面向对象进阶

目录

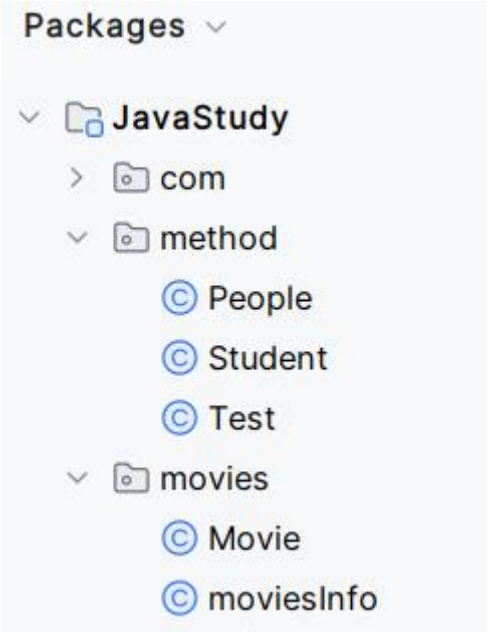
Contents

什么是包？

- 包是用来分门别类的管理各种不同类的，类似于文件夹、建包利于程序的管理和维护。
- 建包的语法格式：package 公司域名倒写.技术名称。报名建议全部英文小写，且具备意义

```
package com.example.project
public class Student {

}
```



- 建包语句必须在第一行，一般IDEA工具会帮助创建

导包

- **相同包下的类**可以直接访问，不同包下的类必须导包,才可以使用！
- 导包格式：**import 包名.类名;**
- 假如一个类中需要用到不同类，而这个两个类的名称是一样的，那么默认只能导入一个类，另一个类要带包名访问。

目录

Contents

什么是权限修饰符？

- 就是是用来限制类中的成员（成员变量、成员方法、构造方法、代码块…）能够被访问的范围。

权限修饰符有几种？各自的作用是什么？

private	只能本类
缺省	本类、同一个包中的类
protected	本类，同一个包中的类、子孙类中
public	任意位置

权限修饰符

- 权限修饰符：是用来控制一个成员能够被访问的范围。
- 可以修饰成员变量，方法，构造方法，内部类，不同权限修饰符修饰的成员能够被访问的范围将受到限制。

权限修饰符的分类和具体作用范围：

- 权限修饰符：有四种作用范围由小到大（private -> 缺省 -> protected - > public ）

修饰符	本类里	同一个包中的类	子孙类	任意类
private	√			
缺省	√	√		
protected	√	√	√	
public	√	√	√	√

权限修饰符的使用规则

- 实际开发中，一般只用private和public
- 成员变量一般私有
- 方法一般公开

目录

Contents

final的作用

- final 关键字是最终的意思，可以修饰（**类、方法、变量**）
- **修饰类**：表明该类是最终类，不能被继承。
- **修饰方法**：表明该方法是最终方法，不能被重写。
- **修饰变量**：表示该变量第一次赋值后，不能再次被赋值(有且仅能被赋值一次)。

final修饰变量的注意

- final修饰的变量是**基本类型**：那么变量存储的**数据值**不能发生改变。
- final修饰的变量是**引用类型**：那么变量存储的**地址值**不能发生改变，但是对象内部的属性值可以发生变化。

目录

Conte
nts

常量

- 常量是使用了public static final修饰的成员变量，必须有初始化值，而且执行的过程中其值不能被改变。
- 常量名的命名规范：**英文单词全部大写**，多个单词下划线连接起来。
- 常量的作用：通常用来记录系统的配置数据。

```
public class Constant {
    public static final String SCHOOL_NAME = “南京邮电大学";
    public static final String LOGIN_NAME = “admin";
    public static final String PASS_WORD = “password";
}
```

目录

Contents

抽象类

- 在Java中abstract是抽象的意思，可以修饰类、成员方法。
- abstract修饰类，这个类就是抽象类；修饰方法，这个方法就是抽象方法。

```
修饰符 abstract class 类名{
        修饰符 abstract 返回值类型 方法名称(形参列表);
}
```

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

注意事项

- 抽象方法只有方法声明，不能声明方法体。
- 一个类中如果定义了抽象方法，这个类必须声明成抽象类，否则报错。

抽象的使用场景

- 抽象类可以理解成不完整的设计图，**一般作为父类**，让子类来继承。
- 当父类知道子类一定要完成某些行为，但是**每个子类该行为的实现又不同**，于是该父类就把

  该行为定义成**抽象方法**的形式，**具体实现**交给子类去完成。此时这个类就可以声明成**抽象类**。

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

特征和注意事项

- 类有的成员（成员变量、方法、构造方法），抽象类都具备
- 抽象类中**不一定**有抽象方法，**有抽象方法的类一定是抽象类**
- 一个类**继承**了抽象类必须**重写**抽象类的**全部抽象方法**，否则这个类也必须定义成抽象类。
- 不能用abstract修饰变量、代码块、构造方法。
- **最重要的特征：**得到了抽象方法，失去了创建对象的能力（**有得有失**）

系统需求

- 某加油站推出了2种支付卡，一种是预存10000的**金卡**，后续加油享受**8折优惠**，另一种是预存5000的**银卡** ,后续加油享受**8.5折优惠**。
- 请分别实现2种卡片进入收银系统后的逻辑，卡片需要包含主人名称，余额，支付功能。

分析实现

- 创建一张卡片父类：定义属性包括主人名称、余额、**支付功能**（**具体实现**交给子类）
- 创建一张金卡类：重写支付功能，按照原价的8折计算输出。
- 创建一张银卡类：重写支付功能，按照原价的8.5折计算输出。

```
public abstract class Card {
    private String useName;
    private double balance;
    public abstract void pay(double payMoney);
    //get和set方法
    public String getUseName() { return useName; }
    public void setUseName(String useName) {this.useName = useName;}
    public double getBalance() {return balance;}
    public void setBalance(double balance) {this.balance = balance;}
}
```

```
public class GoldCard extends Card{
    @Override
    public void pay(double payMoney){
        System.out.println("您当前余额为："+getBalance());
        System.out.println("当前支付："+payMoney*0.8);
        //更新余额
        setBalance(getBalance()-payMoney*0.8);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        GoldCard g = new GoldCard();
        g.setBalance(10000);
        g.setUseName("小明");

        g.pay(400);
        System.out.println("剩余："+g.getBalance());
    }
}
```

```
public class SilverCard extends Card {
    @Override
    public void pay(double payMoney){
        System.out.println("您当前余额为："+getBalance());
        System.out.println("当前支付："+payMoney*0.85);
        //更新余额
        setBalance(getBalance()-payMoney*0.85);
    }
}
```

练习

- 编写带有抽象类的标准Javabean类

猫Cat　　属性：名字，年龄　　行为：吃小鱼，喝水

狗Dog　　属性：名字，年龄　　行为：吃骨头，喝水

标准**JavaBean**须满足如下书写要求：

- 成员变量使用 **private** 修饰。
- 提供成员变量对应的 **setXxx() / getXxx()**方法。
- 必须提供一个**无参构造方法；有参数构造方法是可写可不写的。**

- 为什么要用抽象方法

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

```
class Frog extends Animal {
    吃东西的方法
}
```

```
class Bird extends Animal {
    吃东西的方法
}
```

```
class Dog extends Animal {
    @Override
 public  void eating() {
   System.out.println(“狗吃骨头");
    }
}
```

```
class Cat extends Animal {
    @Override
 public  String eat(){
        System.out.println(“猫吃鱼");
        return “鱼骨头”
}
}
```

final和abstract是什么关系？

- 互斥关系
- abstract定义的**抽象类**作为模板让子类**继承**，final定义的类不能被继承。
- **抽象方法**定义通用功能让子类**重写**，final定义的方法子类不能重写。

总结

1、抽象类、抽象方法是什么样的？

- 都是用abstract修饰的；抽象方法只有方法声明，不能写方法体。
- 一个类中定义了抽象方法，这个类必须声明成抽象类。

2、抽象类基本作用是啥？

- 作为父类，用来被继承的。

3、继承抽象类有哪些要注意？

- 一个类如果继承了抽象类，那么这个类必须重写完抽象类的全部抽象方法，否则这个类也必须定义成抽象类。

目录

Contents

什么是多态？

- 指同类型的对象，表现出不同的形态。

多态的常见形式

```
父类类型 对象名称 = new 子类构造方法;
```

多态的前提

- 有继承/实现关系；有**父类引用**指向**子类对象**；有方法重写（**多态侧重行为多态**）。

- **多态（Polymorphism）：**

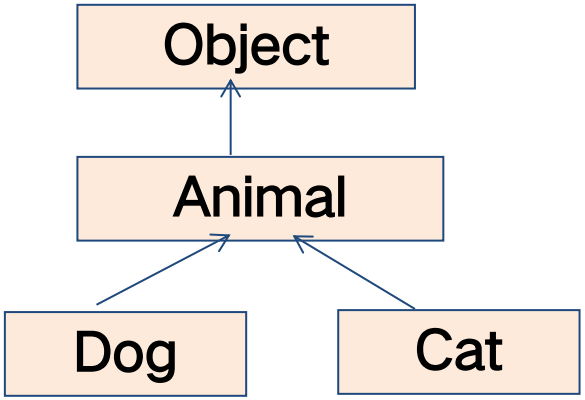
父类引用指向子类对象： 因为子类型可以看成是父类型，所以父类的变量可以引用（指向）子类的对象。

类型 变量名= new 构造方法(…);

类型 变量名= new 子类构造方法(…);

声明类型

实际类型



父类类型 对象名称 = new 子类构造方法;

Animal a = new Cat();

声明类型：Animal
实际类型：Cat

多态：声明类型始终是Animal的变量a可以表现为不同的形态(Dog或Cat)

```
public class Animal {
    void cry() {    }
}
class Dog extends Animal {
    @Override
    void cry() {
        System.out.println("狗叫");
    }
}
class Cat extends Animal {
    @Override
    void cry() {
        System.out.println("猫叫");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.cry();
        a = new Cat();
        a.cry();
    }
}
```

和继承有关的多态性是指父类的某个方法被其子类重写时，可以产生自己的功能行为

即**同一个操作**被**不同类型对象**调用时可能产生不同的行为。

多态中成员访问特点

- **变量调用：编译看左边，运行也看左边。**

编译看左边：javac在编译代码时，看左边父类中有没有这个变量。有则编译成功，无则编译失败。

运行也看左边：java在运行代码时，实际获取的时左边父类中的成员变量的值。

```
"C:\Prog
动物
```

```
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();
        System.out.println(a.name);
    }
}
```

```
public class Animal {
    String name ="动物";
    void cry() {
        System.out.println("动物叫");
    }
}
class Dog extends Animal {
    String name ="狗";
    @Override
    void cry() {
        System.out.println("狗叫");
    }
}
class Cat extends Animal {
    String name ="猫";
    @Override
    void cry() {
        System.out.println("猫叫");
    }
}
```

多态中成员访问特点

- **方法调用：编译看左边，运行看右边。**

编译看左边：javac在编译代码时，看左边父类中有没有这个方法。有则编译成功，无则编译失败。

运行看右边：java在运行代码时，实际运行的是子类中的方法。

```
"C:\Progra
动物
狗叫
```

```
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();
        System.out.println(a.name);
        a.cry();
    }
}
```

```
public class Animal {
    String name ="动物";
    void cry() {
        System.out.println("动物叫");
    }
}
class Dog extends Animal {
    String name ="狗";
    @Override
    void cry() {
        System.out.println("狗叫");
    }
}
class Cat extends Animal {
    String name ="猫";
    @Override
    void cry() {
        System.out.println("猫叫");
    }
}
```

动物
动物叫

多态中成员访问特点

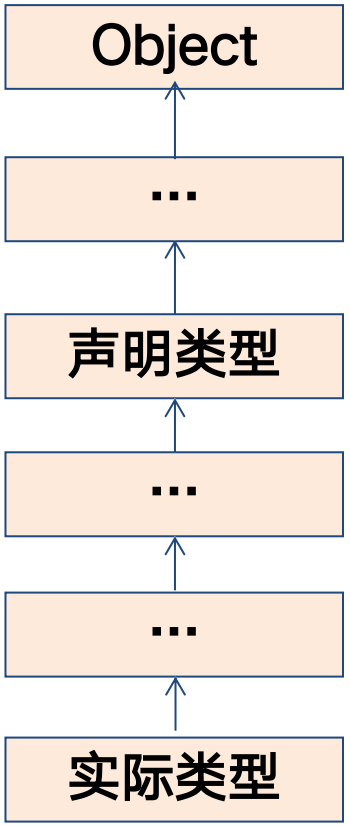
- **方法调用：编译看左边，运行看右边。**

编译看左边：javac在编译代码时，看左边父类中有没有这个方法。有则编译成功，无则编译失败。

声明类型中肯定包含该方法，否则编译不通过。

```
类型  变量名= new 子类构造方法(…);
```

声明类型　　实际类型



运行看右边：java在运行代码时，实际运行的是子类中的方法。

动态绑定的规则：从实际类型出发查找是否有要访问的方法，如果有则调用，如果没有则沿着继承链的顺序继续向上查找，直到找到为止。

填空题 3分

设置

请写出上述程序的运行结果：[填空1] [填空2] [填空3]

```
public class Animal {
    String name = "动物";
    void cry() { System.out.println("动物叫"); }
}
class Cat extends Animal {
    String name = "猫";
    @Override
    void cry() { System.out.println("猫叫"); }
}
class BlackCat extends Cat {
    String name = "黑猫";
}
```

```
public class Test {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.cry();        [填空1]
        a=new Cat();
        a.cry();        [填空2]
        a=new BlackCat();
        a.cry();        [填空3]
    }
}
```

作答

目录

Conte
nts

优势

- 在多态形式下，右边对象可以实现解耦合，便于扩展和维护。

```
Animal a = new Dog();
a.run(); // 后续业务行为随对象而变，后续代码无需修改
```

多态下会产生的一个问题:

- 多态下不能使用子类的独有功能

多态下会产生的一个问题:

- 多态下不能使用子类的独有功能

```
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();
        System.out.println(a.name);
        a.cry();
        a.run();//报错
    }
}
```

```
public class Animal {
    String name ="动物";
    void cry() { System.out.println("动物叫");}
}
class Dog extends Animal {
    String name ="狗";
    @Override
    void cry() {System.out.println("狗叫"); }
    void run() {System.out.println("狗在跑"); }
}
class Cat extends Animal {
    String name ="猫";
    @Override
    void cry() {System.out.println("猫叫");}
}
```

如果就要用这个方法，怎么办？

目录

Contents

自动类型转换（从子到父）： `Animal c = new Cat();`

强制类型转换（从父到子）

- **从父到子**（必须进行强制类型转换,否则报错）：

子类 对象变量 = (子类)父类类型的变量

```
Dog d = (Dog) a;
```

```
Animal a = new Dog();
a.cry();
Dog d = (Dog) a;
d.run();
```

- 作用：可以解决多态下的劣势，**可以实现调用子类独有的功能**。

- 注意： 有继承/实现关系的类就可以在编译阶段进行强制类型转换；但是，如果转型后的类型和对象真实对象的类型不是同一种类型，那么在运行代码时，就会出现ClassCastException

```
Animal a = new Dog();
Cat c = (Cat) a;// 出现异常 ClassCastException
```

Java建议强转转换前使用instanceof判断当前对象的真实类型，再进行强制转换

```
Animal a = new Dog();
Cat c = (Cat) a;// 出现异常 ClassCastException
```

变量名 instanceof 真实类型

判断关键字左边的变量指向的对象的真实类型，是否是右边的类型或者是其子类类型，是则返回true，反之。

```
if(a instanceof Dog){
    Dog d = (Dog) a;
} else if (a instanceof Cat) {
    Cat c = (Cat) a;
}else {
    System.out.println("没有这个类型，无法转换");
}
```

模式匹配写法 (Java 14及以后):

```
if(a instanceof Dog d){
}
```

总结

1. 引用数据类型的类型转换，有几种方式？
   - 自动类型转换、强制类型转换。
2. 强制类型转换能解决什么问题？
   - 可以转换成真正的子类类型，从而调用子类独有功能。
3. 强制类型转换需要注意什么？
   - 有继承关系/实现的2个类型就可以进行强制转换，编译无问题。
   - 运行时，如果发现强制转换后的类型不是对象真实类型则报错（ClassCastException）
4. 强制类型转换前最好做什么事情，如何进行？
   - 使用instanceof判断当前对象的真实类型，再进行强制转换
   - 对象变量名 instanceof 真实类型