day02【static、接口、多态、内部类】

今日内容

- static
- 接口
- 多态
- 内部类

教学目标

- [ ] 能够掌握static关键字修饰的变量调用方式
- [ ] 能够掌握static关键字修饰的方法调用方式
- [ ] 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 static关键字

1.1 概述

static是静态修饰符，一般修饰成员。被static修饰的成员属于类，不属于单个这个类的某个对象。static修饰的成员被多个对象共享。static修饰的成员属于类，但是会影响每一个对象。被static修饰的成员又叫类成员，不叫对象的成员。

1.2 定义和使用格式

类变量

当 `static` 修饰成员变量时，该变量称为**类变量**。该类的每个对象都**共享**同一个类变量的值。任何对象都可以更改该类变量的值，但也可以在不创建该类的对象的情况下对类变量进行操作。

- **类变量**：使用 static关键字修饰的成员变量。

定义格式：

```
static 数据类型 变量名;
```

举例：

```
static String room;
```

比如说，同学们来黑马程序员学校学习,那么我们所有同学的学校都是黑马程序员, 不因每个同学不同而不同。

所以，我们可以这样定义一个静态变量school，代码如下：

```
public class Student {
  private String name;
  private int age;
  // 类变量，记录学生学习的学校
  public static String school = "黑马程序员学校";

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

  // 打印属性值
  public void show() {
    System.out.println("name=" + name + ", age=" + age + ", shool=" + shool );
  }
}

public class StuDemo {
  public static void main(String[] args) {
    Student s1 = new Student("张三", 23);
    Student s2 = new Student("李四", 24);
    Student s3 = new Student("王五", 25);
    Student s4 = new Student("赵六", 26);

    s1.show(); // Student : name=张三, age=23, shool=黑马程序员学校
    s2.show(); // Student : name=李四, age=24, shool=黑马程序员学校
    s3.show(); // Student : name=王五, age=25, shool=黑马程序员学校
    s4.show(); // Student : name=赵六, age=26, shool=黑马程序员学校
  }
}
```

静态方法

当`static`修饰成员方法时，该方法称为**类方法** 。静态方法在声明中有`static`，建议使用类名来调用，而不需要创建类的对象。调用方式非常简单。

- **类方法**：使用 static关键字修饰的成员方法，习惯称为**静态方法。**

定义格式：

```
修饰符 static 返回值类型 方法名 (参数列表){
    // 执行语句
}
```

举例：在Student类中定义静态方法

```
public static void showNum() {
  System.out.println("num:" +  numberOfStudent);
}
```

- **静态方法调用的注意事项：**
  - 静态方法可以直接访问类变量和静态方法。
  - 静态方法**不能直接访问**普通成员变量或成员方法。成员方法可以直接访问类变量或静态方法。
  - 静态方法中，不能使用**this**关键字。

> 小贴士：静态方法只能访问静态成员。

调用格式

被static修饰的成员可以并且建议通过**类名直接访问**。虽然也可以通过对象名访问静态成员，原因即多个对象均属于一个类，共享使用同一个静态成员，但是不建议，会出现警告信息。

格式：

```
// 访问类变量
类名.类变量名;

// 调用静态方法
类名.静态方法名(参数);
```

调用演示，代码如下：

```
public class StuDemo2 {
  public static void main(String[] args) {
    // 访问类变量
    System.out.println(Student.numberOfStudent);
    // 调用静态方法
    Student.showNum();
  }
}
```

> 小结：static修饰的内容是属于类的，可以通过类名直接访问

第二章 接口

2.1 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。

2.2 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
    }
}
```

> 小结：定义接口时就是将定义类的class改成了interface，并且接口中的内容也有了一些变化。

2.3 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~~~
```

> 小结： 类实现接口使用的是implements关键字，并且一个普通类实现接口，必须要重写接口中的所有的抽象方法

2.4 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：

```
DDDDDDDDDDDD
```

> 小结： 一个类可以实现多个接口，多个接口之间使用逗号隔开即可。

2.5 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。

> 小结：接口和接口之间是继承的关系，而不是实现。一个接口可以继承多个接口。

2.6 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。

2.7 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)

第三章 多态

3.1 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】

3.2 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();
    }
}
```

> 多态在代码中的体现为父类引用指向子类对象。

3.3 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。

> 小结：多态的好处是提高程序的灵活性，扩展性

3.4 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");
    }
}

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

```
    public void watchHouse() {
        System.out.println("看家");
    }
}
```

定义测试类：

```
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
        Cat c = (Cat)a;
        c.catchMouse();         // 调用的是 Cat 的 catchMouse
    }
}
```

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：

```
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
        Dog d = (Dog)a;
        d.watchHouse();        // 调用的是 Dog 的 watchHouse 【运行报错】
    }
}
```

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：

```
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。
```

所以，转换前，我们最好先做一个判断，代码如下：

```
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
        }
    }
}
```

> 小结：多态向上转型是将子类类型转成父类类型，多态向下转型是将父类类型转成子类类型。

第四章 内部类

4.1 概述

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。**

4.2 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：

定义类：

```
public class Person {
    private  boolean live = true;
    class Heart {
        public void jump() {
            // 直接访问外部类成员
            if (live) {
                System.out.println("心脏在跳动");
            } else {
                System.out.println("心脏不跳了");
            }
        }
    }

    public boolean isLive() {
        return live;
    }

    public void setLive(boolean live) {
        this.live = live;
    }

}
```

定义测试类：

```
public class InnerDemo {
    public static void main(String[] args) {
        // 创建外部类对象
        Person p  = new Person();
        // 创建内部类对象
        Person.Heart heart = p.new Heart();
        // 调用内部类方法
        heart.jump();
        // 调用外部类方法
        p.setLive(false);
        // 调用内部类方法
        heart.jump();
    }
}
输出结果：
心脏在跳动
心脏不跳了
```

> 内部类仍然是一个独立的类，在编译之后会内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和$符号 。
>
> 比如，Person$Heart.class

> 小结：内部类是定义在一个类中的类。

4.3 匿名内部类

- **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。**

开发中，最常用到的内部类就是匿名内部类了。以接口举例，当你使用一个接口时，似乎得做如下几步操作，

1. 定义子类
2. 重写接口中的方法
3. 创建子类对象
4. 调用重写后的方法

我们的目的，最终只是为了调用方法，那么能不能简化一下，把以上四步合成一步呢？匿名内部类就是做这样的快捷方式。

前提

存在一个**类或者接口**，这里的**类可以是具体类也可以是抽象类。**

格式

```
new 父类名或者接口名(){
    // 方法重写
    @Override
    public void method() {
        // 执行语句
    }
};
```

使用方式

以接口为例，匿名内部类的使用，代码如下：

定义接口：

```
public abstract class FlyAble{
    public abstract void fly();
}
```

匿名内部类可以通过多态的形式接受

```
public class InnerDemo01 {
    public static void main(String[] args) {
        /*
            1.等号右边:定义并创建该接口的子类对象
            2.等号左边:是多态,接口类型引用指向子类对象
       */
        FlyAble  f = new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        };
    }
}
```

匿名内部类直接调用方法

```
public class InnerDemo02 {
    public static void main(String[] args) {
        /*
            1.等号右边:定义并创建该接口的子类对象
            2.等号左边:是多态,接口类型引用指向子类对象
       */
        new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        }.fly();
    }
}
```

方法的形式参数是接口或者抽象类时，也可以将匿名内部类作为参数传递

```
public class InnerDemo3 {
    public static void main(String[] args) {
        /*
        1.等号右边:定义并创建该接口的子类对象
        2.等号左边:是多态,接口类型引用指向子类对象
       */
        FlyAble  f = new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        };
        // 将f传递给showFly方法中
        showFly(f);
    }
    public static void showFly(FlyAble f) {
        f.fly();
    }
}
```

以上可以简化，代码如下：

```
public class InnerDemo2 {
    public static void main(String[] args) {
        /*
        创建匿名内部类,直接传递给showFly(FlyAble f)
        */
        showFly( new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        });
    }

    public static void showFly(FlyAble f) {
        f.fly();
    }
}
```

> 小结：匿名内部类做的事情是创建一个类的子类对象

第五章 引用类型使用小结

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。

5.1 引用类型作为方法参数和返回值

```
public class Person{
  public void eat(){
    System.out.println("吃饭");
  }
}
public class Test{
  public static void main(String[] args){
        method(new Person());
        Person p = createPerson();
  }

  //引用类型作为方法参数,在前面笔记本案例中我们也使用了接口类型作为方法参数
  pubic static void method(Person p){
       p.eat();
  }

  //引用类型作为返回值
  public static Person createPerson(){
        return new Person();
  }
}
```

5.2 引用类型作为成员变量

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

定义IDCard(身份证)类，添加身份证号，签发地等属性：

```
class IDCard {
    String idNum;//身份证号
    String authority;//签发地

    //getter和setter方法
    //...

    //toString方法
    //...
}
```

修改Person类：

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

    IDCard idCard;//表示自己的身份证信息

    //name和age的getter、setter方法
    //...

    public IDCard getIdCard() {
        return idCard;
    }

    public void setIdCard(IDCard idCard) {
        this.idCard = idCard;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", idCard=" + idCard +
                '}';
    }
}
```

测试类：

```
public class TestDemo {
    public static void main(String[] args) {
        //创建IDCard对象
        IDCard idCard = new IDCard();
        //设置身份证号
        idCard.setIdNum("110113201606066666");
        //设置签发地
        idCard.setAuthority("北京市顺义区公安局");

        //创建Person对象
        Person p = new Person();
        //设置姓名
        p.setName("小顺子");
        //设置年龄
```

```
        p.setAge(2);
        //设置身份证信息
        p.setIdCard(idCard);

        //打印小顺子的信息
        System.out.println(p);
    }
}
输出结果:
Person{name='小顺子', age=2, idCard=IDCard{idNum='110113201606066666',
authority='北京市顺义区公安局'}}
```

> 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。