第五章：面向对象

教学内容：

1. 面向对象简介
2. 面向对象的方法实现抽象
3. 类与对象，消息与方法
4. 构造函数，方法的重载
5. 封装
6. 继承，抽象，方法重写
7. 多态
8. 设计模式之工厂模式

一、 面向对象简介

面向对象作为一种思想及编程语言，为软件开发的整个过程：从分析设计到实现，提供了一个完整解决方案。面向对象堪称是软件发展取得的里程碑式的伟大成就。

从 80 年代后期开始，进行了面向对象分析（OOA）、面向对象设计（OOD）和面向对象程序设计（OOP）等新的系统开发方式模型的研究 。

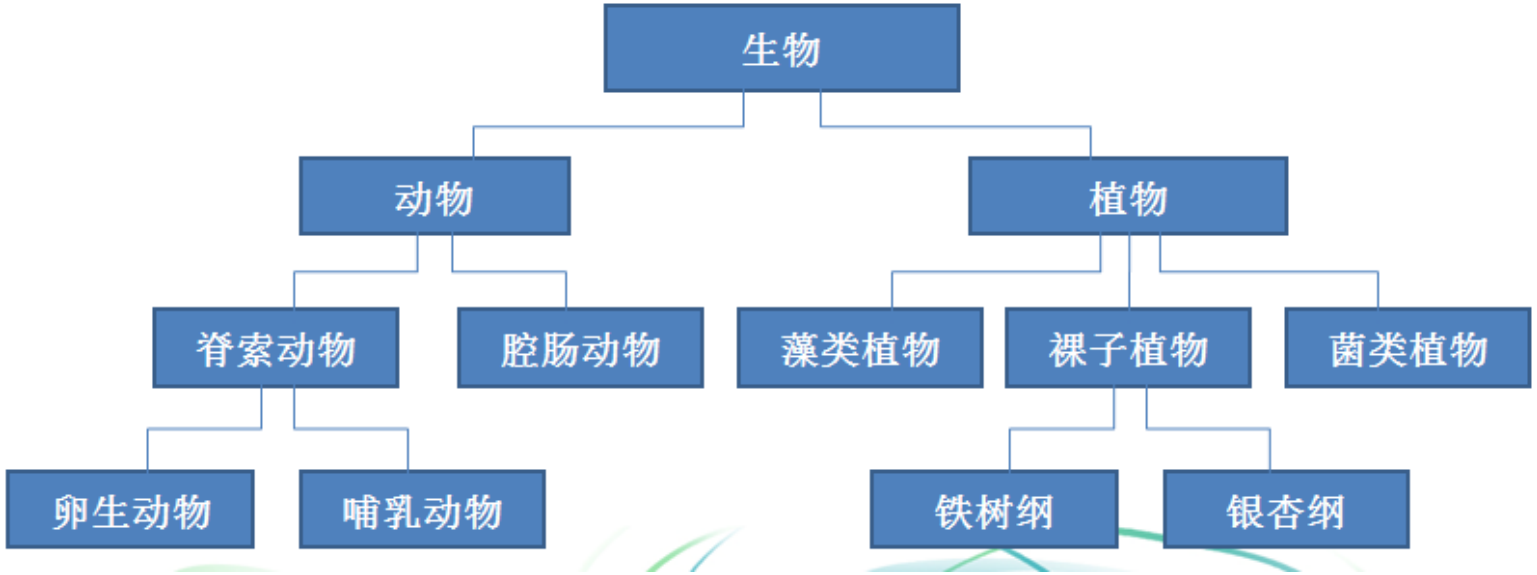
类与对象的概念：

对象：现实世界中每个事务都是一个对象；即某一个类型事务的实例。

类：对象的抽象称之为类；分类、类型、模型。

PS：每一个苹果我们可以把它看作成一个对象，一个实例。将所有的苹果抽象出来的叫做苹果类。蚂蚁和小鸟都是动物类的一个对象、实例。

二、 面向对象的方法实现抽象



抽象：就是抽取，提取。

PS：如比说人，这是一个相对抽象的概念，抽象的概念一般是指每一类事务，而不是某一个事务。而对象是一个具体的概念，一个具体的人具有性别、年龄、身高及技能操作等具体特征。而抽象的“人“不具有这些特征，具体的人是指比如说张三、李四等等。

三、 类与对象，消息与方法

a) OOP 中的类：

1.类实质上定义的是一种对象类型，它是对具有相似属性行为的对象的一种抽象；

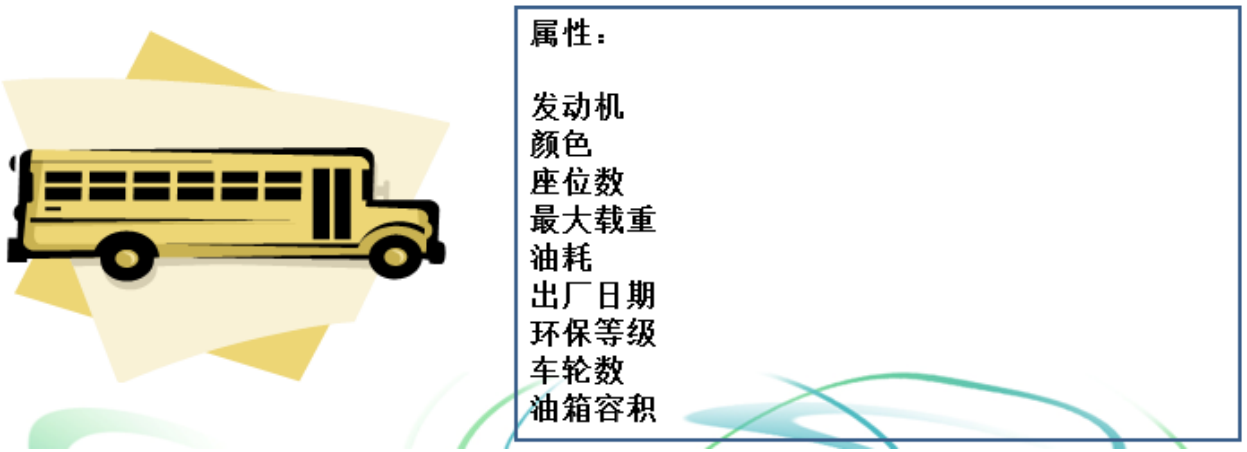
2.描述了属于该类型的对象的性质——统一的属性和操作方式。

3.属性是指描述对象的一组数据，表现为对象的一些变量。

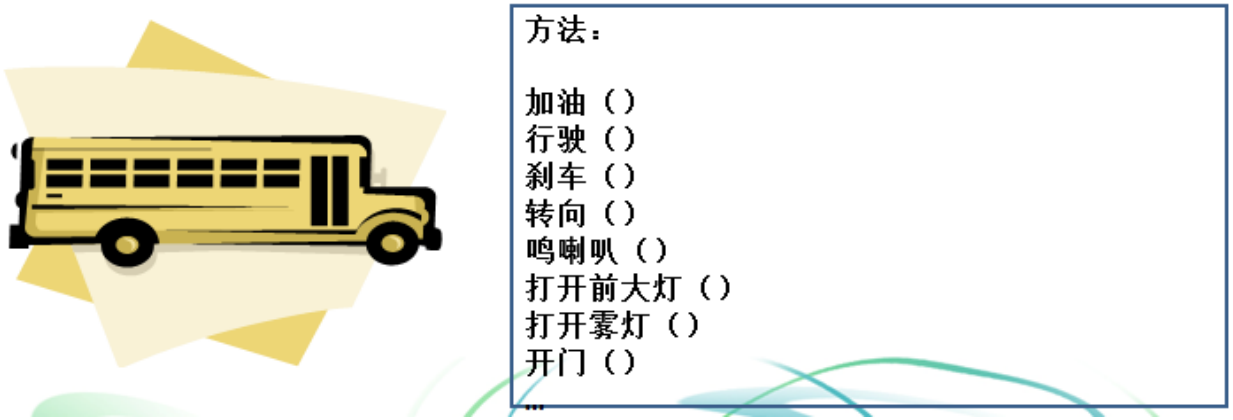
4.操作代码或方法表示对象的行为或所作的工作。

总结：类，是创建对象的模板。

b）类的属性：是指描述对象的一组数据，代码中的具体表现形式为变量。



c）类的方法：操作代码或方法表示对象的行为或所作的工作。



例：创建一个汽车类

- 汽车类
  - 属性：
    - 发动机
    - 颜色
    - 座位数
    - 最大载重
    - 油耗
    - 出厂日期
    - 环保等级
    - 车轮数
    - 油箱容积
    - …
  - 方法：
    - 加油（）
    - 行驶（）
    - 刹车（）
    - 转向（）
    - 鸣喇叭（）
    - 打开前大灯（）
    - 打开雾灯（）
    - 开门（）
    - …

```
public class AutoCar
{
    String engine;
    String color;
    int seatNo;
    int maxLoad;
    Date madeDate;
    int  environmentalGrade;
    int cartwheelNo;
    int oil;

    void addOil(){}
    void drive(){}
    void lock(){}
    void turn(){}
    void honk(){}
    void openLight(){}
    void openFogLight(){}
    void openDoor(){}

}
```

d) 消息与方法

对象是类的实例。尽管对象的表示在形式上与一般数据类型十分相似，但是它们之间存在一种本质区别：

对象之间通过消息传递方式进行通信。 消息传递原是一种与通信有关的概念，OOP 使得对象具有交互能力的主要模型就是消息传递模型。对象被看成用传递消息的方式互相联系的通信实体，它们既可以接收可以拒绝外界发来的消息。一般情况下，对象接收它能够识别的消息，拒绝它不能识别消息。对于一个对象而言，任何外部的代码都不能以任何不可预知或事先不允许的方式与这个对象进行交互。 发送一条消息至少应给出一个对象的名字和要发给这个对象的那条消息的名字。经常，消息的名字就是这个对象中外界可知的某个方法的名字。在消息中，经常还有一组数(也就是那个方法所要求的参数)，将外界的有关信息传给这个对象。

PS：老师有教书的方法，那么老师可以通过教的方法将知识传输给学生，那么学生需要有学习的方法。通过学习的方法来接受知识。

```java
public class Teacher {

    void teach(String content) {
        System.out.println("开始传授" + content + "...");
        Student stu = new Student();
        stu.study(content);
    }

    public static void main(String[] args) {
        Teacher t = new Teacher();
        t.teach("java");
    }

}

class Student {
    void study(String content) {
        System.out.println("学习到了" + content + "...");
    }
}
```

定义类的语法：

```
class 类的名称{
    [访问修饰符]    数据类型    属性名;    //声明成员变量
    ...
    [访问修饰符]    返回值类型        方法名([参数列表])  {
        程序代码语句;
```

```
        return 值或表达式;
    }
}
```

成员变量的定义：又叫实例变量，定义在类中，方法外的变量叫做成员变量，属于某一个对象。

成员方法：又叫实例方法，必须通过某个对象来进行访问调用。

语法说明：[]中的内容可有可无

1. 访问修饰符有 4 种：（在后面访问修饰符的时候详细说明）

   a) public：公共的；

   b) 不写：默认的；

   c) protected：受保护的；

   d) private：私有的；

   PS：目前一般写成是 public 或者不写。

2. 数据类型：包含基本数据类型和引用数据类型

3. 属性名、方法名：同变量的定义规则；

4. 返回值类型：

   a) 没有返回值使用 void；

b）有返回值可以是基本数据类型或者引用数据类型

5．参数列表：包含参数的类型，个数，顺序。

6．程序代码：顺序结构、分支语句、循环语句；

7．return：只有当方法有返回值的时候才需要，值或表达式结果的类型必须和返回值类型相同或者它的小类型。

```
//没有返回值没有传参
void eat(){
     System.out.println("eat");
}

//没有返回值有传参
void eat(String str,int a){
     System.out.println("eat");
}

//有返回值有传参
int eat(String str,int a){
     System.out.println("eat");
     return 100;//换成(byte)100;
}
```

访问类中的属性和方法：如果需要去访问类中的属性和方法，则需要先创建该类的对象。

调用属性：对象名.属性名;

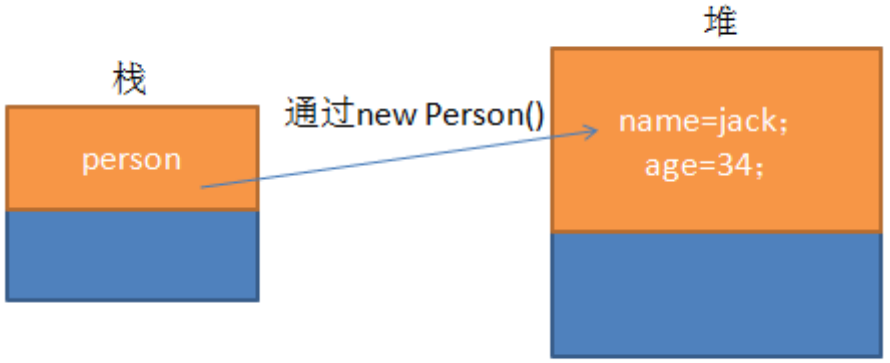
调用方法：对象名.方法名(参数列表);

```
public class Person {

    String name;                            // 定义属性，人的姓名
    int age;                                // 定义属性，人的年龄

    /**
     * 定义方法，显示人的姓名和年龄
     */
    public void show() {
        // +和字符串一起使用的时候表示连接而非加分运算
        System.out.println("姓名:" + name + ",年龄:" + age);
    }

    public static void main(String[] args) {
        Person person = new Person();       // 创建人的对象，即某一个成员
        person.name = "jack";               // 初始化人的姓名，通过赋值=将jack赋值给该对象的姓名属性
        person.age = 34;                    // 初始化人的年龄，通过赋值=将34赋值给该对象的年龄属性
        person.show();                      // 调用show()方法
    }

}
```

PS:通过对象调用属性和方法，可以在同一个类的 main 方法中调用，也可以在不同的类中进行调用。

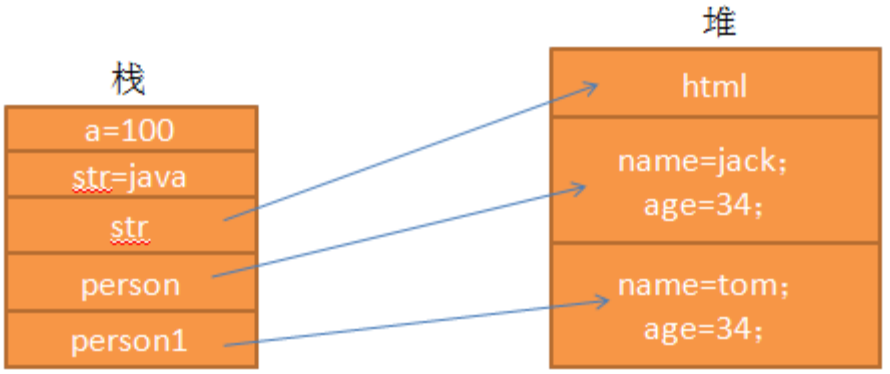
内存的操作，为属性赋值：

```
public static void main(String[] args) {
    Person person = new Person();
    person.name = "jack";
    person.age = 34;
    person.show();
}
```



内存划分

```
public static void main(String[] args) {
    int a = 100;
    String str = "java";

    String str1 = new String("html");

    Person person = new Person();
    person.name = "jack";
    person.age = 34;

    Person person1 = new Person();
    person.name = "tom";
    person.age = 34;
}
```



PS：栈空间保存的是基本数据类型和字符串，包括局部变量的引用，堆空间保存的动态产生的数据，比如 new 创建出来的对象，也就是引用数据类型。上面两个 Person 实例在堆空间开辟了两块内存，那么相互之间进行调用的时候就不会受到影响，只要使用了 new 关键字，必会在堆空间开辟内存。

四、 构造函数，方法重载

要使用面向对象，首先必须构造对象，并指定它们的初始状态，然后通过对象调用方法。

在 java 的语言设计种，使用构造函数(constructor)来构造新的实例，一个构造函数是新的方法，它的作用就是构造对象并进行初始化.

（1）构造函数的方法名与类名相同。

（2）构造函数没有返回类型。

（3）构造函数的主要作用是完成对类对象的初始化工作。

（4）构造函数不能由编程人员显式地直接调用。

（5）在创建一个类的新对象的同时，系统会自动调用该类的构造函数为新对象初始化。

```
public class Person {

    String name;
    int age;

    //要使用Person的方法，必须先创建Person类的对象，在创建对象的同时对name，age属性进行了初始化
    public Person(String name, int age) {
```

```
        super();
        this.name = name;
        this.age = age;
    }

}
```

构造函数的特点有：

（1）构造函数和类具有相同的名字。

（2）一个类可以有多个构造函数。

（3）构造函数可以有 0 个、1 个或多个参数。

（4）构造函数没有返回值。

（5）构造函数总是和 new 关键字一起被调用。

```
public class Person {

    String name;
    int age;

    public Person() {

    }

    public Person(String name, int age) {
```

```
        super();
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        super();
        this.name = name;
    }

    public Person(int age) {
        super();
        this.age = age;
    }

}
```

构造函数的作用：

（1）对象初始化

（2）引入更多的灵活度（变量赋值或更复杂的操作）

（3）Java 中可以不定义构造函数

Java 中可以不定义构造函数，此时系统会自动为该系统生成一个默认的构造函数。这个构造函数的名字与类名相同，它没有任何形式参数，也不完成任何操作。 为了避免失去控制，一般将构造函数的声明与创建分开处理。

构造函数重载：

一个类可以有多个构造函数，如果一个类没有定义一个构造函数，Java 编译器将为这个类自动提供缺省构造函数(即无参的构造函数），缺省构造函数将成员变量的值初始化为缺省值，一旦创建了自己的构造函数， Java 编译器将不再自动提供无参的构造函数。

重载构造函数提供了一组创建对象的方式，可以根据需要决定是否带初始参数。根据参数列表决定调用的是哪个重载的构造函数。

构造函数重载：一个类中可以有多个构造函数，它们具有不同的参数列表(参数的类型、个数、顺序)。

```
public class Person {

    String name;
    int age;

    public Person() {
        System.out.println("无参的构造函数");
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("带两个参数的构造函数1");
    }

    public Person(int age, String name) {
        this.name = name;
```

```
        this.age = age;
        System.out.println("带两个参数的构造函数2");
    }

    public Person(String name) {
        this.name = name;
        System.out.println("参数为name的构造函数");
    }

    public Person(int age) {
        this.age = age;
        System.out.println("参数为age的构造函数");
    }

}
```

this 关键字：与对象关联，表示当前对象（实例），即 new 出的是哪个对象，代表的就是哪个对象.

this 关键字的作用：可以调用调用类中的构造方法，普通方法，成员变量。this 调用构造方法只能是在构造方法中使用，必须是第一行.

```
public class Person {

    String name;
    int age;

    public Person() {
```

```
        this("jack");//如果使用this调用构造方法，必须在第一行，只能在构造方法种调用构造方法。
        System.out.println("无参的构造函数");
        this.show();
    }

    public Person(String name) {
        this(20);
        System.out.println("参数为name的构造函数");
    }

    public Person(int age) {
        System.out.println("参数为age的构造函数");
    }

    public void show(){
        //this(); //不可调用。
        System.out.println("show");
    }

    public static void main(String[] args) {
        new Person();
    }

}
```

方法重载：方法重载是 java 中实现面向对象多态性机制的一种方式。

同一个类中多个方法有相同的名字，不同的参数列表（参数的个数，类型，顺序），和返回值类型无关，这种情况称为方法重载。

当重载方法被调用时，Java 编译器根据参数的类型、数量、顺序来确定实际调用哪个重载方法的版本。方法重载不考虑方法的返回类型。

方法重载的示例：

```
public class Calculation {

    public int add(int a, int b) {
        return a + b;
    }

    public float add(float a, float b) {
        return a + b;
    }

    public String add(String a, String b) {
        return a + b;
    }

    public void add(String a, float b) {
        System.out.println("字符串在前:" + (a + b));
    }

    public void add(float b, String a) {
        System.out.println("字符串在后:" + (a + b));
    }
```

```
}
```

main 函数：

main 方法表示一个 Java 应用程序执行的起点。

Java 是完全面向对象的程序设计语言，main 方法必须放在一个类中定义。

一个 Java 程序可以包含一个或多个类，但在 application 环境下有一个类(只能有一个类)必须定义一个 main 方法。

new 关键字：

使用下列语法可创建对象：

new 构造函数

关键字 new 通常称为创建运算符，用于分配对象内存，并将该内存初始化为缺省值。 一旦 new 完成分配和初始化内存，它就将调用构 造函数来执行对象初始化。

使用对象：

要想使用对象，必须获得一个对象的引用。一般使用赋值语句，例如：

```
Person ren = new Person();
```

上述形式的赋值语句实际上做了 3 件事：

1. Person ren：声明了一个 Person 类型的变量（对象变量）ren，该变量此时不指向任何对象。

2. new Person()：生成了一个 Person 类型的对象，该对象是一个无名对象。

3. = 将这个无名对象的引用赋值给对象变量 ren

当对象变量 ren 获得了一个对象的引用后，就可以把 ren 称作是一个 Person 类型的对象

变量的作用域：

按照变量的作用域分类，类中只有两种变量：

实例变量：在所有方法之外但在类体中声明或定义的变量

局部变量：在方法中声明或定义的变量

实例变量的有效范围是整个类；

局部变量的有效范围在方法体之中，出了方法体就自动消失了；

变量初始化：

任何变量在使用前必须被初始化，对于实例变量，Java 编译器会用缺省初始值自动进行初始化。对于局部变量 java 编译器不会自动初始化，局部变量在使用之前必须赋初值。

类型	缺省值	类型	缺省值
byte	(byte)0	char	'\u0000'
short	(short)0	float	0.0F
int	0	double	0.0D
long	0L	对象引用	null
boolean	false		

方法的参数：

方法的参数只有两种类型：基本数据类型和引用数据类型。

1.在 Java 中，参数的传递只有一种方式，就是按值传递（传递自身的拷贝）：

2.对于基本数据类型，传递的值就是变量自身的值

3.对于对象类型，传递的值就是对象的引用（而不是对象自身！）

方法不能改变传递给它的参数的自身

例 1：基本数据类型作为参数

```
public class Test {

    public void change(int a){
        a=20;
    }

    public static void main(String[] args) {
        int a = 10;
```

```
        System.out.println(a);//10
        Test t = new Test();

        t.change(a);
        System.out.println(a);//10
    }

}
```

例 2：对象类型作为参数

当对象作为参数时，在方法中只能改变对象的状态不能改变对象的引用

```
public class Test {

    public void change(Person person) {
        person.age = 20;// 试试下面的代码
        // Person person1 = new Person();
        // person1.age = 30;
        // person = person1;
    }

    public static void main(String[] args) {

        Person person = new Person();

        person.age = 10;

        System.out.println(person.age);// 10
```

```
        Test t = new Test();

        t.change(person);

        System.out.println(person.age);//20

    }

}

class Person {

    int age;

}
```

例 3：字符串作为传递参数

```
public class Test {

    public void change(String str) {
        str = "jack";
    }

    public static void main(String[] args) {
        //字符串是一个特殊的引用数据类型。它的传递方式和基本数据类型一致
        String str = "tom";

        System.out.println(str);
        Test t = new Test();
```

```
        t.change(str);
        System.out.println(str);
    }

}
```

例 4：数组作为传递参数

```
public class Test {
    public void change(int[] arr) {
        // arr[0]=20;//试试看
        int[] arr2 = { 30 };
        arr = arr2;
    }

    public static void main(String[] args) {

        int[] arr = { 10 };
        System.out.println(arr[0]);// 10
        Test t = new Test();
        t.change(arr);
        System.out.println(arr[0]);

    }

}
```

五、 封装

a)封装是对象的一种隐藏技术，其目的是将对象中的属性和方法组织起来。同时隐藏不想暴露的属性和方法及实现细节。

b)用户或其它对象不能看到也无法修改其实现。只能通过接口去调用对象的方法，达到互相通信的目的。

c)封闭的目的在于将设计者与使用者分开。使用者不必知道实现的细节，只需用设计者提供的方法来访问该对象。

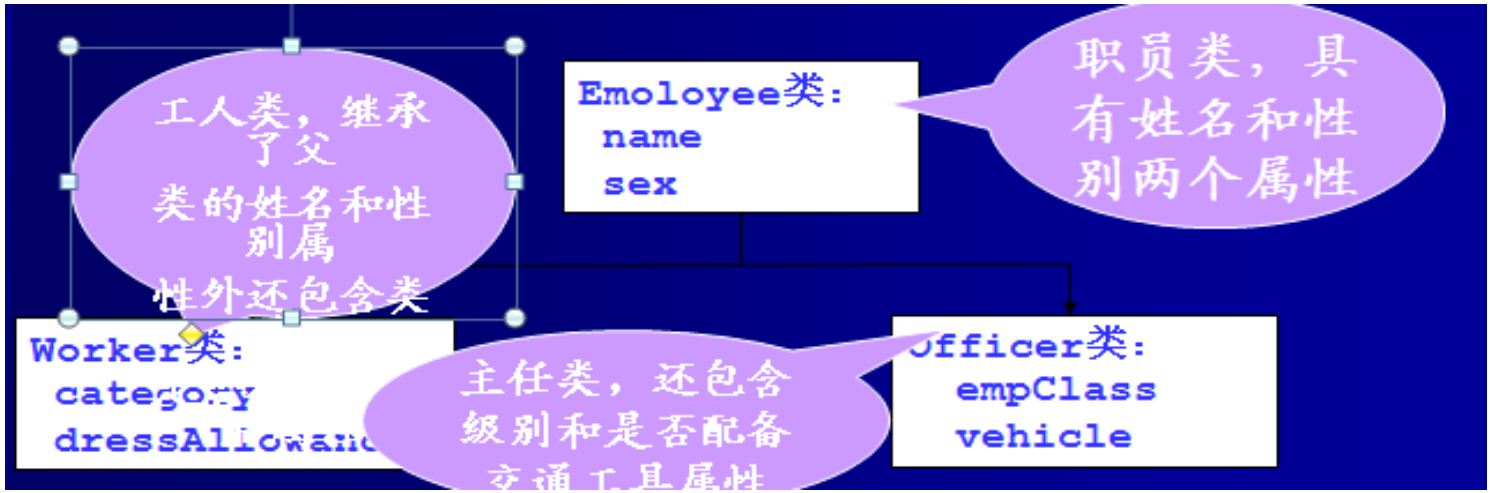
```
public class DrawImage {

    private int weight;

    private int height;

    public void setHeight(int height) {
        if (height > 100)
            System.out.println("高度不能高于100");
        else
            this.height = height;
    }

    public int getHeight() {
        return height;
    }

    public void setWeight(int weight) {
        if (weight > 200)
            System.out.println("宽度不能大于200");
```

```
            else
                this.weight = weight;
        }

        public int getWeight() {
            return weight;
        }

        public void draw() {
            drawLine();
            drawPoint();
            System.out.println("你画了一幅画");
        }

        private void drawPoint() {
            System.out.println("画了一个点");
        }

        private void drawLine() {
            System.out.println("画了一条线");
        }

}
```

六、 继承、方法重写

继承是面向对象编程技术的一块基石，它允许创建分等级层次的类。运用继承，可以创建一个通用类定义一系列一般特性。该类可以被更具体的类继承.

- **被继承的类称为父类或者超类或者基类**
- **继承父类的类称为子类或者派生类**
- **执行继承时，子类将获得父类的所有成员（包括 private 的成员,私有的不能直接访问），并具有自身特有的属性。**



extends 关键字用于继承类：

```
/**
 * 父类
 * @author Administrator
 *
 */
```

```
public class Father {

}
```

```
/**
 * 子类继承父类
 * @author Administrator
 *
 */
public class Son extends Father {

}
```

注意：

没有 extends，默认父类为 Object

一个类只能有一个父类，即单继承

子类继承父类的全部成员

继承的示例：

```
/**
 * 父类
 * @author Administrator
```

```java
 *
 */
public class Person {

	private String name;

	private int age;

	public Person(String name, int age) {
		super();
		this.name = name;
		this.age = age;
	}

	public int getAge() {
		return age;
	}

	public String getName() {
		return name;
	}

}


/**
 * 子类
 * @author Administrator
 *
 */
public class Student extends Person {
```

```
    private String school;// 学校

    public Student(String name, int age, String school) {
        super(name, age);
        this.school = school;
    }

    public String getSchool() {
        return school;
    }

}

/**
 * 测试类
 * @author Administrator
 *
 */
public class Test {

    public static void main(String[] args) {
        Student stu = new Student("jack", 20, "南京艾瑞");

        System.out.println("学生信息：");
        System.out.println("姓名：" + stu.getName());
        System.out.println("年龄：" + stu.getAge());
```

```
        System.out.println("学校：" + stu.getSchool());
    }
}
```

示例说明：学生、工人等都是人类，学生继承了人类，拥有了人类的 name、age 属性，学生自己拥有 school 属性，这是子类特有的。

- 类的继承，是为了子类能够访问父类中已经定义的属性和方法，就如同是自己的属性和方法一样。
- 继承最大限度的重用了代码。子类通过继承可以访问父类的属性和方法，而不必为一个新类编写相同的代码。
- 一个子类可以继承多个父类。但 JAVA 语言只能直接继承一个父类。通过间接继承可以继承多个父类。

super 关键字：

super 是一个引用，专门用来在子类中访问父类中的构造函数、方法和实例变量。

1.使用 super 在子类的构造函数中调用父类的构造

函数，语法：

```
super()或者 super(参数列表)
```

super()必须是在子类构造函数中的第一个执行语句。

2.在子类的方法中调用父类中的方法，语法：

super.方法名(参数列表)

3.在子类的方法中调用父类中的实例变量， 语法：

super.实例变量名

继承总结：

1.子类继承父类，继承了父类的所有属性和方法(包含私有的)，私有的不能直接访问；

2.一个类如果没有使用 extends，那么它将继承 Object 类，Object 类是所有类的父类，始祖类；

3.一个类可以继承多个类，但 java 中规定一个类只能直接继承一个类；可以间接继承；

4.子类具有扩展的功能，扩展子类特有的属性和方法；

5.继承大大提供了代码的重复利用性；

方法重写（覆盖）override：是 java 实现多态机制的另外一种形式。

在不同类中，如果子类中的一个方法与父类中的方法有相同的返回类型、相同的方法名并具有相同数量和类型的参数列表，这种情况称为方法覆盖。

当一个覆盖方法通过父类引用被调用，Java 根据当前被引用对象的实际类型来决定执行哪个版本的方法。

可以通过 super 关键字调用直属父类中被覆盖的方法版本。

方法重写示例：

```
public class Father {

     public String say(){
          return "我是父类";
     }

}


public class Son extends Father {

     @Override
     public String say() {
          // super.say();//当创建的是子类对象的时候，如果需要调用父类的方法可以使用super关键字
          return "我是儿子类";
     }
}


public class Daughter extends Father{

     @Override
     public String say() {
          return "我是女儿类";
     }

}
```

```
public class Test {

     public static void main(String[] args) {
          Father f = new Father();

          f.say();// 创建的是父类对象，调用的是是父类方法

          Father s = new Son();

          s.say();// 创建的是儿子类对象，调用的是儿子类方法

          Father dau = new Daughter();

          dau.say();// 创建的是女儿类对象，调用的是女儿类的方法
     }

}
```

抽象类：

定义：一种类型，只提供部分方法的具体实现。

语法：abstract class 类名{….}

一般情况下，抽象类既包含具体方法，又包含抽象方法。

具体方法：既有方法的声明，又有方法的实现（即有方法体）。

抽象方法：只有方法的声明，而没有方法的实现（即没有方法体)。语法：

```
abstract 返回类型 方法名(参数列表)
```

抽象类的示例：

```
public abstract class Father {

	String name;

	int age;

	public Father() {
	}

	public Father(String name, int age) {
		this.name = name;
		this.age = age;
	}

	public abstract void say();// 抽象方法，没有方法体

	public void see() {
		System.out.println("see");
	}

}


public class Son extends Father {
```

```
    public Son() {
    }

    public Son(String name, int age) {
        super(name, age);
    }

    // 必须要重写父类的所有抽象方法，包含间接父类
    @Override
    public void say() {
        System.out.println("我必须要重写父类的所有抽象方法");
    }

}


public class Test {

    public static void main(String[] args) {
        Father f = new Son();//抽象类不能被实例化，必须通过子类进行实例化

        f.say();
    }

}
```

总结：

抽象类不能被实例化。

抽象类就是用来继承的

子类必须为抽象类中的所有抽象方法提供具体实现，否则，子类也将是一个抽象类

抽象类中可以声明实例变量，这些实例变量就是为了提供给子类继承的

抽象类可以有一个或多个构造函数，它是提供给子类进行调用的

特别地，抽象类中的所有方法都可以是具体方法

里氏代换原则：无论何时，只要程序需要一个超类对象，那么就可以用一个子类对象来替代它。

```
Father f = null;
f = new Father();//创建的是父类对象，需要父类对象
f = new Son();//可以通过创建子类对象类代替它
```

注意：反过来是不行的。

里氏代换原则示例:

需求：需要计算两个数相加

```
/**
 * 算术类：提供了计算的方法，由A这个类来完成
```

```
 * @author Administrator
 *
 */
public class A {

	public int compute(int a, int b) {
		return a + b;
	}
	public int compute_(int a, int b) {
		return 0;
	}


}
```

需求变更：需要在相加的功能的基础上再扩展一个功能 相加后除以 2

```
/**
 * 扩展的功能有B类来完成
 * @author Administrator
 *
 */
public class B extends A {

	public int compute(int a, int b) {
		return a - b;
	}
```

```
    //子类扩展的功能
    public int compute_(int a, int b) {
        return (a + b) / 2;
    }

}


public class Test {


    public static void main(String[] args) {
        A a = new B();//需要使用父类的方法，可以通过创建子类对象类代替它
        System.out.println(a.compute(20, 30));
        System.out.println(a.compute_(20, 30));
    }

}
```

分析：我们发现 B 类中无意重写了 A 类中的方法，导致了运算的结果不对。

里氏替换原则通俗的来讲就是：任何父类可以出现的地方，子类一定可以出现，子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下含义：

- 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
- 子类中可以增加自己特有的方法。

根据上面的说法我们可以将以上需求的代码修改成：

```java
public abstract class A {

    public int compute(int a, int b) {
        return a + b;
    }

    public abstract int compute_(int a, int b);

}


public class B extends A {

    // 子类扩展的功能
    public int compute_(int a, int b) {
        return (a + b) / 2;
    }

}


public class Test {


    public static void main(String[] args) {
        A a = new B();//需要使用父类的方法，可以通过创建子类对象类代替它
        System.out.println(a.compute(20, 30));
        System.out.println(a.compute_(20, 30));
    }
```

```
}
```

七、 多态

多态和动态绑定：

声明类型：对象变量被声明时的类型

实际类型：对象变量实际指向的对象类型

一个对象变量可以指向多种实际类型的现象称为“多态”

在运行时自动选择正确的方法进行调用的现象称为“动态绑定”

Java 根据对象的实际类型来进行方法调用

编译时多态：编译时动态重载.（方法重载）

运行时多态：指一个对象可以具有多个类型。（方法重写）

多态示例 1：

```
public class Test {
    public void add(int a, int b) {
    }
```

```
    public void add(short a, short b) {

    }

}
```

多态示例 2：

```
public abstract class Father {

    public abstract void say();

}


public class Son extends Father {

    @Override
    public void say() {
        System.out.println("Son is say");
    }

    public void play() {
        System.out.println("Son is play");
    }

}
```

```java
public class Daughter extends Father {

	@Override
	public void say() {
		System.out.println("Daughter is say");
	}

	public void sing() {
		System.out.println("Daughter is sing");
	}

}


public class Test {

	public static void main(String[] args) {
		Father f = new Son();

		f.say();

		// instanceof判断一个对象是否属于一个类，必须有继承关系
		if (f instanceof Son) {
			((Son) f).play();//
		}

		f = new Daughter();

		f.say();
		if (f instanceof Daughter) {
```

```
                ((Daughter) f).sing();
            }
        }

}
```

总结：运行时多态的三大前提条件

1.要有继承

2.要有方法的重写

3.父类引用指向子类对象（对于父类中定义的方法，如果子类中重写了该方法，那么父类类型的引用将会调用子类中的这个方法，这就是动态连接）

instanceof 关键字：判断一个类是否属于一个类型，必须要有继承的关系。

内部类：

在一个类中定义的类称为内部类，内部类之外的类称为外部类 ；

内部类可以访问其外部类的所有变量和方法；

内部类完全在其包围类的范围之内；

内部类中的 this 指的是内部类的实例对象本身，如果要用外部类的实例对象就可以用类名.this 的方式获得。

内部类对象中不能有静态成员，原因很简单，内部类的实例对象是外部类实例对象的一个成员

内部类：普通内部类、局部内部类、静态内不列、匿名内部类

1．　普通内部类：没有 static 修饰，且定义再类中方法外的类

```
public class Outer {
     private String name = "jack";

     public static int AGE = 20;

     public void tell() {
          System.out.println("Outer is tell");
     }

     class Inner {

          public void say() {
               System.out.println(name);
               System.out.println(AGE);
               tell();
          }

     }

}


public class Test {
```

```
    public static void main(String[] args) {
        Outer.Inner oi = new Outer().new Inner();//通过外部类的对象创建内部类的对象
        oi.say();
    }

}
```

总结：

a) 普通内部类可以访问外部类的所有属性和方法，包含私有的直接访问；

b) 普通内部类中不能有静态变量和静态方法；

c) 要想创建内部类的对象，需要通过创建外部类的对象来进行创建；

d) 外部类中可以通过创建内部类的对象来访问内部类中的属性和方法；

2． 局部内部类；

如果在程序中只使用一个类一次，可以在一个方法里定义局部类

在方法中定义的内部类称为局部内部类。

与局部变量类似，在局部内部类前不加修饰符 public 和 private，其范围为定义它的代码块。

内部类中使用外部方法中的局部变量，该局部变量必须是最终的(final)

```
public class Outer {
```

```
    private String name = "jack";

    static int COUNT = 100;

    public void say() {
        final String addr = "南京";
        class Inner {

            public void tell() {
                System.out.println(name);
                System.out.println(COUNT);
                System.out.println(addr);//必须是最终的
            }
        }
        Inner in = new Inner();
        in.tell();
    }

}


public class Test {

    public static void main(String[] args) {
        Outer out = new Outer();

        out.say();
    }
```

```
}
```

3． 静态内部类；

静态内部类定义在类中，任何方法外，用 static 定义。

静态内部类只能访问外部类的静态成员。

生成（new）一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成： Outer.Inner in=new Outer.Inner()；

而不需要通过生成外部类对象来生成。这样实际上使静态内部类成为了一个顶级类。

静态内部类不可用 private 来进行定义

```
public class Outer {
    private String name = "jack";
    private static int COUNT = 100;
    public static class Inner {
        public static void say() {
            //System.out.println(name);//只能直接访问外部静态变量和方法
```

```
            System.out.println(COUNT);
        }

        public void tell() {
            //System.out.println(name);//只能直接访问外部静态变量和方法
            System.out.println(COUNT);
        }

    }

}


public class Test {

    public static void main(String[] args) {

        Outer.Inner.say();//可以直接访问内部类的静态变量和方法
        Outer.Inner oi = new Outer.Inner();//和普通内部类的区别，可以直接创建内部类对象
        oi.tell();//成员方法需要通过对象类访问
    }

}
```

4． 匿名内部类；

如果你只需要建立一个内部类的对象，那么甚至不必为该类指定一个名字。我们把这种类称作匿名内部类。

匿名类是特殊内部类，没有名字。

匿名内部类是唯一一种无构造方法类。

因匿名内部类无构造方法，所以其使用范围非常的有限。

内部类中使用外部方法中的局部变量，该局部变量必须是最终的(final)

```
public class Outer {

     public void say(){
          System.out.println("Outer is say");
     }

}


public class Test {

     public static void main(String[] args) {

          final String name = "jack";

          Outer out = new Outer() {
               @Override
               public void say() {
                    System.out.println("Inner is say");
                    System.out.println(name);//name必须是最终的
               }
          };
```

```
            out.say();//必须是重写外部类的方法，通过调用外部类的方法，调用内部类的方法
        }

}
```

5．　为什么要使用内部类？

内部类继承自某个类或实现某个接口，内部类的代码操作 创建其的外围类的对象。所以你可以认为内部类提供了某种进入 其外围类的窗口。

每个内部类都能独立地继承自一个（接口的）实现，所以无论外围类 是否已经继承了某个（接口的）实现，对于内部类都没有影响。

内部类有效地实现了“多重继承”。

简单示例：

```
public abstract class AbsOuter {

     private String out_str="out_str";

     public String getOut_str() {
          return out_str;
     }

}
public abstract class AbsInner {
```

```
    private String inner_str = "inner_str";

    public String getInner_str() {
        return inner_str;
    }
}

public class Outer extends AbsOuter {

    private String outer = "outer";

    public class Inner extends AbsInner{

        public void say(){
            System.out.println(outer);//访问外部类的属性
            System.out.println(Outer.this.getOut_str());//访问外部类的父类属性
            System.out.println(this.getInner_str());//访问内部类的父类属性
        }
    }
}


public class Test {

    public static void main(String[] args) {
        Outer.Inner oi = new Outer().new Inner();
        oi.say();
```

```
    }
}
```

Object 类：

Object 类是 Java 类体系结构的根；

Java 系统中的每个类都是 Object 类直接或间接的子类 ；

Object 类包括在 java.lang 包中此类定义了所有对象都具备的基本状态和行为，可以用类型为 Object 的变量来引用任意类型的对象 ；

最终类：

如果一个类被声明为 final，意味着它不能再派生出新的子类，不能作为父类被继承。因此一个类不能既被声明为 abstract 的，又被声明为 final 的。

被定义成 final 的类，通常是一些有特殊作用的、用来完成标准功能的类，将一个类定义为 final 则可以将它的内容、属性和功能固定下来，与它的类名形成稳定的映射关系，从而保证引用这个类时所实现的功能是准确无误的。

八、 设计模式之工厂模式

1） 简单工厂模式

step-1：请用面向对象的思想实现计算机控制程序，要求输入两个数和运算符号得到结果

```
public class TestDemo {
```

```java
public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    System.out.println("请输入数字A:");
    int a = sc.nextInt();
    System.out.println("请输入运算符号:");
    sc = new Scanner(System.in);
    String c = sc.nextLine();
    System.out.println("请输入数字B:");
    sc = new Scanner(System.in);
    int b = sc.nextInt();
    int result = 0;
    switch (c) {
    case "+":
        result = a + b;
        break;
    case "-":
        result = a - b;
        break;
    case "*":
        result = a * b;
        break;
    case "/":
        result = a / b;
        break;
    default:
        System.out.println(a+c+b+":运算符异常...");
        return;
```

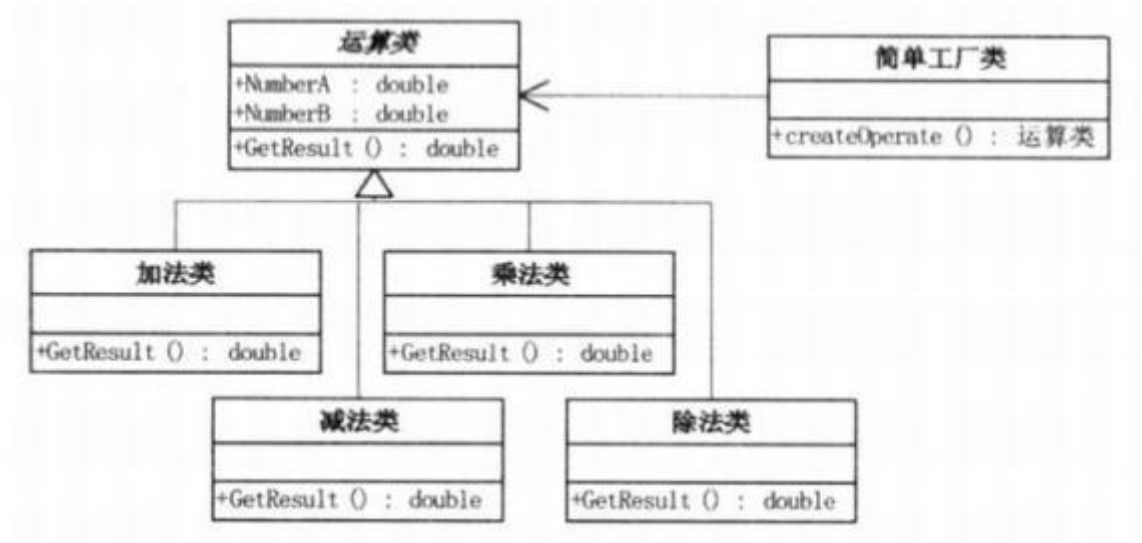
```
            }
            System.out.println(a+c+b+"="+result);
        }

}
```

step-2：使用面向对象的思想进行编程，上面的写法只能说是实现了功能，如果现在有一个层序需要调用这个计算呢？那么上面这种写法就不能满足。根据这个思路我们需要将上面的计算设计成一个类，提供对外访问的接口。让业务逻辑和界面逻辑分开，降低它们之间的耦合度，分开后才能达到容易维护和扩展。

```
public class Operation {

    public static int getResult(int numberA, int numberB, String operate) {
        int result = 0;
        switch (operate) {
        case "+":
            result = numberA + numberB;
            break;
        case "-":
            result = numberA - numberB;
            break;
        case "*":
            result = numberA * numberB;
            break;
        case "/":
            result = numberA / numberB;
            break;
        default:
            System.out.println("运算符异常...");
```

```
        }
        return result;
    }

}


public class Test {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入数字A:");
        int numberA = sc.nextInt();
        System.out.println("请输入运算符号:");
        sc = new Scanner(System.in);
        String operate = sc.nextLine();
        System.out.println("请输入数字B:");
        sc = new Scanner(System.in);
        int numberB = sc.nextInt();

        System.out.println(Operation.getResult(numberA, numberB, operate));

    }

}
```

step-3：上面的写法用到了面向对象的封装，将业务逻辑和界面逻辑分开了，业务进一步增加现在需要一个平方根的运算，再进一步需要将+、-、*、/参与到运算中，这样的业务就有了一定的复杂度，那么如果在操作的过程中不小心输错了一个符号，将会导致运行结果错误。下面就要用到所学的继承和多态。



加减乘除无论如何计算，至少都需要的是两个数，可以设计为类的属性，而不同的知识它的操作，最终都需要返回一个结果，中间的操作过程不同。实现的代码如下

```
public abstract class Operation {

    private double numberA;
    private double numberB;

    public abstract double getResult();

    public void setNumberA(double numberA) {
        this.numberA = numberA;
```

```
    }

    public void setNumberB(double numberB) {
        this.numberB = numberB;
    }

    public double getNumberA() {
        return numberA;
    }

    public double getNumberB() {
        return numberB;
    }

}


public class Add extends Operation {

    @Override
    public double getResult() {
        return getNumberA() + getNumberB();
    }

}


public class Divide extends Operation {

    @Override
    public double getResult() {
```

```
            return getNumberA() / getNumberB();
        }

}


public class Minus extends Operation {

        @Override
        public double getResult() {
            return getNumberA() - getNumberB();
        }

}


public class Multiply extends Operation {

        @Override
        public double getResult() {
            return getNumberA() * getNumberB();
        }

}


public class OperationFactory {

        public static Operation createOperation(String operate) {
            Operation op = null;
            switch (operate) {
            case "+":
```

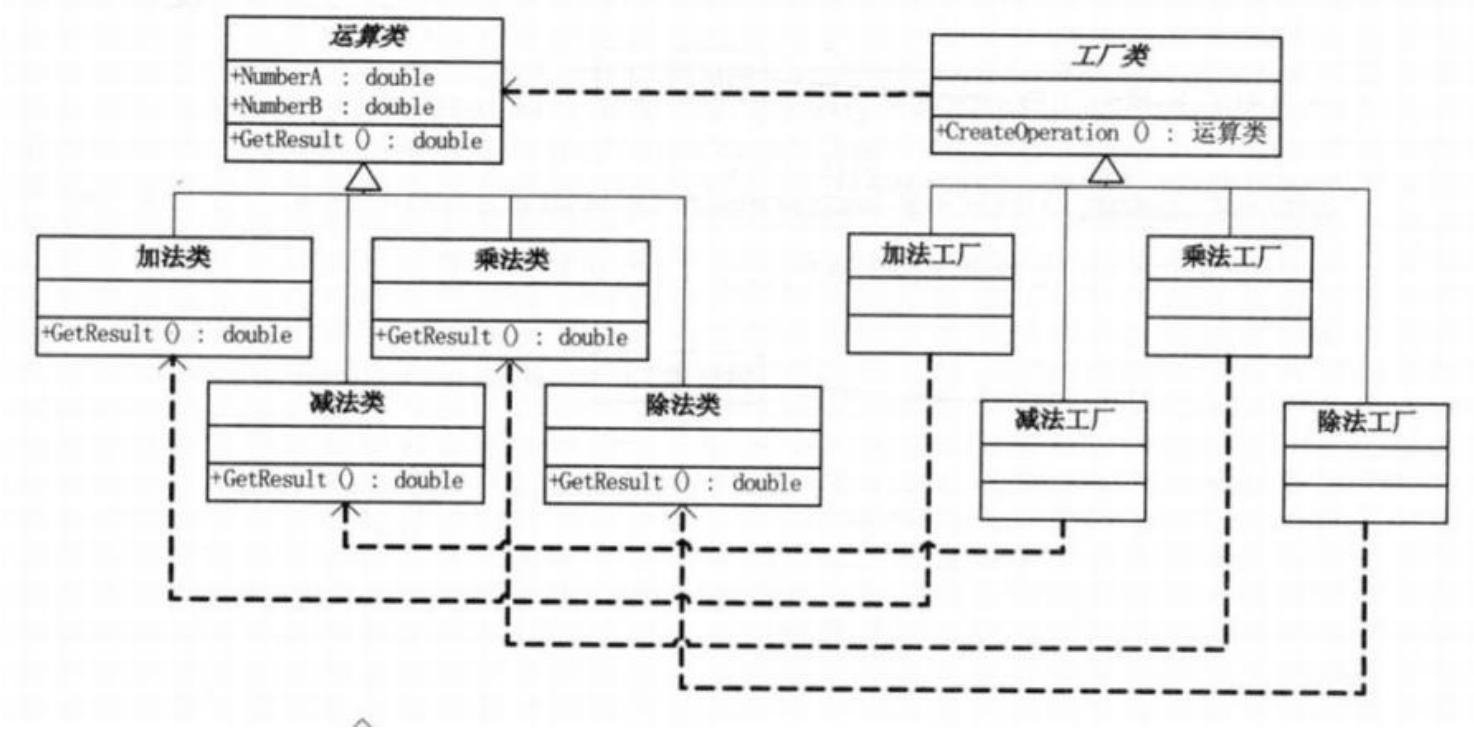
```
            op = new Add();
            break;
        case "-":
            op = new Minus();
            break;
        case "*":
            op = new Multiply();
            break;
        case "/":
            op = new Divide();
            break;
        }
        return op;
    }

}


public class Test {

    public static void main(String[] args) {
        Operation o = OperationFactory.createOperation("+");
        o.setNumberA(9.8);
        o.setNumberB(5.6);
        System.out.println(o.getResult());
    }

}
```

分析：利用了简单工厂模式实现了解耦，如果现在需要一个平方根的运算，只需要继承 Operation 这个类重写它的方法就可以了，而不需要去改动加减乘除的类。利于代码的维护和扩展。通过 OperationFactory 工厂根据运算的需要去创建对象。

2） 工厂方法模式

对于简单工厂模式我们已经实现了初步的解耦，加强了程序的可维护性和可扩展性，在运算类上已经做到了这一点，如果需要增加一个平方根的运算，我们可以去继承 Operation 这个类，但是在工厂类中我们还是需要增加一个 case 来判断，这样对于工厂类的维护和扩展就比较麻烦，需要去修改原有的代码，从而给维护和扩展带来了困难。

工厂方法模式的 UML 图：



实现的代码如下：

```
public abstract class Operation {

	private double numberA;
	private double numberB;

	public abstract double getResult();

	public void setNumberA(double numberA) {
		this.numberA = numberA;
	}

	public void setNumberB(double numberB) {
		this.numberB = numberB;
	}

	public double getNumberA() {
		return numberA;
	}

	public double getNumberB() {
		return numberB;
	}

}


public class Add extends Operation {
```

```
    @Override
    public double getResult() {
        return getNumberA() + getNumberB();
    }

}

public class Divide extends Operation {

    @Override
    public double getResult() {
        return getNumberA() / getNumberB();
    }

}

public class Minus extends Operation {

    @Override
    public double getResult() {
        return getNumberA() - getNumberB();
    }

}

public class Multiply extends Operation {

    @Override
    public double getResult() {
```

```
        return getNumberA() * getNumberB();
    }

}

public abstract class OperationFactory {

    public  abstract  Operation createOperation();

}


public class AddFactory extends OperationFactory{

    @Override
    public Operation createOperation() {
        return new Add();
    }

}


public class DivideFactory extends OperationFactory{

    @Override
    public Operation createOperation() {
        return new Divide();
    }

}
```

```java
public class MinusFactory extends OperationFactory{

	@Override
	public Operation createOperation() {
		return new Minus();
	}

}


public class MultiplyFactory extends OperationFactory {

	@Override
	public Operation createOperation() {
		return new Multiply();
	}

}


public class Test {

	public static void main(String[] args) {

		OperationFactory of = new AddFactory();

		Operation o = of.createOperation();

		o.setNumberA(10);
```

```
            o.setNumberB(20);
            System.out.println(o.getResult());

        }

}
```

分析：通过工厂方法模式，如果需要增加一个平方根的运算，只需要继承 Operation 和 OperationFactory 类重写它的方法，原来的代码结构并不会发生改变，而需要修改的知识客户端。解耦的同时提高了程序的可维护性和可扩展性。