Structural Design Pattern:

LLD: All Structural Design Patterns

Friday, 25 August 2023 3:17 PM

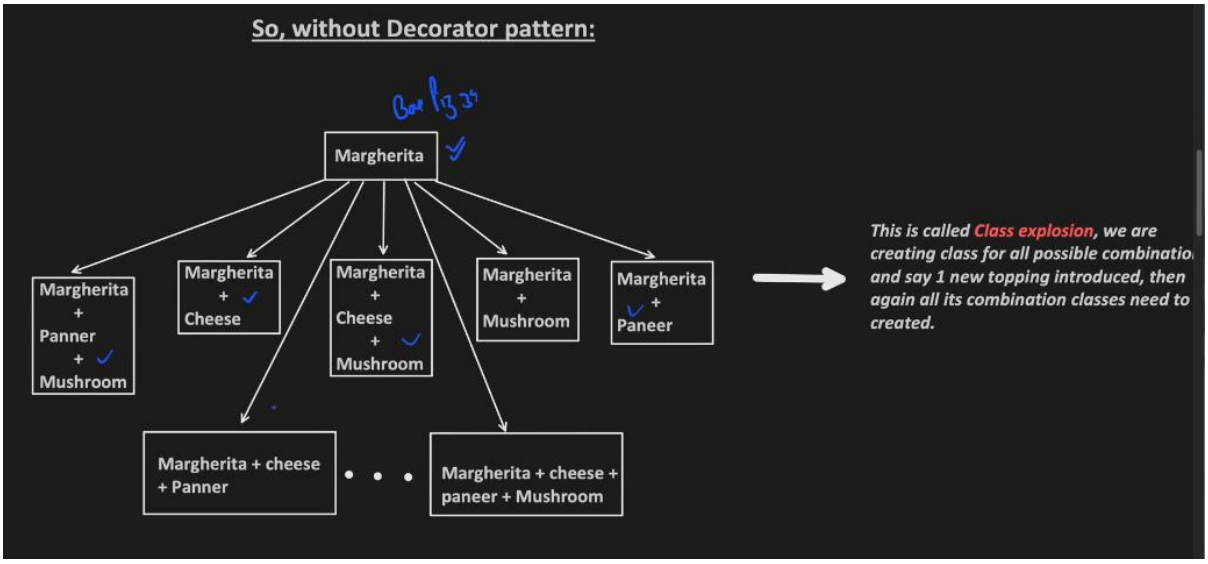
Structural Design Pattern is a way to combine or arrange different classes and objects to form a complex or bigger structure to solve a particular requirement.

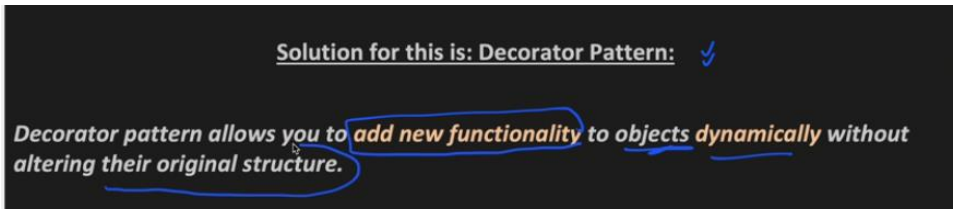
Types:

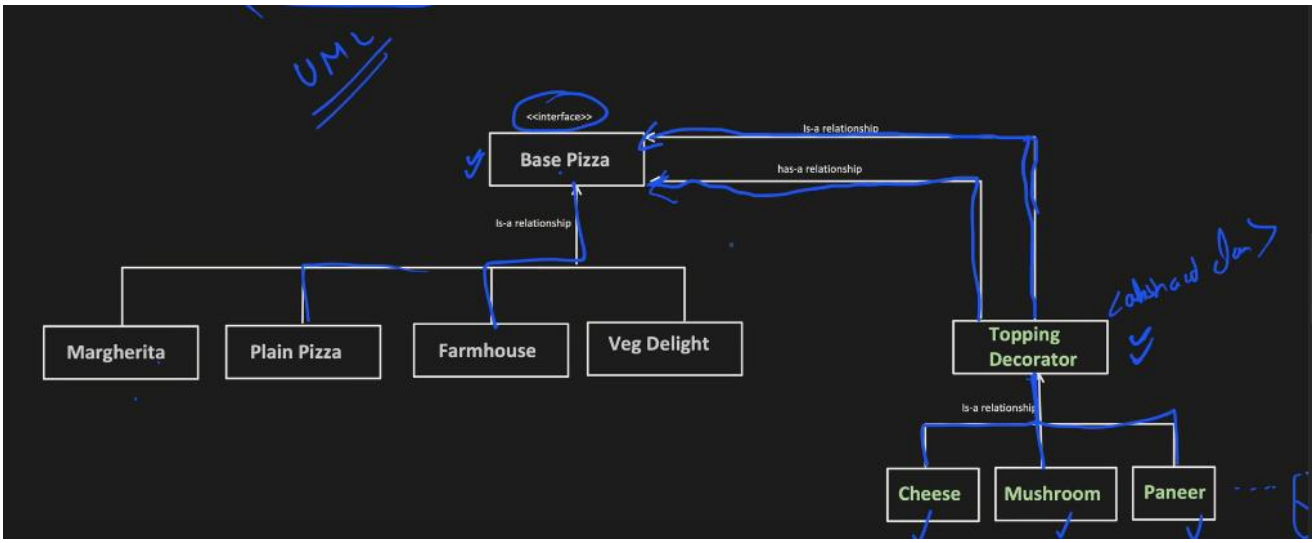
---

1. Decorator Pattern
2. Proxy Pattern
3. Composite Pattern
4. Adapter Pattern
5. Bridge Pattern
6. Facade
7. Flyweight

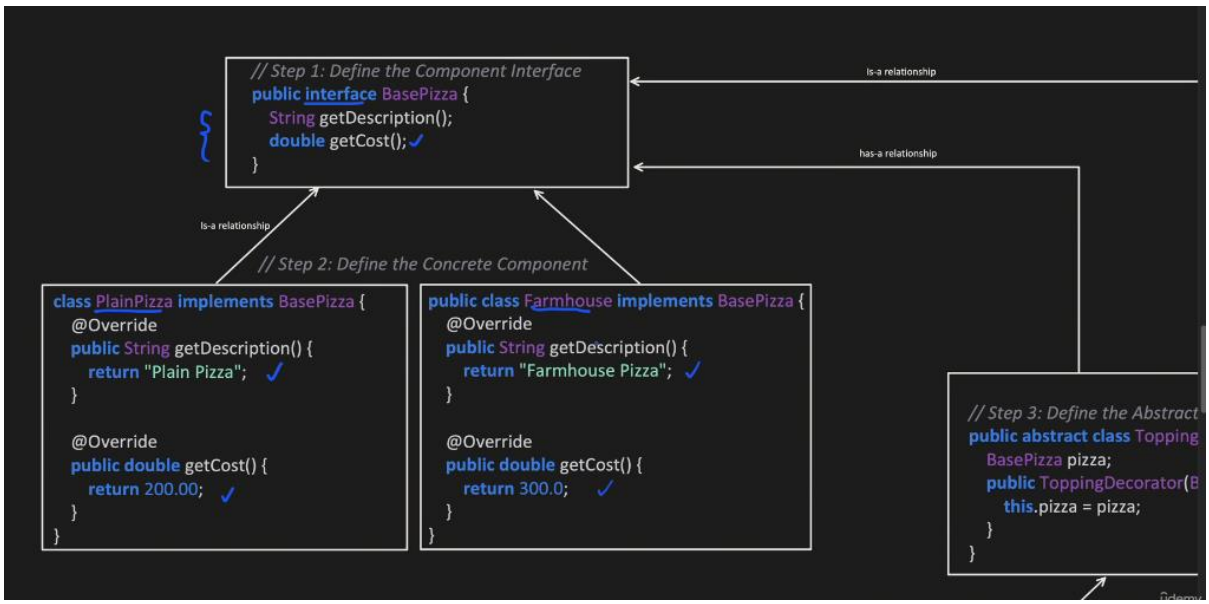
6.Decorator Design Pattern:

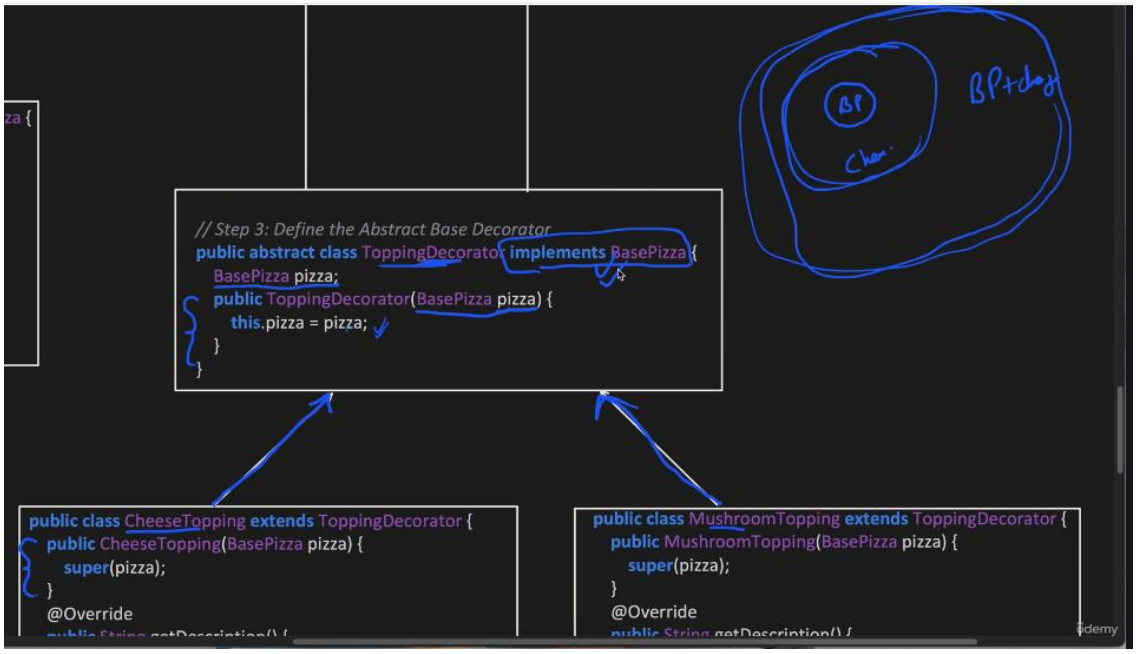

So, without Decorator pattern:
Margherita
Margherita + Panner + Mushroom
Margherita + Cheese
Margherita + Cheese + Mushroom
Margherita + Mushroom
Margherita + Paneer
Margherita + cheese + Panner
Margherita + cheese + paneer + Mushroom
This is called Class explosion, we are creating class for all possible combinatio and say 1 new topping introduced, then again all its combination classes need to created.

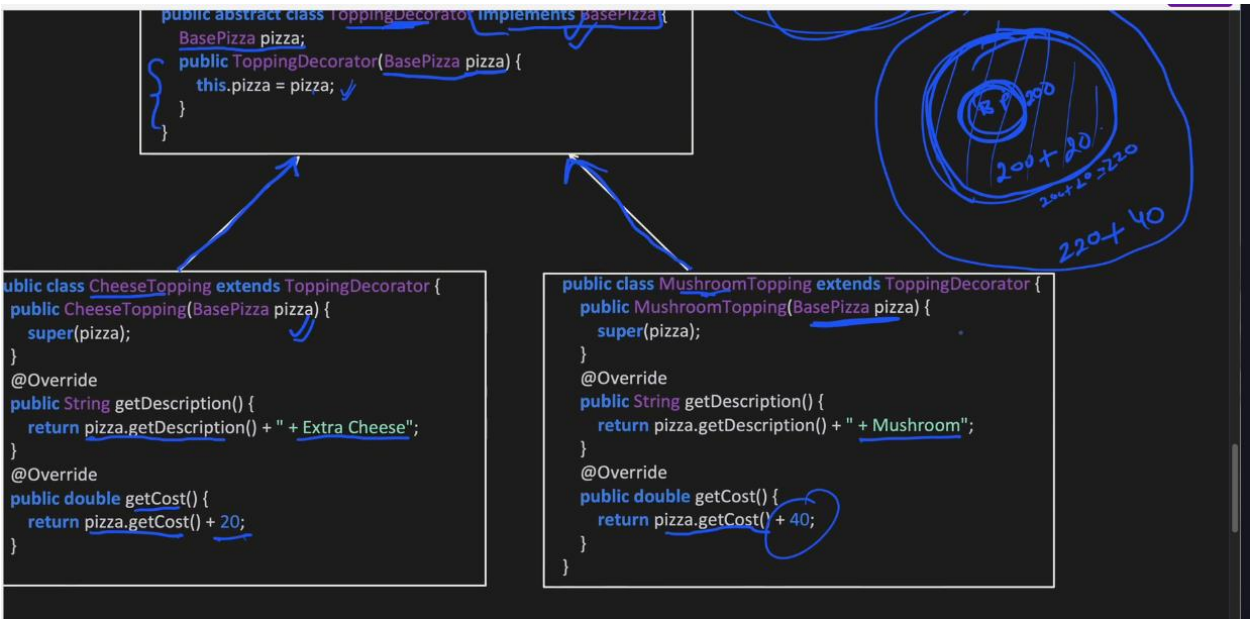

Solution for this is: Decorator Pattern:
Decorator pattern allows you to add new functionality to objects dynamically without altering their original structure.


UML
<<interface>>
Base Pizza
Is-a relationship
has-a relationship
Is-a relationship
Margherita
Plain Pizza
Farmhouse
Veg Delight
Topping Decorator
Is-a relationship
Cheese
Mushroom
Paneer

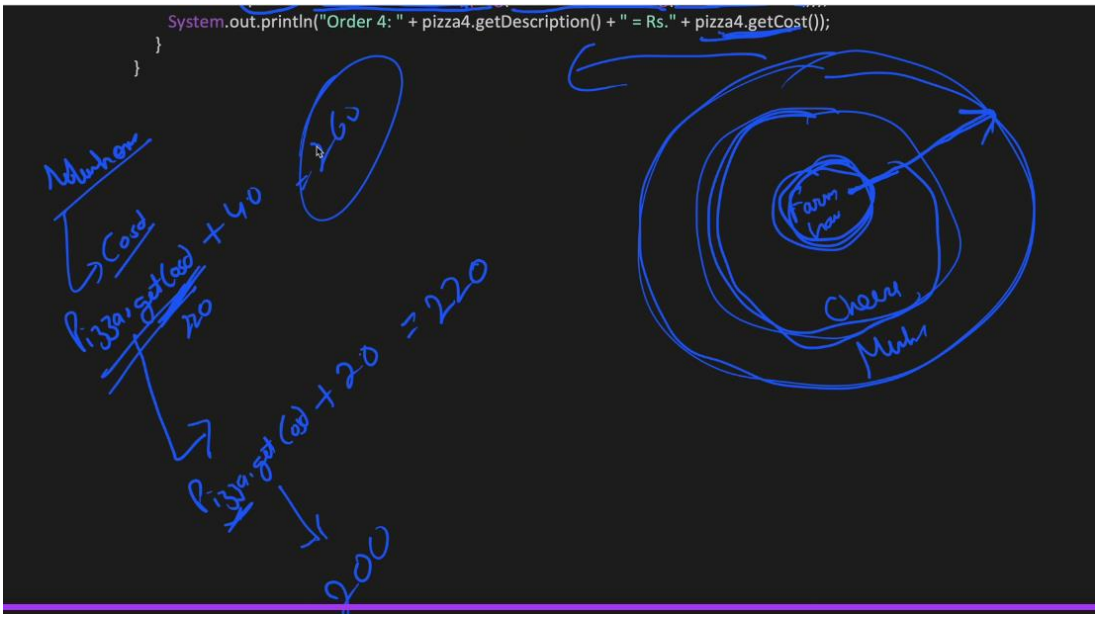
Example :


// Step 1: Define the Component Interface
public interface BasePizza {
String getDescription();
double getCost();
}
Is-a relationship
has-a relationship
Is-a relationship
// Step 2: Define the Concrete Component
class PlainPizza implements BasePizza {
@Override
public String getDescription() {
return "Plain Pizza";
}
@Override
public double getCost() {
return 200.00;
}
}
public class Farmhouse implements BasePizza {
@Override
public String getDescription() {
return "Farmhouse Pizza";
}
@Override
public double getCost() {
return 300.0;
}
}
// Step 3: Define the Abstract
public abstract class Topping
BasePizza pizza;
public ToppingDecorator(
this.pizza = pizza;
}
}


// Step 3: Define the Abstract Base Decorator
public abstract class ToppingDecorator implements BasePizza {
BasePizza pizza;
public ToppingDecorator(BasePizza pizza) {
this.pizza = pizza;
}
}
public class CheeseTopping extends ToppingDecorator {
public CheeseTopping(BasePizza pizza) {
super(pizza);
}
@Override
public class MushroomTopping extends ToppingDecorator {
public MushroomTopping(BasePizza pizza) {
super(pizza);
}
@Override

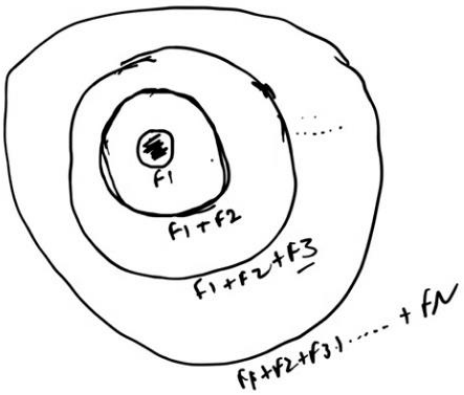

public abstract class ToppingDecorator implements BasePizza {
BasePizza pizza;
public ToppingDecorator(BasePizza pizza) {
this.pizza = pizza;
}
}
ublic class CheeseTopping extends ToppingDecorator {
public CheeseTopping(BasePizza pizza) {
super(pizza);
}
@Override
public String getDescription() {
return pizza.getDescription() + " + Extra Cheese";
}
@Override
public double getCost() {
return pizza.getCost() + 20;
}
public class MushroomTopping extends ToppingDecorator {
public MushroomTopping(BasePizza pizza) {
super(pizza);
}
@Override
public String getDescription() {
return pizza.getDescription() + " + Mushroom";
}
@Override
public double getCost() {
return pizza.getCost() + 40;
}
}

```java
// Step 5: Client Demonstration
public class PizzaShop {

    public static void main(String[] args) {

        // Create a plain pizza
        BasePizza pizza1 = new PlainPizza();

        System.out.println("Order 1: " + pizza1.getDescription() + " = Rs." + pizza1.getCost());

        // Add toppings to the PlainPizza - Extra Cheese Only
        BasePizza pizza2 = new CheeseTopping(new PlainPizza());
        System.out.println("Order 2: " + pizza2.getDescription() + " = Rs." + pizza2.getCost());


        // Farmhouse Pizza
        BasePizza pizza3 = new Farmhouse();
        System.out.println("Order 3: " + pizza3.getDescription() + " = Rs." + pizza3.getCost());

        // Farmhouse Pizza with Extra Cheese and Mushroom
        BasePizza pizza4 = new MushroomTopping(new CheeseTopping(new Farmhouse()));
        System.out.println("Order 4: " + pizza4.getDescription() + " = Rs." + pizza4.getCost());
    }
}
```



Decorator Pattern - Low Level design #3

Thursday, 12 May 2022 8:49 AM



USE CASE

PIZZA SHOP

Coffee

Toppings

Decaf Espresso

→ Cream

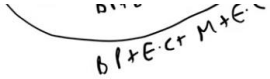
→ Extra Milk

Car

Toppings

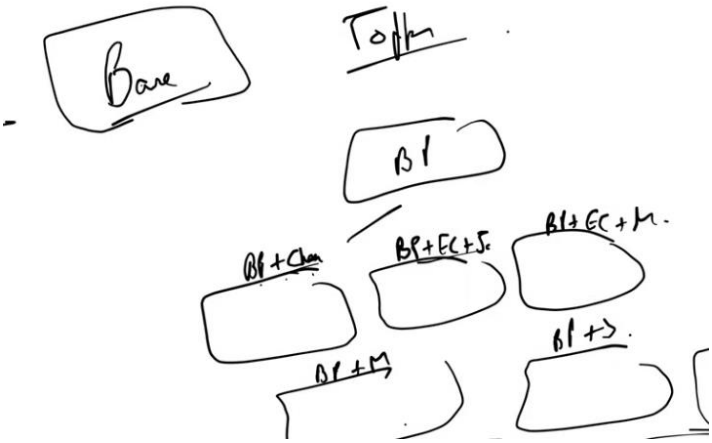
Seat Cover

- A.C
- Power Steering
- Fog Light

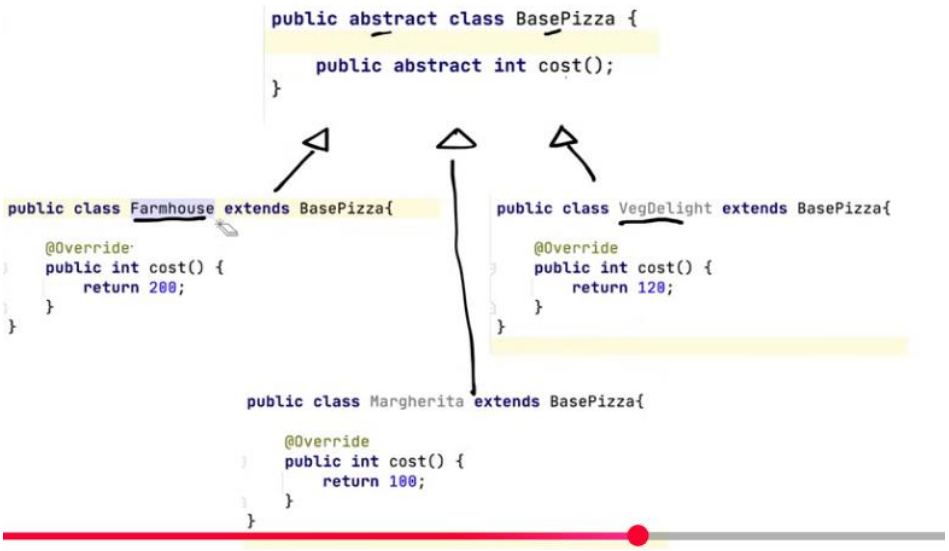


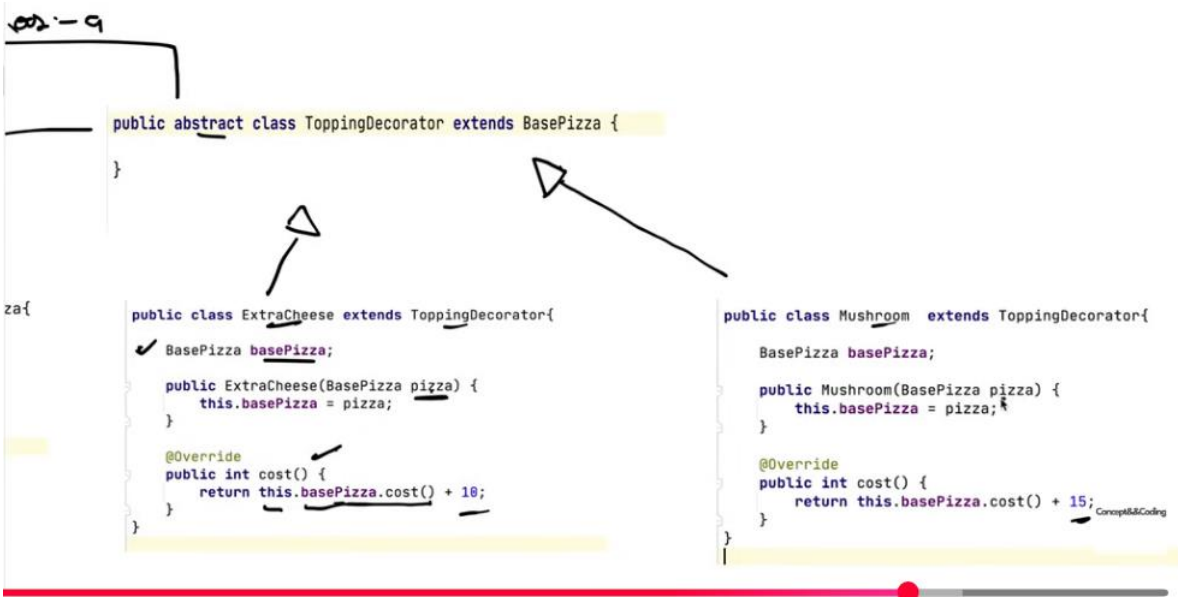
Why you need decorator pattern

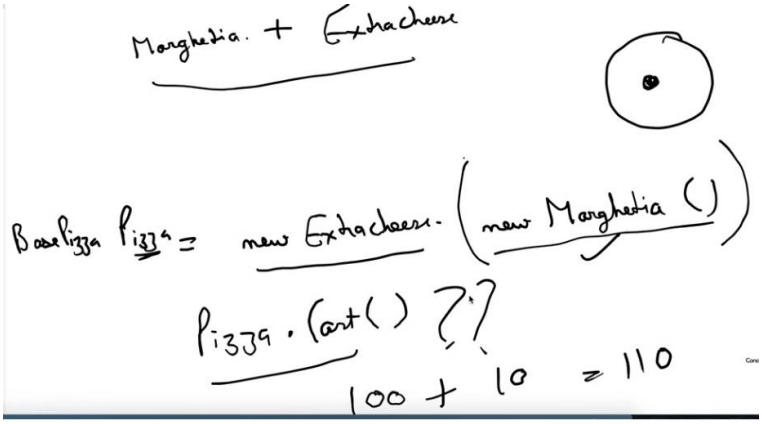
Class Explosion



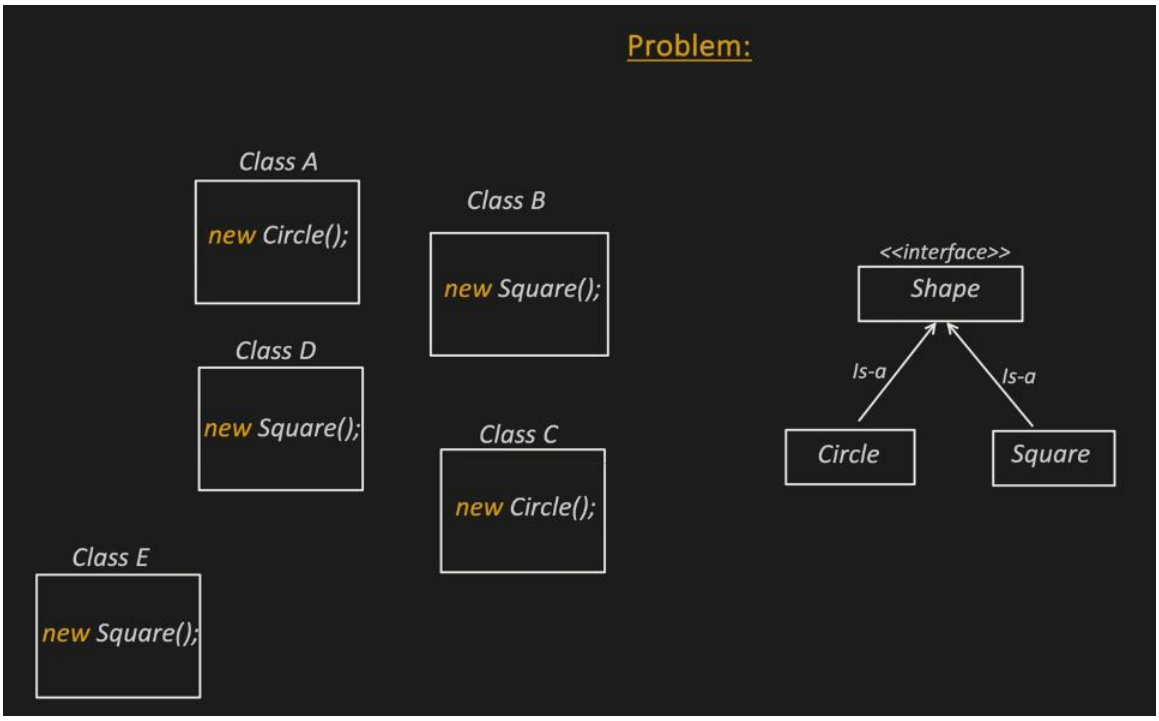
Example :


public abstract class BasePizza {
public abstract int cost();
}
public class Farmhouse extends BasePizza{
@Override
public int cost() {
return 200;
}
}
public class VegDelight extends BasePizza{
@Override
public int cost() {
return 120;
}
}
public class Margherita extends BasePizza{
@Override
public int cost() {
return 100;
}
}


public abstract class ToppingDecorator extends BasePizza {
}
public class ExtraCheese extends ToppingDecorator{
BasePizza basePizza;
public ExtraCheese(BasePizza pizza) {
this.basePizza = pizza;
}
@Override
public int cost() {
return this.basePizza.cost() + 10;
}
}
public class Mushroom extends ToppingDecorator{
BasePizza basePizza;
public Mushroom(BasePizza pizza) {
this.basePizza = pizza;
}
@Override
public int cost() {
return this.basePizza.cost() + 15;
}
}


Margheria. + Extracheese
BasePizza Pizza = new Extracheese. (new Margheria ())
Pizza.cost() ??
100 + 10 = 110

7 . Factory Pattern :



Now lets say in future: Instance creation logic for Circle is changed, say we need to pass radius:

Shape circle = new Circle(4);

Problem with above design is, we need to change at multiple classes across the project.
Which makes things complex and difficult to manage.

Solution:

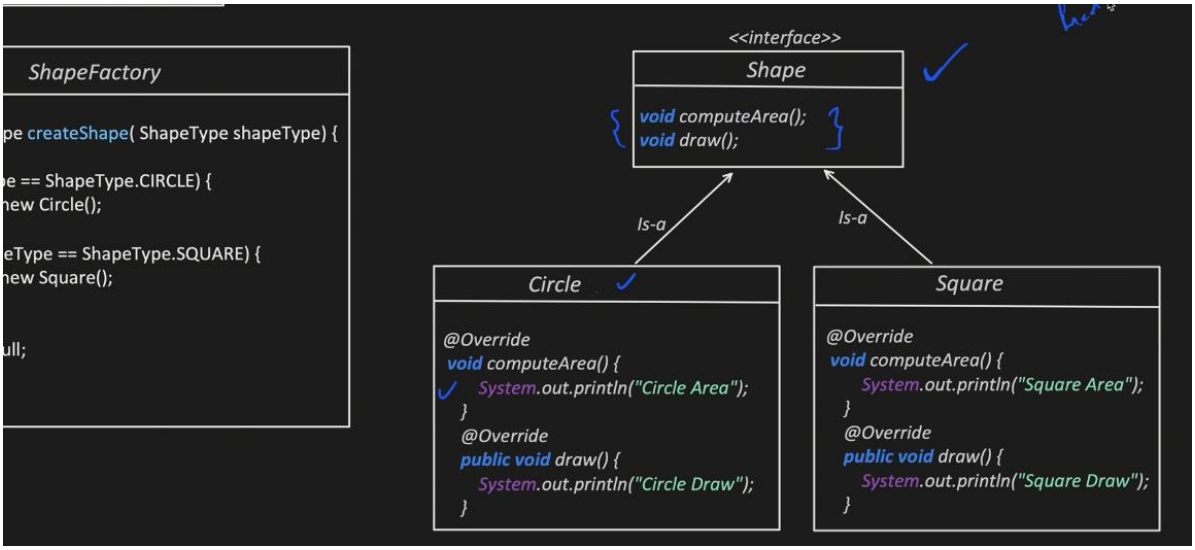
Factory Method Pattern:

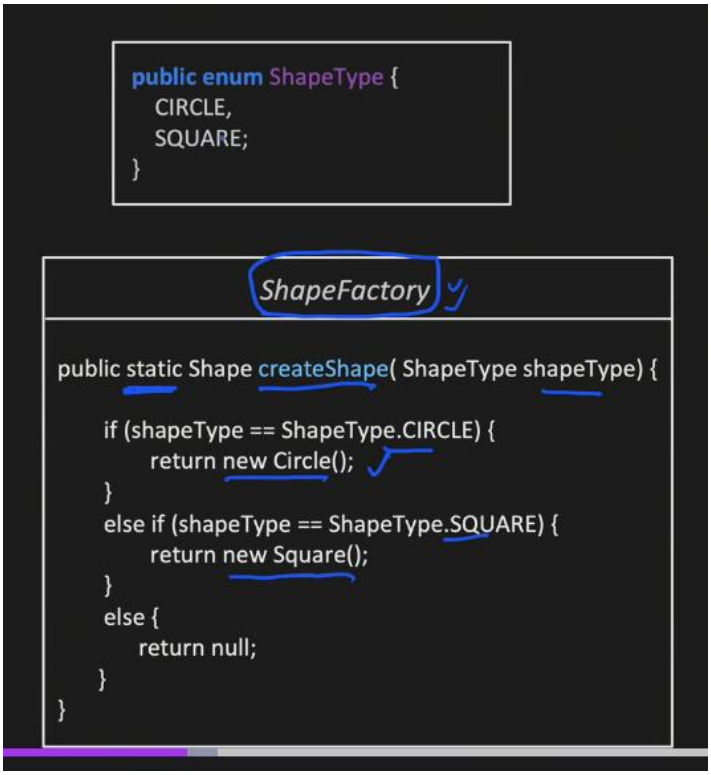
- *It's a Creational Design Pattern.*
- *Used when we want to encapsulate object creation and related creation logic at one place.*

Simple Factory Pattern

Factory Method Pattern

Simple Factory Pattern :





Advantage:

- Which ever class needs a particular shape object, can invoke this simple Factory class:

Shape circle = *ShapeFactory.createShape(ShapeType.CIRCLE);*

So in future, if there is any change in creation logic, it will be changed only at Factory class. Instead of multiple classes across the project.

Disadvantage:

- *Violates Open Closed Principle: If any new Shape is introduced then we have to touch this Factory class.*

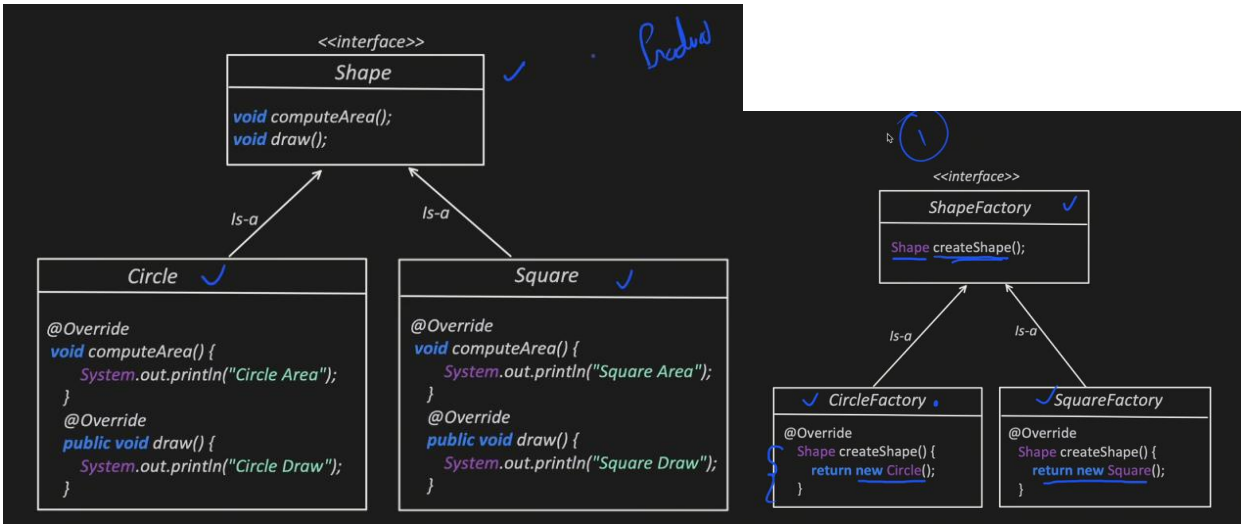
```
public static Shape createShape( ShapeType shapeType) {

    if (shapeType == ShapeType.CIRCLE) {
        return new Circle();
    }
    else if (shapeType == ShapeType.SQUARE) {
        return new Square();
    }
    else if (shapeType == ShapeType.RECTANGLE) {
        return new Rectangle();
    }
    else {
        return null;
    }
}
```

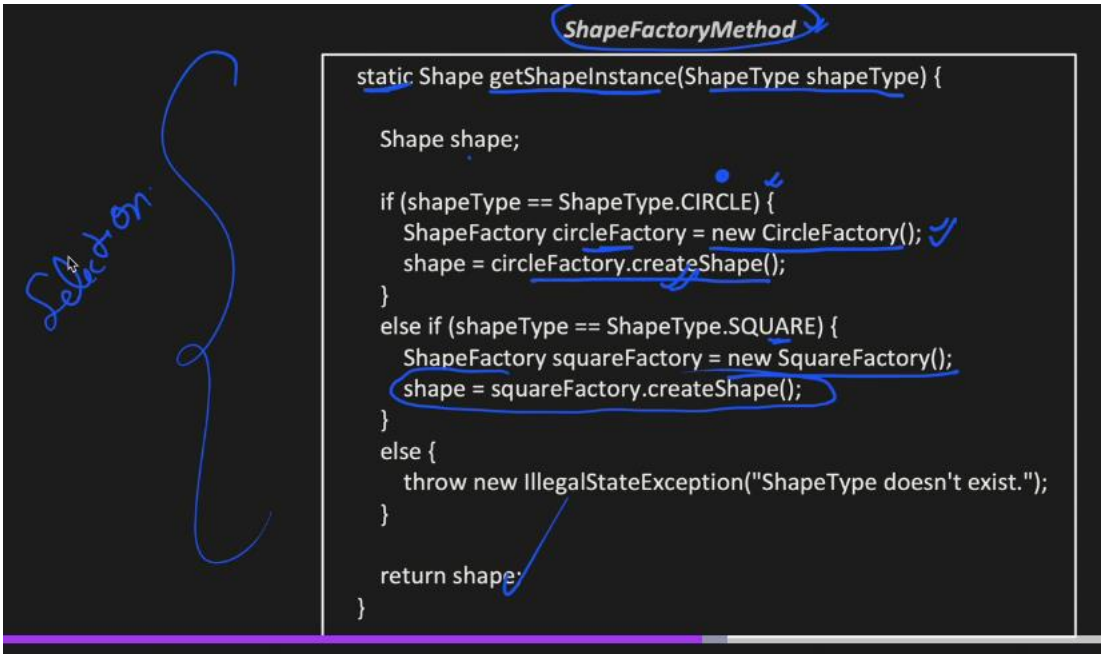
- *Factory class can become Bloated: Say if Object creation logic is complex, then this class becomes difficult to manage.*

Its now even violating Single Responsibility Principle: Factory does 2 things, Selection and Construction logic

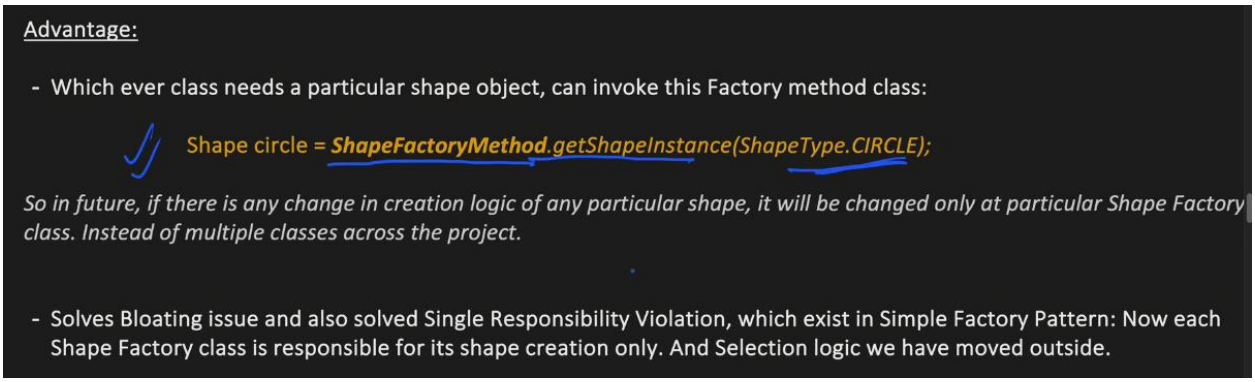
```
public static Shape createShape( ShapeType shapeType) {

    if (shapeType == ShapeType.CIRCLE) {
        //complex creation logic example
        ShapeConfig config = loadConfig("circle");
        validateConfig(config);
        initializeResources();
        return new Circle(config);
    }
    else if (shapeType == ShapeType.SQUARE) {
        //complex creation logic example
        ShapeConfig config = loadConfig("square");
        validateConfig(config);
        initializeResources();
        return new Square(config);
    }
    else {
        return null;
    }
```

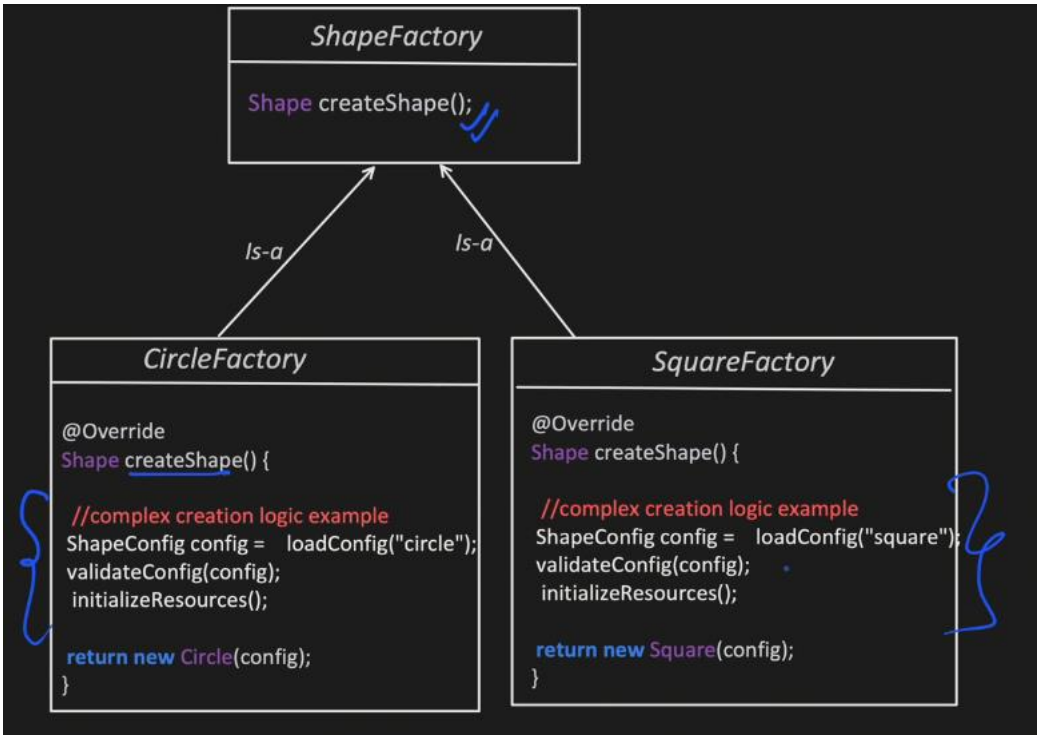
Factory Method Pattern :




ShapeFactoryMethod
static Shape getShapeInstance(ShapeType shapeType) {
Shape shape;
if (shapeType == ShapeType.CIRCLE) {
ShapeFactory circleFactory = new CircleFactory();
shape = circleFactory.createShape();
}
else if (shapeType == ShapeType.SQUARE) {
ShapeFactory squareFactory = new SquareFactory();
shape = squareFactory.createShape();
}
else {
throw new IllegalStateException("ShapeType doesn't exist.");
}
return shape;
}

Advantage :

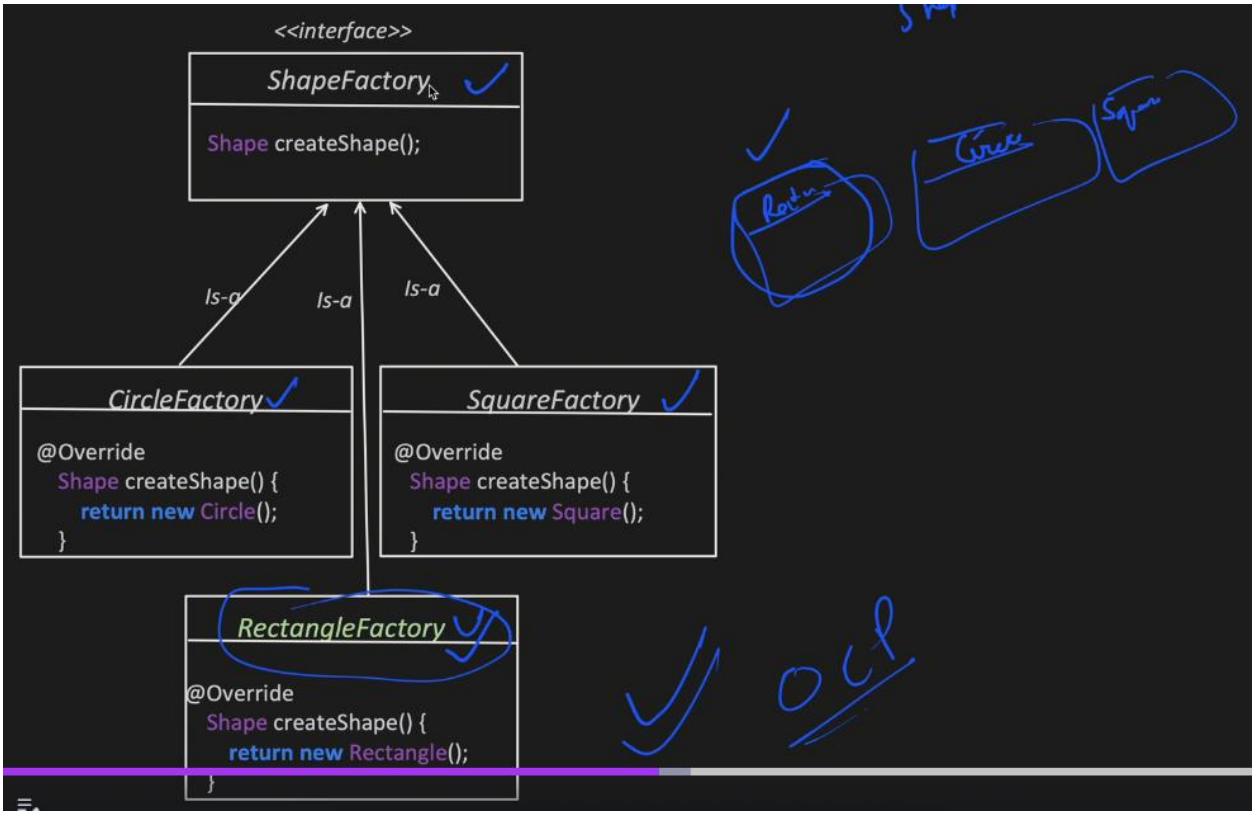

Advantage:
- Which ever class needs a particular shape object, can invoke this Factory method class:
Shape circle = ShapeFactoryMethod.getShapeInstance(ShapeType.CIRCLE);
So in future, if there is any change in creation logic of any particular shape, it will be changed only at particular Shape Factory class. Instead of multiple classes across the project.
- Solves Bloating issue and also solved Single Responsibility Violation, which exist in Simple Factory Pattern: Now each Shape Factory class is responsible for its shape creation only. And Selection logic we have moved outside.


ShapeFactory
Shape createShape();
Is-a
Is-a
CircleFactory
@Override
Shape createShape() {
//complex creation logic example
ShapeConfig config = loadConfig("circle");
validateConfig(config);
initializeResources();
return new Circle(config);
}
SquareFactory
@Override
Shape createShape() {
//complex creation logic example
ShapeConfig config = loadConfig("square");
validateConfig(config);
initializeResources();
return new Square(config);
}

Disadvantage :

Disadvantage:

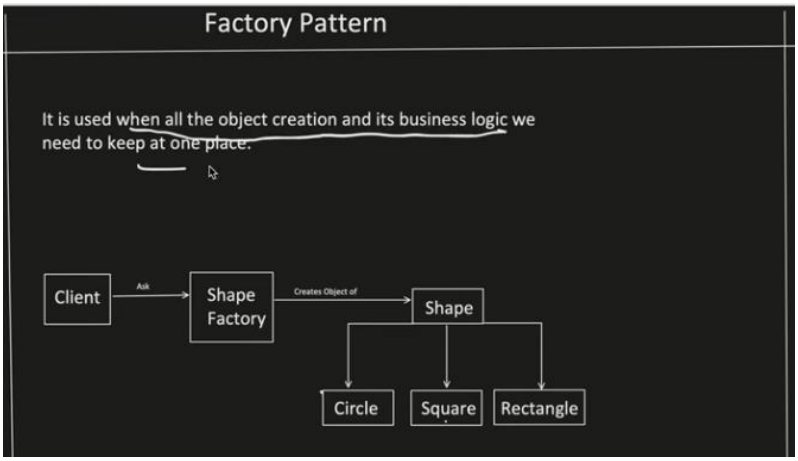
- *Still there is Violation of Open Closed Principle: If any new Shape is introduced then we now have flexibility to create new Shape Factory class which support Open Closed Principle, but the place where we select this Factory class still breaks this principle.*

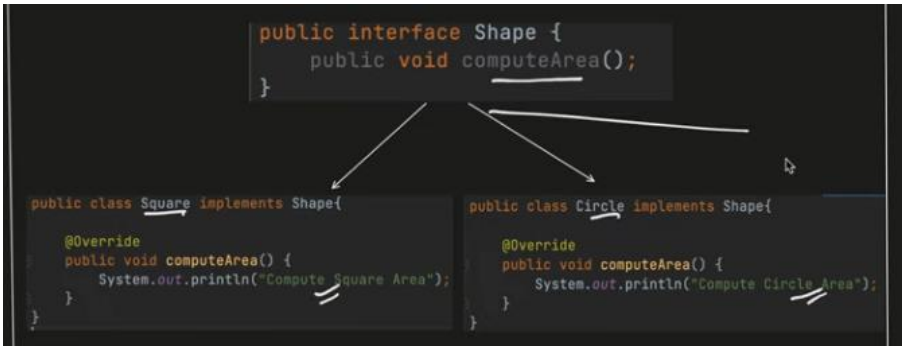



```
static Shape getShapeInstance(ShapeType shapeType) {

    Shape shape;

    if (shapeType == ShapeType.CIRCLE) {
        ShapeFactory circleFactory = new CircleFactory();
        shape = circleFactory.createShape();
    }
    else if (shapeType == ShapeType.SQUARE) {
        ShapeFactory squareFactory = new SquareFactory();
        shape = squareFactory.createShape();
    }
    else if (shapeType == ShapeType.RECTANGLE) {
        ShapeFactory recFactory= new RectangleFactory();
        shape = recFactory.createShape();
    }
    else {
        throw new IllegalStateException("ShapeType doesn't
exist.");
    }

    return shape;
}
```

Still breaking Open Closed Principle, so its not full proof.

Factory Pattern :

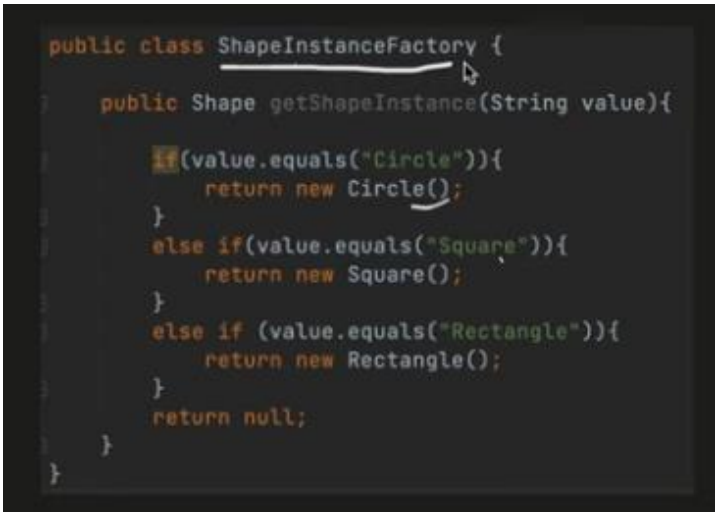




```
public interface Shape {
    public void computeArea();
}
```

```
public class Square implements Shape{

    @Override
    public void computeArea() {
        System.out.println("Compute Square Area");
    }
}
```

```
public class Circle implements Shape{

    @Override
    public void computeArea() {
        System.out.println("Compute Circle Area");
    }
}
```



```
public class ShapeInstanceFactory {

    public Shape getShapeInstance(String value){

        if(value.equals("Circle")){
            return new Circle();
        }
        else if(value.equals("Square")){
            return new Square();
        }
        else if (value.equals("Rectangle")){
            return new Rectangle();
        }
        return null;
    }
}
```

```
public class Main {

    public static void main(String args[]) {

        ShapeInstanceFactory factoryObj = new ShapeInstanceFactory();
        Shape circleObj = factoryObj.getShapeInstance( value: "Circle");
        circleObj.computeArea();
    }
}
```

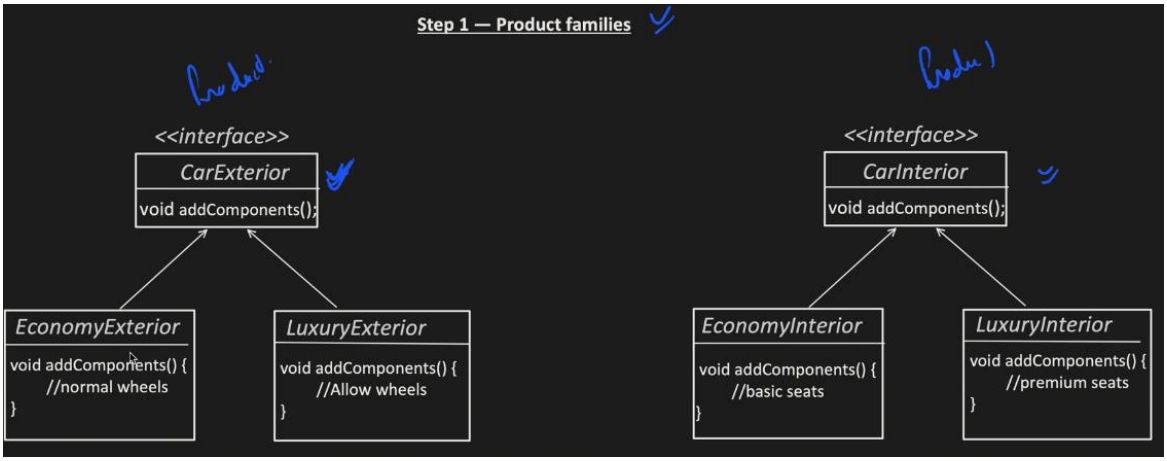
Abstract Factory :

Abstract factory on simple factory =>

Abstract Factory (Built on Simple Factory):

- Its nothing but a Factory of Factories. Where each sub factory is itself a Simple Factory.

Step 1 — Product families



Step 2 — Individual simple factories for each product

```
class ExteriorFactory {

   public static CarExterior getExterior(String type) {

      if (type.equals("economy")) {
            return new EconomyExterior();
      }
      else if (type.equals("luxury")) {
            return new LuxuryExterior();
      }
      else {
            return null;
      }
}
```

```
class InteriorFactory {

   public static CarInterior getInterior(String type) {

      if (type.equals("economy")) {
            return new EconomyInterior();
      }
      else if (type.equals("luxury")) {
            return new LuxuryInterior();
      }
      else {
            return null;
      }
}
```

Step 3 — Abstract Factory (factory of simple factories)

```
class CarFactoryProducer {

   public static Object getFactory(String choice) {

      if (choice.equals("interior")) {
            return new InteriorFactory();
      }
      else if (choice.equals("exterior")) {
            return new ExteriorFactory();
      }
      else {
            return null;
      }
   }
```

Step 4 — Client code

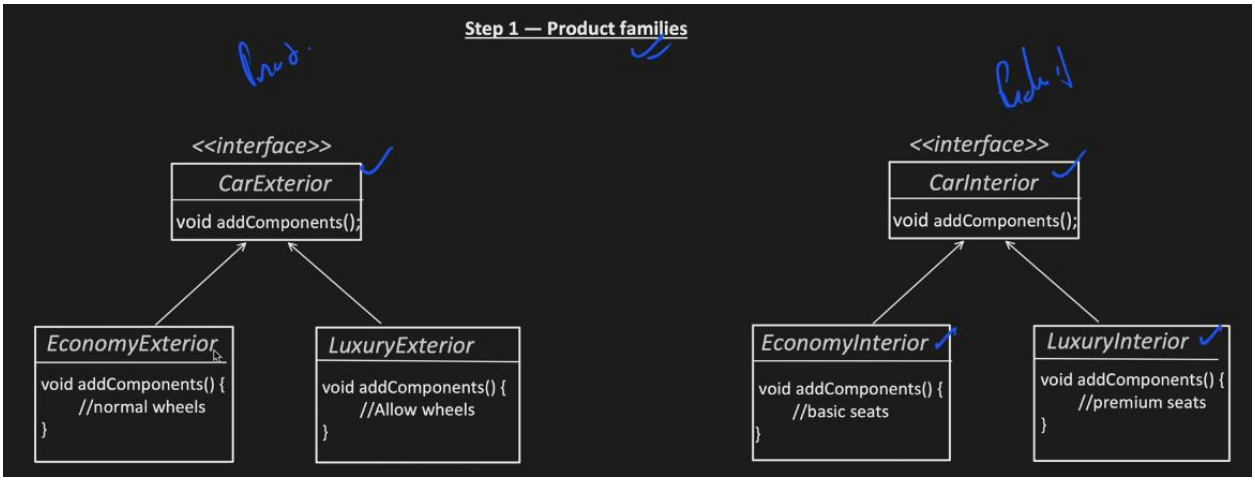
```
public class Client1 {

    public static void main(String[] args) {
        InteriorFactory interiorFactory = (InteriorFactory) CarFactoryProducer.getFactory("interior");
        CarInterior interior = interiorFactory.getInterior("luxury");
        interior.addComponents();
    }
}
```

Abstract factory on factory method =>

Abstract Factory (Built on Factory Method):

- Each Factory creates a related products together.
- Its like having multiple factories where each factory produces a complete family of products that work together.

Step 1 — Product families



Step 2 — Abstract Factory Interface

```
interface CarFactory {

    CarInterior createInterior();
    CarExterior createExterior();


// Template method that uses all factory methods
default void produceCompleteVehicle() {

        CarInterior interior = createInterior();
        CarExterior exterior = createExterior();

        interior.addComponents();
        exterior.addComponents();
    }
}
```

```java
public class EconomyCarFactory implements CarFactory {

    @Override
    public CarInterior createInterior() {
        return new EconomyInterior();
    }

    @Override
    public CarExterior createExterior() {
        return new EconomyExterior();
    }
}
```

```java
public class LuxuryCarFactory implements CarFactory {

    @Override
    public CarInterior createInterior() {
        return new LuxuryInterior();
    }

    @Override
    public CarExterior createExterior() {
        return new LuxuryExterior();
    }
}
```

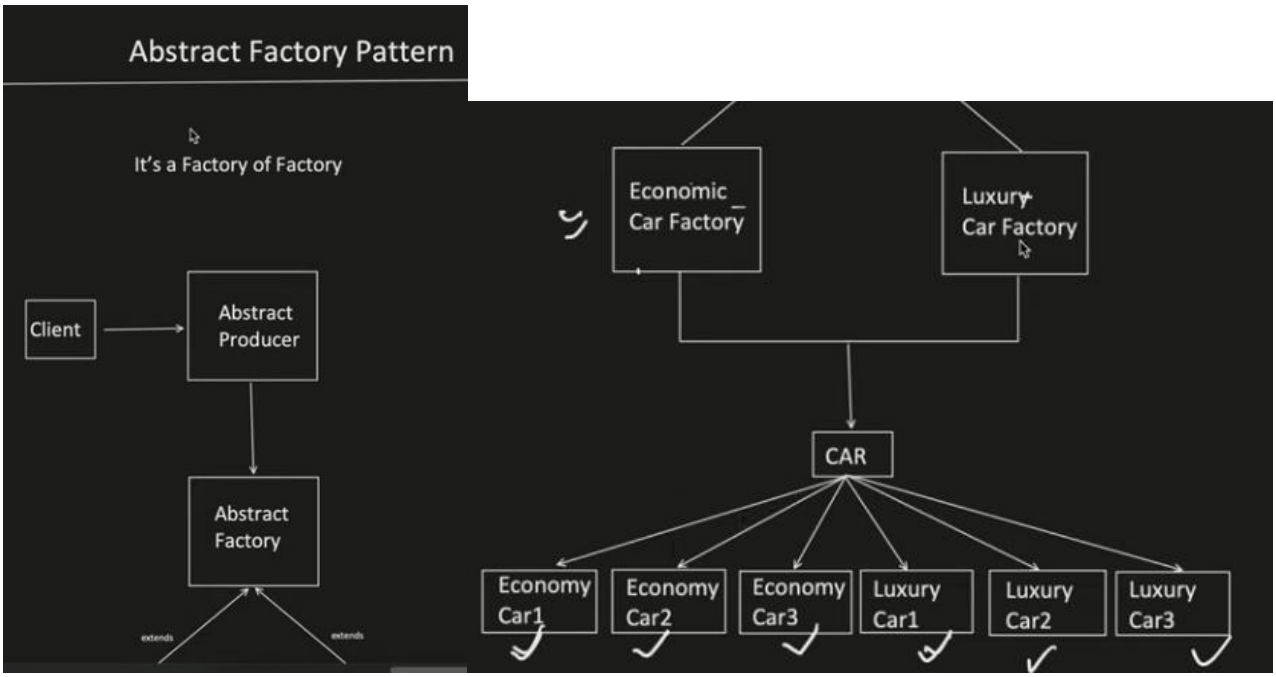
Step 3 — Factory Provider

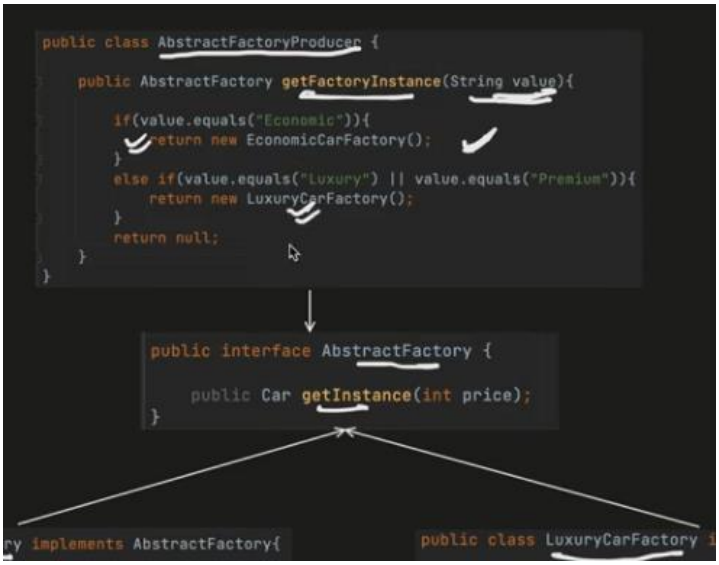
```java
public class CarFactoryProvider {

    public CarFactory getFactory(CarType type) {
        switch (type) {
            case ECONOMY:
                return new EconomyCarFactory(brand);
            case PREMIUM:
            case LUXURY:
                return new LuxuryCarFactory(brand);
            default:
                null
        }
    }
}
```

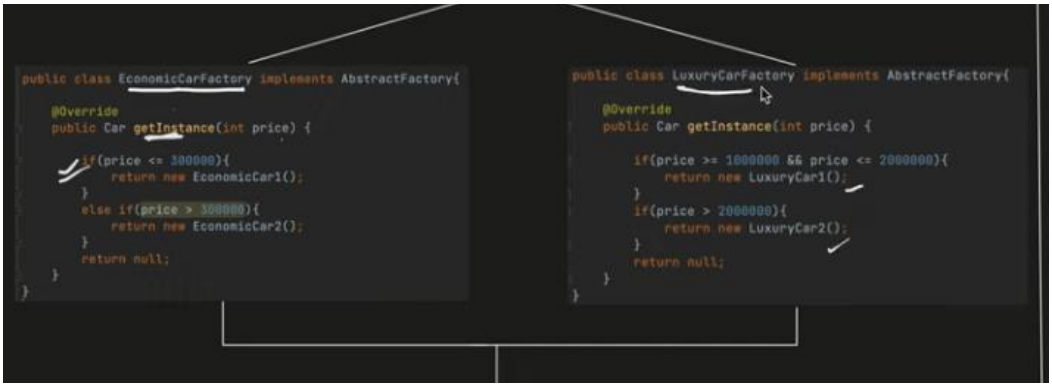
Step 4 — Client

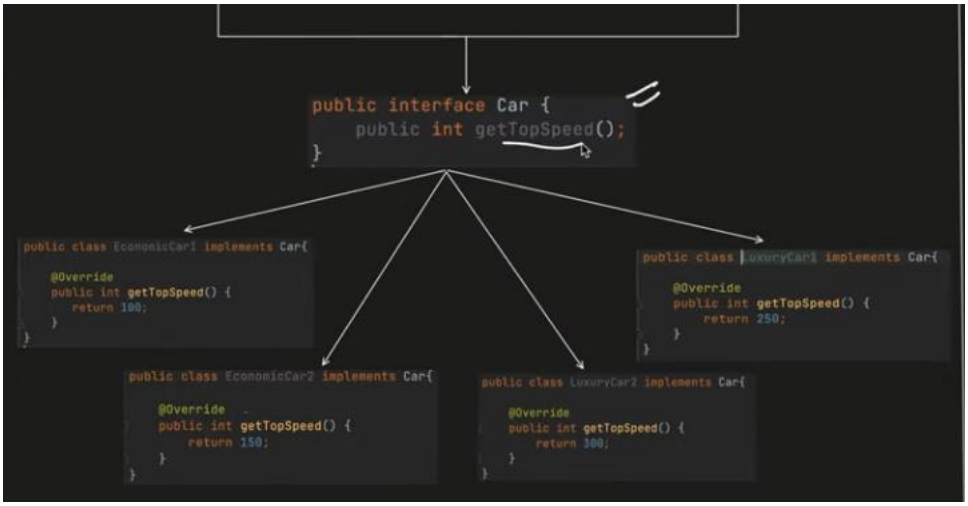
```java
public class AbstractFactoryDemo {

    public static void main(String[] args) {

        // Get Factory Provider
        CarFactoryProvider carFactoryProvider = new CarFactoryProvider();

        // Get Economy Car Factory
        CarFactory economyCar = carFactoryProvider.getFactory(CarType.ECONOMY);

        economyCar.produceCompleteVehicle();

    }
}
```

Abstract Factory Pattern










```
public class Main {

    public static void main(String args[]){

        AbstractFactoryProducer abstractFactoryProducerOb = new AbstractFactoryProducer();
        AbstractFactory abstractFactoryObj = abstractFactoryProducerOb.getFactoryInstance( value: "Premium");
        Car carObj = abstractFactoryObj.getInstance( price: 5000000);
        System.out.println(carObj.getTopSpeed());
    }
}
```