Factory Method Design Pattern

1. Factory Method – Introduction:

We are going to learn about factory method design pattern. Now this is a creational design pattern and it is one of the most versatile creational design pattern. So why we use factory method? Well, whenever we want to move our object creation logic, which may be spreaded throughout our code and encapsulate that in a separate class, we can use factory method. Now you may be wondering, isn't this similar to what we did in Simple Factory? Well, that is true, but Factory method adds something new to this particular design pattern(i.e., simple factory). ~~Whenever we use this pattern, we do not have to know which class we are going to instantiate beforehand~~. And that means we can keep adding new classes to be instantiated and our factory method design pattern can handle that without requiring any change to the client code of our application. And factory method achieves this by delegating the object creation to subclasses. So subclasses decide which object to instantiate, and they do that by overriding our factory method. We're going to see how it is handled in a little bit.

A black and white text on a gray background

Description automatically generated

So, let's look at the UML diagram of the factory method design pattern. To start off, we have a Product role and this is the base class or interface of our product classes. That means we are going to create objects of subclasses of this Product interface or base class. Then we have the actual concrete product classes(i.e., ConcreteProductB and ConcreteProductA), and this will basically implement our Product interface and we are going to create object of these concrete classes/products. Then we have our Creator. Now this is where factory method design pattern starts to be implemented. So this particular class declares the abstract factory method, and additionally this class can use that abstract method and create the final Product instance. Then we have the actual concrete creators(i.e., ConcreteCreatorA and ConcreteCreatorB) and this is the class which overrides or implements the factory method and it returns the instance of one of the concrete product classes.

A screenshot of a computer

Description automatically generated

A diagram of a product

Description automatically generated

1. Steps to implement Factory Method design pattern:

Let's look at some of the steps that we need to follow in order to implement a factory method and it's quite simple process. We start by creating our Creator. Now, Creator itself can be a concrete class if it can provide a default object in its factory method. Remember, factory method is simply a method that can be overridden in one or all of its subclasses. So creator doesn't have to be an abstract class if it can provide a default object and if any other subclass can’t handle the creation. If it cannot provide a default object, then our creator will define an abstract factory method and the class itself will become an abstract. Now implementations or subclasses of our Creator will override our factory method and they will return an actual concrete object. All right, so these are the only steps that you need to follow when implementing the factory method.

A screen shot of a black background

Description automatically generated

1. Factory Method : Hands-on Example UML:

Now it's time to look at the UML of the example that we are going to implement in our Java code. And I promise as we implement this design pattern, it will become more and more clear how we need to implement this design pattern. So we start by defining a Message class. So this will be our product class. If you remember the UML(above) for our Factory method design pattern, we have a product class which will be an interface or a base class, and we want to create objects of this class or one of its subclasses. So we have a Message class here which will be an abstract class and it represents a message in our system. And then we have two subclasses for this message. We have a TextMessage and a JSONMessage. So these subclasses will differ on how they store the contents of the message. So TextMessage will store our message as a simple text string. A JSONMessage, on the other hand, will have a JSON formatted string and then we have our MessageCreator. So this will be our abstract creator and it will define a factory method that creates object of one of these subclasses(TextMessage and JSONMessage). We will have two implementations of our MessageCreator. One will be JSONMessageCreator and this class will override the factory method and instantiate the JSONMessage product class. Then we have TextMessageCreator which will instantiate the TextMessage product class.

A screenshot of a computer

Description automatically generated

1. Factory Method – Implementation:

Refer Creational\_design\_pattern/Simple\_factory\_design\_pattern/factory\_method\_end.zip file and navigate to …\factory\_method\_end\udemy-design-patterns-handson\src\com\coffeepoweredcrew\factorymethod folder.

It's time to implement the factory design pattern. So, as usual, I have written a couple of classes that we are going to use while implementing this design pattern, and I'm going to go over them one by one. So here we have a Message class, and this is our product class and this is an abstract class. So we are going to create objects of subclass of this Message. Now we have JSONMessage, which is a concrete class and which extends from our product class, which is Message on the similar lines. We have a TextMessage, then we have MessageCreator and this is going to be our abstract creator and this class is going to have the abstract factory method. So let's start by implementing our factory method pattern. So to start off, I'm going to declare a method[i.e., getMessage()] that is going to be used by client in order to get object of Message. So our return type is going to be Message. And this Message refers to the abstract product class. And what we are going to do is I'm going to write the abstract factory method called as createMessage(). All right, so this is our factory method, which is an abstract method. Now we are going to get the Message object from this createMessage() method. Remember, createMessage() method will be implemented by one of the concrete creators. And our abstract creator is going to use that method in order to create the Message object and then it can do some additional processing on this, which we are doing inside getMessage() method. For example, it may do add default headers to our Message object or, let’s say, encrypt the message and finally it's going to return that Message object. So this is how you will implement an abstract creator, in our case it is MessageCreator, in factory method. So abstract creator will typically perform some additional processing on the object that is returned by subclasses. Now, this is not mandatory, but this is typically how the abstract creator works, so it will use this abstract method to get the object, provide some additional properties or perform some additional operations on this object and finally return that to the client. Now it's time to implement our concrete creators. Now, as you can see, implementing concrete creators can be quite easy, since we have handled all the common processing that is needed on the product(i.e., Message object) in our abstract creator’s getMessage() method, this class is simply going to override the createMessage() method. And since this is a JSONMessageCreator class, we are going to simply return ‘new JSONMessage()’. In JSONMessageCreator concrete class, inside the overridden createMessage() method, you can do some additional processing specific on JSONMessage class’s object after creating the JSONMessage object and then return that object. So on the similar lines, we also have a TextMessageCreator class. And as you can guess, this class is going to again override the createMessage() method and return a TextMessage object from it. We are now going to jump into Client class, which has a main() method, just see the code. So this is how we can use the factory method design pattern in our client code.

1. Factory Method – Implementation and Design considerations:

Now let's look at some of the implementation and design considerations while implementing factory method. Now we have seen so far our Creator being abstract class, but that doesn't have to be the case. Your creator can be a concrete class if you can provide a default object in its factory method implementation. Now you can also use the simple factory way of providing additional arguments or criteria in your factory method. So if you remember our Simple Factory design pattern, it was a very simple class with a static method that took a single argument, which was the criteria based upon which we decided which object to create. Well, we can use that same logic here. So our factory method currently doesn't take any argument, but that doesn't have to be the case. We can accept a criteria in our factory method and using that criteria, we can decide which object to instantiate. So you can extend your factory method using that particular logic.

A black and white text on a gray background

Description automatically generated

Now let's look at some of the design considerations, the creator hierarchy that we saw, for example, in our hands-on example, we had a MessageCreator and then we had two child classes i.e., JSONMessageCreator and TextMessageCreator. This hierarchy reflects the product(i.e., Message) hierarchy. That means the objects(i.e., Message) that we are going to create and the creators(i.e., MessageCreator) are having similar hierarchy and we typically end up with one concrete creator per concrete product that we add in our application. The only point that I wanted to highlight simply is that the two hierarchies of the product classes and their creators reflect each other. Also, you will see that Template Method design pattern also makes use of the factory method and another creational design pattern that we are going to look at, called as Abstract Factory, also makes use of factory method design pattern. So these are some of the points that you should be aware of when using factory method design pattern.

A screenshot of a computer

Description automatically generated

1. Factory Method – Real world example:

Now it's time to look at an example of factory method in real world. So the java.util.collection interface or the abstract class, which is present in the same package, which is the AbstractCollection class, has an abstract method called as ‘iterator’. Now this abstract method is an example of a factory method. So let's look at the code snippet from the AbstractCollection class. Remember, this code is taken from the java.util.AbstractCollection class from the runtime.jar file. In this class, you will see that we have a method here called as iterator(). And the purpose of this method is that this method will return an object which implements the Iterator interface, which is used by the client code to iterate over the collection. All right, so this iterator() method, which is present in the AbstractCollection, or you can say even the Collection interface is an example of factory method. Now remember the most defining characteristic of factory method pattern is that subclasses are providing the actual instance of another class. So you will see many examples online where people are giving static methods as an example of factory method. But those are not actually the factory method design pattern examples as per Gang of Four book. Because in a factory method design pattern, the base class defines the factory method, which is iterator() method in this example. And subclasses of the base class will actually provide a concrete implementation of that particular method. So we have classes like ArrayList, Linkedlist, which are subclasses of our AbstractCollection class, and those classes will implement the iterator() method and they will create an object in that overridden method and then return that object to the client code. So the iterator() method in the AbstractCollection is an example of factory method design pattern.

A screen shot of a computer

Description automatically generated

1. Factory Method – Pitfalls:

Let's look at some of the pitfalls of using factory method design pattern. Well, first of all, this pattern is a little bit complex to implement because now more classes are involved in our application and those classes need to be unit tested. Also, we have to start with factory method design pattern from the start. It is not quite easy to refactor your existing code because you have this entire hierarchy of creators when implementing the factory method design pattern. Lastly, sometimes you will find yourself in a situation where you have to subclass(I mean you have create a subclass creator) just so you can create an object of the newly added product class. And this can be a pitfall where you end up creating subclasses that match with your product hierarchy. So these are some of the pitfalls of using the factory method design pattern.

A screenshot of a computer

Description automatically generated

1. Factory Method – Summary:

Now let's look at the summary of factory method design pattern. We will use this pattern whenever we want to delegate the object instantiation to subclasses. And the reason we want to do that is because we may have a product inheritance hierarchy. Here, product refers to the classes whose objects we want to create using our factory method. So if there is an inheritance hierarchy, then there is a chance that in future another type of class can be added to that hierarchy. And we want to handle creating objects of that new class without modifying our existing code. And we can easily do that using our factory method. So this is the UML diagram of factory method design pattern. On the left hand side, you will see that we have a inheritance hierarchy that is related to product. So again, product refers to the class whose objects that we want to create. So in this diagram, we have a Product class and there are two child classes of Product, ConcreteProductB and ConcreteProductA. Now there is a chance that, in the future we may add another class called as ConcreteProductC, and we want to ensure that adding a new class won't affect or won't need changing our existing code. On the right hand side, we have a creator hierarchy. So Creator is a class which is in-charge of creating objects of our product class and Creator defines a factory method. Now this factory method is overridden or implemented by all the subclasses of our Creator. Now each subclass of Creator creates one type of object. So in this UML, we have a ConcreteCreatorA which creates objects of ConcreteProductA and ConcreteCreatorB creates instances of ConcreteProductB. So you can see that the creator inheritance hierarchy and the product inheritance hierarchy mirror each other. Now, whenever a new ConcreteProduct.. is added in the product inheritance hierarchy, we can easily add a new subclass of our Creator which will handle creating object of that new class. And this is the benefit of using factory method design pattern.

A screenshot of a computer

Description automatically generated

Now let's look at implementation of the factory method design pattern. To start off here, we have a product class, which is Message, and there are multiple subclasses or child classes of this Message class. So we have a TextMessage, JSONMessage, and in future we can have, let's say an HTMLMessage class. So here we have a clear inheritance hierarchy between the classes whose objects we want to create.

A screenshot of a computer

Description automatically generated

Next, we have the creator. Now this is the class which defines the factory method. So, at the bottom you will see we have a ‘protected abstract Message createMessage()’. So this is our factory method. And this method will be implemented in subclasses of our MessageCreator. So here we have JSONMessageCreator which implements our createMessage() method and creates object of JSONMessage class. On the similar lines, we have TextMessageCreator which creates object of TextMessage. One thing you will often find when studying existing factory method implementation is that the creator(i.e., MessageCreator in our case) also performs some additional operations using the object that is returned by factory method. So, for example, we have a method here called as getMessage(), which is actually used by the client. So our createMessage() method here is a protected method. So we want clients to use our getMessage() method. So this getMessage() method is getting an object from one of its subclasses and then performing some additional operations on it before returning the object to client. Now this is not mandatory, but this is how typically you will find the implementations of factory method design pattern.

A screenshot of a computer

Description automatically generated