Design Pattern:

2.Factory Design Pattern:

(i).It is a creational design pattern.

(ii).it is a way to create object without using the new Keyword directly in the main code.

(iii).Factory class decides which object to create based on given input.

Why:(Advantage)

(i).Loose Coupling

(ii).Hides Object Creation

(iii).Less Code Changes

(iv).Easy to maintain

(v).Reusable

Example:

Step:1

(i).Create Interface:

*Shape.java*

public interface Shape {

void draw();

}

Step:2

Create class implement 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:

*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:

*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 square

shape3.draw();

}

}

**3.Abstract Factory Pattern :**

The **Abstract Factory Pattern** is a design pattern that helps create groups of related objects **without specifying their exact types**. It allows the client to use objects **without worrying about how they are created**.

**Advantages of Abstract Factory Pattern (Simple Explanation)**

✅ **1. Hides Object Creation**

✅ **2. Supports Dependency Injection**

✅ **3. Promotes Loose Coupling**

**Why Do We Use Abstract Factory?**

👉 **To create groups of related objects** without worrying about their exact types.  
👉 **To make code flexible** so we can add new versions without changing existing code.  
👉 **To ensure objects work well together** (e.g., UI components for the same theme).

Step 1:

Create an interface for Shapes.

*Shape.java*

public interface Shape {

void draw();

}

Step 2:

Create concrete classes implementing the same interface.

*RoundedRectangle.java*

public class RoundedRectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside RoundedRectangle::draw() method.");

}

}

*RoundedSquare.java*

public class RoundedSquare implements Shape {

@Override

public void draw() {

System.out.println("Inside RoundedSquare::draw() method.");

}

}

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

Step 3

Create an Abstract class to get factories for Normal and Rounded Shape Objects.

*AbstractFactory.java*

public abstract class AbstractFactory {

abstract Shape getShape(String shapeType) ;

}

Step 4

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

*ShapeFactory.java*

public class ShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

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

return new Rectangle();

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

return new Square();

}

return null;

}

}

*RoundedShapeFactory.java*

public class RoundedShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

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

return new RoundedRectangle();

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

return new RoundedSquare();

}

return null;

}

}

Step 5

Create a Factory generator/producer class to get factories by passing an information such as Shape

*FactoryProducer.java*

public class FactoryProducer {

public static AbstractFactory getFactory(boolean rounded){

if(rounded){

return new RoundedShapeFactory();

}else{

return new ShapeFactory();

}

}

}

Step 6

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

*AbstractFactoryPatternDemo.java*

public class AbstractFactoryPatternDemo {

public static void main(String[] args) {

//get shape factory

AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

//get an object of Shape Rectangle

Shape shape1 = shapeFactory.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape1.draw();

//get an object of Shape Square

Shape shape2 = shapeFactory.getShape("SQUARE");

//call draw method of Shape Square

shape2.draw();

//get shape factory

AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);

//get an object of Shape Rectangle

Shape shape3 = shapeFactory1.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape3.draw();

//get an object of Shape Square

Shape shape4 = shapeFactory1.getShape("SQUARE");

//call draw method of Shape Square

shape4.draw();

}

}

Step 7:

Verify the output.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside RoundedRectangle::draw() method.

Inside RoundedSquare::draw() method.