day02【接口,多态】

今日内容

- 接口
- 多态

教学目标

- [ ] 能够写出接口的定义格式
- [ ] 能够写出接口的实现格式
- [ ] 能够说出接口中的成员特点
- [ ] 能够说出多态的前提
- [ ] 能够写出多态的格式
- [ ] 能够理解多态向上转型和向下转型
- [ ] 能够完成笔记本案例

第一章 接口

1.1 概述

接口，是Java语言中一种引用类型，是方法的集合，如果说类的内部封装了成员变量、构造方法和成员方法，那么接口的内部主要就是**封装了方法**，包含抽象方法（JDK 7及以前），默认方法和静态方法（JDK 8）。

接口的定义，它与定义类方式相似，但是使用 `interface` 关键字。它也会被编译成.class文件，但一定要明确它并不是类，而是另外一种引用数据类型。

public class 类名.java-->.class

public interface 接口名.java-->.class

> 引用数据类型：数组，类，接口。

接口的使用，它不能创建对象，但是可以被实现（ `implements` ，类似于被继承）。一个实现接口的类（可以看做是接口的子类），需要实现接口中所有的抽象方法，创建该类对象，就可以调用方法了，否则它必须是一个抽象类。

1.2 定义格式

```
public interface 接口名称 {
    // 抽象方法
    // 默认方法
    // 静态方法
}
```

含有抽象方法

抽象方法：使用`abstract`关键字修饰，可以省略，没有方法体。该方法供子类实现使用。

代码如下：

```
public interface InterFaceName {
    public abstract void method();
}
```

含有默认方法和静态方法

默认方法：使用`default`修饰，不可省略，供子类调用或者子类重写。

静态方法：使用`static`修饰，供接口直接调用。

代码如下：

```
public interface InterFaceName {
    public default void method() {
        // 执行语句
    }
    public static void method2() {
        // 执行语句
    }
}
```

1.3 基本的实现

实现的概述

类与接口的关系为实现关系，即**类实现接口**，该类可以称为接口的实现类，也可以称为接口的子类。实现的动作类似继承，格式相仿，只是关键字不同，实现使用`implements`关键字。

非抽象子类实现接口：

1. 必须重写接口中所有抽象方法。
2. 继承了接口的默认方法，即可以直接调用，也可以重写。

实现格式：

```
class 类名 implements 接口名 {
    // 重写接口中抽象方法【必须】
    // 重写接口中默认方法【可选】
}
```

抽象方法的使用

必须全部实现，代码如下：

定义接口：

```
public interface LiveAble {
    // 定义抽象方法
    public abstract void eat();
    public abstract void sleep();
}
```

定义实现类：

```
public class Animal implements LiveAble {
    @Override
    public void eat() {
        System.out.println("吃东西");
    }

    @Override
    public void sleep() {
        System.out.println("晚上睡");
    }
}
```

定义测试类：

```
public class InterfaceDemo {
    public static void main(String[] args) {
        // 创建子类对象
        Animal a = new Animal();
        // 调用实现后的方法
        a.eat();
        a.sleep();
    }
}
输出结果：
吃东西
晚上睡
```

默认方法的使用

可以继承，可以重写，二选一，但是只能通过实现类的对象来调用。

1. 继承默认方法，代码如下：

定义接口：

```
public interface LiveAble {
    public default void fly(){
        System.out.println("天上飞");
    }
}
```

定义实现类：

```
public class Animal implements LiveAble {
    // 继承，什么都不用写，直接调用
}
```

定义测试类：

```
public class InterfaceDemo {
    public static void main(String[] args) {
        // 创建子类对象
        Animal a = new Animal();
        // 调用默认方法
        a.fly();
    }
}
输出结果：
天上飞
```

1. 重写默认方法，代码如下：

定义接口：

```
public interface LiveAble {
    public default void fly(){
        System.out.println("天上飞");
    }
}
```

定义实现类：

```
public class Animal implements LiveAble {
    @Override
    public void fly() {
        System.out.println("自由自在的飞");
    }
}
```

定义测试类：

```
public class InterfaceDemo {
    public static void main(String[] args) {
        // 创建子类对象
        Animal a = new Animal();
        // 调用重写方法
        a.fly();
    }
}
输出结果：
自由自在的飞
```

静态方法的使用

静态与.class 文件相关，只能使用接口名调用，不可以通过实现类的类名或者实现类的对象调用，代码如下：

定义接口：

```
public interface LiveAble {
    public static void run(){
        System.out.println("跑起来~~~");
    }
}
```

定义实现类：

```
public class Animal implements LiveAble {
    // 无法重写静态方法
}
```

定义测试类：

```
public class InterfaceDemo {
    public static void main(String[] args) {
        // Animal.run(); // 【错误】无法继承方法,也无法调用
        LiveAble.run(); //
    }
}
输出结果：
跑起来~~~
```

1.4 接口的多实现

之前学过，在继承体系中，一个类只能继承一个父类。而对于接口而言，一个类是可以实现多个接口的，这叫做接口的**多实现**。并且，一个类能继承一个父类，同时实现多个接口。

实现格式：

```
class 类名 [extends 父类名] implements 接口名1,接口名2,接口名3... {
    // 重写接口中抽象方法【必须】
    // 重写接口中默认方法【不重名时可选】
}
```

[]：表示可选操作。

抽象方法

接口中，有多个抽象方法时，实现类必须重写所有抽象方法。**如果抽象方法有重名的，只需要重写一次。**代码如下：

定义多个接口：

```
interface A {
    public abstract void showA();
    public abstract void show();
}

interface B {
    public abstract void showB();
    public abstract void show();
}
```

定义实现类：

```
public class C implements A,B{
    @Override
    public void showA() {
        System.out.println("showA");
    }

    @Override
    public void showB() {
        System.out.println("showB");
    }

    @Override
    public void show() {
        System.out.println("show");
    }
}
```

默认方法

接口中，有多个默认方法时，实现类都可继承使用。**如果默认方法有重名的，必须重写一次。**代码如下：

定义多个接口：

```
interface A {
    public default void methodA(){}
    public default void method(){}
}

interface B {
    public default void methodB(){}
    public default void method(){}
}
```

定义实现类：

```
public class C implements A,B{
    @Override
    public void method() {
        System.out.println("method");
    }
}
```

静态方法

接口中，存在同名的静态方法并不会冲突，原因是只能通过各自接口名访问静态方法。

优先级的问题

当一个类，既继承一个父类，又实现若干个接口时，父类中的成员方法与接口中的默认方法重名，子类就近选择执行父类的成员方法。代码如下：

定义接口：

```
interface A {
    public default void methodA(){
        System.out.println("AAAAAAAAAAA");
    }
}
```

定义父类：

```
class D {
    public void methodA(){
        System.out.println("DDDDDDDDDDD");
    }
}
```

定义子类：

```
class C extends D implements A {
    // 未重写methodA方法
}
```

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        C c = new C();
        c.methodA();
    }
}
输出结果:
DDDDDDDDDDD
```

1.5 接口的多继承【了解】

一个接口能继承另一个或者多个接口，这和类之间的继承比较相似。接口的继承使用 `extends` 关键字，子接口继承父接口的方法。**如果父接口中的默认方法有重名的，那么子接口需要重写一次。**代码如下：

定义父接口：

```
interface A {
    public default void method(){
        System.out.println("AAAAAAAAAAAAAAAAAAA");
    }
}

interface B {
    public default void method(){
        System.out.println("BBBBBBBBBBBBBBBBBBB");
    }
}
```

定义子接口：

```
interface D extends A,B{
    @Override
    public default void method() {
        System.out.println("DDDDDDDDDDDDDD");
    }
}
```

> 小贴士：
>
> 子接口重写默认方法时，default关键字可以保留。
>
> 子类重写默认方法时，default关键字不可以保留。

1.6 其他成员特点

- 接口中，无法定义成员变量，但是可以定义常量，其值不可以改变，默认使用public static final修饰。
- 接口中，没有构造方法，不能创建对象。

- 接口中，没有静态代码块。

1.7 抽象类和接口的练习

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。

可是有的犬还有其他额外功能，而这个功能并不在这个事物的体系中，例如：缉毒犬。缉毒的这个功能有好多种动物都有，例如：缉毒猪，缉毒鼠。我们可以将这个额外功能定义接口中，让缉毒犬继承犬且实现缉毒接口，这样缉毒犬既具备犬科自身特点也有缉毒功能。

```
//定义缉毒接口 缉毒的词组(anti-Narcotics)比较长,在此使用拼音替代
interface JiDu{
    //缉毒
    public abstract void jiDu();
}
//定义犬科,存放共性功能
abstract class Dog{
    //吃饭
    public abstract void eat();
    //吼叫
    public abstract void roar();
}
//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 JiDuQuan extends Dog implements JiDu{
    public void jiDu() {
    }
    void eat() {
    }
    void roar() {
```

```
    }
}

//缉毒猪
class JiDuZhu implements JiDu{
    public void jiDu() {
    }
}
```

> 讲完抽象类和接口后,相信有许多同学会存有疑惑,两者的共性那么多,只留其中一种不就行了,这里就得知道抽象类和接口从根本上解决了哪些问题.
>
> 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
>
> 抽象类为继承体系中的共性内容, 接口为继承体系中的扩展功能
>
> 接口还是后面一个知识点的基础(lambada)

1.8 面试题:接口和抽象类的区别【自学】

> 以下内容需要在掌握接口和抽象类的基础知识后,再进行对比学习

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)

抽象类为继承体系中的共性内容,接口为继承体系中的扩展功能

- 成员区别
  - 抽象类
    - 变量,常量；有构造方法；有抽象方法,也有非抽象方法
  - 接口
    - 常量；抽象方法
- 关系区别
  - 类与类
    - 继承，单继承
  - 类与接口
    - 实现，可以单实现，也可以多实现
  - 接口与接口
    - 继承，单继承，多继承
- 设计理念区别
  - 抽象类

- 对类抽象，包括属性、行为
- 接口
  - 对行为抽象，主要是行为

第二章 多态

1.1 概述

引入

多态是继封装、继承之后，面向对象的第三大特性。

生活中，比如跑的动作，小猫、小狗和大象，跑起来是不一样的。再比如飞的动作，昆虫、鸟类和飞机，飞起来也是不一样的。可见，同一行为，通过不同的事物，可以体现出来的不同的形态。多态，描述的就是这样的状态。

定义

- **多态**： 是指同一行为，具有多个不同表现形式。

前提【重点】

1. 继承或者实现【二选一】
2. 方法的重写【意义体现：不重写，无意义】
3. 父类引用指向子类对象【格式体现】

1.2 多态的体现

多态体现的格式：

```
父类类型 变量名 = new 子类对象；
变量名.方法名();
```

> 父类类型：指子类对象继承的父类类型，或者实现的父接口类型。

代码如下：

```
Fu f = new Zi();
f.method();
```

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，执行的是子类重写后方法。

代码如下：

定义父类：

```
public abstract class Animal {
    public abstract void eat();
}
```

定义子类：

```
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
}
```

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Animal a1 = new Cat();
        // 调用的是 Cat 的 eat
        a1.eat();

        // 多态形式，创建对象
        Animal a2 = new Dog();
        // 调用的是 Dog 的 eat
        a2.eat();
    }
}
```

1.3 多态的好处

实际开发的过程中，父类类型作为方法形式参数，传递子类对象给方法，进行方法的调用，更能体现出多态的扩展性与便利。代码如下：

定义父类：

```
public abstract class Animal {
    public abstract void eat();
}
```

定义子类：

```
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
}
```

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Cat c = new Cat();
        Dog d = new Dog();

        // 调用showCatEat
        showCatEat(c);
        // 调用showDogEat
        showDogEat(d);

        /*
        以上两个方法，均可以被showAnimalEat(Animal a)方法所替代
        而执行效果一致
        */
        showAnimalEat(c);
        showAnimalEat(d);
    }

    public static void showCatEat (Cat c){
        c.eat();
    }

    public static void showDogEat (Dog d){
        d.eat();
    }

    public static void showAnimalEat (Animal a){
        a.eat();
    }
}
```

由于多态特性的支持，showAnimalEat方法的Animal类型，是Cat和Dog的父类类型，父类类型接收子类对象，当然可以把Cat对象和Dog对象，传递给方法。

当eat方法执行时，多态规定，执行的是子类重写的方法，那么效果自然与showCatEat、showDogEat方法一致，所以showAnimalEat完全可以替代以上两方法。

不仅仅是替代，在扩展性方面，无论之后再多的子类出现，我们都不需要编写showXxxEat方法了，直接使用showAnimalEat都可以完成。

所以，多态的好处，体现在，可以使程序编写的更简单，并有良好的扩展。

1.4 引用类型转换

多态的转型分为向上转型与向下转型两种：

向上转型

- **向上转型**：多态本身是子类类型向父类类型向上转换的过程，这个过程是默认的。

当父类引用指向一个子类对象时，便是向上转型。

使用格式：

```
父类类型  变量名 = new 子类类型();
如：Animal a = new Cat();
```

向下转型

- **向下转型**：父类类型向子类类型向下转换的过程，这个过程是强制的。

一个已经向上转型的子类对象，将父类引用转为子类引用，可以使用强制类型转换的格式，便是向下转型。

使用格式：

```
子类类型 变量名 = （子类类型） 父类变量名;
如:Cat c =(Cat) a;
```

为什么要转型

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类有而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

转型演示，代码如下：

定义类：

```
abstract class Animal {
    abstract void eat();
}

class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void catchMouse() {
        System.out.println("抓老鼠");
    }
}
```

```
13
14  class Dog extends Animal {
15      public void eat() {
16          System.out.println("吃骨头");
17      }
18      public void watchHouse() {
19          System.out.println("看家");
20      }
21  }
```

定义测试类：

```
1   public class Test {
2       public static void main(String[] args) {
3           // 向上转型
4           Animal a = new Cat();
5           a.eat();                // 调用的是 Cat 的 eat
6
7           // 向下转型
8           Cat c = (Cat)a;
9           c.catchMouse();         // 调用的是 Cat 的 catchMouse
10      }
11  }
```

转型的异常

转型的过程中，一不小心就会遇到这样的问题，请看如下代码：

```
1   public class Test {
2       public static void main(String[] args) {
3           // 向上转型
4           Animal a = new Cat();
5           a.eat();                // 调用的是 Cat 的 eat
6
7           // 向下转型
8           Dog d = (Dog)a;
9           d.watchHouse();         // 调用的是 Dog 的 watchHouse 【运行报错】
10      }
11  }
```

这段代码可以通过编译，但是运行时，却报出了 `ClassCastException` ，类型转换异常！这是因为，明明创建了Cat类型对象，运行时，当然不能转换成Dog对象的。这两个类型并没有任何继承关系，不符合类型转换的定义。

为了避免ClassCastException的发生，Java提供了 `instanceof` 关键字，给引用变量做类型的校验，格式如下：

```
1   变量名 instanceof 数据类型
2   如果变量属于该数据类型，返回true。
3   如果变量不属于该数据类型，返回false。
```

所以，转换前，我们最好先做一个判断，代码如下：

```java
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat

        // 向下转型
        if (a instanceof Cat){
            Cat c = (Cat)a;
            c.catchMouse();        // 调用的是 Cat 的 catchMouse
        } else if (a instanceof Dog){
            Dog d = (Dog)a;
            d.watchHouse();       // 调用的是 Dog 的 watchHouse
        }
    }
}
```

第三章 综合案例

3.1 案例需求

定义笔记本类，具备开机，关机和使用USB设备的功能。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

3.2 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。

3.3 代码实现

```java
//定义鼠标、键盘，笔记本三者之间应该遵守的规则
public interface USB {
    void open();// 开启功能

    void close();// 关闭功能
}
//鼠标实现USB规则
public 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
public 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
public 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");
    }
}
//测试
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();
    }
}

```