面向对象编程-抽象类与接口

> ## 本节目标
>
> 1. 抽象类的定义与使用
> 2. 模板设计模式
> 3. 接口的定义与使用
> 4. 工厂设计模式
> 5. 代理设计模式
> 6. 抽象类与接口的区别

在以后进行项目开发的过程中，尽可能不要直接继承直接实现好的类，而继承抽象类或接口

对象多态性的核心本质：*方法的覆写*

如果子类没有进行指定方法的覆写，也就不存在对象多态性了。

所以如果要对子类的方法做一些强制性的要求，就会用到抽象类。

1. 抽象类的定义与使用

1.1 抽象类的定义与使用

抽象类只是在普通类的基础上扩充了一些抽象方法而已，所谓的抽象方法指的是只声明而未实现的方法（即没有方法体）。所有抽象方法要求使用abstract关键字来定义，并且抽象方法所在的类也一定要使用abstract关键字来定义，表示抽象类

范例：定义一个抽象类

```
abstract class Person{
    private String name ; // 属性
    public String getName(){ // 普通方法
        return this.name;
    }
    public void setName(String name){
        this.name = name ;
    }
    // {}为方法体，所有抽象方法上不包含方法体
    public abstract void getPersonInfo() ; //抽象方法

}
```

通过上述代码我们会发现，抽象类就是比普通类多了一些抽象方法而已。

抽象类中包含有抽象方法，而抽象方法不包含方法体，即没有具体实现。因此抽象类不能直接产生实例化对象。

对于抽象类的使用原则：

- 所有的抽象类必须有子类。
- 抽象类的子类必须覆写抽象类的所有抽象方法（子类不是抽象类）【方法覆写一定要考虑权限问题，权限尽量都用public】
- 抽象类的对象可以通过对象多态性利用子类为其实例化
- private与abstract不能同时使用。

范例：使用抽象类

```
abstract class Person{
    private String name ; // 属性
    public String getName(){ // 普通方法
        return this.name;
    }
    public void setName(String name){
        this.name = name ;
    }
    // {}为方法体，所有抽象方法上不包含方法体
    public abstract void getPersonInfo() ; //抽象方法

}

class Student extends Person{
    public void getPersonInfo(){
        System.out.println("I am a student");
    }
}

public class Test{
    public static void main(String[] args) {
        Person per = new Student() ; //实例化子类，向上转型
        per.getPersonInfo() ; //被子类所覆写的方法
    }
}
```

从正常开发角度来讲，以上操作就是抽象类使用的标准操作并且是使用最多形式。但是你以后会见到如下形式：

```
abstract class Person{
    private String name ; // 属性
    public String getName(){ // 普通方法
        return this.name;
    }
    public void setName(String name){
        this.name = name ;
    }
    // {}为方法体，所有抽象方法上不包含方法体
```

```
    public abstract void getPersonInfo() ; //抽象方法

    public static Person getInstance() { //取得A类对象
        class Student extends Person{   //定义抽象类的子类 （内部类）
            public void getPersonInfo(){
            System.out.println("I am a student");
            }
        }
        return new Student();
    }

}
public class Test{
    public static void main(String[] args) {
        Person per = Person.getInstance();
        per.getPersonInfo() ; //被子类所覆写的方法
    }
}
```

此类模式属于非正常模式，但是对于一些封装性有一定好处（封装具体子类），不属于开发首选

1.2 抽象类相关规定

- 抽象类只是比普通类多了一些抽象方法而已

因此在抽象类中也允许提供构造方法，并且子类也照样遵循对象实例化流程。实例化子类时一定先调用父类构造方法。

范例：在抽象类中定义构造方法

```
abstract class Person{
    private String name ; // 属性

    public Person(){ //构造方法
        System.out.println("**********");
    }
    public String getName(){ // 普通方法
        return this.name;
    }
    public void setName(String name){
        this.name = name ;
    }
    // {}为方法体，所有抽象方法上不包含方法体
    public abstract void getPersonInfo() ; //抽象方法
}

class Student extends Person{
    public Student(){ //构造方法
        System.out.println("##########");
```

```
    }
    public void getPersonInfo(){
        //空实现。
    }
}

public class Test{
    public static void main(String[] args) {
        new Student();
    }
}
```

如果父类没有无参构造，那么子类构造必须使用super明确指出使用父类哪个构造方法。

范例：一段特殊代码

```
abstract class A{
    public A(){ //3.调用父类构造
        this.print() ; //4.调用被子类覆写的方法
    }
    public abstract void print() ;
}
class B extends A{
    private int num = 100 ;
    public B(int num) { //2.调用子类实例化对象
        super() ; //3.隐含一行语句，实际要先调用父类构造
        this.num = num ; //7.为类中属性初始化
    }
    public void print() { //5.此时子类对象的属性还没有被初始化
        System.out.println(this.num) ; //6.对应其数据类型的默认值
    }
}

public class Test{
    public static void main(String[] args) {
        new B(30) ; //1.实例化子类对象
    }
}
```

结论：如果构造方法，那么对象中的属性一定都是其对应数据类型的默认值。

额外话题：关于对象实例化

对象的实例化操作实际上需要以下几个核心步骤：

> - 进行类加载
> - 进行类对象的空间开辟
> - 进行类对象中的属性初始化(构造方法)

- 抽象类中允许不定义任何的抽象方法，但是此时抽象类依然无法直接创建实例化对象

```
abstract class A{
    public void print(){
        //空实现，普通方法
    }
}

public class Test{
    public static void main(String[] args) {
        A a = new A() ; // 错误：A是抽象的；无法实例化
    }
}
```

- 抽象类一定不能使用final声明，因为使用final声明的类不允许有子类；而抽象类必须有子类；相应的，抽象方法也不能使用private定义，因为抽象方法必须要能被覆写
- 抽象类也分为内部抽象类和外部抽象类。内部抽象类中也可以使用static定义来描述外部抽象类

范例：观察内部抽象类

```
abstract class A{ //此类结构出现几率很低
    public abstract void printA();
    abstract class B {
        public abstract void printB();
    }
}
class X extends A {
    public void printA(){}
    class Y extends B {
        public void printB(){}
    }
}
```

如果现在外部抽象类中使用了static那么就是语法错误，**但是内部抽象类允许使用static**

范例：内部抽象类使用static修饰

```
abstract class A{ //此类结构出现几率很低
    public abstract void printA() ;
    static abstract class B {
        public abstract void printB() ;
    }
}
class X extends A.B {
        public void printB(){}
}
```

从一般的设计角度来讲，上述结构出现概率很低。

2.模板设计模式

开闭原则(OCP): 一个软件实体如类、模块和函数应该对扩展开放、对修改关闭。

开闭原则是Java世界中最基础的设计原则。

模版设计模式是抽象类的一个实际应用场景，讲模板设计模式之前，我们先来看一下星巴克咖啡冲泡师傅的训练手册。

星巴克咖啡冲泡法

---

1. 将水煮沸
2. 用沸水冲泡咖啡
3. 将咖啡倒进杯子
4. 加糖和牛奶

星巴克茶冲泡法

---

1. 将水煮沸
2. 用沸水浸泡茶叶
3. 把茶倒进杯子
4. 加柠檬

下面我们用代码来实现咖啡和茶的类

范例:冲泡咖啡类

```
class Coffee {
    /**
     * 咖啡冲泡法(算法)
     */
    void prepareRecipe() {
        boilWater();
        brewCoffeeGrings();
        pourInCup();
        addSugarAndMilk();
    }
    /**
     * 将水煮沸
     */
    public void boilWater() {
        System.out.println("Boiling Water");
    }

    /**
     * 冲泡咖啡
     */
    public void brewCoffeeGrings() {
        System.out.println("Dripping Coffee through filter");
```

```
    }

    /**
     * 把咖啡倒进杯子中
     */
    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    /**
     * 加糖和牛奶
     */
    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

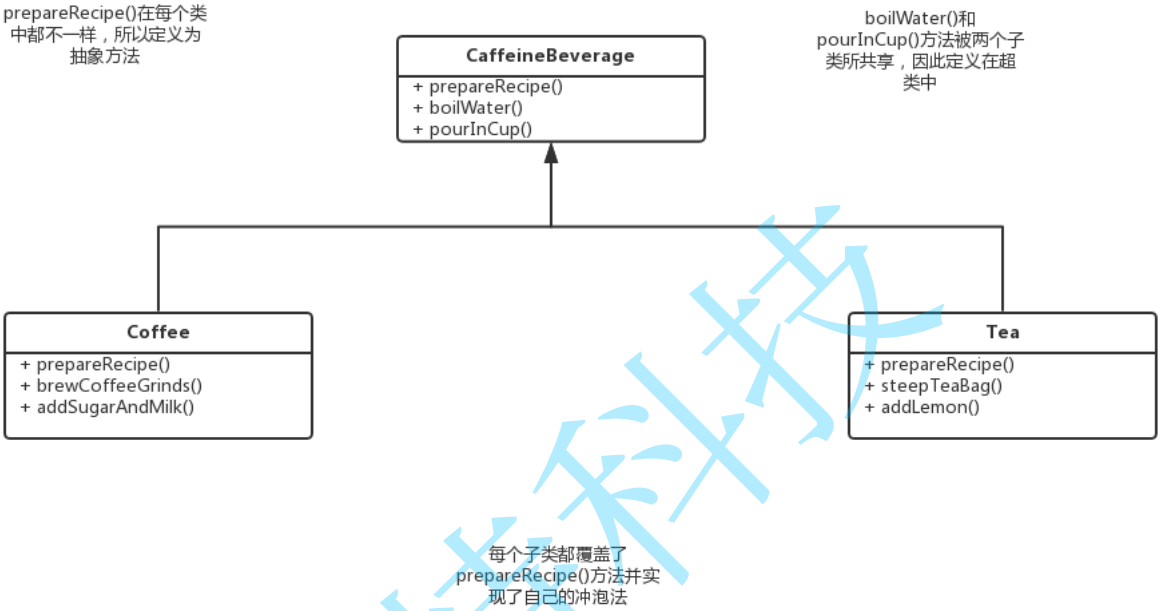
范例:冲泡茶类

```
class Tea {
    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }
    /**
     * 将水煮沸
     */
    public void boilWater() {
        System.out.println("Boiling Water");
    }

    /**
     * 冲泡茶
     */
    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    /**
     * 把茶倒进杯子中
     */
    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    /**
     * 加柠檬
```

```
     */
    public void addLemon() {
        System.out.println("Adding Lemon");
    }
}
```

我们在这两个类中发现了重复代码，因此我们需要重新理一下我们的设计。

既然茶和咖啡是如此的相似，因此我们应该将共同的部分抽取出来，放进一个基类中。



来看咱们重新设计的第一个版本。除了两个方法有共同点之外，咖啡和茶还有什么共同点呢?

让我们先从冲泡法入手。观察咖啡和茶的冲泡法我们会发现，两种冲泡法都采用了相同的算法:

> 1. 将水煮沸
> 2. 用热水泡饮料
> 3. 把饮料倒进杯子
> 4. 在饮料内加入适当的调料

那么。我们是否有办法将prepareRecipe()也抽象化吗?

实际上,浸泡(steep)和冲泡(brew)差异并不大。因此我们给它一个新的方法名称brew(),这样我们无论冲泡的是何种饮料都可以使用这个方法。同样的，加糖、牛奶还是柠檬也很相似，都是在饮料中加入其它调料，因此我们也给它一个通用名称addCondiments()。重新设计后通用的prepareRecipe()方法如下:

```
    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
```

下面我们来分别实现超类与子类。

范例:抽象基类实现

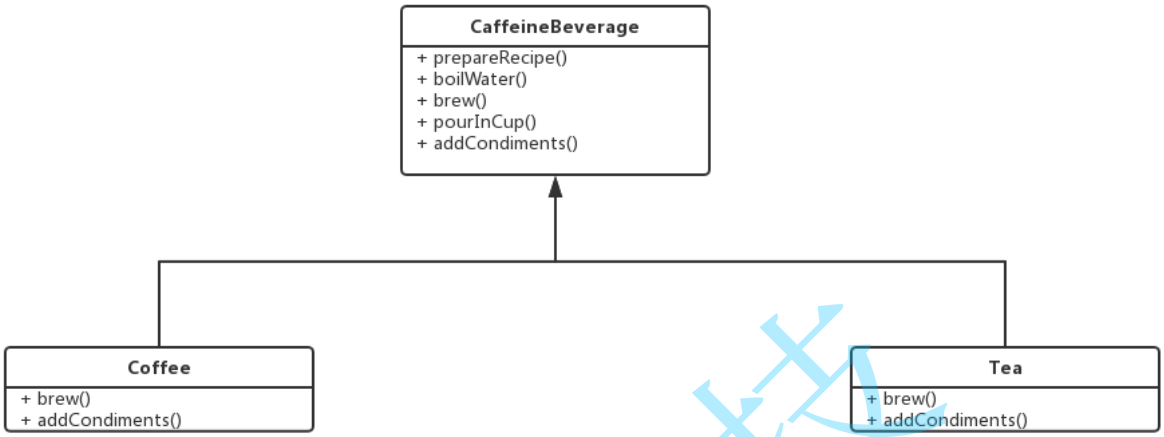
```
/**
 * 咖啡因饮料是一个抽象类
 */
abstract class CaffeineBeverage {
    /**
     * 现在用同一个prepareRecipe()方法处理茶和咖啡。
     * 声明为final的原因是我们不希望子类覆盖这个方法!
     */
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    /**
     * 咖啡和茶处理这些方法不同，因此这两个方法必须被声明为抽象，留给子类实现
     */
    abstract void brew();
    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

现在我们需要处理茶和咖啡类了。这两个类现在都是依赖超类来处理冲泡流程，因此只需要各自处理冲泡和加调料部分即可。

```
class Tea extends CaffeineBeverage {
    void brew() {
        System.out.println("Steeping the tea");
    }
    void addCondiments() {
        System.out.println("Adding Lemon");
    }
}

class Coffee extends CaffeineBeverage {
    void brew() {
        System.out.println("Dripping Coffee through filter");
```

```
    }
    void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

我们来看此时的类图



我们刚刚实现的就是模板设计模式，它包含了实际的"模板方法"

模板方法定义了一个算法的步骤，并允许子类为一个或者多个步骤提供具体实现。

那么，究竟模板方法能带给我们什么呢?

不好的茶或咖啡实现	模板方法提供的咖啡因饮料
Coffee或Tea主导一切，控制算法	由超类主导一切，它拥有算法，并且保护这个算法
Coffee与Tea之间存在重复代码	有超类的存在，因此可以将代码复用最大化
对于算法所做的代码改变，需要打开各个子类修改很多地方	算法只存在一个地方，容易修改
弹性差，新种类的饮料加入需要做很多工作	弹性高，新饮料的加入只需要实现自己的冲泡和加料方法即可
算法的知识和它的实现分散在许多类中	超类专注于算法本身，而由子类提供完整的实现。

模板方法模式:

在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

下面我们来看一个完整的模板模式超类的定义:

```
/**
```

```
 * 基类声明为抽象类的原因是
 * 其子类必须实现其操作
 */
abstract class AbstractClass {
    /**
     * 模板方法，被声明为final以免子类改变这个算法的顺序
     */
    final void templateMethod() {

    }

    /**
     * 具体操作延迟到子类中实现
     */
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();

    /**
     * 具体操作且共用的方法定义在超类中，可以被模板方法或子类直接使用
     */
    final void concreteOperation() {
        // 实现
    }

    /**
     * 钩子方法是一类"默认不做事的方法"
     * 子类可以视情况决定要不要覆盖它们。
     */
    void hook() {
        // 钩子方法
    }
}
```

扩展上述类，引入"钩子"方法

超类实现:

```
abstract class CaffeineBeverage {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        // 如果顾客想要饮料我们才调用加料方法
        if (customerWantsCondiments()){
            addCondiments();
        }
    }

    abstract void brew();
```

```
    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    /**
     * 钩子方法
     * 超类中通常是默认实现
     * 子类可以选择性的覆写此方法
     * @return
     */
    boolean customerWantsCondiments() {
        return true;
    }
}
```

子类实现:

```
class Tea extends CaffeineBeverage {
    void brew() {
        System.out.println("Steeping the tea");
    }
    void addCondiments() {
        System.out.println("Adding Lemon");
    }
}

class Coffee extends CaffeineBeverage {
    void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    /**
     * 子类覆写了钩子函数，实现自定义功能
     * @return
     */
    public boolean customerWantsCondiments() {
        String answer = getUserInput();
        if (answer.equals("y")) {
            return true;
```

```
        }else {
            return false;
        }
    }
    private String getUserInput() {
        String answer = null;
        System.out.println("您想要在咖啡中加入牛奶或糖吗 (y/n)?");
        Scanner scanner = new Scanner(System.in);
        answer = scanner.nextLine();
        return answer;
    }
}
```

测试类:

```
public class Test {
    public static void main(String[] args) {
        CaffeineBeverage tea = new Tea();
        CaffeineBeverage coffee = new Coffee();

        System.out.println("\nMaking tea...");
        tea.prepareRecipe();

        System.out.println("\nMaking Coffee");
        coffee.prepareRecipe();

    }
}
```

最具有代表性的就是后面要学习的Servlet

3.接口的定义与使用

抽象类与普通类相比最大的特点是约定了子类的实现要求，但是抽象类存在单继承局限。如果要约定子类的实现要求并避免单继承局限就需要使用接口。

在以后的开发过程之中：接口优先（在一个操作既可以使用抽象类又可以使用接口的时候，优先考虑使用接口）

3.1 接口的基本概念

接口定义：接口就是抽象方法和全局常量的集合，在Java中接口使用interface关键字定义

范例：定义一个简单接口

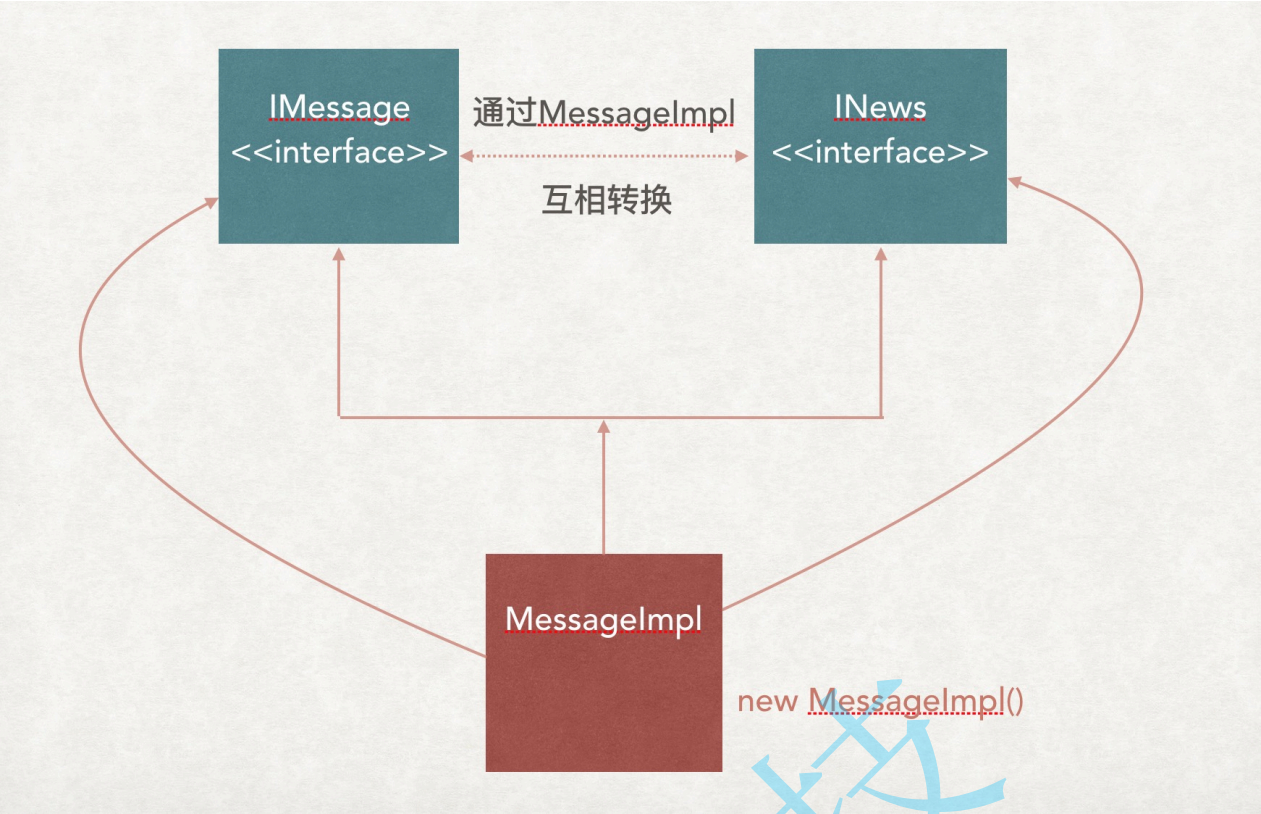
```
interface IMessage{
        public static final String MSG = "I am a biter" ; // 全局常量
        public abstract void print() ; // 抽象方法
}
```

为了区分接口。建议在所有接口前面追加字母I。

子类如果要想使用接口，那么就必须使用implements关键字来实现接口，同时，一个子类可以实现多个接口，【可以使用接口来实现多继承的概念】对于接口的子类（不是抽象类）必须覆写接口中的全部抽象方法。随后可以利用子类的向上转型通过实例化子类来得到接口的实例化对象。

范例：观察子类实现接口&父接口间的相互转换

```
interface IMessage{
        public static final String MSG = "I am a biter" ; // 全局常量
        public abstract void print() ; // 抽象方法
}
interface INews {
        public abstract String getNews() ;
}
class MessageImpl implements IMessage,INews {
        public void print() {
                System.out.println(IMessage.MSG) ;
        }
        public String getNews(){
                return IMessage.MSG ; // 访问常量都建议加上类名称
        }
}

public class Test{
        public static void main(String[] args) {
                IMessage m = new MessageImpl() ; //子类向上转型,为父接口实例化对象
                m.print() ; // 调用被子类覆写过的方法
                INews n = (INews) m ;
                System.out.println(n.getNews()) ;
        }
}
```



真正new的子类才有意义，别被前面的类名称搞晕。

3.2 接口使用限制

- 接口中只允许public权限。（不管是属性还是方法，其权限都是public）

范例：观察错误覆写。

```
interface IMessage{
        abstract void print() ; // 即便不写public，也是public
}

class MessageImpl implements IMessage {
        void print() { //
                System.out.println("I am a biter") ; //权限更加严格了，所以无法覆
写。
        }
}
```

只要是方法就使用public进行定义

由于接口之中只是全局常量和抽象方法的集合，所以以下两种定义格式效果都是一样的。

完整格式	简化格式

interface IMessage{
public static final String MSG = "I am a biter" ;
public abstract void print() ;
}
	interface IMessage{
	String MSG = "I am a biter" ;
	void print() ;
	}

在以后编写接口的时候，99%的接口只提供抽象方法。很少在接口里提供全局常量；

阿里编码规约：接口中的方法和属性不要加任何修饰符号,public也不要加，保持代码的简洁性。

- 当一个子类即需要实现接口又需要继承抽象类时，请先使用extends继承**一个**抽象类，而后使用implements实现**多个**接口。

范例：子类继承抽象类和实现接口。

```
interface IMessage {
        public void print() ;
}
abstract class News {
        // 抽象类中方法前面的abstract不能省略，否则就是普通方法
        public abstract void getNews() ;
}

class MessageImpl extends News implements IMessage {
        public void print() {
                System.out.println("I am a biter") ;
        }
        public void getNews() {
                System.out.println("I am News") ;
        }
}

public class Test{
        public static void main(String[] args) {
                IMessage m = new MessageImpl() ;
                m.print() ;
                // MessageImpl是抽象类和接口的共同子类
                News news = (News) m ;
                news.getNews() ;
        }
}
```

- 一个抽象类可以使用implements实现多个接口，但是接口不能继承抽象类。

范例：抽象类实现接口

```
interface IMessage {
        public void print() ;
}
abstract class News implements IMessage{
    //News为抽象类，可以不实现IMessage中的抽象方法
        // 抽象类中方法前面的abstract不能省略，否则就是普通方法
        public abstract void getNews() ;
}

class MessageImpl extends News {
        public void print() {
                System.out.println("I am a biter") ;
        }
        public void getNews() {
                System.out.println("I am News") ;
        }
}

public class Test{
        public static void main(String[] args) {
                IMessage m = new MessageImpl() ;
                m.print() ;
                // MessageImpl是抽象类和接口的共同子类
                News news = (News) m ;
                news.getNews() ;
        }
}
```

实际上此时的结构关系属于三层继承

在以后读许多第三方类库的时候，可能会出现如下代码：

```
class MessageImpl extends News implements IMessage
```

此时`implements IMessage`只是为了强调`MessageImpl`是`IMessage`的实现类。

- 一个接口可以使用extends继承多个父接口。

```
interface A {
        public void printA() ;
}
interface B {
        public void printB() ;
}
interface C extends A,B { // 接口多继承
        public void printC() ;
```

```
}
class Impl implements C{
        public void printA() {}
        public void printB() {}
        public void printC() {}
}

public class Test{
        public static void main(String[] args) {
        }
}
```

- 接口可以定义一系列的内部结构，包括：内部普通类，内部普通类，内部接口。其中，使用static定义的内部接口就相当于一个外部接口。

范例：使用static定义的内部接口

```
interface A {
        public void printA() ;
        static interface B {
                public void printB() ; // 使用static定义，描述一个外部接口
        }
}

class Impl implements A.B {
        public void printB() {}
}

public class Test{
        public static void main(String[] args) {
        }
}
```

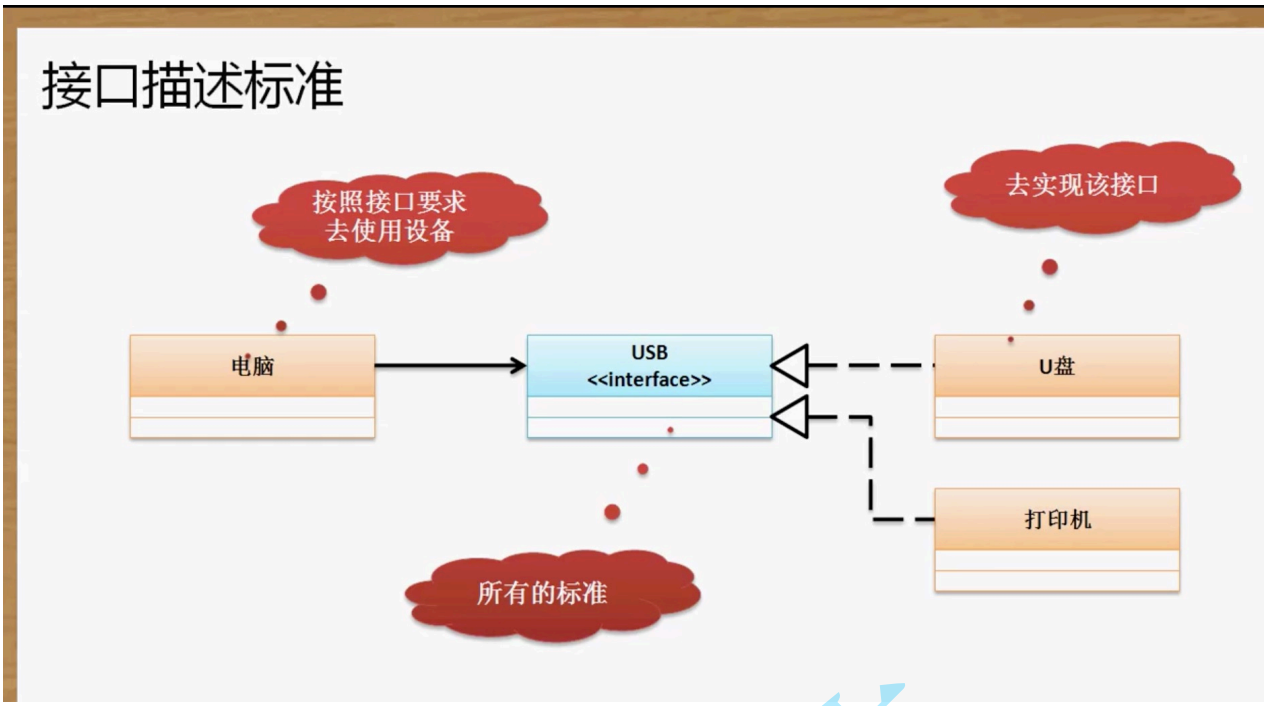
对于内部结构依然不是我们首选，要想清楚接口的实际开发意义，需要大量的项目和代码来论证。目前我们了解接口的定义和使用即可。

3.3 接口的应用

接口在实际开发之中有三大核心应用环境：

- ①定义操作标准
- ②表示能力
- ③在分布式开发之中暴露远程服务方法

现在要求描述一个概念-电脑上可以使用任何的`USB`设备（U盘，打印机等等）：



范例：定义一个usb标准

```
interface USB {
        public void setup() ; // 安装USB驱动
        public void work() ; // 进行工作
}
```

范例：定义电脑类

```
class Computer {
        public void plugin(USB usb) { // 只能插USB设备
                usb.setup() ; // 安装
                usb.work () ; // 工作

        }
}
```

范例：定义USB子类

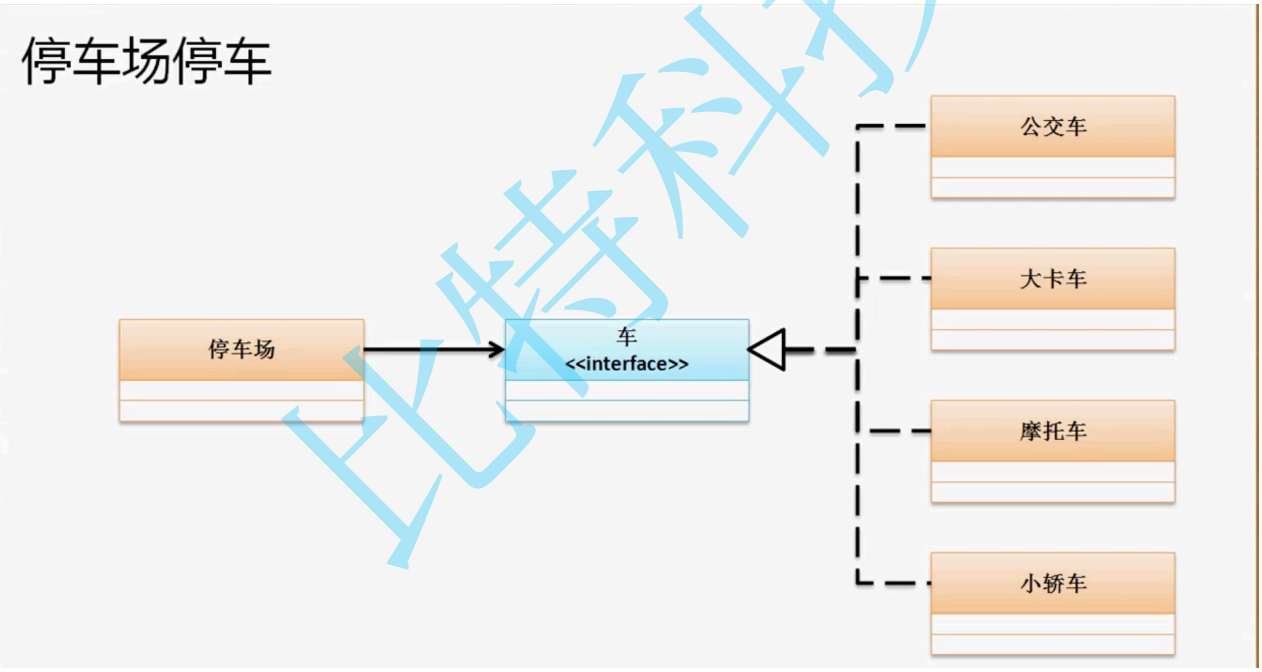
```
class UDisk implements USB{ // 定义一个Usb设备
        public void setup() {
                System.out.println("安装U盘驱动") ;
        }
        public void work() {
                System.out.println("U盘开始工作") ;
        }
}

class PrintDisk implements USB{ // 定义一个Usb设备
        public void setup() {
```

```
                System.out.println("安装打印机驱动") ;
        }
        public void work() {
                System.out.println("打印机开始工作") ;
        }
}
```

范例：测试类

```
public class Test{
        public static void main(String[] args) {
                Computer computer = new Computer() ;
                computer.plugin(new UDisk()) ;
                computer.plugin(new PrintDisk()) ;
        }
}
```

通过以上代码我们发现：接口和对象多态性的概念结合之后，对于参数的统一更加明确了。而且可以发现接口是在类之上的设计抽象。



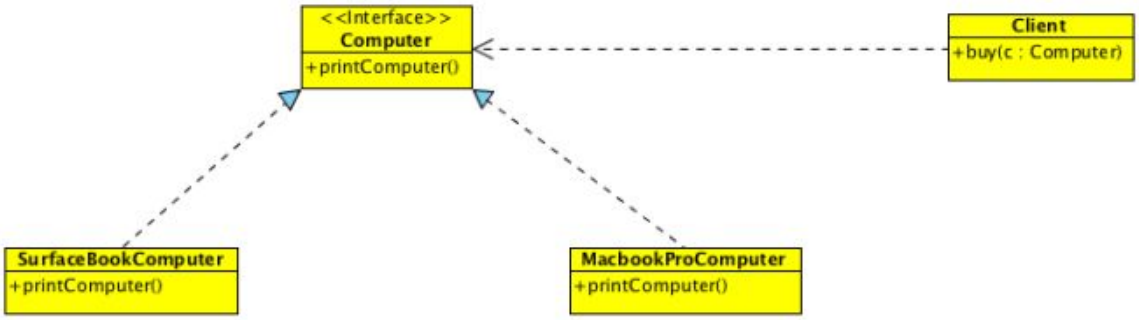
4.工厂设计模式（重点）

Java基础要求掌握三种重点设计模式：**工厂、代理、单例**。

思考如下场景：

有一天，刘同学准备去买笔记本，他到商城发现有两款电脑他特别喜欢, 一款是 Macbook Pro, 另一款是 Surface Pro。

根据以上的场景，类图可以如下表示：




```
interface Computer {
    void printComputer();
}

class MacbookProComputer implements Computer {

    public void printComputer() {
        System.out.println("This is a MacbookPro");
    }
}

class SurfaceBookComputer implements Computer {

    public void printComputer() {
        System.out.println("This is a SurfaceBook");
    }
}

public class Client {
    public void buyComputer(Computer computer) {
        computer.printComputer();
    }
    public static void main(String[] args) {
        Client client = new Client();
        client.buyComputer(new MacbookProComputer());
    }
}
```

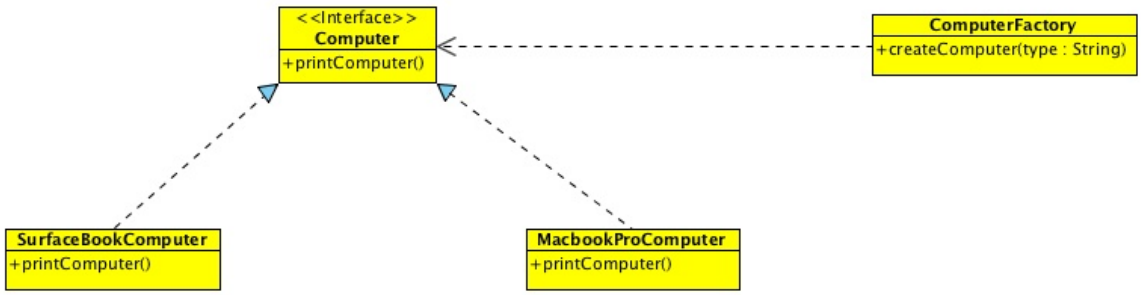

这时候问题就来了，如果此时刘同学又看上了一款外星人(Alienware)笔记本，我们就不得不返回客户端去修改代码，让客户端支持Alienware笔记本。那么，如何将实例化具体类的代码从客户端中抽离，或者封装起来，使它们不会干扰应用的其他部分呢?

4.1 简单工厂模式

简单工厂模式：专门定义一个类用来创建其它类的实例，被创建的实例通常都具有共同的父类。

这里我们相当于是创建生产电脑的工厂，客户需要购买什么样的电脑，只要输入类型编号就可以获取该电脑。**将类的实例化交给工厂易于解耦。**

类图如下所示：




```
import java.util.Scanner;

interface Computer {
    void printComputer();
}

class MacbookProComputer implements Computer {

    public void printComputer() {
        System.out.println("This is a MacbookPro");
    }
}

class SurfaceBookComputer implements Computer {

    public void printComputer() {
        System.out.println("This is a SurfaceBook");
    }
}

class ComputerFactory {
    public static Computer getInstance(String type) {
        Computer computer = null;
        if (type.equals("macbook")) {
            computer = new MacbookProComputer();
        }else if (type.equals("surface")) {
            computer = new SurfaceBookComputer();
        }
        return computer;
    }
}

public class Client {
    public void buyComputer(Computer computer) {
        computer.printComputer();
    }
```

```
    public static void main(String[] args) {
        Client client = new Client();
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入您想要的电脑型号...");
        String type = scanner.nextLine();
        Computer computer = ComputerFactory.getInstance(type);
        client.buyComputer(computer);
    }
}
```

以上就是简单工厂模式

> 概要
>
> 1. 一个抽象产品类
> 2. 具体产品类
> 3. 一个工厂

优点：

- 简单易于实现
- 把类的实例化交给工厂,易于解耦

缺点：

- 添加具体产品需要修改工厂违反OCP开放封闭原则

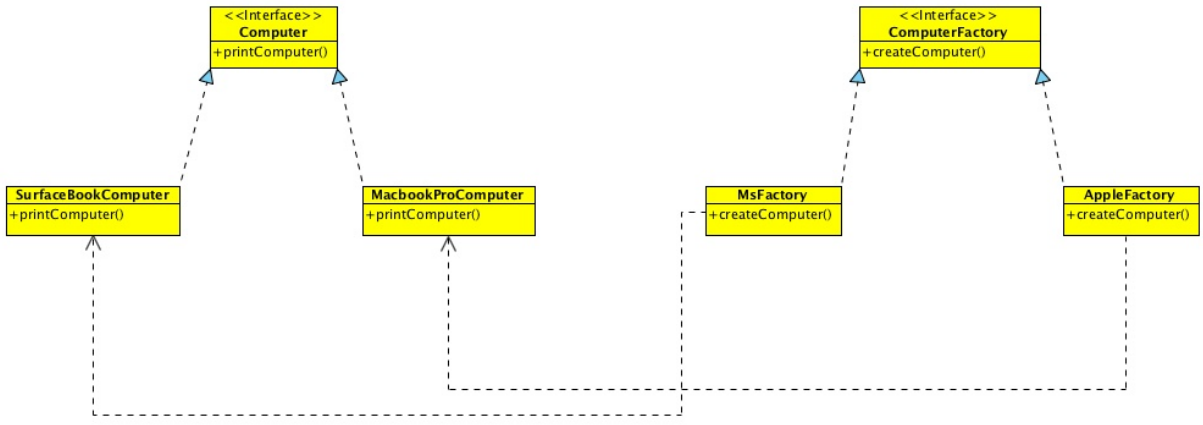
4.2 工厂方法模式

工厂方法模式：定义一个用来创建对象的接口，让子类决定实例化哪一个类，让子类决定实例化延迟到子类。

工厂方法模式是针对每个产品提供一个工厂类，在客户端中判断使用哪个工厂类去创建对象。

我们将之前的 ComputerFactory 抽象成一个接口，那么创建相应具体的工厂类去实现该接口的方法。

具体类图的实现：




```
interface Computer {
```

```
    void printComputer();
}

class MacbookProComputer implements Computer {

    public void printComputer() {
        System.out.println("This is a MacbookPro");
    }
}

class SurfaceBookComputer implements Computer {

    public void printComputer() {
        System.out.println("This is a SurfaceBook");
    }
}

interface ComputerFactory {
    Computer createComputer();
}

class MsFactory implements ComputerFactory {

    public Computer createComputer() {
        return new SurfaceBookComputer();
    }
}

class AppleFactory implements ComputerFactory {

    public Computer createComputer() {
        return new MacbookProComputer();
    }
}

public class Client {
    public void buyComputer(Computer computer) {
        computer.printComputer();
    }
    public static void main(String[] args) {
        Client client = new Client();
        ComputerFactory factory = new AppleFactory();
        client.buyComputer(factory.createComputer());
    }
}
```


工厂方法模式是针对每个产品提供一个工厂类，在客户端中判断使用哪个工厂类去创建对象。

简单工厂模式 VS 工厂方法模式：

- 对于简单工厂模式而言，创建对象的逻辑判断放在了工厂类中，客户不感知具体的类，但是其违背了开闭原则，如果要增加新的具体类，就必须修改工厂类。
- 对于工厂方法模式而言，是通过扩展来新增具体类的，符合开闭原则，但是在客户端就必须要感知到具体的工厂类，也就是将判断逻辑由简单工厂的工厂类挪到客户端。
- **工厂方法横向扩展很方便**，假如该工厂又有新的产品 Macbook Air 要生产，那么只需要创建相应的工厂类和产品类去实现抽象工厂接口和抽象产品接口即可，而不用去修改原有已经存在的代码。

> 概要
>
> 1. 一个抽象产品类
> 2. 多个具体产品类
> 3. 一个抽象工厂
> 4. 多个具体工厂 - 每一个具体产品对应一个具体工厂

优点：

- 降低了代码耦合度，对象的生成交给子类去完成
- 实现了开放封闭原则 - 每次添加子产品 不需要修改原有代码

缺点：

- 增加了代码量，每个具体产品都需要一个具体工厂
- 当增加抽象产品 也就是添加一个其他产品族 需要修改工厂 违背OCP

4.3 抽象工厂模式

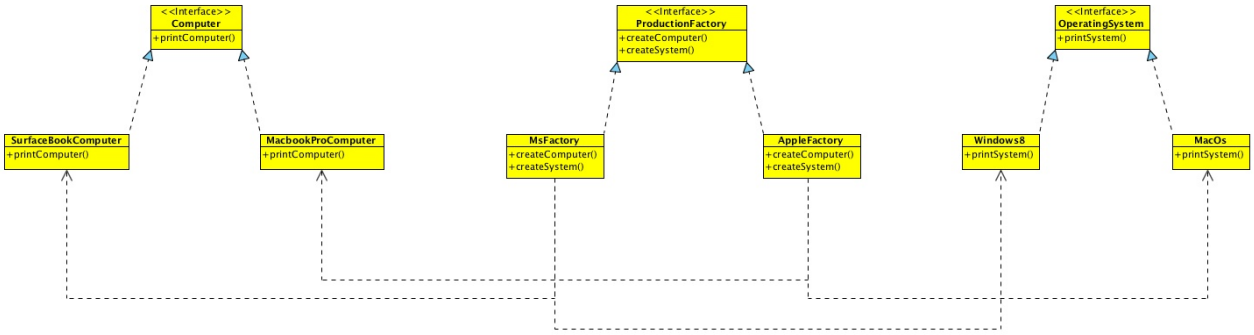
这时候负责该工厂的产品经理说要生产新的一类产品操作系统 Mac Os 和 Windows 8，这时候就引申出了抽象工厂模式。

抽象工厂模式：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。

工厂方法模式和抽象工厂模式基本类似，可以这么理解：当工厂只生产一个产品的时候，即为工厂方法模式，而工厂如果生产两个或以上的商品即变为抽象工厂模式。

我们在抽象工厂接口中新增创建系统的方法，并由实例工厂类去实现。

类图可由下表示：




```
interface Computer {
    void printComputer();
}

class MacbookProComputer implements Computer {
```

```
    public void printComputer() {
        System.out.println("This is a MacbookPro");
    }
}

class SurfaceBookComputer implements Computer {

    public void printComputer() {
        System.out.println("This is a SurfaceBook");
    }
}

interface OperatingSystem {
    void printSystem();
}

class MacOsSystem implements OperatingSystem {
    public void printSystem() {
        System.out.println("This is a mac os");
    }
}

class Windows8System implements OperatingSystem {
    public void printSystem() {
        System.out.println("This is a window 8");
    }
}

interface ProductionFactory {
    Computer createComputer();
    OperatingSystem createSystem();
}

class AppleFactory implements ProductionFactory {
    public Computer createComputer() {
        return new MacbookProComputer();
    }
    public OperatingSystem createSystem() {
        return new MacOsSystem();
    }
}

class MsFactory implements ProductionFactory {
    public Computer createComputer() {
        return new SurfaceBookComputer();
    }
    public OperatingSystem createSystem() {
        return new Windows8System();
```

```
    }
}


public class Client {
    public void buyComputer(Computer computer) {
        computer.printComputer();
    }
    public void use(OperatingSystem s) {
        s.printSystem();
    }
    public static void main(String[] args) {
        Client client = new Client();
        ProductionFactory factory = new AppleFactory();
        Computer computer = factory.createComputer();
        OperatingSystem system = factory.createSystem();
        client.buyComputer(computer);
        client.use(system);
    }
}
```

> 概要
>
> 1. 多个抽象产品类
> 2. 具体产品类
> 3. 抽象工厂类 - 声明(一组)返回抽象产品的方法
> 4. 具体工厂类 - 生成(一组)具体产品

优点:

- 代码解耦
- 实现多个产品族(相关联产品组成的家族)，而工厂方法模式的单个产品,可以满足更多的生产需求
- 很好的满足OCP开放封闭原则
- 抽象工厂模式中我们可以定义实现不止一个接口，一个工厂也可以生成不止一个产品类 对于复杂对象的生产相当灵活易扩展

缺点:

- 扩展产品族相当麻烦 而且扩展产品族会违反OCP,因为要修改所有的工厂
- 由于抽象工厂模式是工厂方法模式的扩展 总体的来说 很笨重

总结

> 简单工厂模式最大的优点就是工厂内有具体的逻辑去判断生成什么产品，将类的实例化交给了工厂，这样当我们需要什么产品只需要修改工厂的调用而不需要去修改客户端，对于客户端来说降低了与具体产品的依赖
>
> 工厂方法模式是简单工厂的扩展，工厂方法模式把原先简单工厂中的实现那个类的逻辑判断交给了客户端，如果像添加功能只需要修改客户和添加具体的功能，不用去修改之前的类。

> 抽象工厂模式进一步扩展了工厂方法模式，它把原先的工厂方法模式中只能有一个抽象产品不能添加产品族的缺点克服了，抽象工厂模式不仅仅遵循了OCP原则，而且可以添加更多产品(抽象产品),具体工厂也不仅仅可以生成单一产品，而是生成一组产品，抽象工厂也是声明一组产品，对应扩展更加灵活，但是要是扩展族系就会很笨重。

JDK中用到工厂模式的典型操作：

> 1. Collection中的iterator方法 （集合类中的迭代器）
> 2. java.util包中的sql相关操作

5. 代理设计模式（重点）

两个子类共同实现一个接口，其中一个子类负责真实业务实现，另外一个子类完成辅助真实业务主题的操作。

范例：实现代理模式

```
interface ISubject {
        public void buyComputer() ; // 核心功能是买电脑
}

class RealSubject implements ISubject {
        public void buyComputer() {
                System.out.println("买一台外星人电脑") ;
        }
}
class ProxySubject implements ISubject {
        private ISubject subject ; // 真正的操作业务
        public ProxySubject(ISubject subject) {
                this.subject = subject ;
        }
        public void produceComputer() {
                System.out.println("1.生产外星人电脑") ;
        }
        public void afterSale() {
                System.out.println("3.外星人电脑售后团队") ;
        }
        public void buyComputer() {
                this.produceComputer() ; // 真实操作前的准备
                this.subject.buyComputer() ; // 调用真实业务
                this.afterSale() ; // 操作后的收尾
        }
}
class Factory {
        public static ISubject getInstance(){
                return new ProxySubject(new RealSubject()) ;
        }
}

public class Test{
```

```
        public static void main(String[] args) {
                ISubject subject = Factory.getInstance() ;
                subject.buyComputer() ;
        }
}
```

代理模式的本质：所有的真实业务操作都会有一个与之辅助的工具类（功能类）共同完成。

代理模式在JavaEE中有着广泛的应用。EJB、WebService，Spring等技术都是代理模式的应用

6.抽象类与接口的区别（面试）

No	区别	抽象类(abstract)	接口(interface)
1	结构组成	普通类+抽象方法	抽象方法+全局常量
2	权限	各种权限	public
3	子类使用	使用extends关键字继承抽象类	使用implements关键字实现接口
4	关系	一个抽象类可以实现若干接口	接口不能继承抽象类，但是接口可以使用extends关键字继承多个父接口
5	子类限制	一个子类只能继承一个抽象类	一个子类可以实现多个接口

除了单继承的局限之外，实际上使用抽象类和接口都是类似的。在实际开发中，抽象类的设计比接口复杂。现在要求掌握的就是如何定义接口以及更好的实现子类。

> 1. 接口是Java的核心，慢慢会学到接口更多的使用与设计。
> 2. 开发之中优先考虑接口，以避免单继承局限。
> 3. 抽象类是模板，有层次感。
> 4. 接口则更关心行为与混合。