Day_1

Java 的注释

1.java 规范的三种注释
单行注释
多行注释
文档注释（Java 特有）

2.单行注释和多行注释的作用
①对所写的程序进行解释说明，增强可读性。方便自己，方便他人。
②调试所写的代码

3.特点：
单行注释和多行注释的内容不参与编译。换句话说，编译以后生产的.class 结尾的字节码文件不包含注释掉的信息

4.文档注释的使用
注释内容可以被 JAVADOC 解析，生成一套以网页文件形式体现的该程序的说明文档

5.多行注释不可以嵌套使用

对第一个 java 程序的总结

1.java 程序编写-编译-运行的过程
编写：将编写的文件保存在以.java 结尾的源文件中
编译：使用 javac.exe 命令编译我们的 java 源文件。格式：javac 源文件名.java
运行：使用 java.exe 命令解释运行我们的字节码文件 格式：java 类名

2.一个 java 源文件中可以声明多个 class。但是，只能有一个类声明为 public。
而且要求声明为怕 public 的类的类名必须与源文件名相同。

3.程序的入口是 main 方法。格式是固定的。

4.输出语句：
System.out.println():先输出数据 自动换行
System.out.print():只输出数据不换行

5.每个执行语句末尾都有“;”结束。

6。编译的过程：编译以后，会生成一个或多个字节码文件。字节码文件的文件名与 java 源文件中的类名相同。

Day_2

标识符的使用

1.标识符：凡是自己可以起名字的地方都叫标识符
比如：类名，变量名，方法名，接口名，包名。。

2.标识符的命名规则： -->如不遵守如下规则，编译不通过
26 个英文字母大小写，0-9，_或$组成
数字不可开头
不可以用关键字和保留字，但能包含关键字和保留字
Java 中严格区分大小写，长度无限制
标识符不能包含空格

3.标识符的命名规范 -->不遵守编译也可以通过，建议遵守。
包名：多单词组成的所有字母都小写：xxxyyyzzz
类名，接口名：多单词组成时，所有单词的首字母大写：XxxYyyZzz
变量名，方法名：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写： xxxYyyZzz
常量名；所有字母都大写。多单词时每个单词用下划线连接：XXX_YYY_ZZZ

4. 注意 1：起名要见名知意！
注意 2：java 采用 unicode 字符集，因此标识符可以使用汉字声明，但是不建议使用。

String 类型变量的使用

1. String 属于引用数据类型

2. 声明 String 类型变量时，使用一对""(双引号)

3. String 可以和 8 种基本数据类型变量做运算，且运算只能做连接运算

4.运算之后依然是 String 类型

变量的使用

1. java 定义变量的格式：数据类型 变量名 = 变量值；

2.说明：
变量必须先声明（定义和赋值），后使用。
变量都定义在其作用域内。在作用域内，它是有效的。换句话说，除了作用域，它就失效了。
同一个作用域内，不可声明同一个名的变量。

java 定义的数据类型

一 • 变量按照数据类型来分

基本数据类型
数值型
整型(byte(1 字节),short(2 字节),int(4 字节),long(8 字节))
浮点型(float(4 字节),double(8 字节))
字符型(char) (2 字节),
布尔型(boolean)
引用数据类型
类(class)
接口(interface)
数组(array[])

二、变量在类中声明的位置：
成员变量 vs 局部变量

基本数据类型之间的运算规则：

前提：这里讨论的是 7 种基本数据类型间的运算，不包含 boolean 类型的

1.自动类型提升：

byte、char、short -> int -> long ->float -> double

当容量小的数据类型的变量与容量大的数据类型的变量做运算时，结果自动提升为容量大的数据类型。

特别地，当 byte、char、short 三种类型地变量做运算（相互和自身）时，结果为 int 类型。

说明：此时的容量大小指的是，表示数的范围的大和小。比如：float 容量要大于 long 的容量

2.强制类型转换：自动类型提升运算的逆运算。见 VariableTest3。

1.需要使用强转符：（）

2.强制类型转换，可能导致精度损失。

```
class VariableTest3 {
	public static void main(String[] args) {
		double d1 = 12.9;
		//精度损失举例 1
		int i1 = (int)d1;//截断操作
		System.out.println(i1);.//12
		//没有精度损失
		long l1 = 123;
		short s2 = (short)l1;	//123
		//精度损失举例 2
		int i2 = 128;
		byte b = (byte)i2;
		System.out.println(b);//-128
	}
}
```

计算机中不同进制的使用说明

对于整数，有四种表示方式：

> 二进制(binary)：0,1 ，满 2 进 1.以 0b 或 0B 开头。

> 十进制(decimal)：0-9 ，满 10 进 1。

> 八进制(octal)：0-7 ，满 8 进 1. 以数字 0 开头表示。

> 十六进制(hex)：0-9 及 A-F，满 16 进 1.以 0x 或 0X 开头表示。此处的 A-F 不区分大小写。

如：0x21AF +1= 0X21B0

Day_3

运算符之一：算术运算符

+ - + - * / % (前)++ (后)++ (前)-- (后)-- +

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

        //除号：/
        int num1 = 12;
        int num2 = 5;
        int result1 = num1 / num2;
        System.out.println(result1);//2

        int result2 = num1 / num2 * num2;
        System.out.println(result2);//10

        double result3 = num1 / num2;
        System.out.println(result3);//2.0

        double result4 = num1 / num2 + 0.0;//2.0
        double result5 = num1 / (num2 + 0.0);//2.4
        double result6 = (double)num1 / num2;//2.4
        double result7 = (double)(num1 / num2);//2.0
        System.out.println(result5);
        System.out.println(result6);

        // %:取余运算
        //结果的符号与被模数的符号相同
        //开发中，经常使用%来判断能否被除尽的情况。
        int m1 = 12;
        int n1 = 5;
        System.out.println("m1 % n1 = " + m1 % n1);//2

        int m2 = -12;
        int n2 = 5;
```

```
        System.out.println("m2 % n2 = " + m2 % n2);//-2

        int m3 = 12;
        int n3 = -5;
        System.out.println("m3 % n3 = " + m3 % n3);//2

        int m4 = -12;
        int n4 = -5;
        System.out.println("m4 % n4 = " + m4 % n4);//-2
```
结果的符号与被除数相同!!!

```
        //(前)++ :先自增 1，后运算
        //(后)++ :先运算，后自增 1
        int a1 = 10;
        int b1 = ++a1;
        System.out.println("a1 = " + a1 + ",b1 = " + b1);//11 11

        int a2 = 10;
        int b2 = a2++;
        System.out.println("a2 = " + a2 + ",b2 = " + b2);//11 10

        int a3 = 10;
        ++a3;//a3++;
        int b3 = a3; //11 11

        //注意点：
        short s1 = 10;
        //s1 = s1 + 1;//编译失败 1 是 int 型 s1 是 short 型
        //s1 = (short)(s1 + 1);//正确的
        s1++;//自增 1 不会改变本身变量的数据类型
        System.out.println(s1);

        //问题：
        byte bb1 =127;
        bb1++;
        System.out.println("bb1 = " + bb1); //-128

        //(前)-- :先自减 1，后运算
        //(后)-- :先运算，后自减 1

        int a4 = 10;
        int b4 = a4--;//int b4 = --a4;
        System.out.println("a4 = " + a4 + ",b4 = " + b4);
}
```

```
}
```

运算符之二：赋值运算符

```
=   +=   -=   *=   /=   %=
class SetValueTest {
    public static void main(String[] args) {
        //赋值符号：=
        int i1 = 10;
        int j1 = 10;

        int i2,j2;
        //连续赋值
        i2 = j2 = 10;
        int i3 = 10,j3 = 20;

        //**********************
        int num1 = 10;
        num1 += 2;//num1 = num1 + 2;
        System.out.println(num1);//12

        int num2 = 12;
        num2 %= 5;//num2 = num2 % 5;
        System.out.println(num2);

        short s1 = 10;
        //s1 = s1 + 2;//编译失败
        s1 += 2;//结论：不会改变变量本身的数据类型
        System.out.println(s1);

        //开发中，如果希望变量实现+2 的操作，有几种方法？(前提：int num = 10;)
        //方式一：num = num + 2;
        //方式二：num += 2; (推荐)

        //开发中，如果希望变量实现+1 的操作，有几种方法？(前提：int num = 10;)
        //方式一：num = num + 1;
        //方式二：num += 1;
        //方式三：num++; (推荐)

        //练习 1
        int i = 1;
        i *= 0.1;
        System.out.println(i);//0
        i++;
        System.out.println(i);//1
```

```
        //练习 2
        int m = 2;
        int n = 3;
        n *= m++; //n = n * m++;
        System.out.println("m=" + m);//3
        System.out.println("n=" + n);//6

        //练习 3
        int n1 = 10;
        n1 += (n1++) + (++n1);//n1 = n1 + (n1++) + (++n1);
        System.out.println(n1);//32

    }
}
```

运算符之三：比较运算符

== != > < >= <= instanceof

结论：

1.比较运算符的结果是 boolean 类型

2.区分 == 和 =

```
class CompareTest {
    public static void main(String[] args) {
        int i = 10;
        int j = 20;

        System.out.println(i == j);//false
        System.out.println(i = j);//20

        boolean b1 = true;
        boolean b2 = false;
        System.out.println(b2 == b1);//false
        System.out.println(b2 = b1);//true
    }
}
```

运算符之四：逻辑运算符

& && | || ! ^

说明：

逻辑运算符操作的都是 boolean 类型的变量

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

        //区分& 与 &&
```

```
//相同点 1：& 与  && 的运算结果相同
//相同点 2：当符号左边是 true 时，二者都会执行符号右边的运算
//不同点：当符号左边是 false 时，&继续执行符号右边的运算。&&不再执行符号右边的运算。
//开发中，推荐使用&&
boolean b1 = true;
b1 = false;
int num1 = 10;
if(b1 & (num1++ > 0)){
    System.out.println("我现在在北京");
}else{
    System.out.println("我现在在南京");
}

System.out.println("num1 = " + num1);


boolean b2 = true;
b2 = false;
int num2 = 10;
if(b2 && (num2++ > 0)){
    System.out.println("我现在在北京");
}else{
    System.out.println("我现在在南京");
}

System.out.println("num2 = " + num2);

// 区分：| 与 ||
//相同点 1：| 与  || 的运算结果相同
//相同点 2：当符号左边是 false 时，二者都会执行符号右边的运算
//不同点 3：当符号左边是 true 时，|继续执行符号右边的运算，而||不再执行符号右边的运算
//开发中，推荐使用||
boolean b3 = false;
b3 = true;
int num3 = 10;
if(b3 | (num3++ > 0)){
    System.out.println("我现在在北京");
}else{
    System.out.println("我现在在南京");
}
System.out.println("num3 = " + num3);
```

```
        boolean b4 = false;
        b4 = true;
        int num4 = 10;
        if(b4 || (num4++ > 0)){
            System.out.println("我现在在北京");
        }else{
            System.out.println("我现在在南京");
        }
        System.out.println("num4 = " + num4);
    }
}
```

运算符之五：位运算符 （了解）

结论：

1. 位运算符操作的都是整型的数据
2. << ：在一定范围内，每向左移 1 位，相当于 * 2
   >> :在一定范围内，每向右移 1 位，相当于 / 2

```
class BitTest {
    public static void main(String[] args) {
        int i = 21;
        i = -21;
        System.out.println("i << 2 :" + (i << 2));
        System.out.println("i << 3 :" + (i << 3));
        System.out.println("i << 27 :" + (i << 27));

        int m = 12;
        int n = 5;
        System.out.println("m & n :" + (m & n));
        System.out.println("m | n :" + (m | n));
        System.out.println("m ^ n :" + (m ^ n));

        //练习：交换两个变量的值
        int num1 = 10;
        int num2 = 20;
        System.out.println("num1 = " + num1 + ",num2 = " + num2);

        //方式一：定义临时变量的方式
        //推荐的方式
        int temp = num1;
        num1 = num2;
        num2 = temp;

        //方式二：好处：不用定义临时变量
```

```
            //弊端：① 相加操作可能超出存储范围 ② 有局限性：只能适用于数值类型
            //num1 = num1 + num2;
            //num2 = num1 - num2;
            //num1 = num1 - num2;

            //方式三：使用位运算符 (^:异或)
            //有局限性：只能适用于数值类型
            //num1 = num1 ^ num2;
            //num2 = num1 ^ num2;
            //num1 = num1 ^ num2;
            System.out.println("num1 = " + num1 + ",num2 = " + num2);
        }
}
```

分支结构中的 if-else（条件判断结构）

一、三种结构

第一种：

```
if(条件表达式){
    执行表达式
}
```

第二种：二选一

```
if(条件表达式){
    执行表达式 1
}else{
    执行表达式 2
}
```

第三种：n 选一

```
if(条件表达式){
    执行表达式 1
}else if(条件表达式){
    执行表达式 2
}else if(条件表达式){
    执行表达式 3
}
...
else{
    执行表达式 n
}
```

Day_4

if-else 说明：

1. else 结构是可选的。
2. 针对于条件表达式：
> 如果多个条件表达式之间是“互斥”关系(或没有交集的关系),哪个判断和执行语句声明在上面还是下面，无所谓。
> 如果多个条件表达式之间有交集的关系，需要根据实际情况，考虑清楚应该将哪个结构声明在上面。
> 如果多个条件表达式之间有包含的关系，通常情况下，需要将范围小的声明在范围大的上面。否则，范围小的就没机会执行了。
3.if-else 结构是可以相互嵌套的。
4. 如果 if-else 结构中的执行语句只有一行时，对应的一对{}可以省略的。但是，不建议大家省略。

For 循环结构的使用
一、循环结构的 4 个要素
① 初始化条件
② 循环条件 --->是 boolean 类型
③ 循环体
④ 迭代条件

二、for 循环的结构
for(①;②;④){
③
}

执行过程：① - ② - ③ - ④ - ② - ③ - ④ - ... - ②

如何从键盘获取不同类型的变量：需要使用 Scanner 类
具体实现步骤：
1.导包：import java.util.Scanner;
2.Scanner 的实例化:Scanner scan = new Scanner(System.in);
3.调用 Scanner 类的相关方法（next() / nextXxx()），来获取指定类型的变量

注意：
需要根据相应的方法，来输入指定类型的值。如果输入的数据类型与要求的类型不匹配时，会报异常：InputMisMatchException 导致程序终止。

分支结构之二：switch-case
1.格式
switch(表达式){
case 常量 1:
执行语句 1;
//break;

case 常量 2:

```
        执行语句 2;
        //break;
...

default:
        执行语句 n;
        //break;
}
```

2.说明：

① 根据 switch 表达式中的值，依次匹配各个 case 中的常量。一旦匹配成功，则进入相应 case 结构中，调用其执行语句。

当调用完执行语句以后，则仍然继续向下执行其他 case 结构中的执行语句，直到遇到 break 关键字或此 switch-case 结构末尾结束为止。

② break,可以使用在 switch-case 结构中，表示一旦执行到此关键字，就跳出 switch-case 结构

③ switch 结构中的表达式，只能是如下的 6 种数据类型之一：

byte 、short、char、int、枚举类型(JDK5.0 新增)、String 类型(JDK7.0 新增)

④ case 之后只能声明常量。不能声明范围。

⑤ break 关键字是可选的。

⑥ default:相当于 if-else 结构中的 else.

default 结构是可选的，而且位置是灵活的。

说明：

1. 凡是可以使用 switch-case 的结构，都可以转换为 if-else。反之，不成立。
2. 我们写分支结构时，当发现既可以使用 switch-case,（同时，switch 中表达式的取值情况不太多），

又可以使用 if-else 时，我们优先选择使用 switch-case。原因：switch-case 执行效率稍高。

Day_5

break 和 continue 关键字的使用

	使用范围	循环中使用的作用(不同点)	相同点
break:	switch-case 循环结构中	结束当前循环	关键字后面不能声明执行语句
continue:	循环结构中	结束当次循环	关键字后面不能声明执行语句

While 循环的使用

一、循环结构的 4 个要素

① 初始化条件

② 循环条件 --->是 boolean 类型

③ 循环体

④ 迭代条件

二、while 循环的结构

```
①
while(②){
    ③;
    ④;
}
```

执行过程：① - ② - ③ - ④ - ② - ③ - ④ - ... - ②

说明：

1.写 while 循环千万小心不要丢了迭代条件。一旦丢了，就可能导致死循环！

2.我们写程序，要避免出现死循环。

3.for 循环和 while 循环是可以相互转换的！

区别：for 循环和 while 循环的初始化条件部分的作用范围不同。

算法：有限性。

do-while 循环的使用

一、循环结构的 4 个要素

① 初始化条件

② 循环条件 --->是 boolean 类型

③ 循环体

④ 迭代条件

二、do-while 循环结构：

```
①
do{
    ③;
    ④;

}while(②);
```

执行过程：① - ③ - ④ - ② - ③ - ④ - ... - ②

说明：

1.do-while 循环至少会执行一次循环体！

2.开发中，使用 for 和 while 更多一些。较少使用 do-while

嵌套循环的使用

1.嵌套循环：将一个循环结构 A 声明在另一个循环结构 B 的循环体中,就构成了嵌套循环

2.

外层循环：循环结构 B

内层循环：循环结构 A

3. 说明

① 内层循环结构遍历一遍，只相当于外层循环循环体执行了一次

② 假设外层循环需要执行 m 次，内层循环需要执行 n 次。此时内层循环的循环体一共执行了 m * n 次

4. 技巧：

外层循环控制行数，内层循环控制列数

说明：

1. break 关键字的使用：一旦在循环中执行到 break，就跳出循环
2. 不在循环条件部分限制次数的结构：for(;;) 或 while(true)
3. 结束循环有几种方式？

方式一：循环条件部分返回 false

方式二：在循环体中，执行 break

Day_6

一、数组的概述

1.数组的理解：

数组(Array)，是多个相同类型数据按一定顺序排列的集合，并使用一个名字命名,并通过编号的方式对这些数据进行统一管理。

2.数组相关的概念：

>数组名

>元素

>角标、下标、索引

>数组的长度：元素的个数

3.数组的特点：

1）数组是有序排列的

2）数组属于引用数据类型的变量。数组的元素，既可以是基本数据类型，也可以是引用数据类型

3）创建数组对象会在内存中开辟一整块连续的空间

4）数组的长度一旦确定，就不能修改。

4. 数组的分类：

① 按照维数：一维数组、二维数组、。。。

② 按照数组元素的类型：基本数据类型元素的数组、引用数据类型元素的数组

5. 一维数组的使用

① 一维数组的声明和初始化

② 如何调用数组的指定位置的元素

③ 如何获取数组的长度

④ 如何遍历数组

```
public class ArrayTest2 {
    public static void main(String[] args) {
        //1.二维数组的声明和初始化
        int[] arr = new int[]{1,2,3};//一维数组
        //静态初始化
        int[][] arr1 = new int[][]{{1,2,3},{4,5},{6,7,8}};
        //动态初始化 1
        String[][] arr2 = new String[3][2];
        //动态初始化 2
        String[][] arr3 = new String[3][];
        //错误的情况
//      String[][] arr4 = new String[][4];
//      String[4][3] arr5 = new String[][];
//      int[][] arr6 = new int[4][3]{{1,2,3},{4,5},{6,7,8}};

        //也是正确的写法：
        int[] arr4[] = new int[][]{{1,2,3},{4,5,9,10},{6,7,8}};
        int[] arr5[] = {{1,2,3},{4,5},{6,7,8}};

        //2.如何调用数组的指定位置的元素
        System.out.println(arr1[0][1]);//2
        System.out.println(arr2[1][1]);//null

        arr3[1] = new String[4];
        System.out.println(arr3[1][0]);//null

        //3.获取数组的长度
        System.out.println(arr4.length);//3
        System.out.println(arr4[0].length);//3
        System.out.println(arr4[1].length);//4

        //4.如何遍历二维数组
        for(int i = 0;i < arr4.length;i++){
            for(int j = 0;j < arr4[i].length;j++){
                System.out.print(arr4[i][j] + "    ");
```

```
                }
                System.out.println();
            }

        }
    }
```

⑤ 数组元素的默认初始化值：

> 数组元素是整型：0
> 数组元素是浮点型：0.0
> 数组元素是 char 型：0 或'\u0000'，而非'0'
> 数组元素是 boolean 型：false
> 数组元素是引用数据类型：null

⑥ 数组的内存解析 ：见 ArrayTest1.java

二、二维数组的使用

1.理解：

对于二维数组的理解，我们可以看成是一维数组 array1 又作为另一个一维数组 array2 的元素而存在。

其实，从数组底层的运行机制来看，其实没有多维数组。

2. 二维数组的使用:

① 二维数组的声明和初始化
② 如何调用数组的指定位置的元素
③ 如何获取数组的长度
④ 如何遍历数组

规定：二维数组分为外层数组的元素，内层数组的元素

```
int[][] arr = new int[4][3];
外层元素：arr[0],arr[1]等
内层元素：arr[0][0],arr[1][2]等
```

⑤ 数组元素的默认初始化值

针对于初始化方式一：比如：int[][] arr = new int[4][3];

外层元素的初始化值为：地址值

内层元素的初始化值为：与一维数组初始化情况相同

针对于初始化方式二：比如：int[][] arr = new int[4][];

外层元素的初始化值为：null

内层元素的初始化值为：不能调用，否则报错。

⑥ 数组的内存解析

Day_7

数组中的常见异常：

* 1. 数组角标越界的异常：ArrayIndexOutOfBoundsExcetion
* 2. 空指针异常：NullPointerException

java.util.Arrays:操作数组的工具类，里面定义了很多操作数组的方法

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

        //1.boolean equals(int[] a,int[] b):判断两个数组是否相等。
        int[] arr1 = new int[]{1,2,3,4};
        int[] arr2 = new int[]{1,3,2,4};
        boolean isEquals = Arrays.equals(arr1, arr2);
        System.out.println(isEquals);

        //2.String toString(int[] a):输出数组信息。
        System.out.println(Arrays.toString(arr1));[1,2,3,4]


        //3.void fill(int[] a,int val):将指定值填充到数组之中。
        Arrays.fill(arr1,10);
        System.out.println(Arrays.toString(arr1));[10,10,10,10]


        //4.void sort(int[] a):对数组进行排序。
        Arrays.sort(arr2);
        System.out.println(Arrays.toString(arr2));[1,2,3,4]

        //5.int binarySearch(int[] a,int key)
        int[] arr3 = new int[]{-98,-34,2,34,54,66,79,105,210,333};
        int index = Arrays.binarySearch(arr3, 210);
        if(index >= 0){
            System.out.println(index);//8
        }else{
            System.out.println("未找到");
        }
    }
}
```

算法的考查：求数值型数组中元素的最大值、最小值、平均数、总和等

```
public class ArrayTest1 {
    public static void main(String[] args) {
        int[] arr = new int[10];

        for(int i = 0;i < arr.length;i++){
            arr[i] = (int)(Math.random() * (99 - 10 + 1) + 10);
        }

        //遍历
```

```
        for(int i = 0;i < arr.length;i++){
            System.out.print(arr[i] + "\t");
        }
        System.out.println();

        //求数组元素的最大值
        int maxValue = arr[0];
        for(int i = 1;i < arr.length;i++){
            if(maxValue < arr[i]){
                maxValue = arr[i];
            }
        }
        System.out.println("最大值为：" + maxValue);

        //求数组元素的最小值
        int minValue = arr[0];
        for(int i = 1;i < arr.length;i++){
            if(minValue > arr[i]){
                minValue = arr[i];
            }
        }
        System.out.println("最小值为：" + minValue);
        //求数组元素的总和
        int sum = 0;
        for(int i = 0;i < arr.length;i++){
            sum += arr[i];
        }
        System.out.println("总和为：" + sum);
        //求数组元素的平均数
        int avgValue = sum / arr.length;
        System.out.println("平均数为：" + avgValue);
    }
}
```

算法的考查：数组的复制、反转、查找(线性查找、二分法查找)

```
public class ArrayTest2 {
    public static void main(String[] args) {

        String[] arr = new String[]{"JJ","DD","MM","BB","GG","AA"};

        //数组的复制(区别于数组变量的赋值：arr1 = arr)
        String[] arr1 = new String[arr.length];
        for(int i = 0;i < arr1.length;i++){
            arr1[i] = arr[i];
        }
```

```
        //数组的反转
        //方法一：
//      for(int i = 0;i < arr.length / 2;i++){
//          String temp = arr[i];
//          arr[i] = arr[arr.length - i -1];
//          arr[arr.length - i -1] = temp;
//      }

        //方法二：
//      for(int i = 0,j = arr.length - 1;i < j;i++,j--){
//          String temp = arr[i];
//          arr[i] = arr[j];
//          arr[j] = temp;
//      }

        //遍历
        for(int i = 0;i < arr.length;i++){
            System.out.print(arr[i] + "\t");
        }

        System.out.println();
        //查找（或搜索）
        //线性查找：
        String dest = "BB";
        dest = "CC";

        boolean isFlag = true;

        for(int i = 0;i < arr.length;i++){

            if(dest.equals(arr[i])){
                System.out.println("找到了指定的元素，位置为：" + i);
                isFlag = false;
                break;
            }

        }
        if(isFlag){
            System.out.println("很遗憾，没有找到的啦！");

        }
        //二分法查找：(熟悉)
        //前提：所要查找的数组必须有序。
```

```
        int[] arr2 = new int[]{-98,-34,2,34,54,66,79,105,210,333};

        int dest1 = -34;
        dest1 = 35;
        int head = 0;//初始的首索引
        int end = arr2.length - 1;//初始的末索引
        boolean isFlag1 = true;
        while(head <= end){

            int middle = (head + end)/2;

            if(dest1 == arr2[middle]){
                System.out.println("找到了指定的元素，位置为：" + middle);
                isFlag1 = false;
                break;
            }else if(arr2[middle] > dest1){
                end = middle - 1;
            }else{//arr2[middle] < dest1
                head = middle + 1;
            }


        }

        if(isFlag1){
            System.out.println("很遗憾，没有找到的啦！");
        }


    }
}
```

数组的冒泡排序的实现

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

        int[] arr = new int[]{43,32,76,-98,0,64,33,-21,32,99};

        //冒泡排序
        for(int i = 0;i < arr.length - 1;i++){

            for(int j = 0;j < arr.length - 1 - i;j++){

                if(arr[j] > arr[j + 1]){
```

```
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }

            }

        }



        for(int i = 0;i < arr.length;i++){
            System.out.print(arr[i] + "\t");
        }

    }
}
```

快速排序

通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分关键字小，则分别对这两部分继续进行排序，直到整个序列有序。

```
public class QuickSort {
    private static void swap(int[] data, int i, int j) {
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
    private static void subSort(int[] data, int start, int end) {
        if (start < end) {
            int base = data[start];
            int low = start;
            int high = end + 1;
            while (true) {
                while (low < end && data[++low] - base <= 0)
                    ;
                while (high > start && data[--high] - base >= 0)
                    ;
                if (low < high) {
                    swap(data, low, high);
                } else {
                    break;
                }
            }
```

```
                swap(data, start, high);

                subSort(data, start, high - 1);//递归调用
                subSort(data, high + 1, end);
            }
        }
        public static void quickSort(int[] data){
            subSort(data,0,data.length-1);
        }


        public static void main(String[] args) {
            int[] data = { 9, -16, 30, 23, -30, -49, 25, 21, 30 };
            System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
            quickSort(data);
            System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
        }
}
```

Day_8

一、Java 面向对象学习的三条主线：（第 4-6 章）

1.Java 类及类的成员：属性、方法、构造器；代码块、内部类

2.面向对象的三大特征：封装性、继承性、多态性、(抽象性)

3.其它关键字：this、super、static、final、abstract、interface、package、import 等

“大处着眼，小处着手”

二、“人把大象装进冰箱”

*

1.面向过程：强调的是功能行为，以函数为最小单位，考虑怎么做。

① 把冰箱门打开
② 抬起大象，塞进冰箱
② 把冰箱门关闭

2.面向对象：强调具备了功能的对象，以类/对象为最小单位，考虑谁来做。

```
人{
        打开(冰箱){
            冰箱.开开();
        }
```

```
        抬起(大象){
            大象.进入(冰箱);
        }

        关闭(冰箱){
            冰箱.闭合();
        }

}


冰箱{
        开开(){}
        闭合(){}
}

大象{
        进入(冰箱){
        }
}
```

三、面向对象的两个要素：

类：对一类事物的描述，是抽象的、概念上的定义

对象：是实际存在的该类事物的每个个体，因而也称为实例(instance)

>面向对象程序设计的重点是类的设计

>设计类，就是设计类的成员。

一、设计类，其实就是设计类的成员

属性 = 成员变量 = field = 域、字段

方法 = 成员方法 = 函数 = method

创建类的对象 = 类的实例化 = 实例化类

二、类和对象的使用（面向对象思想落地的实现）：

1.创建类，设计类的成员

2.创建类的对象

3.通过“对象.属性”或“对象.方法”调用对象的结构

三、如果创建了一个类的多个对象，则每个对象都独立的拥有一套类的属性。（非 static 的）

意味着：如果我们修改一个对象的属性 a，则不影响另外一个对象属性 a 的值。

四、对象的内存解析

类中属性的使用

属性（成员变量）　vs　局部变量

1.相同点：

1.1　定义变量的格式：数据类型　变量名 = 变量值

1.2 先声明，后使用

1.3 变量都有其对应的作用域

2.不同点：

2.1 在类中声明的位置的不同

属性：直接定义在类的一对{}内

局部变量：声明在方法内、方法形参、代码块内、构造器形参、构造器内部的变量

2.2 关于权限修饰符的不同

属性：可以在声明属性时，指明其权限，使用权限修饰符。

常用的权限修饰符：private、public、缺省、protected　--->封装性

目前，大家声明属性时，都使用缺省就可以了。

局部变量：不可以使用权限修饰符。

2.3 默认初始化值的情况：

属性：类的属性，根据其类型，都有默认初始化值。

整型（byte、short、int、long）：0

浮点型（float、double）：0.0

字符型（char）：0　（或'\u0000'）

布尔型（boolean）：false

引用数据类型（类、数组、接口）：null

局部变量：没有默认初始化值。

意味着，我们在调用局部变量之前，一定要显式赋值。

特别地：形参在调用时，我们赋值即可。

2.4 在内存中加载的位置：

属性：加载到堆空间中　（非 static）

局部变量：加载到栈空间

类中方法的声明和使用

方法：描述类应该具有的功能。

比如：Math 类：sqrt()\random() \...

Scanner 类：nextXxx() ...

Arrays 类：sort() \ binarySearch() \ toString() \ equals() \ ...

1.举例：

public void eat(){}
public void sleep(int hour){}
public String getName(){}
public String getNation(String nation){}

2. 方法的声明：权限修饰符　返回值类型　方法名(形参列表){
　　方法体
　　}
注意：static、final、abstract 来修饰的方法，后面再讲。

3. 说明：
3.1 关于权限修饰符：默认方法的权限修饰符先都使用 public
Java 规定的 4 种权限修饰符：private、public、缺省、protected　-->封装性再细说

3.2 返回值类型：　有返回值　vs 没有返回值
3.2.1　如果方法有返回值，则必须在方法声明时，指定返回值的类型。同时，方法中，需要使用 return 关键字来返回指定类型的变量或常量：“return 数据”。
如果方法没有返回值，则方法声明时，使用 void 来表示。通常，没有返回值的方法中，就不需要使用 return.但是，如果使用的话，只能“return;”表示结束此方法的意思。

3.2.2 我们定义方法该不该有返回值？
① 题目要求
② 凭经验：具体问题具体分析

3.3 方法名：属于标识符，遵循标识符的规则和规范，“见名知意”

3.4 形参列表：　方法可以声明 0 个，1 个，或多个形参。
3.4.1 格式：数据类型 1 形参 1,数据类型 2 形参 2,...
3.4.2 我们定义方法时，该不该定义形参？
① 题目要求
② 凭经验：具体问题具体分析

3.5 方法体：方法功能的体现。

4.return 关键字的使用：
1.使用范围：使用在方法体中
2.作用：① 结束方法
② 针对于有返回值类型的方法，使用"return 数据"方法返回所要的数据。
3.注意点：return 关键字后面不可以声明执行语句。

5. 方法的使用中，可以调用当前类的属性或方法
特殊的：方法 A 中又调用了方法 A:递归方法。
方法中，不可以定义方法。

Day_9

一、理解“万事万物皆对象”

1.在 Java 语言范畴中，我们都将功能、结构等封装到类中，通过类的实例化，来调用具体的功能结构

>Scanner,String 等

>文件：File

>网络资源：URL

2.涉及到 Java 语言与前端 Html、后端的数据库交互时，前后端的结构在 Java 层面交互时，都体现为类、对象。

二、内存解析的说明

1.引用类型的变量，只可能存储两类值：null 或 地址值（含变量的类型）

三、匿名对象的使用

1.理解：我们创建的对象，没有显式的赋给一个变量名。即为匿名对象

2.特征：匿名对象只能调用一次。

3.使用：如下

```
public class InstanceTest {
	public static void main(String[] args) {
		Phone p = new Phone();
//		p = null;
		System.out.println(p);

		p.sendEmail();
		p.playGame();


		//匿名对象
//		new Phone().sendEmail();
//		new Phone().playGame();

		new Phone().price = 1999;
		new Phone().showPrice();//0.0

		//**********************************
		PhoneMall mall = new PhoneMall();
//		mall.show(p);
		//匿名对象的使用
		mall.show(new Phone());

	}
}
```

```
class PhoneMall{
	public void show(Phone phone){
		phone.sendEmail();
		phone.playGame();
	}
}

class Phone{
	double price;//价格

	public void sendEmail(){
		System.out.println("发送邮件");
	}

	public void playGame(){
		System.out.println("玩游戏");
	}

	public void showPrice(){
		System.out.println("手机价格为：" + price);
	}

}
```

关于变量的赋值：

如果变量是基本数据类型，此时赋值的是变量所保存的数据值。

如果变量是引用数据类型，此时赋值的是变量所保存的数据的地址值。

方法的形参的传递机制：值传递

1. 形参：方法定义时，声明的小括号内的参数

   实参：方法调用时，实际传递给形参的数据

2.值传递机制：

如果参数是基本数据类型，此时实参赋给形参的是实参真实存储的数据值。

如果参数是引用数据类型，此时实参赋给形参的是实参存储数据的地址值。

可变个数形参的方法

1. jdk 5.0 新增的内容

2.具体使用：

2.1 可变个数形参的格式：数据类型 ... 变量名

2.2 当调用可变个数形参的方法时，传入的参数个数可以是：0 个，1 个,2 个，。。。

2.3 可变个数形参的方法与本类中方法名相同，形参不同的方法之间构成重载

2.4 可变个数形参的方法与本类中方法名相同，形参类型也相同的数组之间不构成重载。

换句话说，二者不能共存。

2.5 可变个数形参在方法的形参中，必须声明在末尾

2.6 可变个数形参在方法的形参中,最多只能声明一个可变形参。

```
****************************************
public void show (String ... strs) {          //形参个数可变 0~若干个
        System.out.println("show(string ... strs)");
public void show (String[] strs) {        两个方法是相同的
        System.out.println("show(string ... strs)");
**********************************************
```

方法的重载（overload） loading...

1.定义：在同一个类中，允许存在一个以上的同名方法，只要它们的参数个数或者参数类型不同即可。

"两同一不同":同一个类、相同方法名

参数列表不同：参数个数不同，参数类型不同

2. 举例：

Arrays 类中重载的 sort() / binarySearch()

3.判断是否是重载：

跟方法的权限修饰符、返回值类型、形参变量名、方法体都没有关系！

4. 在通过对象调用方法时，如何确定某一个指定的方法：

方法名 ---> 参数列表

Day_10

面向对象的特征一：封装与隐藏 3W:what? why? how?

一、问题的引入：

当我们创建一个类的对象以后，我们可以通过"对象.属性"的方式，对对象的属性进行赋值。这里，赋值操作要受到属性的数据类型和存储范围的制约。除此之外，没有其他制约条件。但是，在实际问题中，我们往往需要给属性赋值加入额外的限制条件。这个条件就不能在属性声明时体现，我们只能通过方法进行限制条件的添加。（比如：setLegs()）

同时，我们需要避免用户再使用"对象.属性"的方式对属性进行赋值。则需要将属性声明为私有的(private).

-->此时，针对于属性就体现了封装性。

二、封装性的体现：

我们将类的属性 xxx 私有化(private),同时，提供公共的(public)方法来获取(getXxx)和设置(setXxx)此属性的值

拓展：封装性的体现：① 如上 ② 不对外暴露的私有的方法 ③ 单例模式 ...

三、封装性的体现，需要权限修饰符来配合。

1.Java 规定的 4 种权限（从小到大排列）：private、缺省、protected 、public
2.4 种权限可以用来修饰类及类的内部结构：属性、方法、构造器、内部类
3.具体的，4 种权限都可以用来修饰类的内部结构：属性、方法、构造器、内部类
修饰类的话，只能使用：缺省、public

总结封装性：Java 提供了 4 种权限修饰符来修饰类及类的内部结构，体现类及类的内部结构在被调用时的可见性的大小。

类的结构之一：构造器（或构造方法、constructor）的使用

construct：建设、建造。 construction:CCB constructor:建设者

一、构造器的作用：

1.创建对象

2.初始化对象的信息

二、说明：

1.如果没有显式的定义类的构造器的话，则系统默认提供一个空参的构造器

2.定义构造器的格式：权限修饰符 类名(形参列表){}

3.一个类中定义的多个构造器，彼此构成重载

4.一旦我们显式的定义了类的构造器之后，系统就不再提供默认的空参构造器

5.一个类中，至少会有一个构造器。

总结：属性赋值的先后顺序

① 默认初始化

② 显式初始化

③ 构造器中初始化

④ 通过"对象.方法" 或 "对象.属性"的方式，赋值

以上操作的先后顺序：① - ② - ③ - ④

JavaBean 是一种 Java 语言写成的可重用组件。

所谓 JavaBean，是指符合如下标准的 Java 类：

>类是公共的

>有一个无参的公共的构造器

>有属性，且有对应的 get、set 方法

一、package 关键字的使用

1.为了更好的实现项目中类的管理，提供包的概念

2.使用 package 声明类或接口所属的包，声明在源文件的首行

3.包，属于标识符，遵循标识符的命名规则、规范(xxxyyyzzz)、“见名知意”

4.每"."一次，就代表一层文件目录。

补充：同一个包下，不能命名同名的接口、类。

不同的包下，可以命名同名的接口、类。

二、import 关键字的使用

import:导入

1. 在源文件中显式的使用 import 结构导入指定包下的类、接口

2. 声明在包的声明和类的声明之间
3. 如果需要导入多个结构，则并列写出即可
4. 可以使用"xxx.*"的方式，表示可以导入 xxx 包下的所有结构
5. 如果使用的类或接口是 java.lang 包下定义的，则可以省略 import 结构
6. 如果使用的类或接口是本包下定义的，则可以省略 import 结构
7. 如果在源文件中，使用了不同包下的同名的类，则必须至少有一个类需要以全类名的方式显示。
8. 使用"xxx.*"方式表明可以调用 xxx 包下的所有结构。但是如果使用的是 xxx 子包下的结构，则仍需要显式导入
9. import static:导入指定类或接口中的静态结构:属性或方法。

this 关键字的使用：

1.this 可以用来修饰、调用：属性、方法、构造器

2.this 修饰属性和方法：

this 理解为：当前对象　或　当前正在创建的对象

2.1　在类的方法中，我们可以使用"this.属性"或"this.方法"的方式，调用当前对象属性或方法。但是，通常情况下，我们都选择省略"this."。特殊情况下，如果方法的形参和类的属性同名时，我们必须显式的使用"this.变量"的方式，表明此变量是属性，而非形参。

2.2 在类的构造器中，我们可以使用"this.属性"或"this.方法"的方式，调用当前正在创建的对象属性或方法。但是，通常情况下，我们都选择省略"this."。特殊情况下，如果构造器的形参和类的属性同名时，我们必须显式的使用"this.变量"的方式，表明此变量是属性，而非形参。

3. this 调用构造器

① 我们在类的构造器中，可以显式的使用"this(形参列表)"方式，调用本类中指定的其他构造器

② 构造器中不能通过"this(形参列表)"方式调用自己

③ 如果一个类中有 n 个构造器，则最多有 n - 1 构造器中使用了"this(形参列表)"

④ 规定："this(形参列表)"必须声明在当前构造器的首行

⑤ 构造器内部，最多只能声明一个"this(形参列表)"，用来调用其他的构造器

Day_11

*** Eclipse 中的快捷键：**

* 1.补全代码的声明：alt + /
* 2.快速修复: ctrl + 1
* 3.批量导包：ctrl + shift + o
* 4.使用单行注释：ctrl + /
* 5.使用多行注释：　ctrl + shift + /
* 6.取消多行注释：ctrl + shift + \
* 7.复制指定行的代码：ctrl + alt + down　或　ctrl + alt + up

* 8.删除指定行的代码：ctrl + d
* 9.上下移动代码：alt + up　或 alt + down
* 10.切换到下一行代码空位：shift + enter
* 11.切换到上一行代码空位：ctrl + shift + enter
* 12.如何查看源码：ctrl + 选中指定的结构　或　ctrl + shift + t
* 13.退回到前一个编辑的页面：alt + left
* 14.进入到下一个编辑的页面(针对于上面那条来说的)：alt + right
* 15.光标选中指定的类，查看继承树结构：ctrl + t
* 16.复制代码：　ctrl + c
* 17.撤销：　ctrl + z
* 18.反撤销：　ctrl + y
* 19.剪切：ctrl + x
* 20.粘贴：ctrl + v
* 21.保存：　ctrl + s
* 22.全选：ctrl + a
* 23.格式化代码：　ctrl + shift + f
* 24.选中数行，整体往后移动：tab
* 25.选中数行，整体往前移动：shift + tab
* 26.在当前类中，显示类结构，并支持搜索指定的方法、属性等：ctrl + o
* 27.批量修改指定的变量名、方法名、类名等：alt + shift + r
* 28.选中的结构的大小写的切换：变成大写：　ctrl + shift + x
* 29.选中的结构的大小写的切换：变成小写：ctrl + shift + y
* 30.调出生成 getter/setter/构造器等结构：　alt + shift + s
* 31.显示当前选择资源(工程 or 文件)的属性：alt + enter
* 32.快速查找：参照选中的 Word 快速定位到下一个 ：ctrl + k
* 33.关闭当前窗口：ctrl + w
* 34.关闭所有的窗口：ctrl + shift + w
* 35.查看指定的结构使用过的地方：ctrl + alt + g
* 36.查找与替换：ctrl + f
* 37.最大化当前的 View：ctrl + m
* 38.直接定位到当前行的首位：home
* 39.直接定位到当前行的末位：end

面向对象的特征之二：继承性　　why?

一、继承性的好处：

① 减少了代码的冗余，提高了代码的复用性

② 便于功能的扩展

③ 为之后多态性的使用，提供了前提

二、继承性的格式：

class A extends B{}

A:子类、派生类、subclass

B:父类、超类、基类、superclass

2.1 体现：一旦子类 A 继承父类 B 以后，子类 A 中就获取了父类 B 中声明的所有的属性和方法。

特别的，父类中声明为 private 的属性或方法，子类继承父类以后，仍然认为获取了父类中私有的结构。

只有因为封装性的影响，使得子类不能直接调用父类的结构而已。

2.2 子类继承父类以后，还可以声明自己特有的属性或方法：实现功能的拓展。

子类和父类的关系，不同于子集和集合的关系。

extends：延展、扩展

三、Java 中关于继承性的规定：

1.一个类可以被多个子类继承。

2.Java 中类的单继承性：一个类只能有一个父类

3.子父类是相对的概念。

4.子类直接继承的父类，称为：直接父类。间接继承的父类称为：间接父类

5.子类继承父类以后，就获取了直接父类以及所有间接父类中声明的属性和方法

四、1. 如果我们没有显式的声明一个类的父类的话，则此类继承于 java.lang.Object 类

2. 所有的 java 类（除 java.lang.Object 类之外）都直接或间接的继承于 java.lang.Object 类

3. 意味着，所有的 java 类具有 java.lang.Object 类声明的功能。

Day_12

方法的重写(override / overwrite)

1.重写：子类继承父类以后，可以对父类中同名同参数的方法，进行覆盖操作

2.应用：重写以后，当创建子类对象以后，通过子类对象调用子父类中的同名同参数的方法时，实际执行的是子类重写父类的方法。

3. 重写的规定：

方法的声明： 权限修饰符 返回值类型 方法名(形参列表) throws 异常的类型{

//方法体

}

约定俗称：子类中的叫重写的方法，父类中的叫被重写的方法

① 子类重写的方法的方法名和形参列表与父类被重写的方法的方法名和形参列表相同

② 子类重写的方法的权限修饰符不小于父类被重写的方法的权限修饰符

>特殊情况：子类不能重写父类中声明为 private 权限的方法

③ 返回值类型：

>父类被重写的方法的返回值类型是 void，则子类重写的方法的返回值类型只能是 void

>父类被重写的方法的返回值类型是 A 类型，则子类重写的方法的返回值类型可以是 A 类或 A 类的子类

>父类被重写的方法的返回值类型是基本数据类型(比如：double)，则子类重写的方法的返回值类型必须是相同的基本数据类型(必须也是 double)

④ 子类重写的方法抛出的异常类型不大于父类被重写的方法抛出的异常类型（具体放到异常处理时候讲）

****

子类和父类中的同名同参数的方法要么都声明为非 static 的（考虑重写），要么都声明为 static 的（不是重写）。

面试题：区分方法的重载与重写

子类对象实例化的全过程

1. 从结果上来看：（继承性）

子类继承父类以后，就获取了父类中声明的属性或方法。

创建子类的对象，在堆空间中，就会加载所有父类中声明的属性。

2. 从过程上来看：

当我们通过子类的构造器创建子类对象时，我们一定会直接或间接的调用其父类的构造器，进而调用父类的父类的构造器，直到调用了 java.lang.Object 类中空参的构造器为止。正因为加载过所有的父类的结构，所以才可以看到内存中有父类中的结构，子类对象才可以考虑进行调用。

明确：虽然创建子类对象时，调用了父类的构造器，但是自始至终就创建过一个对象，即为 new 的子类对象。

super 关键字的使用

1.super 理解为：父类的

2.super 可以用来调用：属性、方法、构造器

3.super 的使用：调用属性和方法

3.1 我们可以在子类的方法或构造器中。通过使用"super.属性"或"super.方法"的方式，显式的调用父类中声明的属性或方法。但是，通常情况下，我们习惯省略"super."

3.2 特殊情况：当子类和父类中定义了同名的属性时，我们要想在子类中调用父类中声明的属性，则必须显式的

使用"super.属性"的方式，表明调用的是父类中声明的属性。

3.3 特殊情况：当子类重写了父类中的方法以后，我们想在子类的方法中调用父类中被重写的方法时，则必须显式的使用"super.方法"的方式，表明调用的是父类中被重写的方法。

4.super 调用构造器

4.1 我们可以在子类的构造器中显式的使用"super(形参列表)"的方式，调用父类中声明的指定的构造器

4.2 "super(形参列表)"的使用，必须声明在子类构造器的首行！

4.3 我们在类的构造器中，针对于"this(形参列表)"或"super(形参列表)"只能二选一，不能同时出现

4.4 在构造器的首行，没有显式的声明"this(形参列表)"或"super(形参列表)"，则默认调用的是父类中空参的构造器：super()

4.5 在类的多个构造器中，至少有一个类的构造器中使用了"super(形参列表)"，调用父类中的构造器

面向对象特征之三：多态性

1. 理解多态性：可以理解为一个事物的多种形态。

2.何为多态性：

对象的多态性：父类的引用指向子类的对象（或子类的对象赋给父类的引用）

3. 多态的使用：虚拟方法调用

有了对象的多态性以后，我们在编译期，只能调用父类中声明的方法，但在运行期，我们实际执行的是子类重写父类的方法。

总结：编译，看左边；运行，看右边。

4.多态性的使用前提： ① 类的继承关系 ② 方法的重写

5.对象的多态性，只适用于方法，不适用于属性（编译和运行都看左边）

Day_13

面试题： == 和 equals() 区别

一、回顾 ==使用：

== ：运算符

1. 可以使用在基本数据类型变量和引用数据类型变量中

2. 如果比较的是基本数据类型变量：比较两个变量保存的数据是否相等。（不一定类型要相同）

如果比较的是引用数据类型变量：比较两个对象的地址值是否相同.即两个引用是否指向同一个对象实体

补充： == 符号使用时，必须保证符号左右两边的变量类型一致。

二、equals()方法的使用：

1. 是一个方法，而非运算符

2. 只能适用于引用数据类型

3. Object 类中 equals()的定义：

```
public boolean equals(Object obj) {
        return (this == obj);
 }
```

说明：Object 类中定义的 equals()和==的作用是相同的：比较两个对象的地址值是否相同.即两个引用是否指向同一个对象实体

4. 像 String、Date、File、包装类等都重写了 Object 类中的 equals()方法。重写以后，比较的不是两个引用的地址是否相同，而是比较两个对象的"实体内容"是否相同。

5. 通常情况下，我们自定义的类如果使用 equals()的话，也通常是比较两个对象的"实体内容"是否相同。那么，我们就需要对 Object 类中的 equals()进行重写.重写的原则：比较两个对象的实体内容是否相同.

java.lang.Object 类

1.Object 类是所有 Java 类的根父类

2.如果在类的声明中未使用 extends 关键字指明其父类，则默认父类为 java.lang.Object 类

3.Object 类中的功能(属性、方法)就具有通用性。

属性：无

方法：equals() / toString() / getClass() /hashCode() / clone() / finalize()
wait() 、 notify()、notifyAll()

4. Object 类只声明了一个空参的构造器

Object 类中 toString()的使用：

1. 当我们输出一个对象的引用时，实际上就是调用当前对象的 toString()
2. Object 类中 toString()的定义：

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

3. 像 String、Date、File、包装类等都重写了 Object 类中的 toString()方法。
使得在调用对象的 toString()时，返回"实体内容"信息
4. 自定义类也可以重写 toString()方法，当调用此方法时，返回对象的"实体内容"

Java 中的 JUnit 单元测试

步骤：

1.选中当前工程 - 右键选择：build path - add libraries - JUnit 4 - 下一步
2.创建 Java 类，进行单元测试。
此时的 Java 类要求：① 此类是 public 的 ②此类提供公共的无参的构造器
3.此类中声明单元测试方法。
此时的单元测试方法：方法的权限是 public,没有返回值，没有形参
4.此单元测试方法上需要声明注解：@Test,并在单元测试类中导入：import org.junit.Test;
5.声明好单元测试方法以后，就可以在方法体内测试相关的代码。
6.写完代码以后，左键双击单元测试方法名，右键：run as - JUnit Test

说明：

1.如果执行结果没有任何异常：绿条
2.如果执行结果出现异常：红条

包装类的使用：

1.java 提供了 8 种基本数据类型对应的包装类，使得基本数据类型的变量具有类的特征
2.掌握的：基本数据类型、包装类、String 三者之间的相互转换

```
public class WrapperTest {

    //String 类型 --->基本数据类型、包装类：调用包装类的 parseXxx(String s)
    @Test
    public void test5(){
        String str1 = "123";
        //错误的情况：
//      int num1 = (int)str1;
//      Integer in1 = (Integer)str1;
        //可能会报 NumberFormatException
        int num2 = Integer.parseInt(str1);
        System.out.println(num2 + 1);
```

```
        String str2 = "true1";
        boolean b1 = Boolean.parseBoolean(str2);
        System.out.println(b1);
    }

    //基本数据类型、包装类--->String 类型：调用 String 重载的 valueOf(Xxx xxx)
    @Test
    public void test4(){

        int num1 = 10;
        //方式 1：连接运算
        String str1 = num1 + "";
        //方式 2：调用 String 的 valueOf(Xxx xxx)
        float f1 = 12.3f;
        String str2 = String.valueOf(f1);//"12.3"

        Double d1 = new Double(12.4);
        String str3 = String.valueOf(d1);
        System.out.println(str2);
        System.out.println(str3);//"12.4"

    }

    /
      JDK 5.0 新特性：自动装箱 与自动拆箱
     /
    @Test
    public void test3(){
//      int num1 = 10;
//      //基本数据类型-->包装类的对象
//      method(num1);

        //自动装箱：基本数据类型 --->包装类
        int num2 = 10;
        Integer in1 = num2;//自动装箱

        boolean b1 = true;
        Boolean b2 = b1;//自动装箱

        //自动拆箱：包装类--->基本数据类型
        System.out.println(in1.toString());

        int num3 = in1;//自动拆箱
```

```
    }

    public void method(Object obj){
        System.out.println(obj);
    }

    //包装类--->基本数据类型:调用包装类 Xxx 的 xxxValue()
    @Test
    public void test2(){
        Integer in1 = new Integer(12);

        int i1 = in1.intValue();
        System.out.println(i1 + 1);


        Float f1 = new Float(12.3);
        float f2 = f1.floatValue();
        System.out.println(f2 + 1);
    }

    //基本数据类型 --->包装类：调用包装类的构造器
    @Test
    public void test1(){

        int num1 = 10;
//      System.out.println(num1.toString());
        Integer in1 = new Integer(num1);
        System.out.println(in1.toString());

        Integer in2 = new Integer("123");
        System.out.println(in2.toString());

        //报异常
//      Integer in3 = new Integer("123abc");
//      System.out.println(in3.toString());

        Float f1 = new Float(12.3f);
        Float f2 = new Float("12.3");
        System.out.println(f1);
        System.out.println(f2);

        Boolean b1 = new Boolean(true);
        Boolean b2 = new Boolean("TrUe");
```

```
        System.out.println(b2);
        Boolean b3 = new Boolean("true123");
        System.out.println(b3);//false


        Order order = new Order();
        System.out.println(order.isMale);//false
        System.out.println(order.isFemale);//null
    }

}

class Order{

    boolean isMale;
    Boolean isFemale;
}
```

Day_14

static 关键字的使用

1.static:静态的

2.static 可以用来修饰：属性、方法、代码块、内部类

3.static 修饰属性：静态变量（或类变量）

3.1 属性，按是否使用 static 修饰，又分为：静态属性 vs 非静态属性(实例变量)

实例变量：我们创建了类的多个对象，每个对象都独立的拥有一套类中的非静态属性。当修改其中一个对象中的非静态属性时，不会导致其他对象中同样的属性值的修改。

静态变量：我们创建了类的多个对象，多个对象共享同一个静态变量。当通过某一个对象修改静态变量时，会导致其他对象调用此静态变量时，是修改过了的。

3.2 static 修饰属性的其他说明：

① 静态变量随着类的加载而加载。可以通过"类.静态变量"的方式进行调用

② 静态变量的加载要早于对象的创建。

③ 由于类只会加载一次，则静态变量在内存中也只会存在一份：存在方法区的静态域中。

④ 类变量 实例变量

类	yes	no
对象	yes	yes

3.3 静态属性举例：System.out; Math.PI;

4.使用 static 修饰方法：静态方法

① 随着类的加载而加载，可以通过"类.静态方法"的方式进行调用

②

	静态方法	非静态方法
类	yes	no
对象	yes	yes

③ 静态方法中，只能调用静态的方法或属性
非静态方法中，既可以调用非静态的方法或属性，也可以调用静态的方法或属性

5. static 注意点：
5.1 在静态的方法内，不能使用 this 关键字、super 关键字
5.2 关于静态属性和静态方法的使用，大家都从生命周期的角度去理解。

6. 开发中，如何确定一个属性是否要声明为 static 的？
> 属性是可以被多个对象所共享的，不会随着对象的不同而不同的。
> 类中的常量也常常声明为 static

开发中，如何确定一个方法是否要声明为 static 的？
> 操作静态属性的方法，通常设置为 static 的
> 工具类中的方法，习惯上声明为 static 的。 比如：Math、Arrays、Collections

main()方法的使用说明：
1. main()方法作为程序的入口
2. main()方法也是一个普通的静态方法
3. main()方法可以作为我们与控制台交互的方式。（之前：使用 Scanner）

*** 单例设计模式：**
1. 所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例。

2. 如何实现？
饿汉式 vs 懒汉式

3. 区分饿汉式 和 懒汉式
饿汉式：
坏处：对象加载时间过长。
好处：饿汉式是线程安全的

懒汉式：好处：延迟对象的创建。
目前的写法坏处：线程不安全。--->到多线程内容时，再修改

```
public class SingletonTest1 {
    public static void main(String[] args) {
//        Bank bank1 = new Bank();
//        Bank bank2 = new Bank();

        Bank bank1 = Bank.getInstance();
        Bank bank2 = Bank.getInstance();

        System.out.println(bank1 == bank2);
```

```
    }
}

//饿汉式
class Bank{

    //1.私有化类的构造器
    private Bank(){
    }

    //2.内部创建类的对象
    //4.要求此对象也必须声明为静态的
    private static Bank instance = new Bank();

    //3.提供公共的静态的方法，返回类的对象
    public static Bank getInstance(){
        return instance;
    }
}
```

单例模式的懒汉式实现

```
public class SingletonTest2 {
    public static void main(String[] args) {

        Order order1 = Order.getInstance();
        Order order2 = Order.getInstance();

        System.out.println(order1 == order2);

    }
}

class Order{
    //1.私有化类的构造器
    private Order(){

    }
    //2.声明当前类对象，没有初始化
    //4.此对象也必须声明为 static 的
    private static Order instance = null;
    //3.声明 public、static 的返回当前类对象的方法
    public static Order getInstance(){
        if(instance == null){
            instance = new Order();
```

```
            }
            return instance;
        }
}
```

final:最终的

1. final 可以用来修饰的结构：类、方法、变量

2. final 用来修饰一个类:此类不能被其他类所继承。
   比如：String 类、System 类、StringBuffer 类

3. final 用来修饰方法：表明此方法不可以被重写
   比如：Object 类中 getClass();

4. final 用来修饰变量：此时的"变量"就称为是一个常量
   4.1 final 修饰属性：可以考虑赋值的位置有：显式初始化、代码块中初始化、构造器中初始化
   4.2 final 修饰局部变量：尤其是使用 final 修饰形参时，表明此形参是一个常量。当我们调用此方法时，给常量形参赋一个实参。一旦赋值以后，就只能在方法体内使用此形参，但不能进行重新赋值。

static final 用来修饰属性：全局常量

类的成员之四：代码块（或初始化块）

1. 代码块的作用：用来初始化类、对象
2. 代码块如果有修饰的话，只能使用 static.
3. 分类：静态代码块 vs 非静态代码块
4. 静态代码块
   >内部可以有输出语句
   >随着类的加载而执行,而且只执行一次
   >作用：初始化类的信息
   >如果一个类中定义了多个静态代码块，则按照声明的先后顺序执行
   >静态代码块的执行要优先于非静态代码块的执行
   >静态代码块内只能调用静态的属性、静态的方法，不能调用非静态的结构
5. 非静态代码块
   >内部可以有输出语句
   >随着对象的创建而执行
   >每创建一个对象，就执行一次非静态代码块
   >作用：可以在创建对象时，对对象的属性等进行初始化
   >如果一个类中定义了多个非静态代码块，则按照声明的先后顺序执行
   >非静态代码块内可以调用静态的属性、静态的方法，或非静态的属性、非静态的方法

总结：由父及子，静态先行

对属性可以赋值的位置：

①默认初始化

②显式初始化/⑤在代码块中赋值

③构造器中初始化

④有了对象以后，可以通过"对象.属性"或"对象.方法"的方式，进行赋值

执行的先后顺序：① - ② / ⑤ - ③ - ④

Day_15

abstract 关键字的使用

1.abstract:抽象的

2.abstract 可以用来修饰的结构：类、方法

3. abstract 修饰类：抽象类

> 此类不能实例化

> 抽象类中一定有构造器，便于子类实例化时调用（涉及：子类对象实例化的全过程）

> 开发中，都会提供抽象类的子类，让子类对象实例化，完成相关的操作

4. abstract 修饰方法：抽象方法

> 抽象方法只有方法的声明，没有方法体

> 包含抽象方法的类，一定是一个抽象类。反之，抽象类中可以没有抽象方法的。

> 若子类重写了父类中的所有的抽象方法后，此子类方可实例化

若子类没有重写父类中的所有的抽象方法，则此子类也是一个抽象类，需要使用 abstract 修饰

abstract 使用上的注意点：

1.abstract 不能用来修饰：属性、构造器等结构

2.abstract 不能用来修饰私有方法、静态方法、final 的方法、final 的类

抽象类的匿名子类

```
public class PersonTest {

    public static void main(String[] args) {
        method(new Student());//匿名对象

        Worker worker = new Worker();
        method1(worker);//非匿名的类非匿名的对象

        method1(new Worker());//非匿名的类匿名的对象

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

        //创建了一匿名子类的对象：p
        Person p = new Person(){

            @Override
```

```
            public void eat() {
                System.out.println("吃东西");
            }

            @Override
            public void breath() {
                System.out.println("好好呼吸");
            }

        };

        method1(p);
        System.out.println("**********************");
        //创建匿名子类的匿名对象
        method1(new Person(){
            @Override
            public void eat() {
                System.out.println("吃好吃东西");
            }

            @Override
            public void breath() {
                System.out.println("好好呼吸新鲜空气");
            }
        });
    }
    public static void method1(Person p){
        p.eat();
        p.breath();
    }
    public static void method(Student s){

    }
}
class Worker extends Person{

    @Override
    public void eat() {
    }

    @Override
    public void breath() {
    }
```

```
}

//抽象类的应用：模板方法的设计模式
public class TemplateMethodTest {

	public static void main(String[] args) {
		BankTemplateMethod btm = new DrawMoney();
		btm.process();

		BankTemplateMethod btm2 = new ManageMoney();
		btm2.process();
	}
}
abstract class BankTemplateMethod {
	// 具体方法
	public void takeNumber() {
		System.out.println("取号排队");
	}

	public abstract void transact(); // 办理具体的业务 //钩子方法

	public void evaluate() {
		System.out.println("反馈评分");
	}

	// 模板方法，把基本操作组合到一起，子类一般不能重写
	public final void process() {
		this.takeNumber();

		this.transact();// 像个钩子，具体执行时，挂哪个子类，就执行哪个子类的实现代码

		this.evaluate();
	}
}

class DrawMoney extends BankTemplateMethod {
	public void transact() {
		System.out.println("我要取款!!! ");
	}
}

class ManageMoney extends BankTemplateMethod {
	public void transact() {
		System.out.println("我要理财！我这里有 2000 万美元!!");
```

```
    }
}
```

抽象类的应用：模板方法的设计模式

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

        SubTemplate t = new SubTemplate();

        t.spendTime();
    }
}

abstract class Template{
    //计算某段代码执行所需要花费的时间
    public void spendTime(){

        long start = System.currentTimeMillis();

        this.code();//不确定的部分、易变的部分

        long end = System.currentTimeMillis();

        System.out.println("花费的时间为：" + (end - start));

    }
    public abstract void code();
}
class SubTemplate extends Template{

    @Override
    public void code() {

        for(int i = 2;i <= 1000;i++){
            boolean isFlag = true;
            for(int j = 2;j <= Math.sqrt(i);j++){

                if(i % j == 0){
                    isFlag = false;
                    break;
                }
            }
            if(isFlag){
                System.out.println(i);
```

```
            }
        }

    }

}
```

接口的使用

1.接口使用 interface 来定义
2.Java 中，接口和类是并列的两个结构
3.如何定义接口：定义接口中的成员

3.1 JDK7 及以前：只能定义全局常量和抽象方法
>全局常量：public static final 的.但是书写时，可以省略不写
>抽象方法：public abstract 的

3.2 JDK8：除了定义全局常量和抽象方法之外，还可以定义静态方法、默认方法（略）
4. 接口中不能定义构造器的！意味着接口不可以实例化
5. Java 开发中，接口通过让类去实现(implements)的方式来使用.
如果实现类覆盖了接口中的所有抽象方法，则此实现类就可以实例化
如果实现类没有覆盖接口中所有的抽象方法，则此实现类仍为一个抽象类
6. Java 类可以实现多个接口 --->弥补了 Java 单继承性的局限性
格式：class AA extends BB implements CC,DD,EE
7. 接口与接口之间可以继承，而且可以多继承
8. 接口的具体使用，体现多态性
9. 接口，实际上可以看做是一种规范

面试题：抽象类与接口有哪些异同？

接口的使用

1.接口使用上也满足多态性
2.接口，实际上就是定义了一种规范
3.开发中，体会面向接口编程！

接口的应用：代理模式

```
public class NetWorkTest {
    public static void main(String[] args) {
        Server server = new Server();
//      server.browse();
        ProxyServer proxyServer = new ProxyServer(server);

        proxyServer.browse();

    }
```

```
}

interface NetWork{

	public void browse();

}

//被代理类
class Server implements NetWork{

	@Override
	public void browse() {
		System.out.println("真实的服务器访问网络");
	}

}
//代理类
class ProxyServer implements NetWork{

	private NetWork work;

	public ProxyServer(NetWork work){
		this.work = work;
	}

	public void check(){
		System.out.println("联网之前的检查工作");
	}

	@Override
	public void browse() {
		check();

		work.browse();

	}

}
```

类的内部成员之五：内部类

1. Java 中允许将一个类 A 声明在另一个类 B 中，则类 A 就是内部类，类 B 称为外部类

2.内部类的分类：成员内部类（静态、非静态） vs 局部内部类(方法内、代码块内、构

造器内)

3.成员内部类：

一方面，作为外部类的成员：

>调用外部类的结构

>可以被 static 修饰

>可以被 4 种不同的权限修饰

另一方面，作为一个类：

> 类内可以定义属性、方法、构造器等

> 可以被 final 修饰，表示此类不能被继承。言外之意，不使用 final，就可以被继承

> 可以被 abstract 修饰

4.关注如下的 3 个问题

4.1 如何实例化成员内部类的对象

4.2 如何在成员内部类中区分调用外部类的结构

4.3 开发中局部内部类的使用　见《InnerClassTest1.java》

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

		//创建 Dog 实例(静态的成员内部类):
		Person.Dog dog = new Person.Dog();
		dog.show();
		//创建 Bird 实例(非静态的成员内部类):
//		Person.Bird bird = new Person.Bird();//错误的
		Person p = new Person();
		Person.Bird bird = p.new Bird();
		bird.sing();

		System.out.println();

		bird.display("黄鹂");

	}
}

class Person{

	String name = "小明";
	int age;

	public void eat(){
		System.out.println("人：吃饭");
	}
```

```
    //静态成员内部类
    static class Dog{
        String name;
        int age;

        public void show(){
            System.out.println("卡拉是条狗");
//            eat();
        }

    }
    //非静态成员内部类
    class Bird{
        String name = "杜鹃";

        public Bird(){

        }

        public void sing(){
            System.out.println("我是一只小小鸟");
            Person.this.eat();//调用外部类的非静态属性
            eat();
            System.out.println(age);
        }

        public void display(String name){
            System.out.println(name);//方法的形参
            System.out.println(this.name);//内部类的属性
            System.out.println(Person.this.name);//外部类的属性
        }
    }


    public void method(){
        //局部内部类
        class AA{

        }
    }

    {
```

```
        //局部内部类
        class BB{

        }
    }

    public Person(){
        //局部内部类
        class CC{

        }
    }

}

**********************************************************************
public class SubClassTest {

    public static void main(String[] args) {
        SubClass s = new SubClass();

//          s.method1();
//          SubClass.method1();
            //知识点 1：接口中定义的静态方法，只能通过接口来调用。
            CompareA.method1();
            //知识点 2：通过实现类的对象，可以调用接口中的默认方法。
            //如果实现类重写了接口中的默认方法，调用时，仍然调用的是重写以后的方法
            s.method2();
            //知识点 3：如果子类(或实现类)继承的父类和实现的接口中声明了同名同参数的默认方法，
          //那么子类在没有重写此方法的情况下，默认调用的是父类中的同名同参数的方法。-->类优先原则
            //知识点 4：如果实现类实现了多个接口，而这多个接口中定义了同名同参数的默认方法，
            //那么在实现类没有重写此方法的情况下，报错。-->接口冲突。
            //这就需要我们必须在实现类中重写此方法
            s.method3();

    }

}

class SubClass extends SuperClass implements CompareA,CompareB{
```

```
    public void method2(){
        System.out.println("SubClass：上海");
    }

    public void method3(){
        System.out.println("SubClass：深圳");
    }

    //知识点 5：如何在子类(或实现类)的方法中调用父类、接口中被重写的方法
    public void myMethod(){
        method3();//调用自己定义的重写的方法
        super.method3();//调用的是父类中声明的
        //调用接口中的默认方法
        CompareA.super.method3();
        CompareB.super.method3();
    }
}

/*
 *
 * JDK8：除了定义全局常量和抽象方法之外，还可以定义静态方法、默认方法
 *
 */
public interface CompareA {

    //静态方法
    public static void method1(){
        System.out.println("CompareA:北京");
    }
    //默认方法
    public default void method2(){
        System.out.println("CompareA：上海");
    }

    default void method3(){
        System.out.println("CompareA：上海");
    }
}
public interface CompareB {

    default void method3(){
        System.out.println("CompareB：上海");
    }
```

```
}
```

Day_16

内部类：

```
public class InnerClassTest {

//    public void onCreate(){
//        int number = 10;
//        View.OnClickListern listener = new View.OnClickListener(){
//            public void onClick(){
//                System.out.println("hello!");
//                System.out.println(number);
//            }
//        }
//        button.setOnClickListener(listener);
//    }
    /*
     * 在局部内部类的方法中（比如：show）如果调用局部内部类所声明的方法(比如：method)中的局部变量(比如：num)的话,
     * 要求此局部变量声明为 final 的。
     * jdk 7 及之前版本：要求此局部变量显式的声明为 final 的
     * jdk 8 及之后的版本：可以省略 final 的声明
     *
    public void method(){
        //局部变量
        int num = 10;
        class AA{

                        public void show(){
//                  num = 20;
                    System.out.println(num);
                }
        }
    }
}
```

一、异常体系结构

```
 * java.lang.Throwable
 *      |-----java.lang.Error:一般不编写针对性的代码进行处理。
 *      |-----java.lang.Exception:可以进行异常的处理
 *          |------编译时异常(checked)
 *                  |-----IOException
 *                      |-----FileNotFoundException
```

```
 *                          |-----ClassNotFoundException
 *                  |------运行时异常(unchecked,RuntimeException)
 *                          |-----NullPointerException
 *                          |-----ArrayIndexOutOfBoundsException
 *                          |-----ClassCastException
 *                          |-----NumberFormatException
 *                          |-----InputMismatchException
 *                          |-----ArithmeticException
 *
 *
 *
 * 面试题：常见的异常都有哪些？举例说明
 */
public class ExceptionTest {

    //******************以下是编译时异常**************************
    @Test
    public void test7(){
//        File file = new File("hello.txt");
//        FileInputStream fis = new FileInputStream(file);
//
//        int data = fis.read();
//        while(data != -1){
//            System.out.print((char)data);
//            data = fis.read();
//        }
//
//        fis.close();

    }

    //******************以下是运行时异常**************************
    //ArithmeticException
    @Test
    public void test6(){
        int a = 10;
        int b = 0;
        System.out.println(a / b);
    }

    //InputMismatchException
    @Test
    public void test5(){
        Scanner scanner = new Scanner(System.in);
```

```
        int score = scanner.nextInt();
        System.out.println(score);

        scanner.close();
    }

    //NumberFormatException
    @Test
    public void test4(){

        String str = "123";
        str = "abc";
        int num = Integer.parseInt(str);

    }

    //ClassCastException
    @Test
    public void test3(){
        Object obj = new Date();
        String str = (String)obj;
    }

    //IndexOutOfBoundsException
    @Test
    public void test2(){
        //ArrayIndexOutOfBoundsException
//      int[] arr = new int[10];
//      System.out.println(arr[10]);
        //StringIndexOutOfBoundsException
        String str = "abc";
        System.out.println(str.charAt(3));
    }

    //NullPointerException
    @Test
    public void test1(){

//      int[] arr = null;
//      System.out.println(arr[3]);

        String str = "abc";
        str = null;
        System.out.println(str.charAt(0));
```

```
    }
}
```

一、异常的处理：抓抛模型

过程一："抛"：程序在正常执行的过程中，一旦出现异常，就会在异常代码处生成一个对应异常类的对象。并将此对象抛出。一旦抛出对象以后，其后的代码就不再执行。

关于异常对象的产生：① 系统自动生成的异常对象

② 手动的生成一个异常对象，并抛出（throw）

过程二："抓"：可以理解为异常的处理方式：① try-catch-finally ② throws

二、try-catch-finally 的使用

```
try{
        //可能出现异常的代码

}catch(异常类型 1 变量名 1){
        //处理异常的方式 1
}catch(异常类型 2 变量名 2){
        //处理异常的方式 2
}catch(异常类型 3 变量名 3){
        //处理异常的方式 3
}
....
finally{
        //一定会执行的代码
}
```

说明：

1. finally 是可选的。
2. 使用 try 将可能出现异常代码包装起来，在执行过程中，一旦出现异常，就会生成一个对应异常类的对象，根据此对象的类型，去 catch 中进行匹配
3. 一旦 try 中的异常对象匹配到某一个 catch 时，就进入 catch 中进行异常的处理。一旦处理完成，就跳出当前的 try-catch 结构（在没有写 finally 的情况）。继续执行其后的代码
4. catch 中的异常类型如果没有子父类关系，则谁声明在上，谁声明在下无所谓。

   catch 中的异常类型如果满足子父类关系，则要求子类一定声明在父类的上面。否则，报错
5. 常用的异常对象处理的方式： ① String getMessage() ② printStackTrace()
6. 在 try 结构中声明的变量，再出了 try 结构以后，就不能再被调用
7. try-catch-finally 结构可以嵌套

体会 1：使用 try-catch-finally 处理编译时异常，是得程序在编译时就不再报错，但是运行时仍可能报错。相当于我们使用 try-catch-finally 将一个编译时可能出现的异常，延迟到运行时出现。

体会 2：开发中，由于运行时异常比较常见，所以我们通常就不针对运行时异常编写 try-catch-finally 了。针对于编译时异常，我们说一定要考虑异常的处理。

异常处理的方式二：throws + 异常类型

1. "throws + 异常类型"写在方法的声明处。指明此方法执行时，可能会抛出的异常类型。

一旦当方法体执行时，出现异常，仍会在异常代码处生成一个异常类的对象，此对象满足 throws 后异常类型时，就会被抛出。异常代码后续的代码，就不再执行！

2. 体会：try-catch-finally:真正的将异常给处理掉了。throws 的方式只是将异常抛给了方法的调用者。 并没有真正将异常处理掉。

3. 开发中如何选择使用 try-catch-finally 还是使用 throws？

3.1 如果父类中被重写的方法没有 throws 方式处理异常，则子类重写的方法也不能使用 throws，意味着如果子类重写的方法中有异常，必须使用 try-catch-finally 方式处理。

3.2 执行的方法 a 中，先后又调用了另外的几个方法，这几个方法是递进关系执行的。我们建议这几个方法使用 throws 的方式进行处理。而执行的方法 a 可以考虑使用 try-catch-finally 方式进行处理。

try-catch-finally 中 finally 的使用：

1.finally 是可选的

2.finally 中声明的是一定会被执行的代码。即使 catch 中又出现异常了，try 中有 return 语句，catch 中有 return 语句等情况。

3.像数据库连接、输入输出流、网络编程 Socket 等资源，JVM 是不能自动的回收的，我们需要自己手动的进行资源的释放。此时的资源释放，就需要声明在 finally 中。

方法重写的规则之一：

子类重写的方法抛出的异常类型不大于父类被重写的方法抛出的异常类型

如何自定义异常类？

1. 继承于现有的异常结构：RuntimeException 、Exception
2. 提供全局常量：serialVersionUID
3. 提供重载的构造器

Error:

Java 虚拟机无法解决的严重问题。如：JVM 系统内部错误、资源耗尽等严重情况。比如：StackOverflowError 和 OOM。一般不编写针对性的代码进行处理。

```
public class ErrorTest {
    public static void main(String[] args) {
        //1.栈溢出：java.lang.StackOverflowError
//        main(args);
        //2.堆溢出：java.lang.OutOfMemoryError
        Integer[] arr = new Integer[1024*1024*1024];
```