Simple Factory Design Pattern

1. Simple Factory – Introduction:

Now let's learn about Simple Factory. Now, before we begin, I would like to make you aware that a lot of people do not consider simple factory as a design pattern. And you will see why that is the case. So before we begin, let's look at a code snippet where simple factory often finds its use. So here I have a simple if else block and I'm comparing a variable against a known value. And depending upon the variable value, I'm instantiating object of a particular class. And this is the type of code where simple factory is often used.

A screenshot of a computer program

Description automatically generated

So what exactly is a simple factory?

Well, we simply take this instantiation logic that we just saw, and we put that in a separate class. And typically, we create a static method to place this logic because we don't need any state information. Now you will see why this particular simple factory is not considered a design pattern because it is simply an encapsulation of our object instantiation logic in a method. Nothing complex goes on. In that method, we simply compare a single variable and depending upon its value, we create different type of object. Now, the reason we are studying simple factory is because it is often confused with **factory method pattern** by newcomers to design patterns. So by learning about simple factory as well as factory method, you will be in a better position to differentiate between these two. Now, typically we want to create a simple factory if we have more than one option when instantiating object. And that choice or that decision is based upon a simple logic like comparison of a variable value.

So now let's look at the UML diagram of Simple Factory. Now this is a very simple UML diagram. We have a product class here, so this can be an interface or an abstract class and we will have multiple subclasses that implement our product interface or abstract class. Next up, we have static simple factory. Now, typically this is a separate class, and inside this class we will have a static method that accepts a parameter. And depending upon value of this parameter, we will decide which class to instantiate.

A diagram of a product

Description automatically generated

1. Steps to implement a Simple Factory:

Let's look at some of the steps that we need to follow when implementing a simple factory. Now, as you can guess, it's pretty straightforward to implement a simple factory. We start by creating a separate class, which will be our simple factory. Now, inside this class, we will add a method that returns instances of classes that we want. Now, this method is typically created as a static method, and it will accept some type of argument. And that argument will decide which class is instantiated inside that static method. Now we can also provide additional arguments and those arguments can be used as constructor arguments while instantiating those classes.

A black and white text

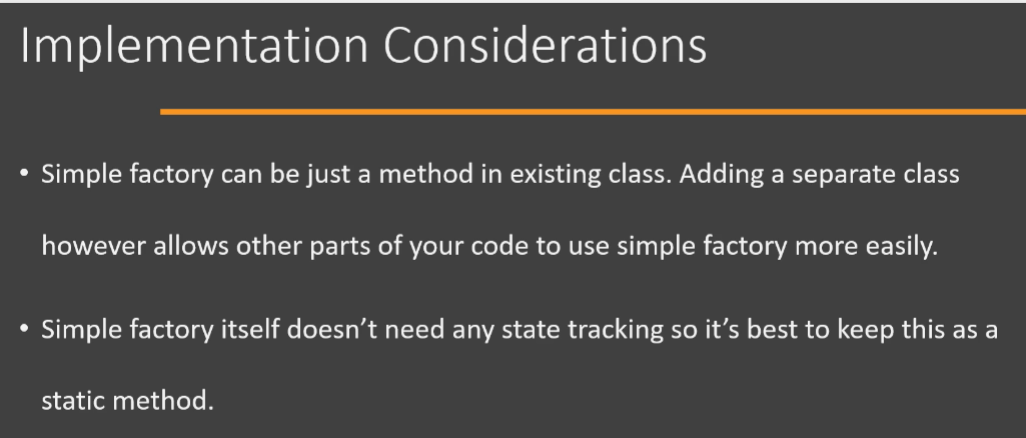
Description automatically generated with medium confidence

1. Simple Factory – Implementation

We're going to see how we can implement Simple Factory in Java code. Extract Creational\_design\_pattern/Simple\_factory\_design\_pattern/simple\_factory\_end.zip file and navigate to …\simple\_factory\_end\udemy-design-patterns-handson\src\com\coffeepoweredcrew\simplefactory folder, you will see that I have written a couple of classes that we are going to use while implementing Simple Factory. So I will go through these classes one by one and explain what their purpose is. So first of all, we have a Post class. This is our abstract product(final object) class. So this Post represents an article that is published on a website. Now there are multiple subclasses of this Post. We have a NewsPost and this class represents a news article that is published on our website. On similar lines, we have a BlogPost which represents a blog article that is published. Then we have a ProductPost which represents a product information page on our website. So PostFactory is a class where we are going to implement a simple factory. Now, in order to implement a simple factory, we are going to start by creating a static method(i.e., public static) and we are going to return an object which is subclass of our Post class. I'm going to name this method as createPost(). Now we have discussed that while declaring simple factory or while implementing simple factory, we need a criteria, a variable whose value decides which object we are going to create. So I'm going to accept a simple string argument(string parameter) here called as ‘type’ and based upon value of this ‘type’, we are going to instantiate one of these three subclasses of Post. So I will start by writing a switch statement and inside this switch statement I'm going to compare value of our ‘type’ parameter and in case, the value of ‘type’ is “blog”, we are going to return a ‘new BlogPost()’(i.e., an instance of BlogPost class). On the similar line, if the ‘type’ variable is set to “news”, then we are going to return a ‘new NewsPost()’ (i.e., an instance of NewsPost class). You can see that how easy it is to implement a simple factory. And lastly, we are going to say that if the ‘type’ is ‘product’, then we are going to return ‘new ProductPost()’ (i.e., an instance of ProductPost class). In case we get a value, in ‘type’ variable, that is not known to us then we are going to add a default case here and here we can simply throw IllegalArgumentException to abort the program. So this is how we can implement a simple factory. Now, in order to use this Simple Factory, we are going to create a new class here, let's call this class Client. Now, inside this main method, we are simply going to use our simple factory.

1. Implementation and Design considerations:

Let's look at some of the implementation and design considerations when working with a simple factory. Now, simple factory can just be a method in your existing class. Adding a separate class, however, allows other parts of our code to use that simple factory without importing an unrelated class. So we will have a single simple class with a single static method in it. And this separation allows, that class to be used from anywhere in our code. Now, simple factory itself doesn't need any state tracking, and that is the reason we typically have a static method that creates our object instances.



Now let's look at some of the design considerations. Simple Factory can in turn use other design patterns like builder in order to create the objects. Okay, so you're free to use any other design pattern inside of your method that creates your objects. Now, in case you want to specialize your simple factory in subclasses, then you need to use factory method design pattern.

A screenshot of a computer

Description automatically generated

1. Real-world examples of Simple Factory:

Now let's look at some real world examples of Simple Factory and you will find that there are no problems finding examples of simple factory in almost every application or every framework out there. In fact, I would like to warn you that many examples that you will find online about factory method are in fact examples of simple factory. So let's look at one of the examples. So we have a class here called as NumberFormat, which is present in java.text package. Now this class has a getInstance() method, which is an example of simple factory. So let's look at the code of this class. So this code is taken from the NumberFormat.class file in the runtime.jar file which comes with your JDK. So you will see that we have an getInstance() method, which is a static method that is used internally by the NumberFormat class and you will see that we have our familiar code snippet here. We are checking an incoming parameter(i.e., choice) for our simple factory method and based on the value of that ‘choice’, we are going to instantiate a particular class and then we are going to return that instance/object to the outside world. All right, so NumberFormat class is an example of simple factory.

A computer screen shot of a computer program

Description automatically generated

1. Simple Factory – Pitfalls:

Now, let's look at the pitfalls of simple factory. The only pitfall in simple factory is that the criteria that is used by simple factory to decide which object to instantiate, it can get more convoluted and complex over time. And typically, when you find yourself in such a situation, then that is a good indication that you need to use factory method design pattern. So you would often see in your goal that you start with a simple factory which then evolves to a factory method design pattern implementation. So this is the only pitfall for simple factory.

A screenshot of a computer

Description automatically generated

1. Simple Factory – Summary:

Now let's look at the summary for our Simple Factory. Now, as we have discussed multiple times, simple factory is not considered a design pattern by many people, and the reason for that is simple factory is simply an encapsulation of our object creation logic. So what we do is we take our object creation logic and we encapsulate that logic in a static method in a separate class. And that is all there in simple factory. Now, while defining this method, we can accept a parameter which will act as the criteria and based upon which we will decide which object will be created. We can also accept additional parameters in this method, and these parameters will be used in the constructor of the classes that we are going to instantiate. Now this is the UML diagram of Simple Factory. We have a Product interface or an abstract class and there are multiple classes which implement Product. Now we are going to create object of one of these subclasses in our simple factory. Simple factory itself is usually a separate class and we have a static method that acts as our simple factory. We provide the criteria based upon which this method decides which object is created. The client will use the static method and pass the criteria and get the desired object from this method.

A screenshot of a computer

Description automatically generated