第01天 java面向对象

今日内容介绍

* 接口
* 匿名对象&final
* 多态

# 接口

## 接口的概述

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。

## 接口的格式&使用

### 接口的格式

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;

}

### 接口的使用

接口中的方法全是抽象方法,直接new接口来调用方法没有意义,Java也不允许这样干

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。

### 案例代码一:

**package** com.itheima\_01;

/\*

\* Java语言的继承是单一继承，一个子类只能有一个父类（一个儿子只能有一个亲爹）

\* Java语言给我们提供了一种机制，用于处理继承单一的局限性的，接口

\*

\* 接口：接口是一个比抽象类还抽象的类，接口里所有的方法全是抽象方法，接口和类的关系是实现，implements

\* interface

\*

\* 格式：

\* interface 接口名 {

\*

\* }

\*

\*/

**public** **class** InterfaceDemo {

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

BillGates gates = **new** BillGates();

gates.code();

}

}

**class** Boss {

**public** **void** manage() {

System.*out*.println("管理公司");

}

}

**class** Programmer {

**public** **void** code() {

System.*out*.println("敲代码");

}

}

//比尔盖茨

**class** BillGates **extends** Programmer {

}

## 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fnal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。

### 案例代码二:

**package** com.itheima\_01;

/\*

\* 接口的成员特点：

\* 只能有抽象方法

\* 只能有常量

\* 默认使用public&abstract修饰方法

\* 只能使用public&abstract修饰方法

\* 默认使用public static final来修饰成员变量

\*

\* 建议：建议大家手动的给上默认修饰符

\*

\* 注意：

\* 接口不能创建对象（不能实例化）

\* 类与接口的关系是实现关系，一个类实现一个接口必须实现它所有的方法

\*/

**public** **class** InterfaceDemo2 {

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

//Animal a = new Animal();

//Animal.num;

}

}

**interface** Animal {

**public** **static** **final** **int** *num* = 10;

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

}

**class** Cat **implements** Animal {

**public** **void** eat() {

}

}

## 接口和类的关系

A:类与类之间:继承关系,一个类只能直接继承一个父类,但是支持多重继承

B:类与接口之间:只有实现关系,一个类可以实现多个接口

C:接口与接口之间:只有继承关系,一个接口可以继承多个接口

### 案例代码三:

**package** com.itheima\_01;

/\*

\*

\* 类与类：继承关系，单一继承，多层继承

\* 类与接口：实现关系，多实现

\* 接口与接口的关系：继承关系，多继承

\*/

**public** **class** InterfaceDemo3 {

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

}

}

**interface** InterA **extends** InterB {

**public** **abstract** **void** method();

}

**interface** InterB {

**public** **abstract** **void** function();

}

**interface** InterC **extends** InterA {

}

**class** Demo **implements** InterC {

@Override

**public** **void** method() {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** function() {

// **TODO** Auto-generated method stub

}

}

## 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。

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）。

集合体系中大量使用接口

Collection接口

List接口

ArrayList实现类

LinkedList实现类

Set接口

## 接口优点

1.类与接口的关系，实现关系，而且是多实现，一个类可以实现多个接口，类与类之间是继承关系，java中的继承是单一继承，一个类只能有一个父类，打破了继承的局限性。

2.对外提供规则（USB接口）

3.降低了程序的耦合性（可以实现模块化开发，定义好规则，每个人实现自己的模块，提高了开发的效率）

## 接口和抽象类的区别

1.共性：

不断的进行抽取，抽取出抽象的，没有具体实现的方法,都不能实例化（不能创建对象）

2.区别1: 与类的关系

(1)类与接口是实现关系，而且是多实现，一个类可以实现多个接口，类与抽象类是继承关系，Java中的继承是单一继承，多层继承，一个类只能继承一个父类，但是可以有爷爷类

(2)区别2： 成员

a.成员变量

抽象类可以有成员变量，也可以有常量

接口只能有常量，默认修饰符public static final

b.成员方法

抽象类可以有抽象方法，也可以有非抽象方法

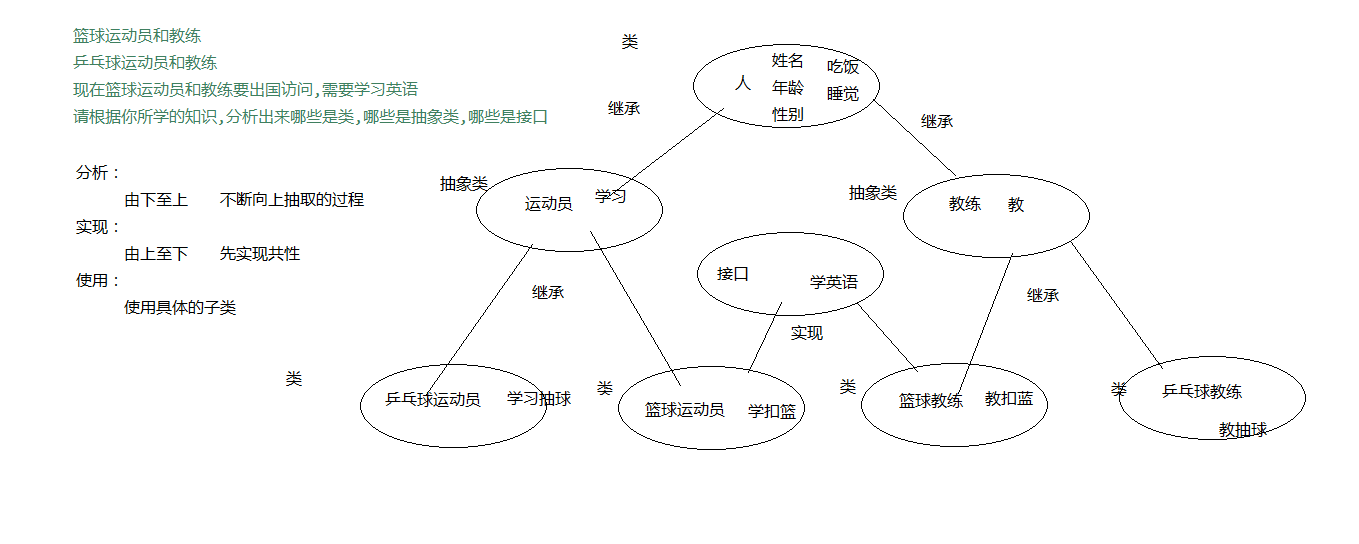
接口只能有抽象方法，默认修饰符 public abstract

c.构造方法

抽象类有构造方法，为子类提供

接口没有构造方法

## 运动员案例



### 案例代码四:

package com.itheima\_02;

/\*

\* 篮球运动员和教练

乒乓球运动员和教练

现在篮球运动员和教练要出国访问,需要学习英语

请根据你所学的知识,分析出来哪些是类,哪些是抽象类,哪些是接口

\*/

public class InterfaceTest {

public static void main(String[] args) {

//创建篮球运动员对象

BasketBallPlayer bbp = new BasketBallPlayer();

bbp.name = "女兆月日";

bbp.age = 35;

bbp.gender = "男";

bbp.sleep();

bbp.study();

bbp.speak();

System.out.println("-------------");

//创建乒乓球教练对象

PingpangCoach ppc = new PingpangCoach();

ppc.name = "刘胖子";

ppc.age = 40;

ppc.gender = "男";

ppc.sleep();

ppc.teach();

//ppc.speak();

}

}

class Person {

String name;//姓名

int age;//年龄

String gender;//性别

//无参构造

public Person() {}

//有参构造

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

this.name = name;

this.age = age;

this.gender = gender;

}

//吃

public void eat() {

System.out.println("吃饭");

}

//睡

public void sleep() {

System.out.println("睡觉");

}

}

//学习说英语

interface SpeakEnglish {

public abstract void speak();

}

//运动员

abstract class Player extends Person {

//学习

public abstract void study();

}

//教练

abstract class Coach extends Person {

//教

public abstract void teach();

}

//篮球运动员

class BasketBallPlayer extends Player implements SpeakEnglish{

@Override

public void study() {

System.out.println("学扣篮");

}

@Override

public void speak() {

System.out.println("说英语");

}

}

//乒乓球运动员

class PingpangPlayer extends Player {

@Override

public void study() {

System.out.println("学抽球");

}

}

//篮球教练

class BasketBallCoach extends Coach implements SpeakEnglish {

@Override

public void teach() {

System.out.println("教扣篮");

}

@Override

public void speak() {

System.out.println("说英语");

}

}

//乒乓球教练

class PingpangCoach extends Coach {

@Override

public void teach() {

System.out.println("教抽球");

}

}

# 匿名对象&final

## 匿名对象定义&使用

匿名对象即无名对象,直接使用new关键字来创建对象

### 案例代码五:

**package** com.itheima\_01;

/\*

\* 匿名对象:没有名字的对象

\* 匿名对象的应用场景：

\* 当方法只调用一次的时候可以使用匿名对象

\* 可以当作参数进行传递，但是无法在传参之前做其他的事情

\*

\* 注意：匿名对象可以调用成员变量并赋值，但是赋值并没有意义

\*

\*/

**public** **class** AnonymousObejctDemo {

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

//Student s = new Student();

//s.study();

//s.study();

//s.study();

//new Student();//匿名对象，没有变量引用的对象

//new Student().study();

//new Student().study();

//new Student().study();

//new Student().age = 18;

//System.out.println(new Student().age);

//Student s = new Student();

//s.age = 18;

//s.name = "张三";

//method(s);

*method*(**new** Student());

}

**public** **static** **void** method(Student s) {

}

}

**class** Student {

String name;

**int** age;

**public** **void** study() {

System.*out*.println("好好学习，高薪就业");

}

}

## final关键字

final： 修饰符，可以用于修饰类、成员方法和成员变量

final所修饰的类：不能被继承，不能有子类

final所修饰的方法:不能被重写

final所修饰的变量：是不可以修改的，是常量

### 案例代码六:

**package** com.itheima\_01;

/\*

\* final： 修饰符，可以用于修饰类、成员方法和成员变量

\* final所修饰的类：不能被继承，不能有子类

\* final所修饰的方法:不能被重写

\* final所修饰的变量：是不可以修改的，是常量

\*

\* 常量：

\* 字面值常量：1,2,3

\* 自定义常量：被final所修饰的成员变量，一旦初始化则不可改变

\*

\* 注意：自定义常量必须初始化，可以选择显示初始化或者构造初始化

\*

\*

\*/

**public** **class** FinalDemo {

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

//Animal a = new Animal();

//a.eat();

Dog d = **new** Dog();

//d.eat();

//d.num = 20;

System.*out*.println(d.NUM);

}

}

/\*final\*/ **class** Animal {

**public** **final** **void** eat() {

System.*out*.println("吃东西");

}

}

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

/\*public void eat() {}\*/

**final** **int** NUM;

**public** Dog() {

NUM = 10;

}

}

# 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。

## 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();

A: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();

B: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();

C: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();

### 案例代码七:

**package** com.itheima\_01;

/\*

\* 多态的前提：

\* 子父类的继承关系

\* 方法的重写

\* 父类引用指向子类对象

\*

\* 动态绑定：运行期间调用的方法，是根据其具体的类型

\*

\*

\*

\*

\*/

**public** **class** PoymorphicDemo {

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

/\*Cat c = new Cat();

c.eat();\*/

//父类引用 Animal a

//指向 =

//子类对象 new Cat()

Animal a = **new** Cat();

a.eat();

}

}

**class** Animal {

**public** **void** eat() {

System.*out*.println("吃东西");

}

}

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

**public** **void** eat() {

System.*out*.println("猫吃鱼");

}

}

## 多态成员的特点

A: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。

B: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

### 案例代码八:

**package** com.itheima\_01;

/\*

\*

\* 多态的成员特点:

\* 成员变量 编译时看的是左边，运行时看的左边

\* 成员方法 编译时看的是左边，运行时看右边

\* 静态方法 编译时看的是左边，运行时看的也是左边

\*

\*

\* 编译时看的都是左边，运行时成员方法看的是右边，其他（成员变量和静态的方法）看的都是左边

\*

\*/

**public** **class** PoymorphicDemo2 {

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

Dad d = **new** Kid();

//System.out.println(d.num);

//d.method();

d.*function*();//使用变量去调用静态方法，其实相当于用变量类型的类名去调用

}

}

**class** Dad {

**int** num = 20;

**public** **void** method() {

System.*out*.println("我是父类方法");

}

**public** **static** **void** function() {

System.*out*.println("我是父类静态方法");

}

}

**class** Kid **extends** Dad {

**int** num = 10;

**public** **void** method() {

System.*out*.println("我是子类方法");

}

**public** **static** **void** function() {

System.*out*.println("我是子类静态方法");

}

}

## 多态中向上转型与向下转型

多态的转型分为向上转型与向下转型两种：

A: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();

B: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

### 案例代码九:

**package** com.itheima\_01;

/\*

\*

\* 多态中的向上转型和向下转型:

\*

\* 引用类型之间的转换

\* 向上转型

\* 由小到大(子类型转换成父类型)

\* 向下转型

\* 由大到小

\* 基本数据类型的转换

\* 自动类型转换

\* 由小到大

\* byte short char --- int --- long --- float --- double

\* 强制类型转换

\* 由大到小

\*

\*

\*

\*/

**public** **class** PoymorphicDemo3 {

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

Animal2 a = **new** Dog();//向上转型

//a.eat();

Dog d = (Dog)a;//向下转型

d.swim();

}

}

**class** Animal2 {

**public** **void** eat() {

System.*out*.println("吃东西");

}

}

**class** Dog **extends** Animal2 {

**public** **void** eat() {

System.*out*.println("啃骨头");

}

**public** **void** swim() {

System.*out*.println("狗刨");

}

}

## 多态的优缺点

### 案例代码十:

**package** com.itheima\_01;

/\*

\*

\* 多态的优缺点

\* 优点：可以提高可维护性（多态前提所保证的），提高代码的可扩展性

缺点：无法直接访问子类特有的成员

\*/

**public** **class** PoymorphicDemo4 {

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

MiFactory factory = **new** MiFactory();

factory.createPhone(**new** MiNote());

factory.createPhone(**new** RedMi());

}

}

**class** MiFactory {

/\*public void createPhone(MiNote mi) {

mi.call();

}

public void createPhone(RedMi mi) {

mi.call();

}\*/

**public** **void** createPhone(Phone p) {

p.call();

}

}

**interface** Phone {

**public** **void** call();

}

//小米Note

**class** MiNote **implements** Phone{

**public** **void** call() {

System.*out*.println("小米Note打电话");

}

}

//红米

**class** RedMi **implements** Phone {

**public** **void** call() {

System.*out*.println("红米打电话");

}

}