Factory method design pattern in Java

Difficulty Level : Easy Last Updated : 03 Aug, 2022

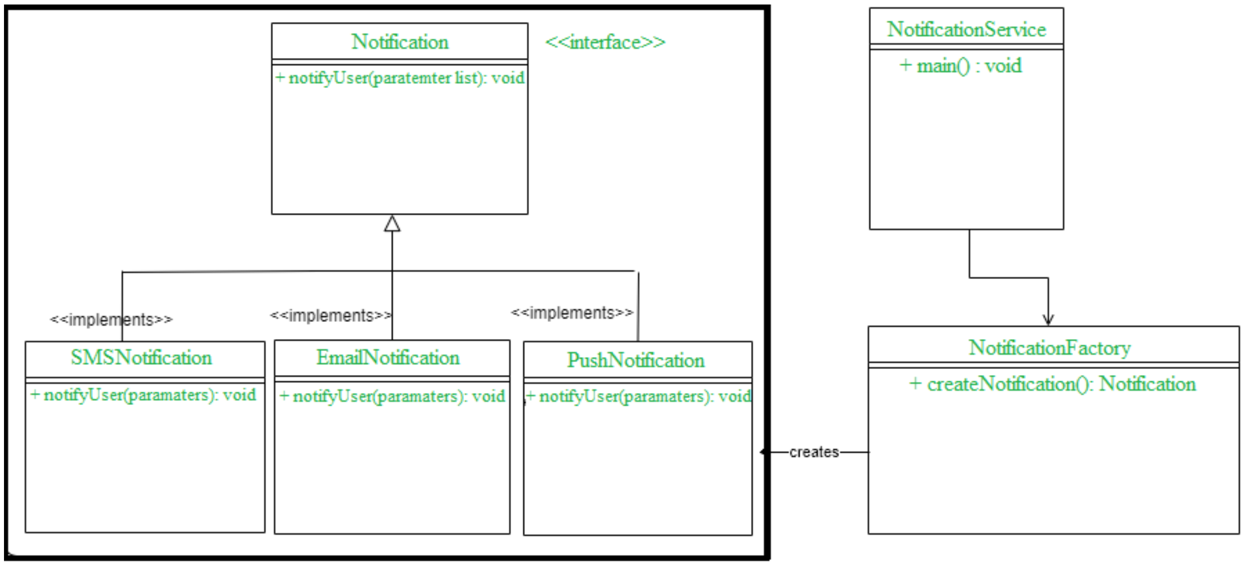
It is a creational design pattern that talks about the creation of an object. The factory design pattern says that define an interface (A java interface or an abstract class) for creating object and let the subclasses decide which class to instantiate. The factory method in the interface lets a class defers the instantiation to one or more concrete subclasses. Since these design patterns talk about the instantiation of an object and so it comes under the category of creational design pattern. If we notice the name **Factory method**, that means there is a method which is a factory, and in general, factories are involved with creational stuff and here with this, an object is being created. It is one of the best ways to create an object where object creation logic is hidden from the client. Now Let's look at the implementation.

Implementation:

1. Define a factory method inside an interface.
2. Let the subclass implements the above factory method and decides which object to create.
3. In Java, constructors are not polymorphic, **but by allowing subclass to create an object, we are adding polymorphic behavior to the instantiation**. In short, we are trying to achieve Pseudo polymorphism by letting the subclass to decide what to create, and so this Factory method is also called a **virtual constructor**.

Let us try to implement it with a real-time problem and some coding exercises.

Problem Statement: Consider we want to implement a notification service through email, SMS, and push notifications. Let's try to implement this with the help of the factory method design pattern. First, we will design a UML class diagram for this.



In the above class diagram, we have an interface called **Notification**, and three concrete classes are implementing the Notification interface. A factory class NotificationFactory is created to get a Notification object. Let’s jump into the coding now.

Create Notification interface

java

```java
public interface Notification {
    void notifyUser();
}
```

Note- Above interface could be created as an abstract class as well.

Create all implementation classes

```
SMSNotification.java
```

java

```java
public class SMSNotification implements Notification {

    @Override
    public void notifyUser()
    {
        // TODO Auto-generated method stub
        System.out.println("Sending an SMS notification");
    }
}
```

```
EmailNotification.java
```

java

```java
public class EmailNotification implements Notification {

    @Override
    public void notifyUser()
    {
        // TODO Auto-generated method stub
        System.out.println("Sending an e-mail notification");
    }
}
```

PushNotification.java

java

```java
public class PushNotification implements Notification {

    @Override
    public void notifyUser()
    {
        // TODO Auto-generated method stub
        System.out.println("Sending a push notification");
    }
}
```

Create a factory class NotificationFactory.java to instantiate concrete class.

java

```java
public class NotificationFactory {
    public Notification createNotification(String channel)
    {
        if (channel == null || channel.isEmpty())
            return null;
        switch (channel) {
        case "SMS":
            return new SMSNotification();
        case "EMAIL":
            return new EmailNotification();
        case "PUSH":
            return new PushNotification();
        default:
            throw new IllegalArgumentException("Unknown channel "+channel);
        }
    }
```