工厂模式主要是为创建对象提供了接口。工厂模式按照《Java与模式》中的提法分为三类：

1. 简单工厂模式(Simple Factory)

2. 工厂方法模式(Factory Method)

3. 抽象工厂模式(Abstract Factory)

这三种模式从上到下逐步抽象，并且更具一般性。还有一种分类法，就是将简单工厂模式看为工厂方法模式的一种特例，两个归为一类。下面是使用工厂模式的两种情况：

1.在编码时不能预见需要创建哪种类的实例。

2.系统不应依赖于产品类实例如何被创建、组合和表达的细节

简单工厂模式

顾名思义，这个模式本身很简单，而且使用在业务较简单的情况下。

它由三种角色组成（关系见下面的类图）：

1、工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑。在java中它往往由一个具体类实现。

2、抽象产品角色：它一般是具体产品继承的父类或者实现的接口。在java中由接口或者抽象类来实现。

3、具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个或多个具体类实现。

|  |
| --- |
| **interface** IFood { // 定义食物标准  **public** **void** eat(); // 食物的核心功能：吃  } //抽象产品角色  //以下两个类为具体产品角色  **class** Bread **implements** IFood { // 食物：面包  **public** **void** eat() { // 覆写方法  System.***out***.println("吃面包。");  }  }  **class** Milk **implements** IFood { // 食物：牛奶  **public** **void** eat() { // 覆写方法  System.***out***.println("喝牛奶。");  }  }  **class** Factory { //工厂类角色  /\*\*  \* 获取IFood接口实例化对象，利用此方法对外隐藏子类，由于Factory类没有属性，所以定义static方法  \* **@param** className 要获取的子类标记  \* **@return** 存在指定标记返回对应子类实例，否则返回null  \*/  **public** **static** IFood getInstance(String className) {  **if** ("bread".equals(className)) { // 判断子类标记  **return** **new** Bread(); // 返回子类实例  } **else** **if** ("milk".equals(className)) { // 判断子类标记  **return** **new** Milk(); // 返回子类实例  } **else** {  **return** **null**; // 没有匹配类型返回null  }  }  }  **public** **class** JavaDemo {  **public** **static** **void** main(String args[]) {  IFood food = Factory.*getInstance*("bread"); // 通过工厂获取实例  food.eat(); // 调用公共标准  }  }  程序执行结果：吃面包。 |

在程序中定义了一个Factory工厂类，并在类中提供了一个静态方法，用于返回Ifood接口实例化对象，这样在主类不需要关注具体的接口子类（即具体产品类），只需要传入指定的类型标记就可以获取接口对象。

使用了简单工厂模式后，我们的程序加符合现实中的情况；而且客户端免除了直接创建产品对象的责任，而仅仅负责"消费"产品。比如上面的程序中，我们只需要传入指令 “bread”,不需要关注怎样创建对象，将相当于我们去餐馆点菜，工厂类就相当于服务员，我们只需要告诉服务员我们需要的菜的种类就可以，而不需要了解菜品烹饪的过程。若没有工厂类。我们在餐馆点菜时需要直接与厨师沟通，直接调用具体产品，这样肯定会造成不好的体验。

但这样也有缺点：1、扩展性差（我想增加一种面条，除了新增一个面条产品类，还需要**修改**工厂类方法） 2、不同的产品需要不同额外参数的时候，不支持

工厂方法模式

组成：

1、抽象工厂角色：这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。

2、具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。在java中它由具体的类来实现。

3、抽象产品角色：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。

4、具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。

工厂方法模式就是把简单工厂中具体的工厂类，划分成两层：**抽象工厂层+具体的工厂子类层**。

|  |
| --- |
| **public interface** IFood {  **public void** eat(); *// 食物的核心功能：吃* }  **class** Bread **implements** IFood { *// 食物：面包* **public void** eat() { *// 覆写方法* System.***out***.println(**"吃面包。"**);  } }  **class** Milk **implements** IFood { *// 食物：牛奶* **public void** eat() { *// 覆写方法* System.***out***.println(**"喝牛奶。"**);  } }  **public abstract class** FoodFactory {  **public abstract** IFood creatFood(); }  **public class** breadFactory **extends** FoodFactory {  @Override  **public** IFood creatFood(){  **return new** Bread();  } }  **public class** milkFactory **extends** FoodFactory{  @Override  **public** IFood creatFood(){  **return new** Milk();  } }  **public class** JavaDemo {  **public static void** main(String args[]) {  IFood food=**new** breadFactory().creatFood();  food.eat();  } }  运行结果：吃面包。 |

抽象工厂模式

意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

主要解决：主要解决接口选择的问题。

何时使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

如何解决：在一个产品族里面，定义多个产品。每个具体的工厂负责一个产品族。抽象工厂的返回值为最高级抽象产品。

关键代码：在一个工厂里聚合多个同类产品（在同一个产品族中）。

应用实例：举一个衣服与衣柜的例子。家里边，有男装（产品族，其中包含休闲男装和商务男装）、女装（产品族，其中包含休闲女装和商务女装）。商务女装、商务男装、时尚女装、时尚男装，这些都是具体产品。男装专门放在男衣柜（具体工厂）中，女装专门放在女衣柜（具体工厂）中。当我们需要拿衣服时候，从衣柜（抽象工厂）中获取。

创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

|  |
| --- |
| public interface Sender {  public void Send();  }  //两个实现类  public class MailSender implements Sender {  @Override  public void Send() {  System.out.println("this is mailsender!");  }  }  public class SmsSender implements Sender {  @Override  public void Send() {  System.out.println("this is sms sender!");  }  }  //两个工厂类  public class SendMailFactory implements Provider {  @Override  public Sender produce(){  return new MailSender();  }  }  public class SendSmsFactory implements Provider{  @Override  public Sender produce() {  return new SmsSender();  }  }  //工厂接口  public interface Provider {  public Sender produce();  }  public class Test {  public static void main(String[] args) {  Provider provider = new SendMailFactory();  Sender sender = provider.produce();  sender.Send();  }  } |

这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现Sender接口，同时做一个工厂类，实现Provider接口，就OK了，无需去改动现成的代码。这样做，拓展性较好！