JAVA 面向对象（OOP 面向对象编程）（扩展 OOAD 面向对象的分析与设计）

1 学习方法

这部分内容大多数是以理解为主。比较抽象。理解是比较重要。比较困难。

分阶段。授课的难度分二个阶段。

第一个阶段（必须掌握）：基本语法。比如：要求创建一个类。你必掌握怎么创建一个类。

第二个阶段（目前不做强制要求）：理解面向对象思想。比如：要求创建一个类。为什么要创建这个类。

练习题少一些，想的多一些。

2 对比面向过程与面向对象

大家之前学习过 C 语言-面向过程的语言。

Java-面向对象的语言。

2.1 面向过程-C

面向过程语言-最小单位“过程（函数，方法）”。过程代表就是一个功能。

一个函数一旦定义完成，功能不能改变。

当我们想要一个新的功能时，要新去创建一个函数。

在我们自己写 C 时调用 A 这个函数时，A 是不是必须提前存在。是。

2.2 面向对象-Java

面向对象语言-最小单位“对象”。对象在过程实现中使用的内容。

一个函数一旦定义完成，功能可能改变。

当我们想要一个新的功能时，不用创建一个新的函数的。只要修改对象。

在我们自己写 Java 时调用 A 这个函数时，A 是不是必须提前存在。不是。

2.3 例：盖房子。

盖房子是一个功能-函数。

在面向过程：：：盖房子这个函数一旦定义。功能就死了。

```
盖四方房子（）{
    四方地基（）
    四方墙（）
    四方顶（）
}
```

在面向对象::: 盖房子这个函数。在这个过程中使用到哪些对象。

地基-地基对象。墙-墙对象。顶-顶对象。

盖房子（地基对象，墙对象，顶对象）{

地基对象.挖地基（）；

墙对象.砌墙（）；

顶对象.铺顶（）；

}

盖房子的方法的功能不再单一的一种。变的可以有很多的可能性。

3 对象的概念

区分：对象的概念是在两个地方 。

1 现实世界的对象。

现实世界中一切事务都是对象。（现实世界中的姚明）

2 java 程序中的对象。

Java 中的一个变量。是 Java 程序中类这个类型的变量。内存空间。（NBA2015 这个游戏中的姚明）

int a = 10; a 是一个变量。

String s = "abc";s 是一变量。s 是一个 String 数据类型

的对象。

对象中有属性和方法。

属性是用来描述对象的特征。比如，姚明的身高，体重，姓名，。。。。

方法是用来描述对象的行为。比如，姚明可以吃，打蓝球，投蓝，盖帽。。。。

程序中如果要获得一个对象？必须得先有类型。

想要一个对象必须先创建这个对象（变量）的类（数据类型）。

4 类的概念

类就是一个 Java 中的自定义的数据类型。

类这个数据类型里面有属性和方法。

类是对象的模版。（类是对象的数据类型。）

5 定义类（数据类型）

定义类的关键字：class

格式：修饰符 class 类名{ 类中的属性和方法 }

现在使用：public class Stu{ 学员的属性和方法 }

```
//学员类
public class Stu {
}
```

6 使用类（自定义的数据类型）创建对象（变量）

关键字：new ： new 是实例化关键字。

创建对象的格式：new 类名（）；

类名 引用名 = new 类名( );

```
Stu stu = new Stu();//创建一个学员类型的对象。
```

new Stu();表示要实例化一个 Stu 类的对象。

stu 实际上是一个引用的名字。

Stu 是一个我们自己创建的数据类型（类）。是引用数据类型

7 访问符“.”

使用点（“.”）访问对象中的属性和方法。

格式：引用名.方法();

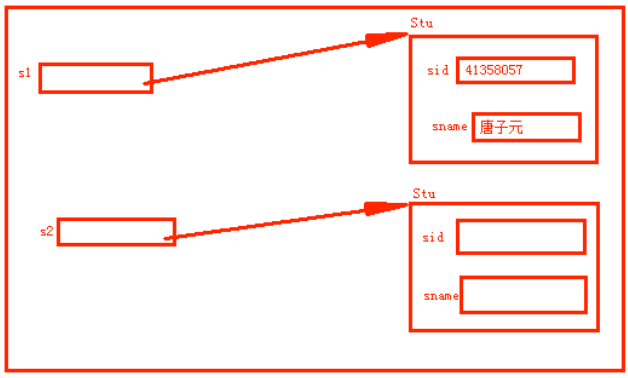
8 属性，又称为实例变量，实例属性

实例属性是每个实例都有一个属性。每个实例之间属性是没关系。

```java
@Test
public void test01() {
    // 创建一个对象
    Stu s1 = new Stu();
    s1.sid = 41358057;
    s1.sname = "唐子元";
    System.out.println(s1.sname);

    Stu s2 = new Stu();

    System.out.println(s2.sname);
}
```



实例属性是有默认值的。原始数据是各种的0，引用数据是null。

属性在本类的方法中可以直接使用。而且属性定义的位置没有要求，但为了代码的可读性一般放在上面。

9 两个访问修饰符

public:公有，表示在任意位置都可以使用。

private:私有，只能在定义的类中使用。

10 JavaBean规范

1 类一定要有默认构造方法。

2 属性要 private。

3 通过 public 的方法来操作属性。

方法的命名是有规范的。

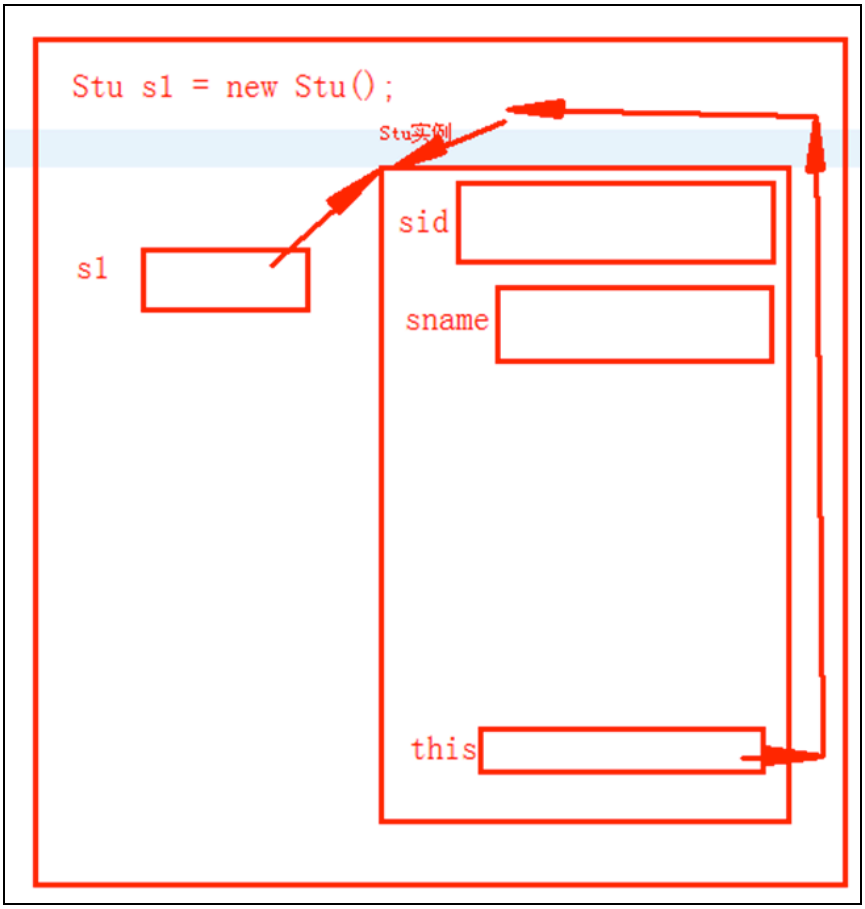
为属性赋值的方法前缀用 setXxx()：属性的名称首字母大写

获取属性的值方法前缀用 getXxx()：属性的名称首字母大写

```
public class Stu {
    private int sid;// 学号
    public int getSid() {
        return sid;
    }
    public void setSid(int sid) {
        this.sid = sid;
    }
}
```

11 this 关键字

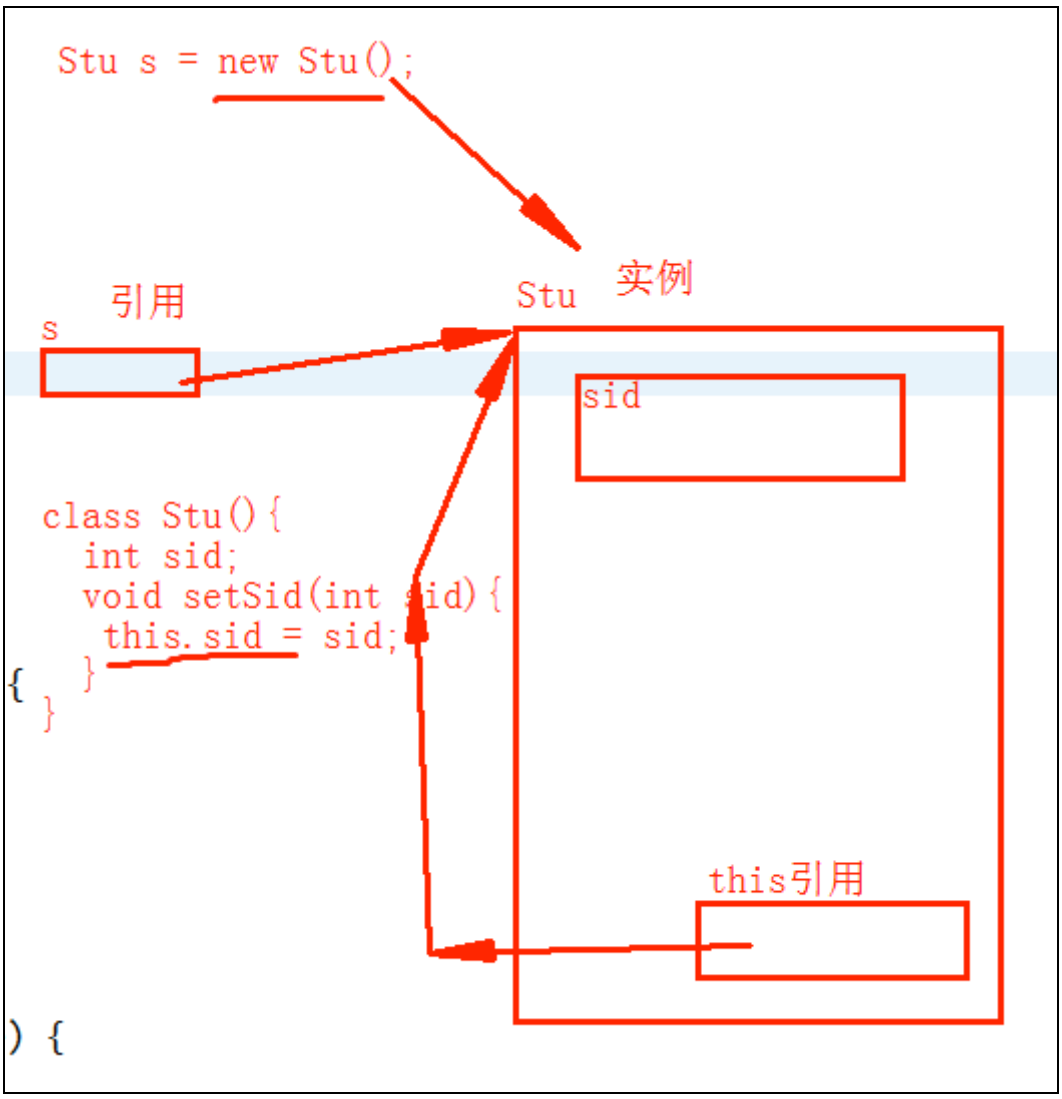
在实例内部代表实例本身的一个引用。



一般在参数名称与属性名称一致时，用来区分哪个是属性用的。在属性前面加上"this."

```
public class Stu {
    private int sid;// 学号
    public int getSid() {
        return sid;
    }
    public void setSid(int sid) {
        this.sid = sid;
    }
```

```
}
```



12 标准的代码规范。类中有属性和方法

```
public class Stu {

    private int sid;// 学号

    private String name;// 姓名
```

```
    private String phone;// 电话
    public int getSid() {
        return sid;
    }
    public void setSid(int sid) {
        this.sid = sid;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

```
    public void test01(){
        Stu stu1 = new Stu();//创建一个学员类型的对象。
```

```
        stu1.setSid(41355011);
        stu1.setName("李壮");
    }
```

13 构造方法

构造方法是一个特殊的方法。

1 方法名称必须与类名一致。完全一致。大小写都要一致。

2 方法没有返回类型说明。注意不是没有返回值，是没有返回类型的说明。

构造方法是在 new 时（实例化），由系统自动调用的方法。（在实例化一个类的对象的过程中一定要调用构造方法）

构造方法的作用是用来构造实例并初始化属性。

构造方法的作用：实例化属性。

一个类中必须具有构造方法。

一个类的实例化必须调用构造方法。

13.1 构造方法的分类。

13.1.1 隐式构造方法：当一个类没有明确声明构造方法时，系统会自动增加一个构造方法。

会加一个无参，没有方法体的构造方法。

如果显示的去声明应该长成这个样子。

```
public class Birthday{
public Birthday(){}----系统加时的样子
}
```

13.1.2 默认构造方法：不是隐式构造方法。无参，我们明确声明的。

13.1.3 参数化构造方法：有参，我们明确声明的。

当我们的类中有构造方法，会同时限制实例化时（new ）的格式。

14 方法重载

同类同名不同参：

在一个类中有多个同名的方法，但是方法的参数不同时，把这几个同名的方法叫方法重载。

参数不同包含：

1 参数个数。

2 参数类型。

构造方法也是方法，所以构造方法也可以重载。

15 对象，引用，实例

```
Stu s = new Stu();
```

引用是：s

实例是：内存空间，通过 new 关键字实例化，实例化的过程实际上就是开辟内存空间的过程。

对象是：s 就是一个现实世界的对象的一个名字。s 现在是学员类型的对象。

16 类和对象

人是类。 姚明是对象。

学员是类。 呼月圆是对象。

17 this 关键字补全

this 是一个在实例里引用实例本身的一个引用。

this 关键字在实例内部可以调用三个内容：

1 属性：调用实例本身自己的属性。

this.属性名

2 方法：调用实例本身自己的方法。

this.方法名（）；

3 构造方法：调用实例本身自己的构造方法。

this();

只能写在构造方法中。

只能写在构造方法的第一行。

```
public class Stu {

    private int sid;// 学号
    private String sname;// 姓名
    private Birthday birthday;

    public Stu() {
        this.birthday = new Birthday();
    }
    public Stu(int sid) {
        this();
        this.sid = sid;
    }
    public Stu(String sname) {
        this();
        this.sname = sname;
    }
    public Stu(int sid, String sname) {
        this();
        this.sid = sid;
        this.sname = sname;
    }
    public Stu(int sid, String sname, Birthday birthday) {
        this.sid = sid;
        this.sname = sname;
        this.birthday = birthday;
    }
```

18 值传递和引用传递

class 创建的数据类型是引用数据类型。

如果我们的参数是对象的话，引用传递，在做引用传递时形参和实参实际上是同一个引用的空间。

练习：创建一个笔记本的类。笔记本有颜色，价格，CPU 三个属性。CPU 有型号，主频二个属性。使用这个类创建一个你的笔记本的对象。

面向对象的三大特性-封装，继承，多态

1 封装

面向过程语言中也有封装

1.1 什么是封装？封装表现在哪里？

封装一个表现方式：{ }

比如 if,while,for

For(int I =0 ; i<10;i++){

封装在循珠中的语句。

}

方法是不是封装之后的结果？是。

对大量相同的代码（功能）进行抽象，再进行封装就变成了一个方法。

类也是封装之后的结果。

比如：有一个学员类。

我们在大量学员对象的基础上抽象出了相同的属性和行为。我们再把这些抽象出来的属性和行为进行封装操作变成一个class。

抽象：在面向对象语言，抽象很重要的事情。

1.2 封装目的

封装的目的是为了实现模块化开发。

具体体现方式：封装的目的是为了获得类和方法。

1.3 封装的好处

隐藏细节。自己做饭对比买饭。

1.4 数据封装的实现方式

我们使用 javaBean 规范实现数据封装。

1 公有默认构造方法

2 私有属性

3 公有方法 set/get

1.5 访问修饰符

可以修饰类、属性、方法

在 Java 中一共有四个访问级别。但只有三个访问修饰符。

修饰符	访问范围	是否能被子类继承
private	本类内部	不能被继承
(default)	本类内部+同包的其他类	能被同包的子类继承
protected	本类内部+同包的其他类+非同包的子类	能被继承
public	公开，能被所有类访问	能被继承

private ： 私有，只能类内使用。子类不能访问。

(default)：默认，包级别，本类和同包可用。

protected：保护，同包和子类。

public ： 公有，任何位置都可以访问

2 继承（extends）

2.1 在 Java 程序中怎么实现继承？使用 extends 关键字。

格式： public class 子类 extends 父类 { }

```
public class A {
```

```
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```
public class B extends A{
}
```

B 类继承 A 类。

我们叫 B 类为子类。A 类叫父类

```
public class A {
    public A() {

        System.err.println("A 类的构造方法");

    }
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

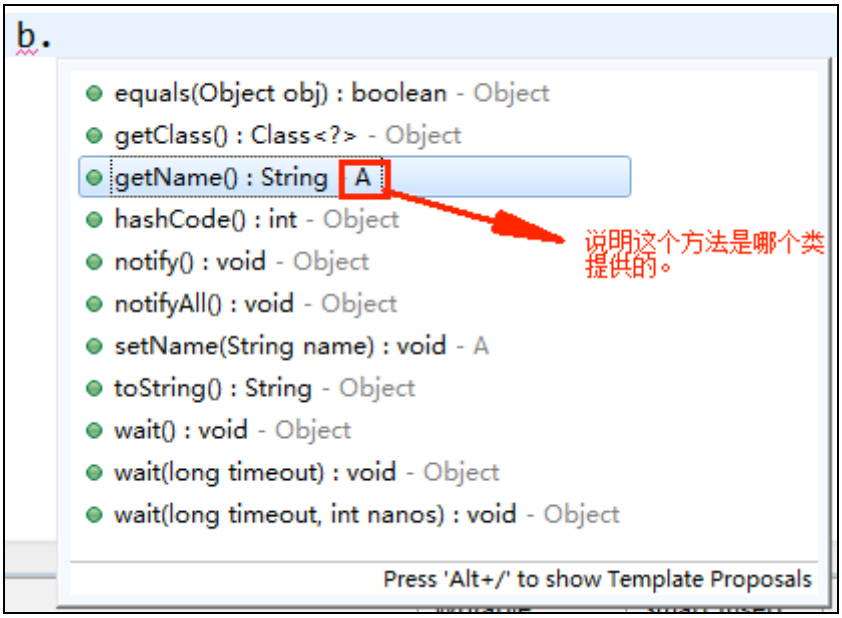
```
public class B extends A{
    public B(){

        System.out.println("B 类的构造方法");

    }
}
```

2.2 继承的特点是什么？

子类可以复用父类中的属性和方法。(必须参考访问修饰符，如果属性或方法是以 private 修饰时，在子类中不能使用父类的方法了。)

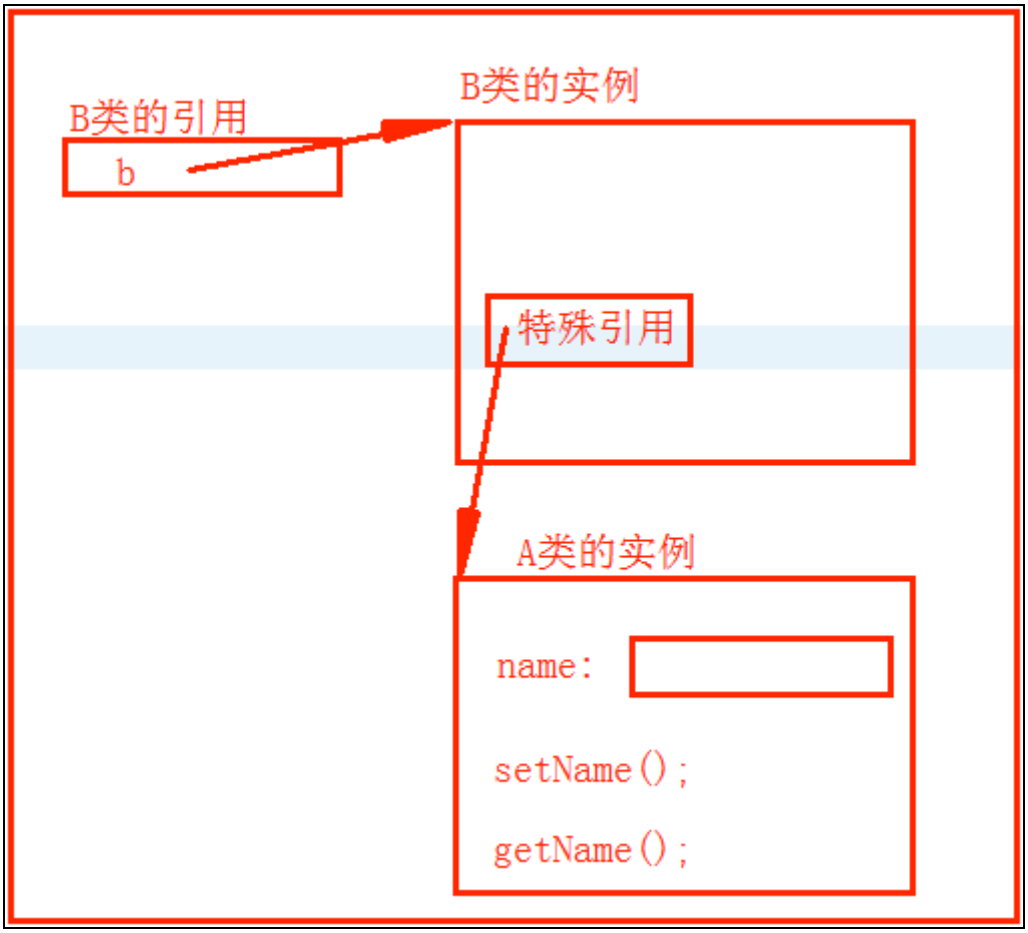


2.3 继承在 Java 内存中是一个什么样的表现形式？

继承不是单纯的代码复制。

会为这个子类在创建子类实例时同时也创建一个属于这个子类实例的父类实例。

Java 会为每个子类实例都关联一个父类实例。



2.5 在继承关系时，创建子类实例时，要不要有一个父类的实例？

必须有一个属于子类实例的父类实例。

在创建子类实例时实际上创建了 2 个实例，子类实例和父类实例

2.6 子类实例和父类实例，在实例化有没有顺序的问题？

先创建好父类的实例，再创建好子类的实例。（先父后子）

2.7 在继承关系时，构造方法的执行过程。

B b = new B();

因为 new B();，所以一定是先去调用 B()这个构造方法。但是在子类构造方法的第一行一定是调用父类的构造方法。

2.8 在开发时先有父类再有子类。

2.9 在开发时，子类是通过继承得到的。父类是怎么得到？

类是对大量相同类型的对象的同性的抽象。

父类是对大量相同类型的子类的同性的抽象。

```
public class Student {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

```
public class Teacher {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
```

```
    }
}
```

当我们发现两个类中有大量相同的属性和方法时。就可以考虑使用继承关系做到代码复用。

可以创建一个父类，将两个类中的共性放到这个父类中。

```
public class Person {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

之后再分别继承这个父类。

```
public class Student extends Person{
}
```

```
public class Teacher extends Person{
}
```

2.10 是任意的两个类之间都可产生继承关系？

标准：is-a。是一个。两个类必须满足是一个关系才能创建继承。

我们在使用继承时必须满足 is-a 关系。也就是说子类必须是一个父类。

子类是一个特殊的父类

扩展：类与类之间的关系：

1 is-a： 是一个。使用继承

2 has-a：有一个。做成属性

3 use-a：用一个。做成参数

BMW 类，汽车类，发动机类，司机类。四个类。描述这四个类之间的关系。

BMW 是汽车。

```
public class 汽车{ }
public class BMW extends 汽车 { }
```

汽车有发动机。

```
public class 发动机 { }
public class 汽车{
    private 发动机  发动机对象;
}
```

司机使用汽车-开。

```
public class 司机{
    public void 开车(汽车  汽车对象){ }
}
```

2.11 子类我们可以认为具有父类的属性和方法。子类也可以具有自己的属性和方法。

父类更通过，子类更具体。

子类是一个特殊的父类

父类中都是所有子类共性的属性和方法。

子类中具体自己特性的属性和方法。

```java
public class Person {
    private String name;//姓名，学员和教员的共性属性
    private int age;//年龄，学员和教员的共性属性
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

```java
public class Teacher extends Person{
    //教员有工资
    private double sal;//教员类的特性的属性
    public double getSal() {
        return sal;
    }
    public void setSal(double sal) {
        this.sal = sal;
    }
}
```

```java
public class Student extends Person{
    //学员有学号
    private int sid;
    public int getSid() {
        return sid;
    }
    public void setSid(int sid) {
        this.sid = sid;
    }
}
```

2.12 当继承时发现继承的方法已经在子类中不适用时要怎么办？

改。把方法更新。

注意，不能随便在父类中做更新。因为父类的更新会影响到所有的子类。

子类本身发现父类继承的方法对于子类自己不适用时，只能在子类本身修改这个继承的方法。

在 Java 中可以通过“方法重写”实现在子类中修改继承父类的方法。

2.13 方法重写(Override)

两个类要有继承关系。

在子类重新编写相同的方法。方法名称，返回类型，参数都必须相同。

我们就叫方法重写。

```java
public class Person {
    private String name;// 姓名，学员和教员的共性属性
    private int age;// 年龄，学员和教员的共性属性

    public String getName() {

    public void setName(String name) {

    public int getAge() {

    public void setAge(int age) {

    public void dis() {
        System.out.println("姓名：" + this.name);
        System.out.println("年龄：" + this.age);
    }
}
```

```java
public class Student extends Person{
    //学员有学号
    private int sid;
    public int getSid() {
    public void setSid(int sid) {

    public void dis(){
        System.out.println("学号："+this.sid);
        System.out.println("姓名："+this.getName());
        System.out.println("年龄："+this.getAge());
    }  这就是我们在子类中重写的方法。方法的声明与父类必须一致
}
```

2.14 super 关键字

在实例内部引用其父类实例的一个引用。

使用 super 关键字可以调用父类的属性和方法。

关于 super 常见的用法

1 super.父类方法（）； 调用父类的方法。一般在子类重写

的方法中调用父类的方法。

2 super（）；　　调用父类的构造方法。只能在子类的构造方法的第一行出现。用来明确指定调用父类的哪个构造方法。

2.15 继承的目的？

在没有多态时，使用继承的目的是为了代码复用。

在没有多态时，使用继承的目的是为了修改现有类，创建新类。

2.16 练习：某公司的雇员分为以下若干类

某公司的雇员分为以下若干类：

Employee：这是所有员工总的父类，属性：员工的姓名（name），员工的生日月份（month）。方法：getSalary(int month) 根据参数月份来确定工资，如果该月员工过生日，则公司会额外奖励 100 元。

SalariedEmployee：Employee 的子类，拿固定工资的员工。属性：月薪（monthlySalary）

HourlyEmployee：Employee 的子类，按小时拿工资的员工，每月工作超出 160 小时的部分按照 1.5 倍工资发放。属性：每小时的工资（hourlySalary）、每月工作的小时数（hours）

SalesEmployee：Employee 的子类，销售人员，工资由月销售额和提成率决定。属性：月销售额（sales）、提成率（commission）

BasePlusSalesEmployee：SalesEmployee 的子类，有固定底薪的销售人员，工资由底薪加上销售提成部分。属性：底薪（baseSalary）。

- Employee 类：

```
package com.domain;
public class Employee {
    private String name;// 姓名
    private int month;// 月份
```

```
    public Employee() {
    }
    public Employee(String name, int month) {
        this.name = name;
        this.month = month;
    }
    /**
     * 计算工资的方法
     * @param month 发放工资的月份
     * @return 只是奖励金额
     */
    public double getSalary(int month) {
        if (this.month == month) {
            return 100;
        } else {
            return 0;
        }
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getMonth() {
        return month;
    }
    public void setMonth(int month) {
        this.month = month;
    }
}
```

- SalariedEmployee

```
package com.domain;
public class SalariedEmployee extends Employee {
    private double monthlySalary;// 月薪
    public SalariedEmployee() {
    }
    public double getMonthlySalary() {
        return monthlySalary;
    }
    public void setMonthlySalary(double monthlySalary) {
        this.monthlySalary = monthlySalary;
    }
```

```
    @Override
    public double getSalary(int month) {
        // 获得奖励金额，必须调用父类被重写的方法。使用super关键字在子类中调用父类的方法或属性
        double salary = super.getSalary(month);
        return this.monthlySalary + salary;
    }
}
```

- HourlyEmployee

```
package com.domain;
public class HourlyEmployee extends Employee {
    private int hours;// 当月的工时
    private double hourlySalary;// 每小时的工资
    public HourlyEmployee() {
    }
    @Override
    public double getSalary(int month) {
        double salary = super.getSalary(month);
        if (this.hours > 160) {
            return salary + (160 * hourlySalary)
                    + ((hours - 160) * (hourlySalary * 1.5));
        } else {
            return salary + hours * hourlySalary;
        }
    }
    public int getHours() {
        return hours;
    }
    public void setHours(int hours) {
        this.hours = hours;
    }
    public double getHourlySalary() {
        return hourlySalary;
    }
    public void setHourlySalary(double hourlySalary) {
        this.hourlySalary = hourlySalary;
    }
}
```

- SalesEmployee

```
package com.domain;
public class SalesEmployee extends Employee {
    private double sales;
```

```
    private double commission;
    public SalesEmployee() {
        // TODO Auto-generated constructor stub
    }
    public SalesEmployee(String name, int month, double sales, double
commission) {
        super(name, month);
        // this.setName(name);
        // this.setMonth(month);
        this.sales = sales;
        this.commission = commission;
    }
    public double getSales() {
        return sales;
    }
    public void setSales(double sales) {
        this.sales = sales;
    }
    public double getCommission() {
        return commission;
    }
    public void setCommission(double commission) {
        this.commission = commission;
    }
    @Override
    public double getSalary(int month) {
        return super.getSalary(month) + (this.sales * this.commission);
    }
}
```

BasePlusSalesEmployee

```
package com.domain;
public class BasePlusSalesEmployee extends SalesEmployee {
    private double baseSalary;
    public BasePlusSalesEmployee() {
        // TODO Auto-generated constructor stub
    }
    public BasePlusSalesEmployee(String name, int month, double sales,
            double commission, double baseSalary) {
        super(name, month, sales, commission);
        this.baseSalary = baseSalary;
    }
    public double getBaseSalary() {
        return baseSalary;
```

```
    }
    public void setBaseSalary(double baseSalary) {
        this.baseSalary = baseSalary;
    }
    @Override
    public double getSalary(int month) {
        return super.getSalary(month) + this.baseSalary;
    }
}
```

3 多态（是面向对象语言的最核心的特性）

封装和继承都是为了实现多态。

多态是 Java 语言的一种特性：使用类的一种固定方式。

3.1 多态的基础：

1 要有继承：extends

2 要有方法重写：在子类中重写父类继承的方法。声明一个与父类一样的方法。但是方法的内容可以重新写。

3.2 多态的格式：

父类引用指向子类实例

```
父类类型 父类引用  = new 子类类型();
```

3.3 父类引用指向了一个子类的实例时，父类引用可以调用二类方法。

1 父类中声明的自己的方法。

2 子类中重写父类的方法。

3.4 多态的一般使用方法的格式：

1 将父类引用做为方法的形参。

2 将子类实例做为方法的实参。

```
public void print(A a){
    a.dis();
}
@Test
public void testprint(){
    ExtendsTest extendsTest = new ExtendsTest();
    extendsTest.print(new A());
    extendsTest.print(new B());
    extendsTest.print(new C());
}
```

3.5 多态的应用：以面向对象设计的方式去理解

封装的目的是为了得到类和方法，父类。面向对象设计的要求：程序中变化和不变化的内容分开封装。

继承是为修改类。在继承时在子类中可以对父类的方法进行修改。

开放关闭原则：对扩展开放，对修改关闭。

多态就是为了在不修改源文件的情况下可以使用新的子类。对扩展开放。

3.6 多态的案例：薪资专员的工资统计

创建一个薪资专员（Persionnel），负责汇总当前月份所有员工的总工资数。

Persionnel.java:薪资专员类

```
package com.sunxu.org.personnel;

import com.sunxu.org.Employee;

/**
 * 人事专员，负责统计每个月的工资总额。
 *
 * @author Administrator
 *
 */
public class Persionnel {
    private double sumSalary;// 总工资

    private int month;// 月份

    public Persionnel(int month) {
        this.month = month;
    }
    //扩展内容，不理解也没关系。
    public void addSalary(Employee[] employees) {
        for (int i = 0 ;i <employees.length;i++) {
            if (employees[i] != null) {//数组元素为空的对象是不能调用方法的。所以我们要加这个判断。
                this.sumSalary += employees[i].getSalary(month);
            }
        }
    }
    /**
     * 多态的累加员工工资的方法
     * @param employee 父类类型，所以可以通用于父类和其所有的子类，这就是多态应用中最广泛的一种。
     */
    public void addSalary(Employee employee){
        this.sumSalary += employee.getSalary(month);
        //employee.getSalary(month);是多态应用最核心的语句。
        //根据传递给employee对象的实例的不同调用的getSalary()方法也不同。
        //由些我们实现的多态的效果。
        //一条employee.getSalary(month);语句，根据实例的不同执行不同的代码。
    }
    public void disSumSalary() {
        System.out.println("当前" + this.month + "月的总工资是:$" + this.sumSalary
                + "元");
    }
}
```

Test.java 测试类：

```
package com.sunxu.org.test;

import com.sunxu.org.BasePlusSalesEmployee;
import com.sunxu.org.HourlyEmployee;
import com.sunxu.org.SalariedEmployee;
import com.sunxu.org.SalesEmployee;
import com.sunxu.org.personnel.Persionnel;

public class Test {
    public static void main(String[] args) {
        Persionnel persionnel = new Persionnel(1);

//        Employee[] employees = new Employee[100];
//        employees[0] = new SalariedEmployee("闫创", 1, 10);
//        employees[1] = new SalariedEmployee("付德金", 2, 20);
//        employees[2] = new HourlyEmployee("李旻玲", 1, 100, 1);
//        //employees[3] = null;数组默认值为null，所以从索引3以后的数组元素的值都是null的。
//        persionnel.addSalary(employees);

        persionnel.addSalary(new HourlyEmployee("李旻玲",1,100,1));//方法调用，多态应用
        persionnel.addSalary(new SalariedEmployee("付德金", 2, 10));//方法调用，多态应用
        SalesEmployee salesEmployee = new SalesEmployee();
        salesEmployee.setName("杨光");
        salesEmployee.setMonth(1);
        salesEmployee.setSales(1000);
        salesEmployee.setCommission(0.1);
        persionnel.addSalary(salesEmployee);//方法调用，多态应用
        BasePlusSalesEmployee basePlusSalesEmployee = new BasePlusSalesEmployee();
        basePlusSalesEmployee.setName("闫创");
        basePlusSalesEmployee.setMonth(1);
        basePlusSalesEmployee.setBaseSalary(100);
        basePlusSalesEmployee.setSales(10000);
        basePlusSalesEmployee.setCommission(0.001);
        persionnel.addSalary(basePlusSalesEmployee);//我们可以在这一个方法中传入不同的
参数，实现相同的功能。

        persionnel.disSumSalary();
    }
```

[illegible] Shape，[illegible] 打印信息的方式 [illegible] 求面积

[illegible] Rect(矩形)、Circle(圆形)，

[illegible] Square(正方形) [illegible] 注册会计老师

3.7 多态应用练习

[illegible] 时间

1 人员信息练习

[illegible] 注册会计老

[illegible] 计师证， 注册会计财时间

写一个学校类。学校类负责打印每位教员的信息。

2 图形练习

4 final 修饰符

不可改变。

4.1 可以修饰三个对象

1 变量：局部变量，常量，值不可改变。

类中的属性：

1 属性可以初始化，但不能通过 set 方法赋值。

2 属性的初始化一定要使用类的构造方法，所以默认构造方法不可用。

基本数据类型：值不可改变。

引用数据类型：引用不可改变。

2 方法：方法不可改变。方法不能被重写。

3 类：类不可改变。类不能被继承。

5 static 修饰符

叫静态修饰符。

1 代码块：静态代码块，静态块。

代码块：{ }

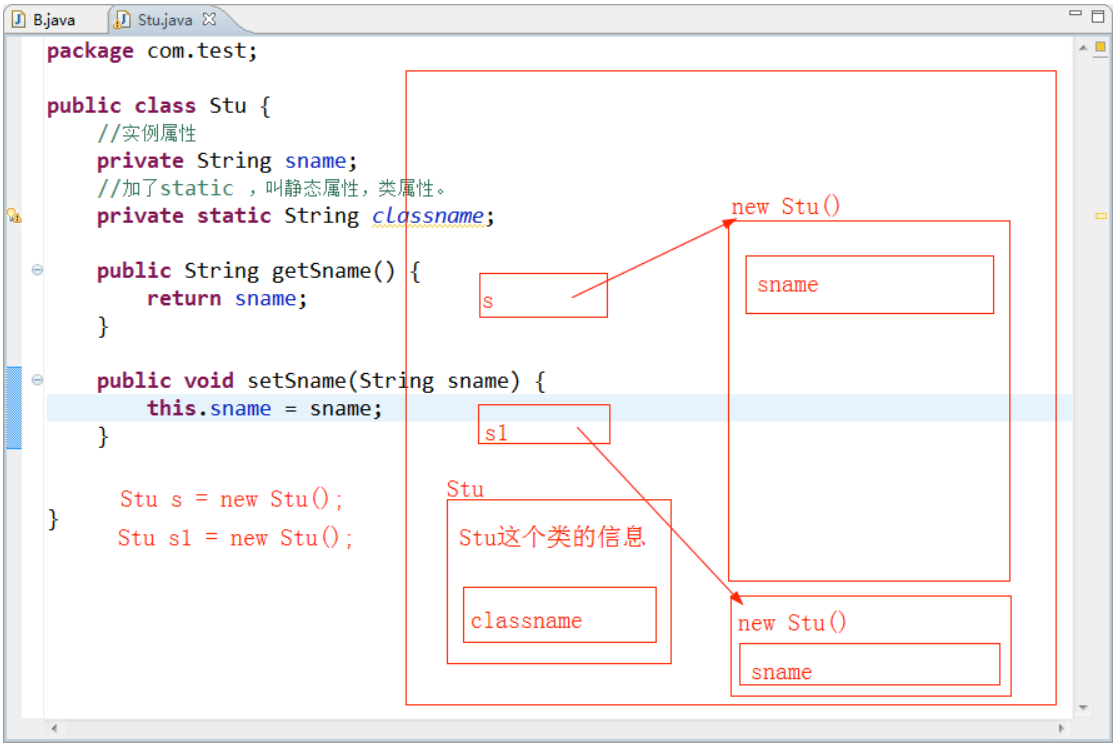
静态代码块：static{ }

给静态属性初始化用的。

2 属性：静态属性

没加的 static 的属性叫实例属性。是实例的。

加了 static 叫静态属性，也叫类属性。是类的。



是在类中存在，而且只有一份，类和类的所有实例可以共用。

3 方法：静态方法

静态方法只可以访问静态属性和静态方法。

？？为什么不能访问实例属性？？因为调用执行静态方法时是可以不用先创建实例。

针对静态属性和方法多了一种新的访问方式：类名.静态方法 类名.静态属性

？？？静态内容的访问，有不有先创建实例？？？不用。

在静态方法和静态代码块中不能使用 this 和 super

6 abstract 修饰符

Abstract 中文叫抽象。

可以修饰二个地方：方法和类。

设计一个形状类 Shape,方法:求周长和求面积

形状类的子类:Rect(矩形),Circle(圆形),

Rect 类的子类:Square(正方形),

不同的子类会有不同的计算周长和面积的方法,

创建三个不同的形状对象，分别打印出每个对象的周长和面积,

创建一个操作的类，可以计算每一种图形的周长和面积。

6.1 抽象方法:

使用 abstract 修饰符修饰的方法。

抽象方法有自己的固定格式：没有方法体。

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

6.2 抽象类:

类前面加 abstract 修饰符的类,叫抽象类。

一个类中如果有抽象方法，这个类必须是抽象类。

在一个抽象类中可以有属性，方法，抽象方法，但也可不存在抽象方法。

抽象类不能被实例化。

抽象类在使用时一定要做为父类，让子类继承，重写抽象方法。

6.3 图形练习

Shape.java：形状类

```
package com.shape;
```

```
public abstract class Shape {
    public abstract double jszc();

    public abstract double jsmj();
}
```

Rect.java：矩形类

```
package com.shape;

public class Rect extends Shape {
    private double chang;
    private double kuan;

    public Rect(double chang, double kuan) {
        super();
        this.chang = chang;
        this.kuan = kuan;
    }

    public double jszc() {
        return 2 * (chang + kuan);
    }

    public double jsmj() {
        return chang * kuan;
    }

}
```

Test.java：测试类，多态应用的方法在这里。

```
package com.test;

import com.shape.Circle;
import com.shape.Rect;
import com.shape.Shape;

public class Test {
    //应用多态的方法
    public void js(Shape shape) {
        System.out.println("周长：" + shape.jszc());
        System.out.println("面积：" + shape.jsmj());
    }
```

```
    public static void main(String[] args) {
        Test test = new Test();
        Circle circle = new Circle(50);
        Rect rect = new Rect(10, 20);
        test.js(circle);
        test.js(rect);

    }
}
```

6.4 练习：

1. 写一个“愤怒的小鸟”的作业：

    1. 我们有很多种小鸟，每种小鸟都有飞和叫的行为。
    2. 还有一个弹弓，弹弓有一个弹射的方法，弹射的方法将把小鸟弹出去。
    3. 之后小鸟使用自己飞行的方法飞向小猪。（这是我们不讨论猪（不是鄙视它），只讨论小鸟）
    4. 各种小鸟不同飞的方式。
        1. 红火：红色小鸟，飞行方式：正常。
        2. 蓝冰：蓝色小鸟，飞行方式：分成3个。
        3. 黄风：黄色小鸟，飞行方式：加速。

7 接口（interface）

7.1 如何定义接口？

使用 interface 关键字定义一个接口。

与创建类的格式基本一致。

类和接口是同一级别的内容。

public interface 接口名{

}

7.2 接口中有什么？

1 公有静态常量，public static final

2 公有抽象方法，抽象方法的修饰符 abstract。但是在接口中只能有抽象方法，在接口可以省略 abstract

3 注意，以上只适用于 jdk1.7，到 JDK1.8 之后在接口中可以加入其他内容。

7.3 接口的使用？

1 不能被 new，实例化。

2 一定要找一个类，去实现接口。实现就是一特别关键字 implements

public class 实现类名 implements 接口名{ }

3 在实现类中一定要对接口中声明的抽象方法给出具体的实现。

重写接口中的抽象方法。

4 当实现类中不能将接口中所有的抽象方法全部给出实现时，这个实现类就必须是抽象类。

5 接口同样支持面向对象的多态这个特性。

接口类型的引用可以指向一个其实现类的实例。

接口类型 接口类型引用 = new 实现类（）；

6 一个类只能继承自一个父类。Extends 类 只能有一个。Java是一种单继承的语言。

7 一个类可以实现多个接口。Implements 接口1，接口2

8 一个类可以同时继承一个父类，再同时实现多个接口。

8 接口的应用

8.1 理解接口？

父类是对大量子类共性的抽象。Is-a。

我们认为接口是大量实现类的共性行为的抽象。

接口是行为的抽象，一个接口就表示的一些行为，或能力。

例如：照相行为。有照相行为的对象：照相机，手机，笔记本，大师兄。

把照相行为封装成一个接口，叫照相接口。

写一个美女类，美女都喜欢自拍。我们不会写四个方法。

Public class 美女类{

Public 照片 自拍（照相接口 照相设备）{照相设备.拍照功能() }

}

8.2 接口的实际应用

2. 写一个“愤怒的小鸟”的作业：

1. 我们有很多种小鸟，每种小鸟都有飞（fly）和叫的行为。
2. 还有一个弹弓(slingshot)，弹弓有一个弹射的方法，弹射的方法将把小鸟弹出去。
3. 之后小鸟使用自己飞行的方法飞向小猪。（这是我们不讨论猪（不是鄙视它），只讨论小鸟）
4. 各种小鸟不同飞的方式。
    1. 红火：红色小鸟，飞行方式：正常。
    2. 蓝冰：蓝色小鸟，飞行方式：分成3个。
    3. 黄风：黄色小鸟，飞行方式：加速。

对于经常改变和一定不变的要分别封装。

飞行的行为经常在变化。对飞行的行为进行独立封装。

有一个比是一个好用。

鸟有飞行的行为。

现在我们所每一种鸟飞行的行为都封装到了每一种鸟类中。

现在我们来使用接口，独立封装各种飞行的行为。

1 创建飞行接口

2 每一种飞行的行为都是飞行接口的一个实现类。

更好的处理方式是：鸟有一个飞行行为。

当我们发现每一种鸟都有飞行的行为，同时飞行的行为有很多种的实现方式。

1 创建飞行接口

```
public interface Fly {
   public void fly();
}
```

2 每一种飞行的行为都是飞行接口的一个实现类。

```
public class ZhengFly implements Fly {
   @Override
   public void fly() {
      System.out.println("正常飞");
   }
}
```

3 在鸟这个父类中 有一个 飞行行为。

```
public class Birds {
   private String color;// 颜色
   // 鸟还有飞行行为
   private Fly fly;
   public Fly getFly() {
```

```
        return fly;
    }
    public void setFly(Fly fly) {
        this.fly = fly;
    }
```

4 在鸟这个父类中编写飞行方法，飞行方法实现，调用飞行行为的飞行方法。

```
public class Birds {
    private String color;// 颜色
    // 鸟还有飞行行为
    private Fly fly;
    //sets&gets
    //鸟类的飞行行为，是根据飞行对象确定飞行方式。
    public void fly() {
        this.fly.fly();
    }
}
```

5 使用多态调用鸟类的飞行方法。

```
public class Slingshot {
    public void zou(Birds birds){
        System.out.print(birds.getColor()+"的鸟用：");
```

```
        birds.fly();
    }
}
```

以上内容我们实现一种叫"策略模式"的设计模式。

8.3 接口应用的练习

收银员结算：将商品总价以参数的方式传给结算的方法，在结算方法中对总价进行打折的处理。

```
/**
 @param double sumprice 是实际商品总价
 @return double 是经过打折之后的应收商品总价
*/
```

结算方法原型 public double 方法名(double sumprice);

打折的方式有以下几种：

1、 不打折

2、 按折扣率，如 7 折、8 折

3、 按比例返现，如满 200 减 20，满 300 减 69 等。

这时我们发现打折的行为是在程序中要变化的部分。所以要单独封装。

实现步骤：

1、 定义打折接口，声明打折方法。

```
/**
```

@param double sumprice 是实际商品总价

@return double 是经过打折之后的应收商品总价

*/

Public double 打折方法(double sumprice);

2、实现打折的三种方式，分别编写三种方式的实现类。

3、编写收银员类，让收银员“有”打折的行为。在收银员的结算方法中调用打折行为进行打折计算。

9 面向对象设计原则

1、找出应用中可能需求变化的代码，把它们独立出来，不要和那些不需求变化的的代码混在一起

2、针对接口编程，而不要针对实现类编程

3、多用组合，少用继承

4、为了交互对象之间的松耦合设计而努力

5、类应该对扩展开放，对修改关闭

6、依赖倒置，要依赖抽象，不要依赖具体类

9.1 案例 1：做披萨-设计模式（简单工厂模型）

```
public abstract class Pizza {
    public abstract void prepare();
    public abstract void make();
    public void bake() {
        System.out.println("烘焙");
    }
```

```
    public void cut() {
        System.out.println("用刀切");
    }
    public void box() {
        System.out.println("装盒");
    }

}
```

```
public class PizzaStore {
    public void createPizza(String pizzaname){
        Pizza pizza = Factory.createPizza(pizzaname);
        //pizza制作的过程这5个步骤是不变的。
        pizza.prepare();
        pizza.make();
        pizza.bake();
        pizza.cut();
        pizza.box();
    }
}
```

```
public class SeafoodSupremePizza extends Pizza {

    @Override
    public void prepare() {
        System.out.println("大虾、蟹柳等丰富海鲜美味荟萃，配上酸甜菠萝、青椒，海鲜美味扑面而来。");
    }

    @Override
    public void make() {
        System.out.println("制作个人装比萨(海鲜至尊) ");
    }

}
```

```
public class Factory {
    public static Pizza createPizza(String pizzaname) {
        // 要随需求变化而改变的部分。
        if (pizzaname.equals("海鲜至尊")) {
            return new SeafoodSupremePizza();
        } else if ("超级至尊".equals(pizzaname)) {
            return new SuperSupremePizza();
        } else if ("必胜客缤纷双享比萨".equals(pizzaname)) {
            return new SausageBitePizza();
        }
        return null;
```

```
    }
}
```

9.2 案例 2：商店打折

参见：8.3 接口应用的练习

收银员结算：将商品总价以参数的方式传给结算的方法，在结算方法中对总价进行打折的处理。

```
/**
 @param double sumprice 是实际商品总价
 @return double 是经过打折之后的应收商品总价
*/
```

结算方法原型 public double 结算方法(double sumprice);

结算方法中有打折的方式有以下几种：

1、 不打折

2、 按折扣率，如 7 折、8 折

3、 按比例返现，如满 200 减 20，满 300 减 69 等。

这时我们发现打折的行为是在程序中要变化的部分。所以要单独封装。

实现步骤：

1、 定义打折接口，声明打折方法。

```
/**
```

@param double sumprice 是实际商品总价

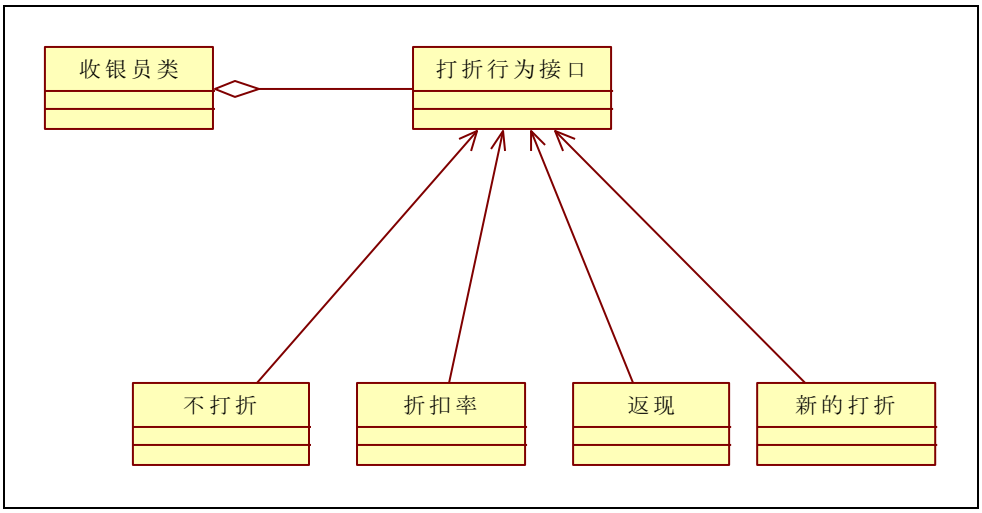
@return double 是经过打折之后的应收商品总价

*/

Public double 打折方法(double sumprice);

2、 实现打折的三种方式，分别编写三种方式的实现类。

3、 编写收银员类，让收银员“有”打折的行为。在收银员的结算方法中调用打折行为进行打折计算。



策略模式：Strategy

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

10 Object 类

是 Java 中所有类的顶层父类。

如果我们创建一个类，但在这个类上没有指明父类，系统默认加

Object 做为这个类的父类。

一个方法如果把 Object 作为这个方法的形参，说明什么？这个方法可以操作所有类型的对象。

10.1 Object 类的方法

boolean	equals(Object obj) 指示其他某个对象是否与此对象"相等"。
String	toString() 返回该对象的字符串表示。
Class<?>	getClass() 返回此 Object 的运行时类。
int	hashCode() 返回该对象的哈希码值。

10.1.1 equals 方法

比较是否相等。在 Object 类中 equals 方法比较是两个引用是不是引用同一个内存空间。

在 Object 类中 equals 方法与==是一个功能。

String 类中对 equals 方法进行了重写，重写成比较两个字符串的值（字面量）是否相等。

在我们自己定义的类中，重写 equals 方法可以将比较的方式从引用是否相等，重写成值（字面量）是否相等。

10.1.2 getClass 方法-反射时

返回对象的类。

10.1.3 toString 方法

返回该对象的字符串表示

当我们输出一个对象时，默认就是调用这个对象的 toString 方法。

10.1.4 hashCode 方法-集合类

返回该对象的哈希码值。

11 包装类

Java 中有 8 个原始数据类型，byte short int long,float double,char,Boolean

byte b = 10;

原始数据与引用数据类型之间不能进行转换（自动向上转型，强制类型转换）的。

自动类型转换：float f = 10;

强制类型转换：int i = (int)10.2;

这个是错的：String s = 10; int j = (int)"20"; XXXX

包装类作用就是将原始数据类型与引用数据类型之间进行转换的

类。

11.1 String → int

static int	parseInt(String s) 将字符串参数作为有符号的十进制整数进行解析。

```
String s1 = "10";
int i = Integer.parseInt(s1);
System.out.println(i+1);
```

其他的包装类也有类似的 parse 方法。功能一致。用法相同。

11.2 在包装类中的常量

11.3 在包装类中的方法（静态方法）

11.4 在 JDK1.5 中加入一个新特性：自动拆装箱。

Jdk1.5 之前：必须通过方法将 int 类型封装成 Integer 类型的对象。

```
Integer i1 = new Integer(10);
System.out.println(i1.intValue());
```

Jdk1.5 时，加自动拆装箱

12 内部类

在一个类中定义的类，称为内部类。

12.1 成员内部类：

就像在类中定义一个属性。内部类是属于实例的。

12.1.1 声明一个成员内部类。

```
public class OutClass {
   public class InnerClass{
   }
}
```

12.1.2 使用成员内部类

1 在外部类中使用：在外部类中的任意位置都可以直接使用成员内部类。

```
public class OutClass {
   public class InnerClass {
   }
   private InnerClass class1;
   public void dis() {
      InnerClass class1;
   }
}
```

2 在外部类外使用：

成员内部类就是外部类的一个成员。与属性的使用方法相似。

必须先有外部类的实例，才能通过这个外部类的实例创建成员内部类的实例。

```
OutClass outClass = new OutClass();

InnerClass innerClass =  outClass.new InnerClass();

InnerClass innerClass1 = new OutClass().new InnerClass();
```

12.2 静态内部类

一个使用 static 修饰符修饰的成员内部类就是一个静态内部类。

12.2.1 创建一个静态内部类

```
public class OutClass {
    public static class InnerClass {
        public void dddd(){
            System.out.println("ddd");
        }
    }
}
```

12.2.2 使用静态内部类

1 在外部类中使用：

与使用这个类中的静态属性的方式是一样的。

与成员内部类的使用方式一致。

2 在外部类外使用：

```
//成员内部类的使用方式

OutClass.innerClass1 class1 = new OutClass().new innerClass1();
```

```
        //静态内部类的使用方式
        OutClass.InnerClass innerClass = new OutClass.InnerClass();
```

12.3 局部内部类

在一个方法中定义的类，称为局部内部类。

12.3.1 创建局部内部类

局部内部类没有访问修饰符

```
public void test(){
    //局部内部类
    class InnerClass2{};
}
```

12.3.2 局部内部类的使用

只能在定义局部内部类的方法中使用。

```
public void test(){
    //局部内部类
    class InnerClass2{};
    InnerClass2 class2 = new InnerClass2();
}
```

12.4 匿名内部类

现在使用最多的一种内部类的形式。

匿名：没有名字。

匿名内部类是一种没有名字的类的形式。

匿名内部类时不用去创建的。

12.4.1 匿名内部类的使用

针对接口和针对抽象类，使用过程中，

抽象类一般要有子类（extends），重写抽象方法。才能实例化子类。

接口一般要有实现（implements）类，重写抽象方法。才能实例化实现类。

也可以使用在普通类上。

```
public interface MyInterface {
    public void dis();
}
```

```
MyInterface myinter = new MyInterface(){
    @Override
    public void dis() {
        System.out.println("disdisdis!!!#$%$@%^#$");
    }
};
myinter.dis();
```