# Lesson 5 Factory Method

## The Factory Method Pattern

1. Intent

   Define an interface for creating an object, but let subclasses (concrete class that implements it) decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

2. Motivation

   a. Sometimes an object may only know that it needs an object of a certain type but does not know exactly which one from the set of subclasses of the parent class is to be selected.

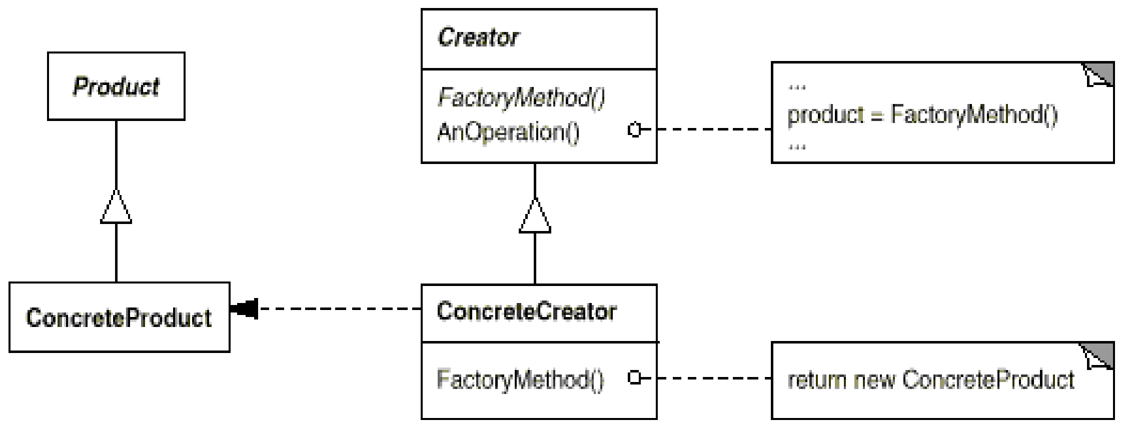
   b. The choice of an appropriate class may depend on factors such as:
   – The state of the running application.
   – Application configuration settings.
   – Expansion of requirements or enhancements.

   c. Factory Method recommends encapsulating the functionality required, to select and instantiate an appropriate class, inside a designated method referred to as a factory method.

   d. A factory method can be defined as a method in a class that:
   – Selects an appropriate class from a class hierarchy based on the application context and other influencing factors.

– Instantiates the selected class and returns it as an instance of the parent class type.

## 3. Structure



## 4. Participants

a. Product
- defines the interface of objects the factory method creates.

b. ConcreteProduct
- implements the Product interface.

c. Creator
- declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- may call the factory method to create a Product object.

d. ConcreteCreator
- overrides the factory method to return an instance of a ConcreteProduct.

## 5. Applicability

Use the Factory Method pattern when

a. A class can't anticipate the class of objects it must create.
b. A class wants its subclasses to specify the objects it creates.
c. Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

## 6. Consequences

a. Factory methods eliminate the need to bind application-specific classes into your client code. The client code only deals with the Product interface; therefore it can work with any user-defined concrete product classes.
b. Any change in a concrete product class does not have any impact on the client code.

## 7. How to implement the Factory Method Pattern

a. Factory interface (or an abstract class)

```java
public interface PizzaFactory {
	public Pizza createPizza(String type);
}
```

b. Concrete factory implementation

```java
public class SimplePizzaFactory implements PizzaFactory{
	//This factory should often times be a Singleton
	private static PizzaFactory factory = new SimplePizzaFactory();

	private SimplePizzaFactory(){}
```

```
        public static PizzaFactory getFactory(){
              return factory;
        }

        public Pizza createPizza(String type) {
              Pizza pizza = null;

              if (type.equals("cheese")) {
                    pizza = new CheesePizza();
              } else if (type.equals("pepperoni")) {
                    pizza = new PepperoniPizza();
              } else if (type.equals("clam")) {
                    pizza = new ClamPizza();
              } else if (type.equals("veggie")) {
                    pizza = new VeggiePizza();
              }
              return pizza;
        }
  }
```

## c. Your product class hierarchy – Pizza superclass and its subclasses, like CheesePizza, PepperoniPizza, etc.

```
abstract public class Pizza {
      String name;
      String dough;
      String sauce;
//…
```

## d. Client

```
public class FactoryMethodClient {

   public static void main(String[] args) {
         PizzaFactory factory = SimplePizzaFactory.getFactory();
         Pizza pizza = factory.createPizza("cheese");

         //different from below on when the decision has to be made -
         runtime or compile time?
         //which is one difference between framework and application
         development
         Pizza pizza1 = new CheesePizza();
         System.out.println(pizza.getClass().getSimpleName());
   }

}
```

## Lab 5-1

Suppose you are writing a simple debug tool for your colleagues. To start with, you are going to provide 2 implementations - one writes the messages out to the command line (ConsoleTrace), while another writes them to a file (FileTrace). All you want your colleagues to know about the tool is

1) The interface (or what my tool can do for you)

```
public interface Trace {
    // turn on and off debugging
    public void setDebug( boolean debug );
    // write out a debug message
    public void debug( String message );
    // write out an error message
    public void error( String message );
}
```

2) How to choose a certain debugger by giving an argument to your main(String args[]) method. (for example, use “trace.log” to choose the FileTrace implementation or “console” for the ConsoleTrace debugger.

You may want to add more implementations later as they need it. But they won't have to change any code to switch from one implementation to another. Design/write your debugging tool in Java.