Article

Factory Design Pattern With Java

By **Aman Gupta** on **Updated date Oct 08, 2020**

Introduction

As discussed in the last article, we will talk about what a design pattern is, and what it provides with a singleton design pattern.

A Design Pattern provides a simple solution to common problems that are faced in day to day life by software developers. Three basic types of design patterns exist, which are listed below:

- Creational Pattern
- Structural Pattern
- Behavioral Pattern

Along with these, 23 design patterns are classified in the above three patterns.

Thus, the pattern described here is like a blueprint that you can customize to solve a particular design problem and they only differ by complexity and scalability of the code.

So here in this article, we will learn about the Factory design pattern. Let's get started.

Factory Design Pattern

As the name suggests, we are dealing with a "factory", i.e. a warehouse, so the factory design pattern has a warehouse of an object what the user wants.

Factory Design Pattern is a part of the creational design pattern. Also known as Virtual Constructor.

In the Software Development Life Cycle (SDLC) or Software Engineering (SE), whenever working with the project, we have to work to work with models and all these models have loose-coupling & high cohesion. So to achieve loose-coupling, we use something called factory i.e instead of manually creating the object, ask someone else to create the object.

Consider a scenario of an operating system in the mobile industry. Let's assume your choice of mobile depends on the operating system before purchasing. Depending upon your requirement, we have different-different operating systems.

Let understand this example practically:

Consider an interface OS implemented by Android, IOS, & Windows OS, as shown below:

```
1. package FactoryDesign;
2. public interface OS {
3.     void show();
4. }
```

Here we have an Android Class:

```
1. package FactoryDesign;
2. public class Android implements OS {
3.     public void specification() {
4.         System.out.println("Most Powerful Operating System");
5.     }
6. }
```

Here we have an IOS Class:

```
1. package FactoryDesign;
2. public class IOS implements OS {
3.     public void specification() {
4.         System.out.println("Most Secured Operating System");
5.     }
6. }
```

Here we have a Windows Class:

```
1. package FactoryDesign;
2. public class Window implements OS {
3.     public void specification() {
4.         System.out.println("Most Wired Operating System");
5.     }
6. }
```

Let's take an interface. We have multiple implementations for that interface.

We have an interface OS and we have its implemented as Android OS, IOS, and Windows OS. So to instantiate that OS we have to use one of the OS mentioned in class.

Depending upon the requirements, we get the instance of the OS. That’s why we need a pattern called the Factory Design Pattern.

Note
An instance of any class is provided according to the requirement.

The above scenario is just for an example. Let's understand it with another example. Here is the code.

First, create an abstract class or an Interface.

```
1. package FactoryDesign;
2. //rule 1
3. public interface CSharpCorner {
4.     void role();
5. }
```

Second, create a class and implement the above interface.

```
1. package FactoryDesign;
2. //rule 2
3. public class Author implements CSharpCorner {
```

```
    @Override
    public void role() {
        System.out.println("Author publishes content. ");
    }
}
```

```
package FactoryDesign;
// rule 2
public class Editor implements CSharpCorner {
    @Override
    public void role() {
        System.out.println("Editor checks & edit the content. ");
    }
}
```

Third, add a factory class and one factory method to it along with the fourth and final step. (i.e. use the factory class in any other class.)

```
package FactoryDesign;
//rule 3
public class CsharpCornerMain {
    public static void main(String[] args) {
        //rule 4
        Csharp obj = CsharpFactory.getPerson("AUTHOR");
        obj.role();
        Csharp obj1 = CsharpFactory.getPerson("Editor");
        obj1.role();
    }
}
```

```
package FactoryDesign;
/*RULES
 * 1.-> Create abstract class or Interface
 * 2.-> Create class and Implement Interface.
 * 3.-> Add factory class and add one Factory Method
 * 4.-> Use Factory Class in any other class
 */
//Factory Design is used to create object explicitly i.e without manually (Author a = new Author();)
public class CsharpFactory {
    public static Csharp getPerson(String personType) {
        Csharp csharp = null;
        if (personType.equalsIgnoreCase("Author")) {
            csharp = new Author();
        } else if (personType.equalsIgnoreCase("Editor")) {
            csharp = new Editor();
        } else {
            csharp = null;
        }
        return csharp;
    }
}
```

In this scenario, we have t make changes in the code. That means our client knows which file we are working with.

So, what the factory pattern says is that instead of creating a direct object, we can have a factory class that will return the type of person depend upon the string passed.

Note
Since we are using factory objects thus known as a factory design pattern and if we have an extra class in the future it will not change your client application and is used majorly in Java development.

Advantage

- Reusability of the code.
- Helps in ensuring encapsulation.
- Ensures abstraction between class and client by inheritance.

Realtime Example in JDK

- DriverManger getConnection()
- openConnection()
- new Instance()
- getInstance()
- forName()
- Calender
- Resource Binding
- NumberFormat

Summary

Thus, coming towards the end of this article, we learned a factory design pattern.

What did we learn?

- What is the design pattern?
- Why design patterns?
- Types of design patterns.
- Factory design pattern.
- Advantage of factory design pattern
- Realtime example in JDK