## 单例模式

### 懒汉模式

public class SingleTon{

private static SingleTon INSTANCE = null;

private SingleTon(){}

public static SingleTon getInstance() {

if(INSTANCE == null){

INSTANCE = new SingleTon();

}

return INSTANCE；

}

}

### 静态内部类模式

public class Singleton

{

private Singleton(){}

static {

System.out.println("This's static code block!");

}

private static class SingletonHandler {

private static Singleton singleton = new Singleton();

static {

System.out.println("This's innerClass's static code block");

}

}

public static Singleton getInstance(){

return SingletonHandler.singleton;

}

public static void display(){

System.out.println("This's display!");

}

}

## 装饰着设计模式

接口A 实现类 C

代理类B 实现接口A ,构造器参数为A

在类B 中创建A 的属性,用于存放接口A 的其他实现类

增强实现类C 的方法

## 策略模式

思路:针对在不同类上有不同表现的方法,放到接口和抽象类上实现继承不合适,所以将在不同类上表现相同的行为的方法通过继承来实现,而不同的表现的方法用类属性方式来表达.多种表现类型的行为用接口表示,然后多种表现同时实现一个接口.在类里边想要表现那种行为就实现那种类.

## 观察者模式

### 主题接口

Subject:

registerObserver();//注册观察者

removeObserver();//移除观察者

notifyObserver();//修改观察者

### 主题实现类

WeatherData implements Subject:

registerObserver();//注册观察者

removeObserver();//移除观察者

notifyObserver();//修改观察者

getTemperature();//获取温度

getHumidity();//获取湿度

getPressure();//获取压力

measurementsChanged();//参数改变

### 观察者接口

Observer:

Update();

### 展示接口

DisplayElement:

Display();

### 观察者实现者

CurrentConditionsDisplay implement Observer, DisplayElement:

Update();

Display();

在观察者实现类中添加主题的属性,同时实现类初始化的时候需要在构造器中添加主题的实现类,来表示订阅哪个主题,构造器中执行主题的注册观察者的方法.

这主题里边的参数改变方法中调用主题的修改观察者的方法,遍历每个观察者,每个观察者的实体类中执行各自的update,来修改参数.

观察者的update中调用本类中的display方法进行展示.

## 动态代理模式

必须是final

final UserService service = new UserServiceImpl();

UserService ProxyService = (UserService)Proxy.newProxyInstance(

ProxyDemo.class.getClassLoader(),

service.getClass().getInterfaces(),

new InvocationHandler(){

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

// ProxyService add (参数);

//System.out.println("1");

/\*\*

\* 进行增强 如果是add 打印 增强

\*/

if(method.getName().equals("add")){

System.out.println("增强");

}

//放行 调用原来的service

return method.invoke(service);

}

});

## 装饰者模式

例子:

饮料抽象类下有houseblend darkroast decaf espresso 四种不同的咖啡

同时还有各种调料,蒸奶/豆浆/摩卡/奶泡.

饮料抽象类中有描述和花费的方法.

需求是用不同的咖啡搭配不同的调料,获得最后的饮品并且计算出正确的价格.

实现各种各样的饮品将会出现n多个类型,将会难以维护.

这里通过组合和委托的方式可以达到类继承的效果.设计原则类应该对扩展开放,对修改关闭.

具体实现代码有代码示例.

在jdk中应用该模式的地方为java的io部分

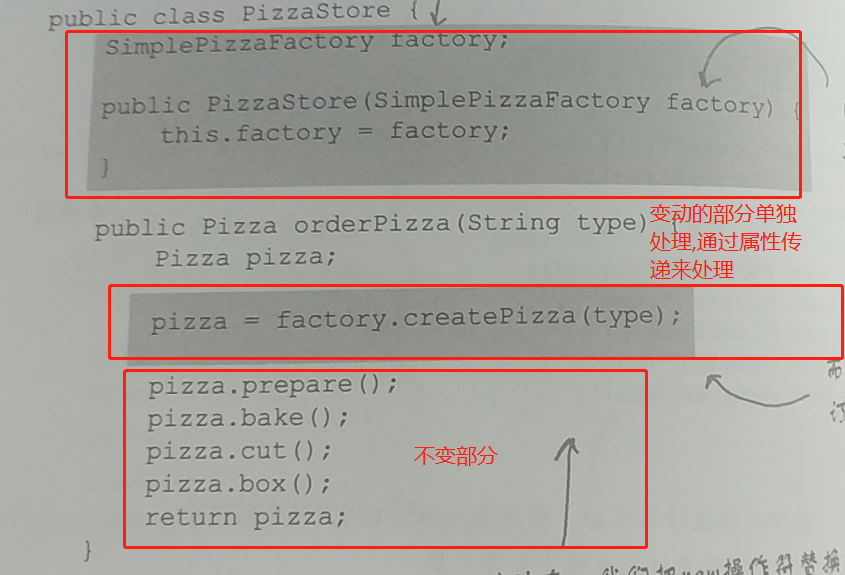
## 工厂模式

针对接口编程可以隔离掉有系统可能发生的一大堆改变.

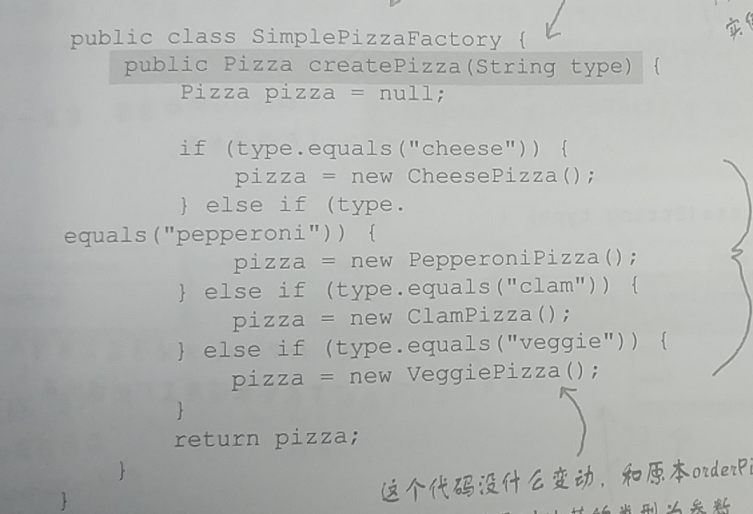
**工厂方法:**

思想:将代码变动的部分和不变动的部分隔离开来.

实例:披萨店中不同风味的披萨,变动的部分是创建出不同风味的披萨,不变动的部分是准备,烘烤,切片,装盒.所以只需要把创建披萨的部分隔离出来.

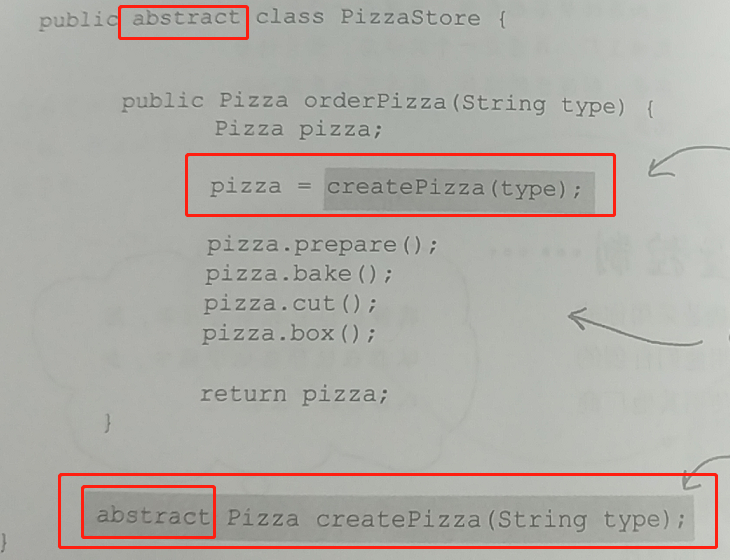


创建披萨工厂

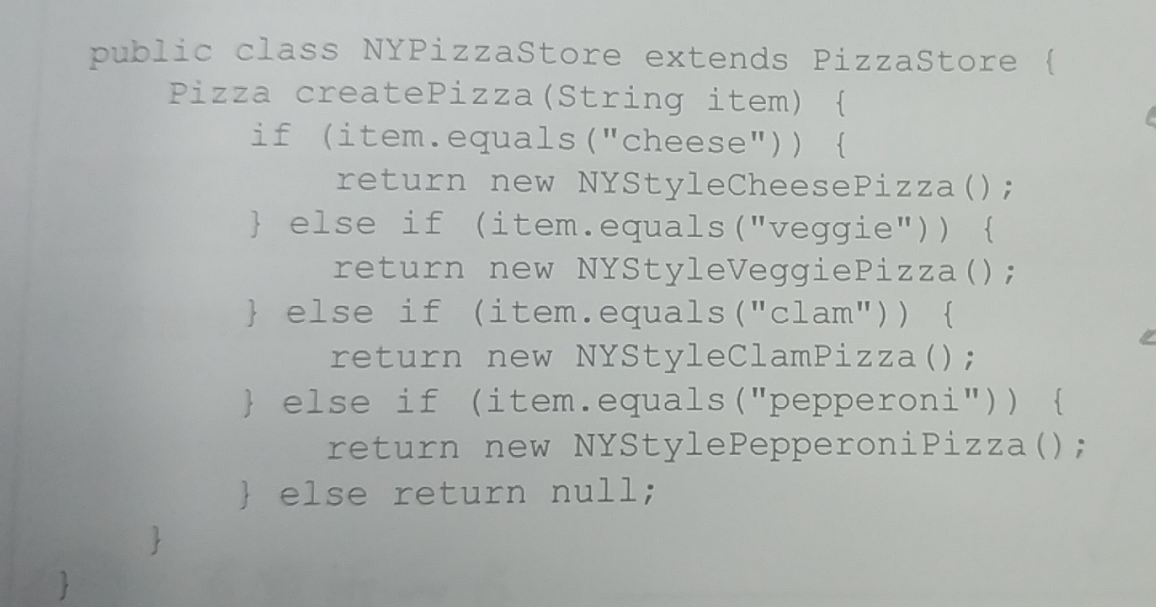


**工厂模式方法:**

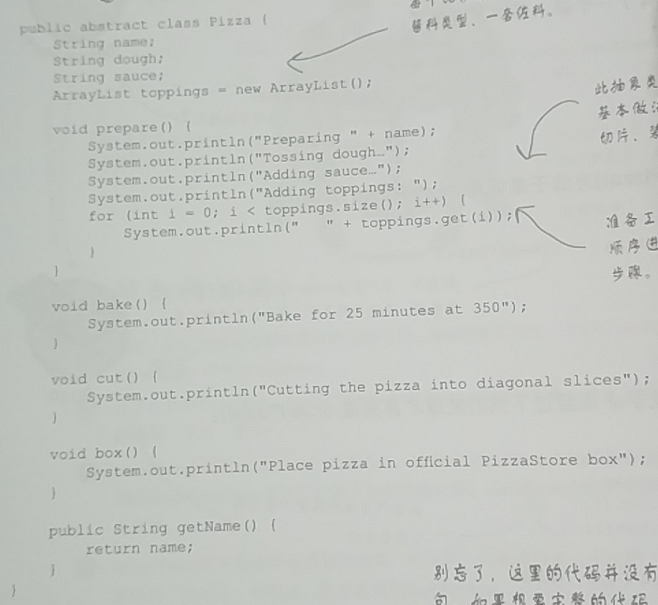
定义抽象类披萨商店,子类继承披萨商店,必须要重写创建披萨的方法.这样就可以根据区域来创建不同风味的披萨了.



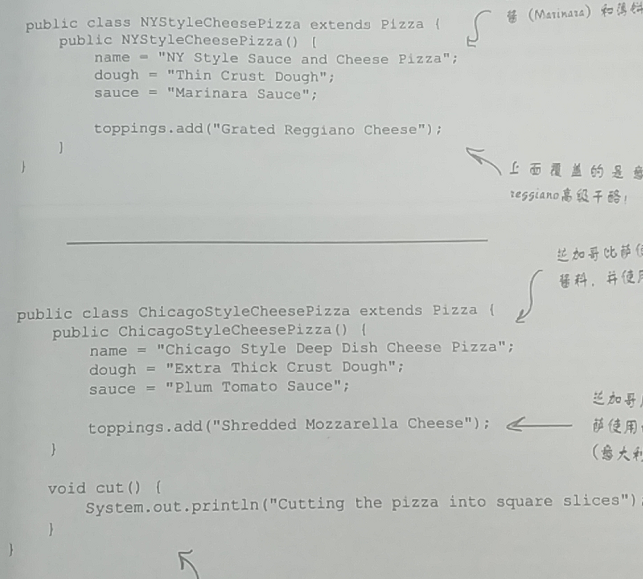
子类披萨店



披萨自身的代码

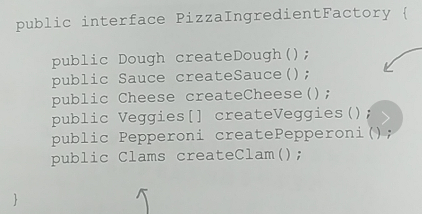


创建不同的披萨

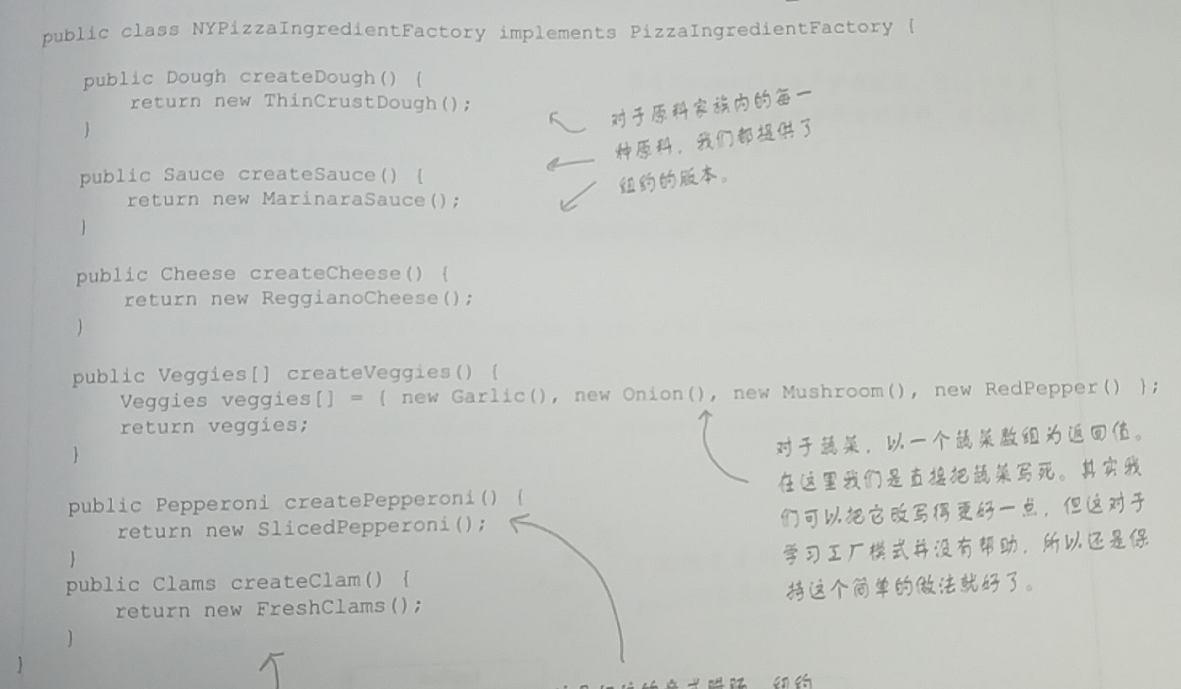


在这里相当于每多一个披萨类型披萨商店就多了一个依赖.这里使用依赖倒置的原则.将披萨的调料通过注入调料工厂的模式来灵活注入.

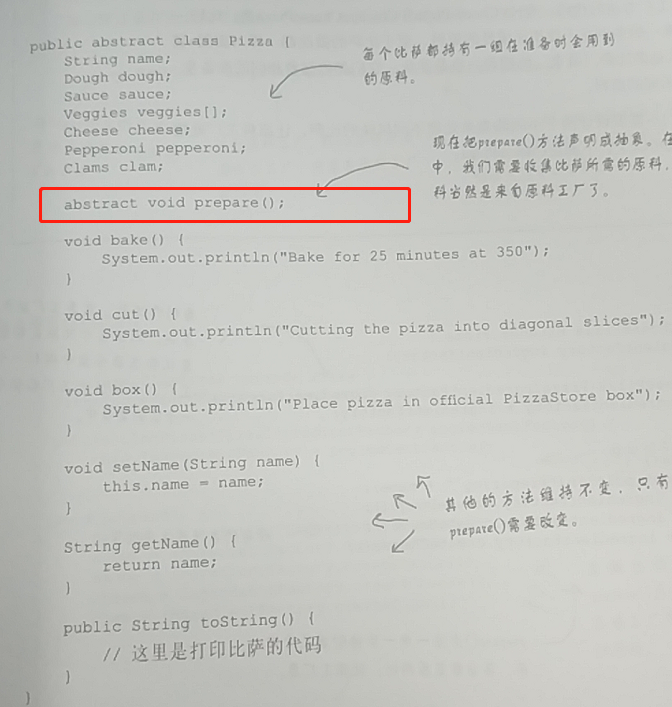
创建调料工厂:



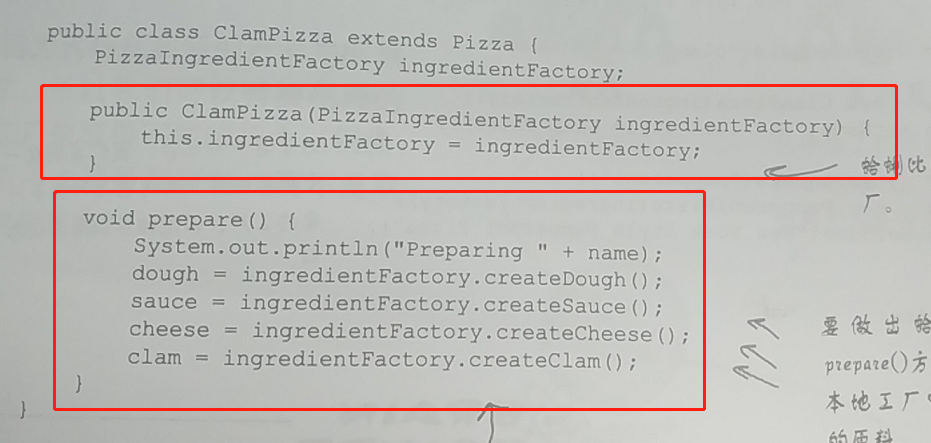
创建一个调料工厂



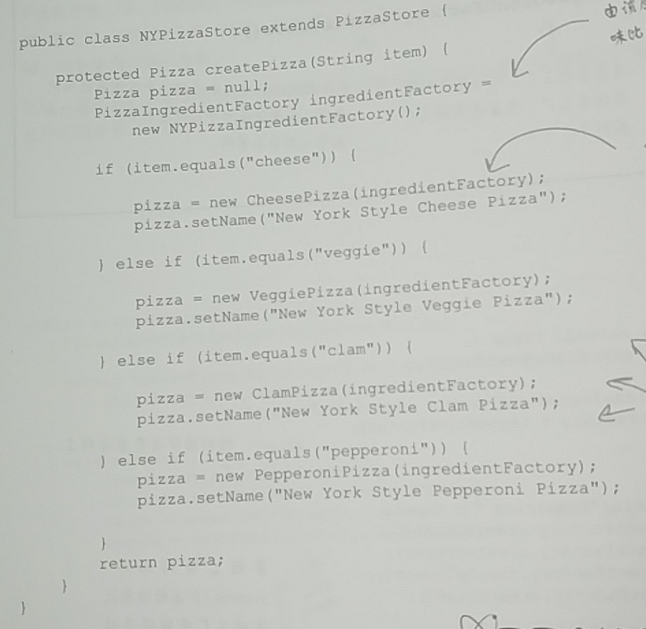
重新制作披萨的抽象类,其中通过prepare方法来完成工厂中调料的使用.



创建披萨实体类



最后的披萨店:

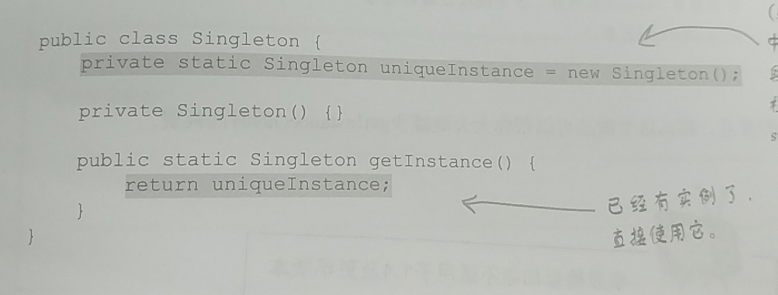


## 单件模式

防止类被别的类创建只需要把构造函数改为private就行了.

懒汉模式的风险可以通过使用synchronized关键字来进行解决.但是会影响性能.

使用急切创建实例,在线程访问之前就创建好这个实例



双重检查加锁

