

Java基础

ZCX

一、面向对象高级

1. final关键字

- final修饰的类，方法和变量
- final修饰的类成为**修饰类**，被称为最终类，无法被继承
- final修饰的方法称为**修饰方法**，不能被重写
- final修饰的变量称为**修饰变量**，有且仅能被赋值一次(final修饰的变量，必须被初始化)
 - 如public static final String SCHOOL_NAME = "XXX";(这就是常量了其实，因为无法被修改，常量的名称大写，下划线连接)
 - 注意：final修饰的基本类型的变量，变量存储的数据不能被改变。**final修饰的引用类型的变量，变量存储的地址不能被改变，但地址所指向的数据是可以被改变的**
- 常量：使用了static final修饰的变量
 - 作用：常用于记录系统的配置信息(比如公司项目中专门的常量包，方便编程，维护)
 - 程序编译后，常量会被宏替换，直接变成真正的字面量，不会占用内存

2. 单例类(设计模式)

- 什么是设计模式：解决某一类问题的最优的解法
- 设计模式有20多种，对应20多种软件开发中遇到的问题
- 单例设计模式：确保某个类只能创建一个对象。如虚拟机对象，任务管理器对象。
- 如何实现单例类：把类的构造器私有，定义一个类变量(静态变量)记录类的一个对象，定义一个类方法，返回对象。

```
public class a {  
    饿汉式单例类：拿对象时，对象早就创建好了  
    private a(){ //私有化构造器，确保无法对外创建对象  
    }  
    //定义一个静态变量，用于保存单例对象  
    public static final a instance= new a(); //变量名为 instance  
    //用final修饰后，就无法再次赋值，避免修改该对象为null导致错误  
}
```

```
public class test {  
    public static void main(String[] args) {  
        a a1 = a.instance; //因为是静态变量只加载一份  
        a a2 = a.instance; //a1,a2 都指向同一份  
    }  
}
```

```
}  
}
```

- 单例类有很多形式，饿汉式，懒汉式(拿对象时创建)，静态方法式，静态内部类式，枚举式
- 懒汉式单例类：用对象时，才开始创建对象
 - 把类的构造器私有，定义一个静态变量用于存储对象，提供一个静态方法，保证返回的是同一个对象

```
public class b {  
    private static b instance; //静态变量,一开始是null  
    private b()  
    {  
    }  
    public static b getInstance()  
    {  
        if(instance == null) //判断对象是否第一次创建  
        {  
            instance = new b();  
        }  
        return instance; //不是第一次创建，就返回之前创建的对象  
    }  
}  
  
public class test {  
    public static void main(String[] args) {  
        b b1 = b.getInstance(); //拿对象时，创建对象  
    }  
}
```

3. 枚举类

- 枚举类是一种特殊类(多例类，如果第一行就一个枚举对象，那么可视为单例类)
- 写法：**public enum 枚举类名{名称1, 名称2, ...;
其他成员 }**
- 特点：枚举类的第一行只能写枚举类的对象的名称，用逗号隔开，这些名称本质是常量，每个常量都记住了枚举类的一个对象

- ```
public enum a {
 X,Y,Z ; //只能罗列枚举对象的名称，本质是常量
} //枚举类都是最终类，每个对象都经过了final修饰，不可以被继承，枚举类都是继承
java.lang.Enum类。枚举类构造器是私有的，故枚举类对外不能创建对象
```

- 枚举类还有一些方法:

```
◦ a a1 = a.X;
 System.out.println(a1.name()); //打印枚举对象的名称
 System.out.println(a1.ordinal()); //打印枚举对象的索引
```

- **枚举类的应用场景**：适合做信息分类和标志

- 比如游戏运动中的上下左右键方向的移动，用枚举类枚举4个方向
- 比如小数的精度，用枚举类枚举4种精度，如四舍五入，直接截取等各个策略

```
public enum m { //枚举类名
 UP, DOWN, LEFT, RIGHT; // 枚举值四个方向
}

public class test2 {
 public static void main(String[] args)
 {
 move(m.UP); //调用方法
 }

 public static void move(m em){ //枚举类作为参数
 switch (em){ //switch语句
 case UP:
 System.out.println("向上");
 break;
 case DOWN:
 System.out.println("向下");
 break;
 case LEFT:
 System.out.println("向左");
 break;
 case RIGHT:
 System.out.println("向右");
 break;
 }
 }
}
```

#### 4. 抽象类

- 关键字**abstract**，可以修饰类和成员方法
- 抽象类：抽象类不能创建对象，只能被继承

- ```
public abstract class a { //抽象类
```

- 抽象方法：抽象方法没有方法体，方法体由子类实现

- ```
public abstract void show(); //抽象方法
```

- 注意事项、特点：**抽象类中不一定要有抽象方法，但有抽象方法的类必须是抽象类。**类有的成员：成员变量、方法、构造器、抽象类。**抽象类不能创建对象，仅作为一种特殊的父类，让子类继承并实现。**一个类继承抽象类，必须重写完抽象类的全部抽象方法，否则这个类也必须声明为抽象类。

- ```
public abstract class a { //抽象类a
    private int a;
    private String b;
    public a(){
    }
    public abstract void wo();
}
public class b extends a{ //类b继承抽象类a
    @Override //重写抽象方法
    public void wo()
    {
        System.out.println("wo");
    }
}
```

- **抽象类的好处：**为了更好的支持多态

- ```
public class test { //测试类
 public static void main(String[] args)
 {
 animal a = new cat();
 a.voice();
 animal b = new dog();
 b.voice();
 }
}
public abstract class animal { //抽象类animal
 public abstract void voice();
}
public class cat extends animal{ //类cat继承抽象类animal
 @Override
 public void voice() {
 System.out.println("喵喵喵");
 }
}
public class dog extends animal{ //类dog继承抽象类animal
 @Override
 public void voice()
 {
 System.out.println("汪汪汪");
 }
}
```

- **模版方法设计模式**：提供一个方法作为完成某类功能的模版，模版封装了每个实现步骤，但允许子类提供特定步骤的实现(提高代码的复用性，简化子类设计)
- 步骤：定义一个抽象类，在里面定义两个方法，一个是模版方法，把共同的步骤放里面去，另一个是抽象方法，不确定的实现步骤，交给子类实现
- 建议使用final修饰模版方法，让子类不能重写

```

public abstract class write { //抽象类write
 public final void write2(){
 System.out.println("写"); //模版方法，相同的步骤
 write1(); //抽象方法，不同的步骤
 System.out.println("写完"); //模版方法，相同的步骤
 }
 public abstract void write1();
}

public class teacher extends write{
 public void write1() //抽象方法，不同的步骤重写
 {
 System.out.println("老师写代码");
 }
}

public class teacher extends write{
 public void write1() //抽象方法，不同的步骤重写
 {
 System.out.println("老师写代码");
 }
}

public class test { //测试类
 public static void main(String[] args) {
 student s = new student();
 teacher t = new teacher();
 s.write2();
 t.write2();
 }
}

写
写student代码
写完
写
老师写代码
写完

```

## 5. 接口

- java提供了一个关键字 **interface**，用来定义接口
- 格式：**public interface 接口名{成员变量(默认是常量);成员方法(抽象方法);}**

```
public interface a {
 String NAME = "cxz"; //默认public static final不写
 void show(); //默认public abstract不写
}
```

- **接口不能创建对象**
- 接口是用来被类实现(implements)的，实现接口的类成为实现类，**一个类可以同时实现多个接口**
- 格式：**public class 实现类类名 implements 接口名1，接口2，接口3...{实现接口的成员变量;实现接口的成员方法;}**
- 实现类实现全部接口，需要重写接口中的所有抽象方法，不然需声明为抽象类

```
public class test { //测试类
 public static void main(String[] args) {
 System.out.println(a.NAME);
 c c1 = new c(); //创建实现类对象
 c1.play(); //调用实现类方法
 c1.show(); //调用实现类方法
 }
}

public interface a { //接口a
 String NAME = "cxz"; //默认public static final不写
 void show(); //默认public abstract不写
}

public interface b { //接口b
 String NAME="cxz";
 void play();
}

public class c implements a, b { //类c实现接口a，b
 public void show(){ //实现接口a的show方法
 System.out.println("show");
 }
 public void play(){ //实现接口b的play方法
 System.out.println("play");
 }
}
```

- **接口的好处**:弥补了类单继承的不足，一个类可以实现多个接口，是类的角色更多，功能更强大。让程序可以面向接口编程，可以让程序员灵活方便的切换各种业务实现，更利于程序的解耦合
- ```
class student extends people implements driver,teacher{ //类student继承people类，同时实现driver和teacher接口
```

- 综合小案例：设计一个班级学生的信息管理模块，学生的数据有姓名，性别，成绩。功能1：要求打印出全部学生的信息，功能2：要求打印出全班的平均成绩。(以上业务的实现有多套方案，要求系统可以灵活的切换)

```
public class test { //测试类
    public static void main(String[] args) {
        student[] students = new student[10]; //创建10个学生对象
        students[0] = new student("张三", "男",100);
        students[1] = new student("张三1", "男",10);
        students[2] = new student("张三2", "女",20);
        students[3] = new student("张三3", "男",30);
        students[4] = new student("张三4", "男",40);
        students[5] = new student("张三5", "女",50);
        students[6] = new student("张三6", "女",60);
        students[7] = new student("张三7", "男",70);
        students[8] = new student("张三8", "女",80);
        students[9] = new student("张三9", "男",90);
        classdatainterimpl2 impl1 = new classdatainterimpl2(students); //创建实现类对象,这个完全可以无缝换成classdatainterimpl1，后面的代码都不变，实现了系统灵活切换
        impl1.printallstudentinfos();
        impl1.printaveragestudentinfos();
    }
}

public interface classdatainter { //接口classdatainter
    void printallstudentinfos(); //打印全部学生的信息
    void printaveragestudentinfos(); //打印平均成绩
}

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
@Data
@AllArgsConstructor
@NoArgsConstructor
public class student { //学生类
    private String name;
    private String sex;
    private double score;
}

public class classdatainterimpl1 implements classdatainter{ //实现类classdatainterimpl1
    private student[] students; //声明学生数组
    public classdatainterimpl1(student[] students) {
```

```
this.students = students; //导入学生数组
}
@Override //重写接口方法
public void printallstudentinfos() { //打印全部学生的信息
for (int i = 0; i < students.length; i++) {
System.out.println(students[i]);
}
}
@Override //重写接口方法
public void printaveragestudentinfos() { //打印平均成绩
double sum = 0;
for (int i = 0; i < students.length; i++) {
sum += students[i].getScore();
}
System.out.println("平均分是： " + sum / students.length);
}
}
public class classdatainterimpl2 implements classdatainter{ //实现类classdatainterimpl2
private student[] students;
public classdatainterimpl2(student[] students) {
this.students = students;
}
@Override
public void printallstudentinfos() {
//打印所有信息，以及打出学生是男生的人数
int count = 0;
for (int i = 0; i < students.length; i++) {
System.out.println(students[i]);
if (students[i].getSex().equals("男")) {
count++;
}
}
System.out.println("男生人数是： " + count);
}
@Override
public void printaveragestudentinfos() {
//打印平均分，并输出男生的平均分
double sum = 0;
for (int i = 0; i < students.length; i++) {
sum += students[i].getScore();
```

```

}
System.out.println("平均分是：" + sum / students.length);
double sum1 = 0;
int count = 0;
for (int i = 0; i < students.length; i++) {
    if (students[i].getSex().equals("男")) {
        sum1 += students[i].getScore();
        count++;
    }
}
System.out.println("男生平均分是：" + sum1 / count);
}
}

```

- JDK8开始，接口新增的三种方法:

- ```

public class test { //测试类
 public static void main(String[] args) {
 b b1= new b(); //创建实现类对象
 b1.print(); //调用默认方法
 a.print3(); //用当前接口名来调用静态方法，不能用子类名在调
 }
}

public interface a { //接口a
 //默认方法
 //能定义普通方法，必须加default修饰，默认会用public修饰
 //只能找实现类调用，因为接口没有对象
 default void print(){
 System.out.println("接口的默认方法");
 print2();
 }
 //私有方法
 //私有的实例方法，不能用实现类对象调用，因为是私有的
 //所以只能在该接口中的其他实例方法中调用
 //比如在上面的print方法中调用
 private void print2(){
 System.out.println("接口的私有方法");
 }
 //静态方法
 //默认用public修饰，只能使用当前接口名调用
 static void print3(){
 System.out.println("接口的静态方法");
 }
}

```

```
}
}
public class b implements a{ //实现类b
}
```

- 上述基本上自己没有用的必要，仅做了解。不过用处就是增强了接口的能力，更便于项目的扩展和维护(比如为了在接口增加一个新功能，只需要在接口中增加一个方法，不需要上百个实现类全部报错修改了)
- **接口的注意事项：**
  - 接口与接口可以多继承，一个接口可以同时继承多个接口。
  - `public interface a extends b,c{ //接口a继承了接口b和c }`
  - 一个接口继承多个接口，如果多个接口中存在**方法签名冲突**，则此时不支持多继承，也不支持多实现。
  - 比如一个接口有`void show();`另一个接口有`String show();`则不支持多继承，也不支持多实现  
但是如果两个接口中含有的都是`void show();`方法，则此时可以多实现和继承
  - 一个类继承了父类，又同时实现了接口，如果父类中和接口有同名的默认方法，实现类会优先调用父类的。
  - 一个类实现了多接口，如果多个接口中存在同名的默认方法，可以不冲突，这个类重写该方法即可
- **接口与抽象类的区别**
  - 相同点：
    - 都是抽象形式，都可以有抽象方法，不能创建对象。
    - 都是派生子类形式，抽象类是被子类继承，接口是被子类实现。
    - 继承抽象类和实现接口都必须重写全部的抽象方法，否则会报错或直接成为抽象类。
    - 都能支持多态，能够实现解耦合
  - 不同点：
    - 抽象类可以定义类的全部普通成员，接口只能定义常量、抽象方法，和那三个方法
    - 抽象类只能被单继承，而接口可以多实现和继承
    - 抽象类体现模版思想，更利于做父类实现代码的复用性，接口体现解耦合性，更利于做功能的解耦合，解耦合性更强更灵活