What is a Creational Pattern?

Creational Patterns are concerned with object creation problems faced during software design. Object creation often results in design problems, creational patterns solve this problem by controlling the object creation.

Factory pattern

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as Virtual Constructor.

- A Factory returns an instance of an object based on the data supplied to it.
- The instance returned can be one of many classes that extend a common parent class or interface. ("Animal" as a parent class, then "Dog", "Cat", "Zebra" as child classes.)
- Create objects without exposing their instantiation logic.
- Consequences: The requestor is independent of the concrete object that is created (how that object is created, and which class is actually created).

Advantage of Factory Design Pattern

- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

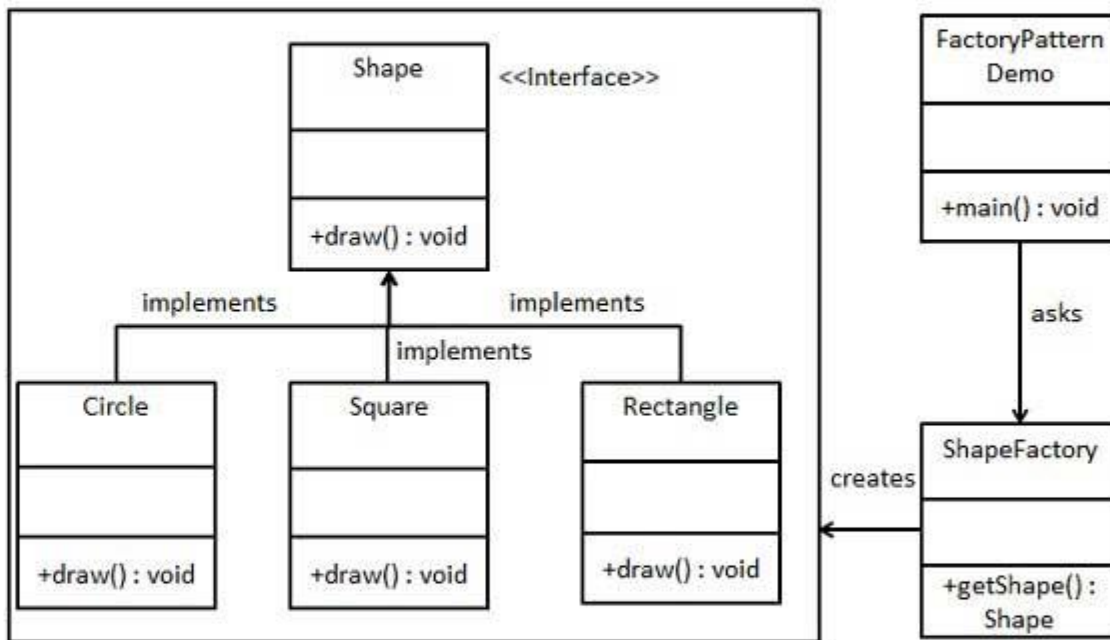
Usage of Factory Design Pattern

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.

- When the parent classes choose the creation of objects to its sub-classes.

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape*object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



Step 1

Create an interface.

Shape.java

```
public interface Shape {

   void draw();

}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {


   //use getShape method to get object of type shape

   public Shape getShape(String shapeType){

      if(shapeType == null){

         return null;

      }

      if(shapeType.equalsIgnoreCase("CIRCLE")){

         return new Circle();


      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){

         return new Rectangle();


      } else if(shapeType.equalsIgnoreCase("SQUARE")){

         return new Square();

      }


      return null;

   }

}
```

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");
      //call draw method of circle
      shape3.draw();
   }
}
```

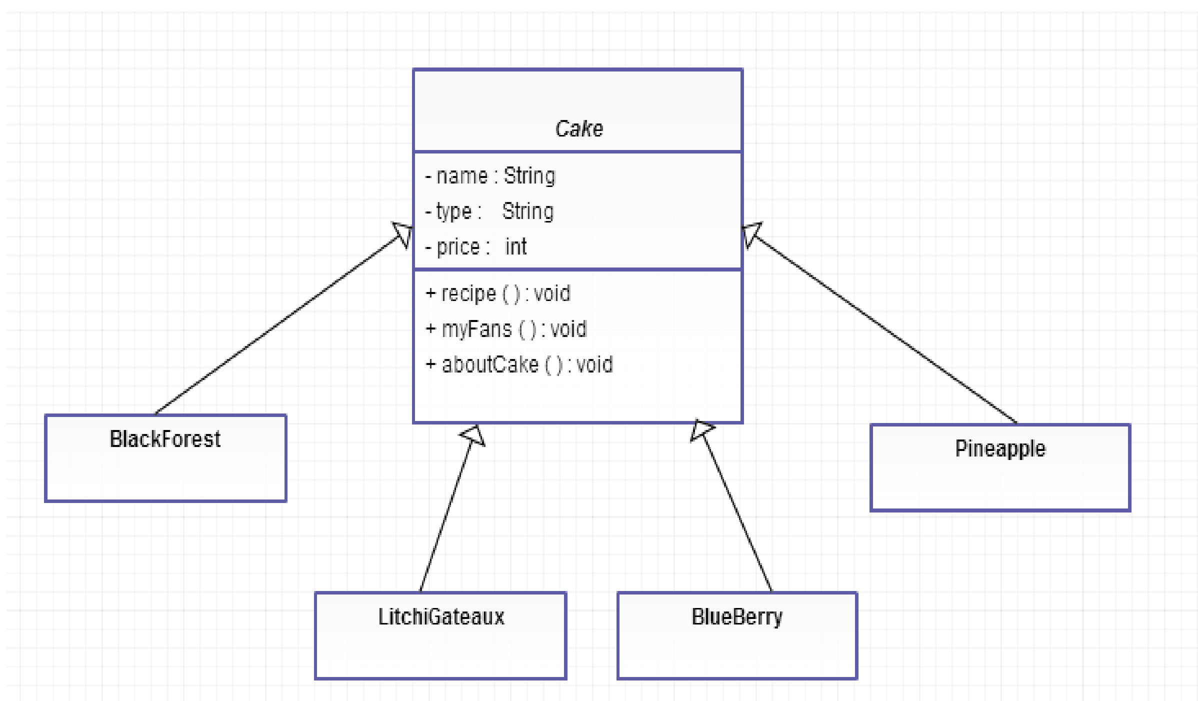
Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Another example in Factory pattern. Let`s understand the problem first then we will solve the problem using Factory Pattern.

Consider the below UML diagram , where we have cake abstract class and concrete sub-classes BlackForest, LitchiGateaux, BlueBerry, Pineapple.



Let's see what goes inside these classes. Cake is the abstract class, all different cake classes inherit from this class. Cake class has three properties; name of the cake, type of the cake (whether cake contains egg or it is egg less) and the last one is price.

Cake.Java

```
public abstract class Cake {

	String name;
	String type;
	int price;

	public String getName(){
		return name;
	}
	public String getType(){
		return type;
	}
	public int getPrice(){
		return price;
	}
	public void setName(String name){
		this.name=name;
	}
	public void setPrice(int price){
		this.price=price;
	}
	public void setType(String type){
		this.type=type;
	}

	public abstract void recipe();
	public abstract void myFans();
	public void aboutCake(){
```

```
                System.out.println("I am "+name+" Cake");
                System.out.print("My Fans  : ");
                myFans();
                System.out.println("You can get a "+name+" Cake at "+price);
        }
}
```

Let's see some of the sub-classes of cake.

BlackForest.Java

```
public class BlackForest extends Cake {
        public BlackForest(){
                setName("Black Forest");
                setType("Eggless");
                setPrice(800);
        }
        public void recipe() {
           System.out.println("---BlackForest Recipe---");
           System.out.println("Sieve together Maida and Cocoa powder");
           System.out.println("Add Milk and Vanilla essence");
           System.out.println("Top with Whipped Cream and Cherries");
           System.out.println("Chill and Serve");
        }
        public void myFans() {
                System.out.println("Both adults and Kids love me");
        }
}
```

BlueBerry.Java

```
public class BlueBerry extends Cake {
	BlueBerry(){
		setName("Blue Berry");
		setType("Egg");
		setPrice(700);
	}
	public void recipe() {
	System.out.println("---BlueBerry Recipe---");
	System.out.println("First prepare Flour and Baking powder mixture");
	System.out.println("Add Milk and Egg yolks");
	System.out.println("Coat Berries");
	System.out.println("Bake for 50 minutes");
	}
	public void myFans() {
		 System.out.println("Moms love me");
	}
 }
```

LitchiGateaux.Java

```
public class LitchiGateaux extends Cake {
	LitchiGateaux(){
```

```
                setName("Litchi Gateaux");
                setType("Eggless");
                setPrice(750);
        }
        public void recipe() {
                System.out.println("---LitchiGateaux Recipe---");
                System.out.println("Take some fresh Litchies");
                System.out.println("Wash them and Grind for 5 minutes");
                System.out.println("Put some groundnuts and bake for 45 minutes");
        }
        public void myFans() {
    System.out.println("Litchi lovers love me");
        }
}
```

Now we are all set to create different types of cakes.

CakeTest.Java

```
import java.util.Scanner;
public class CakeTest {
        public static void main(String args[]) {
                Cake cake = null;
                System.out.println("Which Cake you would like to eat ");
                Scanner scanner = new Scanner(System.in);
                String choice = scanner.nextLine();
                scanner.close();
                if (choice.equals("BlackForest")) {
                        cake = new BlackForest();
                }
                else if (choice.equals("BlueBerry")) {
                        cake = new BlueBerry();
```

```
                }
                else if (choice.equals("LitchiGateaux")) {
                        cake = new LitchiGateaux();
                }
                else if(choice.equals("Pineapple")){
                        cake=new Pineapple();
                }
                cake.aboutCake();
        }
}
```

Notice the code in CakeTest class, we have got several concrete classes BlackForest, BlueBerry, LitchiGateaux, Pineapple being instantiated, and the decision of which class to instantiate is made at run-time depending on the user's choice.

When you see code like the above and when the time comes for changes or extension, you will have to reopen the code and examine what needs to be added or deleted. Suppose I don't want to offer pineapple cake, I have to reopen the code and delete some codes. Similarly, if later I decide to add 10 new cake types, I have to reopen the code and type 10 more "else if" statements. Which means your code is not "closed for modifications".

<u>Note</u> : **Your design should always be "open for extension" but "closed for modifications".**

The code in the CakeTest will certainly vary as we decide to delete some cake type or add new cake types.

```
        if (choice.equals("BlackForest")) {
                        cake = new BlackForest();
                }
                else if (choice.equals("BlueBerry")) {
                        cake = new BlueBerry();
                }
```

```
else if (choice.equals("LitchiGateaux")) {
        cake = new LitchiGateaux();
}
else if(choice.equals("Pineapple")){
        cake=new Pineapple();
}
```

There are three big problems with our current design

1. CakeTest class is tightly coupled to cake subclasses, as in the main method we are instantiating one concrete cake class depending on the user's selection. Which means CakeTest class have knowledge of all sub-classes of cake, which are not a good design.
2. If we add or delete some cake types we have to modify the code in the main method.
3. There is no code reusability if some other classes also require a particular cake depending on some condition. We have to duplicate the code, which is bad. Wouldn't it be great if we separate the cake creation code to a separate factory class ?

So, anyone that needs a cake object can directly call factory class.

Now that we have understood the problem in our design, let's separate the code that vary.

Encapsulating Object Creation

We are going to define a factory interface and a concrete factory class (CakeFactory) that implements factory interface which will encapsulate the object creation code for different types of cakes.

Factory.Java

```
public interface Factory {
        Cake createCake(String cakeName);
}
```

CakeFactory.Java

```
public class CakeFactory implements Factory{


	public Cake createCake(String cakeName){
		Cake cake=null;
		if (cakeName.equals("BlackForest")) {
			cake = new BlackForest();
		}
		else if (cakeName.equals("BlueBerry")) {
			cake = new BlueBerry();
		}
		else if (cakeName.equals("LitchiGateaux")) {
			cake = new LitchiGateaux();
		}
		else if(cakeName.equals("Pineapple")){
			cake=new Pineapple();
		}
		return cake;
	}
}
```

Now, let’s see how our CakeTest class is going to change.

CakeTest.Java

```
import java.util.Scanner;
public class CakeTest {
	public static void main(String args[]) {
		Cake cake = null;
		System.out.println("Which Cake you would like to eat
BlackForest/BlueBerry/LitchiGateaux/Pineapple : ");
		Scanner scanner = new Scanner(System.in);
```

```
        String choice = scanner.nextLine();
        scanner.close();
        CakeFactory cakeFactory=new CakeFactory();
        cake=cakeFactory.createCake(choice);
        cake.aboutCake();
    }
}
```

By separating the cake creation code in a separate factory class, we get following benefits:

1. Our object creation code is at one place and any class that requires a particular cake type can use factory class. We need not duplicate object creation code at all the places.
2. The classes that requires particular cake object do not require to have knowledge of all cake types. All they need to do is call the method in the factory class passing the argument.
3. Now, our code is dynamic because we can swap in and out different factory implementations. Let's see how to do that.