## 一句话概括：

**建立一个工厂类，对实现了同一接口的一些类 根据传入的不同参数进行不同实例的创建**。

## 补充介绍：

简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（这些产品类继承自一个父类或接口）的实例。

## 参与角色:

1）抽象基类（也可以是接口）

2）多个具体的实现类

3）工厂类（包括创建a的实例的方法）

## 优点：

工厂类是整个模式的关键.包含了必要的逻辑判断,根据外界给定的信息,决定究竟应该创建哪个具体类的对象.通过使用工厂类,外界可以从直接创建具体产品对象的尴尬局面摆脱出来,仅仅需要负责“消费”对象就可以了。而不必管这些对象究竟如何创建及如何组织的．明确了各自的职责和权利，有利于整个软件体系结构的优化。

## 缺点：

由于工厂类集中了所有实例的创建逻辑，违反了高内聚责任分配原则，将全部创建逻辑集中到了一个工厂类中；它所能创建的类只能是事先考虑到的，如果需要添加新的类，则就需要改变工厂类了。当系统中的具体产品类不断增多时候，可能会出现要求工厂类根据不同条件创建不同实例的需求．这种对条件的判断和对具体产品类型的判断交错在一起，很难避免模块功能的蔓延，对系统的维护和扩展非常不利；

## 使用案例或场景：

工厂模式使用案例非常广泛，尤其是在各种框架里面去创建复杂对象的时候。

一般使用工厂模式的工厂类，会以Factory结尾。

比如hibernate框架中通过SessionFactory创建session。

Spring中可以通过不同的配置方式（XML或者注解）来创建bean

再比如我们需要将日志文件存放在本地、数据库或云端则可以定义工厂类根据不同需要创建不同的存储实例。

## 示例程序

程序简介：

这段示例程序的作用是制作身份证，它其中有五个类：

Product类和Factory类属于framework包，这两个类组成了生成实例的框架。

IDCard类和IDCardFactory类负责实际的加工处理，他们属于idcard包。

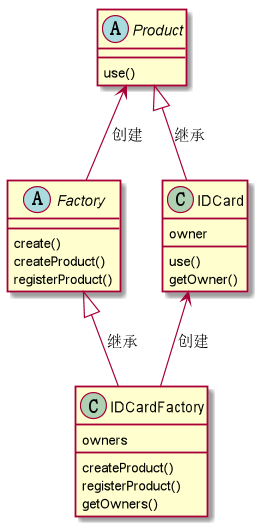
Main类是用于测试程序行为的类。

阅读时，请注意：

生成实例的框架（framework包）

加工处理（idcard包）

类图：



代码：

framework包：

**public abstract class** Product {  
 **public abstract void** use();  
}

**public abstract class** Factory {  
 **public final** Product create(String owner) {  
 Product p = createProduct(owner);  
 registerProduct(p);  
 **return** p;  
 }  
 **protected abstract** Product createProduct(String owner);  
 **protected abstract void** registerProduct(Product product);  
}

idcard包：

**public class** IDCard **extends** Product {  
 **private** String **owner**;  
  
 IDCard(String owner) {  
 System.***out***.println(**"制作"** + owner + **"的身份证"**);  
 **this**.**owner** = owner;  
 }  
 **public void** use() {  
 System.***out***.println(**"使用"** + **owner** + **"的身份证"**);  
 }  
 **public** String getOwner() {  
 **return owner**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "IDCard{"** +  
 **"owner='"** + **owner** + **'\''** +  
 **'}'**;  
 }  
}

**public class** IDCardFactory **extends** Factory {  
  
 **private** List<IDCard> **owners** = **new** ArrayList<>();  
  
 @Override  
 **protected** Product createProduct(String owner) {  
 **return new** IDCard(owner);  
 }  
  
 @Override  
 **protected void** registerProduct(Product product) {  
 **owners**.add((IDCard) product);  
 }  
  
 **public** List<IDCard> getOwners() {  
 **return owners**;  
 }  
}

测试的Main方法：

**public static void** main(String[] args) {  
 Factory factory = **new** IDCardFactory();  
 Product xiaoming = factory.create(**"小明"**);  
 Product xiaohong = factory.create(**"小红"**);  
 Product xiaogang = factory.create(**"小刚"**);  
 xiaoming.use();  
 xiaohong.use();  
 xiaogang.use();  
  
 System.***out***.println(**"Registered Owners: "** + ((IDCardFactory) factory).getOwners());  
}

运行结果：

制作小明的身份证

制作小红的身份证

制作小刚的身份证

使用小明的身份证

使用小红的身份证

使用小刚的身份证

Registered Owners: [IDCard{owner='小明'}, IDCard{owner='小红'}, IDCard{owner='小刚'}]