day03 【接口、多态、内部类】

今日内容

- 接口----重要\掌握
  - 定义接口
  - 实现接口
  - 接口中成员访问特点
- 多态-----重要\掌握
  - 实现多态
  - 多态成员访问特点
  - 多态的好处和弊端
  - 解决弊端---引用类型转换
- 内部类
  - 匿名内部类---->重要\掌握

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{}-->.class

public interface 接口名{}->.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语言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)
  - 默认方法和静态方法(jdk8额外增加)
  - 私有方法(jdk9额外增加)
- 定义接口使用interface关键字---接口编译后产生class文件
- 接口不能创建对象,需要使用实现类实现接口(类似于继承),实现接口的类叫做实现类(子类)

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)
    // 静态方法(jdk8)
    // 私有方法(jdk9)
}
```

案例

```
package com.itheima.demo1_定义接口;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 9:05
 */
```

```java
public interface IA {
    // 常量(jdk7及其以前) 使用public static final关键字修饰,这三个关键字都可以省略
    public static final int NUM1 = 10;
    int NUM2 = 20;

    // 抽象方法(jdk7及其以前) 使用public abstract关键字修饰,这2个关键字都可以省略
    public abstract void method1();
    void method2();

    // 默认方法(jdk8) 使用public default关键字修饰,public可以省略,default不可以省略
    public default void method3(){
        System.out.println("默认方法 method3");
    }

    // 静态方法(jdk8) 使用public static关键字修饰,public可以省略,static不可以省略
    public static void method4(){
        System.out.println("静态方法 method4");
    }
    // 私有方法(jdk9) 使用private关键字修饰,private不可以省略
    private static void method5(){
        System.out.println("私有静态方法  method5");
    }

    private void method6(){
        System.out.println("私有非静态方法  method6");
    }
}
```

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

```
package com.itheima.demo3_接口中成员的访问特点;

/**
```

```
 * @Author: pengzhilin
 * @Date: 2020/9/8 9:38
 */
public interface IA {
    // 常量
    public static final int NUM = 10;

    // 抽象方法
    public abstract void method1();

    // 默认方法
    public default void method2(){
        //method4();
        //method5();
        System.out.println("IA 接口中的默认方法method2");
    }

    // 静态方法
    public static void method3(){
        //method5();
        System.out.println("IA 接口中的静态方法method3");
    }

    // 私有方法
    private void method4(){
        System.out.println("IA 接口中的私有方法method4");
    }

    private static void method5(){
        System.out.println("IA 接口中的私有方法method5");
    }
}


package com.itheima.demo3_接口中成员的访问特点;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 9:39
 */
public class Imp implements IA {
    // 重写接口的抽象方法
    @Override
    public void method1() {
        System.out.println("实现类重写IA接口中的抽象方法");
    }

    // 重写接口的默认方法
    @Override
    public void method2() {
        System.out.println("实现类重写IA接口中的默认方法");
    }

}

package com.itheima.demo3_接口中成员的访问特点;

/**
```

```
 * @Author: pengzhilin
 * @Date: 2020/9/8 9:37
 */
public class Test {
    public static void main(String[] args) {
        /*
            接口中成员的访问特点:
                常量:主要是供接口名直接访问
                抽象方法:就是供实现类重写
                默认方法:就是供实现类重写或者实现类对象直接调用
                静态方法: 只供接口名直接调用
                私有方法: 只能在本接口中调用

         */
        // 访问接口常量
        System.out.println(IA.NUM);// 10  推荐
        //System.out.println(Imp.NUM);// 10 不推荐 常量被实现类继承了

        // 创建实现类对象调用方法
        Imp imp = new Imp();

        // 访问抽象方法
        imp.method1();

        // 访问默认方法
        imp.method2();

        // 接口名访问静态方法
        IA.method3();
        //Imp.method3();// 编译报错,没有继承
    }
}
```

小结

- 接口中成员访问特点:
  - 常量:主要是供接口名直接访问
  - 抽象类:就是用来给实现类重写的
  - 默认方法:只供实现类重写或者实现类对象直接调用
  - 静态方法:只供接口名直接调用
  - 私有方法:只能在本接口中调用

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常量

```
package com.itheima.demo4_多实现时的几种冲突情况.demo1_公有静态常量的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 9:56
 */
interface A{
    public static final int NUM1 = 10;
}
interface B{
    public static final int NUM1 = 20;
    public static final int NUM2 = 30;
}
class Imp implements A,B{

}
public class Test {
    public static void main(String[] args) {
        /*
            公有静态常量的冲突: 如果多个接口中有相同的常量,那么实现类就无法继承
         */
        //System.out.println(Imp.NUM1);// 编译报错,无法访问
        System.out.println(Imp.NUM2);// 30
    }
}
```

公有抽象方法的冲突

- 实现类只需要重写一个

```
interface A{
    public abstract void method();
}
interface B{
    public abstract void method();
}
class Imp implements A,B{
    @Override
    public void method() {
        System.out.println("实现类重写");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            公有抽象方法的冲突:实现类只需要重写一个
         */
    }
```

```
}
```

公有默认方法的冲突

- 实现类必须重写一次最终版本

```
package com.itheima.demo4_多实现时的几种冲突情况.demo3_公有默认方法的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:03
 */
interface A{
    public default void method(){
        System.out.println("A 接口的默认方法method");
    }
}
interface B{
    public default void method(){
        System.out.println("B 接口的默认方法method");
    }
}
class Imp implements A,B{
    @Override
    public void method() {
        System.out.println("实现类重写的默认方法");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            公有默认方法的冲突:实现类必须重写一次最终版本
         */
        Imp imp = new Imp();
        imp.method();
    }
}
```

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

```
package com.itheima.demo4_多实现时的几种冲突情况.demo4_公有静态方法的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:07
 */
interface A{
    public static void method(){
        System.out.println("A接口的静态方法method");
    }
}
interface B{
    public static void method(){
```

```
        System.out.println("B接口的静态方法method");
    }
}
class Imp implements A,B{

}
public class Test {
    public static void main(String[] args) {
        /*
            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
         */
    }
}
```

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

多实现时的几种冲突情况:

- 公有静态常量的冲突:实现类不继承冲突的常量
- 公有抽象方法的冲突:实现类只需要重写一个
- 公有默认方法的冲突:实现类必须重写一次最终版本
- 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
- 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

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”
}
```

接口多继承接口的冲突情况

公有静态常量的冲突

- 子接口无法继承父接口中冲突的常量

```
package com.itheima.demo5_接口和接口的关系.demo1_公有静态常量的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:14
 */
interface A{
    public static final int NUM1 = 10;
}
interface B{
    public static final int NUM1 = 20;
    public static final int NUM2 = 30;
}
interface C extends A,B{

}
public class Test {
    public static void main(String[] args) {
        /*
            公有静态常量的冲突: 子接口无法继承父接口中冲突的常量
         */
        //System.out.println(C.NUM1);// 编译报错,说明无法继承
        System.out.println(C.NUM2);// 30
    }
}
```

公有抽象方法冲突

- 子接口只会继承一个有冲突的抽象方法

```
package com.itheima.demo5_接口和接口的关系.demo2_公有抽象方法的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:18
 */
interface A{
    public abstract void method();
}
interface B{
    public abstract void method();
}
interface C extends A,B{

}
```

```
class Imp implements C{
    @Override
    public void method() {
        System.out.println("实现接口的抽象方法");
    }
}

public class Test {
    public static void main(String[] args) {
        /*
            公有抽象方法的冲突:子接口只会继承一个有冲突的抽象方法
         */
        Imp imp = new Imp();
        imp.method();
    }
}
```

公有默认方法的冲突

```
package com.itheima.demo5_接口和接口的关系.demo3_公有默认方法的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:21
 */
interface A{
    public default void method(){
        System.out.println("A 接口中的默认方法method");
    }
}
interface B{
    public default void method(){
        System.out.println("B 接口中的默认方法method");
    }
}

interface C extends A,B{

    @Override
    public default void method() {
        System.out.println("重写父接口中的method方法");
    }
}

class Imp implements C{

}

public class Test {
    public static void main(String[] args) {
        /*
            公有默认方法的冲突:子接口中必须重写一次有冲突的默认方法
            面试题:
                实现类重写接口中的默认方法,不需要加default
                子接口重写父接口中的面容方法,必须加default
         */
```

```
        Imp imp = new Imp();
        imp.method();// 重写父接口中的method方法
    }
}
```

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,只能使用本接口直接访问,而私有方法只能在接口中访问,也没有冲突

```
package com.itheima.demo5_接口和接口的关系.demo4_公有静态方法的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:26
 */
interface A{
    public static void method(){
        System.out.println("A 接口的静态方法method");
    }
}
interface B{
    public static void method(){
        System.out.println("B 接口的静态方法method");
    }
}
interface C extends A,B{

}
public class Test {
    public static void main(String[] args) {
        /*
            公有静态方法的冲突: 不存在冲突,因为静态方法是直接属于接口的,只供本接口直接调用
         */
        //C.method();// 编译报错,因为没有继承
    }
}
```

小结

```
- 接口与接口之间的关系: 继承关系
                    单继承: A接口继承B接口
                    多继承: A接口同时继承B接口,C接口,...
                    多层继承: A接口继承B接口,B接口,继承C接口
                    格式:
                        public interface 接口名 extends 接口名1,接口名2,...{

                        }

- 接口多继承时的冲突情况
    - 公有静态常量的冲突:子接口无法继承父接口中冲突的常量
    - 公有抽象方法的冲突:子接口只会继承一个有冲突的抽象方法
```

- 公有默认方法的冲突:子接口中必须重写一次有冲突的默认方法(注意要加default)
- 公有静态方法和私有方法的冲突:
  不冲突,因为静态方法是直接属于接口的,只能使用本接口直接访问,而私有方法只能在接口中访问,也没有冲突

面试题:
实现类重写接口中的默认方法,不需要加default
子接口重写父接口中的默认方法,必须加default

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

- 子类无法继承有冲突的常量

```
package com.itheima.demo6_实现类继承父类又实现接口时的冲突.demo1_公有静态常量的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:43
 */
class Fu{
    public static final int NUM1 = 10;
    public static final int NUM2 = 100;
}
interface A{
    public static final int NUM1 = 20;

}
class Zi extends Fu implements A{

}
public class Test {
    public static void main(String[] args) {
        /*
            公有静态常量的冲突:子类无法继承有冲突的常量
         */
        //System.out.println(Zi.NUM1);// 编译报错
        System.out.println(Zi.NUM2);
```

```
    }
}
```

父类和接口的抽象方法冲突

```
abstract class Fu{
    public abstract void method();
}
interface A{
    public abstract void method();
}
class Zi extends Fu implements A{
    @Override
    public void method() {
        System.out.println("Zi 重写有冲突的抽象方法");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            公有抽象方法的冲突:子类必须重写一次有冲突的抽象方法
         */
        Zi zi = new Zi();
        zi.method();
    }
}
```

父类和接口的公有默认方法的冲突

- 优先访问父类的

```
package com.itheima.demo6_实现类继承父类又实现接口时的冲突.demo3_公有默认方法的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:49
 */
class Fu{
    public void method(){
        System.out.println("Fu 类中的默认方法method");
    }
}
interface A{
    public default void method(){
        System.out.println("A 接口中的默认方法method");
    }
}
class Zi extends Fu implements A{

}
public class Test {
    public static void main(String[] args) {
        /*
            公有默认方法的冲突:优先访问父类的
         */
        Zi zi = new Zi();
```

```
        zi.method();// Fu 类中的默认方法method
    }
}
```

父类和接口的公有静态方法

- 只会访问父类的静态方法

```
package com.itheima.demo6_实现类继承父类又实现接口时的冲突.demo4_公有静态方法的冲突;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 10:52
 */
class Fu{
    public static void method(){
        System.out.println("Fu 类中的静态方法method");
    }
}
interface A{
    public static void method(){
        System.out.println("A 接口中的静态方法method");
    }
}
class Zi extends Fu implements A{

}
public class Test {
    public static void main(String[] args) {
        /*
            公有静态方法的冲突:只会访问父类的静态方法
         */
        Zi.method();
    }
}
```

父类和接口的私有方法

- 不存在冲突

小结

实现类继承父类又实现接口时的冲突:

- 公有静态常量的冲突:子类无法继承有冲突的常量
- 公有抽象方法的冲突:子类必须重写一次有冲突的抽象方法
- 公有默认方法的冲突:优先访问父类的
- 公有静态方法的冲突:只会访问父类的静态方法
- 私有方法的冲突: 不存在冲突

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: ---抽象父类

行为:吼叫;吃饭;

缉毒犬:继承犬类,实现缉毒接口

行为:吼叫;吃饭;缉毒;

缉毒接口:

缉毒

- 如果一个父类中的某个方法,所有子类都有不同的实现,那么该方法就应该定义成抽象方法,所以该父类就是抽象类 (父类一般都是抽象类)
- 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:

```
package com.itheima.demo7_抽象类和接口的练习;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 11:17
 */
// 抽象父类
public abstract class Dog {
    public abstract void houJiao();
    public abstract void eat();
}


package com.itheima.demo7_抽象类和接口的练习;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 11:24
 */
public interface JiDu {
    public abstract void jiDu();
}
```

```java
package com.itheima.demo7_抽象类和接口的练习;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 11:18
 */
public class JiDuDog extends Dog implements JiDu{
    @Override
    public void houJiao() {
        System.out.println("缉毒犬找到了毒品,开始吼叫,汪汪汪....");
    }

    @Override
    public void eat() {
        System.out.println("缉毒之前,开始吃骨头...");
    }

    @Override
    public void jiDu() {
        System.out.println("吃完东西后,开始使用鼻子查找毒品....");
    }
}


package com.itheima.demo7_抽象类和接口的练习;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 11:17
 */
public class Test {
    public static void main(String[] args) {
        // 创建缉毒狗对象
        JiDuDog jd = new JiDuDog();
        jd.eat();
        jd.jiDu();
        jd.houJiao();
    }
}
```

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\接口引用指向实现类对象【格式体现】
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\接口引用指向实现类对象
    - 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。

```
package com.itheima.demo8_实现多态;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 11:42
 */
class Animal{
    public void eat(){
        System.out.println("吃东西");
    }
}

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

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

public class Test1 {
    public static void main(String[] args) {
        /*
            多态: 同一种行为,不同的事物具有不同的表现形态
            实现多态:
                1.继承或者实现
                2.父类引用指向子类对象\接口引用指向实现类对象
                3.方法重写
         */
        // 父类引用指向子类对象
        Animal anl = new Dog();// 多态
        anl.eat();// 狗吃骨头...

        Animal anl1 = new Cat();
        anl1.eat();// 猫吃鱼...
    }
}
```

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:

```
package com.itheima.demo9_多态时访问成员的特点;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 11:51
 */
class Animal{
    int num = 10;
    public void method1(){
        System.out.println("Animal 非静态method1方法");
    }
    public static void method2(){
        System.out.println("Animal 静态method2方法");
    }
}
class Dog extends Animal{
    int num = 20;

    public void method1(){
        System.out.println("Dog 非静态method1方法");
    }

    public static void method2(){
        System.out.println("Dog 静态method2方法");
    }
}

public class Test {
    public static void main(String[] args) {
        /*
            多态时访问成员的特点:
                成员变量:编译看父类,运行看父类(编译看左边,运行看左边)
```

```
                成员方法:
                    非静态方法:编译看父类,运行看子类(编译看左边,运行看右边)
                    静态方法: 编译看父类,运行看父类(编译看左边,运行看左边)
                结论:除了非静态方法是编译看父类,运行看子类,其余都是看父类
         */
        // 父类的引用指向子类的对象
        Animal anl = new Dog();
        System.out.println(anl.num);// 10

        anl.method1();// Dog 非静态method1方法

        anl.method2();// Animal 静态method2方法

    }
}
```

小结:

```
多态时访问成员的特点:
                成员变量:编译看父类,运行看父类(编译看左边,运行看左边)
                成员方法:
                    非静态方法:编译看父类,运行看子类(编译看左边,运行看右边)
                    静态方法: 编译看父类,运行看父类(编译看左边,运行看左边)
                结论:除了非静态方法是编译看父类,运行看子类,其余都是看父类
```

知识点-- 多态的形式

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

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 -----> 意义不大
- 形参多态----> 常用
- 返回值多态---> 常用

讲解:

多态的几种应用场景:

- 变量多态 -----> 意义不大

```
package com.itheima.demo12_多态的应用场景.demo1_变量多态;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 12:23
 */
class Animal{
    public void eat(){
        System.out.println("吃东西...");
    }
}
class Dog extends Animal{
```

```
    @Override
    public void eat() {
        System.out.println("狗吃骨头");
    }
}
class Cat extends Animal{
    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
}
public class Test {
    public static void main(String[] args) {
        // 变量多态: 父类类型的变量指向子类类型的对象
        // 如果变量的类型为父类类型,该变量就可以接收该父类类型的对象或者其所有子类对象
        Animal anl = new Dog();
        anl.eat();

        anl = new Cat();
        anl.eat();
    }
}
```

- 形参多态----> 常用

```
package com.itheima.demo12_多态的应用场景.demo2_形参多态;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 12:26
 */
class Animal{
    public void eat(){
        System.out.println("吃东西...");
    }
}
class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头");
    }
}
class Cat extends Animal{
    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
}
public class Test {
    public static void main(String[] args) {
        // 形参多态:参数类型为父类类型,该参数就可以接收该父类类型的对象或者其所有子类对象
        Dog d = new Dog();
        method(d);
```

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

        Cat c = new Cat();
        method(c);
    }

    // 需求: 定义一个方法,带有一个参数,该参数可以接收Animal类对象以及Animal类的所有子类对象
    // method(d); ====实参赋值给形参的时候==> Animal anl = new Dog();
    // method(c); ====实参赋值给形参的时候==> Animal anl = new Cat();
    public static void method(Animal anl){
        anl.eat();
    }

}
```

- 返回值多态---> 常用

```
package com.itheima.demo12_多态的应用场景.demo3_返回值多态;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 12:31
 */
class Animal{
    public void eat(){
        System.out.println("吃东西...");
    }
}
class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头");
    }
}
class Cat extends Animal{
    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
}
public class Test {
    public static void main(String[] args) {
        // 返回值多态:如果返回值类型为父类类型,那么就可以返回该父类类型的对象或者其所有子类对象
        Animal anl = method();
        anl.eat();
    }

    public static Animal method(){
        //return new Animal();
        //return new Dog();
        return new Cat();
    }
```

```
    public static Animal method1(){
        if (1==1){
            // 条件1成立
            return new Animal();
        }else if (2==2){
            // 条件2成立
            return new Dog();
        }else{
            // 否则
            return new Cat();
        }
    }
}
```

小结:

```
变量多态 -----> 意义不大:如果变量的类型为父类类型,该变量就可以接收该父类类型的对象或者其所有子类对象
形参多态----> 常用: 如果参数类型为父类类型,该参数就可以接收该父类类型的对象或者其所有子类对象
返回值多态---> 常用: 如果返回值类型为父类类型,那么就可以返回该父类类型的对象或者其所有子类对象
```

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

```
package com.itheima.demo13_多态的好处和弊端;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 14:45
 */
class Animal{
    public void eat(){
```

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

    // 特有的功能
    public void lookHome(){
        System.out.println("狗在看家...");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            多态的好处和弊端:
                好处:提高代码的复用性
                弊端:无法访问子类独有的方法或者成员变量,因为多态成员访问的特点是,编译看父类
         */
        // 父类的引用指向子类的对象
        Animal anl = new Dog();
        anl.eat();
        //anl.lookHome();// 编译报错,因为多态成员访问的特点是,编译看父类,而父类中没有子类独有的功能
    }
}
```

小结:

```
多态的好处和弊端:
        好处:提高代码的复用性
        弊端:无法访问子类独有的方法或者成员变量,因为多态成员访问的特点是,编译看父类
```

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
```

- 示例代码

```
package com.itheima.demo14_引用类型转换;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 14:52
 */
class Animal{
    public void eat(){
        System.out.println("吃东西...");
    }
}
class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头...");
    }

    // 特有的功能
    public void lookHome(){
        System.out.println("狗在看家...");
    }
}
class Cat extends Animal{
    @Override
    public void eat() {
        System.out.println("猫吃鱼...");
    }
}

class Person{}
public class Test {
    public static void main(String[] args) {
        /*
            引用类型转换:
                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\自动的。
                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\手动的。
                        格式: 子类类型 对象名 = (子类类型)父类类型的变量;
                        注意:
                            1.向下转型的时候:右边父类类型的变量一定要指向要转型的子类类型的对象

                            2.不管是向上转型还是向下转型,一定满足父子类关系或者实现关系
         */
        // 向上转型:
        Animal anl = new Dog();
```

```
        // 向下转型:
        Dog dog = (Dog)anl;

        System.out.println("====================================");
        // 注意:右边父类类型的变量一定要指向要转型的子类类型的对象
        //Animal anl1 = new Animal();
        //Dog d1 = (Dog)anl1;// 运行报错,类型转换异常ClassCastException


        //Animal anl2 = new Cat();
        //Dog d2 = (Dog)anl2;// 运行报错,类型转换异常ClassCastException

        //Animal anl3 = new Person();// 编译报错,因为Animal和Person不是父子关系
        //Animal anl3 = (Animal) new Person();// 编译报错,因为Animal和Person不是父子关系

    }
}
```

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
}
```

- 示例代码

```
package com.itheima.demo15_instanceof关键字;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 15:06
 */
class Animal{
    public void eat(){
        System.out.println("吃东西...");
    }
}
class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头...");
    }
```

```
    // 特有的功能
    public void lookHome(){
        System.out.println("狗在看家...");
    }
}
class Cat extends Animal{
    @Override
    public void eat() {
        System.out.println("猫吃鱼...");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            instanceof关键字:
                为什么要有instanceof关键字?
                    因为在引用类型转换的时候很容易出现类型转换异常,所以为了提高代码的严谨性,转型之前得先判断一下
                怎么使用instanceof关键字判断呢?
                    if(变量名 instanceof 数据类型){

                    }
                 执行:
                    判断前面变量指向的对象类型是否是后面的数据类型:
                        如果前面变量指向的对象类型是属于后面的数据类型,那么就返回true
                        如果前面变量指向的对象类型不是属于后面的数据类型,那么就返回false
         */
        // 向上转型
        Animal anl = new Cat();

        // 向下转型
        //Dog  d = (Dog)anl;// 运行的时候会出现类型转换异常
        // 先判断,再转型
        if (anl instanceof Dog){
            Dog  d = (Dog)anl;
        }

        System.out.println("正常结束");
    }
}
```

小结

```
 引用类型转换:
    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\自动的。
    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\手动的。
        格式: 子类类型 对象名 = (子类类型)父类类型的变量;
注意:
    1.向下转型的时候:右边父类类型的变量一定要指向要转型的子类类型的对象
    2.不管是向上转型还是向下转型,一定满足父子类关系或者实现关系

instanceof关键字:
    if(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。
```

解决多态的弊端

```
package com.itheima.demo16_解决多态的弊端;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 15:14
 */
class Animal{
    public void eat(){
        System.out.println("吃东西...");
    }
}
class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头...");
    }

    // 特有的功能
    public void lookHome(){
        System.out.println("狗在看家...");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            解决多态的弊端:
                弊端:无法访问子类独有的方法或者成员变量,因为多态成员访问的特点是,编译看父类
         */
        // 父类的引用指向子类的对象
        Animal anl = new Dog();// 向上转型
        anl.eat();// 狗吃骨头...

        //anl.lookHome();// 编译报错,因为多态成员访问的特点是,编译看父类,而父类中没有子类独有的功能

        // 先判断,后转型
        if (anl instanceof Dog){
            Dog d = (Dog)anl;// 向下转型
            d.lookHome();// 狗在看家...
```

```
        }

        System.out.println("正常结束");
    }
}
```

多态的应用场景综合案例

```
package com.itheima.demo17_多态的应用场景综合案例;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 15:19
 */
class Animal{
    public void eat(){
        System.out.println("吃东西...");
    }
}
class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头...");
    }

    // 特有的功能
    public void lookHome(){
        System.out.println("狗在看家...");
    }
}
class Cat extends Animal{
    @Override
    public void eat() {
        System.out.println("猫吃鱼...");
    }
    // 特有的功能
    public void catchMouse(){
        System.out.println("猫抓老鼠...");
    }
}
public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        method(d);

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

        Cat c = new Cat();
        method(c);
    }

    // 形参多态: 如果父类类型作为方法的形参类型,那么就可以接收该父类类型的对象或者其所有子类的对象
    public static void method(Animal anl){
        anl.eat();
```

```
        //anl.lookHome();// 编译报错
        // anl.catchMouse();// 编译报错
        if (anl instanceof Dog){
            Dog d = (Dog)anl;// 向下转型 Dog类型
            d.lookHome();
        }

        if (anl instanceof Cat){
            Cat c = (Cat)anl;// 向下转型 Cat类型
            c.catchMouse();
        }

    }
}
```

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：

```
package com.itheima.demo18_内部类;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 15:40
 */
// 外部类
public class Body {

    public void methodW1(){
        // 访问内部类的成员
        //Body.Heart bh = new Body().new Heart();
        Heart bh = new Heart();
        System.out.println(bh.numN);// 10
        bh.methodN1();// 内部类的成员方法 methodN1
    }

    // 成员变量
    private int numW = 100;

    // 成员方法
    private void methodW2(){
        System.out.println("外部类的成员方法 methodw2");
    }


    // 内部类
    public class Heart{
        // 成员变量
        int numN = 10;

        // 成员方法
        public void methodN1(){
            System.out.println("内部类的成员方法 methodN1");
        }

        public void methodN2(){
            // 访问外部类的成员
            System.out.println(numW);
```

```
            methodW2();
        }
    }


}

package com.itheima.demo18_内部类;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 15:38
 */
public class Test {
    public static void main(String[] args) {
        /*
            - 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，外面的那个B类则称为外部类。
            - 成员内部类的格式:
                public class 外部类{
                    public class 内部类{

                    }
                }
            - 成员内部类的访问特点:
                在其他类中,访问内部类的成员,得先创建内部类对象:
                    外部类名.内部类名 对象名 = new 外部类名().new 内部类名();
                在外部类中,访问内部类的成员,得先创建内部类对象:
                    外部类名.内部类名 对象名 = new 外部类名().new 内部类名();
                    内部类名 对象名 = new 内部类名();

                在内部类中,可以直接访问外部类的一切成员(包含私有的):

         */
        // 创建内部类的对象
        Body.Heart bh = new Body().new Heart();
        System.out.println(bh.numN);// 10
        bh.methodN1();// 内部类的成员方法 methodN1

        System.out.println("======================");
        // 创建外部类对象
        Body b = new Body();
        b.methodW1();

        System.out.println("======================");
        bh.methodN2();// 100     外部类的成员方法 methodW2

    }
}
```

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

```
package com.itheima.demo19_匿名内部类1;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 15:54
 */
abstract class Animal{
    public abstract void eat();
}
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:
                概述:本质其实就是一个类的匿名子类的对象
                作用:就是用来简化代码的,没有其他的功能
```

```
                格式:
                    new 类名(){
                        实现抽象方法
                    };
         */
        // 需求:调用Animal类的eat方法
        // 1.创建一个子类继承Animal类
        // 2.在子类中重写eat抽象方法
        // 3.创建子类对象
        // 4.使用子类对象调用eat方法
        Dog d = new Dog();// 创建Animal子类对象
        d.eat();// d---->是Animal类的子类的对象
        // 问题:以上4步一步都不能少,有点麻烦,是否可以简化代码?
        // 解决:匿名内部类可以简化代码,因为它可以不创建子类的情况下,直接得到一个类的子类对象

        System.out.println("==========================");
        // 创建Animal子类对象<=====>Animal类的匿名内部类
        // 父类的引用指向子类的对象
        Animal anl = new Animal() {
            @Override
            public void eat() {
                System.out.println("匿名内部类");
            }
        };// 是Animal类的子类的对象
        anl.eat();
    }
}
```

代码二

```
package com.itheima.demo20_匿名内部类2;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 16:08
 */
interface A{
    public abstract void show();
}
class Imp implements A{
    public void show(){
        System.out.println("实现类实现show方法");
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            匿名内部类:
                概述:本质是一个接口的匿名实现类的对象
                格式:
                    new 接口名(){
                        实现抽象方法
                    };
         */
        // 需求:调用A接口的show方法
        // 1.创建实现类实现A接口
```

```
        // 2.在实现类中重写show方法
        // 3.创建实现类对象
        // 4.使用实现类对象调用show方法
        Imp imp = new Imp();// imp就是接口的实现类的对象
        imp.show();

        System.out.println("============================");
        // 简化: 匿名内部类
        A a = new A() {
            @Override
            public void show() {
                System.out.println("匿名内部类");
            }
        };
        a.show();
    }
}
```

小结

```
对于类:
    概述:本质其实就是一个类的匿名子类的对象
    格式:
        new 类名(){
            实现抽象方法
        };
对于接口:
    概述:本质是一个接口的匿名实现类的对象
    格式:
        new 接口名(){
            实现抽象方法
        };
匿名内部类作用:就是用来简化代码的,没有其他的功能
使用场景:
    如果方法的形参类型为抽象类或者接口类型,那么为了简化代码,可以直接传入该抽象类或者接口的匿名内部类
```

- 补充

```
// 匿名子类的匿名对象
        new Imp().show();// 实现类的匿名对象调用show方法
        new A() {
            @Override
            public void show() {
                System.out.println("匿名内部类");
            }
        }.show();// 匿名实现类的匿名对象调用show方法
```

第四章 引用类型使用小结

目标

实际的开发中，引用类型的使用非常重要，也是非常普遍的。我们可以在理解基本类型的使用方式基础上，进一步去掌握引用类型的使用方式。基本类型可以作为成员变量、作为方法的参数、作为方法的返回值，那么当然引用类型也是可以的。在这我们使用两个例子，来学习一下。

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 不常见
- 接口作为成员变量 不常见

讲解

6.1 类名作为方法参数和返回值

```
package com.itheima.demo21_引用类型使用小结.demo1_类名作为方法参数和返回值;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 16:21
 */
class Person{
    public String name;
    public int age;

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

    public void show(){
        System.out.println(name+","+age);
    }
}
public class Test {
    public static void main(String[] args) {
        /*
            类名作为方法参数和返回值:
         */
        // 创建Person
        Person p = new Person("冰冰",18);
        method1(p);
        System.out.println("========================================");
        // 调用method2;
        Person person = method2(p);
        person.show();// 冰冰,20
    }

    // 类作为方法的参数类型
    public static void method1(Person p){
        p.show();// 冰冰,18
    }

    // 类作为方法的参数类型和返回值类型
```

```
    public static Person method2(Person p){
        p.age = 20;// 把age改为20
        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"；

```
package com.itheima.demo21_引用类型使用小结.demo2_抽象类作为方法参数和返回值;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 16:27
 */
abstract class Animal{
    public abstract void eat();
}
class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头...");
    }
}
public class Test {
    public static void main(String[] args) {
        // 调用method1,就得传入Animal抽象类的子类对象
        method1(new Dog());

        System.out.println("========================");
        // 调用method1,就得传入Animal抽象类的子类对象
        method1(new Animal() {
            @Override
            public void eat() {
                System.out.println("匿名内部类的方式...");
            }
        });

        System.out.println("========================");
        // 调用method2方法,会返回一个Animal类的子类对象
        //Animal anl = method2();
        Dog d = (Dog)method2();
    }

    // 抽象类作为方法参数类型
    public static void method1(Animal anl){
        anl.eat();
    }

    // 抽象类作为方法返回值类型
    public static Animal method2(){
        return new Dog();
    }
}
```

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】

```
package com.itheima.demo21_引用类型使用小结.demo3_接口作为方法参数和返回值;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 16:33
 */
interface A{
    void show();
}
class Imp implements A{
    public void show(){
        System.out.println("实现类的方式实现show方法");
    }
}
public class Test {
    public static void main(String[] args) {
        // 接口作为方法参数和返回值
        // 调用method1方法,就得传入A接口的实现类对象
        method1(new Imp());

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

        // 调用method1方法,就得传入A接口的匿名内部类
        method1(new A() {
            @Override
            public void show() {
                System.out.println("匿名内部类的方式实现show方法");
            }
        });

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

        // 调用method2方法,就会返回A接口的实现类对象
        //A a = method2();
        Imp imp = (Imp) method2();


    }

    // 接口作为方法参数
    public static void method1(A a){
        a.show();
    }

    // 接口作为方法返回值
    public static A method2(){
        return new Imp();
    }
}
```

6.4 类名作为成员变量

我们每个人(Person)都有一个身份证(IDCard)，为了表示这种关系，就需要在Person中定义一个IDCard的成员变量。定义Person类时，代码如下：

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

使用`String`类型表示姓名，`int`类型表示年龄。其实，`String`本身就是引用类型，我们往往忽略了它是引用类型。如果我们继续丰富这个类的定义，给`Person`增加身份证号，身份证签发机关等属性，我们将如何编写呢？这时候就需要编写一个IDCard类了

修改Person类：

```
package com.itheima.demo21_引用类型使用小结.demo4_类作为成员变量;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 16:47
 */
class Person{
    String name;// 引用数据类型定义成员变量  String类
    int age;// 基本类型定义成员变量
    IdCard idCard;

    public Person(String name, int age, IdCard idCard) {
        this.name = name;
        this.age = age;
        this.idCard = idCard;
    }
    // ...
}
class IdCard{
    String idNum;// 身份证号码
    String address;// 地址

    public IdCard(String idNum, String address) {
        this.idNum = idNum;
        this.address = address;
    }
    // ....
}
public class Test {
    public static void main(String[] args) {
        // 创建IdCard对象
        IdCard idCard = new IdCard("440330200010101919","广东省深圳市宝安区公安局");
        // 创建Person对象
        Person p = new Person("张三",18,idCard);

 System.out.println(p.name+","+p.age+","+p.idCard.idNum+","+p.idCard.address);//
java支持链式编程
    }
}
```

> 类作为成员变量时，对它进行赋值的操作，实际上，是赋给它该类的一个对象。同理，接口也是如此，例如我们笔记本案例中使用usb设备。在此我们只是通过小例子，让大家熟识下引用类型的用法，后续在咱们的就业班学习中，这种方式会使用的很多。

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

```
package com.itheima.demo21_引用类型使用小结.demo5_抽象类作为成员变量;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 16:55
 */
abstract class Pet{
    String name;

    public Pet(String name) {
        this.name = name;
    }
}
class Dog extends Pet{

    public Dog(String name) {
        super(name);
    }
}
class Person{
    String name;
    int age;
    Pet pet;

    public Person(String name, int age, Pet pet) {
        this.name = name;
        this.age = age;
        this.pet = pet;
    }
}
public class Test {
    public static void main(String[] args) {
        // 抽象类作为成员变量:传入抽象类的子类对象
        Pet pet = new Dog("旺财");
        Person p = new Person("张三",18,pet);
        System.out.println(p.name);
        System.out.println(p.age);
        System.out.println(p.pet.name);

    }
}
```

6.6 接口作为成员变量

- 接口类型作为成员变量——【同抽象类】

```
package com.itheima.demo21_引用类型使用小结.demo6_接口作为成员变量;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/8 16:59
 */
abstract interface Pet{

}
class Dog implements Pet{


}
class Person{
    String name;
    int age;
    Pet pet;

    public Person(String name, int age, Pet pet) {
        this.name = name;
        this.age = age;
        this.pet = pet;
    }
}
public class Test {
    public static void main(String[] args) {
        // 接口作为成员变量:传入接口的实现类对象
        Pet pet = new Dog();
        Person p = new Person("张三",18,pet);
        System.out.println(p.name);
        System.out.println(p.age);
        System.out.println(p.pet);

    }
}
```

英雄: name,皮肤,法术(接口)

小结

- 类名作为方法参数和返回值:可以直接传入该类的对象;返回该类的对象
- 抽象类作为方法参数和返回值:只能传入该类的子类对象;返回该类的子类对象
- 接口作为方法参数和返回值:只能传入该接口的实现类对象;返回该接口的实现类对象
  传递的都是地址值,返回的也是地址值

- 类作为成员变量 : 赋该类的对象
- 抽象类作为成员变量 ; 赋该类的子类对象
- 接口作为成员变量 : 赋该接口的实现类对象

总结

必须理解:
    - 接口----重要\掌握
      - 定义接口
      - 实现接口
      - 接口中成员访问特点
    - 多态-----重要\掌握
      - 实现多态
      - 多态成员访问特点
      - 多态的应用场景
      - 多态的好处和弊端
      - 解决弊端---引用类型转换
    - 内部类
      - 匿名内部类---->重要\掌握

- 能够写出接口的定义格式
```
public interface 接口名{
            常量
            抽象方法
            默认方法
            静态方法
            私有方法
}
```
- 能够写出接口的实现格式
```
public class 实现类名 implements 接口名1,接口名2,...{

}

public class 实现类名 extends 父类名 implements 接口名1,接口名2,...{

}
```

- 能够说出接口中的成员特点
    常量   :主要供接口名直接访问
    抽象方法:就是供实现类重写的
    默认方法:就是供实现类重写或者实现类对象直接调用
    静态方法:只供接口直接访问
    私有方法:只能在接口中访问

- 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的对象\接口的引用指向子类的对象
    3.方法的重写

- 能够写出多态的格式
    父类的引用指向子类的对象\接口的引用指向子类的对象

- 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
    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;
```
注意:
    1.向下转型的时候:右边父类类型的变量一定要指向要转型的子类类型的对象
    2.不管是向上转型还是向下转型,一定满足父子类关系或者实现关系

- 能够说出内部类概念
    一个类定义在另一个类的里面

- 能够理解匿名内部类的编写格式
  new 类名\接口名(){重写抽象方法};