



# Creational Design Patterns

FAROUQ HAIDER



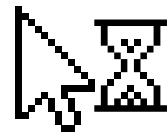
01 Factory Pattern

02 Abstract Factory Pattern

03 Builder Pattern



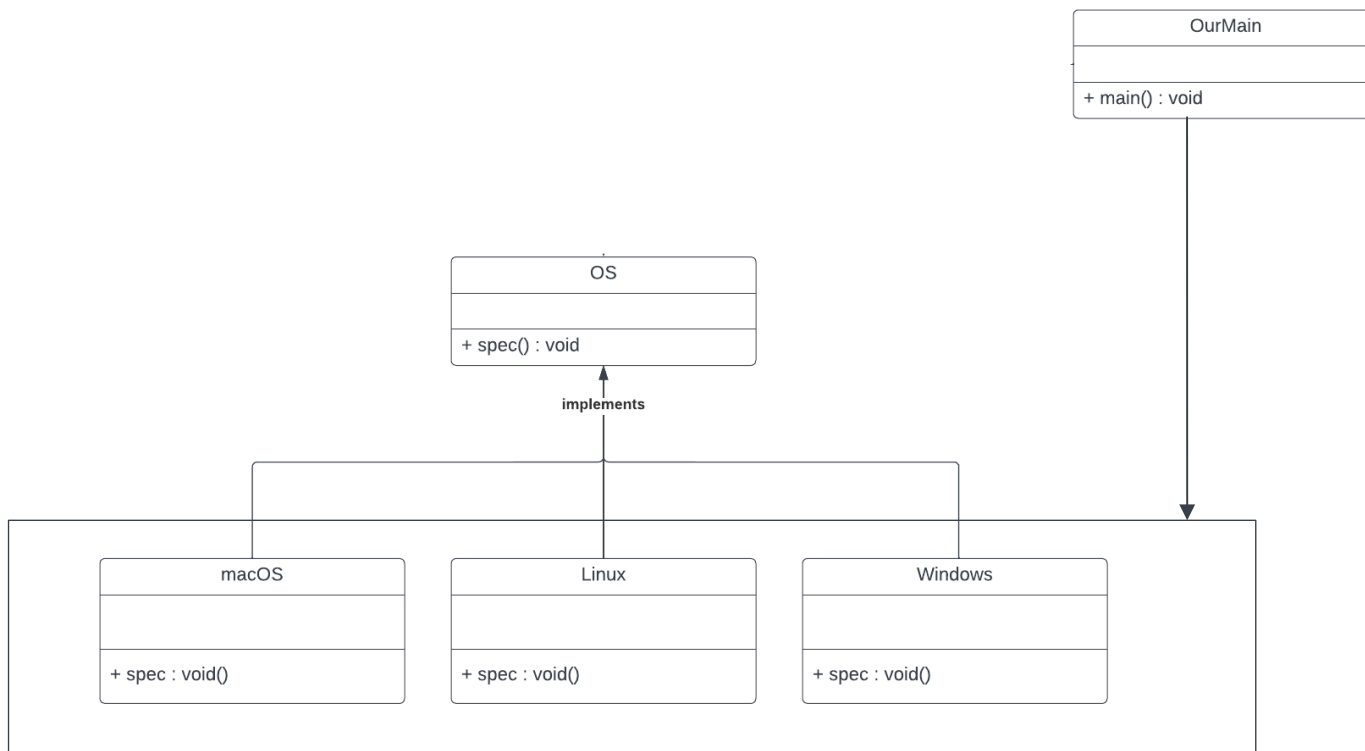
# What are Creational Patterns?





# 01

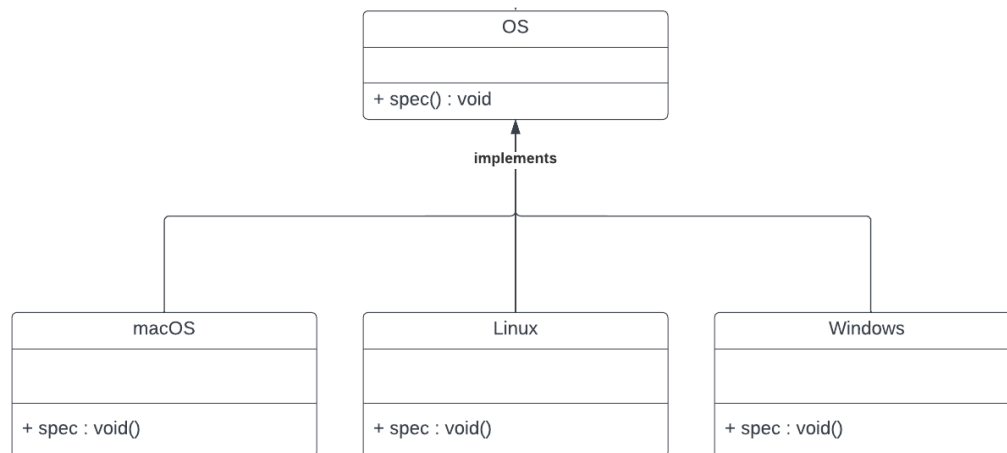
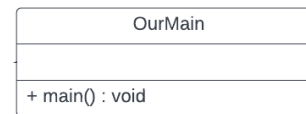
## Factory Pattern





```
public class Main
{
    Windows myW = new Window();
    Linux myL = new Linux();
    macOS myM = new macOS();
}
```







# Introduction

- Super-class with multiple sub-classes.
- Responsibility of initiation = Factory class.





```
public interface OS
{
    public void spec();
}
```



```
public class macOS implements OS
{
    public void spec(){
        System.out.print("This is Macintosh");
    };
}
```



```
public class Linux implements OS
{
    public void spec(){
        System.out.print("This is Linux");
    };
}
```



```
public class Windows implements OS
{
    public void spec(){
        System.out.print("This is Windows");
    };
}
```



```
public class OSFactory
{
    public OS getInstance(String str)
    {
        if(str.equals("Broken"))
            return new Windows();
        else if(str.equals("Open"))
            return new Linux();
        else
            return new macOS();
    }
}
```



```
public class OurMain
{
    public static void main(String args[]){
        OSFactory OSF = new OSFactory();
        OS obj = OSF.getInstance("Open");
    }
}
```



# Why Use?



Loose  
Coupling



Addition  
of Objects







# Disadvantages?

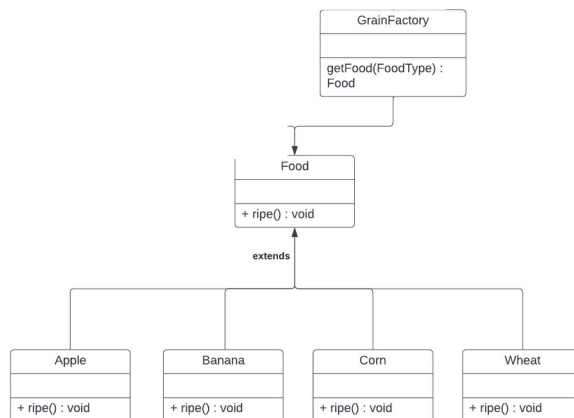
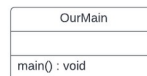


High number of  
classes



# 02

## Abstract Factory Pattern





FruitFactory

getFood(FoodType) :  
Food

GrainFactory

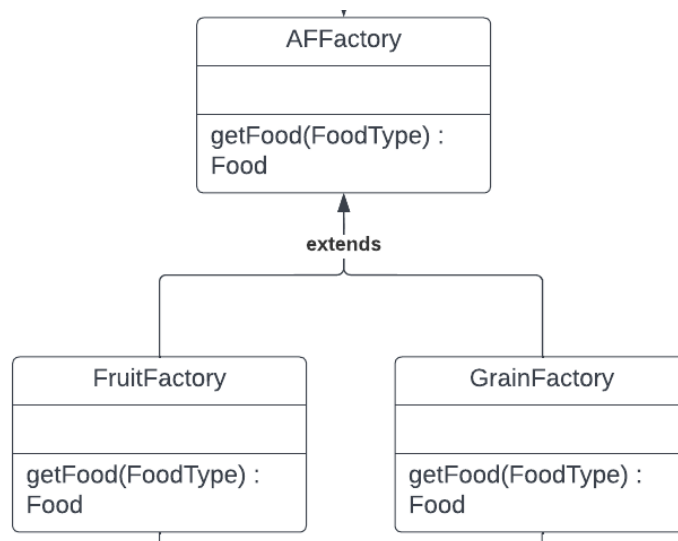
getFood(FoodType) :  
Food



```
public class GrainFactory extends AFFactory
{
    Food getFood(FoodType foodType){
        if(FT == "CORN")
            return new Corn();
        else if(FT == "CARROT")
            return new Banana();
    }
}
```



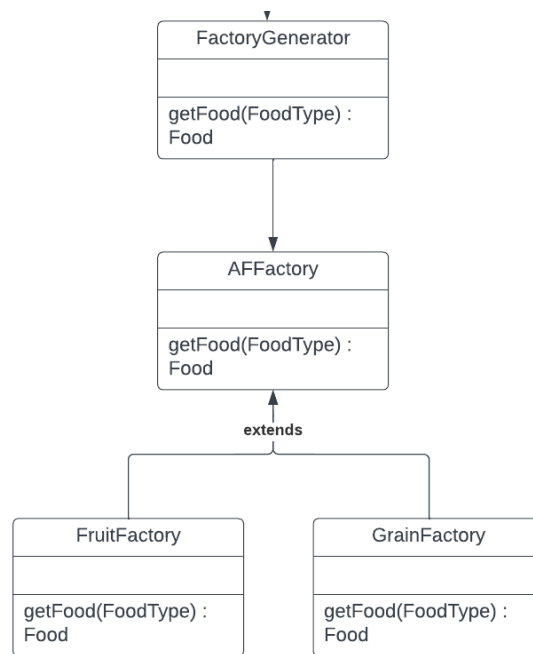
```
public class FruitFactory extends AFFactory
{
    Food getFood(FoodType foodType){
        if(FT == "APPLE")
            return new Apple();
        else if(FI == "BANANA")
            return new Banana();
        }
    }
```







```
public abstract class AFactory
{
    abstract Food getFood(FoodType foodType);
}
```





```
public abstract class FactoryGenerator
{
    public AFactory getFactory(FactoryType FT){
        if(FT == "FRUITFACTORY")
            return new FruitFactory();
        else if(FT == "GRAINFACORY")
            return new GrainFactory();
    };
}
```



```
public class ourMain
{
    Food anApple =
    FactoryGenerator.getFactory("FRUITFACTORY").get("APPLE");
}
```



# Why Use?



# Disadvantages?



- Hard to Manage



**03**

# **Builder Pattern**





# Introduction

- Builder Class for assigning parameters.
- More than a few parameters.



# Why Use?



Safe

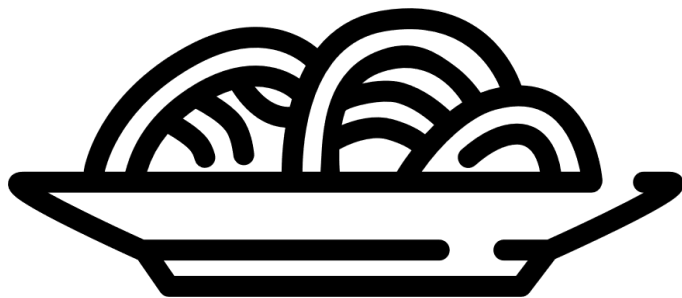


Easy to  
read





Immutability





```
public class Pasta
{
    private String noodleType;
    private boolean hot sauce;
    private boolean chicken;
    private boolean egg;
    private int calories;
}
```



**METHOD 1...**



```
public class Pasta
{
    private String noodleType;
    private boolean hotSauce;
    private boolean chicken;
    private boolean egg;
    private int calories;

    public Pasta(String noodleType, Boolean hotSauce,
        Boolean chicken, Boolean egg, int calories){

    }
}
```





```
public class Pasta
{
    private String noodleType;
    private boolean hotSauce;
    private boolean chicken;
    private boolean egg;
    private int calories;

    public Pasta(String noodleType, Boolean hotSauce,
        Boolean chicken, Boolean egg, int calories){
        this.noodleType = noodleType;
        this.hotSauce = hotSauce;
        this.chicken = chicken;
        this.egg = egg;
        this.calories = calories;
    }
}
```



```
public class Main
{
    Pasta P = new Pasta(true,true,true,652);
}
```





**METHOD 2...**




```
public class Pasta
{
    private String noodleType;
    private boolean hot sauce;
    private boolean chicken;
    private boolean egg;
    private int calories;

    public void setNoodleType(String noodleType){
        this.noodleType = noodleType;
    }
    public void setHot sauce(String noodleType){
        this.hot sauce = hot sauce;
    }
    .
    .
    .
}
```



```
public class Main
{
    Pasta P = new Pas
    P.setNoodleType("Ra
    P.setHotsauce(true);
    P.setChicken(true);
    P.setEgg(true);
    P.setCalories(65
}
```





```
public class Pasta
{
    private String noodleType;
    private boolean hotSauce;
    private boolean chicken;
    private boolean egg;
    private int calories;

    public static class PastaBuilder{
        ...
    }

    private Pasta(PastaBuilder PB){
        ...
    }
}
```



```
public static class PastaBuilder
{
    //shadow variables

    public PastaBuilder setNoodleType(String noodleType){
        this.noodleType = noodleType;
        return this;
    }
    public PastaBuilder hotSauce(String hotSauce){
        this.hotSauce = hotSauce;
        return this;
    }
}
```



```
public static class PastaBuilder
{
    //shadow variables

    public PastaBuilder setNoodleType(String noodleType){
        this.noodleType = noodleType;
        return this;
    }
    public PastaBuilder hotsauce(String hotsauce){
        this.hotsauce = hotsauce;
        return this;
    }

    ...

    public Pasta build(){
        return new Pasta(this);
    }
}
```





```
private Pasta(PastaBuilder PB){  
    noodleType = PB.noodleType;  
    hotSauce = PB.hotSauce;  
    chicken = PB.chicken;  
    egg = PB.egg;  
    calories = PB.calories;  
}
```



```
public class Main
{
    Pasta P = new
Pasta.PastaBuilder().setNoodleType("Ravioli").
    setChicken(true).setHotsauce(true).setEgg(true).
    setCalories(625).build();
}
```



# Disadvantages?



- Increased LOC
- Separate Builder for every Class.



- **RULE OF THUMB:** More than 4-5 Parameters.



- **RULE OF THUMB:** More than 4-5 Parameters.



**THANK YOU**