# 第11天面向对象

今日内容介绍

接口

多态

# 接口

## 接口概念

接口是功能的集合，同样可看做是一种数据类型，是比抽象类更为抽象的”类”。

接口只描述所应该具备的方法，并没有具体实现，具体的实现由接口的实现类(相当于接口的子类)来完成。这样将功能的定义与实现分离，优化了程序设计。

请记住：一切事物均有功能，即一切事物均有接口。

## 接口的定义

与定义类的class不同，接口定义时需要使用interface关键字。

定义接口所在的仍为.java文件，虽然声明时使用的为interface关键字的编译后仍然会产生.class文件。这点可以让我们将接口看做是一种只包含了功能声明的特殊类。

定义格式：

public interface 接口名 {

抽象方法1;

抽象方法2;

抽象方法3;

}

使用interface代替了原来的class，其他步骤与定义类相同：

接口中的方法均为公共访问的抽象方法

接口中无法定义普通的成员变量

## 类实现接口

类与接口的关系为实现关系，即类实现接口。实现的动作类似继承，只是关键字不同，实现使用implements。

其他类(实现类)实现接口后，就相当于声明：”我应该具备这个接口中的功能”。实现类仍然需要重写方法以实现具体的功能。

格式：

class 类 implements 接口 {

重写接口中方法

}

在类实现接口后，该类就会将接口中的抽象方法继承过来，此时该类需要重写该抽象方法，完成具体的逻辑。

接口中定义功能，当需要具有该功能时，可以让类实现该接口，只声明了应该具备该方法，是功能的声明。

在具体实现类中重写方法，实现功能，是方法的具体实现。

于是，通过以上两个动作将功能的声明与实现便分开了。(此时请重新思考：类是现实事物的描述，接口是功能的集合。)

## 接口中成员的特点

1、接口中可以定义变量，但是变量必须有固定的修饰符修饰，public static final 所以接口中的变量也称之为常量，其值不能改变。后面我们会讲解static与final关键字

2、接口中可以定义方法，方法也有固定的修饰符，public abstract

3、接口不可以创建对象。

4、子类必须覆盖掉接口中所有的抽象方法后，子类才可以实例化。否则子类是一个抽象类。

interface Demo { ///定义一个名称为Demo的接口。

public static final int NUM = 3;// NUM的值不能改变

public abstract void show1();

public abstract void show2();

}

//定义子类去覆盖接口中的方法。类与接口之间的关系是 实现。通过 关键字 implements

class DemoImpl implements Demo { //子类实现Demo接口。

//重写接口中的方法。

public void show1(){}

public void show2(){}

}

## 接口的多实现

了解了接口的特点后，那么想想为什么要定义接口，使用抽象类描述也没有问题，接口到底有啥用呢？

接口最重要的体现：解决多继承的弊端。将多继承这种机制在java中通过多实现完成了。

interface Fu1

{

void show1();

}

interface Fu2

{

void show2();

}

class Zi implements Fu1,Fu2// 多实现。同时实现多个接口。

{

public void show1(){}

public void show2(){}

}

怎么解决多继承的弊端呢？

弊端：多继承时，当多个父类中有相同功能时，子类调用会产生不确定性。

其实核心原因就是在于多继承父类中功能有主体，而导致调用运行时，不确定运行哪个主体内容。

为什么多实现能解决了呢？

因为接口中的功能都没有方法体，由子类来明确。

## 类继承类同时实现接口

接口和类之间可以通过实现产生关系，同时也学习了类与类之间可以通过继承产生关系。当一个类已经继承了一个父类，它又需要扩展额外的功能，这时接口就派上用场了。

子类通过继承父类扩展功能，通过继承扩展的功能都是子类应该具备的基础功能。如果子类想要继续扩展其他类中的功能呢？这时通过实现接口来完成。

class Fu {

public void show(){}

}

interface Inter {

pulbic abstract void show1();

}

class Zi extends Fu implements Inter {

public void show1() {

}

}

接口的出现避免了单继承的局限性。父类中定义的事物的基本功能。接口中定义的事物的扩展功能。

## 接口的多继承

学习类的时候，知道类与类之间可以通过继承产生关系，接口和类之间可以通过实现产生关系，那么接口与接口之间会有什么关系。

多个接口之间可以使用extends进行继承。

interface Fu1{

void show();

}

interface Fu2{

void show1();

}

interface Fu3{

void show2();

}

interface Zi extends Fu1,Fu2,Fu3{

void show3();

}

在开发中如果多个接口中存在相同方法，这时若有个类实现了这些接口，那么就要实现接口中的方法，由于接口中的方法是抽象方法，子类实现后也不会发生调用的不确定性。

## 接口的思想

前面学习了接口的代码体现，现在来学习接口的思想，接下里从生活中的例子进行说明。

**举例：**我们都知道电脑上留有很多个插口，而这些插口可以插入相应的设备，这些设备为什么能插在上面呢？主要原因是这些设备在生产的时候符合了这个插口的使用规则，否则将无法插入接口中，更无法使用。发现这个插口的出现让我们使用更多的设备。

**总结：**接口在开发中的它好处

1. 接口的出现扩展了功能。
2. 接口其实就是暴漏出来的规则。
3. 接口的出现降低了耦合性，即设备与设备之间实现了解耦。

接口的出现方便后期使用和维护，一方是在使用接口（如电脑），一方在实现接口（插在插口上的设备）。例如：笔记本使用这个规则（接口），电脑外围设备实现这个规则（接口）。

## 接口和抽象的区别

明白了接口思想和接口的用法后，接口和抽象类的区别是什么呢？接口在生活体现也基本掌握，那在程序中接口是如何体现的呢？

通过实例进行分析和代码演示抽象类和接口的用法。

1、举例：

犬：

行为：

吼叫；

吃饭；

缉毒犬：

行为：

吼叫；

吃饭；

缉毒；

2、思考：

由于犬分为很多种类，他们吼叫和吃饭的方式不一样，在描述的时候不能具体化，也就是吼叫和吃饭的行为不能明确。当描述行为时，行为的具体动作不能明确，这时，可以将这个行为写为抽象行为，那么这个类也就是抽象类。

可是当缉毒犬有其他额外功能时，而这个功能并不在这个事物的体系中。这时可以让缉毒犬具备犬科自身特点的同时也有其他额外功能，可以将这个额外功能定义接口中。

如下代码演示：

interface 缉毒{

public abstract void 缉毒();

}

//定义犬科的这个提醒的共性功能

abstract class 犬科{

public abstract void 吃饭();

public abstract void 吼叫();

}

// 缉毒犬属于犬科一种，让其继承犬科，获取的犬科的特性，

//由于缉毒犬具有缉毒功能，那么它只要实现缉毒接口即可，这样即保证缉毒犬具备犬科的特性，也拥有了缉毒的功能

class 缉毒犬 extends 犬科 implements 缉毒{

public void 缉毒() {

}

void 吃饭() {

}

void 吼叫() {

}

}

class 缉毒猪 implements 缉毒{

public void 缉毒() {

}

}

3、通过上面的例子总结接口和抽象类的区别：

**相同点:**

都位于继承的顶端,用于被其他类实现或继承;

都不能直接实例化对象;

都包含抽象方法,其子类都必须覆写这些抽象方法;

**区别:**

抽象类为部分方法提供实现,避免子类重复实现这些方法,提高代码重用性;接口只能包含抽象方法;

一个类只能继承一个直接父类(可能是抽象类),却可以实现多个接口;(接口弥补了Java的单继承)

抽象类是这个事物中应该具备的你内容, 继承体系是一种 is..a关系

接口是这个事物中的额外内容,继承体系是一种 like..a关系

二者的选用:

优先选用接口,尽量少用抽象类;

需要定义子类的行为,又要为子类提供共性功能时才选用抽象类;

# 多态

## 多态概述

多态是继封装、继承之后，面向对象的第三大特性。

现实事物经常会体现出多种形态，如学生，学生是人的一种，则一个具体的同学张三既是学生也是人，即出现两种形态。

Java作为面向对象的语言，同样可以描述一个事物的多种形态。如Student类继承了Person类，一个Student的对象便既是Student，又是Person。

Java中多态的代码体现在一个子类对象(实现类对象)既可以给这个子类(实现类对象)引用变量赋值，又可以给这个子类(实现类对象)的父类(接口)变量赋值。

如Student类可以为Person类的子类。那么一个Student对象既可以赋值给一个Student类型的引用，也可以赋值给一个Person类型的引用。

最终多态体现为父类引用变量可以指向子类对象。

多态的前提是必须有子父类关系或者类实现接口关系，否则无法完成多态。

在使用多态后的父类引用变量调用方法时，会调用子类重写后的方法。

## 多态的定义与使用格式

多态的定义格式：就是父类的引用变量指向子类对象

父类类型 变量名 = new 子类类型();

变量名.方法名();

普通类多态定义的格式

父类 变量名 = new 子类();

如： class Fu {}

class Zi extends Fu {}

//类的多态使用

Fu f = new Zi();

抽象类多态定义的格式

抽象类 变量名 = new 抽象类子类();

如： abstract class Fu {

public abstract void method();

}

class Zi extends Fu {

public void method(){

System.out.println(“重写父类抽象方法”);

}

}

//类的多态使用

Fu fu= new Zi();

接口多态定义的格式

接口 变量名 = new 接口实现类();

如： interface Fu {

public abstract void method();

}

class Zi implements Fu {

public void method(){

System.out.println(“重写接口抽象方法”);

}

}

//接口的多态使用

Fu fu = new Zi();

注意事项

同一个父类的方法会被不同的子类重写。在调用方法时，调用的为各个子类重写后的方法。

如 Person p1 = new Student();

Person p2 = new Teacher();

p1.work(); //p1会调用Student类中重写的work方法

p2.work(); //p2会调用Teacher类中重写的work方法

当变量名指向不同的子类对象时，由于每个子类重写父类方法的内容不同，所以会调用不同的方法。

## 多态-成员的特点

掌握了多态的基本使用后，那么多态出现后类的成员有啥变化呢？前面学习继承时，我们知道子父类之间成员变量有了自己的特定变化，那么当多态出现后，成员变量在使用上有没有变化呢？

多态出现后会导致子父类中的成员变量有微弱的变化。看如下代码

class Fu {

int num = 4;

}

class Zi extends Fu {

int num = 5;

}

class Demo {

public static void main(String[] args) {

Fu f = new Zi();

System.out.println(f.num);

Zi z = new Zi();

System.out.println(z.num);

}

}

多态成员变量

当子父类中出现同名的成员变量时，多态调用该变量时：

编译时期：参考的是引用型变量所属的类中是否有被调用的成员变量。没有，编译失败。

运行时期：也是调用引用型变量所属的类中的成员变量。

简单记：编译和运行都参考等号的左边。编译运行看左边。

多态出现后会导致子父类中的成员方法有微弱的变化。看如下代码

class Fu {

int num = 4;

void show() {

System.out.println("Fu show num");

}

}

class Zi extends Fu {

int num = 5;

void show() {

System.out.println("Zi show num");

}

}

class Demo {

public static void main(String[] args) {

Fu f = new Zi();

f.show();

}

}

多态成员方法

编译时期：参考引用变量所属的类，如果没有类中没有调用的方法，编译失败。

运行时期：参考引用变量所指的对象所属的类，并运行对象所属类中的成员方法。

简而言之：编译看左边，运行看右边。

## instanceof关键字

我们可以通过instanceof关键字来判断某个对象是否属于某种数据类型。如学生的对象属于学生类，学生的对象也属于人类。

使用格式：

boolean b = 对象 instanceof 数据类型;

如

Person p1 = new Student(); // 前提条件，学生类已经继承了人类

boolean flag = p1 instanceof Student; //flag结果为true

boolean flag2 = p2 instanceof Teacher; //flag结果为false

## 多态-转型

多态的转型分为向上转型与向下转型两种：

向上转型：当有子类对象赋值给一个父类引用时，便是向上转型，多态本身就是向上转型的过程。

使用格式：

父类类型 变量名 = new 子类类型();

如：Person p = new Student();

向下转型：一个已经向上转型的子类对象可以使用强制类型转换的格式，将父类引用转为子类引用，这个过程是向下转型。如果是直接创建父类对象，是无法向下转型的！

使用格式：

子类类型 变量名 = (子类类型) 父类类型的变量;

如:Student stu = (Student) p; //变量p 实际上指向Student对象

## 多态的好处与弊端

当父类的引用指向子类对象时，就发生了向上转型，即把子类类型对象转成了父类类型。向上转型的好处是隐藏了子类类型，提高了代码的扩展性。

但向上转型也有弊端，只能使用父类共性的内容，而无法使用子类特有功能，功能有限制。看如下代码

//描述动物类，并抽取共性eat方法

**abstract** **class** Animal {

**abstract** **void** eat();

}

// 描述狗类，继承动物类，重写eat方法，增加lookHome方法

**class** Dog **extends** Animal {

**void** eat() {

System.***out***.println("啃骨头");

}

**void** lookHome() {

System.***out***.println("看家");

}

}

// 描述猫类，继承动物类，重写eat方法，增加catchMouse方法

**class** Cat **extends** Animal {

**void** eat() {

System.***out***.println("吃鱼");

}

**void** catchMouse() {

System.***out***.println("抓老鼠");

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Animal a = **new** Dog(); //多态形式，创建一个狗对象

a.eat(); // 调用对象中的方法，会执行狗类中的eat方法

// a.lookHome();//使用Dog类特有的方法，需要向下转型，不能直接使用

// 为了使用狗类的lookHome方法，需要向下转型

// 向下转型过程中，可能会发生类型转换的错误，即ClassCastException异常

// 那么，在转之前需要做健壮性判断

if( !a instanceof Dog){ // 判断当前对象是否是Dog类型

System.out.println("类型不匹配，不能转换");

return;

}

Dog d = (Dog) a; //向下转型

d.lookHome();//调用狗类的lookHome方法

}

}

我们来总结一下：

什么时候使用向上转型：

当不需要面对子类类型时，通过提高扩展性，或者使用父类的功能就能完成相应的操作，这时就可以使用向上转型。

如：Animal a = **new** Dog();

a.eat();

什么时候使用向下转型

当要使用子类特有功能时，就需要使用向下转型。

如：Dog d = (Dog) a; //向下转型

d.lookHome();//调用狗类的lookHome方法

向下转型的好处：可以使用子类特有功能。

弊端是：需要面对具体的子类对象；在向下转型时容易发生ClassCastException类型转换异常。在转换之前必须做类型判断。

如：if( !a instanceof Dog){…}

## 多态-举例

我们明确多态使用，以及多态的细节问题后，接下来练习下多态的应用。

毕老师和毕姥爷的故事

/\*

描述毕老师和毕姥爷，

毕老师拥有讲课和看电影功能

毕姥爷拥有讲课和钓鱼功能

\*/

**class** 毕姥爷 {

**void** 讲课() {

System.***out***.println("政治");

}

**void** 钓鱼() {

System.***out***.println("钓鱼");

}

}

// 毕老师继承了毕姥爷，就有拥有了毕姥爷的讲课和钓鱼的功能，

// 但毕老师和毕姥爷的讲课内容不一样，因此毕老师要覆盖毕姥爷的讲课功能

class 毕老师 extends 毕姥爷 {

**void** 讲课() {

System.***out***.println("Java");

}

**void** 看电影() {

System.***out***.println("看电影");

}

}

public class Test {

**public** **static** **void** main(String[] args) {

// 多态形式

毕姥爷 a = **new** 毕老师(); // 向上转型

a.讲课(); // 这里表象是毕姥爷，其实真正讲课的仍然是毕老师，因此调用的也是毕老师的讲课功能

a.钓鱼(); // 这里表象是毕姥爷，但对象其实是毕老师，而毕老师继承了毕姥爷，即毕老师也具有钓鱼功能

// 当要调用毕老师特有的看电影功能时，就必须进行类型转换

毕老师 b = (毕老师) a; // 向下转型

b.看电影();

}

}

学习到这里，面向对象的三大特征学习完了。

总结下封装、继承、多态的作用：

封装：把对象的属性与方法的实现细节隐藏，仅对外提供一些公共的访问方式

继承：子类会自动拥有父类所有可继承的属性和方法。

多态：配合继承与方法重写提高了代码的复用性与扩展性；如果没有方法重写，则多态同样没有意义。

# 笔记本电脑案例

## 案例介绍

定义USB接口（具备开启功能、关闭功能），笔记本要使用USB设备，即笔记本在生产时需要预留可以插入USB设备的USB接口，即就是笔记本具备使用USB设备的功能，但具体是什么USB设备，笔记本并不关心，只要符合USB规格的设备都可以。鼠标和键盘要想能在电脑上使用，那么鼠标和键盘也必须遵守USB规范，不然鼠标和键盘的生产出来无法使用

进行描述笔记本类，实现笔记本使用USB鼠标、USB键盘

USB接口，包含开启功能、关闭功能

笔记本类，包含运行功能、关机功能、使用USB设备功能

鼠标类，要符合USB接口

键盘类，要符合USB接口

## 案例需求分析

阶段一：

使用笔记本，笔记本有运行功能，需要笔记本对象来运行这个功能

阶段二：

想使用一个鼠标，又有一个功能使用鼠标，并多了一个鼠标对象。

阶段三：

还想使用一个键盘 ，又要多一个功能和一个对象

问题：每多一个功能就需要在笔记本对象中定义一个方法，不爽，程序扩展性极差。

降低鼠标、键盘等外围设备和笔记本电脑的耦合性。

## 实现代码步骤

定义鼠标、键盘，笔记本三者之间应该遵守的规则

interface USB {

void open();// 开启功能

void close();// 关闭功能

}

鼠标实现USB规则

class Mouse implements USB {

public void open() {

System.*out*.println("鼠标开启");

}

public void close() {

System.*out*.println("鼠标关闭");

}

}

键盘实现USB规则

class KeyBoard implements USB {

public void open() {

System.*out*.println("键盘开启");

}

public void close() {

System.*out*.println("键盘关闭");

}

}

定义笔记本

class NoteBook {

// 笔记本开启运行功能

public void run() {

System.*out*.println("笔记本运行");

}

// 笔记本使用usb设备，这时当笔记本对象调用这个功能时，必须给其传递一个符合USB规则的USB设备

public void useUSB(USB usb) {

// 判断是否有USB设备

if (usb != null) {

usb.open();

usb.close();

}

}

public void shutDown() {

System.*out*.println("笔记本关闭");

}

}

public class Test {

public static void main(String[] args) {

// 创建笔记本实体对象

NoteBook nb = new NoteBook();

// 笔记本开启

nb.run();

// 创建鼠标实体对象

Mouse m = new Mouse();

// 笔记本使用鼠标

nb.useUSB(m);

// 创建键盘实体对象

KeyBoard kb = new KeyBoard();

// 笔记本使用键盘

nb.useUSB(kb);

// 笔记本关闭

nb.shutDown();

}

}

# 总结

## 知识点总结

接口：理解为是一个特殊的抽象类，但它不是类，是一个接口

* + 接口的特点：

1，定义一个接口用interface关键字

interface Inter{}

2，一个类实现一个接口，实现implements关键字

class Demo implements Inter{}

3, 接口不能直接创建对象

通过多态的方式，由子类来创建对象，接口多态

* + 接口中的成员特点：

成员变量：

只能是final 修饰的常量

默认修饰符： public static final

构造方法：

无

成员方法：

只能是抽象方法

默认修饰符: public abstract

* + 类与类，类与接口，接口与接口之间的关系

类与类之间：继承关系，单继承，可以是多层继承

类与接口之间: 实现关系，单实现，也可以多实现

接口与接口之间：继承关系，单继承，也可以是多继承

Java中的类可以继承一个父类的同时，实现多个接口

多态：理解为同一种物质的多种形态

* + 多态使用的前提：

1，有继承或者实现关系

2，要方法重写

3，父类引用指向子类对象

* + 多态的成员访问特点：

方法的运行看右边，其他都看左边

* + 多态的好处：

提高了程序的扩展性

* + 多态的弊端：

不能访问子类的特有功能

* + 多态的分类
    - 类的多态

abstract class Fu {

public abstract void method();

}

class Zi extends Fu {

public void method(){

System.out.println(“重写父类抽象方法”);

}

}

//类的多态使用

Fu fu= new Zi();

* + - 接口的多态

interface Fu {

public abstract void method();

}

class Zi implements Fu {

public void method(){

System.out.println(“重写接口抽象方法”);

}

}

//接口的多态使用

Fu fu = new Zi();

instanceof 关键字

格式： 对象名 instanceof 类名

返回值： true, false

作用： 判断指定的对象 是否为 给定类创建的对象