OOP思维导图

一、对象和类

1.面向对象的程序设计

1.1 抽象的数据类型

- 将不同类型的数据的集合组成的整体用来描述一种新的事物。
  - 人： 包含了 (String name , int age ,char sex ,double salary) 4中不同类型的数据组成的整体
  - 学生 : 包含了(String name, double score,String grade) 也是不同类型的数据组成的整体
- 思考 : 人把大象装进冰箱分几步?
  - 人　数据类型有(String name ,int age ....) 功能(人装大象　打开冰箱门　关上冰箱门)行为
  - 大象 数据类型(double weight ,String name)　行为(被人拿起，装进冰箱)
  - 冰箱 数据类型(String singe)　行为(关门，开门)
- 上面所出现的人、学生、大象、冰箱都是一种抽象数据类型----类!
- 类是Java语言最基本单位。

1.2 什么是类

- 类的成员 :
  - 属性(成员变量)
  - 行为(方法、函数)

```
public class Person{//该类只定义了4个成员变量

    //成员变量
    String name;
    int age;
    char sex;
    double salary;
}
```

1.3 类的转化过程

- 打印学生的第一个方法

```
    /*
     * 打印学生信息的方法
     *  姓名、年龄、性别、学号
     */
    public static void sudentInfo(String name,int age,char sex,double
number) {
        System.out.println("---------------");
        System.out.println("姓名: " + name);
        System.out.println("年龄:" + age);
        System.out.println("性别: " + sex);
        System.out.println("学号:" + number);
    }
```

  - 参数中多种不同类型的数据，我们考虑采用抽象数据类型-类来替换参数类型

- 打印学生信息的第二种方式

```
    /*
     * 打印学生信息的方法
     *  姓名、年龄、性别、学号
     */
    public static void sudentInfolijian(Student student) {
        System.out.println("---------------");
        System.out.println("姓名: " + student.name);
        System.out.println("年龄:" + student.age);
        System.out.println("性别: " + student.sex);
        System.out.println("学号:" + student.number);
    }
    public class Student {
        String name;
        int age;
        char sex;
        double number;

    }
```

  - 打印学生信息方法只能针对Student数据操作，是属于Student自身方法，因此没有实现数据与操作数据的行为统一

- 打印学生信息的第三种方法

```
public class Student {

    String name;
    int age;
    char sex;
    double number;

    public void sudentInfo() {
        System.out.println("---------------");
        System.out.println("姓名: " + name);
        System.out.println("年龄:" + age);
        System.out.println("性别: " + sex);
        System.out.println("学号:" + number);
    }
}
public static void main(String[] args) {
```

```
        Student stu = new Student();
        stu.sudentInfo();
}
```

    - Student打印学生的信息方法只针对与Student操作，是Student内部的方法。
    - 类的组成：属性(数据本身)和方法(操作数据的行为)
- 总结:
    - 类是一种抽象的数据类型
    - OOP(Object Oriented Programming)面向对象编程实际是分类型思想。将过程种出现的数据泛化称为类。
    - 类的组成:属性和方法
    - main方法也属于类的方法

2.定义一个类

2.1定义类的成员变量

- 用于描述该类型对象共同的数据结构

```
public class Person{
    //成员变量   数据类型  变量名称
    String name;
    int age;
    ...
}
```

- 成员变量如果不赋初始值时，则系统会提供默认的初始值
    - 整型(byte short int long): 默认值为0
    - 浮点型(float double): 默认为0.0
    - 字符型(char):默认为空字符
    - 布尔型(boolean):默认为false

2.2定义类的成员方法

- 用于描述对象的行为，封装对象的功能。

```
public class Person {
    //成员方法     ！！！main方法也时类的组成元素之一
    public void show() {
        System.out.println("show....")
    }
}
```

3.创建并使用对象

3.1使用new关键字创建对象

- 通过 new 类名(); 表示创建了该类的对象，也叫做!类的实例化(instantiation)!

```
new Person();//实例化人类对象
```

3.2 引用类型变量

- 为了能够对实例化的对象进行访问控制，需要使用一个特殊的变量--引用

```
Person p = new Person();
//p：指向对象的引用
```

- 引用存储的是对象的地址信息，"指向对象的引用"。
- 可以通过引用采用打点的形式访问对象的成员。
- 在Java中，除了8种基本类型外，其他类型都为引用数据类型--且默认值为null

```
Person p = new Person();//实例化人类对象
p.name = "zhang3";//引用访问属性
p.age = 16;
p.show();//引用访问方法
Person p1 = p;
Person p2 = new Person();
p2.name = "zhang3";
p2.age = 16;
p2.show();
System.out.println(p == p1);//t    p和p1由同样的地址值
System.out.println(p == p2);//f    p和p2两个new关键字。只要new就会创建对象，地址就不一样
```

3.3 引用类型变量的赋值

- 相同类型之间引用相互赋值
- 引用类型之间的赋值不会创建新的对象，但有可能会使两个引用指向同一对象

3.4 null和nullPointerException

- 对于引用类型变量可以赋值为null，null的含义为“空”，表示没有指向任何对象。
- 当引用的值为null时,再去调用其成员会抛出nullPointerException

```
Person p4 = null;
p4.name = "wangmazi";
```

- 总结:
  - 不同类型组成的抽象的数据类型---类　Java中基本组成单位
  - 类的成员 : 属性-成员变量和方法-成员方法
  - 如何使用类: 通过new 使用类
  - 引用 : 存储了对象地址值的变量 指向对象的引用
  - 引用数据类型: 除了8种基本类型外都叫引用类型 类!
  - 空指针异常: 当对象为空时，使用它会报空指针

二、方法

1.方法的重载

1.1 方法的标识

- 方法的唯一标识就是: 方法的名字 和 参数列表
- 一个类中不能出现两个方法的标识完全一样的方法。

1.2 方法的重载

- 方法名相同但参数列表不同称为方法的重载

```
public void show(){}
//互相构成重载
public void show(int i) {}
```

1.3 访问重载方法

- 编译器在编译时会根据方法的标识调用不同的方法

2.构造方法

2.1构造方法的语法结构

- 构造方法是类的成员之一--特殊的方法，有如下两个规则:
  - 方法名与类名相同
  - 没有返回值类型，且不写void

```
public class Person{

    public Person() {//构造方法

    }
}
```

2.2 通过构造方法初始化成员变量

- 构造方法的意义是 : 初始化成员变量
- 当实例化一个对象时: new Person();实际是执行了对应的构造方法

```
public class Person() {
    Public Person() {
        System.out.println("执行了无参构造方法");
    }
}
public class Test{
    public void static main(String[] args){
        Person p = new Person();//执行了无参构造方法
    }
}
```

2.3 this关键字

- this关键解决了构造方法参数名称和属性名称同名的问题。

```
public Person(String name,int age) {
    this.age = age;
    this.name = name;//this解决同名问题。增加代码可读性
}
```

- this关键字是谁？ 谁调用了this.属性或方法中的某个属性和方法，则this就指谁。

```
public Person(String name) {
    System.out.println("Person的无参构造方法");
    this.name = name;//this.指Person对象。
}
```

2.4 默认的构造方法

- 一个类必须有构造方法，当类中没有定义时，编译器会提供一个默认的无参构造方法。
- 当我们显式的定义了任意一个构造方法时，系统将不会提供默认的无参构造方法。

```
public class Person {

    String name;
    int age;

    public Person(String name) {//类名相同  且无返回值类型--- 构造方法
        System.out.println("Person的无参构造方法");
        this.name = name;//区分同名的
    }
    public void show () {
        System.out.println("show1");
    }
    public void show (int i) {
        System.out.println("show2");
    }
    public void show (double d) {
        System.out.println("show3");
    }
    public void show (int i ,double d) {
        System.out.println("show4");
    }
    public void Show (int i ,double d) {
        System.out.println("show5");
    }
}

public class TestMethod {

    public static void main(String[] args) {
        Person p = new Person();//会报错，系统不会提供默认的构造方法
    }
}
```

2.5 构造方法的重载

- 类名相同，但参数列表不同的构造方法，我们称之互相构成重载

```
public Person(String name,int age) {
    this.name = name;
    this.age = age;
}
public Person(String name) {//类名相同  且无返回值类型--- 构造方法
    System.out.println("Person的无参构造方法");
    this.name = name;//区分同名的
}
```

三、数组(补充)

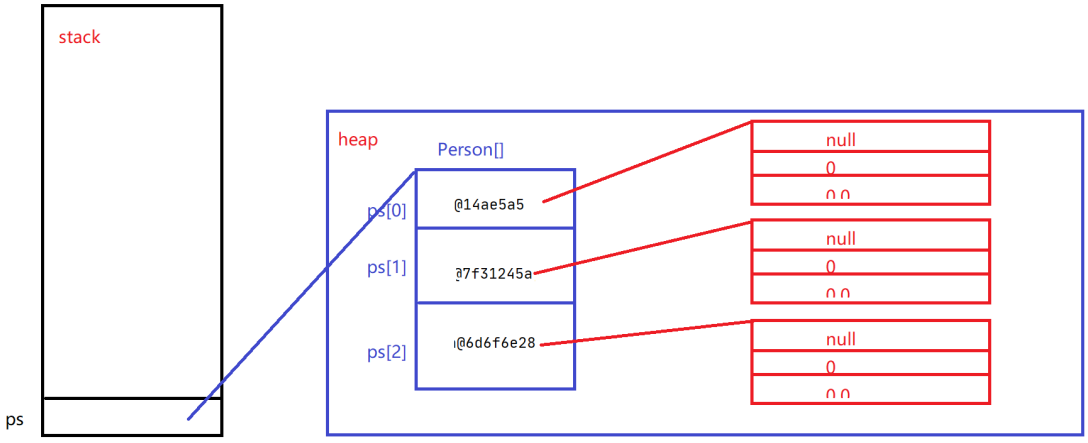
引用类型数组

1.1 数组是对象

- 数组是引用数据类型
- 数组对象在堆中创建，声明的变量是引用类型，引用存储的是数组对象的地址值，指向数组对象
- 可以将数组中的元素看成对象的成员变量，只不过类型完全一致而已。

1.2 引用类型数组的声明

- 数组的元素可以为任意类型，当然也可以为引用类型。
- **引用类型数组元素存储的不是对象本身，而是存储元素对象的引用。**




```
public class Cell {

    int row;
    int col;

    public Cell(int col,int row) {
        this.col = col;
        this.row = row;
    }
}

public class TestCell {
```

```
    public static void main(String[] args) {
        Cell[] cells = new Cell[3];//
        cells[0] = new Cell(1,2);
        cells[1] = new Cell(1,2);
        cells[2] = new Cell(1,2);
        System.out.println(Arrays.toString(cells));

    }
}
```

1.3 数组的初始化(重点)

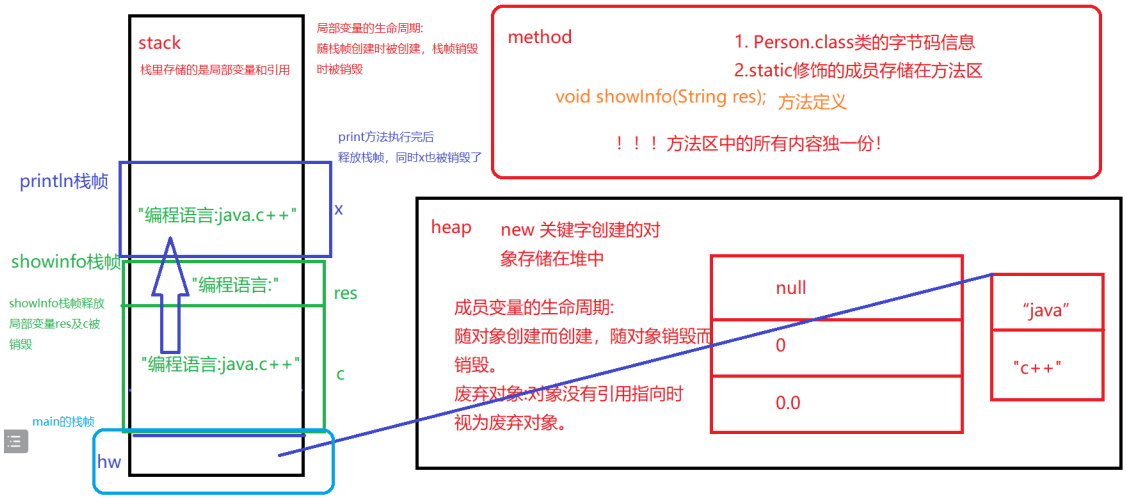
- 数组元素的默认值都为null。
- 如果希望每个元素都指向具体的对象，则需要对每一个元素都使用new创建实例。

1.4数组的元素是基本类型数组

- 数组的元素可以为任意类型，当然也可以为基本类型数组--二维数组

```
//第一种方式
//      int[][] arr = new int[][]{new int[]{0,1,2},new int[]{2,3,4}
//                          ,new int[]{4,5,6,9}};
        //第二种方式
//      int[][] arr = new int[3][];
//      arr[0] = new int[]{0,1,2};
//      arr[1] = new int[]{2,3,4};
//      arr[2] = new int[]{4,5,6,9};
        //第三种方式，数组元素中的基本类型数组的元素必须相同
        int[][] arr = new int[3][4];
        arr[0] = new int[]{0,1,2,5};
        arr[1] = new int[]{2,3,4,5};
        arr[2] = new int[]{4,5,6,9};
        System.out.println(arr[2][3]);//9
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                System.out.println(arr[i][j]);
            }
        }
        //这个方法打印出来的是里面数组的三个地址
        System.out.println(Arrays.toString(arr));
```

四、对象内存管理(重点)



1.堆内存

1.1 对象存储在堆中

- JVM分为三个区域 : 堆(heap) 、栈(stack)、方法区 （method）
- JVM为Java程序提供并管理所需要的内存空间。
- 堆中存储使用new关键字创建的对象---数组、String、 Person等等。

1.2 成员变量的生命周期

- 当一个引用值为null时，该对象没有任何引用指向，则该对象被视为废弃对象，属于被回收(GC线程)范围。
- 成员变量的生命周期: 从对象在堆中创建，到对象从堆中被回收为止。

1.3垃圾回收机制(面试，建议深究)

- 垃圾回收器(Garbage Collection ,GC),是JVM自带的一个守护线程(自动运行的),用于回收没有任何引用指向的对象。
- 垃圾回收器会自动帮Java程序员处理垃圾。

1.4 内存泄漏

- 内存泄漏: 不再使用的对象没有被及时回收,积攒过多导致程序奔溃。

1.5 System.gc()方法

- GC线程不会刚发现无用对象就会回收，如果需要强制回收，则使用System.gc()方法
- 这个方法强制调度GC线程回收无用对象。

2.非堆-栈

2.1栈存放方法中的局部变量

- 栈里存储了方法中的局部变量。

2.2局部变量的生命周期

- 生命周期: 栈帧被创建开始-方法执行完毕栈帧被销毁结束，局部变量也随之销毁。
- 栈帧中存储的是: 局部变量和参数等。

2.3 案例

- 绘制下列代码内存区域图:

```
public class HomeWork1 {
    String a = "java";
    String b = "c++";
    public void showInfo(String res) {
        String c = res + a +"."+ b  ;
        System.out.println(c);
    }
    public static void main(String[] args) {
        HomeWork1 hw = new HomeWork1();
        hw.showInfo("编程语言:");
    }
}
```

2.4 局部变量和成员变量区别(面试)

3.非堆-方法区

3.1 存放类的信息

- Java运行时，会通过类装载器载入类文件的字节码信息，解析后放在方法区中。

3.2 方法只有一份

- 当类的信息加载到方法区时，类中的类方法的定义也被加载在方法区中。
- 无论创建多少对象，所有的对象是公用方法区中一个方法的定义。

五、继承

1.继承

1.1 继承的格式

- 通过extends关键字可以实现继承
- 子类可以通过继承获取父类的属性和方法，也可以定义自己独有的属性和方法。
- 继承单一性: 一个子类只能有一个父类(1个儿子只能有一个爸爸)，但一个父类可以有多个子类。

```
/**
 * 子类
 * @author JeffLee
 *
```

```
 */
public class SubClass extends SuperClass{

    public static void main(String[] args) {
        new SubClass().SuperInfo();//子类对象调用了子类中继承的SuperInfo方法。
    }
}
//父类
class SuperClass {

    String name;
    int age;

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

1.2 继承中的构造方法

- 子类的构造方法必须通过super关键字调用父类的构造方法，目的是:初始化父类的成员变量

```
public class SubClass extends SuperClass{

    public SubClass() {
        super("zhang3",16);
    }

    public static void main(String[] args) {
        //new SubClass().SuperInfo();
        SubClass sb = new SubClass();
        System.out.println(sb.name);//zhang3

    }
}
class SuperClass {
    String name;
    int age;
    public SuperClass(String name,int age) {
        this.name = name;
        this.age = age;
    }
    public void SuperInfo() {
        System.out.println("SuperInfo....");
    }
}
```

- 调用子类的构造方法时，如果没有使用super关键字，则程序会先执行父类的无参构造方法，再执行子类的构造方法，如果父类没有无参构造方法，则会编译错误

   image-20211121113415065

1.3父类的引用指向子类对象

- 父类的引用可以指向子类的对象，也叫做子类对象向上造型为父类类型。
- 当父类的引用指向子类对象时，父类的引用只能访问父类的属性和方法。

```
public class Person extends PersonFather{
    String name2;
    public void personTest() {
        System.out.println("person....");
    }
    public static void main(String[] args) {
        PersonFather pf1 = new PersonFather();//父类对象
        Person p = new Person();//实例化子类对象
        PersonFather pf2 = new Person();
        System.out.println(pf2.name);
        //System.out.println(pf2.name2);只能访问父类中定义的属性和方法
    }
}
class PersonFather{
    String name;
    public void test(){
        System.out.println("Father...");
    }
}
```

2.重写

2.1方法的重写

- 子类可以重写(覆盖)父类中定义的方法，即方法名和参数列表与父类方法相同，但方法体不同
- 当子类对象重写的方法被调用时，**无论引用是父类还是子类都执行重写后的方法。**
- 作业:定义父类SuperClass，子SubClass show

```
Son s = new Son();
s.sum();//z
Father f = new Father();
f.sum();//f
Father f1 = new Son();
f1.sum();//  z
```

2.2 重写时的super

- 子类在重写父类的方法时，可以使用super关键字调用父类的方法

```
@Override
public void area() {
    super.area();//通过super关键字可以调用父类的area方法
    System.out.println(3.14*4*4);
}
```

2.3 重载和重写的区别

- 重载
  - 发生在编译期
  - 方法名相同，但参数列表不同
  - 调用时根据方法名和参数列表去判断
- 重写

- 发生在运行期
- 前提是: 存在继承关系时，子类重写的方法与父类完全一致
- 调用时根据引用指向的对象类型去判断

六、访问控制

1.包的概念

1.1 package语句

- 在Java中使用package关键字区别同名的类
- package的目的: 解决命名冲突的问题
- 包语句必须写在java文件的开头
- 格式如下:

```
package com.tedu.oop.day01;//指定包名，不同的包下可以存在同名的类
```

- 一个类的全类名(全限定名)指的是: 包名 + 类名

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

- 包名的命名有如下规则:

```
package org.apache.commons.lang.StringUtils;
```

  - StringUtils : 类名
  - org.apache ： 公司或者组织域名的反写
  - commons： 项目名称信息
  - lang : 项目模块信息
- 包名实际上在本地工程目录中是一个多级文件目录，以 “.” 分割
- java.lang包下的所有类不用导包
- 常见Java包
  - java.math 数学运算
  - java.io io包
  - java.utils 集合等
  - java.net 网络编程
  - java.sql 数据库

1.2 import语句

- 格式如下:

```
package com.tedu.oop.day06;

import java.util.Scanner;

public class TestPackage {

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

```
        /*
         * 1.当使用全类名书写时很繁琐因此采用简写
         * 2.简写的前提时必须要使用import语句导入这个包
         * 3.导入的包只限于当前的java文件。
         */
        //java.util.Scanner scan = new java.util.Scanner(System.in);
        Scanner scan = new Scanner(System.in);
    }
}
```

2.访问修饰符

2.1 访问修饰符

修饰符	本类	同一个包中的类	其他包的子类	其他包的其他类
public	可以访问	可以访问	可以访问	可以访问
protected	可以访问	可以访问	可以访问	不能访问
默认	可以访问	可以访问	不能访问	不能访问
private	可以访问	不能访问	不能访问	不能访问

2.2 访问修饰符修饰成员

- public 修饰的成员可以在任意类访问，意义: 对外提供可以被调用的功能。
- private 修饰的成员只能在本类中使用，意义 : 对内的封装，减少维护成本。
- 默认的(不写) 的成员可以在本类和同一个包中的类访问。
- protected修饰成员 可以在本类、同一个包中的类及其他包中子类访问。

2.3 访问修饰符修饰类

- 类的修饰： public 和 默认的
  - 一个Java文件中只能有一个public修饰的类
  - 类的修饰词只有 : final 、abstract 、 public 、默认的
- 内部类的修饰可以使用任意修饰词

七、封装

1. 封装概念

1.1 封装的意义

- 即隐藏对象的属性和实现细节，仅对外公开接口。
- 将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体(类)。

1.2 封装的实现

- 将属性私有化，提供供外部访问的公共接口(set和get方法)

2. 实现封装

- 属性使用private修饰，提供公共的set和get方法

```
/**
 * 实体类
 * @author JeffLee
 *
 */
public class User {

    private String username;
    private String password;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

- 总结:
  - 私有化的目的: 对数据进行隐藏
  - set和get方法目的: 对外提供操作数据的公共 “接口“
- IDE可以自动生成
  - IDEA 右键选择Generate

    image-20220221162033512

    image-20220221162108911

八、static 和 final(重点)

1. static

1.1 static 修饰成员变量

- 使用static修饰的成员变量不在对象的数据结构，而是类的基本信息(参数)
- 使用static修饰的成员可以直接使用 类名.成员 的方式访问，而不需要再new对象了
  - 使用static修饰的成员存储在方法区(方法区中的成员独一份)，static修饰的成员只有一份。

```
/**
 * 静态关键字
```

```
 * @author JeffLee
 *
 */
public class TestStatic {

    static int age = 0;//存储在方法区 且独一份
    int score = 0;

    public TestStatic() {
        age += 1;
        score += 1;
    }

    public static void main(String[] args) {
        new TestStatic();//age : 1  score : 1
        new TestStatic();//age : 2  score : 1
        System.out.println(TestStatic.age);//2
        System.out.println(new TestStatic().score);//score : 1 age:3

    }
}
```

1.2 案例-模拟统计网站访问人数

- 模拟网站类Server

```
public class Server {

    String picture;//网站图片
    String ref;//网站某一连接
    String button;//网站某一按钮
    String model;//网站某一模块
    static int clientCount;
    public void getPicture() {
        System.out.println("访问了图片");
    }
    public void getRef() {
        System.out.println("访问了链接");
    }
    public void getButton() {
        System.out.println("访问了按钮");
    }
    public void getModel() {
        System.out.println("访问了模块");
    }

    public Server() {
        clientCount ++;
    }
}
```

- 模拟用户类Client

```java
import java.util.Scanner;

/**
 * 案例-模拟统计网站访问人数
 * @author JeffLee
 *
 */
public class Client {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("=======各位用户您好=======");
        System.out.println("请选择要访问的内容:");
        System.out.println("0.退出网站");
        System.out.println("1.图片");
        System.out.println("2.链接");
        System.out.println("3.按钮");
        System.out.println("4.模块");
        while(true) {
            int choose = scan.nextInt();
            if(choose == 0) {
                break;
            }
            switch (choose) {
            case 1:
                new Server().getPicture();
                break;
            case 2:
                new Server().getRef();
                break;
            case 3:
                new Server().getButton();//^
                break;
            case 4:
                new Server().getModel();
                break;
            }
        }

        System.out.println("当前有" + Server.clientCount + "人访问了网站");
    }
}
```

1.3 static修饰方法

- static 修饰的方法使用类名去调用，它不涉及对象的操作
- static 一般用在工厂方法或类中。
- static 方法不能调用非static变量。

1.4 static块和非static块

- 格式:

```
static {
    //静态代码块
}
{
    //非静态代码块
}
```

- 静态代码块是属于类的一部分，在类加载期间就会执行完毕，一般用来加载静态资源(图片、音频等)。

扩展

- 静态代码块、非静态代码块和构造方法的执行流程

```
public class Test01 {

    public Test01() {
        System.out.println("1无参构造方法");
    }
    static {
        System.out.println("2static块");
    }
    {
        System.out.println("3非static块");
    }
    public static void main(String[] args) {
        Test01 t = new Test01();
    }
}
```

  - 结论: static块 > 非静态代码块 > 构造方法

总结

1. 静态修饰的成员可以直接使用类名去调用
2. 静态修饰的成员会优先加载，在类的加载期间就完成了加载。
3. 静态修饰方法不能调用非静态成员，但非静态方法可以调用静态的成员
4. 静态修饰的成员存储在方法区中，且独一份

2. final

2.1 final修饰变量

- final修饰的成员变量不可变
- 初始化的方式只有两种:
  - 声明时初始化

    ```
    final int age = 15;
    ```

  - 构造方法初始化

```
final int age;
public TestFinal() {
    age = 15;
}
```

- final 也可以局部变量使用之前初始化即可

```
final int age;
age = 15;
```

2.2 final修饰方法

- final修饰的方法不能被重写 ， 意义: 造成"不经意"重写

```
public final void show(){}//final关键字和public关键字顺序无关可以互换
```

2.3 final修饰类

- final修饰的类不能被继承
- 常见的final修饰的类有String、Math等等。
- 意义: 防止滥用继承，可以保护类不被继承修改，降低对系统造成的危害
- 面试: String能不能被继承？ 不能，原因是String类是final修饰的、我研究过源码

```
public final String{}
```

2.4 常量

- 常量:static final修饰的变量
- 常量的命名规范为全大写
- 特点: 初始化时只能通过在声明时初始化，否则会编译错误

```
//The blank final field AGE may not have been initialized
//public static final int AGE;
public static final int AGE = 16;
```

- 常量在编译时，会被替换为实际存储的值

```
System.out.println(TestConstant.AGE);
//等同于
System.out.println(16);
```

总结

1. static 修饰的成员 可以直接使用类名去调用
2. static 修饰的成员 可以优先加载(类加载期间就已经执行了)
3. static 修饰的成员 存储在方法区中，且独一份
4. static 方法不能调用no-static成员
5. final 修饰变量不能改变 修饰方法不能被重写 修饰类不能被继承
6. final 修饰的变量初始化: 构造方法中初始化和声明时初始化

7. static final 修饰的变量叫常量，只能在声明时初始化

案例-窗体创建的三种方式

- 第一种：

```
import java.awt.Toolkit;

import javax.swing.JFrame;

/**
 * 我的第一个窗体 main里面写所有
 * @author JeffLee
 *
 */
public class MyFirstJFrame {
    public static void main(String[] args) {
        //JFrame:窗体类
        JFrame jf = new JFrame();
        //1.窗体的标题
        jf.setTitle("My first JFrame");
        //获取屏幕的宽
        int width = Toolkit.getDefaultToolkit().getScreenSize().width;
        //获取屏幕的高
        int height = Toolkit.getDefaultToolkit().getScreenSize().height;
        //定义窗体宽
        int jfWidth = 400;
        //定义窗体的高
        int jfHeight = 600;
        //计算水平居中x初始坐标
        int jframeX = width / 2 - jfWidth / 2;
        //计算垂直居中y初始坐标
        int jframeY = height / 2 - jfHeight / 2;
        //2.窗体的大小及位置
        jf.setBounds(jframeX, jframeY, jfWidth, jfHeight);
        //3.设置默认的关闭方式
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //4.窗体可见
        jf.setVisible(true);
    }
}
```

- 第二种

```
import java.awt.Toolkit;

import javax.swing.JFrame;

/**
 * 我的第一个窗体:采用构造方法，
 * 使程序顺序更加清晰
 * @author JeffLee
 *
 */
```

```
public class MySecondJFrame {

    JFrame jf;

    public MySecondJFrame() {
        jf = new JFrame();
        //1.窗体的标题
        jf.setTitle("My first JFrame");
        //获取屏幕的宽
        int width = Toolkit.getDefaultToolkit().getScreenSize().width;
        //获取屏幕的高
        int height = Toolkit.getDefaultToolkit().getScreenSize().height;
        //定义窗体宽
        int jfWidth = 400;
        //定义窗体的高
        int jfHeight = 600;
        //计算水平居中x初始坐标
        int jframeX = width / 2 - jfWidth / 2;
        //计算垂直居中y初始坐标
        int jframeY = height / 2 - jfHeight / 2;
        //2.窗体的大小及位置
        jf.setBounds(jframeX, jframeY, jfWidth, jfHeight);
        //3.设置默认的关闭方式
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //4.窗体可见
        jf.setVisible(true);
    }

    public static void main(String[] args) {
        new MySecondJFrame();
    }
}
```

- 第三种

```
import java.awt.Toolkit;

import javax.swing.JFrame;

/**
 * 我的第一个窗体:采用继承和this关键字，属性工厂
 * 体现了代码的复用，及面向对象的编程思想
 *
 * @author JeffLee
 *
 */
public class MyThirdJFrame extends JFrame{

    public MyThirdJFrame() {
        this.setTitle("My first JFrame");
        this.setBounds(Factory.jframeX, Factory.jframeY, Factory.jfWidth,
Factory.jfHeight);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }

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

```
        new MyThirdJFrame();
    }
}
class Factory{
    public static int width =
Toolkit.getDefaultToolkit().getScreenSize().width;
    public static int height =
Toolkit.getDefaultToolkit().getScreenSize().height;
    public static int jfWidth = 400;
    public static int jfHeight = 600;
    public static int jframeX = width / 2 - jfWidth / 2;
    public static int jframeY = height / 2 - jfHeight / 2;
}
```

- 思考: 三种创建方式的优缺点，及OOP编程的好处。

九、抽象类

1. 抽象方法和抽象类

1.1 抽象方法

- 由abstract修饰的方法为抽象方法
- 抽象方法只有方法的定义，没有方法的实现
- 抽象方法没有方法体

```
public abstract void show();//以分号结尾
```

1.2 抽象类

- 由abstract修饰的类是抽象类
- **一个类如果包含抽象方法，那么一定是一个抽象类**
- 抽象类中可以有抽象方法，也可以有非抽象方法
- 继承抽象类后必须实现抽象类中定义的所有抽象方法
- 不同的子类可以有不同的实现

```
//图形类
public abstract class Shape {//抽象类

    public abstract void area();//抽象方法
}
//圆类
public class Circle extends Shape{
    /*
     * The type Circle must implement the inherited
     * abstract method Shape.area()
     */
    @Override
    public void area() {
        System.out.println(4*4*3.14);
```

```
    }
}
//正方形类
public class Square extends Shape{

    @Override
    public void area() {
        System.out.println(4*4);
    }
}
```

2. 抽象类不能实例化

- 抽象类不能被实例化
- 一个类中没有抽象方法也可以定义为抽象类，同样也不能实例化

```
public class TestAbstract {
    public static void main(String[] args) {
        //Cannot instantiate the type Shape  不能实例化shape类型
        //Shape s = new Shape();
        Circle c = new Circle();
        c.area();
    }
}
```

- 抽象类不能使用final修饰，因为final修饰的类不能被继承，则抽象类没有意义。

十、接口(重)

1.接口的定义

- 使用interface定义的是接口，但 !!!接口不是类!!! 接口 是特殊的抽象类
- 接口中的属性默认为常量!
- 接口中的方法默认为抽象方法!

2.接口的实现

- 接口实现需要实现类实现接口中没有被实现的方法
- 使用implements关键字实现接口

```
public interface Shape1 {
    public int A = 1;//默认为常量
    public void area();//默认为抽象方法
}
class CircleImp implements Shape1{

    @Override
    public void area() {
        System.out.println(4*4*3.14);
```

```
    }
}
class SquareImp implements Shape1{

    @Override
    public void area() {
        System.out.println(4*4);
    }

}
```

- 接口可以作为类型指向实现类对象，调用时不同实现类的实现方法

```
public class TestInterface {

    public static void main(String[] args) {
        //Shape1 s = new Shape1();//Cannot instantiate the type Shape1
        CircleImp ci = new CircleImp();
        ci.area();
        SquareImp si = new SquareImp();
        si.area();
        //接口类型指向不同实现类对象
        Shape1 s = new CircleImp();
        s.area();//计算圆面积
        Shape1 s1 = new SquareImp();
        s1.area();//计算正方形面积
    }
}
```

3.接口的继承

- 接口之间可以有继承关系，但实现类实现某一接口时，必须实现所有继承关系中的抽象方法。
- 一个类可以实现多个接口，同样也必须实现多个接口中定义的抽象方法

```
class UserImp implements UserDao,UserMpper{ }
```

4.接口和抽象类的区别

十一、多态

1.多态的意义

- 同一类型引用指向不同对象

```
Shape s1 = new Circle();
s1.area();
Shape s2 = new Square();
s2.area();
```

- 不用引用指向同一对象

```
Circle c = new Circle();//实现类对象
c.area();
Shape s1 = new Circle();
s1.area();
```

2.向上造型

- 向上造型必须满足的条件为:
  - 父类的类型
  - 其实现的接口类型
- Java编译器会根据引用类型调用方法

```
Father f = new Son();//继承关系，子类对象向上造型为父类类型
UserDao userdao = new UserDaoImp();//接口实现关系，实现类UserDaoImp向上造型为接口UserDao类型
```

3.向下造型

- 通过强制类型转化可以将父类类型转化为子类类型
  - 前提：必须写出父类引用指向子类对象作为前题

```
//前提 父类引用指向子类对象
Father f = new Son();
//将父类类型向下造型为子类类型
Son s1 = (Son)f;
```

- 通过强制类型转化可以将接口类型转化为实现类类型
  - 前提：必须**写出**接口引用指向实现类对象作为前题

```
//前提  接口类型引用指向实现类对象
Shape s = new Circle();
//将接口类型向下造型为实现类类型
Circle c = (Circle)s;
```

  - 如果没有前提则会报类型转化异常

4.instanceof

- instanceof是判断某个引用指向的对象是否为指定类型

```
Father f = new Son();
System.out.println(f instanceof Son);//true
System.out.println(f instanceof Father);//true
```

十二、内部类

1.定义内部类

- 当一个类定义在另一个类的内部，成之为内部类。
- 内部类(Inner)只服务于外部类(Outer)，可以获取外部类的属性和方法
- 定义内部类对象时，一般通过外部类的构造方法或普通方法创建对象

```
public class Outer {

    private String name = "zhang3";//外部类的私有属性

    public Outer() {
        Inner i = new Inner();
        i.nameInfo();
    }

    /*内部类Inner*/
    class Inner{

        public void nameInfo(){
            name = name + "info";
        }
    }

    public static void main(String[] args) {
        Outer o = new Outer();
        System.out.println(o.name);
    }
}
```

2.匿名内部类

- 匿名内部类，就是去new 接口 和 new 抽象类。

```
//User接口
public interface User {

    public void login();
}
//测试类
public class Test01 {

    public static void main(String[] args) {
        User u = new User() {

            @Override
            public void login() {
                System.out.println("show..");
            }
        };
        u.login();
    }

}
```