# day02【static、接口、多态、内部类】

## 今日内容

- static
  - 概述
  - static的使用
  - 开发中的应用
- 接口
  - 概述
  - 接口的定义
  - 接口的基本实现
  - 接口的多实现
  - 接口的多继承[了解]
  - 抽象类与接口的练习
- 多态
  - 概述
  - 多态的实现
  - 访问成员的特点
  - 多态的表现形式
  - 多态的应用场景
  - 多态的好处和弊端
  - 引用类型转换
- 内部类
  - 成员内部类
  - 匿名内部类
- 引用类型的使用小结

## 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换和向下转换
- [ ] 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

## 知识点--static修饰变量

# 目标:

- 掌握static修饰变量的应用

# 路径:

- 概述
- static修饰成员位置变量
- 演示static修饰成员位置变量

# 讲解:

## 1.1.1概述

static是静态修饰符，表示静态的意思,可以修饰成员变量和成员方法以及代码块。

## 1.1.2 static修饰成员位置变量

static修饰成员位置变量，称为类变量。该类的每个对象都共享同一个类变量的值。任何对象都可更改该变量的值，且可在不创建该类对象情况下对该变量操作。

定义格式

```
修饰符 static 数据类型 变量名;
```

使用格式

```
类名.类变量名;
```

## 1.1.3演示类变量的定义和使用

需求:定义一个中国人类，利用类变量定义所有人的国籍。

//ChinesePerson类

```
public class ChinesePeople {
  public String name;
  public int age;
  public static String country;
  public ChinesePeople(){}
  public ChinesePeople(String name,int age){
  this.name=name;
  this.age=age;
  }
}
```

//测试类

```
public class Test {
  public static void main(String[] args) {
    ChinesePeople cp1 = new ChinesePeople("张三",18);
    cp1.country="中国";
    System.out.println("姓名:"+cp1.name+",年龄:"+cp1.age+",国家:"+cp1.country);
    System.out.println("-------------------------");
    ChinesePeople cp2 = new ChinesePeople();
```

```
    //应为country被所有类的对象共享，所以被定义的country一旦定义，其他的类也能获取该值。
    System.out.println("姓名:"+cp2.name+",年龄:"+cp2.age+",国家:"+cp2.country);
    ChinesePeople.country="大中国";
    ChinesePeople cp3 = new ChinesePeople("李四",22);
    System.out.println("姓名:"+cp1.name+",年龄:"+cp1.age+",国家:"+cp1.country);
    System.out.println("姓名:"+cp2.name+",年龄:"+cp2.age+",国家:"+cp2.country);
    System.out.println("姓名:"+cp3.name+",年龄:"+cp3.age+",国家:"+cp3.country);
  }
}
```

# 知识点--static修饰方法

## 目标:

- 掌握static修饰方法的应用

## 路径:

- static修饰成员位置方法
- 演示static修饰成员位置方法

## 讲解:

### 1.2.1static修饰成员位置方法

static修饰成员位置的方法，称为类方法。类方法可以且建议直接使用类名调用。

定义格式

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

使用格式:

```
类名.静态方法名(参数);
```

需求：通过Utils类定义一个静态方法，快速计算两个数的和

//Utils类代码

```
class Utils{
    public static int getSum(int num1,int num2) {
        return num1+num2;
    }
}
```

//测试类

```
public class Test {
  public static void main(String[] args) {
   int sum =Utils.getSum(1,2)
   System.out.println("sum");
  }
```

### 1.2.3类方法注意事项

- 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。

### 1.2.4演示类方法注意事项

需求：通过测试类演示静态方法注意事项

//测试类代码

```
public class Test{
    static int num=10;
     int num2=10;
    public static void main(String[] args) {
        //静态方法可以直接访问类变量和静态方法。
          System.out.println("num="+num);
          staticShow();
        //静态方法不能直接访问普通成员变量或成员方法。
          System.out.println("num2="+num2);
          show();
    }
    public static void staticShow() {
          System.out.println("我是静态方法");
        //静态方法中，不能使用this关键字。
         // this.num2=10;
    }
    public void show() {
        //成员方法可以直接访问类变量或静态方法
          System.out.println("num="+num);
          staticShow();
    }
}
```

> 小贴士：static修饰的内容是属于类的，可以通过类名直接访问

# 知识点--static修饰代码块

## 目标:

- 掌握static修饰代码块的应用

## 路径:

- static修饰成员位置方法
- 演示static修饰成员位置方法

## 讲解:

### 1.3.1 static修饰成员位置方法

`static` 修饰代码块`{}`：称为**静态代码块。**位于类中成员位置(类中方法外)，**随着类的加载而执行且执行一次**，优先于main方法和构造方法的执行。

定义格式

```
static {
    // 静态代码块
}
```

**构造代码块{}：位于类中成员位置(类中方法外)，随着对象的创建而执行且执行一次，优先于main方法和构造方法的执行。**

```
{
  // 构造代码块
}
```

### 1.3.2演示static修饰成员位置方法

需求：通过测试类，演示静态代码块执行

//测试类代码

```
public class Test {
    static {
        System.out.println("我是一个静态代码块");
    }

    public Test() {
        System.out.println("我是一个无参构造方法");
    }

    public static void main(String[] args) {
        System.out.println("我是一个main方法");
        new Test();
        new Test();
    }
}
```

# 知识点--static在开发中的应用

## 目标:

- 掌握static在开发中的应用

## 路径:

- static在开发中的应用场景
- 演示开发中static的应用

## 讲解:

### 1.4.1static在开发中的应用场景

开发项目中，通常需要一些“全局变量”或“全局方法”，这些全局变量和方法。

可以单独定义在一个类中，并声明为static(静态)的，方便通过类名访问，这样的类被称为工具类。

java中如Math类，Random类等也都是工具类

### 1.4.2 演示开发中static的应用

需求：在一个工具类中，定义一个π变量和获取数组最大值方法

//工具类代码

```
public class Utils {
  //定义全局变量
  public static double PI = 3.14;

  //定义全局方法
  public static int getMax(int[] arr) {
    int max = arr[0];
    //输入itar快速生产普通for循环
    for (int i = 1; i < arr.length; i++) {
      if (arr[i] > max) {
        max = arr[i];
      }
    }
    return max;
  }
}
```

//测试类代码

```
public class Test {
  public static void main(String[] args) {
    //  调用全局变量
    System.out.println("全局变量PI: " + Utils.PI);
    //  调用全局方法
    int[] arr = {1, 5, 8, 12, 0};
    int max = Utils.getMax(arr);
    System.out.println("最大值是: " + max);
  }
}
```

## 小结:

# 第二章 接口

## 知识点--概述

## 目标:

- 理解接口中的基本知识

## 路径:

- 接口的概述

## 讲解:

### 2.1接口的概述

什么是接口

- java的一种引用类型，是方法的集合。
- 如果说类中封装了成员变量、构造方法和成员方法，那么接口中封装了方法。

接口中的成员

- 没有静态代码块，没有成员变量，没有构造方法，只能定义常量。
- 有抽象方法（JDK 7及以前），默认方法(类似于类中的成员方法)和静态方法（JDK 8）。

接口的编译

- 接口的定义，它与定义类方式相似，但是使用 `interface` 关键字。
- 接口会被编译成.class文件，但它并不是类，而是另外一种引用数据类型。
- 接口中没有构造方法，不能创建对象，可以被实现(重写方法)，类似于继承，通过其实现类创建对象

## 小结:

# 知识点--定义格式

## 目标:

- 学会定义接口

## 路径:

- 接口的定义格式
- 接口成员的定义规则
- 演示接口的定义

## 讲解:

### 2.2.1接口的格式

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

### 2.2.2 接口成员的定义规则

```
静态常量格式：
    public static final 数据类型 变量名 = 值；
    public static final可以省略
```

```
抽象方法格式：
    public abstract 返回值 方法名(参数列表);
    `abstract` 可以省略，供实现类重写。
```

```
默认方法格式：
    权限修饰符 default 返回值 方法名(参数列表) {
        // 执行语句
    }
    `default` 不可省略，供实现类调用或者实现类重写。
```

```
静态方法格式：
    权限修饰符 static 返回值 方法名(参数列表) {
        // 执行语句
    }
    `default` 不可省略，只能通过接口名调用。
```

### 2.2.3演示接口的定义

需求:定义一个接口，演示接口的成员的定义

//接口代码

```
public interface MyInter {
    // final int NUM = 10;
    public static final int NUM = 10;
    // void abstractMethod();
    public abstract void abstractMethod();
    //void defaultMethod(){}//Interface abstract methods cannot have body
    public default void defaultMethod() {
        System.out.println("我是一个非抽象方法");
    }
    // void staticMethod(){}//nterface abstract methods cannot have body
    public static void staticMethod() {
        System.out.println("我是一个静态方法");
    }
}
```

测试类代码

```
public class Test {
    public static void main(String[] args) {
        System.out.println(MyInter.NUM);//证明了NUM是被static修饰的
        //MyInter.NUM=30;//Cannot assign a value to final variable 'NUM'证明了NUM被final修饰
    }
}
```

## 小结:

```

```

# 知识点--基本的实现

## 目标:

- 掌握接口的实现

## 路径:

- 概述
- 基本实现的格式
- 接口中成员的使用特点
- 演示接口的基本实现

## 讲解:

### 2.3.1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。

实现的动作类似继承，只是关键字不同，实现使用 `implements` 关键字。

实现情况分类：非抽象类实现接口、抽象类实现接口。

### 2.3.2基本实现的格式

非抽象类实现格式

```
public class 类名 implements 接口名 {
    // 重写接口中抽象方法【必须】
}
```

抽象类实现格式

```
public abstract class 类名 implements 接口名 {
    // 重写接口中默认方法【可选】
}
```

### 2.3.3接口中成员的使用特点

| | |
|---|---|
| 静态常量 | 通过所在接口名调用(推荐)或实现类直接访问。 |
| 抽象方法 | 实现类为非抽象类必须重写，为抽象类，则可以不实现 |
| 默认方法 | 实现类可以直接继承，可以重写，通过实现类的对象来调用。 |
| 静态方法 | 只能通过所在接口名调用 |

### 2.3.4演示接口的基本实现

需求:定义一个父接口演示基本实现中的格式及访问规则

//父接口代码

```
public interface MyInter {
    int num = 10;

    public void abstractMethod();

    public default void defaultMethod() {
        System.out.println("接口中的默认方法");
    }
    public default void defaultMethod2() {
        System.out.println("接口中的默认方法2");
    }
    public static void staticMethod() {
        System.out.println("接口中的静态方法");
    }
}
```

实现类代码

```
public class InterImpl implements MyInter {
    @Override
    public void abstractMethod() {
        System.out.println("子类重写抽象方法");
    }
    @Override
    public void defaultMethod2(){
        System.out.println("重写接口中的默认方法2");
    }
}
```

//抽象类代码

```
public abstract class AbstractImpl implements MyInter {
}
```

测试类代码

```
public class Test {
    public static void main(String[] args) {
        InterImpl ii = new InterImpl();
        System.out.println(ii.num);
        System.out.println(MyInter.num);
        ii.abstractMethod();
        ii.defaultMethod();
        ii.defaultMethod2();
        // fii.staticMethod();
        MyInter.staticMethod();
    }
}
```

## 小结:

```

```

# 知识点--接口的多实现

## 目标:

- 理解多实现的用法

## 路径:

- 概述
- 多实现格式
- 多实现的成员使用特点
- 演示接口的多实现

### 2.4.1多实现概述

实现类可以同时实现多个接口的，这叫做接口的**多实现。**

### 2.4.2多实现格式

```
public class implements 接口名1,接口名2... {
    // 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】
}
```

### 2.4.3多实现的同名成员使用特点

```
静态常量    只能通过所在接口名调用。
抽象方法    实现类为非抽象类必须重写1次(含同名)，抽象类，则可以不实现
默认方法    实现类是否抽象都必须重写1次。
静态方法    只能通过所在接口名调用
```

### 2.4.4演示接口的多实现

需求:定义两个个父接口演示基本实现中的格式及访问规则

//父接口A代码

```
public interface InterA {
    public static final int NUM = 10;

    public abstract void abstractMethod();

    public default void defaultMethod() {
        System.out.println("A接口中的defaultMethod");
    }

    public static void staticMethod() {
        System.out.println("A接口中的staticMethod");
    }
}
```

//父接口B代码

```
public interface InterB {
    public static final int NUM = 999;

    public abstract void abstractMethod();

    public default void defaultMethod() {
        System.out.println("B接口中的defaultMethod");
    }

    public static void staticMethod() {
        System.out.println("B接口中的staticMethod");
    }
}
```

//实现类代码

```
public class InterImpl implements InterA, InterB {
    public void abstractMethod() {
        System.out.println("重写父类中的同名抽象方法");
    }

    public void defaultMethod() {
        System.out.println("重写父类中的同名默认方法");
    }
}
```

//抽象类代码

```
public abstract class AbstractImpl implements InterA, InterB {
    public void defaultMethod() {
        System.out.println("重写父类中的同名默认方法");
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        InterImpl ii = new InterImpl();
        // System.out.println(idi.NUM);
        System.out.println(InterA.NUM);
        System.out.println(InterB.NUM);
        ii.abstractMethod();
        ii.defaultMethod();
        InterA.staticMethod();
        InterB.staticMethod();
    }
}
```

# 知识点--类继承并实现

## 目标:

- 理解多实现的用法

## 路径:

- 继承并实现格式
- 继承并实现同名成员使用特点
- 演示继承并实现

## 讲解:

### 2.5.1类继承并实现格式

类可以在继承一个类的同时，实现多个接口。

```
//[ ]表示可选操作
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】
}
```

### 2.5.2继承并实现同名成员使用特点

```
静态常量    同多实现。
抽象方法    同多实现。
父类与接口成员/默认方法相同，子类优先继承及使用类中的成员方法
父类与接口静态方法相同，子类优先使用父类中的静态方法，且可以不通过所在类名调用。
```

### 2.5.3演示继承并实现

需求：定义一个类，继承一个父类并实现一个接口，演示优先级的问题

//父接口代码

```
public interface MyInter {

    public default void defaultMethod() {
        System.out.println("接口中的默认方法");
    }

    public static void staticMethod() {
        System.out.println("接口中的静态方法");
    }
}
```

//父类代码

```
public class Fu {

    public void defaultMethod() {
        System.out.println("父类中的默认方法");
    }

    public static void staticMethod() {
        System.out.println("父类中的静态方法");
    }
}
```

//子类代码

```
public class Zi extends Fu implements InterDemo {

}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        Zi z = new Zi();
        z.defaultMethod();
        z.staticMethod();
    }
}
```

## 小结:

```

```

# 知识点--接口的多继承【了解】

## 目标:

- 了解接口的多继承

## 路径:

- 概述
- 多继承同名成员使用特点
- 接口多继承的格式
- 演示接口多继承

## 讲解:

### 2.5.1概述

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。

### 2.5.2多继承同名成员使用特点

> 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。

### 2.5.3多继承格式

```
权限修饰符 interface 子接口名 extends 父接口名1,父接口名2,...{}
```

### 2.5.4演示接口多继承

需求：定义两个父接口一个子接口演示接口的继承

//父接口A

```
public interface InterA {
    public default void defaultMethod() {
        System.out.println("A接口中的defaultMethod");
    }
}
```

//父接口B

```
public interface InterB {
    public default void defaultMethod() {
        System.out.println("B接口中的defaultMethod");
    }
}
```

//子接口

```
public interface InterC extends InterA, InterB {
    @Override
    default void defaultMethod() {
        System.out.println("子接口重写父接口中的默认方法");
    }
}
```

//实现类

```
public class InterCImpl implements InterC {
}
```

//测试类

```
public class Test {
    public static void main(String[] args) {
        InterCImpl idi=new InterCImpl();
        idi.defaultMethod();
    }
}
```

## 小结:

```

```

# 案例--抽象类和接口的练习

## 2.6.1需求:

针对下面的类，演示抽象类和接口的用法

```
犬：
行为：吼叫；吃饭；
缉毒犬：
行为：吼叫；吃饭；缉毒；
```

## 2.6.2分析:

吼叫和吃饭是所有狗都具备的功能，应该定义在父类中，即对于属性和行为的抽取放到父类中。
缉毒功能，除狗之外，还有缉毒猪，缉毒鼠等。属于狗可能拥有的额外功能，应该定义到接口中。

## 2.6.3实现:

//缉毒接口

```
public interface JiDuInter {
    public default  void jiDu(){
        System.out.println("搜索毒品");
    }
}
```

//犬类

```
public abstract class Dog {
    public void cry() {
        System.out.println("汪汪汪");
    }

    public abstract void eat();
}
```

//缉毒犬类

```
public class JiDuDog extends Dog implements JiDuInter {
    @Override
    public void eat() {
        System.out.println("吃军粮");
    }
}
```

//测试类

```
public class Test {
    public static void main(String[] args) {
        JiDuDog jdd = new JiDuDog();
        jdd.eat();
        jdd.jiDu();
        jdd.cry();
    }
}
```

> 小贴士：为什么有了抽象类还要有接口.
>
> > 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)

### 小结:

```

```

# 第三章 多态

## 知识点--概述

### 目标:

- 理解多态的含义和实现的前提

### 路径:

- 引入
- 定义
- 多态的前提

### 讲解:

#### 3.1.1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。

### 3.1.2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.

### 3.1.3多态的前提

1. 继承或者实现【二选一】
2. 父类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:

```

```

# 知识点--实现多态

## 目标:

- 掌握多态的书写

## 路径:

- 多态的体现格式
- 演示多态的使用

## 讲解:

### 3.2.1多态的体现格式

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

> 小贴士：父类类型：指实现类对象继承的父类类型，或者实现的父接口类型。

### 3.2.2演示多态的使用

需求：通过子父类类演示多态的使用

//父类动物类代码

```
public  class FU {
}
```

//实现类猫类代码

```
class Zi extends Fu {
}
```

//测试类

```
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Fu f = new Zi();
    }
}
```

> tips：多态在代码中的体现为父类引用指向实现类对象。

## 小结:

```

```

# 知识点--多态时访问成员的特点

## 目标

- 掌握多态使用的成员细节

## 路径:

- 多态时成员访问特点
- 演示多态时成员访问特点

## 讲解:

### 3.3.1多态时成员访问特点

成员变量

- 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

成员方法

- 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,运行的时候去实现类中查找方法来执行

静态方法:

- 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

### 3.3.2演示多态时成员访问特点

需求：定义子父类，演示多态时成员访问特点

//父类

```
public class Fu {
    public int num =11;

    public void method(){
        System.out.println("父类中的method方法");
    }

    public static void staticMethod(){
        System.out.println("父类中的staticMethod方法");
    }
}
```

//实现类

```
public class Zi extends Fu {
    public int num = 999;
    public int num2 = 20;

    @Override
    public void method() {
        System.out.println("子类重写父类的method方法");
    }

    public void show() {
        System.out.println("子类特有的show方法");
    }

    public static void staticMethod() {
        System.out.println("子类中的staticMethod方法");
    }

    public static void staticShow() {
        System.out.println("子类中的staticShow方法");
    }

}
```

//测试类

```
public class Test {
    public static void main(String[] args) {
        //多态的关系:父类引用指向实现类对象
        Fu f = new Zi();
        //成员变量访问特点
        System.out.println(f.num);//11
        //System.out.println(f.num2);//因为父类中没有num2，所以编译报错
        //成员方法访问特点
        f.method();
        //f.show();//因为父类中没有show方法，所以编译报错
        //静态方法访问特点
        f.staticMethod();
        //f.staticShow();//因为父类中没有staticShow方法，所以编译报错

    }
}
```

## 小结:

# 知识点--多态常见的3种表现形式

## 目标:

- 理解多态常见的3种表现形式

## 路径:

- 多态的3中表现形式
- 演示多态的3中表现形式

## 讲解:

### 3.4.1 多态的3中表现形式

```
普通父类引用指向子类对象
抽象父类引用指向子类对象
父接口引用指向子类对象
```

### 3.4.2 演示多态的3中表现形式

需求：分别定义一个普通父类，抽象父类，父接口，并创建对应实现类,演示3种多态

//父类代码

```
public class ClassFu {
    public int num =10;
    public void method(){
        System.out.println("父类中的method方法");
    }
}

public abstract class AbstractFu {
    public int num =20;
    public abstract void method();
}


public interface MyInter {
    public int num =30;
    public void method();
}
```

//实现类代码

```
public class ClassZi extends ClassFu {
    public int num =11;
    public void method(){
        System.out.println("普通类实现类重写method方法");
    }
```

```
}


public class AbstractZi extends AbstractFu {
    public int num = 21;
    @Override
    public void method() {
        System.out.println("抽象类实现类重写的method方法");
    }
}


public class InterImpl implements MyInter {
    public int num = 31;
    @Override
    public void method() {
        System.out.println("接口实现类重写method方法");
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //普通父类指向实现类对象
        ClassFu cz = new ClassZi();
        System.out.println(cz.num);
        cz.method();
        System.out.println("----------");
        //抽象父类指向实现类对象
        AbstractFu af = new AbstractZi();
        System.out.println(af.num);
        af.method();
        System.out.println("----------");
        //父接口指向实现类对象
        MyInter mi = new InterImpl();
        System.out.println(mi.num);
        mi.method();
    }
}
```

## 小结:

```

```

# 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:

- 多态的使用介绍
- 演示多态的使用

## 讲解:

### 3.5.1多态的使用介绍

变量多态的使用

```
父类名   变量名    = 实现类对象;
变量名.方法名();
```

形参多态的使用

```
修饰符 返回值 方法名(父类名 变量名){
变量名.方法名();
}
```

返回值多态的使用

```
修饰符 父类名 方法名(参数) {
return 实现类对象;
}
```

### 3.5.2演示多态的使用

需求：通过如下类演示变量多态使用

```
动物类：
    行为：吃
猫类：
    行为：吃
狗类：
    行为：吃
```

//父类代码

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

//子猫类

```
public class Cat extends Animal {

    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
}
```

//子狗类

```
public class Dog extends Animal {

    @Override
    public void eat() {
        System.out.println("狗吃骨头");
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //使用变量接收
        Animal a = new Cat();
        a.eat();
        a = new Dog();
        a.eat();
        System.out.println("---------");
        //父类引用作为形参--展示动物
        Cat c = new Cat();
        showCat(c);
        Dog d = new Dog();
        showDog(d);
        //我们发现，如果动物的种类很多，这样写下去，就很麻烦
        //所以，我们能不能只写一个方法，就对所有的动物进行展示
        showAnimal(c);
        showAnimal(d);
        System.out.println("---------");
        //父类引用作为返回值--获取动物
        Cat c2 = getCat();
        Dog d2 = getDog();
        //我们发现，如果动物的种类很多，这样写下去，就很麻烦
        //所以，我们能不能只写一个方法，就对所有的动物进行展示
        Animal a2 = getAnimal("猫");
        a2.eat();
        a2 = getAnimal("狗");
        a2.eat();
    }

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

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

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

    public static Cat getCat() {
        return new Cat();
    }
```

```
    public static Dog getDog() {
        return new Dog();
    }

    public static Animal getAnimal(String type) {
        if ("猫".equals(type)) {//"猫字符串放前面，是为了避免空指针异常
            return new Cat();
        } else if ("狗".equals(type)) {
            return new Dog();
        } else {
            System.out.println("您输入的类型有误");
            return null;
        }
    }
}
```

## 小结:

```

```

# 知识点--多态的好处和弊端

## 目标:

- 理解多态的好处与弊端，合理使用多态

## 步骤:

- 多态的好处和弊端介绍
- 演示多态的好处和弊端

## 讲解:

### 3.6.1多态的好处和弊端

**多态的好处**:可以将方法的参数定义为父类引用，使程序编写的更简单，提高程序的灵活性，扩展性

**多态的弊端**: 无法访问实现类的独有方法

### 3.6.2演示多态的好处和弊端

需求：通过如下类演示形参多态好处和弊端

```
Fu类
    行为:method
Zi类
    行为:method show
```

//Fu类代码

```
public class Fu{
    public void method(){
        System.out.println("父类中的method方法");
    }
}
```

//Zi类代码

```
public class Zi extends Fu{
    public void method(){
        System.out.println("子类中的method方法");
    }
    public void show(){
        System.out.println();
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        Fu f = new Zi();
        f.method();
        //f.show();//父类引用无法使用子类特有的方法
    }
}
```

## 小结:

```

```

# 知识点--引用类型转换

## 目标:

- 掌握引用类型的转换操作，灵活处理多态中的类型使用问题

## 步骤:

- 为什么要转换
- 引用类型转换的分类
- 演示类型转换及转换的异常问题

## 讲解:

### 3.7.1为什么要转换

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**实现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，想要调用实现类特有的方法，必须做向下转换。

### 3.7.2引用类型转换的分类

**向上转型**是实现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实现类对象时，便是向上转换。

格式：

```
父类类型   变量名 = new 实现类类型()   或 实现类对象引用;
```

**向下转型**：父类类型向实现类类型向下转换的过程，这个过程是强制的。

一个已经向上转换的实现类对象，将父类引用转为实现类引用，可以使用强制类型转换的格式，便是向下转换。

格式：

```
目标类型 变量名 = (目标类型) 父类变量名;
```

### 3.7.3演示引用类型转换

需求：根据如下类演示类型转换

```
Fu类
    行为:method
Zi类
    行为:method show
```

//Fu类代码

```
public class Fu{
    public void method(){
        System.out.println("父类中的method方法");
    }
}
```

//Zi类代码

```
public class Zi extends Fu{
    public void method(){
        System.out.println("子类中的method方法");
    }
    public void show(){
        System.out.println();
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //向上类型转换
        Fu f = new Zi();
        f.method();
        //f.show();//报错
        //向下类型转换
        Zi z = (Zi) a1;
        z.show();
    }
}
```

### 3.7.4演示类型转换的异常问题

需求:通过下述需求,演示类型转换中存在的异常问题

```
猫类:
    行为:吃,看家
狗类:
    行为:吃,抓耗子
```

//父类代码

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

//子猫类代

```
public class Cat extends Animal {

    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
    public void catchHouse(){
        System.out.println("抓耗子");
    }
}
```

//子狗类代码

```
public class Dog extends Animal {

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

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

//测试类

```
public class Test {
    public static void main(String[] args) {
        Cat c = new Cat();
        showAnimal(c);
        Dog d = new Dog();
        showAnimal(d);
    }

    public static void showAnimal(Animal a) {
        Cat c = (Cat) a;//java.lang.ClassCastException
        c.eat();
        c.catchMouse();
    }
}
```

> 小贴士：`ClassCastException`，类型转换异常，被转换的两个类之间不存在子父类关系。

### 3.7.3类型转换的异常问题总结及解决

转换的过程中，经常容易遇到一个异常`ClassCastException`

- 异常原因
  - 子类引用指向父类对象。
  - 转换对象不存在继承或实现关系
- 解决办法:instanceof关键字

```
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。
```

### 3.7.4 演示类型转换异常问题的解决

需求:使用instanceof解决上述问题

//动物类代码略

//狗类代码略

//猫类代码略

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        Cat c = new Cat();
        showAnimal(c);
        Dog d = new Dog();
        showAnimal(d);
    }

    public static void showAnimal(Animal a) {
        if (a instanceof Cat) {
            Cat c = (Cat) a;
            c.eat();
            c.catchMouse();
        } else if (a instanceof Dog) {
            Dog d = (Dog) a;
            d.eat();
            d.lookHouse();
        } else {
            System.out.println("类型不存在");
        }
    }
}
```

### 小结:

```

```

# 第四章 内部类

## 知识点--内部类

### 目标:

- 理解内部类及内部类的基本使用

### 步骤:

- 什么是内部类
- 成员内部类定义和介绍
- 演示成员内部类定义和使用
- 成员内部类的访问特点

### 讲解:

### 4.1.1什么是内部类

- 将类B定义在类A里面，类B就称为内部类，类A则称为类B的外部类。
- 内部类仍然是一个独立的类，在编译之后，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 。
- 内部类的分类
  - 成员内部类
  - 匿名内部类
  - 局部内部类(自行了解)

### 4.1.2成员内部类定义和使用

**成员内部类**： 定义在成员位置（**类中方法外**）的类。

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。

定义格式

```
class 外部类 {
    class 内部类{
    }
}
```

使用格式

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

### 4.1.3成员内部类访问特点

- 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。

### 4.1.4演示成员内部类定义和使用

需求：使用成员内部类的关系定义如下类

```
人类
    属性：是否存活
心脏：
    行为：跳动
```

//Person类代码

```
public class Person {
  private boolean isLive = true;

  public boolean getIsLive() {
    return isLive;
  }

  public void setIsLive(boolean isLive) {
    this.isLive = isLive;
  }
```

```
  //内部类
  class Heart {
    public void jump() {
      if (isLive) {
        System.out.println("心脏在调动");
      } else {
        System.out.println("心脏不动了");
      }
    }
  }
}
```

//测试类

```
public class Test {
    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 ph = p.new Heart();
        //调用内部类对象方法
        ph.jump();
        //  调用外部类方法
        p.setLive(false);
        //调用内部类方法
        ph.jump();
    }
}
```

## 小结:

```

```

# 知识点--匿名内部类

## 目标:

- 理解匿名内部类的本质，掌握用法

## 步骤:

- 概述
- 使用格式
- 使用场景
- 演示匿名内部类的使用

## 讲解:

### 4.2.1概述

**匿名内部类**： 它的本质是一个`带具体实现的` `父类或者父接口的` `匿名的` **实现类对象。**

匿名内部类的使用意义

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

- 创建自定义类，继承父类或实现接口
- 重写接口或父类中的方法
- 创建自定义类对象
- 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。**

### 4.2.2使用格式

```
new 父类名或者接口名(){
    // 方法重写
};
```

### 4.2.3使用场景

需要注意理解，匿名内部类本身就是一个对象，并且是指定的类的子类对象或接口的实现类对象。匿名内部类，与我们的对象一样，可以做以下的事情。

1. 通过多态的形式指向父类引用
2. 直接调用方法
3. 作为方法参数传递

### 4.2.4演示匿名内部类使用

需求：通过定义如下接口，并演示匿名内部类使用

```
飞行接口
    行为：飞行
```

//接口代码

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

//父类代码

```
public class FlyClass implements Flyable {
    @Override
    public void fly() {
        System.out.println("我要飞得更高...");
    }
}
```

//测试类

```
public class Test {
    public static void main(String[] args) {
        //正常步骤
```

```
        FlyClass f1 = new FlyClass();
        f1.fly();
        //使用变量接收匿名内部类
        Flyable f = new Flyable() {
            public void fly() {
                System.out.println("比第一次飞得更高");
            }
        };
        f.fly();
        //匿名内部类直接调用方法
        new Flyable() {
            public void fly() {
                System.out.println("比第二次飞得更高");
            }
        }.fly();
        //匿名内部类作为方法参数
        showFly(new Flyable() {
            public void fly() {
                System.out.println("比第三次飞得更高");
            }
        });
    }

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

### 小结:

```
```

# 第五章 引用类型使用小结

## 知识点--引用类型使用小结

### 目标:

- 掌握引用类型的使用方式

### 步骤:

- 演示引用类型作为方法参数和返回值类型
- 演示引用类型作为成员变量

### 讲解:

## 5.1 演示引用类型作为方法参数和返回值

需求：定义如下类，演示引用类型作为方法的参数和返回值类型

```
人类
属性：姓名
行为：吃饭
```

//人类代码

```
public class Person {
    public String name;
    public void eat(){
        System.out.println(name+"：正在吃饭");
    }
}
```

//测试类

```
public class Test {
    public static void main(String[] args) {
        //展示人类--引用类型作为形参
        Person p = new Person();
        p.name = "张三";
        showPerson(p);
        System.out.println("--------");
        //获取人类--引用类型作为返回值类型
        Person p2 = getPerson();
        p2.name = "李四";
        p2.eat();
    }

    public static void showPerson(Person p) {
        System.out.println(p.name);
        p.eat();
    }

    public static Person getPerson() {
        return new Person();
    }
}
```

## 5.2 演示引用类型作为成员变量

实际开发中，当我们定义类的时候，类中需要定义的成员属性，JDK并不能完全满足这些属性的定义。需要开发者根据需要灵活定义其他的类来满足该类的成员属性。

例如：空气类的成员属性，会包含水，二氧化碳，这些类都需要我们自己定义。

需求：根据如下定义一个英雄类

```
英雄类
属性：姓名，武器，法术
行为：展示英雄
```

英雄类代码

```
public class Hero {
  //姓名
  public String name;
  //武器
  public Weapon weapon;
  //法术
  public Magic magic;

  public Hero(String name, Weapon weapon, Magic magic) {
    this.name = name;
    this.weapon = weapon;
    this.magic = magic;
  }

  public void showHero() {
    System.out.println("名字：" + name);
    System.out.println("武器：" + weapon.name);
    System.out.println("法术：" + magic.name);
  }
}
```

//魔法类代码

```
public class Magic {
  public String name;
}
```

//武器类代码

```
public class Weapon {
  public String name;
}
```

//测试类代码

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

        //  创建武器
        Weapon weapon = new Weapon();
        weapon.name = "大宝剑";
        //  创建法术
        Magic magic = new Magic();
        magic.name = "大招";
        //  创建英雄
        Hero hero = new Hero("剑圣", weapon, magic);
        //展示英雄
        hero.showHero();
    }
}
```

> 小贴士：类作为成员变量，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。

**小结:**

```
```