            return "Titanium Edge";
        }
        public int GetCreditLimit()
        {
            return 25000;
        }
        public int GetAnnualCharge()
        {
            return 1500;
        }
    }
}
```

Platinum.cs:

Create a class file with the name **Platinum.cs** and then copy and paste the following code in it.

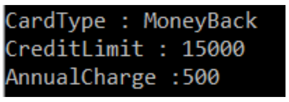
```
namespace FactoryDesignPattern
{
    public class Platinum : CreditCard
    {
        public string GetCardType()
        {
            return "Platinum Plus";
        }
        public int GetCreditLimit()
        {
            return 35000;
        }
        public int GetAnnualCharge()
        {
            return 2000;
        }
    }
}
```

Step3: Client Code

Now in the client code, we will ask the user to select the Credit Card Type. And based on the Selected Credit card, we will create an instance of any one of the above three product implementation classes. So, modify the Main method as shown below.

```
using System;
namespace FactoryDesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            //Generally we will get the Card Type from UI.
            //Here we are hardcoded the card type
            string cardType = "MoneyBack";

            CreditCard cardDetails = null;

            //Based of the CreditCard Type we are creating the
            //appropriate type instance using if else condition
            if (cardType == "MoneyBack")
            {
                cardDetails = new MoneyBack();
            }
            else if (cardType == "Titanium")
            {
                cardDetails = new Titanium();
            }
            else if (cardType == "Platinum")
            {
                cardDetails = new Platinum();
            }

            if (cardDetails != null)
            {
                Console.WriteLine("CardType : " + cardDetails.GetCardType());
                Console.WriteLine("CreditLimit : " + cardDetails.GetCreditLimit());
                Console.WriteLine("AnnualCharge :" + cardDetails.GetAnnualCharge());
            }
            else
            {
                Console.Write("Invalid Card Type");
            }

            Console.ReadLine();
        }
    }
}
```

The above code implementation is very straightforward. Once we get the CardType value, then by using the if-else condition we are creating the appropriate Credit Card instance. Then we are just calling the three methods to display the credit card information in the console window. So, when you run the application, you will get the output as expected as shown below.



```
CardType : MoneyBack
CreditLimit : 15000
AnnualCharge :500
```

What is the Problem of the above code implementation?

The above code implementation introduces the following problems

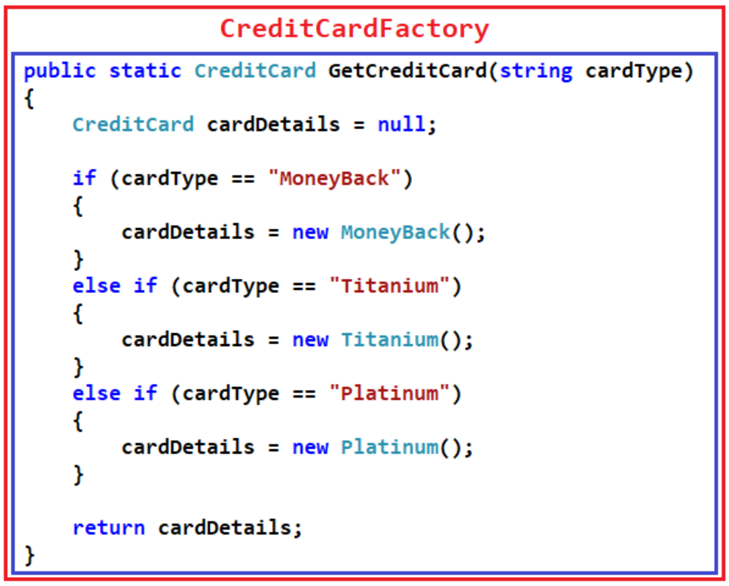
1. First, the tight coupling between the client class (Program) and Product Class (MoneyBack, Titanium, and Platinum).
2. Secondly, if we add a new Credit Card, then also we need to modify the Main method by adding an extra if-else condition which not only overheads in the development but also in the testing process

Let us see how to overcome the above problem by using the factory design pattern.

Factory Design Pattern Implementation in C#:

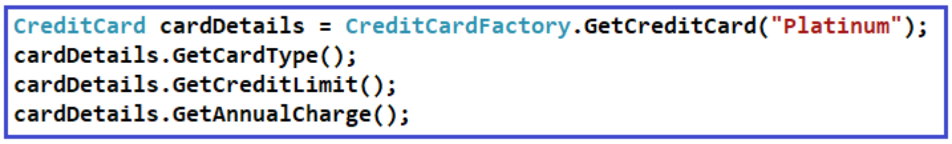
As per the definition of Factory Design Pattern, the Factory Design Pattern create an object without exposing the object creation logic to the client and the client refers to the newly created object using a common interface.

Please have a look at the following image. This is our factory class and this class takes the responsibility of creating and returning the appropriate product object. As you can see this class having one static method i.e. GetCreditcard and this method takes one input parameter and based on the parameter value it will create one of the credit card (i.e. MoneyBack, Platinum, and Titanium) objects and store that object in the superclass (CrditCard) reference variable and finally return that superclass reference variable to the caller of this method.



```
CreditCardFactory
public static CreditCard GetCreditCard(string cardType)
{
    CreditCard cardDetails = null;

    if (cardType == "MoneyBack")
    {
        cardDetails = new MoneyBack();
    }
    else if (cardType == "Titanium")
    {
        cardDetails = new Titanium();
    }
    else if (cardType == "Platinum")
    {
        cardDetails = new Platinum();
    }

    return cardDetails;
}
```

Now the client needs to create the object through CreditCardFactory. For example, if the client wants to create the instance of Platinum Credit then he/she needs to do something like the below. As you can see, he/she needs to pass the Credit card type to the GetCreditcard method of the CreditCardFactory class. Now, the GetCreditcard() method will create a Platinum class instance and return that instance to the client.



```
CreditCard cardDetails = CreditCardFactory.GetCreditCard("Platinum");
cardDetails.GetCardType();
cardDetails.GetCreditLimit();
cardDetails.GetAnnualCharge();
```

Step4: Creating Factory Class

Create a class file with the name **CreditCardfactory.cs** and then copy and paste the following in it.

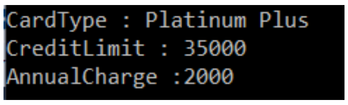
```
namespace FactoryDesignPattern
{
    class CreditCardFactory
    {
        public static CreditCard GetCreditCard(string cardType)
        {
            CreditCard cardDetails = null;

            if (cardType == "MoneyBack")
            {
                cardDetails = new MoneyBack();
            }
            else if (cardType == "Titanium")
            {
                cardDetails = new Titanium();
            }
            else if (cardType == "Platinum")
            {
                cardDetails = new Platinum();
            }

            return cardDetails;
        }
    }
}
```

Step5: Modifying the Client Code

Modify the Main method as shown below.

```
using System;
namespace FactoryDesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            CreditCard cardDetails = CreditCardFactory.GetCreditCard("Platinum");

            if (cardDetails != null)
            {
                Console.WriteLine("CardType : " + cardDetails.GetCardType());
                Console.WriteLine("CreditLimit : " + cardDetails.GetCreditLimit());
                Console.WriteLine("AnnualCharge :" + cardDetails.GetAnnualCharge());
            }
            else
            {
                Console.Write("Invalid Card Type");
            }

            Console.ReadLine();
        }
    }
}
```

Output:



```
CardType : Platinum Plus
CreditLimit : 35000
AnnualCharge :2000
```

Real-Life Example of Factory Pattern:

From Lehman's point of view, we can say that a factory is a place where products are created. In order words, we can say that it is a centralized place for creating products. Later, based on the order received, the appropriate product is delivered by the factory. For example, a car factory can produce different types of cars. If you are ordering a car to the car factory, then based on your requirements or specifications, the factory will create the appropriate car and then delivered that car to you.

The same thing also happens in the factory design pattern. A factory (i.e. a class) will create and deliver products (i.e. objects) based on the incoming parameters.

When to use the Factory Design Pattern in real-time applications?

It would not be a good programming approach to specify the exact class name while creating the objects by the client which leads to tight coupling between the client and the product. To overcome this problem, we need to use the Factory Design Pattern in C#. This design pattern provides the client with a simple mechanism to create the object. So, we need to use the Factory Design Pattern in C# when

1. The Object needs to be extended to the subclasses
2. Classes don't know what exact sub-classes it has to create
3. The Product implementation going to change over time and the Client remains unchanged

Problems of Simple Factory Pattern in C#

1. If we need to add any new product (i.e. new credit card) then we need to add a new if else condition in the GetCreditCard method of the CreditCardFactory class. This violates the open/closed design principle.
2. We also have a tight coupling between the Factory (CreditCardFactory) class and product classes (MoneyBack, Titanium, and Platinum).

In the next article, I am going to discuss how to overcome the above problem by using the **Factory Method Design Pattern** in C#. Here, in this article, I try to explain the **Factory Design Pattern in C#** with an example. I hope this article will help you with your needs. I would like to have your feedback. Please post your feedback, question, or comments about this article.