

# 抽象类

为什么使用抽象类

- 1: 定义Dog类
  - 有颜色属性和叫的方法
- 2: 定义Bird类
  - 有颜色属性和叫的方法
- 3: 定义其父类Animal
  - 1: 抽取共性颜色属性和叫的方法
    - 1: 颜色的属性可以使用默认初始化值。
    - 2: 叫的方法在父类中如何定义?
      - 1: 狗是旺旺
      - 2: 鸟是叽叽喳喳
      - 3: 可以将父类的方法定义为狗叫让鸟继承父类重写叫的方法
        - 1: 鸟怎么确定是否要重写父类方法。
        - 2: 不重写, 编译和运行都没有问题, 只是执行鸟叫的方法就会出现狗叫
      - 4: 父类的方法很难确定。

```
class Animal {
    String color;

    void shout() {
        //如何定义呢?是旺旺还是叽叽喳喳?
    }
}

class Dog extends Animal {

    void shout() {
        System.out.println("旺旺");
    }
}

class Bird extends Animal {

    void shout() {
        System.out.println("叽叽喳喳");
    }
}
```

2: 使用abstract

- 4: 抽象类
  - 1: 当描述一个类的时候, 如果不能确定功能函数如何定义, 那么该类就可以定义为抽象类, 功能函数应该描述为抽象函数。
- 5: 抽象类的实现方式

1: 定义animal类

1: 定义叫的方法, 无法确定方法体, 不写方法体

1: `public void shout ();` 编译失败

2: 根据提示在shout的方法加入abstract修饰

1: 编译失败, 有新的提示

3: 根据提示将类加入abstract修饰

1: 编译通过

```
abstract class Animal {  
    String color;  
  
    abstract void shout();  
}  
  
class Dog extends Animal {  
  
    void shout() {  
        System.out.println("旺旺");  
    }  
}  
  
class Bird extends Animal {  
  
    void shout() {  
        System.out.println("叽叽喳喳");  
    }  
}
```

6: 抽象类的特点

1: 有抽象函数的类, 该类一定是抽象类。

2: 抽象类中不一定要有抽象函数。

3: 抽象类不能使用new创建对象

1: 创建对象, 使用对象的功能, 抽象类的方法, 没有方法体。

4: 抽象类主要为了提高代码的复用性, 让子类继承来使用。

5: 编译器强制子类实现抽象类父类的未实现的方法。

1: 可以不实现, 前提是子类的也要声明为抽象的。

7: 抽象的优点

1: 提高代码复用性

2: 强制子类实现父类中没有实现的功能

2: 提高代码的扩展性, 便于后期的代码维护

8: 抽象类不能创建对象, 那么抽象类中是否有构造函数?

1: 抽象类中一定有构造函数。主要为了初始化抽象类中的属性。通常由子类实现。

9: final和abstract是否可以同时修饰一个类?

一定不能同时修饰。

```

abstract class Animal {

    String name;

    // 抽象类可以有构造函数
    Animal() {

    }

    Animal(String name) {
        this.name = name;
    }

    abstract void shout();
}

class Dog extends Animal {
    Dog() {

    }

    Dog(String name) {
        super(name);
    }

    void shout() {
        System.out.println("旺旺");
    }
}

class Demo3 {

    public static void main(String[] args) {
        // 抽象类不能创建对象
        // Animal a=new Animal();
        Dog d = new Dog("旺财");

        System.out.println();
    }
}

```

## 2: 抽象练习

- 1: 定义抽象类MyShape (图形)
  - 1: 定义抽象方法获取图形的长度和面积
- 2: 定义子类Rect继承父类MyShape
  - 1: 定义自身特有的长和宽 (成员变量)     width height;
  - 2: 实现父类未实现的函数。
- 3: 定义子类 Circle实现父类MyShape
  - 1: 定义自身特有的半径和圆周率 (使用常量)
  - 2: 实现父类为的方法。

```

/*
}
2: 抽象练习
1: 定义抽象类MyShape (图形)
1: 定义抽象方法获取图形的长度和面积
2: 定义子类Rect继承父类MyShape
1: 定义自身特有的长和宽 (成员变量)    width height;
2: 实现父类未实现的函数。
3: 定义子类 Circle实现父类MyShape
1: 定义自身特有的半径和圆周率 (使用常量)
2: 实现父类为实现的方法。
*/
abstract class MyShape {

    abstract double getLen();

    abstract double getArea();

}

class Rect extends MyShape {
    double width;
    double height;

    Rect() {

    }

    Rect(double width, double height) {
        this.width = width;
        this.height = height;
    }

    double getLen() {
        return 2 * (width + height);
    }

    double getArea() {
        return width * height;
    }
}

class Circle extends MyShape {
    double r;
    public static final double PI = 3.14;

    Circle() {

    }

    Circle(double r) {
        this.r = r;
    }

    double getLen() {
        return 2 * PI * r;
    }
}

```

## 抽象类注意细节

抽象类可以没有抽象方法（java.awt.\*的类就是这样子操作的）。

抽象类可以继承普通类与抽象类。

抽象类不能直接使用类名创建实例，但是有构造方法，构造方法是让子类进行初始化。

抽象类一定有构造方法。

abstract与其他修饰符的关系：

**final**与abstract不能共存：

final:它的作用 修饰类代表不可以继承 修饰方法不可重写

abstract修饰类就是用来被继承的，修饰方法就是用来被重写的。

**static** static修饰的方法可以用类名调用，

对于abstract修饰的方法没有具体的方法实现，所有不能直接调用，  
也就是说不可以与static共存。

**private**

private修饰的只能在本类中使用，

abstract方法是用被子类进行重写的，有矛盾

所有不能共存。

练习：使用抽象类计算一个矩形与圆形的面积。

## 值交换

案例：定义交换数值的功能函数，基本类型数据, 数组,实例对象, String。

基本数据类型交换

结果：发现交换值前后没有变量的值发生变化。

原因分析：

数组类型交换

结果：交换值成功。

原因分析：操作的是同一个数组对象。

对象的值交换：

结果：交换值成功。

字符串的值交换：

交换值失败。

# 接口

## 接口的概述

接口(interface): usb接口, 主要是使用来拓展笔记本的功能, 那么在java中的接口主要是使用来拓展定义类的功能, 可以弥补java中单继承的缺点。

```
class Pencil {
    String name;
    Pencil() {
    }
    Pencil(String name) {
        this.name = name;
    }
    void write() {
        System.out.println("写字");
    }
}

interface Eraser {
    public static final String color = "白色";
    public abstract void clean();
}

// 1: 带橡皮的铅笔类继承铅笔类实现橡皮接口
class PencilWithEraser extends Pencil implements Eraser {
    PencilWithEraser() {
    }
    PencilWithEraser(String name) {
        super(name);
    }
    void write() {
        System.out.println(name + ":考试专用");
    }
    public void clean() {
        System.out.println(super.name + ":带橡皮的铅笔, 就是好用");
    }
}

class Demo6 {
    public static void main(String[] args) {
        PencilWithEraser pe = new PencilWithEraser("中华2B");
        pe.write();
        pe.clean();
        System.out.println(pe.color);
        System.out.println(PencilWithEraser.color);
    }
}
```

接口的定义格式:

```
interface 接口名{  
    属性  
    抽象方法  
}
```

接口的体验

```
interface Inter  
{  
    int num = 6; 可以定义属性与方法。  
    void show();  
}
```

注意: 可以通过javap命令查看.

接口中的所有属性 默认的修饰符是 public static final。

接口中的所有方法 默认的修饰符是 public abstract。

疑惑: 干嘛不在PencilWithEraser添加remove功能函数, 而要通过接口?

## 接口的特点

类实现接口可以通过implements实现, 实现接口的时候必须把接口中的所有方法实现, 一个类可以实现多个接口。

接口中定义的所有的属性默认是public static final的, 即静态常量既然是常量, 那么定义的时候必须赋值。

接口中定义的方法不能有方法体。接口中定义的方法默认添加public abstract

有抽象函数的不一定是抽象类, 也可以是接口类。

由于接口中的方法默认都是抽象的, 所以不能被实例化。

对于接口而言, 可以使用子类来实现接口中未被实现的功能函数。

如果实现类中要访问接口中的成员, 不能使用super关键字。因为两者之间没有显示的继承关系, 况且接口中的成员成员属性是静态的。可以使用接口名直接访问。

接口没有构造方法。

## 接口与类、接口之间的关系

1. 大家之前都知道类与类之间的关系继承, 那么接口与类之间又是怎样子的关系呢? 接口与类之间是实现关系。非抽象类实现接口时, 必须把接口里面的所有方法实现。类实现接口用关键字implements, 类与接口之间是可以多实现的(即一个类可以实现多个接口)。

```

interface Eraser {
    public static final String color = "白色";
    public abstract void clean();
}
class Pencil implements Eraser {
    String name;
    Pencil() {
    }
    Pencil(String name) {
        this.name = name;
    }
    void write() {
        System.out.println("写字");
    }
    @Override
    public void clean() {
        System.out.println("涂改...");
    }
}

```

分析：

原本铅笔没有涂改功能的，但是一旦实现了Eraser接口做了实现，那么就具备了涂改功能，那么接口的作用则是拓展功能。

## 2. 接口与接口之间的关系式继承。

```

interface A{
    public void show();
}
interface B{
    public void print();
}
interface C extends A,B{
}

```

接口与接口之间的关系是继承，接口可以多继承接口。

练习：在现实生活中有部分同学在学校期间只会学习，但是有部分学生除了学习外还会赚钱。

# 多态

## 多态的概述

### 1: 什么是多态

一个对象的多种状态

（老师）（员工）（儿子）

教师 a =老钟；

员工 b= 老钟；

### 2: 多态体现



- 1: Father类
  - 1: 非静态成员变量x
  - 2: 静态成员变量y
  - 3: 非静态方法eat, 方法体输出父类信息
  - 4: 静态方法speak();方法体输出父类信息
- 2: Son类
  - 1: 非静态成员变量x
  - 2: 静态成员变量y
  - 3: 非静态方法eat, 方法体输出子类信息
  - 4: 静态方法speak();方法体输出子类信息

```
class Father {
    int x = 1;
    static int y = 2;

    void eat() {
        System.out.println("开吃");
    }

    static void speak() {
        System.out.println("小头爸爸");
    }
}

class Son extends Father {
    int x = 3;
    static int y = 4;

    void eat() {
        System.out.println("大头儿子很能吃");
    }

    static void speak() {
        System.out.println("大头儿子。");
    }
}

class Demo10 {

    public static void main(String[] args) {

        Father f = new Son(); // 父类引用指向了子类对象。
        System.out.println(f.x); // 1
        System.out.println(f.y); // 2

        f.eat(); // 输出的是子类的。

        f.speak().; // 输出的是父类

    }
}
```

- 3: Son类继承父类

1: 创建Father f=new Son();

1: 这就是父类引用指向了子类对象。

2: 问f.x=? (非静态)

3: 问f.y=? (静态)

4: 问f.eat() 输出的是子类还是父类信息? (非静态)

5: 问f.speak() 输出的是子类还是父类信息? (静态)

#### 4: 总结

1: 当父类和子类具有相同的非静态成员变量, 那么多态下访问的是父类的成员变量

2: 当父类和子类具有相同的静态成员变量, 那么多态下访问的是父类的静态成员变量

所以: 父类和子类有相同的成员变量, 多态下访问的是父类的成员变量。

3: 当父类和子类具有相同的非静态方法 (就是子类重写父类方法), 多态下访问的是子类的成员方法。

4: 当父类和子类具有相同的静态方法 (就是子类重写父类静态方法), 多态下访问的是父类的静态方法。

#### 2: 多态体现

1: 父类引用变量指向了子类的对象

2: 父类引用也可以接受自己的子类对象

#### 3: 多态前提

1: 类与类之间有关系, 继承或者实现

#### 4: 多态弊端

1: 提高扩展性, 但是只能使用父类引用指向父类成员。

#### 5: 多态特点

非静态

1: 编译时期, 参考引用型变量所属的类是否有调用的方法, 如果有编译通过。没有编译失败

2: 运行时期, 参考对象所属类中是否有调用的方法。

3: 总之成员函数在多态调用时, 编译看左边, 运行看右边。

在多态中, 成员变量的特点, 无论编译和运行参考左边 (引用型变量所属的类)。

在多态中, 静态成员函数特点, 无论编译和运行都参考左边

#### 6: 多态练习

1: 多态可以作为形参, 接受范围更广的对象, 避免函数重载过度使用。

1: 定义功能, 根据输出任何图形的面积和周长。

1: 定义抽象类abstract MyShape

1: 定义抽象方法public abstract double getArea();

2: 定义抽象方法public abstract double getLen();

2: 定义Rect类继承MyShape

1: 定义长和宽成员变量, double width height;

2: 无参构造, 有参构造。

3: 实现父类方法。

3: 定义Cricle类继承MyShape

- 1: 定义半径成员变量, 和PI常量
- 2: 无参构造, 有参构造
- 3: 实现父类方法。
- 4: 定义静态方法计算任意图形的面积和周长
  - 1: 未知内容参与运算, 不能确定用户传入何种图形, 使用多态。
    - 1: 形参定义为 `MyShape my`
  - 2: 调用计算面积方法, 和计算周长方法。并打印
    - 2: 使用多态特性, 子类重写了父类非静态方法, 会执行子类的方法。

/\*

## 多态练习

1: 多态可以作为形参, 接受范围更广的对象, 避免函数重载过度使用。

1: 定义功能, 根据输出任何图形的面积和周长。

子类重写了父类的抽象方法, 多态下, 会执行子类的非静态方法。

2: 多态可以作为返回值类型。

获取任意一辆车对象

3: 抽象类和接口都可以作为多态中的父类引用类型。

\*/

```
abstract class MyShape{
    public abstract double getArea();
    public abstract double getLen();
}

class Rect extends MyShape{
    double width;
    double height;
    Rect(){

    }
    Rect(double width ,double height){
        this.width=width;
        this.height=height;
    }
    public double getArea(){
        return width*height;
    }
    public double getLen(){
        return 2*(width+height);
    }
}

class Circle extends MyShape{
    double r;
    public static final double PI=3.14;

    Circle(){

    }

    Circle(double r){
        this.r=r;
    }
    public double getLen(){
        return 2*PI*r;
    }

    public double getArea(){
        return PI*r*r;
    }
}

class Demo11{

    public static void main(String[] args){

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

2: 多态可以作为返回值类型。

获取任意一辆车对象

- 1: 定义汽车类, 有名字和颜色, 提供有参和无参构造, 有运行的行为。
- 2: 定义Bmw类, 继承Car类, 提供无参构造和有参构造 (super父类构造), 重写父类运行行为。
- 3: 定义Benz类, 继承Car类, 提供无参构造和有参构造 (super父类构造), 重写父类运行行为。
- 4: 定义Bsj类, 继承Car类, 提供无参构造和有参构造 (super父类构造), 重写父类运行行为。
- 5: 定义静态方法, 汽车工厂, 随机生产汽车。使用多态定义方法返回值类型。
  - 1: 使用 `(int) Math.round(Math.random()*2)`; 生成0-2之间随机

数。

- 2: 使用 `if else` 判断, 指定, 0,1,2 new 不同汽车 并返回。
- 6: 调用该方法, 发现多态的好处。

\*

2: 多态可以作为返回值类型。

获取任意一辆车对象

1: 定义汽车类, 有名字和颜色, 提供有参和无参构造, 有运行的行为。

2: 定义Bmw类, 继承Car类, 提供无参构造和有参构造 (super父类构造), 重写父类运行行为。

3: 定义Benz类, 继承Car类, 提供无参构造和有参构造 (super父类构造), 重写父类运行行为。

4: 定义BsJ类, 继承Car类, 提供无参构造和有参构造 (super父类构造), 重写父类运行行为。

5: 定义静态方法, 汽车工厂, 随机生产汽车。使用多态定义方法返回值类型。

1: 使用 (int) Math.round(Math.random()\*2); 生成0-2之间随机数。

Math 类

2: 使用if else 判断, 指定, 0,1,2 new 不同汽车 并返回。

6: 调用该方法, 发现多态的好处。

\*/

```
class Car {
    String name;
    String color;

    Car() {

    }

    Car(String name, String color) {
        this.name = name;
        this.color = color;
    }

    void run() {
        System.out.println("跑跑。。。");
    }
}

class Bmw extends Car {
    Bmw() {

    }

    Bmw(String name, String color) {
        super(name, color);
    }

    void run() {
        System.out.println("宝马很拉风。。。");
    }
}

class Benz extends Car {
    Benz() {

    }

    Benz(String name, String color) {
```

3: 抽象类和接口都可以作为多态中的父类引用类型。

1: sun Arrays

6: 多态之类型转型

1: 案例定义Father类

1: 定义method1和method2方法

2: 定义Son类继承Father类

1: 定义method1 (重写父类method1) 和method2方法

3: 创建Father f=new Son();

1: f.method1() 调用的子类或者父类?

2: f.method2() 编译和运行是否通过?

3: f.method3() 编译和运行是否通过? (编译报错)

4: 如何在多态下, 使用父类引用调用子类特有方法。

1: 基本类型转换:

1: 自动: 小->大

2: 强制: 大->小

2: 类类型转换

前提: 继承, 必须有关系

1: 自动: 子类转父类

2: 强转: 父类转子类

3: 类型转换

1: Son s=(Son) f

2: s.method3();

```

/*
如何在多态下，使用父类引用调用子类特有方法。
1: 基本类型转换:
1: 自动: 小->大    int x=1 double d=x;
2: 强制: 大->小    int y=(int)d;
2: 类类型转换
前提: 继承, 必须有关系
1: 自动: 子类转父类  Father f=new Son();
2: 强转: 父类转子类  Son s=(Son)f;
1: 类型转换
1: Son s=(Son)f
2: s.method3();
*/
class Father {

    void method1() {
        System.out.println("这是父类1");
    }

    void method2() {
        System.out.println("这是父类2");
    }
}

class Son extends Father {
    void method1() {
        System.out.println("这是子类1");
    }

    void method3() {
        System.out.println("这是子类3");
    }
}

class Demo14 {

    public static void main(String[] args) {
        Father f = new Son();
        f.method1(); // 这是子类1
        f.method2(); // 这是父类2

        // f.method3(); //编译报错。
        // 多态弊端，只能使用父类引用指向父类成员。

        // 类类型转换
        Son s = (Son) f;
        s.method3();

        System.out.println();
    }
}

```



5: 案例:

- 1: 定义Animal类颜色成员变量, 无参构造, 有参构造, run方法
- 2: 定义Dog类, 继承Animal, 定义无参构造, 有参构造 (使用super调用父类有参构造), Dog的特有方法ProtectHome
- 3: 定义Fish类, 继承Animal, 定义无参构造, 有参构造 (使用super调用父类有参构造), Fish特有方法swim
- 4: 定义Bird类, 继承Animal, 定义无参构造, 有参构造 (使用super调用父类有参构造), Bird特有方法fly

5: 使用多态, `Animal a=new Dog();`

6: 调用Dog的特有方法, `ProtectHome`

1: 类类型转换, `Dog d=(Dog)a;`

2: `d.protectHome`

7: 非多态

1: `Animal a=new Animal();`

2: 类类型转换

`Dog d=(Dog)a;`

`d.protectHome();`

3: 编译通过, 运行出现异常

1: `ClassCastException`

8: 多态例外

1: `Animal a=new Dog();`

2: 类类型转换

1: `Fish f=(Fish)a;`

2: `f.fish();`

3: 编译通过, 运行异常

1: `ClassCastException`

4: 虽然是多态, 但是鸟不能转为狗, 狗不能转为鱼, 他们之间没有关系。

```

class Animal {
    String color;

    Animal() {

    }

    Animal(String color) {
        this.color = color;
    }

    void run() {
        System.out.println("跑跑");
    }
}

class Dog extends Animal {
    Dog() {

    }

    Dog(String color) {
        super(color);
    }

    void run() {
        System.out.println("狗儿跑跑");
    }

    void protectHome() {
        System.out.println("旺旺, 看家");
    }
}

class Fish extends Animal {
    Fish() {

    }

    Fish(String color) {
        super(color);
    }

    void run() {
        System.out.println("鱼儿水中游");
    }

    void swim() {
        System.out.println("鱼儿游泳");
    }
}

class Demo15 {

    public static void main(String[] args) {

```

6: 案例2

- 1: 定义一功能, 接收用户传入动物, 根据用于传入的具体动物, 执行该动物特有的方法
- 2: 使用多态, 方法形参, 不能确定用户传入的是那种动物
- 3: 使用instanceof 关键字, 判断具体是何种动物,
- 4: 类转换, 执行该动物的特有方法。

```
package oop04;
```

```
/*
```

案例2

1: 定义一功能, 接收用户传入动物, 根据用于传入的具体动物, 执行该动物特有的方法

2: 使用多态, 方法形参, 不能确定用户传入的是那种动物

3: 使用instanceof 关键字, 判断具体是何种动物,

4: 类转换, 执行该动物的特有方法。

```
*/
```

```
class Animal {  
    String color;  
  
    Animal() {  
  
    }  
  
    Animal(String color) {  
        this.color = color;  
    }  
  
    void run() {  
        System.out.println("跑跑");  
    }  
}
```

```
class Dog extends Animal {  
    Dog() {  
  
    }  
  
    Dog(String color) {  
        super(color);  
    }  
  
    void run() {  
        System.out.println("狗儿跑跑");  
    }  
  
    void protectHome() {  
        System.out.println("旺旺, 看家");  
    }  
}
```

```
class Fish extends Animal {  
    Fish() {  
  
    }  
  
    Fish(String color) {  
        super(color);  
    }  
  
    void run() {  
        System.out.println("鱼儿水中游");  
    }  
}
```

# 作业

抽象类的特点，以及细节？

接口的表现形式的特点。

接口的思想特点，要举例。

多实现和多继承的区别？

抽象类和接口的区别？

多态的体现，前提，好处，弊端。