Factory

Resources

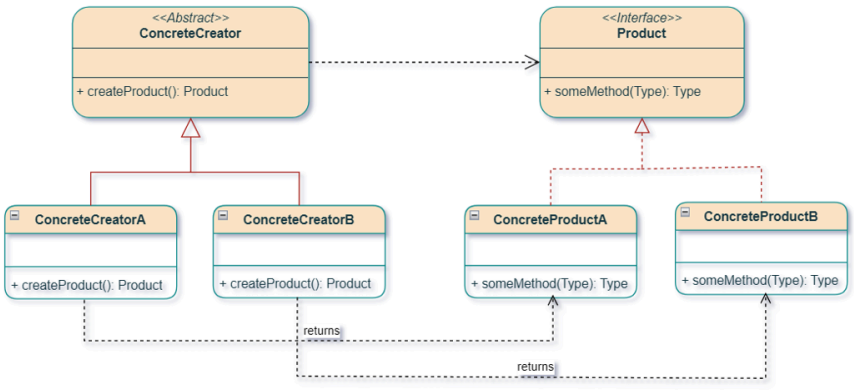
- Video → 27. All Creational Design Patterns | Prototype, Singleton, Factory, AbstractFactory, Builder Pattern
- Video → 5. Factory Pattern Vs Abstract Factory Pattern Explanation (Hindi) | Low Level System Design

Definition

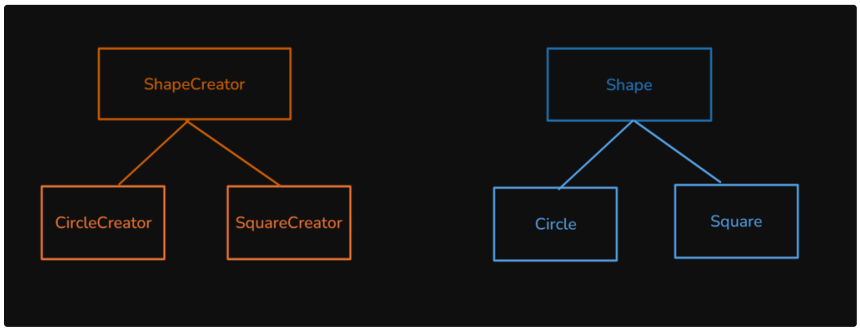
> ***The Factory Method Pattern is used when we want to encapsulate object creation, instantiation, and all related business logic in one place.***

This approach offers an interface for creating objects without needing to specify their exact classes. It delegates the responsibility of object creation to subclasses, which implement specific functionalities. Instead of using the `"new"` keyword directly to create objects, we utilize a factory method that returns objects adhering to a common interface. The actual class that gets instantiated is determined at runtime. This pattern encourages loose coupling and enhances extensibility.

Class Diagram



Example: Shape



1. **Product (Shape)**

- Defines the interface for objects that the factory method creates
- All concrete shapes implement this interface

2. **Concrete Products (Circle, Square, Rectangle, Triangle)**

- Specific implementations of the Shape interface
- Each has its own drawing logic and area calculation

3. **Creator (ShapeFactory)**

- Abstract class that declares the factory method
- May provide default implementation or template methods

4. **Concrete Creators (CircleCreator, RectangleCreator, etc.)**

- Override the factory method to return specific product instances
- Each factory knows how to create one type of shape

Implementation

```java
// Step 1: Define the Product interface
public interface Shape {
    void computeArea();
    void draw();
}

// Step 2: Concrete Product classes
public class Circle implements Shape {

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

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
public class Rectangle implements Shape {

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

    @Override
    public void draw() {
```

```
        System.out.println("Inside Rectangle::draw() method.");
    }
}
public class Square implements Shape {

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

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

Simple Factory

```
public enum ShapeType {
    CIRCLE, RECTANGLE, SQUARE
}
public class ShapeFactory {
    public static Shape createShapeInstance(ShapeType shapeType) {
        if (shapeType == null) {
            return null;
        }
        return switch (shapeType) {
            case CIRCLE -> new Circle();
            case RECTANGLE -> new Rectangle();
            case SQUARE -> new Square();
            default -> throw new IllegalStateException("ShapeType
doesn't exist!");
        };
    }
}
```

```
// Step 3: Simple Factory Demo (Bloated Design)
public class SimpleFactoryDemo {
    public static void main(String[] args) {
        System.out.println("======= Simple Factory Design Pattern
======");

        // set the type you want
        ShapeType shapeType = ShapeType.SQUARE;
        // get the shape
        Shape shape = ShapeFactory.createShapeInstance(shapeType);
        shape.draw();
        shape.computeArea();
    }
}
```

Factory Method

```
// Step 3: Abstract Creator class
public abstract class ShapeFactory {
    // Factory method - to be implemented by subclasses
    public abstract Shape createShape();
}

// Step 4: Concrete Creator classes
public class CircleCreator extends ShapeFactory {

    @Override
    public Shape getShapeInstance() {
        return new Circle();
    }
}
public class RectangleCreator extends ShapeFactory {

```

```
    @Override
    public Shape getShapeInstance() {
        return new Rectangle();
    }
}
public class SquareCreator extends ShapeFactory {
    @Override
    public Shape getShapeInstance() {
        return new Square();
    }
}
```

```
// Step 5: Client code demonstration
public class FactoryMethodDemo {

    public static void main(String[] args) {
        System.out.println("======= Factory Method Design Pattern ======");
        // set the type you want
        ShapeType shapeType = ShapeType.SQUARE;
        // get the shape
        Shape shape = getShapeInstance(shapeType);
        // perform operations
        shape.draw();
        shape.computeArea();
    }

    private static Shape getShapeInstance(ShapeType shapeType) {
        Shape shape = null;
        if (shapeType == null) {
            return null;
        }
        switch (shapeType) {
            case CIRCLE:
                ShapeFactory circleCreator = new CircleCreator();
                shape = circleCreator.createShape();
                break;
            case RECTANGLE:
                ShapeFactory rectangleCreator = new RectangleCreator();
                shape = rectangleCreator.createShape();
                break;
            case SQUARE:
                ShapeFactory squareCreator = new SquareCreator();
                shape = squareCreator.createShape();
                break;
            default:
                throw new IllegalStateException("ShapeType doesn't exist.");
        }
        return shape;
    }
}
```

Factory Method vs Simple Factory

Simple Factory (shown in the example):

- Static method that creates objects based on parameters
- Not a true design pattern, more of a programming idiom
- Violates the Open/Closed principle when adding new types

Factory Method Pattern (shown in the example):

- Uses inheritance and polymorphism
- Each concrete creator handles one product type

- Follows the Open/Closed principle perfectly

> The Factory Method Pattern is especially useful in frameworks and libraries that require extension points for users to customize object creation behavior.