1 简单工厂模式

设计模式是一套解决软件开发过程中某些常见问题的通用解决方案，是已被反复使用且证明其有效性的设计经验的总结。目的是建立具有可复用、可维护、可扩展的软件系统。

- **创建型模式**，共 5 种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
- **结构型模式**，共 7 种：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
- **行为型模式**，共 11 种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

创建型的设计模式: 创建模式是对类的实例化过程的抽象。一些系统在创建对象时，需要动态地决定怎样创建对象，创建哪些对象，以及如何组合和表示这些对象，实现对象的“创建”与“使用”相分离。

简单工厂模式:

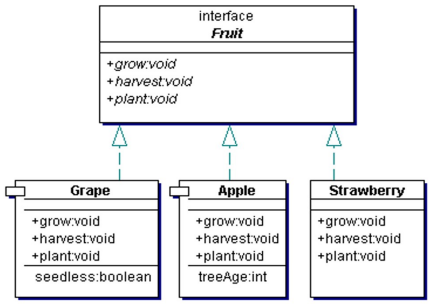
- 简单工厂模式又叫做静态工厂方法 (Static Factory Method) 模式。
- 简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。k
- 工厂模式专门负责将大量有共同接口的类实例化。工厂模式可以动态决定将哪一个类实例化，不必事先知道每次要实例化哪一个类。

```
class 消费者 {
  public void 消费() {
    鼠标 obj1 = 工厂.生产( “鼠标” );
    铅笔 obj2 = 工厂.生产( “铅笔” );
    // ...
  }
}
```

1.1 简单工厂模式的引进

比如说有一个生产水果的农场，专门向市场销售各类水果。在这个系统里需要描述下列的水果：

- 葡萄 Grape
- 草莓 Strawberry
- 苹果 Apple



```java
public interface Fruit {
  void grow(); //生长
  void harvest(); //收获
  void plant(); //种植
}
```

```java
public class Apple implements Fruit {
  private int treeAge;
  //生长
  public void grow() { System.out.println("Apple is growing"); }
  //收获
  public void harvest() { System.out.println("Apple has been harvested"); }
  //种植
  public void plant() { System.out.println("Apple has been planted"); }
  //树龄的取值方法
  public int getTreeAge() { return treeAge; }
  //树龄的赋值方法
  public void setTreeAge(int treeAge) { this.treeAge = treeAge; }
}
```

```java
public class Grape implements Fruit {
  private boolean seedless;
  //生长
  public void grow() { System.out.println ("Grape is growing"); }
  //收获
  public void harvest() { System.out.println("Grape has been harvested"); }
  //种植
  public void plant() { System.out.println("Grape has been planted"); }
  //有无籽的取值方法
  public boolean getSeedless() { return seedless; }
  //有无籽的赋值方法
  public void setSeedless(boolean seedless) { this.seedless = seedless; }
}
```

```java
public class Strawberry implements Fruit {
  public void grow() {
    System.out.println("Strawberry is growing");
  }
  public void harvest() {
    System.out.println("Strawberry has been harvested");
  }
  public void plant() {
```

```
    System.out.println("Strawberry has been planted");
  }
}
```

农场的园丁也是系统的一部分，由 FruitGardener 类代表。其结构由下面的类图描述。FruitGardener 类会根据客户端的要求，创建出不同的水果对象，比如苹果 (Apple)，葡萄 (Grape) 或草莓 (Strawberry) 的实例。

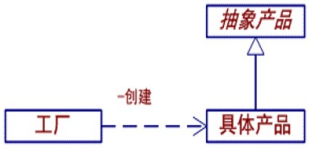
```
public class FruitGardener {
  // 静态工厂方法
  public static Fruit factory(String which)
  throws BadFruitException {
    if (which.equalsIgnoreCase("apple")) {
      return new Apple();
    } else if (which.equalsIgnoreCase("strawberry")) {
      return new Strawberry();
    } else if (which.equalsIgnoreCase("grape")) {
      return new Grape();
    } else {
      throw new BadFruitException("Bad fruit request");
    }
  }
}

class BadFruitException extends Exception {
  public BadFruitException(String msg) {
    super(msg);
  }
}
```

可以看出，园丁类提供了一个静态工厂方法。在客户端的调用下，这个方法创建客户端所需要的水果对象。如果客户端的请求是系统所不支持的，工厂方法就会抛出一个 BadFruitException 异常。在使用时，客户端只需调用 FruitGardener 的静态方法 factory() 即可。

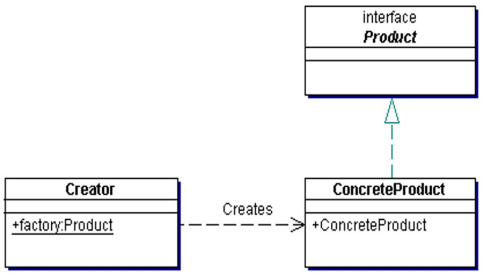
```
try {
  FruitGardener.factory("grape");
  FruitGardener.factory("apple");
  FruitGardener.factory("strawberry");
  // ...
} catch(BadFruitException e) {
  // ...
}
```

1.2 简单工厂模式的结构

简单工厂模式是类的创建模式，这个模式的一般性结构如下图所示。



角色与结构: 简单工厂模式就是由一个工厂类可以根据传入的参数决定创建出哪一种产品类的实例。下图所示为以一个示意性的实现为例说明简单工厂模式的结构。



从上图可看出, 简单工厂模式涉及到工厂角色、抽象产品角色及具体产品角色三个角色:

- 工厂类 (Creator) 角色：担任这个角色的是工厂方法模式的核心，含有与应用紧密相关的业务逻辑。工厂类在客户端的直接调用下创建产品对象，它往往由一个具体 Java 类实现。
- 抽象产品 (Product) 角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口。抽象产品角色可以用一个 Java 接口或者 Java 抽象类实现。
- 具体产品 (Concrete Product) 角色：工厂方法模式所创建的任何对象都是这个角色的实例，具体产品角色由一个具体 Java 类实现。

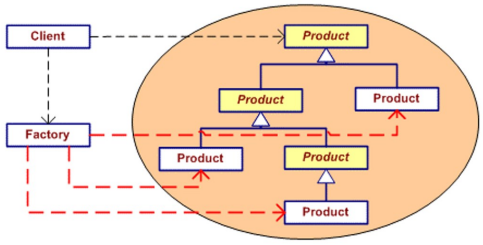
抽象产品角色的主要目的是给所有的具体产品类提供一个共同的类型，在最简单的情况下，可以简化为一个标识接口。

```
1  public class Creator {
2    // 静态工厂方法
3    public static Product factory() {
4      return new ConcreteProduct();
5    }
6  }
7  public interface Product { }
8  public class ConcreteProduct implements Product {
9    public ConcreteProduct() { }
10 }
```

如果系统仅有一个具体产品角色的话，那么就可以省略掉抽象产品角色。

1.3 简单工厂模式的实现

多层次的产品结构: 在实际应用中，产品可以形成复杂的等级结构. 这个时候，简单工厂模式采取的是以不变应万变的策略，一律使用同一个工厂类。如下图所示。图中从 Factory 类到各个 Product 类的虚线代表创建 (依赖) 关系；从 Client 到其他类的联线是一般依赖关系。



这样做的好处是设计简单，产品类的等级结构不会反映到工厂类中来。但是这样做的缺点是，增加新的产品必将导致工厂类的修改.

使用 Java 接口或者 Java 抽象类:

- 如果具体产品类彼此之间没有共同的业务逻辑，那么抽象产品角色可以由一个 Java 接口扮演；
- 相反，如果这些具体产品类彼此之间确有共同的业务逻辑，那么这些公有的逻辑就应当移到抽象角色里面，这就意味着抽象角色应当由一个抽象类扮演。在一个类型的等级结构里面，共同的代码应当尽量向上移动，以达到共享的目的，如下图所示。

工厂角色与抽象产品角色合并: 在有些情况下，工厂角色可以由抽象产品角色扮演。典型的应用就是 java.text.DateFormat 类，一个抽象产品类同时是子类的工厂.

```
1  Date date = new Date();
2  //日期格式，精确到日 2017-4-16
3  DateFormat df1 = DateFormat.getDateInstance();
4  System.out.println(df1.format(date));
5
6   //可以精确到秒 2017-4-16 12:43:37
7  DateFormat df2 = DateFormat.getDateTimeInstance();
8  System.out.println(df2.format(date));
9
10 //只显示出时时分秒 12:43:37
11 DateFormat df3 = DateFormat.getTimeInstance();
12 System.out.println(df3.format(date));
```

三个角色全部合并: 如果抽象产品角色已经被省略，而工厂角色就可以与具体产品角色合并。换言之，一个产品类为创建自身的工厂. 显然，三个原本独立的角色：工厂角色、抽象产品以及具体产品角色都已经合并成为一个类，这个类自行创建自己的实例。

```
1  public class ConcreteProduct {
2    public ConcreteProduct() { }
3    // 静态工厂方法
```

```
4   public static ConcreteProduct factory() {
5     return new ConcreteProduct();
6   }
7 }
```

这种退化的简单工厂模式与单例模式以及多例模式有相似之处，但是并不等于单例或者多例模式。

产品对象的循环使用和登记式的工厂方法: 在很多情况下，产品对象可以循环使用。换言之，工厂方法可以循环使用已经创建出来的对象，而不是每一次都创建新的产品对象。如果工厂方法总是循环使用同一个产品对象，那么这个工厂对象可以使用一个属性来存储这个产品对象。每一次客户端调用工厂方法时，工厂方法总是提供这同一个对象。

```
1  class Factory {
2    static Product obj;
3    public static void factory(int type) {
4      // ...
5      if(obj==null)
6      obj=new ConcreteProduct();
7      return obj;
8      // ...
9    }
10 }
```

1.4 简单工厂模式的优缺点

优点：客户端则可以免除直接创建产品对象的责任，而仅仅负责消费产品。对于消费者角色来说，任何时候需要某种产品，只需要向工厂角色 (下订单) 请求即可，而无需知道产品创建细节。实现了客户端类与产品类的解耦。

缺点：当产品种类增加时，工厂类的工厂方法也必须随之修改。

这个工厂类集中了所有的产品创建逻辑，形成一个无所不知的全能类，有人把这种类叫做上帝类 (God Class)。如果这个全能类代表的是农场的一个具体园丁的话，那么这个园丁就需要对所有的产品负责，成了农场的关键人物，他什么时候不能正常工作了，整个农场都要受到影响。

对开闭原则的支持: 开闭原则要求一个系统的设计能够允许系统在无需修改的情况下，扩展其功能。那么简单工厂模式是否满足这个条件? 要回答这个问题，首先需要将系统划分成不同的子系统，再考虑功能扩展对于这些子系统的要求。一般而言，一个系统总可以划分成为产品的消费者角色（Client）、产品的工厂角色（Factory）以及产品角（Product）三个子系统. 在这个系统中，功能的扩展体现在引进新的产品上。开闭原则要求系统允许当新的产品加入系统中，而无需对现有代码进行修改。这一点对于产品的消费角色是成立的，而对于工厂角色是不成立的。简单工厂角色只在有限的程度上支持开闭原则。