day03 【static、接口、多态、内部类】

今日内容

- 接口
- 多态
- 内部类

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
- [ ] 能够说出内部类概念
- [ ] 能够理解匿名内部类的编写格式

第一章 接口

知识点--3.1 概述

目标:

- 引用数据类型除了类其实还有接口,接下来学习接口的概述

路径:

- 接口的概述

讲解:

概述: 接口是Java语言中的一种引用类型，是方法的"集合"，所以接口的内部主要就是**定义方法**，包含常量,抽象方法（JDK 7及以前），默认方法和静态方法（JDK 8）,私有方法(jdk9)。

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。

小结:

- 接口是java中的一种引用数据类型
- 接口中的成员:
  - 常量(jdk7及其以前)
  - 抽象方法(jdk7及其以前)
  - 默认方法(jdk8额外增加)
  - 静态方法(jdk8额外增加)
  - 私有方法(jdk9额外增加)
- 了解:
  - 定义接口使用interface关键字,接口编译后也会产生class文件
  - 接口不能创建对象,需要使用类来实现接口(继承),实现接口需要使用 implements 关键字

知识点--3.2 定义格式

目标:

- 如何定义一个接口

路径:

- 定义格式的格式

讲解:

格式

```
public interface 接口名称 {
    - 常量(jdk7及其以前)
    - 抽象方法(jdk7及其以前)
    - 默认方法(jdk8额外增加)
    - 静态方法(jdk8额外增加)
    - 私有方法(jdk9额外增加)
}
```

案例

```
public interface IA {
    // 常量 默认修饰符是 public static final 这3个修饰符可以省略不写
    public static final int NUM = 10;
    int NUM1 = 20;

    // 抽象方法 默认修饰符是 public abstract 这2个修饰符可以省略不写
    public abstract void method1();
    void method2();

    // 默认方法 默认修饰符是 public default public可以省略,default不可以省略
    public default void method3(){
        System.out.println("IA 接口默认方法");
```

```
    }

    // 静态方法 默认修饰符 public static public可以省略,static不可以省略
    public static void method4(){
        System.out.println("静态方法");
    }

    // 私有方法 修饰符 private   private不可以省略
    private void method5(){
        System.out.println("私有非静态方法");
    }

    private static void method6(){
        System.out.println("私有静态方法");
    }
}

public class Test {
    public static void main(String[] args) {
        /*
            接口的定义:
                public interface 接口名{
                     jdk7及其以前:  常量,抽象方法
                     jdk8: 额外增加默认方法和静态方法
                     jdk9及其以上: 额外增加了私有方法
                }
         */
        System.out.println(IA.NUM1);// 10
    }

    // 类中的默认方法,使用默认权限修饰符(空)
    void method(){

    }
}
```

小结

略

知识点--3.3 实现接口

目标

- 掌握什么是实现,以及如何实现接口

路径

- 实现概述
- 实现格式

讲解

实现概述

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。

实现格式

- 类可以实现一个接口，也可以同时实现多个接口。
    - 类实现接口后，必须重写接口中所有的抽象方法，否则该类必须是一个“抽象类”。

```
public interface IA{
    public void show1();
}
public interface IB{
    public void show2();
}
public class Zi implements IA ,IB{
    public void show1(){
    }
    public void show2(){
    }
}
```

- 类可以在“继承一个类”的同时，实现一个、多个接口；

```
public class Fu{}
public interface IA{}
public interface IB{}
public class Zi extends Fu implements IA,IB{//一定要先继承，后实现
}
```

小结

略

知识点--3.4接口中成员的访问特点

目标

- 掌握接口中成员访问特点

路径

- 接口中成员访问特点概述
- 案例演示

讲解

接口中成员访问特点概述

接口中成员的访问特点：
接口中的常量：主要是供接口直接使用
接口中的抽象方法：供实现类重写的
接口中的默认方法：供实现类继承的(实现类中可以直接调用,实现类对象也可以直接调用)
接口中的静态方法：只供接口直接调用,实现类继承不了
接口中的私有方法：只能在接口中直接调用,实现类继承不了

案例演示

接口

```
/**
 * @Author:pengzhilin
 * @Date: 2020/4/16 11:57
 */
public interface IA {
    //  接口中的常量：主要是供接口直接使用
    public static final int NUM = 10;

    // 接口中的抽象方法：供实现类重写的
    public abstract void method1();

    // 接口中的默认方法：供实现类继承使用(实现类中可以直接调用,实现类对象也可以直接调用)
    public default void method2(){
        System.out.println("默认方法method2");
        method4();
        method5();
    }

    // 接口中的静态方法：只供接口直接调用,实现类继承不了
    public static void method3(){
        System.out.println("静态方法method3");
        method5();
    }

    // 接口中的私有方法：只能在接口中直接调用,实现类继承不了
    private void method4(){// 只能在接口的默认方法中调用
        // 方法体
        method5();
    }

    private static void method5(){//
        // 方法体
    }
}
```

实现类：

```
/**
 * @Author:pengzhilin
 * @Date: 2020/4/16 11:59
 */
public class ImpA implements IA{

   /* @Override
    public void method2() {
```

```
    }*/

    @Override
    public void method1() {
        System.out.println("重写接口中的method1抽象方法");
    }
}
```

测试类:

```
/**
 * @Author:pengzhilin
 * @Date: 2020/4/16 11:54
 */
public class Test {
    public static void main(String[] args) {
        /*
            接口中成员的访问特点:
                接口中的常量: 主要是供接口直接使用
                接口中的抽象方法: 供实现类重写的
                接口中的默认方法: 供实现类继承的(实现类中可以直接调用,实现类对象也可以直接调用)
                接口中的静态方法: 只供接口直接调用,实现类继承不了
                接口中的私有方法: 只能在接口中直接调用,实现类继承不了
         */
        System.out.println(IA.NUM);// 10

        // 创建实现类对象,访问NUM常量
        ImpA ia = new ImpA();
        System.out.println(ia.NUM);// 10

        // 调用method2方法
        ia.method2();

        // 通过接口名调用接口中的静态方法
        IA.method3();
        //ia.method3();// 编译报错,
    }
}
```

小结

- 略

知识点--3.5 多实现时的几种冲突情况

目标

- 理解多实现时的几种冲突情况

路径

- 公有静态常量的冲突

- 公有抽象方法的冲突
- 公有默认方法的冲突
- 公有静态方法的冲突
- 私有方法的冲突

讲解

公有静态常量的冲突

- 实现类不继承冲突的变量

```
interface IA{
    public static final int a = 10;
    public static final int b= 20;
}
interface IB{
    public static final int a = 30;
}
class Zi implements IA,IB{
    //只继承了b，没有继承a，因为a冲突了
}

public class Demo {
    public static void main(String[] args) {
        Zi z = new Zi();
     //   System.out.println(z.a);//编译错误
        System.out.println(z.b);
    }
}
```

公有抽象方法的冲突

- 实现类只需要重写一个

```
interface IA{
    public void show();
}
interface IB{
    public void show();
}
class Zi implements IA,IB{
    @Override
    public void show() {//子类只需要重写一个show()即可
        System.out.println("子类的show()...");
    }
}
public class Demo {
    public static void main(String[] args) {
        Zi z = new Zi();
        z.show();
    }
}
```

公有默认方法的冲突

- 实现类必须重写一次最终版本

```
interface IA{
    public default void show(){
        System.out.println("IA");
    }
}
interface IB{
    public default void show(){
        System.out.println("IB");
    }
}
class Zi implements IA,IB{
    @Override
    public void show() {//必须重写一次的show()
        System.out.println("Zi的show()....");
    }
}
public class Demo {
    public static void main(String[] args) {
        Zi z = new Zi();
        z.show();
    }
}
```

公有静态方法的冲突

- 静态方法是直接属于接口的,不能被继承,所以不存在冲突

```
interface IA{
    public static  void show(){
        System.out.println("IA");
    }
}
interface IB{
    public static void show(){
        System.out.println("IB");
    }
}
class Zi implements IA,IB{

}
public class Demo {
    public static void main(String[] args) {
        Zi z = new Zi();
        z.show();//编译错误，show()不能被继承。
    }
}
```

私有方法的冲突

- 私有方法只能在本接口中直接使用,不存在冲突

小结

- 略

知识点--3.6 接口和接口的关系

目标

- 理解接口与接口之间的关系,以及接口继承时的冲突情况

路径

- 接口与接口之间的关系
- 接口多继承时的冲突情况
  - 公有静态常量的冲突
  - 公有抽象方法的冲突
  - 公有默认方法的冲突
  - 公有静态方法和私有方法的冲突

讲解

接口与接口之间的关系

- 接口可以“继承”自另一个“接口”，而且可以“多继承”。

```
interface IA{}
interface IB{}
interface IC extends IA,IB{//是“继承”，而且可以“多继承”
}
```

接口继承接口的冲突情况

公有静态常量的冲突

```
interface IA{
    public static final int a = 10;
    public static final int b = 30;
}
interface IB{
    public static final int a = 20;
}
interface IC extends IA,IB{//没有继承a
}
//测试：
main(){
    System.out.println(IC.a);//错误的
}
```

公有抽象方法冲突

```
interface IA{
    public void show();
}
interface IB{
    public void show();
}
interface IC extends IA,IB{//IC只继承了一个show()
}
class Zi implements IC{
    //重写一次show()
    public void show(){
    }
}
```

公有默认方法的冲突

```
interface IA{
    public default void d1(){
    }
}
interface IB{
    public default void d1(){
    }
}
interface IC extends IA,IB{//必须重写一次d1()
    public default  void d1(){
    }
}
```

公有静态方法和私有方法

- 不冲突,因为静态方法是直接属于接口的,只能使用接口直接访问,而私有方法只能在接口中访问,也没有冲突

小结

略

知识点--3.7 实现类继承父类又实现接口时的冲突

目标

- 实现类继承父类又实现接口时的冲突

路径

- 公有静态常量的冲突
- 公有抽象方法的冲突
- 公有默认方法的冲突
- 公有静态方法
- 私有方法的冲突

讲解

父类和接口的公有静态常量的冲突

```
class Fu{
    public static final int a = 10;
}
interface IA{
    public static final int a = 20;
}
class Zi extends Fu implements IA{//没有继承a变量
}
public class Demo {
    public static void main(String[] args) {
        System.out.println(Zi.a);//编译错误
    }
}
```

父类和接口的抽象方法冲突

```
abstract class Fu{
    public abstract void show();
}
interface IA{
    public void show();
}
class Zi extends Fu implements IA{// 必须重写

}
//测试：
main(){
    Zi z = new Zi();
    z.show();//a
}
```

父类和接口的公有默认方法的冲突

```
class Fu{
    public void show(){
        System.out.println("a");
    }
}
interface IA{
    public default void show(){
        System.out.println("b");
    }
}
class Zi extends Fu implements IA{
}
//测试：
main(){
    Zi z = new Zi();
    z.show();//a
}
```

父类和接口的公有静态方法

```
class Fu{
    public static void show(){
        System.out.println("fu...");
    }
}
interface IA{
    public static void show(){
        System.out.println("IA...");
    }
}
class Zi extends Fu implements IA{//只继承了"父类"的静态方法，没有继承接口的静态方法

}
public class Demo {
    public static void main(String[] args) {
        Zi.show();//fu…
    }
}
```

父类和接口的私有方法

- 不存在冲突

小结

略

实操--3.8 抽象类和接口的练习

需求:

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： --->父类 抽象类

行为：吼叫；吃饭； ----- 抽象类

缉毒犬： ---> 继承犬类

行为：吼叫；吃饭；缉毒；

缉毒接口:

缉毒的功能(抽象方法)

- 如果所有子类都有的功能: 通用功能(非抽象方法),非通用功能(抽象方法),定义到父类中
- 如果某个功能是一个类额外增加的,那么就可以把这个额外的功能定义到接口中,再这个类去实现

分析:

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。

- 额外的功能---> 在接口中定义,让实现类实现
- 共性的功能---> 在父类中定义,让子类继承

实现:

```
//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() {
    }
}
```

小结:

- 额外的功能---> 在接口中定义,让实现类实现
    - 如果可以确定的通用功能,使用默认方法
    - 如果不能确定的功能,使用抽象方法
- 共性的功能---> 在父类中定义,让子类继承
    - 如果可以确定的通用功能,使用默认方法
    - 如果不能确定的功能,使用抽象方法

第二章 多态

知识点-- 概述

目标:

- 了解什么是多态,以及形成多态的条件

路径:

- 引入
- 概念
- 形成多态的条件

讲解:

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，对于不同的对象具有多个不同表现形式。
- 程序中多态: 是指同一方法,对于不同的对象具有不同的实现.

前提条件【重点】

1. 继承或者实现【二选一】
2. 父类引用指向子类对象【格式体现】 `Fu fu = new Zi();`
3. 方法的重写【意义体现：不重写，无意义】

小结:

- **多态**： 是指同一行为，对于不同的对象具有多个不同表现形式。
- 条件:
    - 继承或者实现
    - 父类引用指向子类的对象
    - 方法的重写

知识点-- 实现多态

目标:

- 如何实现多态

路径:

- 多态的实现

讲解:

多态的体现：**父类的引用指向它的子类的对象**：

```
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。

```
class Animal{}
class Cat extends Animal{}
class Dog extends Animal{}
clsss Person{}
//测试类:
main(){
    Animal a1 = new Cat();
    Animal a2 = new Dog();
    Animal a3 = new Person();//编译错误，没有继承关系。
}
```

小结:

- 父类的引用指向子类的对象

知识点-- 多态时访问成员的特点

目标

- 掌握多态时访问成员的特点

路径:

- 多态时成员变量的访问特点
- 多态时成员方法的访问特点

讲解:

- 多态时成员变量的访问特点
  - 编译看左边,运行看左边
    - 简而言之:多态的情况下,访问的是父类的成员变量
- 多态时成员方法的访问特点
  - 非静态方法:编译看左边,运行看右边
    - 简而言之:编译的时候去父类中查找方法,运行的时候去子类中查找方法来执行
  - 静态方法:编译看左边,运行看左边
    - 简而言之:编译的时候去父类中查找方法,运行的时候去父类中查找方法来执行
- 注意:多态的情况下是无法访问子类独有的方法
- 演示代码:

```
public class Demo1 {
    public static void main(String[] args) {
        // 父类的引用指向子类的对象
        Animal anl1 = new Dog();
        // 访问非静态方法
        anl1.eat();

        // 访问成员变量num
        System.out.println(anl1.num);//10

        // 访问静态方法
        anl1.sleep();
```

```
        // 多态想要调用子类中独有的方法
        // anl1.lookHome(); 错误的,无法访问   多态的弊端:无法访问子类独有的方法

    }
}

public class Animal {

    int num = 10;

    public void eat(){
        System.out.println("吃东西...");
    }

    public static void sleep(){
        System.out.println("Animal类中的睡觉方法...");
    }

}

public class Dog extends Animal {

    int num = 20;

   // 重写
    public void eat() {
        System.out.println("狗吃骨头");
    }

    public static void sleep(){
        System.out.println("Dog类中的睡觉方法...");
    }

    public void lookHome(){
        System.out.println("狗正在看家...");
    }

}
```

小结:

略

知识点-- 多态的好处和弊端

目标:

- 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。但有好处也有弊端

步骤:

- 多态的好处和弊端

讲解:

- 好处
  - 提高了代码的扩展性
- 弊端
  - 多态的情况下，只能调用父类的共性内容，不能调用子类的特有内容。
- 示例代码

```
// 父类
public abstract class Animal {
    public abstract void eat();
}
// 子类
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void catchMouse(){
         System.out.println("猫抓老鼠");
    }
}
class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
}
```

定义测试类:

```
多态的好处:
public class Test {
    public static void main(String[] args) {
        // 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();
    }
}

// 多态的弊端============
public static void main(String[] args) {
        // 多态形式，创建对象
        Aniaml anl = new Cat();
        anl.eat();
        anl.catchMouse();// 编译报错，编译看父类(左边),父类中没有定义catchMouse方法
}
```

小结:

略

知识点-- 引用类型转换

目标:

- 向上转型与向下转型,instanceof关键字

步骤:

- 向上转型
- 向下转型
- instanceof关键字

讲解:

向上转型

- 子类类型向父类类型向上转换的过程，这个过程是默认的。

```
Aniaml anl = new Cat();
```

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。

```
Aniaml anl = new Cat();
Cat c = (Cat)anl;//向下转型
c.catchMouse();// 可以访问 子类独有的功能,解决多态的弊端
```

instanceof关键字

- 向下强转有风险，最好在转换前做一个验证：
- 格式:

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

if( anl instanceof Cat){//判断anl是否能转换为Cat类型，如果可以返回：true，否则返回：false
    Cat c = (Cat)anl;//安全转换
}
```

小结

略

知识点-- 多态的应用场景:

目标:

- 掌握多态在开发中的应用场景

路径:

- 变量多态 --- 不常见
- 形参多态 ---- 常见
- 返回值多态 --- 常见

讲解:

多态的几种应用场景:

```
定义一个Animal类,让Dog和Cat类继承Animal类:
public class Animal {
    public void eat(){
        System.out.println("吃东西...");
    }
}
public class Cat extends Animal {
    @Override
    public void eat() {
        System.out.println("猫吃鱼...");
    }
}

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

public class Demo1 {
    public static void main(String[] args) {
        /*
            多态的应用场景:
```

```
                1.变量多态
                2.形参多态
                3.返回值多态
         */
        // 1.变量多态
        Animal anl = new Dog();
        anl.eat();

        // 2.形参多态
        Dog dog = new Dog();
        invokeEat(dog);

        Cat cat = new Cat();
        invokeEat(cat);// 实参赋值给形参: Animal anl = new Cat();

        // 3.返回值多态
        Animal anl2 = getAnimal();// 返回值赋值给变量: Animal anl2 = new Dog()
    }

    //  3.返回值多态
    // 结论:如果方法的返回值类型为父类类型,那么就可以返回该父类对象以及其所有子类对象
    public static Animal getAnimal(){
//        return new Animal();
        return new Dog();
//        return new Cat();
    }

    // 形参多态: 当你调用invokeEat方法的时候,传入Animal类的子类对象
    // Animal anl = dog; ====>  Animal anl = new Dog();
    // 结论:如果方法的参数是父类类型,那么就可以接收所有该父类对象以及其所有子类对象
    // Object类:是java中所有类的父类,所以如果参数为Object类型,那么就可以传入一切类的对象
    public static void invokeEat(Animal anl){
        anl.eat();
    }

}
```

小结:

略

知识点-- 多态的几种表现形式

目标:

- 多态的几种表现形式

路径:

- 普通父类多态
- 抽象父类多态
- 父接口多态

讲解:

- 多态的表现形式:

- 普通父类多态

```
public class Fu{}
public class Zi extends Fu{}
public class Demo{
    public static void main(String[] args){
        Fu f = new Zi();//左边是一个“父类”
    }
}
```

- 抽象父类多态

```
public abstract class Fu{}
public class Zi extends Fu{}
public class Demo{
    public static void main(String[] args){
        Fu f = new Zi();//左边是一个“父类”
    }
}
```

- 父接口多态

```
public interface A{}
public class AImp implements A{}
public class Demo{
    public static void main(String[] args){
        A a = new AImp();
    }
}
```

小结:

略

第三章 内部类

知识点-- 内部类

目标:

- 内部类的概述

步骤:

- 什么是内部类
- 成员内部类的格式
- 成员内部类的访问特点

讲解:

什么是内部类

将一个类A定义在另一个类B里面，里面的那个类A就称为**内部类**，B则称为**外部类。**

成员内部类

- **成员内部类**：定义在**类中方法外**的类。

定义格式：

```
class 外部类 {
    class 内部类{

    }
}
```

在描述事物时，若一个事物内部还包含其他事物，就可以使用内部类这种结构。比如，汽车类`Car`中包含发动机类`Engine`，这时，`Engine`就可以使用内部类来描述，定义在成员位置。

代码举例：

```
class Car { //外部类
    class Engine { //内部类

    }
}
```

访问特点

- 内部类可以直接访问外部类的成员，包括私有成员。
- 外部类要访问内部类的成员，必须要建立内部类的对象。

创建内部类对象格式：

```
外部类名.内部类名 对象名 = new 外部类型().new 内部类型();
```

访问演示，代码如下：

```
public class Body {// 外部类

    // 成员变量
    private int numW = 10;

    int num = 100;

    // 成员方法
    public void methodW1(){
        System.out.println("外部类中的methodW1方法...");
    }

    public void methodW2(){
        System.out.println("外部类中的methodW2方法...");
        // 创建内部类对象
        Body.Heart bh = new Body().new Heart();
        // 访问内部类成员变量
        System.out.println("内部类成员变量numN:"+bh.numN);
        // 访问内部类成员方法
        bh.methodN2();
```

```
    }

    public class Heart{// 成员内部类
        // 成员变量
        int numN = 20;

        int num = 200;

        // 成员方法
        public void methodN(){
            System.out.println("内部类中的methodN方法...");
            // 访问外部类成员变量
            System.out.println("外部类成员变量:"+numW);
            // 访问外部类成员方法
            methodW1();
        }

        public void methodN2(){
            System.out.println("内部类中的methodN2方法...");
        }

        public void methodN3(){
            int num = 300;
            System.out.println("局部变量num:"+num);// 300
            System.out.println("内部类成员变量num:"+this.num);// 200
            System.out.println("外部类成员变量num:"+Body.this.num);// 100
        }
    }
}
public class Demo {
    public static void main(String[] args) {
        // 测试
        // 创建外部类对象,调用外部类的方法methodW2
        Body body = new Body();
        body.methodW2();

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

        // 创建内部类对象,调用内部类的methodN方法
        Body.Heart heart = new Body().new Heart();
        heart.methodN();

        System.out.println("======================");
        heart.methodN3();// 300  200  100
    }
}
```

小结:

内部类:将一个类A定义在另一个类B里面，里面的那个类A就称为内部类，B则称为外部类。
成员内部类的格式:

```
public class 外部类名{
     public class 内部类名{

    }
}
```

成员内部类的访问特点:

- 内部类可以直接访问外部类的成员，包括私有成员。
- 外部类要访问内部类的成员，必须要建立内部类的对象。

成员内部类的创建方式:

```
外部类名.内部类名 对象名 = new 外部类名().new 内部类名();
```

知识点-- 匿名内部类

目标:

- 匿名内部类

步骤:

- 匿名内部类的概述
- 匿名内部类的格式

讲解:

概述

- **匿名内部类**：是内部类的简化写法。它的本质是一个`带具体实现的` `父类或者父接口的` `匿名的` **子类对象。**

代码一

```
public abstract class Animal {
    public abstract void eat();
}
public class Dog extends Animal {
    @Override
    public void eat() {
        System.out.println("狗吃骨头...");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            - 匿名内部类的概述:本质就是继承了父类或者实现了接口的匿名子类的对象
            - 匿名内部类的格式:
                new 类名\接口名(){
                    方法重写
                };

            - 匿名内部类的作用: 为了简化代码,并没有什么特殊的功能

            需求: 调用Aniaml类的eat()方法
```

```
            1.创建一个子类继承Animal类
            2.在子类中重写eat方法
            3.创建子类对象
            4.使用子类对象调用eat方法

            想要调用抽象类中的方法,必须具备以上4步,那能不能减后呢? 可以 使用匿名内部类
         */
        Animal anl1 = new Dog();
        anl1.eat();

        Animal anl2 = new Animal() {
            @Override
            public void eat() {
                System.out.println("Animal子类的eat方法...");
            }
        };
        anl2.eat();

    }
}
```

代码二

```
public interface AInterface {
    void method();
}
public class AImp implements AInterface {
    @Override
    public void method() {
        System.out.println("AImp 实现类重写method方法....");
    }
}
public class Demo {
    public static void main(String[] args) {
        /*
             匿名内部类:
                本质是一个继承了父类的匿名子类的对象
                本质是一个实现了接口的匿名实现类的对象

              案例: A接口中有一个抽象方法method(),现在需要调用A接口中的method方法
              思路:
                1.创建一个实现类实现A接口
                2.重写A接口中的抽象方法method()
                3.创建实现类对象
                4.使用实现类对象调用method方法

              想要调用A接口中的method方法,按照传统方式,必须有以上4步,一步都不可少
              前面三步就是为了得到A接口的实现类对象

            现在: 匿名内部类可以表示一个接口的匿名实现类对象,所以,可以直接创建接口的匿名内部类来调用method方法即可
         */
        AInterface a = new AInterface(){
            @Override
            public void method() {
                System.out.println("匿名内部类方式重写method方法....");
            }
```

```
        };
        a.method();

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

        AInterface a2 = new AImp();
        a2.method();

        System.out.println("======================");
        AInterface a3 = new AInterface() {
            @Override
            public void method() {
                // 实现
            }
        };
    }
}
```

小结

略

第四章 引用类型使用小结

目标

- 引用类型: 数组,类,抽象类,接口

路径

- 类名作为方法参数和返回值
- 抽象类作为方法参数和返回值
- 接口作为方法参数和返回值
- 类作为成员变量
- 抽象类作为成员变量
- 接口作为成员变量

讲解

6.1 类名作为方法参数和返回值

```
public class Person {
    String name;// 姓名
    int age;// 年龄

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Test {
```

```
    public static void main(String[] args) {
        // 引用数据类型作为方法的参数传递的是地址值
        // 类名作为方法参数
        // 创建Person类对象
        Person p = new Person("张三",18);
        // 调用method1方法,传入Person类对象
        method1(p);
        System.out.println(p.name+","+p.age);// 张三,19

    }

    // 定义一个方法,Person类作为方法的参数
    public static void method1(Person p){
        p.age = 19;
    }
}

public class Test2 {
    public static void main(String[] args) {
        // 引用数据类型作为方法的返回值返回的是地址值
        // 类名作为方法返回值
        Person p1 = method2();
        System.out.println(p1.name+","+p1.age);// 李四,18
    }

    // 定义一个方法,Person类作为方法的返回值类型
    public static Person method2(){
        Person p = new Person("李四",18);
        return p;
    }

}
```

6.2 抽象类作为方法参数和返回值

- 抽象类作为形参：表示可以接收任何此抽象类的"子类对象"作为实参；
- 抽象类作为返回值：表示"此方法可以返回此抽象类的任何子类对象"；

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

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

public class Test {
    public static void main(String[] args) {
        // 结论: 方法的参数为抽象类,那么就需要传入该抽象类的子类对象
        // 结论: 方法的返回值类型为抽象类,那么就需要返回该抽象类的子类对象

        // 调用method方法,需要传入的是Animal类子类对象
        Dog dog = new Dog();
```

```
        method(dog);

        method(new Animal() {
            @Override
            public void eat() {
                System.out.println("吃...");
            }
        });
    }

    // 抽象类作为方法的参数
    public static void method(Animal anl){
        anl.eat();
    }

    // 抽象类作为方法的返回值
    public static Animal method2(){
        //return new Dog();
        return new Animal() {
            @Override
            public void eat() {
                System.out.println("吃...");
            }
        };
    }
}
```

6.3 接口作为方法参数和返回值

- 接口作为方法的形参：【同抽象类】
- 接口作为方法的返回值：【同抽象类】

```
public interface IA {
    void show();
}

public class Test {
    public static void main(String[] args) {
        // 结论：接口作为方法的参数,那么必须传入该接口的实现类对象
        // 结论：接口作为方法的返回值,那么必须返回该接口的实现类对象

        // 传入接口的实现类对象
        method1(new IA() {
            @Override
            public void show() {
                System.out.println("show...");
            }
        });
    }

    // 接口作为方法的参数
    public static void method1(IA ia){

    }

    // 接口作为方法的返回值
    public static IA method2(){
```

```
        // 返回接口的实现类对象
        return new IA() {
            @Override
            public void show() {

            }
        };
    }
}
```

6.4 类名作为成员变量

```
public class Person {
    // 姓名
    String name;// String 类 jdk提供
    // 身份证
    IdCard idCard;
}

public class IdCard {// 身份证
    String idNum;
}
public class Test {
    public static void main(String[] args) {
        // 创建Person对象
        Person p = new Person();
        // 创建身份证对象
        IdCard id = new IdCard();
        // 给身份证对象赋值一个身份证号码
        id.idNum = "401234567890";

        // 给p对象的name赋值一个名字
        p.name = "张三";
        // 给p对象的身份证赋值一个身份证对象
        p.idCard = id;

        // 需求: 取出p对象的身份证号码
        System.out.println(p.idCard.idNum);// 401234567890
    }
}
```

6.5 抽象类作为成员变量

- 抽象类作为成员变量——为此成员变量赋值时，可以是任何它的子类对象

```
public abstract class Animal {
}
public class Dog extends Animal {
}
public class Person {
    Animal anl;// 抽象类作为成员变量
}

public class Test {
    public static void main(String[] args) {
        // 结论: 抽象类作为成员变量,在赋值的时候只能赋该抽象类的子类对象
```

```
        Person p = new Person();
        p.anl = new Dog();
    }
}
```

6.6 接口作为成员变量

- 接口类型作为成员变量——【同抽象类】

```
interface Animal{

}

class Dog implements Animal{

}
class Person{
    Animal anl;
}
public class Test {
    public static void main(String[] args) {
        // 结论: 接口作为成员变量,赋值的时候需要赋该接口的实现类对象
        Person p = new Person();
        p.anl =  new Dog();
    }
}
```

小结

略

总结

```
- 能够写出接口的定义格式
    public interface 接口名{
        // 常量(jdk7及其以前)  使用public static final修饰, 这3个修饰符可以省略不写
        // 抽象方法(jdk7及其以前) 使用public abstract修饰,这2个修饰符可以省略不写
        // 默认方法(jdk8额外增加) 使用public default修饰,public可以省略,default不可以省
略
        // 静态方法(jdk8额外增加) 使用public static修饰,public可以省略,static不可省略
        // 私有方法(jdk9额外增加) 使用private修饰,private不可以省略
    }
- 能够写出接口的实现格式
    public class 实现类 implements 接口名{}
    public class 实现类 implements 接口名,接口名,...{}
    public class 实现类 extends 父类名 implements 接口名,接口名,...{}

- 能够说出接口中的成员特点
    常量(jdk7及其以前): 主要供接口直接使用
    抽象方法(jdk7及其以前): 供实现类重写的
    默认方法(jdk8额外增加): 供实现类对象直接调用,或者实现类重写
    静态方法(jdk8额外增加): 只供接口直接使用
    私有方法(jdk9额外增加): 只能在本接口中使用
```

- 能够说出多态的前提
  1.继承或者实现
  2.父类的引用指向子类对象 \ 父接口的引用指向实现类对象
  3.重写方法

- 能够写出多态的格式
  父类的引用指向子类对象 \ 父接口的引用指向实现类对象

- 能够理解多态向上转型和向下转型
  向上转型: 子类类型 向 父类类型转换的过程,这个过程是自动的
  父类类型 对象名 = 子类对象;

  向下转型: 父类类型 向 子类类型转换的过程,这个过程是强制的
  子类类型 对象名 = (子类类型)父类类型的变量;
  instanceof关键字 可以做类型判断
  if(变量 instanceof 数据类型){}
  如果变量指向的对象是属于后面的数据类型,那么返回true
  如果变量指向的对象不是属于后面的数据类型,那么返回false

  多态成员访问特点:
  除了非静态方法编译看父类,运行看子类,其余都是看父类

- 能够说出内部类概念
  概念; 一个类中定义在了另一个类里面,里面的那个类就是内部类,外面的类就是外部类
  成员内部类:
  访问特点:
  1.在成员内部类中可以访问外部类的一切成员(包含私有的)
  2.在外部类或者其他类中,要访问成员内部类的成员就需要创建成员内部类对象
  外部类名.内部类名 对象名 = new 外部类名().new 内部类名();
- 能够理解匿名内部类的编写格式
  概述: 本质就是一个类的匿名子类对象 或者是一个接口的匿名实现类对象
  作用: 简化代码
  格式:

```
new 类名\接口名(){
    重写抽象方法
};
```