Java基础

基本语法

关键字和保留字

- 关键字的定义和特点

  > 定义：被Java语言赋予了特殊含义，用作专门用途的字符串（单词）

  > 特点：关键字中所有字母都为小写

  > 用于定义数据类型的关键字：class interface enum byte short int long float double char boolean void

  > 用于定义流程控制的关键字：if else switch case default while do for break continue return

  > 用于定义访问权限修饰符的关键字：private protected public

  > 用于定义类，函数，变量修饰符的关键字：abstract final static synchronized

  > 用于定义类与类之间关系的关键字：extends implements

  > 用于定义建立实例及饮用实例，判断实例的关键字：new this super instanceof

  > 用于异常处理的关键字：try catch finally throw throws

  > 用于包的关键字：package import

  > 其他修饰符关键字：native strictfp transient volatile assert

- 保留字（reserved word）

  > 现有Java版本尚未使用，但以后版本可能会作为关键字使用，自己命名标识符时要避免使用这些保留字

  > goto、const

标识符

- 定义

  > Java对各种变量、方法和类等要素命名时使用的字符序列称为标识符

  > 技巧：凡是自己可以起名字的地方都叫标识符。

- 定义合法标识符规则

```
/*
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
    > 由26个英文字母大小写、0-9、_ 或 $ 组成
    > 数字不可以开头
    > 不可以使用关键字和保留字，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

    如果不遵守如上的规则，编译不通过！需要严格遵守

3.Java的命名规范
    包名：多单词组成时所有字母都小写：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的首字母大写：xxxYyyZzz
    常量名：所有字母都打写。多单词时每个单词用下划线连接：XXX_YYY_ZZZ

    如果不遵守如上规范，编译可以通过，但是建议遵守

4.注意
    1.在起名字时，为了提高阅读性，要尽量有意义，“见名知意”。
    2.Java采用unicode字符集，因此标识符也可以使用汉字声明，但是不建议使用
*/
public class IdentifierTest {
    public static void main(String[] args) {

        // 遵守规则
        int myNumber = 888;
        System.out.println(myNumber);

        // 不遵守规则
        int mynumber = 666;
        System.out.println(mynumber);

    }
}
```

变量

- 变量的概念

  > 内存中的一个存储区域

  > 该区域的数据可以在同一类型范围内不断变化

  > 变量是程序中最基本的存储单位。包含变量类型、变量名和存储的值

- 变量的作用

  > 用于在内存中保存数据

- 使用变量注意

  > Java中每个变量必须先声明，后使用
  >
  > 使用变量名来访问这块区域的数据
  >
  > 变量的作用域：其定义所在的一对{}内
  >
  > 变量只有在其作用域内才生效
  >
  > 同一作用域内，不能定义重名的变量

- Java基本的数据类型

```
/*
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：
        1、整型：byte short int long
        类型      占用存储空间          表数范围
        byte    1字节 = 8bit位      -128 ~ 127
        short   2字节 = 16bit位     -2的15次幂 ~ 2的15次幂 - 1
        int     4字节 = 32bit位     -2的31次幂 ~ 2的31次幂 - 1（约21亿）
        long    8字节 = 64bit位     -2的63次幂 ~ 2的63次幂 - 1

        bit：计算机中的最小存储单位。byte：计算机中基本存储单元
        Java的整型常量默认为int型，除非不足以表示较大的数，才使用long，声明long型 必须以‘l’或‘L’结尾

        2、浮点型：float double
        类型              占用存储空间     表数范围（E表示10，E38代表10的38次幂）
        单精度float       4字节           -3.403E38 ~ 3.403E38
        双精度double      8字节           -1.798E308 ~ 1.798E308

        float：单精度，尾数可以精确到7位有效数字。很多情况下，精度很难满足需求
        double：双精度，精度时float的两倍。通常采用此类型。
        Java的浮点型常量默认为double型，声明float型常量，必须以‘f’或‘F’结尾

        3、字符型：char（1字符 = 2字节）
        定义char型变量，通常使用一对''，内部只能写一个字符
        表示方式：1、声明一个字符 2、转义字符 3.unicode值来表示字符型常量

        4、布尔型：boolean
        只能取两个值之一：true 、 false
        常常在条件判断、循环结构中使用

    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

*/
public class VariableTest {

    public static void main(String[] args){

        // 整型：byte short int long

        byte num1 = -128;
        byte num2 = 127;

        // byte num3 = 128; 编译不通过，超过byte最大值
        System.out.println(num1);
        System.out.println(num2);

        short num3 = 128;
        System.out.println(num3);

        int num4 = 1234;
        System.out.println(num4);

        long num5 = 234543634L;
        System.out.println(num5);


        // 浮点型：float double
        double num6 = 3.14;
        System.out.println(num6);

        float num7 = 31.14F;
        System.out.println(num7);

        // 字符型：char
        // 1.声明一个字符
        char str1 = 'a';
        // str1 = 'ab'; 编译不通过
        System.out.println(str1);

        // 2.转义字符
        char str2 = '\n'; // 换行符
        char str3 = '\t'; // 制表符，相当于tab键

        System.out.print("hello" + str2);
        System.out.println("world");
```

```
        System.out.print("hello" + str3);
        System.out.println("world");

        // 3.unicode值来表示字符型常量
        char str4 = '\u0043';
        System.out.println(str4);

        // 布尔型boolean
        boolean b = true;
        System.out.println(b);

        // 例子
        boolean isCheck = true;
        if (isCheck) {
            System.out.println("success");
        } else {
            System.out.println("fail");
        }
    }
}
```

- 基本数据类型之间的运算规则

```
/*
基本数据类型之间的运算规则

前提：这里只讨论7种基本数据类型变量键的运算。不包含boolean类型的。
  1. 自动类型提升
        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
        byte、char、short --> int --> long --> float --> double

        特别的：当byte、char、short三种变量做运算时，结果为int类型

    2. 强制类型转换
        > 需要使用强转符： ()
        > 注意点：强制类型转换，可能导致精度损失。

说明：此时的容量大小指的是，表示的数的范围的大小。比如：float容量要大于long的容量
*/
public class VariableTest1 {
    public static void main(String[] args) {

        // 自动类型提升
        byte num1 = 1;
        int num2 = 130;

        // byte sum = num1 + num2; 编译不通过
```

```
        int sum1 = num1 + num2;
        System.out.println(sum1);

        long sum2 = num1 + num2;
        System.out.println(sum2);

        double sum3 = num1 + num2;
        System.out.println(sum3);

        // ******************** 特别的 ************************

        char str = 'a'; // a = 97
        byte num3 = 2;
        short num4 = 3;

        // char sum4 = str + num3; 编译不通过
        // short sum4 = str + num3; 编译不通过
        // short sum4 = num3 + num4; 编译不通过

        int sum4 = num3 + num4;
        int sum5 = str + num4;
        System.out.println(sum4);
        System.out.println(sum5);

        // ******************* 强制类型转换 ************************

        double num5 = 12.5;
        // 精度损失举例1
        int num6 = (int)num5; // 截断操作，只取小数点前面的整数，不会进行四舍五入
12
        System.out.println(num6);

        // 没有精度损失
        long num7 = 123;
        short num8 = (short)num7;
        System.out.println(num8);

        // 精度损失举例2
        int num9 = 128;
        byte num10 = (byte)num9;
        System.out.println(num10); // -128
    }
}
```

- 变量运算规则的两个特殊的情况

```
public class VariableTest2 {
    public static void main(String[] args) {
```

```
        // 1.编码情况1
        long num1 = 123213;  // long类型，应该在末尾以‘L’或‘l’结束，不加的话默认为 int类型
        System.out.println(num1);

        // long num2 = 1324234546543654764; 编译失败：过大的整数
        long num3 = 13242345465436547641;
        System.out.println(num3);


        // float num4 = 12.3; 编译失败，float类型，应该在末尾以‘f’或‘F’结束，不加的话默认为double类型，应该double > float，所以编译失败
        float num5 = 12.3F;
        System.out.println(num5);

        // 2.编码情况2
        // 整型常量，默认类型为int类型，浮点型常量，默认类型为double型
        byte num6 = 12;

        // byte num7 = num6 + 1; 编译失败，这里的1为int类型，不能用byte
        int num7 = num6 + 1;// 正确的
        System.out.println(num7);

        // float num8 = num6 + 12.3; 编译失败，这里的12.3为double类型，不能用float
        double num9 = num6 + 12.3;
        System.out.println(num9);
    }
}
```

- String类型变量

```
/*
String类型变量的使用
    1. String属于引用数据类型，翻译为：字符串
    2.声明String变量时，使用一对""
    3.String可以和8种基本数据类型变量做运算，且运算只能是连接运算：  +
    4.运算的结果仍然是String类型

*/
public class StringTest {
    public static void main(String[] args) {
        String str = "hello world";
        System.out.println(str);

        String str1 = "a";
        String str2 = "";
```

```
        // char str3 = ''; 编译不通过 , 必须有值

        // ****************************************

        int num = 101;
        String numberStr = "学号: ";
        String info = numberStr + num;
        System.out.println(info);

        boolean bb = true;
        String info1 = info + bb; // + : 连接运算
        System.out.println(info1);
    }
}
```

- 进制
  - 计算机中不同进制的使用说明

```
/*
计算机中不同进制的使用说明

对于整数，有四种表达方式:
    > 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至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

    十进制->二进制：除2取余的逆向排列
    例如：13->00001101

注意：计算机底层都以补码的方式来存储数据!
*/
public class BinaryTest {
    public static void main(String[] args) {
        int num1 = 0b110;
        int num2 = 110;
        int num3 = 0127;
        int num4 = 0x110A;

        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
        System.out.println(num4);
    }
}
```

运算符

- 算术运算符

```
/*
运算符之一：算术运算符

*/
public 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;
        System.out.println(result4); // 2.0

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

        double result6 = (double)num1 / num2;
        System.out.println(result6); // 2.4

        double result7 = (double)(num1 / num2);
        System.out.println(result7); // 2.0


        // %：取余运算 结果的符号与被模数的符号相同

        // (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);// a1 = 11, b1 =
11

        int a2 = 10;
        int b2 = a2++;
```

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

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

        // 问题:
        byte s2 = 127;
        s2++;
        System.out.println(s2); // -128 原理：二进制 0111 1111(127) + 1 -->
1000 0000(-128)

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

- 赋值运算符

```
/*
运算符之二：赋值运算符
    = += -= %= /= *=
    不会改变变量本身的数据类型
*/
public class SetValueTest {
    public static void main(String[] args) {

        // 写法1
        int num1 = 10;
        int num2 = 10;

        // 写法2
        int num3,num4;
        num3 = 11;
        num4 = 12;

        // 写法3
        int num5 = 15,num6 = 20;

        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
        System.out.println(num4);
        System.out.println(num5);
```

```
        System.out.println(num6);

        // *********************************

        int n = 10;
        n += 2;
        System.out.println(n);

        int n1 = 12;
        n1 -= 2;
        System.out.println(n1);

        int n2 = 10;
        n2 *= 2;
        System.out.println(n2);

        int n3 = 13;
        n3 %= 2;
        System.out.println(n3);

        int n4 = 12;
        n4 /= 2;
        System.out.println(n4);

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

- 比较运算符

```
/*
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 =
*/
public 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

    }
}
```

- 逻辑运算符

```
/*
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
    ! 取反
    ^ a = true b = true  => false
      a = false b = true => true
      a和b不同时，才为true
*/
public 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 ： 当符号左侧是true时，二者都会执行右边的运算
            不同点 ： 当符号左边时false时，&继续执行右侧的运算，&&不再执行符号右边的运
算
            开发中：推荐使用&&
        */

        // &
        boolean i = true;
        i = false;
        int num1 = 10;
        if (i & (num1++ > 0)) {
            System.out.println("success");
        } else {
            System.out.println("error");
        }
        System.out.println(num1);
```

```
        // &&
        boolean j = true;
        j = false;
        int num2 = 10;
        if (j && (num2++ > 0)) {
            System.out.println("success");
        } else {
            System.out.println("error");
        }
        System.out.println(num2);


        // 区分: | 和 ||
        /*
            相同点1: | 和 || 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 a = false;
        a = true;
        int b = 10;
        if (a | (b++ > 0)) {
            System.out.println("success");
        } else {
            System.out.println("error");
        }
        System.out.println("b = " + b);

        // ||
        boolean a1 = false;
        a1 = true;
        int b1 = 10;
        if (a1 || (b1++ > 0)) {
            System.out.println("success");
        } else {
            System.out.println("error");
        }
        System.out.println("b1 = " + b1);
    }
}
```

- 位运算符(了解)

```
运算符      运算          范例
```

```
<<        左移          3 << 2 = 12 --> 3*2*2=12
>>        右移          3 >> 1 = 1 --> 3/2 = 1
>>>       无符号右移    3 >>> 1 = 1 --> 3/2 = 1
&         与运算         6 & 3 = 2
|         或运算         6 | 3 = 7
^         异或运算       6 ^ 3 = 5
~         取反运算       ~6 = -7

位运算符的细节：
  <<：空位补0，被移除的高位丢弃，空缺位补0
  >>：被移位的二进制做高位是0，右移后，空缺位补0；最高位是1，空缺位补1。
  >>>：被移位二进制最高位无论是0或者1，空缺位都用0补。
  &：二进制位进行&运算时，只有1&1时结果是1，否则是0；
  |：二进制位进行|运算时，只有0|0时结果为0，否则是1
  ^：相同二进制位进行^运算，结果是0；1^1=0,0^0=0
     不相同二进制位^运算结果是1。1^0=1,0^1=1
  ~：正数取反，各二进制码按补码各位取反
     负数取反，各二进制码按补码各位取反

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

- 三元运算符

```
/*
运算符之六：三元运算符
1.结构：（条件表达式）？表达式1 ：表达式2
2.说明
    条件表达式的结果为boolean类型
    根据条件表达式真或假，决定执行表达式1，还是表达式2.
    如果表达式为true，则执行表达式1，如果表达式为false，则执行表达式2
    表达式1与表达式2要求是一致的
    三元运算符可以嵌套使用

3.凡是可以使用三元运算符的地方，都可以改写成if-else，反之，不成立

4.如果程序既可以使用三元运算符，右可以使用if-else结构，那么优先选择三元运算符，原因：简洁、执行效率高
*/
public class SanYuanTest {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 5;
```

```
        // 获取两个数的最大值
        int max = (num1 > num2) ? num1 : num2;
        System.out.println(max);

        double max1 = (num1 > num2) ? 2 : 1.0;
        System.out.println(max1);

        // int max2 = (num1 > num2) ? 2 : "num2大"; 编译错误，没有办法统一到同一个类型

        // ******************************************

        String max3 = (num1 > num2) ? "num1大" : "num2大";
        System.out.println(max3);

        num2 = 10;
        String max4 = (num1 > num2) ? "num1大" : ((num1 == num2) ? "相等" : "num2大");
        System.out.println(max4);
    }
}
```

流程控制

- 顺序结构

  程序从上到下逐行的执行，中间没有任何判断和跳转。

- 分支结构

  根据条件，选择性的执行某段代码。

  有if-else(条件判断结构)和switch-case两种分支结构。

```
/*
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实例化： Scanner scan = new Scanner(System.in);
    3.调用Scanner类的相关方法，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。
*/

import java.util.Scanner;
```

```
public class ScannerTest {
    public static void main(String[] args){
        // 实例化Scanner类
        Scanner scan = new Scanner(System.in);

        // 调用Scanner类的相关方法，来获取指定类型的变量

        // 字符串
        String str = scan.next();
        System.out.println(str);

        // 整型
        int num = scan.nextInt();
        System.out.println(num);

        // 浮点
        double weight = scan.nextDouble();
        System.out.println(weight);

        // 布尔
        boolean isTrue = scan.nextBoolean();
        System.out.println(isTrue);

        // 对于char类型的获取，Scanner没有提供相关的方法，只能获取一个字符串
        String gender = scan.next();
        char genderChar = gender.charAt(0);// 获取索引为0位置上的字符
        System.out.println(genderChar);
    }
}
```

- if-else

```
/*
例题：狗的前两年每一年相当于人类的10.5岁，之后每增加一年就增加四岁。那么5岁的狗相当于人类多少年龄呢?
应该是：10.5+10.5+4+4+4 = 33岁。
    编写一个程序，获取用户输入的狗的年龄，通过程序显示其相当于人类的年龄。如果用户输入负数，请显示一个提示信息
*/
import java.util.Scanner;

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

        System.out.println("请输入狗的年龄：");
        Scanner scanner = new Scanner(System.in);
        int dogAge = scanner.nextInt();
```

```
        if (dogAge <= 0) {
            System.out.println("请输入狗的正确年龄");
            return;
        }

        double dogLikePersonAge = 0;
        if (dogAge <= 2) {
            dogLikePersonAge = 10.5 * dogAge;
        } else if (dogAge > 2) {
            dogLikePersonAge = 10.5 * 2 + (4 * (dogAge - 2));
        }

        System.out.println("这条狗相当于人类的岁数为: " + dogLikePersonAge
+ '岁');
    }
}
```

- switch-case

```
结构:
  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;
      break;// 可以不加
  }

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。当调用完执行语句之后，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swtich-case结构中，系统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。

  7.如果switch-case结构中的多个case的执行语句相同，则可以考虑合并。
例题：
  int num = 78;
    switch(score / 10) {
      case 1:
      case 2:
      case 3:
      case 4:
      case 5:
        System.out.println("不及格");
        break;
      case 6:
      case 7:
      case 8:
      case 9:
      case 10:
        System.out.println("及格");
        break;
    }
```

- 循环结构

  根据循环条件，重复性的执行某段代码。

  有while、do...while、for三种循环语句。

  注：JDK1.5提供了foreach循环，方便的遍历集合、数组元素。

  - for

```
/*
For循环结构的使用
一、循环结构的四个要素
    1.初始化条件
    2.循环条件
    3.循环体
    4.迭代条件

例题1：
    输入两个正整数m 和 n ，求两个正整数的最大公约数和最小公倍数
    例如：12和20的最大公约数4，最小公倍数是60；
*/

import java.util.Scanner;

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

```
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入第一个正整数：");
        int m = scanner.nextInt();

        System.out.println("请输入第二个正整数：");
        int n = scanner.nextInt();

        // 求最大公约数
        // 求两个数中的小的数
        int min = (m > n) ? n : m;
        // 循环取出最大公约数
        for (int i = min; i >= 1; i--) {
            if (m % i == 0 && n % i == 0) {
                System.out.println("最大公约数为：" + i);
                break;
            }
        }

        // 求最小公倍数
        // 求两个数中的大的数
        int max = (m >= n) ? m : n;
        // 循环取出最小公倍数
        for (int i = max; i <= m *n; i++) {
            if (i % m == 0 && i % n == 0) {
                System.out.println("最小公倍数为：" + i);
                break;
            }
        }

    }
}

/*
例题2：求水仙花数，是一个三位数，各个位上的数字立方和等于其本身
    例如：153 = 1*1*1 + 3*3*3 + 5*5*5
*/
public class ForTest1 {
    public static void main(String[] args) {
        for (int i = 100; i < 1000; i++) {
            // 百位数
            int bai = i / 100;
            // 十位数
            int shi = i % 100 / 10;
            // 个位数
            int ge = i % 10;

            if (bai * bai * bai + shi * shi * shi + ge * ge * ge == i)
{
```

```
                System.out.println(i + "为水仙花数");
            }
        }

    }
}
```

- while

```
循环结构的四个要素
    1.初始化条件
    2.循环条件
    3.循环体
    4.迭代条件

  while格式:
  1
  while(2){
    3
    4
  }
```

- do-while

```
循环结构的四个要素
    1.初始化条件
    2.循环条件
    3.循环体
    4.迭代条件
结构:
  1
  do{
    3
    4
  } while(2);
```

```
/*
题目:从键盘读入不确定的整数,并判断读入的正数和负数的个数,输入0时结束程序

说明:
    1.不在循环条件部分限制次数的结构:for(;;) 或 while(true)
    2.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
*/
```

```java
import java.util.Scanner;

public class WhileTest {
    public static void main(String[] args){
        Scanner scanner = new Scanner(System.in);

        int positiveNumber = 0;// 记录正数的个数
        int negativeNumber = 0;// 记录负数的个数

        while(true) {

            int number = scanner.nextInt();

            if (number > 0) {
                positiveNumber++;
            } else if (number < 0) {
                negativeNumber++;
            } else {
                break;
            }

        }
        System.out.println("正数的个数为："+ positiveNumber);
        System.out.println("负数的个数为："+ negativeNumber);
    }
}
```

嵌套循环

```java
  /*
例题：输出下面的菱形
    *
   * *
  * * *
 * * * *
* * * * *
 * * * *
  * * *
   * *
    *
*/
public class ForTest {
    public static void main(String[] args) {

        // 上面5行
        for(int i = 1; i <= 5; i++) {
```

```
            // 循环输出*前面的空格
            for(int k = 1; k <= 5 - i; k++) {
                System.out.print(" ");
            }

            // 循环输出*号
            for(int j = 1; j <= i; j++) {
                System.out.print("* ");
            }

            System.out.println();
        }

        // 下面4行
        for(int i = 1; i <= 4; i++) {

            // 循环输出*前面的空格
            for(int k = 1; k <= i; k++) {
                System.out.print(" ");
            }

             // 循环输出*号
            for(int j = 1; j <= 5-i; j++) {
                System.out.print("* ");
            }

            System.out.println();
        }

    }
}


/*
例题2：九九乘法表
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6   3*3=9
4*1=4  4*2=8   4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7  7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
*/
public class NineTest {
    public static void main(String[] args){
        for(int i = 1; i <= 9; i++) {
            for(int j = 1; j <= i; j++) {
```

```
                int sum = i * j;
                System.out.print(i + "*" + j + "=" + sum + "  ");
            }
            System.out.println();
        }
    }
}



/*
例题：求100000里所有的质数，质数就是只能被1和它本身除开的数
*/
public class PrimeNumberTest {
    public static void main(String[] args) {

        // 记录开始执行的时间
        long startTime = System.currentTimeMillis();

        int count = 0;
        label:for(int i = 2; i <= 100000; i++) {
            for(int j = 2; j <= Math.sqrt(i); j++) {// j 被 i去除

                if (i % j == 0) { // i 被 j除进，表明i不为质数
                    continue label;//跳出当前label标签的当次循环
                }

            }
            // 能执行到此的数都是质数
            count++;
        }

        long endTime = System.currentTimeMillis();
        System.out.println("质数的个数为：" + count);
        System.out.println("所花费的时间为：" + (endTime - startTime));
    }
}
```

数组

数组的概述

```
/*
 * 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照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：
 *    > 数组是有序排列的
 *    > 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
 *    > 创建数组对象会在内存中开辟一整块连续的空间
 *    > 数组的长度一旦确定，就不能修改
 *
 *  4.数组的分类
 *   > 按照维数：一维数组、二维数组.....
 *   > 按照数组元素的类型：基于数据类型元素的数组、引用数据类型元素的数组
 *   >
 */
public class ArrayTest {
  public static void main(String[] args) {

  }
}
```

一维数组

```
/*  一维数组的使用
 *   1 一维数组的声明和初始化
 *   2 如何调用数组指定位置的元素
 *   3 如何获取数组的长度
 *   4 如何遍历数组
 *   5 数组元素的默认初始化值
 *     1️⃣ 数组元素是整型：0
 *     2️⃣ 数组元素是浮点型：0.0
 *     3️⃣ 数组元素是char型：0或'\u0000',而非'0'
 *     4️⃣ 数组元素是boolean型：false
 *
 *     5️⃣ 数组元素是引用数据类型：null
 *   6 数组的内存解析
 */
public class ArrayTest {
  public static void main(String[] args) {

    // 1.一维数组的声明和初始化
    int[] ids;// s声明

    // 1.1.静态初始化:数组的初始化和数组元素的赋值操作同时进行
```

```
    ids = new int[]{1001,1002,1003,1004};
    // 也是正确的写法
    int[] ids2 = {1,2{1,2,3,4,5,6};,3,4,5,6};
    int ids3[] = {1,2,3,4,5,6};

    // 1.2动态初始化：数组的初始化和数组元素的赋值分开进行
    String[] names = new String[3];

    // 2.如何调用数组指定位置的元素：通过角标的方式调用
    // 数组的角标（或索引）从0开始的，到数组的长度-1结束。
    names[0] = "张三";
    names[1] = "李四";
    names[2] = "王五";

    // 3.如何获取数组的长度
    // 属性 length
    System.out.println(ids.length); // 4
    System.out.println(names.length);// 3

    // 4.数组的遍历
    for (int i = 0; i < names.length; i++) {
      System.out.println(names[i]);
    }
  }
}
```

```
package com.hbx.ever;

import java.util.Scanner;

/*
 * 例题：从键盘读入学生成绩，找出最高分，并输出学生成绩等级。
 *       成绩 >= 最高分-10   等级为‘A’
 *       成绩 >= 最高分-20   等级为‘B’
 *       成绩 >= 最高分-30   等级为‘C’
 *       其余           等级为‘D’
 */

public class ArrayDemo {

  public static void main(String[] args) {

    // 1.使用Scanner读取学生个数
    Scanner scanner = new Scanner(System.in);
    System.out.print("请输入学生个数：");
    int studentNumber = scanner.nextInt();
```

```
    // 2.创建数组，存储学生成绩：动态初始化
    int[] studentScoreArr = new int[studentNumber];

    // 3.给数组中的元素赋值
    int maxScore = 0;
    System.out.print("请输入"+ studentNumber +"个学生成绩：");
    for (int i = 0; i < studentScoreArr.length; i++) {
      studentScoreArr[i] = scanner.nextInt();

       // 4.获取数组中的元素的最大值：最高分
       if (studentScoreArr[i] > maxScore) {
         maxScore = studentScoreArr[i];
       }
    }

    System.out.println("最高分为：" + maxScore);
    // 5.根据每个学生成绩与最高分的差值，得到每个学生的等级，并输出等级和成绩
    char level;
    for (int i = 0; i < studentScoreArr.length; i++) {
      if (studentScoreArr[i] >= maxScore - 10) {
        level = 'A';
      } else if (studentScoreArr[i] >= maxScore - 20) {
        level = 'B';
      } else if (studentScoreArr[i] >= maxScore - 30) {
        level = 'C';
      } else {
        level = 'D';
      }
      System.out.println("student " + i + " score is " + studentScoreArr[i] +
",grade is " + level);
    }

  }
}
```

多维数组

```
package com.hbx.java;

/*
 *  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
```

```
 *    1️⃣ 二维数组的声明和初始化
 *    2️⃣ 如何调用数组指定位置的元素
 *    3️⃣ 如何获取数组的长度
 *    4️⃣ 如何遍历二维数组
 *    5️⃣ 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：内存地址值
 *       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
 *    6️⃣ 数组的内存解析
 */

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:标准写法
    int[][] arr1 = new int[][] {{1,2,3}, {2,3}, {4,5,6}};
    // 也是正确的写法
    int arr4[][] = new int[][] {{1,2,3}, {2,3}, {4,5,6}};
    int[] arr5[] = {{1,2,3}, {2,3}, {4,5,6}};
    // 动态初始化1
    String[][] arr2 = new String[3][2];
    // 动态初始化2
    String[][] arr3 = new String[3][];

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

    arr3[1] = new String[2];
    System.out.println(arr2[1][0]);// null

    // 3.如何获取数组的长度
    System.out.println(arr4.length);// 3
    System.out.println(arr4[1].length); // 2

    // 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();
    }
```

```
  }
}
```

```
package com.hbx.exer;

/*
 * 例题：使用二维数组打印一个10行杨辉三角。
 *
 * 【提示】
 *  1.第一行有1个元素，第n行有n个元素
 *  2.每一行的第一个元素与最后一个元素都是1
 *  3.从第三行开始，对于非第一个元素和最后一个元素的元素。即：
 *    yangHui[i][j] = yangHui[i-1][j-1] + yangHui[i-1][j];
 */

public class YangHuiTest {
  public static void main(String[] args) {
    // 1.声明并初始化二维数组
    int[][] yangHui = new int[10][];

    // 2.给数组的元素赋值
    for (int i = 0; i < yangHui.length; i++) {
      // 二维数组每一行的个数
      yangHui[i] = new int[i + 1];

      // 给每一行的第一个元素与最后一个元素都赋值1
      yangHui[i][0] = yangHui[i][i] = 1;

      // 给第三行到第十行的元素赋值
      for (int j = 1; j < yangHui[i].length - 1; j++) {
        yangHui[i][j] = yangHui[i-1][j-1] + yangHui[i-1][j];
      }

    }

    // 3.遍历二维数组
    for (int i = 0; i < yangHui.length; i++) {
      for (int j = 0; j < yangHui[i].length; j++) {
        System.out.print(yangHui[i][j] + " ");
      }
      System.out.println();
    }
  }
}
```

算法的考查

```java
package com.hbx.java;

/*
 * 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 ArrayTest {
  public static void main(String[] args) {

    String[] arr = new String[] {"AA","BB","CC","DD","EE","FF","GG"};

    // 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
    // 方式1
    for (int i = 0; i < arr.length / 2; i++) {
      String temp = arr[i];
      arr[i] = arr[arr.length - 1 - i];
      arr[arr.length - 1 - i] = temp;
    }

    // 方式2
    for (int i = 0,j = arr.length - 1; i < j; i++,j--) {
      String temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
    }

    // 遍历展示
    for (int i = 0; i < arr.length; i++) {
      System.out.println(arr[i] + '\t');
    }

    // 查找（或搜索）
    // 线性查找
    String dest = "CC";
    boolean isFlag = true;

    for (int i = 0; i < arr.length; i++) {
      if (dest.equals(arr[i])) {
        System.out.println("找到指定元素。位置为：" + i);
        isFlag = false;
```

```
        break;
      }
    }

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

    // 二分法查找：
    // 前提：所要查找的数组必须有序
    int[] arr2 = new int[] {-56,-34,0,23,56,67,77,88,90};

    int searchNumber = 78;

    int startKey = 0;// 初始的首索引
    int endKey = arr2.length - 1; // 初始的末索引
    boolean isTrue = true;

    while (startKey <= endKey) {
      // 查找中间的索引
      int middle = (endKey + startKey) / 2;

      if (arr2[middle] == searchNumber) {
        System.out.println("要查找的值的索引为：" + middle);
        isTrue = false;
        break;
      } else if (arr2[middle] > searchNumber) {
        endKey = middle - 1;
      } else { // arr2[middle] < searchNumber
        startKey = middle + 1;
      }
    }

    if (isTrue) {
      System.out.println("很遗憾，没有找到该数字的位置");
    }

  }
}
```

```
package com.hbx.java;

/*
 * 数组的冒泡排序的实现
 */
public class BubbleSortTest {
  public static void main(String[] args) {
    int[] arr = new int[] {-32,-43,-153,23,12,45,22,89,56,55};
```

```
    // 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");
    }


  }
}
```

```
package com.hbx.java;

/**
 * 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。
 * @author
 *
 */
public class QuickSortTest {
  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));
  }
}
```

Arrays工具类的使用

```
package com.hbx.java;

import java.util.Arrays;

/*
 * java.util.Arrays类即为操作数组的工具类，包含了用来操作数组（比如排序和搜索）的各种方法
 *
 * 常用的几个方法
 * boolean equals(int[] a,int[] b)     判断两个数组是否相等
 * String toString(int[] a)        输出数组信息
 * void fill(int[] a,int val)        将指定值填充到数组中
 * void sort(int[] a)            对数组进行排序
 * int binarySearch(int[] a,int[] key)   对排序后的数组进行二分法检索指定的值
 *
 */
public class ArraysUtils {
  public static void main(String[] args) {

    int[] arr1 = new int[] {1,2,3,4};
    int[] arr2 = new int[] {1,5,3,4};
    // boolean equals(int[] a,int[] b)     判断两个数组是否相等
```

```
    boolean isTrue = Arrays.equals(arr1, arr2);
    System.out.println(isTrue);

    // String toString(int[] a)       输出数组信息
    System.out.println(Arrays.toString(arr1));

    // void fill(int[] a,int val)       将指定值填充到数组中
    Arrays.fill(arr1, 5);
    System.out.println(Arrays.toString(arr1));

    // void sort(int[] a)           对数组进行排序
    Arrays.sort(arr2);
    System.out.println(Arrays.toString(arr2));

    // int binarySearch(int[] a,int[] key)  对排序后的数组进行二分法检索指定的值
    int[] arr3 = new int[] {-12,-7,0,6,13,15,35};
    int index = Arrays.binarySearch(arr3, 15);
    System.out.println(index);
  }
}
```

数组中的常见异常

```
package com.hbx.java;

/*
 * 数组中的常见异常
 *
 * 1.数组角标越界的异常：ArrayIndexOutOfBoundsException
 *
 * 2.空指针异常：NullPointerException
 *
 */

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

    // 1.数组角标越界的异常：ArrayIndexOutOfBoundsException
    int[] arr = new int[] {1,2,3,4,6};

    // 越界举例1:循环数超过数组的个数
//    for (int i = 0; i <= arr.length; i++) {
//      System.out.println(arr[i]);
//    }

    // 越界举例2:
//    System.out.println(arr[-2]);
```

```
    // 前面代码报错后，下面的代码不会继续执行
    System.out.println("hello");

    // 2.空指针异常：NullPointerException
    // 情况1：
//    int[] arr1 = new int[] {1,2,3};
//    arr1 = null;
//    System.out.println(arr1[0]);

    // 情况2：
//    int[][] arr2 = new int[3][];
//    System.out.println(arr2[0][0]);

    // 情况3：
    String[] arr3 = new String[] {"AA","BB","CC"};
    arr3[0] = null;
    System.out.println(arr3[0].toString());
  }
}
```

面向对象-上

面向过程（POP）与面向对象（OOP）

> 二者都是一种思想，面向对象是相对于面向过程而言的。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。面向对象，将功能封装进对象，强调具备了功能的对象，以类/对象为最小单位，考虑谁来做。

> Java类及类的成员：属性、方法、构造器；代码块、内部类

> 面向对象的三大特征：继承性、封装性、多态性、（抽象性）

> 其他关键字：this super static final abstract interface package import等

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）
>
> 1、面向对象程序设计的重点是类的设计
>
> 2、设计类，就是设计类的成员

```
package com.hbx.java;

/*
 * 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型的属性。（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
 *  堆：存放对象实例
 *  栈：存储局部变量
 *  方法区：存储类信息、常量、静态变量、即时编译器编译后的代码
 */

// 测试类
public class PersonTest {
  public static void main(String[] args) {
    // 2.创建Person类的对象
    Person person = new Person();

    // 3.通过“对象.属性”或“对象.方法”调用对象的结构

    // 调用属性
    person.name = "Tom";
    person.isMale = true;
    System.out.println(person.name);

    // 调用方法
    person.eat();
    person.sleep();

    // ****************************************
    Person person1 = new Person();
    System.out.println(person1.name); // null
    System.out.println(person1.isMale); // false

    // ****************************************

    // 将person变量保存的对象地址值赋给person2，导致person和person2指向了堆空间中的同一个对象实体。
    Person person2 = person;
    System.out.println(person2.name); // Tom

    person2.age = 10;
    System.out.println(person.age); // 10
```

```
    }
}

// 1.创建类、设计类的成员
class Person{

    // 属性：对应类中的成员变量
    String name;
    int age = 1;
    boolean isMale;

    // 方法：对应类中的成员方法

    public void eat() {
        System.out.println("人可以吃饭");
    }

    public void sleep() {
        System.out.println("人可以睡觉");
    }
}
```

类中属性的使用

```
package com.hbx.java;

/*
 * 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 = 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（默认状态）、protected
 *         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，都有默认初始值。
```

```
 *          整型（byte、short、int、long）：0
 *          浮点型（float、double）：0.0
 *          字符串（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。
 *            特别的：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
 *
 */
public class UserTest {
  public static void main(String[] args) {
    User user = new User();
    System.out.println(user.name); // null
    System.out.println(user.age); // 0
    System.out.println(user.isMale); // false
  }
}


class User{
  // 属性（成员变量）
  public String name;
  public int age;
  protected boolean isMale;

  public void talk(String language) { // language:局部变量（形参）
    System.out.println("我们说的语言是"+ language);
  }

  public void eat() {
    String food = "巧克力"; // 局部变量
    System.out.println("有人喜欢吃" + food);
  }

}
```

类中方法的声明和使用

```
package com.hbx.java;

/*
 * 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：nextInt() ...
*     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
*         }
*
*  3.说明：
*     3.1 权限修饰符：
*         Java规定的4种权限修饰符：private、public、缺省、protected -->封装性再细讲
*     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。
*           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了，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?
*           1️⃣ 题目要求
*           2️⃣ 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?
*           1️⃣ 题目要求
*           2️⃣ 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：使用在方法体内
*     2.作用：1️⃣ 结束方法
*           2️⃣ 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
*
```

```
 */
public class CustomerTest {
  public static void main(String[] args) {

  }
}

// 客户类
class Customer{

  // 属性（成员变量）
  String name;
  int age;
  boolean isMale;

  // 方法
  public void eat() {
    System.out.println("eat");
  }

  public void sleep(int hour) {
    System.out.println("我们一天睡" + hour + "小时");
  }

  public String getName() {
    return name;
  }

  public String getNation(String nation) {
    String info = "我们的国籍是" + nation;
    return info;
  }
}
```

练习-对象数组

```
package com.hbx.exer;

/*
 * 对象数组题目：
 * 定义类Student，包含三个属性：学号number(int)、年级state（int），成绩score（int）。创建20个学生对象，学号为1到20
 * 年级和成绩都由随机数确定。
 *
 * 问题1:打印出3年级（state值为3）的学生信息。
 * 问题2:使用冒泡排序按学生成绩排序，并遍历所有学生信息
 *
 * 提示：
```

```
 *  1.生成随机数：Math  .random(),返回值类型double；
 *  2.四舍五入取整Math.round(double d),  返回值类型long
 *
 */
public class StudentTest {
  public static void main(String[] args) {
    Student[] student = new Student[20];

    for (int i = 0; i < student.length; i++) {
      student[i] = new Student();
      student[i].number = (i + 1); // 学号

      // [1, 6] 年级1-6
      student[i].state = (int)(Math.random() * (6 - 1 + 1) + 1);

      // [0, 100] 分数从0到100
      student[i].score = (int)(Math.random() * (100 - 0 + 1));
    }

    // 循环显示该数组
    StudentTest stuTest = new StudentTest();
    stuTest.print(student);

    System.out.println("**************************************************");
    // 打印出3年级（state值为3）的学生信息。
    stuTest.printState(student, 3);

    System.out.println("**************************************************");
    // 使用冒泡排序按学生成绩排序，并遍历所有学生信息
    stuTest.sortStudent(student);
    stuTest.print(student);

  }

  /**
   *
   * @Description  遍历数组操作
   * @author hbx
   * @date 2020年12月16日下午10:21:35
   * @param stus
   */
  public void print(Student[] stus) {
    for (int i = 0; i < stus.length; i++) {
      System.out.println(stus[i].stuMsg());
    }
  }

  /**
   *
```

```
    * @Description 打印出3年级（state值为3）的学生信息。
    * @author hbx
    * @date 2020年12月16日下午10:25:29
    * @param stus
    * @param state
    */
   public void printState(Student[] stus, int state) {
     for (int i = 0; i < stus.length; i++) {
       if (stus[i].state == state) {
         System.out.println(stus[i].stuMsg());
       }
     }
   }

   /**
    *
    * @Description 使用冒泡排序按学生成绩排序，并遍历所有学生信息
    * @author hbx
    * @date 2020年12月16日下午10:32:30
    * @param stus
    */
   public void sortStudent(Student[] stus) {
     for (int i = 0; i < stus.length - 1; i++) {
       for (int j = 0; j < stus.length - i - 1; j++) {
         if (stus[j].score > stus[j + 1].score) {
           Student temp = stus[j];
           stus[j] = stus[j + 1];
           stus[j + 1] = temp;
         }
       }
     }
   }


}

/*
 * 创建student类
 */
class Student{

   int number;
   int state;
   int score;

   public String stuMsg() {
     return ("学号: " + number + " 年级: " + state + " 分数: " + score);
   }
```

```
}
```

万事万物皆对象以及匿名对象

```
package com.hbx.java;

/*
 * 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类
 *    >  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) {

    PhoneMall mall = new PhoneMall();
    mall.show(new Phone());
  }
}

// 手机商场类
class PhoneMall{

  public void show(Phone phone) {
    phone.getPhonePrice();
  }

}

// 手机类
class Phone {

  double price;

  public void getPhonePrice() {
```

```
    System.out.println("该手机价格为1999");
  }

}
```

自定义数组类

```
package com.hbx.java;

/*
 * 自定义数组类
 *
 */
public class ArrayUtil {
  // 求数组的最大值
  public int getMax(int[] arr) {
    int maxValue = arr[0];

    for (int i = 0; i < arr.length; i++) {
      if (maxValue < arr[i]) {
        maxValue = arr[i];
      }
    }

    return maxValue;
  }

  // 求数组的最小值
  public int getMin(int[] arr) {
    int minValue = arr[0];

    for (int i = 0; i < arr.length; i++) {
      if (minValue > arr[i]) {
        minValue = arr[i];
      }
    }

    return minValue;
  }

  // 求数组的总和
  public int getSum(int[] arr) {
    int sumValue = 0;
    for(int i =0; i < arr.length; i++) {
      sumValue += arr[i];
    }
    return sumValue;
  }
```

```java
// 求数组的平均值
public int getAvg(int[] arr) {
  return getSum(arr) / arr.length;
}

// 反转数组
public void reverse(int[] arr) {
  for (int i = 0; i < arr.length / 2; i++) {
    int temp = arr[i];
    arr[i] = arr[arr.length - i - 1];
    arr[arr.length - i - 1] = temp;
  }
}

// 复制数组
public int[] copy(int[] arr) {
  int[] arr1 = new int[arr.length];
  for (int i = 0; i < arr1.length; i++) {
    arr1[i] = arr[i];
  }
  return arr1;
}

// 数组排序
public void sort(int[] arr) {
  // 冒泡排序
  for (int i = 0; i < arr.length - 1; i++) {
    for (int j = 0; j < arr.length - 1 - i; j++) {
      int temp = arr[j];
      arr[j] = arr[j + 1];
      arr[j + 1] = temp;
    }
  }
}

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

// 查找指定元素
public int getIndex(int[] arr, int dest) {
  // 线性查找（二分法需要数组是有序的）
  for (int i = 0; i < arr.length; i++) {
    if (dest == arr[i] ) {
      return i;
```

```
        }
      }
      return -1;
    }
}
```

方法的重载

- 重载的概念

  > 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。

- 重载的特点

  > 与返回值类型无关，只看参数列表，且参数列表必须不同。（参数个数或参数类型）。调用时，根据方法参数列表的不同来区别。

- 举例

```
package com.hbx.java;

/*
 *
 * 方法的重载
 *
 *  1."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类的sort()
 *
 *  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.再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
 */
public class OverLoadTest {
  // 如下的四个方法构成了重载
  public void getSum(int i, int j) {

  }

  public void getSum(double i, double j) {

  }

  public void getSum(String s, int i) {

  }
```

```
  public void getSum(int i, String s) {

  }
}
```

可变个数的形参

```
package com.hbx.java;
/*
 * 可变个数形参的方法
 *
 * 1.jdk5.0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 ... 形参
 *    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.....多个
 *    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 class MethodArgsTest {
  public static void main(String[] args) {

    MethodArgsTest test = new MethodArgsTest();
    test.show(1);
    test.show("123");
    test.show("aa", "bb"); =>等同于 test.show(new String[]{"aa", "bb"})

  }


  public void show(int i) {

  }

  public void show(String s) {

  }

  public void show(String ... strs) {
    System.out.println("show(String ... strs)");

    for (int i = 0; i < strs.length; i++) {
      System.out.println(strs[i]);
    }
```

```
  }

   // 等同于上一个方法
//  public void show(String[] strs) {
//
//  }

  // 2.5
  public void show(int i, String ... strs) {
    System.out.println("show(String ... strs)");
  }
}
```

变量赋值

- 结论

  > 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。

- 值传递

  > 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

  > 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

递归（recursion）方法

- 定义

  > 一个方法内调用它自身

```
package com.hbx.java;

/*
 * 递归方法的使用
 *
 *
 */
public class RecursionTest {

  public static void main(String[] args) {

    // 计算1 - 100之间所有自然数的和
    // 方式1
```

```
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
      sum += i;
    }
    System.out.println(sum);

    // 方式2:
    RecursionTest test = new RecursionTest();
    int sum1 = test.getSum(100);
    System.out.println(sum1);
  }

  /**
   *
   * @Description 计算1到n之间所有自然数的和
   * @author hbx
   * @date 2020年12月20日下午3:00:24
   * @param n
   * @return
   */
  public int getSum(int n) {
    if (n == 1) {
      return 1;
    } else {
      return n + getSum(n - 1);
    }
  }
}
```

- 斐波那契数列

```
package com.hbx.java;

/*
 * 递归方法的使用
 *
 *
 * 输入一个数据n，计算斐波那契数列的第n个值
 * 1  1 2 3 5 8 13  21  34  55
 *
 * 规律：一个数等于前两个数之和
 * 要求：计算斐波那契数列的第n个值，并将整个数列打印出来
 *
 */
public class RecursionTest {

  public static void main(String[] args) {
    RecursionTest test = new RecursionTest();
```

```
    //斐波那契数列
    // 计算第n个值
    int num2 = test.fibonacci(10);
    System.out.println(num2);
    // 打印整个数列
    test.printFibonacci(10);
  }

  /**
   *
   * @Description 斐波那契数列 求第n个值
   * @author hbx
   * @date 2020年12月20日下午3:30:41
   * @param n
   * @return
   */
  public int fibonacci(int n) {
    if (n == 1) {
      return 1;
    } else if (n == 2) {
      return 1;
    } else {
      return (fibonacci(n - 1) + fibonacci(n - 2));
    }
  }

  /**
   *
   * @Description 打印出整个数组
   * @author hbx
   * @date 2020年12月20日下午3:30:21
   * @param n
   */
  public void printFibonacci(int n) {
    for (int i = 1; i <= n; i++) {
      System.out.print(fibonacci(i) + "\t");
    }

  }
}
```

封装与隐藏

- 封装性的体现

> 1️⃣ 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）；
>
> 2️⃣ 不对外暴露的私有的方法

> 3️⃣ 单例模式
>
> ……

- 权限修饰符

> Java规定的四种权限（从小到达排序）：private、缺省、protected、public

修饰符	类内部	同一个包	不同包的子类	同一个工程
private	yes			
缺省	yes	yes		
protected	yes	yes	Yes	
public	yes	yes	yes	yes

> 对于class的权限修饰符只可以用public和default（缺省）。
>
> 1️⃣ public类可以在任意地方被访问
>
> 2️⃣ default类只可以被同一个包内部的类访问

> 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
>
> 修饰类的话，只能使用缺省和public

构造器（构造方法）

- 作用

> 创建对象、初始化对象的信息

- 说明

> 1️⃣ 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
>
> 2️⃣ 定义构造器的格式：权限修饰符 类名（形参列表）{}
>
> 3️⃣ 一个类中定义的多个构造器，彼此构成重载
>
> 4️⃣ 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
>
> 5️⃣ 一个类中，至少会有一个构造器

- 练习

```
package com.java.exer;

/*
 * 编写两个类，TriAngle和TriAngleTest，其中TriAngle类中声明私有的底边长base和高height，同时声明公共方法访问私有变量
 * 此外，提供必要的构造器。另一个类中使用这些公共方法，计算三角形的面积
 *
 *
 */
```

```java
public class TriAngle {

  private double base; // 底边长
  private double height; // 高

  // 构造器
  public TriAngle() {

  }

  // 构造器
  public TriAngle(double b, double h) {
    base = b;
    height = h;
  }

  public void setBase(double b) {
    base = b;
  }

  public double getBase() {
    return base;
  }

  public void setHeight(double h) {
    height = h;
  }

  public double getHeight() {
    return height;
  }

  public double angle() {
     return base * height / 2;
  }
}
```

- 总结：属性赋值的先后顺序

  > 1️⃣ 默认初始化值
  >
  > 2️⃣ 显式初始化
  >
  > 3️⃣ 构造器中初始化
  >
  > 4️⃣ 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
>
> 1️⃣ 类是公共的
>
> 2️⃣ 有一个无参的公共的构造器
>
> 3️⃣ 有属性，且有对应的get、set方法

关键字-this的使用

```
package com.hbx.java;

/*
 * this关键字的使用：
 * 1.this可以修饰、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
 *    3.1 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
 *    3.2 构造器中不能通过"this(形参列表)"方式调用自己
 *    3.3 如果一个类中有n个构造器，则最多有n - 1构造器中可以使用"this(形参列表)"
 *    3.4 规定："this(形参列表)"必须声明在当前构造器的首行
 *    3.5 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
 *
 */
public class PersonTest {
  public static void main(String[] args) {
    Person test = new Person();

    test.setAge(11);

    int age = test.getAge();
    System.out.println(age);

    Person test1 = new Person("tom");
    System.out.println(test1.getName());

  }
```

```
}

class Person{
  private String name;
  private int age;

  public Person() {
    this.eat();// 调用方法
  }

  public Person(String name) {
    this(); // 调用构造器
    this.name = name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name; // 或 this.name 这里的this.可以省略
  }

  public void setAge(int age) {
    this.age = age;
  }

  public int getAge() {
    return age; // 或 this.age 这里的this.可以省略
  }

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

关键字-package的使用

```
package com.hbx.java;
/*
 * 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
 *
 */
public class PackageImportTest {

}
```

关键字-import的使用

```
package com.hbx.java;
/*
 * 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.如果使用"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
 */
public class PackageImportTest {

}
```

面向对象-中

继承性

- 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。

> 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

- 方法的重写（override / overwrite）

1、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，当创建子类对象后，通过子类对象调用子父类中的同名参数的方法时，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

1️⃣ 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

2️⃣ 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

3️⃣ 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引用类型（比如Object），则子类重写的方法的返回值类型可以是该类或该类的子类（可以为Object、String）

> 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（也必须是：double）

4️⃣ 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

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）。

- 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

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，我们想再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：当子类中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"super(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

- 子类对象实例化过程

> 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

多态性

- 理解多态性

> 可以理解为一个事物的多种形态

- 何为多态性

> 父类的引用指向子类的对象（或子类的对象赋值给父类的引用）

- 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，但是在运行期，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

- 多态性的使用前提

> 1️⃣ 要有类的继承关系
>
> 2️⃣ 方法的重写

- 注意：

> 对象的多态性，只适用于方法，不适应于属性

- 如何才能调用子类特有的属性和方法?

> 向下转型：使用强制类型转换符。

- instanceof关键字的使用

> a instanceof A：判断对象a是否是类A的实例。如果是，返回true；如果不是，返回false。

> 使用情镜：为了避免在向下转型时出现ClassCastException的异常，我们在向下转型之前，先进行instanceof的判断，一旦返回true，就进行向下转型。如果返回false，则不进行向下转型。
>
> 如果a instanceof A 返回true，a instanceof B也返回true，则类B是类A的父类。

- 子类继承父类

> 1️⃣ 若子类重写了父类方法，就意味着子类里定义的方法彻底覆盖了父类里的同名方法，系统将不可能把父类里的方法转移到子类中：编译看左边，运行看右边
>
> 2️⃣ 对于实例变量则不存在这样的现象，即使子类里定义了与父类完全相同的实例变量，这个实例变量依然不可能覆盖父类中定义的实例变量：编译运行都看左边

- Object类

> 1️⃣ Object类是所有Java类的根父类
>
> 2️⃣ 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
>
> 3️⃣ 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()
>
> 4️⃣ Object类只声明了一个空参的构造器

- 面试题： == 和equals() 区别

> 一、回顾 == 的使用：
>
> == ： 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。（不一定要类型相同）
>
> 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，我么就需要对　　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

- 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"

- 单元测试方法的使用

```
import java.util.Date;
import org.junit.Test;

/*
 * Java中的JUßnit单元测试
 *
 * 步骤：
 *  1. 首先生成JUnit工具包
 *     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：
 *     1️⃣ 此类是public的
 *     2️⃣ 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，作践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
 */

public class JUnitTest {

  @Test
  public void testEquals() {
```

```
    String s1 = "哈哈哈";
    String s2 = "呵呵呵";
    System.out.println(s1.equals(s2));

    Object obj = new String("123");
    Date date = (Date)obj;
  }
}
```

- 包装类的使用

```
import org.junit.Test;
/*
 * 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
 *
 *  基本数据类型     包装类
 *  byte         Byte
 *  short      Short
 *  int        Integer
 *  long       Long
 *  float      Float
 *  double       Double
 *
 *  以上几个包装类的父类为：Number
 *
 *  boolean      Boolean
 *  char       Character
 *
 * 2. 掌握：基本数据类型、包装类、String三者之间的相互转换
 */
public class WrapperTest {

  // 基本数据类型 -> 包装类：调用包装类的构造器
  @Test
  public void test1() {
    int num = 10;
    Integer in1 = new Integer(num);
    System.out.println(in1.toString());

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

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

```
    Double d1 = new Double(12.3);
    Double d2 = new Double("12.3");
    System.out.println(d1);
    System.out.println(d2);

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

    System.out.println(b1);
    System.out.println(b2);

    // 特殊，不报异常
    Boolean b3 = new Boolean("true123");
    System.out.println(b3);// false

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

// 包装类 -> 基本数据类型:调用包装类的xxxValue()
@Test
public void test2() {
    Integer i1 = new Integer(111);

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

    Float f1 = new Float(12.1);
    float f2 = f1.floatValue();
    System.out.println(f2);
}

// JDK5.0新特性：自动装箱与自动拆箱
@Test
public void test3() {

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

    // 自动拆箱：包装类 -> 基本数据类型
    int num3 = in1; // 自动拆箱

}

// 基本数据类型、包装类 -> String类型
@Test
public void test4() {
```

```
    int num1 = 10;
    // 方式1:连接运算
    String str1 = num1 + "";
    System.out.println(str1);

    // 方式2:调用String的valueOf()
    float f1 = 12.3f;
    String str2 = String.valueOf(f1);
    System.out.println(str2);

    Double d1 = new Double(12.6);
    String str3 = String.valueOf(d1);
    System.out.println(str3); // 12.6
  }

  // String类型  -> 基本数据类型、包装类:调用包装类的parseXXX()
  @Test
  public void test5() {
    String str1 = "123";

    // 错误的情况: int 、Integer与String 不存在子父类的关系，不能强制转换
//    int num1 = (int) str1;
//    Integer in1 = (Integer)str1;

    // 如果str1 = "123a" 则会报错NumberFormatExcption
    int num2 = Integer.parseInt(str1);
    System.out.println(num2 + 1);

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

}

class Order {
  boolean isMale;
  Boolean isFemale;
}
```

- 补充知识点

> Integer内部定义了IntegerCache结构，IntegerCache中定义了Integer[]，保存了-128～127范围的整数，如果我们使用自动装箱的方式，给Integer赋值的范围在-128～127范围内时，可以直接使用数组中的元素，不用再去new了。目的：提高效率

面向对象-下

static关键字

```
/*
*1. static：静态的
* 2. static可以用来修饰：属性、方法、代码块、内部类
*
* 3. 使用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：
*       1️⃣ 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
*       2️⃣ 静态变量的加载要早于对象的创建
*       3️⃣ 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。
*       4️⃣          静态变量      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
*    4.1 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
*    4.2        静态方法     非静态方法
*     类        yes       no
*        对象        yes          yes
*    4.3 静态方法中：能调用静态的方法或属性
*       非静态方法：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，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。
*
*    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
*/
```

单例设计模式

- 定义

  > 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。

- 饿汉式

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

    Bank b1 = Bank.getInstance();
    Bank b2 = Bank.getInstance();

    System.out.println(b1 == b2);

  }
}

// 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;
  }
}
```

- 懒汉式

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

    Order o1 = Order.getInstance();
    Order o2 = Order.getInstance();

    System.out.println(o1 == o2);

  }
```

```
}

// 懒汉式
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.要求此对象也必须声明为静态的
  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;
  }

}
```

- 区分饿汉式 和 懒汉式

> 饿汉式： 面试写这个
>
> > 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：延时对象的创建。
> >
> > 目前的写法坏处：线程不安全。

- 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）

代码块

```
/*
 * 类的成员四：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，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
 *
 * 对属性可以赋值的位置：
 *     1.默认初始化
 *     2.显示初始化
 *     3.构造器中初始化
 *     4.有个对象以后，可以通过"对象.属性"或"对象.方法"的方式，进行赋值
 *     5.在代码块中赋值
 *
 *     执行顺序：1 -> 2 / 5 （谁在上谁先执行） -> 3 -> 4
 */
public class BlockTest {
  public static void main(String[] args) {
    Integer age = Person.age;

    Person p1 = new Person();
  }
}

class Person {

  // 属性
  String name = "一个人";
  static int age;
```

```
  // 构造器
  public Person() {
    super();
  }

  public Person(String name) {
    super();
    this.name = name;
  }

  // 代码块
  static {
    System.out.println("hello static block");
  }

  {
    System.out.println("hello block");
  }


  // 方法
  public void eat() {
    System.out.println("人吃饭");
  }

  @Override
  public String toString() {
    return "Person [name=" + name + "]";
  }

}
```

- 总结

  > 执行顺序：由父及子，静态先行

final关键字

```
/*
* 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修饰一个属性：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
*
*/
public class FinalTest {
    public static void main(String[] args) {
      Man man = new Man();
      man.show(100);
    }
}

class Man {
    public void show(final int num) {
    //    num = 200; 报错  不能重新进行赋值
      System.out.println(num);
    }
}
```

abstract关键字

```
/*
* 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
*     > 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
*
* 抽象类应用
*     > 抽象类是用来模型化那些父类无法确定全部实现，而是由其子类提供具体实现的对象的类。
*
* 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
*
*/
public class AbstractTest {
    public static void main(String[] args) {
      //    Person p1 = new Person();
      //    p1.eat();

      //    Walk w1 = new Walk("张安", 18);
      //    w1.eat();
    }
}

abstract class Person{

    String name;
    int age;

    public Person() {

    }

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

    public void eat() {
      System.out.println(this.name);
      System.out.println(this.age);
    }

    // 不是抽象方法
    //  public void walk() {
    //
    //  }

    // 抽象方法
    public abstract void walk();

}

// abstract修饰
abstract class Walk extends Person{
    public Walk() {

    }
```

```
    public Walk(String name, int age) {
      super(name, age);
    }

    // 重写父类中的方法
    //  public void walk() {
    //
    //  }
}
```

- 抽象类的应用：模板方法设计模式

> 抽象类体现的就是一种模板模式的设计，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展、改造，但子类总体上会保留抽象类的行为方式。
>
> 解决的问题：
>
> > 1．当功能内部一部分实现是确定的，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。

> 2．换句话说，在软件开发实现一个算法时，整体步骤很固定、通用，这些步骤已经在父类中写好了。但是某些部分易变，易变部分可以抽象出来，供不同子类实现。这就是一种模板模式。

```
//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!!");
    }
  }
```

接口

```
/*
* 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，但是书写时，可以省略不写
*
*    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
*
* 4.接口中不能定义构造器。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
```

```java
*/
public class InterfaceTest {
    public static void main(String[] args) {
      Plane p = new Plane();
      p.fly();
      p.stop();
    }
}

// 接口
interface Flyable{

    // 全局常量
    public static final int MAX_SPEED = 7900;// 第一宇宙速度
    int MIN_SPEED = 1;// 省略了public static final

    // 抽象方法
    public abstract void fly();

    void stop();// 省略了public abstract
    }

interface Attackable{

}

// 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
class Plane implements Flyable{

    @Override
    public void fly() {
      System.out.println("飞机起飞");
    }

    @Override
    public void stop() {
      System.out.println("飞机停止");
    }

}

//  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
abstract class Kite implements Flyable{

    @Override
    public void fly() {
      // TODO Auto-generated method stub

    }
```

```
}

class Bullet extends Object implements  Attackable,Flyable,AA,BB {

    @Override
    public void fly() {

    }

    @Override
    public void stop() {

    }

    @Override
    public void method2() {

    }

    @Override
    public void method1() {

    }

}

/*********************************************************/
// 接口与接口之间可以继承，而且可以多继承
interface AA{
    public abstract void method1();
}


interface BB{
    public abstract void method2();
}

interface CC extends AA,BB{

}
```

- 接口的使用

```
package com.java.test1;
/*
 * 接口的使用
```

```java
 *
 * 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
 *
 */

public class USBTEst {

  public static void main(String[] args) {
    Computer com = new Computer();
    Flash flash = new Flash();
    com.transferData(flash);
  }

}

class Computer{

  public void transferData(USB usb) {// 接口使用上也满足多态性
    usb.start();

    System.out.println("具体传输的细节");

    usb.end();
  }

}

interface USB{

  public abstract void start();

  public abstract void end();

}

class Flash implements USB{

  @Override
  public void start() {
    System.out.println("U盘开始工作");
  }

  @Override
  public void end() {
    System.out.println("U盘停止工作");
  }
}
```

- 创建接口匿名实现类的对象

```
package com.java.test1;
/*
 * 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
 *
 */

public class USBTEst {

  public static void main(String[] args) {
    Computer com = new Computer();
    // 1.创建了接口的非匿名实现类的非匿名对象
    Flash flash = new Flash();
    com.transferData(flash);

    // 2.创建了接口的非匿名实现类的匿名对象
    com.transferData(new Printer());

    // 3.创建了接口的匿名实现类的非匿名对象
    USB phone = new USB() {

      @Override
      public void start() {
        System.out.println("手机开始工作");
      }

      @Override
      public void end() {
        System.out.println("手机结束工作");
      }

    };
    com.transferData(phone);

    // 4.创建了接口的匿名实现类的匿名对象
    com.transferData(new USB() {

      @Override
      public void start() {
        System.out.println("Mp3开始工作");
      }

      @Override
      public void end() {
```

```
        System.out.println("Mp3开始工作");
      }

    });


  }

}

class Computer{

  public void transferData(USB usb) {// 接口使用上也满足多态性
    usb.start();

    System.out.println("具体传输的细节");

    usb.end();
  }

}

interface USB{

  public abstract void start();

  public abstract void end();

}

class Flash implements USB{

  @Override
  public void start() {
    System.out.println("U盘开始工作");
  }

  @Override
  public void end() {
    System.out.println("U盘停止工作");
  }

}


class Printer implements USB{

  @Override
  public void start() {
```

```
    System.out.println("打印机开始工作");
  }

  @Override
  public void end() {
    System.out.println("打印机结束工作");
  }

}
```

代理模式

> 代理模式是Java开发中使用较多的一种设计模式。代理设计就是为其他对象提供一种代理以控制对这个对象的访问。

- 应用场景

  1. 安全代理：屏蔽对真实角色的直接访问。
  2. 远程代理：通过代理类处理远程方法调用（RMI）
  3. 延时加载：先加载轻量级的代理对象，真正需要再加载真实对象。

- 分类
  - 静态代理（静态定义代理类）

```
/*
 * 接口的应用：代理模式
 *
 *
 */
public class ProxyTest {
  public static void main(String[] args) {
    Server server = new Server();
    ProxyServer proxy = new ProxyServer(server);

    proxy.browse();
  }
}

interface Network{

  public abstract void browse();

}

// 被代理类
class Server implements Network{

  @Override
  public void browse() {
```

```
    System.out.println("真实网络地址");
  }

}

// 代理类
class ProxyServer implements Network{

  private Network network;

  public ProxyServer(Network network) {
    this.network = network;
  }

  private void check() {
    System.out.println("联网加载中...");
  }

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

    network.browse();
  }
}
```

- 动态代理（动态生成代理类）

  > JDK自带的动态代理，需要反射等知识

Java8中接口的新特性

- CompareA.java

```
package com.java.test8;
/*
 *
 * JDK8:除了定义全局变量和抽象方法之外，还可以定义静态方法、默认方法
 *
 */
public interface CompareA {

  // 静态方法
  public static void mehthod1() {
    System.out.println("Compare:北京");
  }

  // 默认方法
  public default void method2() {
```

```
    System.out.println("Compare:上海");
  }

  // 可以省略public，但是权限还是public
  default void method3() {
    System.out.println("Compare:上海");
  }

}
```

- CompareB.java

```
package com.java.test8;

public interface CompareB {

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

- SuperClass.java

```
package com.java.test8;

public class SuperClass {
  public void method3() {
    System.out.println("SuperClass:天津");
  }
}
```

- SubClassTest.java

```
package com.java.test8;

public class SubClassTest {

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

//    s.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.mehthod1();

    // 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();

    s.myMethod();
  }

}

class SubClass extends SuperClass implements CompareA,CompareB{

  // 默认方法
  public void method2() {
    System.out.println("StbClass:上海");
  }

//  public void method3() {
//    System.out.println("StbClass:杭州");
//  }

  // 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();
  }
}
```

内部类

```
package com.java.test2;
/*
 * 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，可以被继承
 *           > 可以被abstract修饰，表示此类不能被实例化
 *
 *   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
 *
 */
public class InnerClassTest {
  public static void main(String[] args) {
    // 4.1 如何实例化成员内部类的对象
    // 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 p = new Person();
    Person.Bird bird = p.new Bird();
    bird.sing();

  }
}

class Person{

  String name;
  int age;

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

  // 静态成员内部类
  static class Dog{
    public void show() {
      System.out.println("展示");
    }
  }

  //非静态成员内部类
  class Bird{
    String name;
```

```
    public Bird() {

    }

    public void sing() {
      System.out.println("小鸟");
      Person.this.eat();// 调用外部类的非静态方法
    }
  }

  /*********************************************************/

  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{

    }
  }
}
```

- 注意点

> 在局部内部类的方法中，如果调用局部内部类所声明的方法中的局部变量时，要求此局部变量声明为final的。
>
> JDK 7及之前的版本：要求此局部变量显式的声明为final的
>
> JDK 8及以后的版本：可以省略final的声明

```
package com.java.test;

public class InnerClassTest {

  public void method() {
```

```
    // 局部变量
    int num = 10; // 实际上为 final int num

    class AA {

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

异常处理

常见异常

```
package com.java.test4;

import java.io.File;
import java.io.FileInputStream;
import java.util.Date;
import java.util.Scanner;

import org.junit.Test;

/*
 * 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）
 *           |------NullPointerException
 *           |------ArrayIndexOutOfBoundsException
 *           |------ClassCastException
 *           |------NumberFormatException
 *           |------InputMismatchException
 *           |------ArithmeticException
 *
 *
 */
```

```java
public class ExceptionTest {

  /*********************************编译时异常
**********************************************/
  @Test
  public void test7() {
    File file = new File("hello.txt");
    FileInputStream fis = new FileInputStream(file); // FileNotFoundException

    int data = fis.read(); // IOException
    while(data != -1) {
      System.out.println((char)data);
      data = fis.read();// IOException
    }
    fis.close();// IOException
  }

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

  // InputMismatchException 输入信息不匹配
  @Test
  public void test5() {

    // 输入数字正常，输入其他信息就会出现InputMismatchException异常
    Scanner scanner = new Scanner(System.in);
    int score = scanner.nextInt();
    System.out.println(score);
  }


  // NumberFormatException 转换为数值类型异常
  @Test
  public void test4() {

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

  // ClassCastException 类型转换异常
  @Test
```

```
  public void test3() {

    Object obj = new Date();
    String str = (String)obj;

  }


  // 角标越界
  @Test
  public void test2() {
    // ArrayIndexOutOfBoundsException 数组角标越界
    int[] arr = new int[5];
    System.out.println(arr[5]);

    // StringIndexOutOfBoundsException 字符串角标越界
    String str = "abc";
    System.out.println(str.charAt(6));
  }


  // NullPointerException 空指针异常
  @Test
  public void test1() {

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

    String str = "abc";
    str = null;
    System.out.println(str.charAt(1));

  }
}
```

异常的处理：抓抛模型

> 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

- try-catch-finally的使用

```
package com.java.test4;
```

```java
import org.junit.Test;

/*
 *
 * 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，是使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。
 */
public class ExceptionTest1 {

  @Test
  public void test1() {
```

```
    String str = "123";
    str = "abc";

    try {
      int num = Integer.parseInt(str);

      System.out.println("hello------1");
    } catch (NumberFormatException e) {
//      System.out.println("出现数值转换异常");

//      System.out.println(e.getMessage());

      e.printStackTrace();

    } catch(NullPointerException e) {
      System.out.println("出现空指针异常");
    } catch(Exception e) {
      System.out.println("报错了...");
    }

    System.out.println("hello------2");
  }

}
```

```
package com.java.test4;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

import org.junit.Test;

/*
 * 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。
 *
 */
public class FinallyTest {
```

```java
  @Test
  public void test2() {
    FileInputStream fis = null;
    try {
      File file = new File("hello.txt");
      fis = new FileInputStream(file); // FileNotFoundException

      int data = fis.read(); // IOException
      while(data != -1) {
        System.out.println((char)data);
        data = fis.read();// IOException
      }

    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      try {
        if (fis != null)
            fis.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

  }

  @Test
  public void testMethod() {
    int num = method();
    System.out.println(num);
  }

  public int method() {
    try {
      int[] arr = new int[10];
      System.out.println(arr[10]);
      return 1;
    } catch (Exception e) {
//      e.printStackTrace();
      return 2;
    } finally {
      return 3;// 最先返回
    }

  }

  @Test
  public void test1() {
```

```
    try {

      int num1 = 10;
      int num2 = 0;
      System.out.println(num1 / num2);

    } catch (ArithmeticException e) {
//      e.printStackTrace();

      int[] arr = new int[10];
      System.out.println(arr[10]);

    } finally {
      System.out.println("啦啦啦啦");
    }

  }
}
```

- throws + 异常类型

```
package com.java.test4;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

/*
 * 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
 *       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。
 *
 *
```

```
 */
public class ExceptionTest2 {

  public static void main(String[] args) {

//    try {
//      method2();
//    } catch (IOException e) {
//      e.printStackTrace();
//    }

    method3();

  }

  public static void method3() {
    try {
      method2();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public static void method2() throws IOException{
    method1();
  }


  public static void method1() throws FileNotFoundException,IOException{
    File file = new File("hello1.txt");
    FileInputStream fis = new FileInputStream(file); //
FileNotFoundException

    int data = fis.read(); // IOException
    while(data != -1) {
      System.out.println((char)data);
      data = fis.read();// IOException
    }
    fis.close();// IOException
  }

}
```

- 自定义异常类

```
package com.java.test5;
/*
 * 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
 *
 */
public class MyException extends RuntimeException {

  static final long serialVersionUID = -7034897190745766939L;

  public MyException() {

  }

  public MyException(String msg) {
      super(msg);
  }
}
```

Java高级

多线程

多线程的创建

- 方式一：继承于Thread类

```
package com.java.test;

/**
 * 多线程的创建，方式一：继承于Thread类
 * 1. 创建一个继承于Thread类的子类
 * 2. 重写Thread类的run()方法 --> 将此线程执行的操作声明在run()中
 * 3. 创建Thread类的子类的对象
 * 4. 通过此对象调用start()方法。作用：1.启动当前线程 2.调用当前线程的run()
 *
 * 例子：遍历100以内的所有的偶数
 *
 *
 *
 */

// 1. 创建一个继承于Thread类的子类
class MyThread extends Thread{
   // 2. 重写Thread类的run()方法
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
```

```
                System.out.println(Thread.currentThread().getName() + ": "
+ i);// 查看归属于哪个线程
            }
        }
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        //3. 创建Thread类的子类的对象
        MyThread t1 = new MyThread();
        // 4. 通过此对象调用start()方法
        t1.start();// 分线程

        // 问题一：我们不能通过直接调用run()的方式启动线程。
        // t1.run()

        // 问题二：再启动一个线程，遍历100以内的偶数。不可以还让已经start()的线程去执
行。返回异常：IllegalThreadStateException
//        t1.start();

        // 如下操作仍然在main(主线程)线程中执行
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
                System.out.println(Thread.currentThread().getName() + ": "
+ i + "*****************************");
            }
        }

        // 匿名子类的方式
        new Thread(){
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    if (i % 2 == 0) {
                        System.out.println(Thread.currentThread().getName()
+ ": " + i);// 查看归属于哪个线程
                    }
                }
            }
        }.start();

    }

}
```

- 方式2：实现Runnable接口

```java
package com.java.test;

/**
 * 创建多线程的方式二：实现Runnable接口
 * 1. 创建一个实现了Runnable接口的类。
 * 2. 实现类去实现Runnable的抽象方法：run()
 * 3. 创建实现类的对象
 * 4. 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象
 * 5. 通过Thread类的对象调用start()
 *
 *
 */

// 1. 创建一个实现了Runnable接口的类。
class MThread implements Runnable{
    // 2. 实现类去实现Runnable的抽象方法：run()
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
                System.out.println(Thread.currentThread().getName() + ":" +
i);
            }
        }
    }
}


public class ThreadTest1 {
    public static void main(String[] args) {
        // 3. 创建实现类的对象
        MThread mThread = new MThread();
        // 4. 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象
        Thread t1 = new Thread(mThread);
        // 5. 通过Thread类的对象调用start():1.启动线程 2 调用当前线程的run() -->
调用了Runnable类型的target的run()
        t1.setName("线程1");
        t1.start();

        // 在启动一个线程
        Thread t2 = new Thread(mThread);
        t2.setName("线程2");
        t2.start();
    }
}
```

- 比较创建线程的两种方式

开发中：优先选择：实现Runnable接口的方式
 原因：1、实现的方式类的单继承性的局限性
        2、实现的方式更适合来处理多个线程有共享数据的情况。

  联系：class Thread implements Runnable
  相同点：两种方式都需要重写run(),将线程要执行的逻辑声明在run()中。

- 线程的优先级

```
package com.java.exer;

/**
 * 线程的优先级
 * 1.
 * MAX_PRIORITY: 10 --> 最大优先级
 * MIN_PRIORITY: 1 --> 最小优先级
 * NORM_PRIORITY: 5 --> 默认优先级
 * 2. 如何获取和设置当前线程的优先级:
 *    getPriority():获取线程的优先级
 *    setPriority():设置线程的优先级
 *
 * 说明:  高优先级的线程要抢占低优先级线程CPU的执行权。但是只是从概率上讲，高优先级的线程高概率的情况下被执行。
 *        并不意味着只有当高优先级的线程执行完以后，低优先级的线程才执行。
 */

class myThread extends Thread{

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
                System.out.println(Thread.currentThread().getName() + ":" + Thread.currentThread().getPriority()  + ":" + i);
            }
        }
    }
}

public class ThreadMethodTest {
    public static void main(String[] args) {
        myThread m1 = new myThread();
        m1.setPriority(Thread.MAX_PRIORITY); // 设置分线程高优先级
        m1.start();

        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);// 设置主线程低优先级
        for (int i = 0; i < 100; i++) {
```

```
            if (i % 2 == 0) {
                System.out.println(Thread.currentThread().getName() + ":" +
Thread.currentThread().getPriority()  + ":" + i);
            }
        }
    }
}
```

线程的常用方法

```
package com.test.java;

/**
 * 测试Thread中的常用方法：
 * 1. start():启动当前线程：调用当前线程的run()
 * 2. run():通常需要重写Thread类中的此方法，将创建的线程要执行的操作声明在此方法中
 * 3. currentThread():静态方法，返回执行当前代码的线程
 * 4. getName():获取当前线程的名字
 * 5. setName():设置当前线程的名字
 * 6. yield():释放当前cpu的执行权
 * 7. join():在线程a中调用线程b的join()，此时线程a就进入阻塞状态，直到线程b完全执行完以
后，线程a
 *           才结束阻塞状态
 * 8. stop():已过时。当执行此方法时，强制结束当前进程。
 * 9. sleep(long millitime):让当前线程"睡眠"指定的millitime毫秒。在指定的millitime毫
秒时间内，
 *                            当前线程是阻塞状态。
 * 10. isAlive():判断当前线程是否存活。
 */

class MyThread extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {

                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println(Thread.currentThread().getName() + ":" + i);
            }

//            if (i % 20 == 0) {
//                yield();
```

```
//            }
        }
    }

    public MyThread(String name){
        super(name);
    }
}


public class ThreadMethodTest {
    public static void main(String[] args) {
        MyThread m1 = new MyThread("Thread_1");// 方式2
//        m1.setName("线程1");// 方式1
        m1.start();

        // 主线程命名
        Thread.currentThread().setName("主线程");

        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
                System.out.println(Thread.currentThread().getName() + ":" + i);
            }

            if (i == 20) {
                try {
                   m1.join();
                } catch (Exception e) {
                   e.printStackTrace();
                }

            }
        }
    }
}
```

线程安全问题的举例和解决措施

- 方式一：同步代码块
  - 解决使用Runnable接口的方式

```
package com.test.java;

/**
 * 多窗口售票：创建三个窗口售票，总票数为100张：使用Runnable接口的方式
 *
 * 问题：
 * 1.卖票过程中，出现重票、错票的问题 --> 出现线程安全问题
```

```
 * 2.问题出现的原因：当某个线程操作车票的过程中，尚未操作完成时，其他线程参与进来，也操作车票
 * 3.如何解决：当一个线程A在操作ticket的时候，其他的线程不能参与进来，直到线程A操作完ticket后，
 *            其他线程才可以开始操作ticket。这种情况即使线程A出现阻塞，也不能被改变。
 * 4.在Java中，我们通过同步机制，来解决线程的安全问题
 *
 * 方式一：同步代码块
 *
 *  synchronized(同步监视器){
 *      // 需要被同步的代码
 *  }
 *
 *  说明：1.操作共享数据的代码，即为需要被同步的代码。--> 只需要包需要被同步的代码，不能包含代码多了，也不能包含代码少了
 *       2.共享数据：多个线程共同操作的变量。比如：ticket就是共享数据。
 *       3.同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。
 *          要求：多个线程必须要共用同一把锁。
 *
 *       补充：在实现Runnable接口创建多线程的方式中，我们可以考虑使用this充当同步监视器。
 *
 * 方式二：同步方法
 *      如果操作共享数据的代码完整的声明在一个方法中，我们不妨将此方法声明为同步的。
 *      1.同步方法仍然涉及到同步监视器，只是不需要我们显式的声明。
 *      2.非静态的同步方法，同步监视器是this
 *        静态的同步方法，同步监视器是：当前类本身
 *
 * 5、同步的方式，解决了线程的安全问题。 -- 好处
 *    操作同步代码时，只能有一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低 --> 局限性
 */

class Window1 implements Runnable{
    private int ticket = 100;
//    Object obj = new Object();

    @Override
    public void run() {
        while(true){

//            synchronized(obj) { // 方式二
            synchronized(this) { // 此时的this：唯一的window1的对象
                if (ticket > 0) {

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
```

```
                    }

                    System.out.println(Thread.currentThread().getName() +
"售票：当前票号为：" + ticket);
                    ticket--;
                } else {
                    break;
                }
            }

        }
    }
}

public class WindowTicket1 {
    public static void main(String[] args) {
        Window1 w = new Window1();

        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

- 使用同步代码块解决继承Thread类的方式的线程安全问题

```
package com.test.java;

/**
 * 使用同步代码块解决继承Thread类的方式的线程安全问题
 *
 * 例子：多窗口售票：创建三个窗口售票，总票数为100张：使用继承Thread类的方式
 *
 * 说明：在继承Thread类创建多线程的方式中，慎用this充当同步监视器，考虑使用当前类充当同
步监视器（类名.class）
 *
 */
class Window2 extends Thread{
    private static int ticket = 100;
    private static Object obj = new Object();
```

```
    @Override
    public void run() {

        while(true){
//            synchronized (obj) { // 正确的方式
            synchronized(Window2.class){ // Window2.class只会加载一次
//            synchronized (this) { 错误的方式，this代表着w1,w2,w3三个对象

                if (ticket > 0) {

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    System.out.println(getName() + "售票：当前票号为：" +
ticket);
                    ticket--;
                } else {
                    break;
                }
            }
        }

    }
}

public class WindowTicket2 {
    public static void main(String[] args) {
        Window2 w1 = new Window2();
        Window2 w2 = new Window2();
        Window2 w3 = new Window2();

        w1.setName("窗口1");
        w2.setName("窗口2");
        w3.setName("窗口3");

        w1.start();
        w2.start();
        w3.start();
    }
}
```

- 方式二：同步方法
  - 使用同步方法解决实现Runnable接口的线程安全问题

```java
package com.test.java;

/**
 * 使用同步方法解决实现Runnable接口的线程安全问题
 *
 *
 */
class Window3 implements Runnable{
    private int ticket = 100;
//    Object obj = new Object();

    @Override
    public void run() {
        while(true){
            show();
        }
    }

    private synchronized void show(){ // 同步监视器：this
        if (ticket > 0) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(Thread.currentThread().getName() + "售票：当前票号为：" + ticket);
            ticket--;

        }
    }
}

public class WindowTicket3 {
    public static void main(String[] args) {
        Window3 w = new Window3();

        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
        t2.start();
```

```
        t3.start();
    }
}
```

- 使用同步方法解决继承Thread类的方式的线程安全问题

```
package com.test.java;

/**
 * 使用同步方法处理继承Thread类的方式的线程安全问题
 *
 * @author: my.seaTide
 * @create: 2021/3/28 1:39 下午
 */
class Window2 extends Thread {
    private static int ticket = 100;

    @Override
    public void run() {

        while (true) {
            show();
        }
    }

    private static synchronized void show() {// 同步监视器：Window2.class
        if (ticket > 0) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(Thread.currentThread().getName() + "售票：当前票号为：" + ticket);
            ticket--;
        }
    }

}

public class WindowTest4 {
    public static void main(String[] args) {
        Window2 w1 = new Window2();
        Window2 w2 = new Window2();
        Window2 w3 = new Window2();

        w1.setName("窗口1");
```

```
        w2.setName("窗口2");
        w3.setName("窗口3");

        w1.start();
        w2.start();
        w3.start();
    }
}
```

- 使用同步机制将单例模式中的懒汉式改写为线程安全的

```
package com.test.java;

/**
 * @author: my.seaTide
 * @create: 2021/3/28 9:20 下午
 */
public class BankTest {

}

class Bank{

    private Bank(){

    }

    private static Bank instance = null;

    public static Bank getInstance() {

        // 方式一：效率稍差
//        synchronized (Bank.class) {
//            if (instance == null) {
//                instance = new Bank();
//            }
//            return  instance;
//        }

        // 方式二：效率更高
        if(instance == null) {
            synchronized (Bank.class) {
                if (instance == null) {
                    instance = new Bank();
                }
            }
        }
        return  instance;
```

```
    }
}
```

- 方式三：lock锁

```
package com.test.java;

import java.util.concurrent.locks.ReentrantLock;

/**
 * 解决线程安全问题的方式三：Lock锁 --> JDK5.0新增
 *
 * 1、面试题
 * synchronized 与 lock的异同?
 * 相同点：二者都可以解决线程安全问题
 * 不同点：
 *    1）、synchronized机制在执行完响应的同步代码以后，自动的释放同步监视器
 *      lock需要手动的启动同步（lock()），同时结束同步也需要手动的实现（unlock()）
 *
 *    2）、lock只有代码块锁，synchronized有代码块锁和方法锁。
 *    3）、使用lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）
 *
 * 2、优先使用顺序：
 *  lock锁 --> 同步代码块（已经进入了方法体，分配了相应资源） --> 同步方法（在方法体之外）
 *
 * 3、面试题：如何解决线程安全问题？有几种方式
 *
 *
 */

class Window implements Runnable{
    private int ticket = 100;
    // 1.实例化ReentrantLock
    private ReentrantLock lock = new ReentrantLock();
    @Override
    public void run() {
       while(true) {

            try{
                // 2.调用锁定方法：lock()
                lock.lock();

                if (ticket > 0) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }

                System.out.println(Thread.currentThread().getName() +
": 票号: " + ticket);
                ticket--;
            } else {
                break;
            }

        } finally {
            //调用解锁方法: unlock()
            lock.unlock();
        }

    }
}
}

public class LockTest {
    public static void main(String[] args) {
        Window w = new Window();

        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

线程的死锁

```
package com.test.java;

/**
 * 演示线程的死锁问题
 *
 * 1、死锁的理解: 不同的线程分别占用对方需要的同步资源不放弃, 都在等待对方放弃自己需要的同步资源, 就形成了线程的死锁
 *
 * 2、说明:
```

```
 *  1）出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续。
 *  2）我们使用同步时，要避免出现死锁。
 *
 * 解决方法：
 * 1、专门的算法、原则
 * 2、尽量减少同步资源的定义
 * 3、尽量避免嵌套同步
 *
 *
 *
 */
public class ThreadTest {
    public static void main(String[] args) {

        StringBuffer s1 = new StringBuffer();
        StringBuffer s2 = new StringBuffer();

        // 继承Thread方式
        new Thread(){
            @Override
            public void run() {
                synchronized(s1){

                    s1.append("a");
                    s2.append("1");

                    // 造成死锁
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    synchronized (s2) {
                        s1.append("b");
                        s2.append("2");

                        System.out.println(s1);
                        System.out.println(s2);
                    }

                }
            }

        }.start();

        // 实现Runnable方式
        new Thread(new Runnable() {
            @Override
```

```
            public void run() {
                synchronized(s2){

                    s1.append("c");
                    s2.append("3");

                    // 造成死锁
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    synchronized (s1) {
                        s1.append("d");
                        s2.append("4");

                        System.out.println(s1);
                        System.out.println(s2);
                    }

                }
            }
        }).start();

    }
}
```

练习

- 方法一

```
package com.test.exer;
```

/**

- 银行有一个账户
- 有两个储户分别向一个账户存3000元，每次存1000，存三次。每次存完打印账户余额
  *
- 分析：
- 1、是否多线程问题？是 两个储户线程
- 2、是否有共享数据？有 一个账户（或账户余额）
- 3、是否有线程安全问题？有

- 4、需要考虑如何解决线程安全问题？同步机制：有三种方式。
  *
  */
  class Account{
  private double balance;
  public Account(double balance){
  this.balance = balance;
  }
  // 存钱
  public synchronized void deposit(double money) {
  if (money > 0) {
  balance += money;

```
try {
    Thread.sleep(100);
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println(Thread.currentThread().getName() + "存钱成功：当前余额为：" + balance);
}
```

  }
  }

class Customer extends Thread{

```
private Account acct;

public Customer(Account account){
    this.acct = account;
}

@Override
public void run() {
    for (int i = 0; i < 3; i++) {
        acct.deposit(1000);
    }
}
```

}

public class AccountTest {
  public static void main(String[] args) {
  Account account = new Account(0);
  Customer c1 = new Customer(account);
  Customer c2 = new Customer(account);

```
        c1.setName("甲");
        c2.setName("乙");

        c1.start();
        c2.start();
    }
```

}

- 方法二

•~~~java
```
package com.test.exer;

import java.util.concurrent.locks.ReentrantLock;

/**
 * 银行有一个账户
 * 有两个储户分别向一个账户存3000元，每次存1000，存三次。每次存完打印账户余额
 *
 * 分析：
 * 1、是否多线程问题？是 两个储户线程
 * 2、是否有共享数据？有 一个账户（或账户余额）
 * 3、是否有线程安全问题？有
 * 4、需要考虑如何解决线程安全问题？同步机制：有三种方式。
 *
 */
class Account2{
    private double balance;

    private ReentrantLock lock = new ReentrantLock();

    public Account2(double balance){
        this.balance = balance;
    }
    // 存钱
    public void deposit(double money) {
        lock.lock();
        if (money > 0) {
            balance += money;

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            System.out.println(Thread.currentThread().getName() + "存钱成功：当前
余额为：" + balance);
        }
        lock.unlock();
    }
}

class Customer2 implements Runnable{

    private Account2 acct;

    public Customer2(Account2 account){
        this.acct = account;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            acct.deposit(1000);
        }
    }
}

public class AccountTest2 {
    public static void main(String[] args) {
        Account2 acct = new Account2(0);
        Customer2 c = new Customer2(acct);
        Thread t1 = new Thread(c);
        Thread t2 = new Thread(c);

        t1.setName("甲");
        t2.setName("乙");

        t1.start();
        t2.start();
    }
}
```

线程通信

```
package com.test.java2;

/**
 * 线程通信的例子
 * 使用两个线程打印1-100。线程1、线程2交替打印
 *
 * 涉及到的三个方法：
 * wait()：一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器。
```

```
 * notify()：一旦执行此方法，就会唤醒被wait()的一个线程，如果有多个线程被wait()，就唤醒优先级高的那个。
 * notifyAll()：一旦执行此方法，就会唤醒所有被wait()的线程。
 *
 * 说明：
 * 1、wait()、notify()、notifyAll()三个方法必须使用在同步代码块或同步方法中
 * 2、wait()、notify()、notifyAll()三个方法的调用者必须是同步代码块或同步方法中的同步监视器。
 *   否则，会出现IllegalMonitorStateException异常
 * 3、wait()、notify()、notifyAll()三个方法是定义在java.lang.Object类中。
 *
 * 面试题：sleep()和 wait() 的异同?
 * 相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。
 * 不同点：
 *      1）、两个方法声明的位置不同：Thread类中声明sleep(),Object类中声明wait()。
 *      2)、调用的要求不同：sleep()可以在任何需要的场景下调用。wait()必须使用在同步代码块或同步方法中。
 *      3）、关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，wait()会释放锁，sleep()不会释放锁。
 *
 *
 */
class Number implements Runnable{
    private int number = 1;

    @Override
    public void run() {
        while(true) {

            synchronized (this) {

                notify();

                if (number <= 100) {

                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    System.out.println(Thread.currentThread().getName() + ": 当前数值为: " + number);
                    number++;

                    try {
                        // 使得调用如下wait()方法的线程 进入阻塞状态
                        wait();
                    } catch (InterruptedException e) {
```

```
                        e.printStackTrace();
                    }

                } else {
                    break;
                }
            }

        }
    }
}

public class CommunicationTest {
    public static void main(String[] args) {
        Number number = new Number();
        Thread t1 = new Thread(number);
        Thread t2 = new Thread(number);

        t1.setName("线程1");
        t2.setName("线程2");

        t1.start();
        t2.start();
    }
}
```

常用类

String的使用

```
package com.test.java;

import org.junit.Test;

/**
 * String的使用
 */
public class StringTest {
    /*
    String：字符串，使用一对 "" 引起来表示。
    1、String声明为final的，不可被继承
    2、String实现了Serializable接口：表示字符串是支持序列化的
               实现了Comparable接口：表示String可以比较大小
    3、String内部定义了final、char[] value用于存储字符串数据
    4、String：代表不可变的字符序列。简称：不可变性。
      体现：
```

```
        1）、当对字符串重新赋值时，需要重新指定内存区域赋值，不能使用原有的value进行赋值。
        2）、当对现有的字符串进行连接操作时，也需要重新指定内存区域赋值，不能使用原有的value进行赋值。
        3）、当调用String的replace()方法修改指定字符或字符串时，也需要重新指定内存区域赋值，不能使用原有的value进行赋值。
    5、通过字面量的方式（区别于new）给一个字符串赋值，此时的字符串值声明在字符串常量池中。
    6、字符串常量池中是不会存储相同内容的字符串的。
    */
    @Test
    public void test1(){
        String s1 = "aaa";// 字面量的定义方式
        String s2 = "aaa";
//        s1 = "ccc";

        System.out.println(s1 == s2);// 比较s1和s2的地址值
        System.out.println(s1);
        System.out.println(s2);


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

        String s3 = "aaa";
        s3 += "bbb";
        System.out.println(s3);
        System.out.println(s1);


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

        String s4 = "abc";
        String s5 = s4.replace('a', 'm');
        System.out.println(s4);
        System.out.println(s5);
    }

    /*
    String的实例化方式：
    方式一：通过字面量定义的方式
    方式二：通过new + 构造器的方式

    面试题：String s = new String("abc");这种方式创建对象，在内存中创建了几个对象？
          两个：一个是堆空间中的new结构，另一个是char[]对应的常量池中的数据："abc"
     */
    @Test
    public void test2(){
        // 通过字面量定义的方式：此时的s1和S2的数据javaEE声明在方法区中的字符串常量池中。
        String s1 = "javaEE";
```

```
        String s2 = "javaEE";

        // 通过new + 构造器的方式：此时的s3和s4保存的地址值，是数据在堆空间中开辟空间以后对应的地址值。
        String s3 = new String("javaEE");
        String s4 = new String("javaEE");

        System.out.println(s1 == s2); // true
        System.out.println(s1 == s3); // false
        System.out.println(s1 == s4); // false
        System.out.println(s3 == s4); // false
    }

    /*
    结论：
    1、常量与常量的拼接结果在常量池。且常量池中不会存在相同内容的常量。
    2、只要其中有一个是变量，结果就在堆中。
    3、如果拼接的结果调用intern()方法，返回值就在常量池中。
     */
    @Test
    public void test3(){
        String s1 = "Hello";
        String s2 = "World";

        String s3 = "HelloWorld";
        String s4 = "Hello" + "World";
        String s5 = s1 + "World";
        String s6 = s1 + s2;

        System.out.println(s3 == s4);// true
        System.out.println(s3 == s5);// false
        System.out.println(s3 == s6);// false

        String s7 = s6.intern();
        System.out.println(s3 == s7);// true
    }
}
```

String的常用方法

```
package com.test.java;

import org.junit.Test;

/**
 * String常用方法：
 */
public class StringMethodTest {
```

```
    /*
    替换：
    String replace(char oldChar, char newChar)：返回一个新的字符串，它是通过用newChar 替换此字符串中出现的所有 oldChar 得到的。
    String replace(CharSequence target, CharSequence replacement)：使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。
    String replaceAll(String regex,String replacement)：使用给定的replacement替换此字符串所有匹配给定的正则表达式的子字符串。
    String replaceFirst(String regex, String replacement)：使用给定的replacement替换此字符串匹配给定的正则表达式的第一个子字符串。
    匹配：
    boolean matches(String regex)：告知此字符串是否匹配给定的正则表达式。
    切片：
    String[] split(String regex)：根据给定正则表达式的匹配拆分此字符串。
    String[] split(String regex, int limit)：根据匹配给定的正则表达式来拆分此字符串，最多不超过limit个，如果超过了，剩下的全部都放到最后一个元素中。
    */
    @Test
    public void test4(){
        String str1 = "我爱祖国天安门";
        String str2 = str1.replace('我', '天');
        System.out.println(str1);
        System.out.println(str2);

        String str3 = str1.replace("我", "大家");
        System.out.println(str3);

        String str = "12hello34world5java7891mysql456";
        //把字符串中的数字替换成,，如果结果中开头和结尾有，的话去掉
        String string = str.replaceAll("\\d+", ",").replaceAll("^,|,$", "");
        System.out.println(string);

        String str4 = "12345";
        //判断str字符串中是否全部有数字组成，即有1-n个数字组成
        boolean matches = str4.matches("\\d+");
        System.out.println(matches);
        String tel = "0571-4534289";
        //判断这是否是一个杭州的固定电话
        boolean result = tel.matches("0571-\\d{7,8}");
        System.out.println(result);

        String str5 = "hello|world|java";
        String[] strs = str5.split("\\|");
        for (int i = 0; i < strs.length; i++) {
            System.out.println(strs[i]);
        }
        System.out.println();
        String str6 = "hello.world.java";
        String[] strs6 = str6.split("\\.");
```

```
        for (int i = 0; i < strs6.length; i++) {
            System.out.println(strs6[i]);
        }
    }

    /*
    boolean endsWith(String suffix)：测试此字符串是否以指定的后缀结束
    boolean startsWith(String prefix)：测试此字符串是否以指定的前缀开始
    boolean startsWith(String prefix, int toffset)：测试此字符串从指定索引开始的子字符串是否以指定前缀开始

    boolean contains(CharSequence s)：当且仅当此字符串包含指定的 char 值序列时，返回true
    int indexOf(String str)：返回指定子字符串在此字符串中第一次出现处的索引
    int indexOf(String str, int fromIndex)：返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
    int lastIndexOf(String str)：返回指定子字符串在此字符串中最右边出现处的索引
    int lastIndexOf(String str, int fromIndex)：返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索
    注：indexOf和lastIndexOf方法如果未找到都是返回-1
    */
    @Test
    public void test3(){
        String s1 = "helloworld";
        boolean s2 = s1.endsWith("ld");
        System.out.println(s2);// true

        boolean s3 = s1.startsWith("he");
        System.out.println(s3);// true

        boolean s4 = s1.startsWith("lo", 3);
        System.out.println(s4);// true

        boolean s5 = s1.contains("wor");
        System.out.println(s5);// true

        int s6 = s1.indexOf("wo");
        System.out.println(s6);// 5

        String s7 = "abcdefavc";
        int s8 = s7.indexOf("a", 5);
        System.out.println(s8);

        int s9 = s7.lastIndexOf("f");
        System.out.println(s9);

        int s10 = s7.lastIndexOf("fa", 5);
        System.out.println(s10);
    }
```

```
    /*
    int length()：返回字符串的长度： return value.length
    char charAt(int index)： 返回某索引处的字符return value[index]
    boolean isEmpty()：判断是否是空字符串：return value.length == 0
    String toLowerCase()：使用默认语言环境，将 String 中的所有字符转换为小写
    String toUpperCase()：使用默认语言环境，将 String 中的所有字符转换为大写
    String trim()：返回字符串的副本，忽略前导空白和尾部空白
    boolean equals(Object obj)：比较字符串的内容是否相同
    boolean equalsIgnoreCase(String anotherString)：与equals方法类似，忽略大小写
    String concat(String str)：将指定字符串连接到此字符串的结尾。 等价于用“+”
    int compareTo(String anotherString)：比较两个字符串的大小
    String substring(int beginIndex)：返回一个新的字符串，它是此字符串的从beginIndex开始截取到最后的一个子字符串。
    String substring(int beginIndex, int endIndex) ：返回一个新字符串，它是此字符串从beginIndex开始截取到endIndex(不包含)的一个子字符串。
    */
    @Test
    public void test2(){
        String s1 = "abcdefg";
        String s2 = "abCde";

        boolean s3 = s1.equals(s2);
        boolean s4 = s1.equalsIgnoreCase(s2);
        System.out.println(s3); // false
        System.out.println(s4); // true

        String s5 = "hello";
        String s6 = " world";
        String s7 = s5.concat(s6);
        System.out.println(s7);

        int s8 = s1.compareTo(s2);
        System.out.println(s8);

        String s9 = s1.substring(2);
        System.out.println(s9);

        String s10 = s1.substring(2, 5);
        System.out.println(s10);
    }

    @Test
    public void test1() {
        String s1 = "abCdEf";
        int s2 = s1.length();
        System.out.println(s2);// 6

        char s3 = s1.charAt(2);
```

```
        System.out.println(s3);// c

        boolean s4 = s1.isEmpty();
        System.out.println(s4); // false

        String s5 = "";
        boolean s6 = s5.isEmpty();
        System.out.println(s6);// true

        String s7 = s1.toLowerCase();
        System.out.println(s1);
        System.out.println(s7);

        String s8 = s1.toUpperCase();
        System.out.println(s8);

        String s9 = "  a jd sfj  ";
        String s10 = s9.trim();
        System.out.println("------" + s9 + "------");
        System.out.println("------" + s10 + "------");
    }
}
```

涉及到String类与其他结构之间的转换

```
package com.test.java;

import org.junit.Test;

import java.io.UnsupportedEncodingException;
import java.util.Arrays;

/**
 * 涉及到String类与其他结构之间的转换
 *
 * @author: my.seaTide
 * @create: 2021/4/6 9:06 下午
 */
public class StringTest {
    /*
    String 与 byte[]之间的转换

    编码：String  --> byte[]：调用String的getBytes()
    解码：byte[] --> String：调用String的构造器

    编码：字符串 --> 字节（看的懂 --> 看不懂的二进制数据）
    解码：编码的逆过程，字节 --> 字符串（看不懂的二进制数据 --> 看得懂）
```

```
说明：解码时，要求解码使用的字符集必须与编码时使用的字符集一致，否则会出现乱码。
 */
@Test
public void test3() throws UnsupportedEncodingException {

    String str1 = "abc123中国";
    byte[] bytes = str1.getBytes();// 使用默认的字符集进行编码
    System.out.println(Arrays.toString(bytes));

    byte[] gbks = str1.getBytes("gbk");
    System.out.println(Arrays.toString(gbks));

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

    String str2 = new String(bytes);// 使用默认的字符集，进行解码
    System.out.println(str2);

    String str3 = new String(gbks);// 出现乱码。原因：编码集和解码集不一致
    System.out.println(str3);

    String str4 = new String(gbks, "gbk");
    System.out.println(str4);
}


/*
String与char[]之间的转换

String --> char[]：调用String的toCharArray()
char[] --> String：调用String的构造器

 */
@Test
public void test2(){

    String str = "abc123";
    char[] chars = str.toCharArray();
    for (int i = 0; i < chars.length; i++) {
        System.out.println(chars[i]);
    }

    char[] arr = new char[]{'h', 'e', 'l', 'l', 'o'};
    String str2 = new String(arr);
    System.out.println(str2);
}

/*
复习：
String与基本数据类型、包装类之间的转换。
```

```
    String --> 基本数据类型、包装类：调用包装类的静态方法：parseXxx()
    基本数据类型、包装类 --> String：调用String重载的valueOf(xxx)
     */
    @Test
    public void test1(){

        String str = "123";
        int i = Integer.parseInt(str);
        System.out.println(i);// 123

        String str2 = String.valueOf(i);
        String str3 = i + "";
        System.out.println(str2);// "123"
        System.out.println(str3); // "123"

        System.out.println(str2 == str3);// false
        System.out.println(str == str3);// false

    }
}
```

String、StringBuffer、StringBuilder三者的异同?

```
package com.test.java;

import org.junit.Test;

/**
 * 关于StringBuffer 和 StringBuilder的使用
 *
 * @author: my.seaTide
 * @create: 2021/4/6 10:01 下午
 */
public class StringBufferBuilderTest {

    /*
    String、StringBuffer、StringBuilder三者的异同?
    不同点：
    String：不可变的字符序列
    StringBuffer：可变的字符序列；线程是安全的，效率低
    StringBuilder：可变的字符序列；JDK5新增的，线程不安全的，效率高；

    相同点：
    底层使用char[]存储

    源码分析：
    String str = new String(); // char[] value = new char[0];
```

```
    String str1 = new String("abc"); // char[] value = new char[]{'a','b','c'};

    StringBuffer sb1 = new StringBuffer(); // char[] value = new char[16];底层创建了一个长度是16的数组。
    sb1.append('a'); value[0] = 'a';
    sb1.append('b'); value[1] = 'b';

    StringBuffer sb2 = new StringBuffer("abc"); // char[] value = new char["abc".length() + 16]

    问题1：System.out.println(sb2.length()); // 3
    问题2：扩容问题：如果要添加的数据底层盛不下了，那就需要扩容底层的数组。
          默认情况下，扩容为原来容量的2倍 + 2，同时将原有数组中的元素复制到新的数组中。

    指导意义：
        开发中建议大家使用：StringBuffer(int capacity) 或 StringBuilder(int capacity)
     */
    @Test
    public void test1(){
        StringBuffer str1 = new StringBuffer("abc");
        str1.setCharAt(0, 'm');
        System.out.println(str1);// mbc
    }
}
```

StringBuffer的常用方法

```
package com.test.java;

import org.junit.Test;

/**
 * StringBuffer常用方法 StringBuilder和他一样
 */
public class StringBufferTest {

    /*
    StringBuffer append(xxx)：提供了很多的append()方法，用于进行字符串拼接
    StringBuffer delete(int start,int end)：删除指定位置的内容
    StringBuffer replace(int start, int end, String str)：把[start,end)位置替换为str
    StringBuffer insert(int offset, xxx)：在指定位置插入xxx
    StringBuffer reverse() ：把当前字符序列逆转

    public int indexOf(String str)
    public String substring(int start,int end)
    public int length()
```

```
    public char charAt(int n )
    public void setCharAt(int n ,char ch)

    总结:
        增:append()
        删:delete(int start,int end)
        改:setCharAt(int n ,char ch) / replace(int start, int end, String str)
        查:charAt(int n )
        插:insert(int offset, xxx)
        长度:length()
        遍历:for() + charAt() / toString()
     */
    @Test
    public void test1(){
        StringBuffer s1 = new StringBuffer("abc");
        s1.append(1);
        s1.append("1");

        System.out.println(s1);
//        s1.delete(1, 3);

//        s1.replace(1,3, "hello");

//        s1.insert(2, "hello");
//        s1.reverse();
        System.out.println(s1);
    }
}
```

- 三者效率

> StringBuilder > StringBuffer > String

String常见算法

```
package com.atguigu.java;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.junit.Test;

/*
 1.模拟一个trim方法，去除字符串两端的空格。

 2.将一个字符串进行反转。将字符串中指定部分进行反转。比如将“abcdefg”反转为”abfedcg”
```

```
 3.获取一个字符串在另一个字符串中出现的次数。
      比如：获取“ab”在 “cdabkkcadkabkebfkabkskab”
      中出现的次数

 4.获取两个字符串中最大相同子串。比如：
   str1 = "abcwerthelloyuiodef“;str2 = "cvhellobnm"//10
   提示：将短的那个串进行长度依次递减的子串与较长
   的串比较。

 5.对字符串中字符进行自然顺序排序。"abcwerthelloyuiodef"
  提示：
  1）字符串变成字符数组。
  2）对数组排序，选择，冒泡，Arrays.sort(str.toCharArray());
  3）将排序后的数组变成字符串。
 */
public class StringExer {

  // 第1题
  public String myTrim(String str) {
    if (str != null) {
      int start = 0;// 用于记录从前往后首次索引位置不是空格的位置的索引
      int end = str.length() - 1;// 用于记录从后往前首次索引位置不是空格的位置的索引

      while (start < end && str.charAt(start) == ' ') {
        start++;
      }

      while (start < end && str.charAt(end) == ' ') {
        end--;
      }
      if (str.charAt(start) == ' ') {
        return "";
      }

      return str.substring(start, end + 1);
    }
    return null;
  }

  // 第2题
  // 方式一：
  public String reverse1(String str, int start, int end) {// start:2,end:5
    if (str != null) {
      // 1.
      char[] charArray = str.toCharArray();
      // 2.
      for (int i = start, j = end; i < j; i++, j--) {
        char temp = charArray[i];
```

```
            charArray[i] = charArray[j];
            charArray[j] = temp;
        }
        // 3.
        return new String(charArray);

    }
    return null;

}

// 方式二：
public String reverse2(String str, int start, int end) {
    // 1.
    String newStr = str.substring(0, start);// ab
    // 2.
    for (int i = end; i >= start; i--) {
        newStr += str.charAt(i);
    } // abfedc
        // 3.
    newStr += str.substring(end + 1);
    return newStr;
}

// 方式三：推荐 （相较于方式二做的改进）
public String reverse3(String str, int start, int end) {// ArrayList list =
new ArrayList(80);
    // 1.
    StringBuffer s = new StringBuffer(str.length());
    // 2.
    s.append(str.substring(0, start));// ab
    // 3.
    for (int i = end; i >= start; i--) {
        s.append(str.charAt(i));
    }

    // 4.
    s.append(str.substring(end + 1));

    // 5.
    return s.toString();

}

@Test
public void testReverse() {
    String str = "abcdefg";
    String str1 = reverse3(str, 2, 5);
    System.out.println(str1);// abfedcg
```

```
}

// 第3题
// 判断str2在str1中出现的次数
public int getCount(String mainStr, String subStr) {
  if (mainStr.length() >= subStr.length()) {
    int count = 0;
    int index = 0;
    // while((index = mainStr.indexOf(subStr)) != -1){
    // count++;
    // mainStr = mainStr.substring(index + subStr.length());
    // }
    // 改进:
    while ((index = mainStr.indexOf(subStr, index)) != -1) {
      index += subStr.length();
      count++;
    }

    return count;
  } else {
    return 0;
  }

}

@Test
public void testGetCount() {
  String str1 = "cdabkkcadkabkebfkabkskab";
  String str2 = "ab";
  int count = getCount(str1, str2);
  System.out.println(count);
}

@Test
public void testMyTrim() {
  String str = "   a   ";
  // str = " ";
  String newStr = myTrim(str);
  System.out.println("---" + newStr + "---");
}

// 第4题
// 如果只存在一个最大长度的相同子串
public String getMaxSameSubString(String str1, String str2) {
  if (str1 != null && str2 != null) {
    String maxStr = (str1.length() > str2.length()) ? str1 : str2;
    String minStr = (str1.length() > str2.length()) ? str2 : str1;
```

```
      int len = minStr.length();

      for (int i = 0; i < len; i++) {// 0 1 2 3 4 此层循环决定要去几个字符

        for (int x = 0, y = len - i; y <= len; x++, y++) {

          if (maxStr.contains(minStr.substring(x, y))) {

            return minStr.substring(x, y);
          }

        }

      }
    }
    return null;
  }

  // 如果存在多个长度相同的最大相同子串
  // 此时先返回String[]，后面可以用集合中的ArrayList替换，较方便
  public String[] getMaxSameSubString1(String str1, String str2) {
    if (str1 != null && str2 != null) {
      StringBuffer sBuffer = new StringBuffer();
      String maxString = (str1.length() > str2.length()) ? str1 : str2;
      String minString = (str1.length() > str2.length()) ? str2 : str1;

      int len = minString.length();
      for (int i = 0; i < len; i++) {
        for (int x = 0, y = len - i; y <= len; x++, y++) {
          String subString = minString.substring(x, y);
          if (maxString.contains(subString)) {
            sBuffer.append(subString + ",");
          }
        }
        System.out.println(sBuffer);
        if (sBuffer.length() != 0) {
          break;
        }
      }
      String[] split = sBuffer.toString().replaceAll(",$", "").split("\\,");
      return split;
    }

    return null;
  }
  // 如果存在多个长度相同的最大相同子串：使用ArrayList
//  public List<String> getMaxSameSubString1(String str1, String str2) {
//    if (str1 != null && str2 != null) {
//      List<String> list = new ArrayList<String>();
```

```
//      String maxString = (str1.length() > str2.length()) ? str1 : str2;
//      String minString = (str1.length() > str2.length()) ? str2 : str1;
//
//      int len = minString.length();
//      for (int i = 0; i < len; i++) {
//        for (int x = 0, y = len - i; y <= len; x++, y++) {
//          String subString = minString.substring(x, y);
//          if (maxString.contains(subString)) {
//            list.add(subString);
//          }
//        }
//        if (list.size() != 0) {
//          break;
//        }
//      }
//      return list;
//    }
//
//    return null;
//  }

  @Test
  public void testGetMaxSameSubString() {
    String str1 = "abcwerthelloyuiodef";
    String str2 = "cvhellobnmiodef";
    String[] strs = getMaxSameSubString1(str1, str2);
    System.out.println(Arrays.toString(strs));
  }

  // 第5题
  @Test
  public void testSort() {
    String str = "abcwerthelloyuiodef";
    char[] arr = str.toCharArray();
    Arrays.sort(arr);

    String newStr = new String(arr);
    System.out.println(newStr);
  }
}
```

日期时间类的使用

- JDK 8之前日期和时间的API测试

```
package com.test.java;

import org.junit.Test;
```

```java
import java.util.Date;

/**
 * JDK 8之前日期和时间的API测试
 */
public class DateTimeTest {

    /*
    java.util.Date类
            |----java.sql.Date类
    1.两个构造器的使用
        > 构造器一：Date()：创建一个对应当前时间的Date对象
        > 构造器二：创建指定毫秒数的Date对象

    2.两个方法的使用
        > toString()：显示当前的年、月、日、时、分、秒
        > getTime()：获取当前Date对象对应的毫秒数。（时间戳）

    3.java.sql.Date对应着数据库中的日期类型的变量
        > 如何实例化
        > 如何将java.util.Date对象转换为java.sql.Date对象
     */
    @Test
    public void test2(){
        // 构造器一：Date()：创建一个对应当前时间的Date对象
        Date date1 = new Date();
        System.out.println(date1.toString());// Thu Apr 08 09:32:27 CST 2021

        System.out.println(date1.getTime());// 1617845547054

        // 构造器二：创建指定毫秒数的Date对象
        Date date2 = new Date(1617845547054L);
        System.out.println(date2.toString());

        // 创建java.sql.Date对象
        java.sql.Date date3 = new java.sql.Date(1617845547054L);
        System.out.println(date3);// 2021-04-08

        // 情况一：
//        Date date4 = new java.sql.Date(1617845547054L);
//        java.sql.Date date5 = (java.sql.Date) date4;
//        System.out.println(date4);// 2021-04-08
//        System.out.println(date5);// 2021-04-08

        // 情况二：报错
        Date date6 = new Date();
        java.sql.Date date7 = (java.sql.Date) date6;
        System.out.println(date7);// ClassCastException
```

```
    }

    // 1、System类中currentTimeMillis()方法
    @Test
    public void test1() {
        long time = System.currentTimeMillis();
        // 返回当前时间与1970年1月1日0时0分0秒之间以毫秒为单位的时间差
        // 时间戳
        System.out.println(time);
    }

}
```

```
package com.test.java;

import org.junit.Test;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

/**
 * JDK 8之前的日期时间的API测试
 *  1、System类中currentTimeMillis();
 *  2、java.util.Date和子类java.sql.Date
 *  3、SimpleDateFormat
 *  4.Calendar
 *
 *
 *
 * @author: my.seaTide
 * @create: 2021/4/12 9:35 下午
 */
public class DateTimeTest {

    /*
    SimpleDateFormat的使用：SimpleDateFormat对日期Date类的格式化和解析
     */
    @Test
    public void testSimpleDateFormat() throws ParseException {
        // 实例化SimpleDateFormat：使用默认的构造器
        SimpleDateFormat sdf = new SimpleDateFormat();

        // 格式化：日期 ---> 字符串
        Date date1 = new Date();
        System.out.println(date1);// Mon Apr 12 21:50:17 CST 2021
```

```
        String date2 = sdf.format(date1);
        System.out.println(date2);// 2021/4/12 下午9:50

        // 解析：格式化的逆过程，字符串 ---> 日期
        String str = "2021/4/12 下午9:50";
        Date date3 = sdf.parse(str);
        System.out.println(date3);


 System.out.println("*********************************************************");
        // 按照指定的方式格式化和解析：调用带参数的构造器
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

        // 格式化：日期 ---> 字符串
        String format1 = simpleDateFormat.format(date1);
        System.out.println(format1);// 2021-04-12 21:56:50

        // 解析：格式化的逆过程，字符串 ---> 日期
        // 要求字符串必须是符合simpleDateFormat识别的格式（通过构造器参数体现），否则，抛
异常
        Date parse = simpleDateFormat.parse(format1);
        System.out.println(parse);
    }

    // 练习1：字符串"2021-04-12" 转换为java.sql.Date
    @Test
    public void testExer() throws ParseException {
        String str = "2021-04-12";
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");

        Date parse = simpleDateFormat.parse(str);

        java.sql.Date date = new java.sql.Date(parse.getTime());
        System.out.println(date);

    }

    /*
    Calendar日历类（抽象类）的使用


     */
    @Test
    public void testCalendar(){
        // 1.实例化
        // 方式一：创建其子类（GregorianCalendar）的对象
        // 方式二：调用其静态方法getInstance()
```

```
        Calendar instance = Calendar.getInstance();
        System.out.println(instance.getClass());

        // 2.常用方法
        // get()
        int i = instance.get(Calendar.DAY_OF_MONTH);
        System.out.println(i);

        // set()
        instance.set(Calendar.DAY_OF_MONTH, 22);
        int i1 = instance.get(Calendar.DAY_OF_MONTH);
        System.out.println(i1);

        // add()
        instance.add(Calendar.DAY_OF_MONTH, 2);
        int days = instance.get(Calendar.DAY_OF_MONTH);
        System.out.println(days);

        // getTime(): 日历类 --> Date
        Date date = instance.getTime();
        System.out.println(date);

        // setTime():Date --> 日历类
        Date date1 = new Date();
        instance.setTime(date1);
        int days1 = instance.get(Calendar.DAY_OF_MONTH);
        System.out.println(days1);
    }
}
```

- JDK 8日期和时间的API测试

```
package com.test.java;

import org.junit.Test;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;

/**
 * JDK 8日期和时间的API测试
 *
 * @author: my.seaTide
 * @create: 2021/4/13 10:05 下午
 */
public class JDK8DateTimeTest {
```

```
    /*
    LocalDate、LocalTime、LocalDateTime的使用
    说明:
        1、LocalDateTime使用频率高
     */
    @Test
    public void test1() {
        // now()：获取当前的日期、时间、日期+时间
        LocalDate localDate = LocalDate.now();
        LocalTime localTime = LocalTime.now();
        LocalDateTime localDateTime = LocalDateTime.now();

        System.out.println(localDate);
        System.out.println(localTime);
        System.out.println(localDateTime);

        // of()：设置指定的年、月、日、时、分、秒。没有偏移量
        LocalDateTime localDateTime1 = LocalDateTime.of(2021, 4, 13, 22, 16,
15);
        System.out.println(localDateTime1);

        // getXxx()
        int dayOfMonth = localDateTime.getDayOfMonth();
        Month month = localDateTime.getMonth();
        int monthValue = localDateTime.getMonthValue();

        System.out.println(dayOfMonth);
        System.out.println(month);
        System.out.println(monthValue);

        // 体现不可变性
        // withXxx()：设置相关的属性
        LocalDateTime withDayOfMonth = localDateTime.withDayOfMonth(20);
        System.out.println(localDateTime);
        System.out.println(withDayOfMonth);

        LocalDateTime withHour = localDateTime.withHour(12);
        System.out.println(withHour);

    }
}
```

- Instant的使用

```
  /*
    Instant的使用

     */
```

```
    @Test
    public void test2() {
        // now(): 获取本初子午线对应的标准时间
        Instant instant = Instant.now();
        System.out.println(instant);// 2021-04-13T14:58:22.717238Z

        // 添加时间的偏移量
        OffsetDateTime offsetDateTime =
instant.atOffset(ZoneOffset.ofHours(8));
        System.out.println(offsetDateTime);// 2021-04-13T22:59:15.517669+08:00

        // 获取对应的毫秒数 --> Date类的getTimes
        long toEpochMilli = instant.toEpochMilli();
        System.out.println(toEpochMilli);

        // 通过给定的毫秒数，获取Instant实例 --> Date(long mills)
        Instant instant1 = Instant.ofEpochMilli(1618326170442L);
        System.out.println(instant1);

    }
```

- DateTimeFormatter类的使用

```
    /*
    DateTimeFormatter的使用
    类似于SimpleDateFormat
     */
    @Test
    public void test3(){
        DateTimeFormatter dateTimeFormatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String format = dateTimeFormatter.format(LocalDateTime.now());
        System.out.println(format);// 2021-04-14 22:06:10

        // 解析
        TemporalAccessor parse = dateTimeFormatter.parse("2021-04-14
22:06:10");
        System.out.println(parse);
    }
```

Java比较器

```
package com.test.java;

import org.junit.Test;

import java.util.Arrays;
```

```java
/**
 * 一、说明：Java中的对象，正常情况下，只能进行比较：== 或 != 。不能使用 > 或 < 的
 *          但是在开发场景中，我们需要对多个对象进行排序，言外之意，就需要比较对象的大小。
 *          如何实现？使用两个接口中的任何一个：Comparable 或 Comparator
 *
 * 二、Comparable接口与Comparator的使用的对比
 *      Comparable接口的方式一旦确定，保证Comparable接口实现类的对象在任何位置都可以比较大小
 *      Comparator接口属于临时性的比较。
 *
 * @author: my.seaTide
 * @create: 2021/4/14 10:26 下午
 */
public class CompareTest {
    /*
    Comparable接口的使用举例：  自然排序
    1、像String、包装类等实现了Comparable接口，重写了compareTo(obj)方法，给出了比较两个对象大小的方式
    2、像String、包装类重写compareTo()方法以后，进行了从小到大的排列
    3、重写compareTo(obj)的规则：
       如果当前对象this大于形参对象obj，则返回正整数，
       如果当前对象this小于形参对象obj，则返回负整数，
       如果当前对象this等于形参对象obj，则返回零。
    4、对于自定义类来说，如果需要排序，我们可以让自定义类实现Comparable接口，重写compareTo(obj)方法
       在compareTo()方法中指明如何排序
     */
    @Test
    public void test1() {
        String[] arr = new String[]{"BB", "NN", "CC", "WW", "AA"};

        Arrays.sort(arr);
        System.out.println(Arrays.toString(arr));
    }

    @Test
    public void test2(){
        Goods[] arr = new Goods[4];
        arr[0] = new Goods("AA", 20);
        arr[1] = new Goods("BB", 10);
        arr[2] = new Goods("CC", 80);
        arr[3] = new Goods("DD", 30);

        Arrays.sort(arr);

        System.out.println(Arrays.toString(arr));
    }

     /*
```

```
    Comparator接口的使用：定制排序
    1、背景：
      当元素的类型没有实现java.lang.Comparable接口而又不方便修改代码，
      或者实现了java.lang.Comparable接口的排序规则不适合当前的操作，
      那么可以考虑使用Comparator的对象来排序
    2、重写compare(Object o1,Object o2)方法，比较o1和o2的大小：
      如果方法返回正整数，则表示o1大于o2；
      如果返回0，表示相等；
      返回负整数，表示o1小于o2

     */
    @Test
    public void test2() {
        String[] arr = new String[]{"AA", "VV", "FF", "DD", "UU", "BB"};
        Arrays.sort(arr, new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return -o1.compareTo(o2);
            }
        });
        System.out.println(Arrays.toString(arr));
    }
}
```

- Goods类

```
package com.test.java;

/**
 * @author: my.seaTide
 * @create: 2021/4/14 10:46 下午
 */
public class Goods implements Comparable{

    private String name;
    private double price;


    public Goods(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
```

```
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Goods{" +
                "name='" + name + '\'' +
                ", price=" + price +
                '}';
    }

    // 指明商品比较大小的方式：按照价格从低到高排序
    @Override
    public int compareTo(Object o) {
        if (o instanceof Goods) {
            Goods goods = (Goods) o;
            // 方式一
            if (this.price > goods.price) {
                return  1;
            } else if (this.price < goods.price) {
                return  -1;
            } else {
                return 0;
            }

            // 方式二
//            return Double.compare(this.price, goods.price);

        }

        throw new RuntimeException("传入的数据类型不一致！");
    }
}
```

枚举类与注解

枚举类的使用

- 自定义枚举类

```
package com.test.java;

/**
 * 一、枚举类的使用
 * 1、枚举类的理解：类的对象只有有限个，确定的。我们称类为枚举类
 * 2、当需要定义一组常量时，强烈建议使用枚举类
 * 3、如果枚举类中只有一个对象，则可以作为单例模式的实现方式。
 *
 * 二、如何定义枚举类
 * 方式一：JDK 5之前，自定义枚举类
 * 方式二：JDK 5，可以使用enum关键字定义枚举类
 *
 * 三、Enum类中的主要方法
 *  values()方法：返回枚举类型的对象数组。该方法可以很方便的遍历所有的枚举值
 *  valueOf(String str)：可以把一个字符串转为对应的枚举类对象。要求字符串必须是枚举类对象的"名字"。如不是，会有运行时异常：
IllegalArgumentException.
 *  toString()：返回当前枚举类对象常量的名称
 *
 * 四、使用enum关键字定义的枚举类实现接口的情况
 *  情况一：实现接口，在enum类中实现抽象方法
 *  情况二：让枚举类的对象分别实现接口中的抽象方法
 *
 * @author: my.seaTide
 * @create: 2021/4/18 2:01 下午
 */
public class SeasonTest {
    public static void main(String[] args) {
        Season summer = Season.SUMMER;
        System.out.println(summer);

        String seasonName = summer.getSeasonName();
        System.out.println(seasonName);
    }

}

// 自定义枚举类
class Season{
    // 1. 声明Season对象的属性：private final修饰
    private final String seasonName;
    private final String seasonDesc;
```

```
    // 2、私有化类的构造器，并给对象属性赋值
    private Season(String seasonName, String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    // 3、提供当前枚举类的多个对象：public static final修饰
    public static final Season SPRING = new Season("春天", "春暖花开");
    public static final Season SUMMER = new Season("夏天", "烈日炎炎");
    public static final Season AUTUMN = new Season("秋天", "秋高气爽");
    public static final Season WINTER = new Season("冬天", "天寒地冻");

    // 4、其他诉求1：获取枚举类对象的属性

    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }

    // 5、其他诉求2：重写toString方法
    @Override
    public String toString() {
        return "Season{" +
                "seasonName='" + seasonName + '\'' +
                ", seasonDesc='" + seasonDesc + '\'' +
                '}';
    }
}
```

- 使用enum关键字定义枚举类

```
package com.test.java;

/**
 * 使用enum关键字定义枚举类
 * 说明：定义的枚举类默认继承于java.lang.Enum类
 *
 * @author: my.seaTide
 * @create: 2021/4/18 2:47 下午
 */
public class EnumTest {
    public static void main(String[] args) {
        Season1 spring = Season1.SPRING;
        System.out.println(spring);
```

```
        System.out.println(spring.getSeasonName());

        System.out.println(Season1.class.getSuperclass());// 父类java.lang.Enum
    }
}

// 使用enum关键字定义枚举类
enum Season1{
    // 1、提供当前枚举类的对象，多个对象之间用","隔开，末尾对象";"结束
    SPRING("春天", "春暖花开"),
    SUMMER("夏天", "烈日炎炎"),
    AUTUMN("秋天", "秋高气爽"),
    WINTER("冬天", "天寒地冻");

    // 2. 声明Season对象的属性：private final修饰
    private final String seasonName;
    private final String seasonDesc;

    // 3、私有化类的构造器，并给对象属性赋值
    private Season1(String seasonName, String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    // 4、其他诉求1：获取枚举类对象的属性

    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }
}
```

- Enum类中的主要方法

1. values()方法：返回枚举类型的对象数组。该方法可以很方便的遍历所有的枚举值
2. valueOf(String str)：可以把一个字符串转为对应的枚举类对象。要求字符串必须是枚举类对象的"名字"。如不是，会有运行时异常：IllegalArgumentException.
3. toString()：返回当前枚举类对象常量的名称

```
package com.test.java;

/**
 * 使用enum关键字定义枚举类
 * 说明：定义的枚举类默认继承于java.lang.Enum类
 *
 * @author: my.seaTide
```

```
 * @create: 2021/4/18 2:47 下午
 */
public class EnumTest {
    public static void main(String[] args) {
        Season1 spring = Season1.SPRING;

        // toString()方法
        System.out.println(spring);

//        System.out.println(spring.getSeasonName());
//        System.out.println(Season1.class.getSuperclass());// 父类
java.lang.Enum

        // values()
        Season1[] values = Season1.values();
        for (int i = 0; i < values.length; i++) {
            System.out.println(values[i].getSeasonName());
        }

        // valueOf()
        Season1 winter = Season1.valueOf("WINTER");
        System.out.println(winter.getSeasonName());

        // 找不到报异常：IllegalArgumentException
        Season1 winter1 = Season1.valueOf("WINTER1");
        System.out.println(winter1);

    }
}

// 使用enum关键字定义枚举类
enum Season1{
    // 1、提供当前枚举类的对象，多个对象之间用","隔开，末尾对象";"结束
    SPRING("春天", "春暖花开"),
    SUMMER("夏天", "烈日炎炎"),
    AUTUMN("秋天", "秋高气爽"),
    WINTER("冬天", "天寒地冻");

    // 2. 声明Season对象的属性：private final修饰
    private final String seasonName;
    private final String seasonDesc;

    // 3、私有化类的构造器，并给对象属性赋值
    private Season1(String seasonName, String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    // 4、其他诉求1：获取枚举类对象的属性
```

```
    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }
}
```

- 使用enum关键字定义的枚举类实现接口的情况

情况一：实现接口，在enum类中实现抽象方法
情况二：让枚举类的对象分别实现接口中的抽象方法

```
package com.test.java;

/**
 * 使用enum关键字定义枚举类
 * 说明：定义的枚举类默认继承于java.lang.Enum类
 *
 * @author: my.seaTide
 * @create: 2021/4/18 2:47 下午
 */
public class EnumTest {
    public static void main(String[] args) {
        // values()
        Season1[] values = Season1.values();
        for (int i = 0; i < values.length; i++) {
            System.out.println(values[i].getSeasonName());
            values[i].show();
        }

        Season1 winter = Season1.valueOf("WINTER");
        // show方法使用
        winter.show();
    }
}

interface Info{
    public abstract void show();
}

// 使用enum关键字定义枚举类
enum Season1 implements Info{
    // 1、提供当前枚举类的对象，多个对象之间用","隔开，末尾对象";"结束
    SPRING("春天", "春暖花开"){
        @Override
```

```
        public void show() {
            System.out.println("春天在哪里");
        }
    },
    SUMMER("夏天", "烈日炎炎"){
        @Override
        public void show() {
            System.out.println("夏天在哪里");
        }
    },
    AUTUMN("秋天", "秋高气爽"){
        @Override
        public void show() {
            System.out.println("秋天在哪里");
        }
    },
    WINTER("冬天", "天寒地冻"){
        @Override
        public void show() {
            System.out.println("冬天在哪里");
        }
    };

    // 2. 声明Season对象的属性：private final修饰
    private final String seasonName;
    private final String seasonDesc;

    // 3、私有化类的构造器，并给对象属性赋值
    private Season1(String seasonName, String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    // 4、其他诉求1：获取枚举类对象的属性

    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }

    @Override
    public void show() {
        System.out.println("这是季节枚举类");
    }
}
```

注解

- 自定义注解

```
package com.test.java;

import java.util.ArrayList;

/**
 * 如何自定义注解：参照@SupperessWarnings定义
 * 1、注解声明为：@interface
 * 2、内部定义成员，通常使用value表示
 * 3、可以指定成员的默认值，使用default定义
 * 4、如果自定义注解没有成员，表明是一个标识作用。
 *
 * 如果注解有成员，在使用注解时，需要指明成员的值。
 * 自定义注解必须配上注解的信息处理流程（使用反射）才有意义
 * 自定义注解通常都会指明两个元注解：Retention、Target
 * 5、JDK 提供的4种注解
 * 元注解：对现有的注解进行解释说明的注解
 * Retention：指定所修饰的Annotation的生命周期：SOURCE\CLASS(默认行为) \ RUNTIME
 *             只有声明为RUNTIME生命周期的注解，才能通过反射获取
 * Target：用于指定被修饰的Annotation 能用于修饰哪些程序元素
 *
 * ***************出现的频率较低**********************
 * Documented：表示所修饰的注解在被javadoc解析时，保留下来
 * Inherited：被它修饰的 Annotation 将具有继承性
 *
 * 6、JDK 8中注解的新特性
 *    6.1、可重复注解：
 *       ① 在MyAnnotation上声明@Repeatable,成员值为MyAnnotations.class
 *       ② MyAnnotation的Target和Retention等元注解与MyAnnotations相同
 *    6.2、类型注解：
 *       ① ElementType.TYPE_PARAMETER 表示该注解能写在类型变量的声明语句中（如：泛型声明）。
 *       ② ElementType.TYPE_USE 表示该注解能写在使用类型的任何语句中。
 *
 *
 * @author: my.seaTide
 * @create: 2021/4/18 9:44 下午
 */
public class AnnotationTest {

    public static void main(String[] args) {

        @SuppressWarnings({"unused", "rawtypes"})
        ArrayList list = new ArrayList();
    }
}
```

```
// 注解的使用
@MyAnnotation(value = "world")
class Person{
    String name;
    String age;

    public Person() {
    }

    // 注解的使用
    @MyAnnotation
    public Person(String name, String age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }
}
```

```
package com.test.java;

/**
 * 注解的定义
 *
 * @author: my.seaTide
 * @create: 2021/4/18 9:58 下午
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    String value() default "hello";
}
```

- JDK 8中注解的新特性

```
public class MyAnnotationTest {}

// 可重复注解
@MyAnnotation(value = "abc")
@MyAnnotation(value = "123")
class Person{
    String name;
    int age;

    public Person() {
    }

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

// 类型注解：ElementType.TYPE_PARAMETER
class Generic<@MyAnnotation T>{

    // 类型注解：ElementType.TYPE_USE
    public void show() throws @MyAnnotation RuntimeException{
        ArrayList<@MyAnnotation String> list = new ArrayList<>();

        int num = (@MyAnnotation int)100L;
    }
}
```

- MyAnnotation自定义注解

```
package com.test.java;

import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.*;

@Repeatable(MyAnnotations.class)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, MODULE,
TYPE_PARAMETER, TYPE_USE})
@Retention(RetentionPolicy.CLASS)
public @interface MyAnnotation {
    String value() default "hello";
```

```
}
```

- MyAnnotations自定义注解

```
package com.test.java;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.*;

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, MODULE})
@Retention(RetentionPolicy.CLASS)
public @interface MyAnnotations {
    MyAnnotation[] value();
}
```

集合

集合框架的概述及集合框架

```
/**
 * 一、集合框架的概述
 * 1、集合、数组都是对多个数据进行存储操作的结构，简称Java容器。
 *    说明：此时的存储，主要指的是内存层面的存储，不涉及到持久化的存储（.txt,.jpg,.avi）
 *
 * 2、数组在存储多个数据方面
 *     特点：> 一旦初始化以后，其长度就确定了。
 *          > 数组一旦定义好，其元素的类型也就确定了。我们也就只能操作指定类型的数据了。
 *            比如：String[] arr;int[] arr1; Object[] arr2;
 *     缺点：> 一旦初始化以后，其长度就不可修改。
 *          > 数组中提供的方法非常有限，对于添加、删除、插入数据等操作，非常不便，同时效率不
高。
 *          > 获取数组中实际元素的个数的需求，数组没有现成的属性或方法可用
 *          > 数组存储数据的特点：有序、可重复。对于无序、不可重复的需求，不能满足。
 *
 *
 *   二、集合框架
 *    |---Collection接口：单列集合，用来存储一个一个的对象
 *        |---List接口：存储有序的，可重复的数据。   ---> "动态"数组
 *            |---ArrayList、LinkedList、Vector
 *
 *        |---Set接口：存储无序的，不可重复的数据     ---> 高中讲的"集合"
 *            |---HashSet、LinkedHashSet、TreeSet
 *
 *    |---Map接口：双列集合、用来存储一对（key - value）的数据 --->高中函数：y=f(x)
 *        |---HashMap、LinkedHashMap、TreeMap、HashTable、Properties
```

```
 *
 */
```

Collection接口中的方法的使用

```
public class CollectionTest {
    @Test
    public void test1() {
        Collection coll = new ArrayList();

        // add(Object e)：将元素e添加到集合coll中
        coll.add("AA");
        coll.add("BB");
        coll.add(123);
        coll.add(new Date());

        // size()：获取添加的元素的个数
        System.out.println(coll.size());// 4

        // addAll(Collection coll1)：将coll1集合中的元素添加到当前的集合中
        Collection coll1 = new ArrayList();
        coll1.add("DD");
        coll1.add(456);
        coll.addAll(coll1);

        System.out.println(coll.size());// 6
        System.out.println(coll);// [AA, BB, 123, Mon Apr 19 22:49:49 CST 2021,
DD, 456]

        // clear()：清空集合元素
        coll.clear();

        // isEmpty()：判断当前集合是否为空
        System.out.println(coll.isEmpty());

    }
}
```

```
package com.test.java;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

/**
```

```
 * Collection接口中声明的方法的测试
 *
 * 向Collection接口的实现类的对象中添加数据obj时，要求obj所在类要重写equals()
 *
 */
public class CollectionTest {

    @Test
    public void test1(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new String("abc"));
        coll.add(false);
        coll.add(new Person("张三", 20));

        // 1.contains(Object obj)：判断当前集合中是否包含obj
        System.out.println(coll.contains(123));
        System.out.println(coll.contains(new String("abc")));// true

        // 我们在判断时会调用obj对象所在类的equals()方法
        System.out.println(coll.contains(new Person("张三", 20)));// true

        // 2.containsAll(Collection coll1)：判断形参coll1中的所有元素是否都存在于当前
集合中
        Collection coll1 = Arrays.asList(123,456);
        System.out.println(coll.containsAll(coll1));// true


    }

    @Test
    public void test2() {
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new String("abc"));
        coll.add(false);
        coll.add(new Person("张三", 20));

        // 3.remove(Object obj)：从当前集合中移除obj元素。
        coll.remove(123);
        System.out.println(coll);
        coll.remove(new Person("张三", 20));
        System.out.println(coll);

        // 4.removeAll(Collection coll1)：差集：从当前集合中移除coll1中所有的元素。
        Collection coll1 = Arrays.asList(123,456);
        coll.removeAll(coll1);
```

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

    @Test
    public void test3(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new String("abc"));
        coll.add(false);
        coll.add(new Person("张三", 20));

        // 5.retainAll(Collection coll1)：交集：获取当前集合和coll1集合的交集，并返回给当前集合
//        Collection coll1 = Arrays.asList(123, 456, 890);
//        coll.retainAll(coll1);
//        System.out.println(coll);// [123, 456]

        // 6.equals(Object obj)：要想返回true，需要当前集合和形参集合的元素都相同。
        Collection coll2 = new ArrayList();
        coll2.add(123);
        coll2.add(456);
        coll2.add(new String("abc"));
        coll2.add(false);
        coll2.add(new Person("张三", 20));

        System.out.println(coll.equals(coll2));// true
    }

    @Test
    public void test4(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new String("abc"));
        coll.add(false);
        coll.add(new Person("张三", 20));

        // 7.hashCode()：返回当前对象的哈希值
        System.out.println(coll.hashCode());

        // 8.集合 ---> 数组：toArray()
        Object[] array = coll.toArray();
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }

        // 拓展：数组 ---> 集合：调用Arrays类的静态方法asList()
        List<String> list = Arrays.asList(new String[]{"AA", "BB", "CC"});
```

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

        // 注意：
        // 错误的
        List<int[]> ints = Arrays.asList(new int[]{123, 456});
        System.out.println(ints.size()); // 1

        // 正确的
        List<Integer> integers = Arrays.asList(new Integer[]{123, 456});
        System.out.println(integers.size());// 2

        // 9.iterator()：返回Iterator接口的实例，用于遍历集合元素。

    }
}
```

使用迭代器Iterator遍历Collection

```
package com.test.java;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;


/**
 * 集合元素的遍历操作，使用迭代器Iterator接口
 * 1、内部的方法：hasNext() 和 next()
 * 2、集合对象每次调用iterator()方法都得到一个全新的迭代器对象，默认游标都在集合的第一个元素之前。
 * 3、内部定义了remove()，可以在遍历的时候，删除集合中的元素。此方法不同于集合直接调用remove
 *
 * @author: my.seaTide
 * @create: 2021/4/20 10:08 下午
 */
public class Iterator {

    @Test
    public void test1(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(false);
        coll.add(new String("Tom"));

        java.util.Iterator iterator = coll.iterator();
```

```
//        System.out.println(iterator.next());
        // hasNext()：判断是否还有下一个元素
        while (iterator.hasNext()){
            // next()：指针下移，将下移以后集合位置上的元素返回
            System.out.println(iterator.next());
        }
    }

    // 测试Iterator中的remove()
    // 如果还未调用next()或在上一次调用next()方法之后已经调用了remove方法，再调用
remove()都会报IllegalStateException。
    @Test
    public void test2(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(false);
        coll.add(new String("Tom"));

        java.util.Iterator iterator = coll.iterator();

        while (iterator.hasNext()){
            Object next = iterator.next();
            if ("Tom".equals(next)) {
                iterator.remove();
            }
        }

        iterator = coll.iterator();
        while (iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }
}
```

foreach循环遍历集合和数组

```
package com.test.java;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;

/**
 * JDK5.0，新增了foreach循环，用于遍历集合、数组
 */
public class ForTest {
```

```java
    @Test
    public void test1() {
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new String("Tom"));
        coll.add(false);

        // for(集合元素的类型 局部变量 : 集合对象)
        // 内部仍然调用了迭代器。
        for (Object obj : coll) {
            System.out.println(obj);
        }
    }

    @Test
    public void test2() {
        int[] arr = new int[]{1,2,3,4,5,6};

        // for(数组元素的类型 局部变量 : 数组对象)
        for (int i : arr) {
            System.out.println(i);
        }
    }

    // 面试题
    @Test
    public void test3(){
        String[] arr = new String[]{"MM", "MM", "MM"};

        // for()
//        for (int i = 0; i < arr.length; i++) {
//            arr[i] = "GG";
//        }
//
//        for (int i = 0; i < arr.length; i++) {
//            System.out.println(arr[i]); // GG
//        }

        // foreach方法
        for (String s : arr) {
            s = "AA";
        }

        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);// MM
        }
    }
}
```

List接口常用实现类

```
package com.test.java;

import java.util.LinkedList;

/**
 * 1.List接口框架
 *      |---Collection接口：单列集合，用来存储一个一个的对象
 *          |---List接口：存储有序的、可重复的数据。 ---> “动态”数组，替换原有的数组
 *              |---ArrayList：作为List接口的主要实现类：线程不安全的，效率高；底层使用
Object[] elementData存储
 *              |---LinkedList：对于频繁的插入、删除操作，使用此类效率比ArrayList高；底
层使用双向链表存储
 *              |---Vector：作为List接口的古老实现类：线程安全的，效率低；底层使用
Object[] elementData存储
 *
 * 2.ArrayList源码分析：
 *    2.1.jdk 7 的情况下：
 *      ArrayList list = new ArrayList(); //底层创建了长度是10的Object[]数组
elementData
 *      list.add(123);// elementData[0] = new Integer(123);
 *      ...
 *      list.add(11);// 如果此次的添加导致底层elementData数组容量不够，则扩容。
 *      默认情况下，扩容为原来的容量的1.5倍，同时需要将原有数组中的数据复制到新的数组中。
 *
 *      结论：建议开发中使用带参的构造器：ArrayList list = new ArrayList(int
capacity)
 *    2.2.jdk 8 的情况下：
 *      ArrayList list = new ArrayList();// 底层Object[] elementData初始化为{}，并
没有创建长度为10的数组
 *      list.add(123);// 第一次调用add()时，底层才创建了长度10的数组，并将数据123添加到
elementData[0]
 *      ...
 *      后续的添加和扩容操作与jdk 7 无异。
 *    2.3.小结：
 *      jdk 7中的ArrayList的对象的创建类似于单例的饿汉式，而jdk 8中的ArrayList的对象的创
建类似于单例的懒汉式，
 *      延迟了数组的创建，节省内存。
 *
 * 3.LinkedList的源码分析：
 *    LinkedList list = new LinkedList();// 内部声明了Node类型的first和last属性，默
认值为null。
 *    list.add(123);// 将123封装到Node中，创建了Node对象。
 *
 *    其中，Node定义为：体现了LinkedList的双向链表的说法。
 *     private static class Node<E> {
 *         E item;
 *         Node<E> next;
```

```
 *          Node<E> prev;
 *
 *          Node(Node<E> prev, E element, Node<E> next) {
 *              this.item = element;
 *              this.next = next;
 *              this.prev = prev;
 *          }
 *      }
 *
 * 4.Vector的源码分析：jdk7 和 jdk8 中通过Vector()构造器创建对象时，底层都创建了长度为10的数组
 *                      在扩容方面，默认扩容为原来的数组长度的2倍。
 *
 * 面试题：ArrayList、LinkedList、Vector三者的异同?
 * 相同点：三个类都实现了List接口，存储数据的特点相同：存储有序的、可重复的数据
 * 不同点：见上1
 *
 */
public class ListTest {
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
    }
}
```

List接口中的常用方法

```
package com.test.java;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

/**
 * List接口中的常用方法
 *
 */
public class ListMethodTest {
    /*
    void add(int index, Object ele):在index位置插入ele元素
    boolean addAll(int index, Collection eles):从index位置开始将eles中的所有元素添加进来
    Object get(int index):获取指定index位置的元素
    int indexOf(Object obj):返回obj在集合中首次出现的位置
    int lastIndexOf(Object obj):返回obj在当前集合中末次出现的位置
    Object remove(int index):移除指定index位置的元素，并返回此元素
    Object set(int index, Object ele):设置指定index位置的元素为ele
```

```
List subList(int fromIndex, int toIndex):返回从fromIndex到toIndex位置的子集合

总结：常用方法
增：add(Object obj)
删：remove(int index) / remove(Object obj)
改：set(int index, Object ele)
查：get(int index)
插：add(int index, Object ele)
长度：size()
遍历：
    1、Iterator迭代器方式
    2、增强for循环（foreach）
    3、普通的循环

 */

@Test
public void test3() {
    ArrayList list = new ArrayList();
    list.add(123);
    list.add(456);
    list.add(new String("abc"));
    list.add(false);
    // 1、Iterator迭代器方式
    Iterator iterator = list.iterator();
    while (iterator.hasNext()){
        System.out.println(iterator.next());
    }

    System.out.println("**********************");
    // 2、增强for循环（foreach）
    for (Object obj : list) {
        System.out.println(obj);
    }

    System.out.println("**********************");
    // 3、普通的循环
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}

@Test
public void test2() {
    ArrayList list = new ArrayList();
    list.add(123);
    list.add(456);
    list.add(new String("abc"));
    list.add(false);
```

```
        list.add(456);

        // int indexOf(Object obj):返回obj在集合中首次出现的位置
        int i = list.indexOf(456);
        System.out.println(i);// 1

        // int lastIndexOf(Object obj):返回obj在当前集合中末次出现的位置
        int i1 = list.lastIndexOf(456);
        System.out.println(i1);// 4

        // Object remove(int index):移除指定index位置的元素，并返回此元素
        Object remove = list.remove(2);
        System.out.println(remove);// abc

        // Object set(int index, Object ele):设置指定index位置的元素为ele
        list.set(1, "MM");
        System.out.println(list);

        // List subList(int fromIndex, int toIndex):返回从fromIndex到toIndex位置
的子集合
        List list1 = list.subList(0, 2);
        System.out.println(list1);
    }

    @Test
    public void test1() {
        ArrayList list = new ArrayList();
        list.add(123);
        list.add(456);
        list.add(new String("abc"));
        list.add(false);

        System.out.println(list);
        // void add(int index, Object ele):在index位置插入ele元素
        list.add(1, "poi");
        System.out.println(list);

        //  boolean addAll(int index, Collection eles):从index位置开始将eles中的所
有元素添加进来
        List list1 = Arrays.asList(1, 2, 3);
        list.addAll(2, list1);
        System.out.println(list.size());// 8

        // Object get(int index):获取指定index位置的元素
        Object o = list.get(1);
        System.out.println(o);
    }
}
```

Set接口实现类对比

```
package com.test.java;

import org.junit.Test;

import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;

/**
 * Set接口的框架：
 *
 * |---Collection接口：单列集合，用来存储一个一个的对象
 *      |---Set接口：存储无序的，不可重复的数据     ---> 高中讲的"集合"
 *           |---HashSet：作为Set接口的主要实现类：线程不安全的；可以存储null值
 *              |---LinkedHashSet：作为HashSet的子类；遍历其内部数据时，可以按照添加的顺序遍历
 *                                    对于频繁的遍历操作，LinkedHashSet效率高于HashSet
 *           |---TreeSet：可以按照添加对象的指定属性，进行排序。
 *
 *  1、Set接口中没有额外定义新的方法，使用的都是Collection中声明过的方法。
 *
 *  2、要求：
 *      > 向Set中添加的数据，其所在的类一定要重写hashCode()和equals()
 *      > 重写的equals()和hashCode()尽可能保持一致性：相等的对象必须具有相等的散列码
 *      重写两个方法的小技巧：对象中用作equals()方法比较的Field，都应该用来计算hashCode值。
 *
 * @author: my.seaTide
 * @create: 2021/4/23 10:26 下午
 */
public class SetTest {
    /*
    一、Set：存储无序的，不可重复的数据
        以HashSet为例说明：
        1. 无序性：不等于随机性。存储的数据在底层数组中并非按照数组索引的顺序添加。而是根据数据的哈希值决定的。

        2. 不可重复性：保证添加的元素按照equals()判断时，不能返回true，即：相同的元素只能添加一个。

    二、添加元素的过程：以HashSet为例：
        我们向HashSet中添加元素a，首先调用元素a所在类的hashCode()方法，计算元素a的哈希值，
        此哈希值接着通过某种算法计算出在HashSet底层数组中的存放位置（即为：索引位置），判断数组此位置上
        是否已经有元素：
```

```
            如果此位置上没有其他元素，则元素a添加成功。 --->  情况1
            如果此位置上有其他元素b（或以链表形式存在的多个元素），则比较元素a与元素b的hash值：
                如果哈希值不相同，则元素a添加成功。--->情况2
                如果哈希值相同，进而需要调用元素a所在类的equals()方法：
                    equals()返回true，元素a添加失败
                    equals()返回false，则元素a添加成功。 --->情况3

        对于添加成功的情况2和情况3而言：元素a 与已经存在指定索引位置上数据以链表的方式存储。
        jdk 7：元素a放到数组中，指向原来的元素。
        jdk 8：原来的元素在数组中，指向元素a。
        总结：七上八下

        HashSet底层：数组+链表的结构。

     */
    @Test
    public void test1(){
        Set set = new HashSet();
        set.add(123);
        set.add(456);
        set.add(new String("abc"));
        set.add(false);
        set.add(new String("456"));

        Iterator iterator = set.iterator();
        while (iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }

    /*
    LinkedHashSet的使用
    LinkedHashSet作为HashSet的子类，在添加数据的同时，每个数据还维护了两个引用，记录此数据前一个数据和后一个数据。
    优点：对于频繁的遍历操作，LinkedHashSet效率高于HashSet
     */
    @Test
    public void test2(){
        Set set = new LinkedHashSet();
        set.add(123);
        set.add(456);
        set.add(new String("abc"));
        set.add(false);
        set.add(new String("456"));

        Iterator iterator = set.iterator();
        while (iterator.hasNext()){
            System.out.println(iterator.next());
```

```
        }
    }
}
```

- TreeSet的使用

```
package com.test.java;

import org.junit.Test;

import java.util.Comparator;
import java.util.Iterator;
import java.util.TreeSet;

/**
 * @author: my.seaTide
 * @create: 2021/4/24 3:28 下午
 */
public class TreeSetTest {
    /*
    1、向TreeSet中添加的数据，要求是相同类的对象。
    2、两种排序方式：自然排序（实现Comparable接口） 和 定制排序(实现Comparator接口)


    3、自然排序中，比较两个对象是否相同的标准为：compareTo() 返回0，不再是equals()。
    4、定制排序中，比较两个对象是否相同的标准为：compare() 返回0，不再是equals()。
     */
    @Test
    public void test1() {
        TreeSet set = new TreeSet();

        // 失败：要求是相同类的对象。
//        set.add(123);
//        set.add("abc");
//        set.add(false);

        // 举例1
//        set.add(123);
//        set.add(789);
//        set.add(456);

        // 举例2
        set.add(new User("Tom", 12));
        set.add(new User("Jack", 20));
        set.add(new User("Andy", 40));
        set.add(new User("Jack", 10));

        Iterator iterator = set.iterator();
```

```
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }

    @Test
    public void test2() {
        // 定制排序 按照年龄从小到达排序
        Comparator com = new Comparator() {
            @Override
            public int compare(Object o1, Object o2) {
                if (o1 instanceof User && o2 instanceof User) {
                    User u1 = (User) o1;
                    User u2 = (User) o2;

                    return Integer.compare(u1.getAge(), u2.getAge());

                } else {
                    throw new RuntimeException("数据类型不匹配");
                }
            }
        };

        TreeSet set = new TreeSet(com);
        // 举例2
        set.add(new User("Tom", 12));
        set.add(new User("Jack", 20));
        set.add(new User("Mary", 40));
        set.add(new User("Andy", 40));
        set.add(new User("Jack", 10));

        Iterator iterator = set.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

- User类

```
package com.test.java;

import java.util.Objects;

/**
 * @author: my.seaTide
 * @create: 2021/4/24 3:46 下午
 */
```

```java
public class User implements Comparable{
    private String name;
    private int age;

    public User() {
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return age == user.age && Objects.equals(name, user.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

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

```
    // 按照名字从大到小，年龄从小到大排序
    @Override
    public int compareTo(Object o) {
        if (o instanceof User) {
            User user = (User) o;
            int compareTo = -this.name.compareTo(user.name);

            // 判断名字是否相同
            if (compareTo != 0) {
                return compareTo;
            } else {
                return Integer.compare(this.age, user.age);
            }

        } else {
            throw new RuntimeException("数据类型不一致");
        }
    }
}
```

Map实现类的结构

```
package com.test.java1;

import org.junit.Test;

import java.util.HashMap;
import java.util.LinkedHashMap;

/**
 * 一、Map的实现类的结构：
 * |---Map：双列数据，存储key-value对的数据 ---类似于高中的函数：y = f(x)
 *       |---HashMap：作为Map的主要实现类：线程不安全的，效率高；能存储null的key和value
 *           |---LinkedHashMap：保证在遍历map元素时，可以按照添加的顺序实现遍历。
 *                                原因：在原有的HashMap底层结构基础上，添加了一对指针，指向前一个和后一个元素
 *                                对于频繁的遍历操作，此类执行效率要高于HashMap。
 *       |---TreeMap：保证按照添加的key-value对进行排序，实现排序遍历。此时考虑key的自然排序或定制排序
 *                    底层使用红黑树
 *       |---Hashtable：作为古老的实现类：线程安全的，效率低；不能存储null的key和value
 *           |---Properties：常用来处理配置文件。key和value都是String类型
 *
 * HashMap的底层:数组 + 链表 （jdk7及以前）
 *               数组 + 链表 + 红黑树（jdk 8）
 *
 * 面试题：
 * 1、HashMap的底层实现原理?
```

```
 * 2、HashMap和Hashtable的异同?
 * 3、CurrentHashMap 与 Hashtable的异同?
 *
 *
 * 二、Map结构的理解
 *  Map中的key：无序的，不可重复的，使用Set存储所有的key ---> key所在的类要重写equals()和hashCode() (以HashMap为例)
 *  Map中的value：无序的，可重复的，使用Collection存储所有的value ---> value所在的类要重写equals()
 *  一个键值对：key-value构成了一个Entry对象。
 *  Map中的entry：无序的、不可重复的，使用Set存储所有的entry
 *
 *
 * 三、HashMap的底层实现原理?以jdk7为例说明：
 *      HashMap map = new HashMap();
 *      在实例化以后，底层创建了长度是16的一维数组Entry[] table。
 *      ...可能已经执行过多次put...
 *      map.put(key1,value1)：
 *      首先，调用key1所在类的hashCode()计算key1的哈希值，此哈希值经过某种算法计算以后，得到在Entry数组中的存放位置。
 *      如果此位置上的数据为空，此时的key1-value1添加成功。 ---情况1
 *      如果此位置上的数据不为空，（意味着此位置上存在一个或多个数据（以链表的形式存在）），比较key1和已经存在的一个或镀铬数据的哈希值：
 *          如果key1的哈希值与已经存在的数据的哈希值都不相同，此时key1-value1添加成功。---情况2
 *          如果key1的哈希值与已经存在的某一个数据（key2-value2）的哈希值相同，继续比较：调用key1所在类的equals()方法：
 *              如果equals()返回false：此时key1-value1添加成功。---情况3
 *              如果equals()返回true：使用value1替换value2。
 *
 *      补充：关于情况2和情况3：此时key1-value1和原来的数据以链表的方式存储。
 *
 *      在不断的添加过程中，会涉及到扩容问题，当超出临界值（且要存放的位置非空）时，默认的扩容方式：扩容为原来容量的2倍，并将原有的数据复制过来。
 *
 *      jdk8 相较于 jdk7在底层实现方面的不同：
 *      1、new HashMap()：底层没有创建一个长度为16的数组
 *      2、jdk8底层的数组是：Node[]，而非Entry[]
 *      3、首次调用put方法时，底层创建长度为16的数组
 *      4、jdk7底层结构只有：数组 + 链表。jdk8中底层结构：数组 + 链表 + 红黑树。
 *          当数组的某一个索引位置上的元素以链表形式存在的数据个数 >8 且当前数组的长度 >64时，
 *          此时此索引位置上的所有数据改为使用红黑树存储。
 *
 *      DEFAULT_INITIAL_CAPACITY : HashMap的默认容量：16
 *      DEFAULT_LOAD_FACTOR：HashMap的默认加载因子：0.75
 *      threshold：扩容的临界值，等于容量 * 填充因子：16 * 0.75 = 12
 *      TREEIFY_THRESHOLD：Bucket中链表长度大于该默认值，转化为红黑树：8
 *      MIN_TREEIFY_CAPACITY：桶中的Node被树化时最小的hash表容量：64
```

```
 *
 * 四、LinkedHashMap的底层实现原理（了解）
 *    源码中：
 *    static class Entry<K,V> extends HashMap.Node<K,V> {
 *         Entry<K,V> before, after;// 能够记录添加的元素的先后顺序
 *         Entry(int hash, K key, V value, Node<K,V> next) {
 *             super(hash, key, value, next);
 *         }
 *     }
 *
 *
 * @author: my.seaTide
 * @create: 2021/4/24 8:57 下午
 */
public class MapTest {
    @Test
    public void test1() {
        HashMap map = new HashMap();
        map = new LinkedHashMap();
        map.put("abc", "123");
        map.put("abcd", "456");
        map.put("abcde", "789");

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

Map中的常用方法

```
package com.test.java1;

import org.junit.Test;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Map中定义的方法
 *
 * 添加、删除、修改操作：
 *    Object put(Object key,Object value)：将指定key-value添加到(或修改)当前map对象
中
 *    void putAll(Map m):将m中的所有key-value对存放到当前map中
 *    Object remove(Object key)：移除指定key的key-value对，并返回value
 *    void clear()：清空当前map中的所有数据
 * 元素查询的操作：
```

```
 *    Object get(Object key)：获取指定key对应的value
 *    boolean containsKey(Object key)：是否包含指定的key
 *    boolean containsValue(Object value)：是否包含指定的value
 *    int size()：返回map中key-value对的个数
 *    boolean isEmpty()：判断当前map是否为空
 *    boolean equals(Object obj)：判断当前map和参数对象obj是否相等
 * 元视图操作的方法：
 *    Set keySet()：返回所有key构成的Set集合
 *    Collection values()：返回所有value构成的Collection集合
 *    Set entrySet()：返回所有key-value对构成的Set集合
 *
 *  总结：常用方法
 *  增：put(Object key,Object value)
 *  删：remove(Object key)
 *  改：put(Object key,Object value)
 *  查：get(Object key)
 *  长度：size()
 *  遍历：Set keySet() / Collection values() / Set entrySet()
 *
 */
public class MapMethodTest {
    /*
     添加、删除、修改操作：
      Object put(Object key,Object value)：将指定key-value添加到(或修改)当前map对象
中
      void putAll(Map m):将m中的所有key-value对存放到当前map中
      Object remove(Object key)：移除指定key的key-value对，并返回value
      void clear()：清空当前map中的所有数据
     */
    @Test
    public void test1() {
        Map map = new HashMap();
        map.put("AA", "123");
        map.put("BB", "456");
        map.put(3, "789");
        System.out.println(map);

        Map map1 = new HashMap();
        map1.put("CC", "111");
        map1.put("DD", "222");
        map.putAll(map1);
        System.out.println(map);

        Object value = map.remove("CC");
        System.out.println(value);

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

```
/*
元素查询的操作：
  Object get(Object key)：获取指定key对应的value
  boolean containsKey(Object key)：是否包含指定的key
  boolean containsValue(Object value)：是否包含指定的value
  int size()：返回map中key-value对的个数
  boolean isEmpty()：判断当前map是否为空
  boolean equals(Object obj)：判断当前map和参数对象obj是否相等
 */
@Test
public void test2() {
    Map map = new HashMap();
    map.put("AA", "123");
    map.put("BB", "456");
    map.put(3, "789");

    System.out.println(map.get(3));

    System.out.println(map.containsKey("AA"));
    System.out.println(map.containsValue("123"));

    System.out.println(map.size());

    map.clear();
    System.out.println(map.isEmpty());
}

/*
元视图操作的方法：
 Set keySet()：返回所有key构成的Set集合
 Collection values()：返回所有value构成的Collection集合
 Set entrySet()：返回所有key-value对构成的Set集合
*/
@Test
public void test3() {
    Map map = new HashMap();
    map.put("AA", "123");
    map.put("BB", "456");
    map.put(3, "789");

    Set set = map.keySet();
    for (Object obj : set) {
        System.out.println(obj);
    }

    System.out.println();

    Collection values = map.values();
```

```
        for (Object obj : values) {
            System.out.println(obj);
        }

        System.out.println();
        //  Set entrySet()：返回所有key-value对构成的Set集合
        // 方式一
        Set entrySet = map.entrySet();
        for (Object obj : entrySet) {
            // entrySet集合中的元素都是entry
            Map.Entry entry = (Map.Entry) obj;
            System.out.println(entry.getKey() + "--->" + entry.getValue());
        }

        // 方式二
        Set set1 = map.keySet();
        for (Object key : set1) {
            Object value = map.get(key);
            System.out.println(key + "--->" + value);
        }

    }
}
```

Properties处理属性文件

```
package com.test.java1;

import java.io.FileInputStream;
import java.util.Properties;

/**
 * @author: my.seaTide
 * @create: 2021/4/26 10:11 下午
 */
public class PropertiesTest {

    // Properties：常用来处理配置文件，key和value都是String类型
    public static void main(String[] args) throws Exception {
        Properties pros = new Properties();

        FileInputStream fis = new FileInputStream("jdbc.properties");
        pros.load(fis);// 加载流对应的文件

        String name = pros.getProperty("name");
        String password = pros.getProperty("password");

        System.out.println("name = " + name + ", password = " + password);
    }
```

```
}
```

- jdbc.properties配置文件

```
name=Tom
password=abc123
```

Collections工具类

```
package com.test.java1;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

/**
 * Collections: 操作Collection、Map的工具类
 *
 * 面试题：Collection 和 Collections的区别?
 *
 * @author: my.seaTide
 * @create: 2021/4/26 10:34 下午
 */
public class CollectionsTest {

    /*
    排序操作：
    reverse(List)：反转 List 中元素的顺序
    shuffle(List)：对 List 集合元素进行随机排序
    sort(List)：根据元素的自然顺序对指定 List 集合元素按升序排序
    sort(List, Comparator)：根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
    swap(List, int,  int)：将指定 list 集合中的 i 处元素和 j 处元素进行交换

    查找、替换：
    Object max(Collection)：根据元素的自然顺序，返回给定集合中的最大元素
    Object max(Collection, Comparator)：根据 Comparator 指定的顺序，返回给定集合中的
最大元素
    Object min(Collection)
    Object min(Collection, Comparator)
    int frequency(Collection, Object)：返回指定集合中指定元素的出现次数
    void copy(List dest,List src)：将src中的内容复制到dest中
    boolean replaceAll(List list, Object oldVal, Object newVal)：使用新值替换List
对象的所有旧值
     */
```

```java
    @Test
    public void test2() {
        List list = new ArrayList();
        list.add(1);
        list.add(8);
        list.add(-2);
        list.add(9);
        list.add(6);

        // 报异常：IndexOutOfBoundsException("Source does not fit in dest")
//        List dest = new ArrayList();
//        Collections.copy(dest, list);

        // 正确写法
        List dest = Arrays.asList(new Object[list.size()]);
        System.out.println(dest.size());
        Collections.copy(dest, list);

        System.out.println(dest);
    }


    @Test
    public void test1() {
        List list = new ArrayList();
        list.add(1);
        list.add(8);
        list.add(-2);
        list.add(9);
        list.add(6);
        list.add(6);

        System.out.println(list);

//        Collections.reverse(list);
//        Collections.shuffle(list);
//        Collections.sort(list);
//        Collections.swap(list, 1,2);
//        System.out.println(list);

        Comparable max = Collections.max(list);
        System.out.println(max);

        Comparable min = Collections.min(list);
        System.out.println(min);

        int i = Collections.frequency(list, 6);
        System.out.println(i);
    }
```

```
    /*
    Collections 类中提供了多个synchronizedXxx()方法，该方法可使将指定集合包装成线程同步的集合，
    从而可以解决多线程并发访问集合时的线程安全问题
     */
    @Test
    public void test3() {
        List list = new ArrayList();
        list.add(1);
        list.add(8);
        list.add(-2);
        list.add(9);
        list.add(6);

        // 返回的list1即为线程安全的集合
        List list1 = Collections.synchronizedList(list);
    }
}
```

泛型

泛型的使用

```
package com.test.java;

import org.junit.Test;

import java.util.*;

/**
 * 泛型的使用：
 * 1.jdk 5.0新增的特性
 *
 * 2.在集合中使用泛型：
 *   总结：
 *   1.1、集合接口或集合类在jdk5.0时都修改为带泛型的结构。
 *   1.2、在实例化集合类时，可以指明具体的泛型类型。
 *   1.3、指明完以后，在集合类或接口中凡是定义类或接口时，内部结构（比如：方法、构造器、属性等）使用到类的泛型的位置，都指定为实例化的泛型类型。
 *        比如：add(E e) --> 实例化以后：add(Integer e)
 *   1.4、注意点：泛型的类型必须是类，不能是基本数据类型。需要用到基本数据类型的位置，拿包装类替换。
 *   1.5、如果实例化时，没有指明泛型的类型。默认类型为java.lang.Object类型。
 *
 */
public class GenericTest {
```

```java
    // 在集合中使用泛型之前的情况
    @Test
    public void test1() {
        ArrayList list = new ArrayList();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        // 问题一：类型不安全
        list.add("abc");

        for (Object obj : list) {
            // 问题二：强转时，可能出现ClassCastException异常
            int i = (Integer) obj;
            System.out.println(i);
        }

    }

    // 在集合中使用泛型的情况：以ArrayList举例
    @Test
    public void test2() {
          // 泛型的类型不能是基本数据类型
//        ArrayList<int> list = new ArrayList<int>();
        ArrayList<Integer> list = new ArrayList<Integer>();

        list.add(89);
        list.add(88);
        list.add(95);
        list.add(94);
        list.add(91);
        // 编译时，就会进行类型检查，保证数据的安全
//        list.add("abc");

         // 方式一：
//        for (Integer i : list) {
//            // 避免了强转操作
//            int score = i;
//            System.out.println(score);
//        }

        // 方式二：
        Iterator<Integer> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
```

```
    // 在集合中使用泛型的情况：以HashMap举例
    @Test
    public void test3() {
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        //        HashMap<String, Integer> map = new HashMap<>(); // jdk 7 新特性 类型推断
        map.put("a", 123);
        map.put("b", 789);
        // 编译时，就会进行类型检查，保证数据的安全
//        map.put(123, "789");

        Set<Map.Entry<String, Integer>> entrySet = map.entrySet();

        // 方式一：
//        Iterator<Map.Entry<String, Integer>> iterator = entrySet.iterator();
//        while (iterator.hasNext()) {
//            Map.Entry<String, Integer> next = iterator.next();
//            String key = next.getKey();
//            Integer value = next.getValue();
//            System.out.println("key = " + key + ", value = " + value);
//        }

        // 方式二：
        for (Map.Entry<String, Integer> entry : entrySet) {
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println("key = " + key + ", value = " + value);
        }
    }
}
```

自定义泛型类

```
package com.test.java;

import org.junit.Test;

import java.util.ArrayList;
import java.util.List;

/**
 * 如何自定义泛型结构：泛型类、泛型接口；泛型方法
 *
 * 一、关于自定义泛型类、泛型接口
 *
 * 注意点：
 * 1. 泛型类可能有多个参数，此时应将多个参数一起放在尖括号内。比如：<E1,E2,E3>
```

```
 * 2. 泛型类的构造器如下：public GenericClass(){}。而下面是错误的：public
GenericClass<E>(){}
 * 3. 实例化后，操作原来泛型位置的结构必须与指定的泛型类型一致。
 * 4. 泛型不同的引用不能相互赋值。
 * >尽管在编译时ArrayList<String>和ArrayList<Integer>是两种类型，但是，在运行时只有一个
ArrayList被加载到JVM中。
 * 5. 泛型如果不指定，将被擦除，泛型对应的类型均按照Object处理，但不等价于Object。
 *    经验：泛型要使用一路都用。要不用，一路都不要用。
 * 6. 如果泛型结构是一个接口或抽象类，则不可创建泛型类的对象。
 * 7. jdk1.7，泛型的简化操作：ArrayList<Fruit> flist = new ArrayList<>();
 * 8. 泛型的指定中不能使用基本数据类型，可以使用包装类替换。
 * 9. 在类/接口上声明的泛型，在本类或本接口中即代表某种类型，可以作为非静态属性的类型、
    非静态方法的参数类型、非静态方法的返回值类型。但在静态方法中不能使用类的泛型。
 * 10. 异常类不能是泛型的
 * 11. 不能使用new E[]。但是可以：E[] elements = (E[])new Object[capacity];
 *     参考：ArrayList源码中声明：Object[] elementData，而非泛型参数类型数组。
 * 12.父类有泛型，子类可以选择保留泛型也可以选择指定泛型类型：
 *  子类不保留父类的泛型：按需实现
 *    > 没有类型 擦除
 *    > 具体类型
 *  子类保留父类的泛型：泛型子类
 *    > 全部保留
 *    > 部分保留
 * 结论：子类必须是“富二代”，子类除了指定或保留父类的泛型，还可以增加自己的泛型
 *
 * 二、关于自定义泛型方法：
 * 1、泛型方法：在方法中出现了泛型的结构，泛型参数与类的泛型参数没有任何关系
 *    换句话说：泛型方法所属的类是不是泛型类都没有关系。
 * 2、泛型方法可以声明为静态的。原因：泛型参数是在调用方法时确定的，并非在实例化类时确定。
 *
 *
 */
public class GenericTest1 {

    @Test
    public void test1(){
        // 如果定义了泛型类，实例化没有指明类的泛型，则认为此泛型类型为Object类型
        // 要求：如果定义了类是带泛型的，建议在实例化时要指明类的泛型。
//        Order order = new Order("Tom", 13, "ss");
//        Object orderT = order.getOrderT();
//        System.out.println(orderT);

        // 建议实例化时，指明类的泛型
        Order<String> order = new Order<String>("Tom", 13, "abc");
        String orderT = order.getOrderT();
        System.out.println(orderT);
    }

    @Test
```

```
    public void test2() {
        SubOrder order = new SubOrder();
        // 由于子类在继承带泛型的父类时，指明了泛型类型。则实例化子类对象时，不再需要指明泛
型。
        order.setOrderT(123);
        System.out.println(order.getOrderT());

        // 子类也是泛型类的情况
        SubOrder1<String> order1 = new SubOrder1<>();
        order1.setOrderT("abc");
        System.out.println(order1.getOrderT());
    }

    @Test
    public void test3() {
        // 泛型不同的引用不能相互赋值
        ArrayList<String> list1 = null;
        ArrayList<Integer> list2 = null;
//        list1 = list2;
    }

    // 泛型方法测试
    @Test
    public void test4() {
        Order<String> order = new Order<>();
        Integer[] integer = new Integer[]{1, 2, 3, 4, 5, 6};

        // 泛型方法再调用时，指明泛型参数的类型
        List<Integer> list = Order.copyFromArrayToList(integer);
        System.out.println(list);

        // 调用子类中重写父类中的泛型方法
        List<Integer> list1 = SubOrder.copyFromArrayToList(integer);
        System.out.println(list1);
    }
}
```

- Order类

```
package com.test.java;

import java.util.ArrayList;
import java.util.List;

public class Order<T> {
    String orderName;
    int orderId;
    T orderT;
```

```
    public Order(){};

    public Order(String orderName, int orderId, T orderT) {
        this.orderName = orderName;
        this.orderId = orderId;
        this.orderT = orderT;
    }

    // 如下的三个方法都不是泛型方法
    public T getOrderT() {
        return orderT;
    }

    public void setOrderT(T orderT) {
        this.orderT = orderT;
    }

    @Override
    public String toString() {
        return "Order{" +
                "orderName='" + orderName + '\'' +
                ", orderId=" + orderId +
                ", orderT=" + orderT +
                '}';
    }

    // 静态方法中不能使用类的泛型
//    public static void show(T orderT) {
//        System.out.println(orderT);
//    }

    // 泛型方法：在方法中出现了泛型的结构，泛型参数与类的泛型参数没有任何关系
    // 换句话说：泛型方法所属的类是不是泛型类都没有关系。
    public static <E> List<E> copyFromArrayToList(E[] arr){
        ArrayList<E> list = new ArrayList<>();

        for (E i : arr) {
            list.add(i);
        }
        return list;
    }
}
```

- SubOrder类

```
package com.test.java;
```

```
import java.util.ArrayList;
import java.util.List;

public class SubOrder extends Order<Integer>{// SubOrder不是泛型类

    // 子类中也可以重写父类中的泛型方法
    public static <E> List<E> copyFromArrayToList(E[] arr){
        ArrayList<E> list = new ArrayList<>();

        for (E i : arr) {
            list.add(i);
        }
        return list;
    }
}
```

- SubOrder1类

```
package com.test.java;

public class SubOrder1<T> extends Order<T>{// SubOrder1<T>仍然是泛型类
}
```

泛型在继承方面的体现和通配符的使用

```
package com.test.java1;

import org.junit.Test;

import java.util.*;

/**
 * 1.泛型在继承方面的体现
 *
 * 2.通配符的使用
 *
 * 3.有限制条件的通配符的使用
 */
public class GenericTest1 {

    /*
    1.泛型在继承方面的体现

    虽然类A是类B的父类，但是G<A> 和G<B>二者不具备子父类关系，二者是并列关系。
    补充：类A是类B的父类，A<G> 是 B<G>的父类

     */
    @Test
```

```
    public void test1() {
        Object obj = null;
        String str = null;
        obj = str;

        Object[] arr1 = null;
        String[] arr2 = null;
        arr1 = arr2;

        Date date = new Date();
        // 编译不通过
//        str = date;

        List<Object> list1 = null;
        List<String> list2 = null;

        // 编译不通过，此时的list1和list2的类型不具有子父类关系
//        list1 = list2;

        /*
        反证法：
        假设list1 = list2
        list1.add(123); 导致混入非String的数据，出错
         */

        show(list1);
        show1(list2);

    }

    public void show(List<Object> list) {}
    public void show1(List<String> list) {}

    @Test
    public void test2() {
        List<String> list = null;
        AbstractList<String> list1 = null;
        ArrayList<String> list2 = null;

        list = list2;
        list1 = list2;

        List<String> list3 = new ArrayList<>();
    }

    /*
    2.通配符的使用
      通配符：?
```

```
        类A是类B的父类，G<A> 和 G<B>是没有关系的，二者共同的父类是：G<?>
     */
    @Test
    public void test3() {
        List<Object> list1 = null;
        List<String> list2 = null;

        List<?> list = null;
        list = list1;
        list = list2;

        // 编译通过
//        print(list1);
//        print(list2);

        //
        List<String> list3 = new ArrayList<>();
        list3.add("AA");
        list3.add("BB");
        list3.add("CC");
        list = list3;
        // 添加：对于List<?>不能向其内部添加数据。
        // 除了添加null之外。
//        list.add("DD");
//        list.add("?");

        list.add(null);

        // 读取
        Object o = list.get(1);
        System.out.println(o);

    }

    public void print(List<?> list) {
        Iterator<?> iterator = list.iterator();
        while(iterator.hasNext()) {
            Object next = iterator.next();
            System.out.println(next);
        }
    }

    /*
    3.有限制条件的通配符的使用
    ? extends A: <= A (无穷小, A]
            G<? extends A> 可以作为G<A>和G<B>的父类，  其中B是A的子类

    ? super A: >= A [A, 无穷大)
            G<? super A>可以作为G<A>和G<B>的父类，其中B是A的父类
```

```
     */
    @Test
    public void test4(){
        List<? extends Person> list = null;

        List<Student> list1 = new ArrayList<>();
        List<Person> list2 = new ArrayList<>();
        List<Object> list3 = new ArrayList<>();

        list = list1;
        list = list2;
//        list = list3;编译不通过

        List<? super Person> list4 = null;
//        list4 = list1;编译不通过
        list4 = list2;
        list4 = list3;

        // 读取数据
        list = list1;
        Person p = list.get(0);
        // 编译不通过
//        Student p1 = list.get(0);

        list4 = list2;
        Object o = list4.get(0);
        // 编译不通过
//        Person o1 = list4.get(0);

        // 写入数据
        // 编译不通过
//        list.add(new Student());

        list4.add(new Person());
        list4.add(new Student());
        // 编译不通过
//        list4.add(new Object());

    }
}
```

Person是Student的父类

IO流

File类的使用

```
package com.test.java;

import org.junit.Test;

import java.io.File;

/**
 * File类的使用
 *
 * 1、File类的一个对象，代表一个文件或一个文件目录（俗称：文件夹）
 * 2、File类声明在java.io包下
 *
 *
 *
 * @author: my.seaTide
 * @create: 2021/4/29 10:13 下午
 */
public class FileTest {
    /*
    1、如何创建File类的实例
        new File(String filepath)
        new File(String parentPath, String childPath)
        new File(String parentFile, String childPath)

    2、
    相对路径：相较于某个路径下，指明的路径。
    绝对路径：包含盘符在内的文件或文件目录的路径

    3、路径分隔符
    windows：\\
    unix：/

     */
    @Test
    public void test1() {
        // 构造器1：
        File file1 = new File("hello.txt");// 相对路径
        File file2 = new
File("/Users/houbingxu/java/workspace_idea/day08/hello.txt");

        System.out.println(file1);
        System.out.println(file2);

        // 构造器2：
```

```
        File file3 = new File("/Users/houbingxu/java/",
"workspace_idea/day08");
        System.out.println(file3);

        // 构造器3:
        File file4 = new File(file3, "hi.txt");
        System.out.println(file4);

    }
}
```

File类的常用方法

```
package com.test.java2;

import org.junit.Test;

import java.io.File;
import java.io.IOException;
import java.util.Date;

/**
 * File类的常用方法
 *
 * 1.File类中涉及到关于文件或文件目录的创建、删除、重命名、修改时间、文件大小等方法，并未涉及
 *    到写入或读取文件内容的操作。如果需要读取或写入文件内容，必须使用IO流来完成。
 * 2.后续File类的对象常会作为参数传递到流的构造器中，指明读取或写入的“终点”。
 */
public class FileMethodTest {

    /*
    File类的获取功能:
    public String getAbsolutePath():获取绝对路径
    public String getPath() ：获取路径
    public String getName() ：获取名称
    public String getParent():获取上层文件目录路径。若无，返回null
    public long length() ：获取文件长度（即：字节数）。不能获取目录的长度。
    public long lastModified() ：获取最后一次的修改时间，毫秒值
     */
    @Test
    public void test1() {
        File file1 = new File("hello.txt");
        File file2 = new File("D:\\workspace_idea\\IO\\hi.txt");

        System.out.println(file1.getAbsolutePath());
        System.out.println(file1.getPath());
        System.out.println(file1.getName());
        System.out.println(file1.getParent());
```

```
        System.out.println(file1.length());
        System.out.println(new Date(file1.lastModified()));

        System.out.println();

        System.out.println(file2.getAbsolutePath());
        System.out.println(file2.getPath());
        System.out.println(file2.getName());
        System.out.println(file2.getParent());
        System.out.println(file2.length());
        System.out.println(new Date(file2.lastModified()));
    }

    /*
    如下的两个方法适用于文件目录:
    public String[] list() ：获取指定目录下的所有文件或者文件目录的名称数组
    public File[] listFiles() ：获取指定目录下的所有文件或者文件目录的File数组
    */
    @Test
    public void test2() {
        File file = new File("D:\\workspace_idea\\JavaSenior");

        String[] list = file.list();
        for (String f : list) {
            System.out.println(f);
        }

        System.out.println();

        File[] files = file.listFiles();
        for (File f : files) {
            System.out.println(f);
        }
    }

    /*
    File类的重命名功能:
    public boolean renameTo(File dest):把文件重命名为指定的文件路径

    比如: file1.renameTo(file2)为例:
    要抢保证返回为true，需要file1在硬盘中是存在的，且file2不能在硬盘中存在。
     */
    @Test
    public void test3() {
        File file1 = new File("hello.txt");
        File file2 = new File("D:\\workspace_idea\\IO\\hi.txt");
        boolean renameTo = file1.renameTo(file2);
        System.out.println(renameTo);
    }
```

```
    /*
    File类的判断功能：
    public boolean isDirectory()：判断是否是文件目录
    public boolean isFile() ：判断是否是文件
    public boolean exists() ：判断是否存在
    public boolean canRead() ：判断是否可读
    public boolean canWrite() ：判断是否可写
    public boolean isHidden() ：判断是否隐藏
     */
    @Test
    public void test4() {
        File file1 = new File("hello.txt");
        File file2 = new File("D:\\workspace_idea\\IO");

        System.out.println(file1.isDirectory());
        System.out.println(file1.isFile());
        System.out.println(file1.exists());
        System.out.println(file1.canRead());
        System.out.println(file1.canWrite());
        System.out.println(file1.isHidden());

        System.out.println();

        System.out.println(file2.isDirectory());
        System.out.println(file2.isFile());
        System.out.println(file2.exists());
        System.out.println(file2.canRead());
        System.out.println(file2.canWrite());
        System.out.println(file2.isHidden());
    }

    /*
    File类的创建功能：
    public boolean createNewFile() ：创建文件。若文件存在，则不创建，返回false
    public boolean mkdir() ：创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。
    public boolean mkdirs() ：创建文件目录。如果上层文件目录不存在，一并创建

    注意事项：如果你创建文件或者文件目录没有写盘符路径，那么，默认在项目路径下。
     */
    @Test
    public void test5() throws IOException {
        File file1 = new File("hello.txt");
        if (!file1.exists()) {
            boolean newFile = file1.createNewFile();
            System.out.println(newFile);
        } else {
            System.out.println("该文件已存在1");
```

```
        }

        // mkdir()
        File file2 = new File("D:\\workspace_idea\\IO\\IO1");
        if (!file2.exists()) {
            file2.mkdir();
            System.out.println("创建目录成功2");
        } else {
            System.out.println("该文件已存在2");
        }

        // mkdirs()
        File file3 = new File("D:\\workspace_idea\\IO\\IO1\\IO2\\IO3");
        if (!file3.exists()) {
            file3.mkdirs();
            System.out.println("创建目录成功3");
        } else {
            System.out.println("该文件已存在3");
        }
    }

    /*
    File类的删除功能：
    public boolean delete()：删除文件或者文件夹
    删除注意事项：
    Java中的删除不走回收站。
    要删除一个文件目录，请注意该文件目录内不能包含文件或者文件目录
     */
    @Test
    public void test6() {
        File file1 = new File("D:\\workspace_idea\\IO\\IO1\\IO2");
        if (file1.exists()) {
            boolean delete = file1.delete();
            System.out.println(delete);
        } else {
            System.out.println("文件不存在");
        }
    }
}
```

流的分类和流的体系结构

- 流的分类
    1. 操作数据单位：字节流、字符流
    2. 数据的流向：输入流、输出流
    3. 流的角色：节点流、处理流
- 流的体系结构

抽象基类	节点流（或文件流）	缓冲流（处理流的一种）
InputStream	FileInputStream (read(byte[] cbuf))	BufferedInputStream (read(byte[] buffer))
OutputStream	FileOutputStream (write(byte[] cbuf, 0, len))	BufferedOutputStream (write(byte[] buffer, 0, len))
Reader	FileReader (read(char[] cbuf))	BufferedReader (read(char[] cbuf) / readLine())
Writer	FileWriter (write(char[] cbuf, 0, len))	BufferedWriter (write(char[] cbuf, 0, len))

节点流（或文件流）方法的使用

- FileReader读入数据

```
/*
将day09下的hello.txt文件内容读入到程序中，并输出到控制台

说明点：
1、read()的理解：返回读入的一个字符，如果达到文件末尾，返回-1
2、异常的处理：为了保证流资源一定可以执行关闭操作，需要使用try-catch-finally处理
3、读入的文件一定要存在，否则就会报FileNotFoundException。
 */
@Test
public void TestFileReader() {
    FileReader fr = null;
    try {
        // 1.实例化File类的对象，指明要操作的文件
        File file = new File("hello.txt");

        // 2.提供具体的流
        fr = new FileReader(file);

        // 3.数据的写入
        // read()：返回读入的一个字符，如果达到文件末尾，返回-1
        int data;
        while ((data = fr.read()) != -1) {
            System.out.print((char) data);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        // 4.流的关闭操作
        try {
            if (fr != null)
                fr.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
    }

    /*
    对read()操作升级：使用read的重载方法
     */
    @Test
    public void TestFileReader1() {
        FileReader fr = null;
        try {
            // 1、File类的实例化
            File file = new File("hello.txt");

            // 2、FileReader流的实例化
            fr = new FileReader(file);

            // 3、读入的操作
            // read(char[], cubf)：返回每次读入cubf数组中的字符的个数。如果达到文件末尾，返回-1
            char[] cubf = new char[5];
            int len;
            while((len = fr.read(cubf)) != -1) {
                // 方式一：
                for (int i = 0; i < len; i++) {
                    System.out.print(cubf[i]);
                }

                // 方式二：
//                String str = new String(cubf, 0, len);
//                System.out.print(str);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fr != null) {
                // 4、资源的关闭
                try {
                    fr.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

- FileWriter输出数据

```
 /*
    从内存中写出数据到硬盘的文件里
```

```
    说明：
    1、输出操作，对应的File可以不存在的。并不会报异常
    2、File对应的硬盘中的文件如果不存在：在输出的过程中，会自动创建此文件。
       File对应的硬盘中的文件如果存在：
            如果流使用的构造器是：FileWriter(file, false) / FileWriter(file)：对原有文件的覆盖
            如果流使用的构造器是：FileWriter(file,true)：不会对原有文件覆盖，而是在原有文件基础上追加内容

     */
    @Test
    public void testFileWriter() {
        FileWriter fr = null;
        try {
            // 1、提供File类的对象，指明写出到的文件
            File file = new File("hello1.txt");

            // 2、提供FileWriter的对象，用于数据的写出
            fr = new FileWriter(file);

            // 3、写出的操作
            fr.write("I have a dream!\n");
            fr.write("you need to have a dream!");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fr != null) {
                // 4、流资源的关闭
                try {
                    fr.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

- 使用FileReader和FileWriter实现文本文件的复制

```
/*
    使用FileReader和FileWriter实现文本文件的复制
     */
    @Test
    public void testFileReaderFileWriter() {
        FileReader fr = null;
        FileWriter fw = null;
        try {
```

```
            // 1、创建File类的对象，指明读入和写出的文件
            File srcFile = new File("hello.txt");
            File destFile = new File("hello2.txt");

            // 2、创建输入流和输出流的对象
            fr = new FileReader(srcFile);
            fw = new FileWriter(destFile);

            // 3、数据的读入和写出操作
            char[] cbuf = new char[5];
            int len;// 记录每次读入到cbuf数组中的字符的个数
            while ((len = fr.read(cbuf)) != -1) {
                // 每次写出len个字符
                fw.write(cbuf, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 4、关闭流资源
            try {
                if (fw != null)
                    fw.close();
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    if (fr != null)
                        fr.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

- FIleInputStream和FIleOutputStream的使用

```
package com.test.java;

import org.junit.Test;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * 测试FileInputStream和FileOutputStream的使用
```

```
 *
 * 结论:
 *  1、对于文本文件(.txt,.java,.c,.cpp...)，使用字符流处理
 *  2、对于非文本文件(.jpg,.mp3,.mp4,.avi,.doc,.ppt...)，使用字节流处理
 *  3、
 *
 * @author: my.seaTide
 * @create: 2021/5/2 2:22 下午
 */
public class FileInputOutputStreamTest {

    // 使用字节流FileInputStream处理文本文件，可能出现乱码
    @Test
    public void testFileInputOutputStream() {
        FileInputStream fis = null;
        try {
            // 1、造文件
            File file = new File("hello.txt");

            // 2、创建流
            fis = new FileInputStream(file);

            // 3、读数据
            byte[] b = new byte[5];
            int len;// 记录每次读取的字节的个数
            while ((len = fis.read(b)) != -1) {
                String str = new String(b, 0, len);
                System.out.print(str);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fis != null) {
                try {
                    // 4、关闭流
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    // 实现对图片的复制操作
    @Test
    public void testFileInputOutputStream1() {
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
```

```
            File srcFile = new File("图片.jpg");
            File destFile = new File("图片1.jpg");

            fis = new FileInputStream(srcFile);
            fos = new FileOutputStream(destFile);

            // 复制的过程
            byte[] buffer = new byte[1024];
            int len;
            while ((len = fis.read(buffer)) != -1) {
                fos.write(buffer, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {

            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

缓冲流方法的使用

```
package com.test.java;

import org.junit.Test;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/**
 * 处理流之一：缓冲流的使用
 *
 * 1、缓冲流
 * BufferedInputStream
 * BufferedOutputStream
 * BufferedReader
 * BufferedWriter
 *
 * 2、作用：提高流的读取、写入的速度
 *    提高读写速度的原因：内部提供了一个缓冲区
 *
 * 3、处理流，就是"套接"在已有流的基础上
 *
 * @author: my.seaTide
 * @create: 2021/5/5 8:23 下午
 */
public class BufferedTest {

    // 使用BufferedInputStream和BufferedOutputStream实现非文本文件的复制
    @Test
    public void testBufferedStream() {
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
        try {
            // 1、造文件
            File srcFile = new File("图片.jpg");
            File destFile = new File("图片2.jpg");

            // 2、造流
            // 2.1、创建节点流
            FileInputStream fis = new FileInputStream(srcFile);
            FileOutputStream fos = new FileOutputStream(destFile);

            // 2.2、创建缓冲流
            bis = new BufferedInputStream(fis);
            bos = new BufferedOutputStream(fos);

            // 3、复制的细节：读取、写入
            byte[] buffer = new byte[5];
            int len;
            while ((len = bis.read(buffer)) != -1) {
                bos.write(buffer, 0, len);
            }
        } catch (IOException e) {
```

```
            e.printStackTrace();
        } finally {
            // 4、资源关闭
            // 要求：先关闭外层的流，再关闭内层的流
            if (bis != null) {
                try {
                    bis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (bos != null) {
                try {
                    bos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            // 说明：关闭外层流的同时，内层流也会自动的进行关闭。关于内层流的关闭，我们可以省略。
//          fis.close();
//          fos.close();
        }
    }

    // 使用BufferedReader和BufferedWriter实现文本文件的复制
    @Test
    public void testBufferedReaderWriter() {
        BufferedReader br = null;
        BufferedWriter bw = null;
        try {
            // 1、创建缓冲流
            br = new BufferedReader(new FileReader(new File("hello.txt")));
            bw = new BufferedWriter(new FileWriter(new File("hello3.txt")));

            // 2、复制文件操作
            // 方式一：
//            char[] buffer = new char[1024];
//            int len;
//            while ((len = br.read(buffer)) != -1) {
//                bw.write(buffer, 0, len);
//            }

            // 方式二：
            String data;
            while ((data = br.readLine()) != null) {
                // 方法1：
//                bw.write(data + "\n");// data中不包含换行符
```

```
                // 方法2:
                bw.write(data);
                bw.newLine();// 提供换行的操作
            }


        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 3、关闭资源
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (bw != null) {
                try {
                    bw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

转换流的使用

```
package com.test.java;

import org.junit.Test;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

/**
 * 处理流之二：转换流的使用
 * 1、转换流：属于字符流
 *    InputStreamReader：将一个字节的输入流转换为字符的输入流
 *    OutputStreamWriter：将一个字符的输出流转换为字节的输出流
 *
```

```
 * 2、作用：提供字节流与字符流之间的转换
 *
 * 3、解码：字节、字节数组 ---> 字符数组、字符串
 *     编码：字符数组、字符串 ---> 字节、字节数组
 *
 * 4、字符集
 * 常见的编码：
 * ASCII：美国标准信息交换码。用一个字节的7位可以表示。
 * ISO8859-1：拉丁码表。欧洲码表用一个字节的8位表示。
 * GB2312：中国的中文编码表。最多两个字节编码所有字符
 * GBK：中国的中文编码表升级，融合了更多的中文文字符号。最多两个字节编码
 * Unicode：国际标准码，融合了目前人类使用的所有字符。为每个字符分配唯一的字符码。所有的文字都用两个字节来表示。
 * UTF-8：变长的编码方式，可用1-4个字节来表示一个字符。
 *
 * @author: my.seaTide
 * @create: 2021/5/5 9:37 下午
 */
public class InputStreamReaderTest {

    // InputStreamReader的使用，实现字节的输入流到字符的输入流的转化
    @Test
    public void test1() {
        InputStreamReader isr = null;
        try {
            FileInputStream fis = new FileInputStream("hello.txt");

//            isr = new InputStreamReader(fis);// 使用默认的字符集
            // 参数2指明了字符集，具体使用哪个字符集，取决于文件保存时使用的字符集
            isr = new InputStreamReader(fis, "UTF-8");

            char[] cbuf = new char[20];
            int len;
            while ((len = isr.read(cbuf)) != -1) {
                String s = new String(cbuf, 0, len);
                System.out.println(s);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (isr != null) {
                try {
                    isr.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```
    // 综合使用InputStreamReader和OutputStreamWriter
    @Test
    public void test2() {
        InputStreamReader isr = null;
        OutputStreamWriter osw = null;
        try {
            // 1、创建文件和流
            isr = new InputStreamReader(new FileInputStream(new
File("hello.txt")), "utf-8");
            osw = new OutputStreamWriter(new FileOutputStream(new
File("hello_gbk.txt")), "gbk");

            // 2、读写过程
            char[] cbuf = new char[20];
            int len;
            while ((len = isr.read(cbuf)) != -1) {
                osw.write(cbuf, 0, len);
            }
            System.out.println("success");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 3、关闭资源
            if (isr != null) {
                try {
                    isr.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (osw != null) {
                try {
                    osw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

标准的输入、输出流

```
/*
    1、标准的输入、输出流
    System.in：标准的输入流，默认从键盘输入
    System.out：标准的输出流，默认从控制台输出

    System.in和System.out分别代表了系统标准的输入和输出设备
    默认输入设备是：键盘，输出设备是：显示器
    System.in的类型是InputStream
    System.out的类型是PrintStream，其是OutputStream的子类
    FilterOutputStream 的子类
    重定向：通过System类的setIn，setOut方法对默认设备进行改变。
        > public static void setIn(InputStream in)
        > public static void setOut(PrintStream out)

    练习：
    从键盘输入字符串，要求将读取到的整行字符串转成大写输出。然后继续
    进行输入操作，直至当输入“e”或者“exit”时，退出程序。

    方法一：使用Scanner实现，调用next()返回一个字符串
    方法二：使用System.in实现。System.in ---> 转换流 ---> BufferedReader()的
readLine()方法
    */
    public static void main(String[] args) {
        BufferedReader br = null;
        try {
            InputStreamReader isr = new InputStreamReader(System.in);
            br = new BufferedReader(isr);

            while (true) {
                System.out.print("请输入字符串：");
                String data = br.readLine();
                if ("e".equalsIgnoreCase(data) ||
"exit".equalsIgnoreCase(data)) {
                    System.out.println("程序结束！");
                    break;
                }

                String s = data.toUpperCase();
                System.out.println(s);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
```

```
                    e.printStackTrace();
                }
            }
        }
    }
```

对象流

```
package com.test.java;

import org.junit.Test;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * 对象流的使用
 * 1、ObjectInputStream 和 ObjectOutputStream
 * 2、作用：用于存储和读取基本数据类型数据或对象的处理流。它的强大之处就是可以把Java中的对象写入到数据流中，也能把对象从数据源中还原回来。
 *
 * 3、要想一个java对象是可序列化的，需要满足相应的要求。见Person.java
 * 4、序列化机制：
 *      对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久的保存在磁盘上，
 *      或通过网络将这种二进制流传输到另一个网络节点。当其他程序获取了这种二进制流，就可以恢复成原来的Java对象。
 *
 * @author: my.seaTide
 * @create: 2021/5/10 10:12 下午
 */
public class ObjectOutputInputStreamTest {
    /*
    序列化过程：将内存中的java对象保存到磁盘中或通过网络传输出去
    使用ObjectOutputStream实现
     */
    @Test
    public void testObjectOutputStream() {
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("object.dat"));

            oos.writeObject(new String("我爱北京天安门"));
            oos.flush();// 刷新操作
```

```
            oos.writeObject(new Person("张三", 26));
            oos.flush();

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (oos != null) {
                try {
                    oos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    /*
    反序列化：将磁盘文件中的对象还原为内存中的一个java对象
    使用ObjectInputStream实现
     */
    @Test
    public void testObjectInputStream() {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream("object.dat"));

            Object obj = ois.readObject();
            String str = (String) obj;
            System.out.println(str);

            Person p = (Person) ois.readObject();
            System.out.println(p);

        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            if (ois != null) {
                try {
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

- 自定义类实现序列化和反序列化操作

```java
package com.test.java;

import java.io.Serializable;

/**
 * 自定义类实现序列化和反序列化操作
 * Person类需要满足如下的要求，方可序列化
 *
 * 1、需要实现接口：Serializable
 * 2、当前类提供一个全局常量：serialVersionUID
 * 3、除了当前Person类需要实现Serializable接口之外，还必须保证其内部所有属性也必须是可序列化的。
 *   （默认情况下，基本数据类型可序列化）
 *
 * 补充：
 *   ObjectOutputStream和ObjectInputStream不能序列化static和transient修饰的成员变量
 * @author: my.seaTide
 * @create: 2021/5/10 10:46 下午
 */
public class Person implements Serializable {

    private static final long serialVersionUID = 443654365472L;

    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

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

RandomAccessFile随机存取文件流的使用

```
package com.test.java;

import org.junit.Test;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

/**
 * RandomAccessFile的使用：
 * 1、RandomAccessFile直接继承于java.lang.Object类，实现了DataInput和DataOutput接口
 * 2、RandomAccessFile既可以作为一个输入流、又可以作为一个输出流
 * 3、如果RandomAccessFile作为输出流时，写出到的文件如果不存在，则在执行过程中自动创建
 *    如果写出到的文件存在，则会对原有文件内容进行覆盖。（默认情况下，从头覆盖）
 * 4、可以通过相关的操作，实现RandomAccessFile“插入”数据的效果
 */
public class RandomAccessFileTest {

    /*
    RandomAccessFile既可以作为一个输入流、又可以作为一个输出流
     */
    @Test
    public void test1() {
        RandomAccessFile raf1 = null;
        RandomAccessFile raf2 = null;
        try {
            raf1 = new RandomAccessFile(new File("图片.jpg"), "r");
            raf2 = new RandomAccessFile(new File("图片1.jpg"), "rw");

            byte[] buffer = new byte[1024];
            int len;
            while((len = raf1.read(buffer)) != -1) {
                raf2.write(buffer, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
```

```
            if (raf1 != null) {
                try {
                    raf1.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (raf2 != null) {
                try {
                    raf2.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    /*
    如果RandomAccessFile作为输出流时，写出到的文件如果不存在，则在执行过程中自动创建
    如果写出到的文件存在，则会对原有文件内容进行覆盖。（默认情况下，从头覆盖）
     */
    @Test
    public void test2() {
        RandomAccessFile raf1 = null;
        try {
            raf1 = new RandomAccessFile(new File("hello.txt"), "rw");

            raf1.seek(3);// 将指针调到角标为3的位置
            raf1.writeBytes("xyzy");// 还是覆盖操作，不是插入
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (raf1 != null) {
                try {
                    raf1.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    /*
    使用RandomAccessFile实现数据的插入效果
     */
    @Test
    public void test3() {
        RandomAccessFile raf1 = null;
```

```
        try {
            raf1 = new RandomAccessFile(new File("hello.txt"),"rw");

            raf1.seek(3);// 将指针调到角标为3的位置

            // 保存指针3后面的所有数据到StringBuilder中
            StringBuilder builder = new StringBuilder((int) new
File("hello.txt").length());
            byte[] buffer = new byte[20];
            int len;
            while ((len = raf1.read(buffer)) != -1) {
                builder.append(new String(buffer, 0, len));
            }

            // 调回指针，写入"xyz"
            raf1.seek(3);
            raf1.writeBytes("xyz");

            // 将StringBuilder中的数据写入到文件中
            raf1.writeBytes(builder.toString());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (raf1 != null) {
                try {
                    raf1.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

网络编程

概要

```
package com.test.java1;

import java.net.InetAddress;
import java.net.UnknownHostException;

/**
 * 一、网络编程中有两个主要的问题：
 * 1、如何准确的定位网络上一台或多台主机；定位主机上的特定的应用
 * 2、找到主机后如何可靠高效的进行数据传输。
 *
```

```
 * 二、网络编程中的两个要素：
 * 1、对应问题一：IP和端口号
 * 2、对应问题二：提供网络通信协议：TCP/IP参考模型（应用层、传输层、网络层、物理+数据链路层）
 *
 * 三、通信要素一：IP和端口号
 * 1、IP：唯一的标识 Internet 上的计算机（通信实体）
 * 2、在Java中使用InetAddress类代表IP
 * 3、IP分类：IPv4 和 IPv6；  万维网 和 局域网
 * 4、域名：www.baidu.com www.jd.com
 * 5、本地回路地址：127.0.0.1 对应着：localhost
 * 6、如何实例化InetAddress：两个方法：getByName(String host) 、 getLocalhost()
 *    两个常用方法：getHostName() / getHostAddress()
 *
 * 7、端口号：正在计算机上运行的进程。
 * 要求：不同的进程有不同的端口号
 * 范围：被规定为一个16位的整数 0 - 65535。
 *
 * 8、端口号与IP地址的组合得出一个网络套接字：Socket
 * @author: my.seaTide
 * @create: 2021/5/12 10:53 下午
 */
public class InetAddressTest {

    public static void main(String[] args) {
        try {
            InetAddress address = InetAddress.getByName("www.baidu.com");
            System.out.println(address);

            InetAddress address1 = InetAddress.getByName("127.0.0.1");
            System.out.println(address1);

            // 获取本地IP
            InetAddress address2 = InetAddress.getLocalHost();
            System.out.println(address2);

            // getHostName(): 获取域名
            System.out.println(address.getHostName());

            // getHostAddress(): 获取IP地址
            System.out.println(address.getHostAddress());

        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

TCP网络编程举例1

```
package com.test.java1;

import org.junit.Test;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 实现TCP的网络编程
 * 例子1：客户端发送信息给服务端，服务端将数据显示在控制台上
 *
 * @author: my.seaTide
 * @create: 2021/5/16 9:07 下午
 */
public class TCPTest1 {

    // 客户端
    @Test
    public void client() {
        Socket socket = null;
        OutputStream os = null;
        try {
            // 1、创建Socket对象，指明服务器端的ip和端口号
            InetAddress inet = InetAddress.getByName("127.0.0.1");
            socket = new Socket(inet, 8899);

            // 2、获取一个输出流，用于输出数据
            os = socket.getOutputStream();

            // 3、写出数据的操作
            os.write("你好，我是客户端".getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {

            // 4、资源的关闭
            if (os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
```

```
            }

            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    // 服务端
    @Test
    public void server() {
        ServerSocket ss = null;
        Socket socket = null;
        InputStream is = null;
        ByteArrayOutputStream baos = null;
        try {
            // 1、创建服务器端的ServerSocket，指明自己的端口号
            ss = new ServerSocket(8899);
            // 2、调用accept()，表示可以接收来自于客户端的socket
            socket = ss.accept();
            // 3、获取输入流
            is = socket.getInputStream();

            // 4、读取输入流中的数据
            baos = new ByteArrayOutputStream();
            byte[] buffer = new byte[5];
            int len;
            while ((len = is.read(buffer)) != -1) {
                baos.write(buffer, 0, len);
            }

            System.out.println(baos.toString());
            System.out.println("收到来自于：" +
socket.getInetAddress().getHostAddress() + "的请求");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {

            // 5、资源的关闭
            if (baos != null) {
                try {
                    baos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
```

```
            }

            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (ss != null) {
                try {
                    ss.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

TCP网络编程举例2

```
package com.test.java;

import org.junit.Test;

import java.io.*;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 实现TCP的网络编程
 *
 * 例题2：客户端发送文件给服务端，服务端将文件保存在本地。
 *
 */
public class TCPTest2 {
```

```
    // 客户端
    @Test
    public void client() {
        OutputStream os = null;
        FileInputStream fos = null;
        try {
            Socket socket = new Socket(InetAddress.getByName("127.0.0.1"),
9090);
            os = socket.getOutputStream();

            fos = new FileInputStream(new File("图片.jpg"));

            byte[] buffer = new byte[1024];
            int len;
            while ((len = fos.read(buffer)) != -1) {
                os.write(buffer, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (os != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    // 服务端
    @Test
    public void server() {
        ServerSocket ss = null;
        Socket socket = null;
        InputStream is = null;
        FileOutputStream fos = null;
        try {
            ss = new ServerSocket(9090);
            socket = ss.accept();
```

```
            is = socket.getInputStream();

            fos = new FileOutputStream(new File("图片1.jpg"));

            byte[] buffer = new byte[1024];
            int len;
            while ((len = is.read(buffer)) != -1) {
                fos.write(buffer, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (ss != null) {
                try {
                    ss.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

TCP网络编程举例3

```
package com.test.java;

import org.junit.Test;

import java.io.*;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 实现TCP的网络编程
 *
 * 从客户端发送文件给服务端，服务端保存到本地。并返回“发送成功”给客户端。并关闭相应的连接。
 */
public class TCPTest3 {
    // 客户端
    @Test
    public void client() {
        OutputStream os = null;
        FileInputStream fos = null;
        InputStream is = null;
        try {
            Socket socket = new Socket(InetAddress.getByName("127.0.0.1"),
9090);
            os = socket.getOutputStream();

            fos = new FileInputStream(new File("图片.jpg"));

            byte[] buffer = new byte[1024];
            int len;
            while ((len = fos.read(buffer)) != -1) {
                os.write(buffer, 0, len);
            }

            // 关闭数据的输出
            socket.shutdownOutput();

            // 接收来自于服务端的数据，并输入到控制台上
            is = socket.getInputStream();
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            byte[] buffer1 = new byte[20];
            int len1;
            while ((len1 = is.read(buffer1)) != -1) {
                baos.write(buffer1,0, len1);
            }

            System.out.println(baos.toString());
```

```
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (os != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    // 服务端
    @Test
    public void server() {
        ServerSocket ss = null;
        Socket socket = null;
        InputStream is = null;
        FileOutputStream fos = null;
        OutputStream os = null;
        try {
            ss = new ServerSocket(9090);
            socket = ss.accept();

            is = socket.getInputStream();

            fos = new FileOutputStream(new File("图片2.jpg"));

            byte[] buffer = new byte[1024];
            int len;
            while ((len = is.read(buffer)) != -1) {
```

```
                fos.write(buffer, 0, len);
            }

            // 服务器端给与客户端反馈
            os = socket.getOutputStream();
            os.write("图片已收到".getBytes());

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (ss != null) {
                try {
                    ss.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
```

```
        }
    }
}
```

UDP协议的网络编程

```
package com.test.java;

import org.junit.Test;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

/**
 * UDP协议的网络编程
 */
public class UDPTest {

    // 发送端
    @Test
    public void send() {
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();

            byte[] data = "我是UDP方式发送的数据".getBytes();
            InetAddress inet = InetAddress.getLocalHost();
            DatagramPacket packet = new DatagramPacket(data, 0, data.length,
inet, 9090);

            socket.send(packet);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                socket.close();
            }
        }
    }

    // 接收端
    @Test
    public void receiver() throws IOException {
        DatagramSocket socket = new DatagramSocket(9090);

        byte[] buffer = new byte[100];
```

```
        DatagramPacket packet = new DatagramPacket(buffer, 0, buffer.length);

        socket.receive(packet);

        System.out.println(new String(packet.getData(), 0,
packet.getLength()));

        socket.close();
    }
}
```

反射

反射的举例以及Class类的理解

```
package com.test.java;

import org.junit.Test;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

/**
 * @author: my.seaTide
 * @create: 2021/5/23 1:49 下午
 */
public class ReflectionTest {

    // 反射之前，对于Person的操作
    @Test
    public void test1() {

        // 1.创建Person类的对象
        Person p1 = new Person("Tom", 12);

        // 2.通过对象，调用其内部的属性、方法
        p1.age = 10;
        System.out.println(p1.toString());

        p1.show();

        // 在Person类外部，不可以通过Person类的对象调用其内部私有结构。
    }

    // 反射之后，对于Person的操作
    @Test
    public void test2() throws Exception{
```

```
        Class clazz = Person.class;

        // 1.通过反射，创建Person类的对象
        Constructor cons = clazz.getConstructor(String.class, int.class);
        Object obj = cons.newInstance("Tom", 12);
        Person p = (Person) obj;
        System.out.println(p.toString());

        // 2、通过反射，调用对象指定的属性、方法
        // 调用属性
        Field age = clazz.getDeclaredField("age");
        age.set(p, 10);
        System.out.println(p.toString());

        // 调用方法
        Method show = clazz.getDeclaredMethod("show");
        show.invoke(p);

        // 通过反射，可以调用Person类的私有结构的。比如私有构造器、方法、属性
        // 调用私有的构造器
        Constructor cons1 = clazz.getDeclaredConstructor(String.class);
        cons1.setAccessible(true);
        Person p1 = (Person) cons1.newInstance("Jerry");
        System.out.println(p1);

        // 调用私有属性
        Field name = clazz.getDeclaredField("name");
        name.setAccessible(true);
        name.set(p1, "Jack");
        System.out.println(p1);

        // 调用私有方法
        Method showNation = clazz.getDeclaredMethod("showNation",
String.class);
        showNation.setAccessible(true);
        String invoke = (String) showNation.invoke(p1, "中国");// 相当于
p1.showNation("中国")
        System.out.println(invoke);
    }


    // 疑问：通过直接new的方式或反射的方式都可以调用公共的结构，开发中到底用哪个?
    // 建议：直接new的方式。
    // 什么时候会使用：反射的方式。反射的特征：动态性

    /*
    疑问：反射机制与面向对象中的封装性是不是矛盾的？如何看待两个技术？
    不矛盾。
     */
```

```
    /*
     * 关于java.lang.Class类的理解
     * 1、类的加载过程：
     *     程序经过javac.exe命令以后，会生成一个或多个字节码文件（.class结尾），接着我们使
用
     *     java.exe命令对某个字节码文件进行解释运行。相当于将某个字节码文件加载到内存中。此
过程
     *     就称为类的加载。加载到内存中的类，我们就称为运行时类，此运行时类，就作为Class的一
个实例。
     * 2、换句话说，class的实例就对应着一个运行时类。
     *
     * 3、加载到内存中的运行时类，会缓存一定的时间。在此时间之内，我们可以通过不同的方式来获
取此运行时类。
     *
     *
     */

    // 获取Class的实例的方式(前三种方式需要掌握);方式三使用频率最高
    @Test
    public void test3() throws ClassNotFoundException {
        // 方式一：调用运行时类的属性：.class
        Class clazz1 = Person.class;
        System.out.println(clazz1);

        // 方式二：通过运行类的对象：调用getClass()
        Person person = new Person();
        Class clazz2 =  person.getClass();
        System.out.println(clazz2);

        // 方式三：调用Class的静态方法：forName(String classPath)
        Class clazz3 = Class.forName("com.test.java.Person");
        System.out.println(clazz3);

        System.out.println(clazz1 == clazz2);// true
        System.out.println(clazz1 == clazz3);// true

        // 方式四：使用类的加载器：ClassLoader
        ClassLoader classLoader = ReflectionTest.class.getClassLoader();
        Class clazz4 = classLoader.loadClass("com.test.java.Person");
        System.out.println(clazz4);

        System.out.println(clazz1 == clazz4);// true
    }
}
```

类的加载与ClassLoader的理解

- 了解类的加载器

```
@Test
public void test1() {
    // 对于自定义类，使用系统类加载器进行加载
    ClassLoader classLoader1 = ClassLoaderTest.class.getClassLoader();
    System.out.println(classLoader1);

    // 调用系统类加载器的getParent()：获取扩展类加载器
    ClassLoader classLoader2 = classLoader1.getParent();
    System.out.println(classLoader2);

    // 调用扩展类加载器的getParent()：无法获取引导类加载器
    // 引导类加载器主要负责加载java的核心类库，无法加载自定义类的
    ClassLoader classLoader3 = classLoader2.getParent();
    System.out.println(classLoader3);
}
```

- 使用ClassLoader加载配置文件

```
 /*
properties：用来读取配置文件
 */
@Test
public void test2() throws Exception {
    Properties pros = new Properties();
    // 此时的文件默认在当前的module下。
    // 读取配置文件的方式一
//        FileInputStream fis = new FileInputStream("src\\jdbc1.properties");
//        pros.load(fis);

    // 读取配置文件的方式二：使用ClassLoader
    // 配置文件默认识别为：当前module的src下
    ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
    InputStream is = classLoader.getResourceAsStream("jdbc1.properties");
    pros.load(is);

    String user = pros.getProperty("user");
    String password = pros.getProperty("password");
    System.out.println("user = " + user + ",password = " + password);
}
```

创建运行时类的对象

- 举例

```
@Test
public void test1() throws Exception{

    Class<Person> clazz = Person.class;

    /*
    newInstance()：调用此方法，创建对应的运行时类的对象。内部调用了运行时类的空参的构造器
    要想此方法正常的创建运行时类的对象，要求：
    1、运行时类必须提供空参的构造器
    2、空参的构造器的访问权限得够。通常，设置为public

    在javabean中要求提供一个public的空参构造器。原因：
    1、便于通过反射，创建运行时类的对象
    2、便于子类继承此运行时类时，默认调用super()时，保证父类有此构造器
     */

    // jdk 8的写法
//        Person person = clazz.newInstance();
    // jdk 8 以上的写法
    Person person = clazz.getDeclaredConstructor().newInstance() ;
    System.out.println(person);
}
```

- 举例体会反射的动态性

```
// 举例体会反射的动态性
@Test
public void test2() throws Exception {
    for (int i = 0; i < 100; i++) {
        int num = new Random().nextInt(3);
        String classPath = "";
        switch (num) {
            case 0:
                classPath = "java.util.Date";
                break;
            case 1:
                classPath = "java.lang.Object";
                break;
            case 2:
                classPath = "com.test.java.Person";
                break;
        }
        Object obj = getInstance(classPath);
        System.out.println(obj);
```

```
    }
}

/*
创建一个指定类的对象
classPath：指定类的全类名
 */
public Object getInstance(String classPath) throws Exception {
    Class aClass = Class.forName(classPath);
    return aClass.getDeclaredConstructor().newInstance();
}
```

获取运行时类的完整结构

- 获取运行时类的属性结构及其内部结构

```
@Test
public void test1() {
    Class clazz = Person.class;

    // 获取属性结构
    // getFields()：获取当前运行时类及其父类中声明为public访问权限的属性
    Field[] fields = clazz.getFields();
    for (Field f : fields) {
        System.out.println(f);
    }
    System.out.println();

    // getDeclaredFields()：获取当前运行时类中声明的所有属性。（不包含父类中声明的属性）
    Field[] fields1 = clazz.getDeclaredFields();
    for (Field f : fields1) {
        System.out.println(f);
    }
    System.out.println();
}

// 权限修饰符 数据类型 变量名
@Test
public void test2() {
    Class clazz = Person.class;
    Field[] fields1 = clazz.getDeclaredFields();
    for (Field f : fields1) {
        // 获取权限修饰符
        int i = f.getModifiers();
        System.out.print(Modifier.toString(i) + "\t");

        // 获取数据类型
        Class type = f.getType();
        System.out.print(type.getName() + "\t");
```

```
        // 获取变量名
        String fName = f.getName();
        System.out.print(fName);

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

- 获取运行时类的方法结构及其内部结构

```
@Test
public void test1() {
    Class clazz = Person.class;

    // getMethods()：获取当前运行时类及其所有父类中声明为public权限的方法
    Method[] methods = clazz.getMethods();
    for (Method m : methods) {
        System.out.println(m);
    }
    System.out.println();

    //getDeclaredMethods()：获取当前运行时类中声明的所有方法。（不包含父类中声明的方法）
    Method[] methods1 = clazz.getDeclaredMethods();
    for (Method m : methods1) {
        System.out.println(m);
    }
}

/*
@XxxAnnotation
权限修饰符 返回值类型  方法名（参数类型1 形参名1,...）throws XxxException{}
 */
@Test
public void test2() {
    Class clazz = Person.class;
    Method[] methods = clazz.getDeclaredMethods();
    for (Method m : methods) {

        // 1、获取方法声明的注解
        Annotation[] annotations = m.getAnnotations();
        for (Annotation a : annotations) {
            System.out.println(a);
        }

        // 2、权限修饰符
        System.out.print(Modifier.toString(m.getModifiers()) + "\t");
```

```
            // 3、返回值类型
            System.out.print(m.getReturnType().getName() + "\t");

            // 4、方法名
            System.out.print(m.getName());
            System.out.print("(");
            //5、形参列表
            Class[] types = m.getParameterTypes();
            if (!(types == null && types.length == 0)) {
                for (int i = 0; i < types.length; i++) {
                    if (i == types.length - 1) {
                        System.out.print(types[i].getName() + " args_" + i);
                        break;
                    }
                    System.out.print(types[i].getName() + " args_" + i + ",");
                }
            }
            System.out.print(")");

            // 6、抛出的异常
            Class[] exceptionTypes = m.getExceptionTypes();
            if (exceptionTypes.length > 0) {
                System.out.print("throws ");
                for (int i = 0; i < exceptionTypes.length; i++) {
                    if (i == exceptionTypes.length - 1) {
                        System.out.print(exceptionTypes[i].getName());
                        break;
                    }
                    System.out.print(exceptionTypes[i].getName() + ",");
                }
            }

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

- 获取运行时类的构造器结构

```
@Test
public void test1() {
  Class<Person> clazz = Person.class;

  // getConstructors()：获取当前运行时类中声明为public的构造器
  Constructor[] cons = clazz.getConstructors();
  for (Constructor c : cons) {
    System.out.println(c);
  }
```

```
  // getDeclaredConstructors()：获取当前运行时类中声明的所有构造器
  Constructor[] constructors = clazz.getDeclaredConstructors();
  for (Constructor c : constructors) {
    System.out.println(c);
  }
}
```

- 获取运行时类的父类以及父类的泛型

```
// 获取运行时类的父类
@Test
public void test2() {
  Class clazz = Person.class;
  Class superclass = clazz.getSuperclass();
  System.out.println(superclass);
}

// 获取运行时类的带泛型的父类
@Test
public void test3() {
  Class clazz = Person.class;
  Type genericSuperclass = clazz.getGenericSuperclass();
  System.out.println(genericSuperclass);
}

// 获取运行时类的带泛型的父类的泛型
@Test
public void test4() {
  Class clazz = Person.class;
  Type genericSuperclass = clazz.getGenericSuperclass();

  ParameterizedType paramType = (ParameterizedType) genericSuperclass;
  Type[] actualTypeArguments = paramType.getActualTypeArguments();

  System.out.println(((Class) actualTypeArguments[0]).getName());
}
```

- 获取运行时类的实现的接口、包、注解

```
 // 获取运行时类实现的接口
@Test
public void test5() {
  Class clazz = Person.class;
  Class[] interfaces = clazz.getInterfaces();
  for (Class c : interfaces) {
    System.out.println(c);
  }
```

```
    System.out.println();

    // 获取运行时类的父类实现的接口
    Class[] interfaces1 = clazz.getSuperclass().getInterfaces();
    for (Class c : interfaces1) {
      System.out.println(c);
    }
}

// 获取运行时类所在的包
@Test
public void test6() {
    Class clazz = Person.class;
    Package pack = clazz.getPackage();
    System.out.println(pack);
}

// 获取运行时类声明的注解
@Test
public void test7() {
    Class clazz = Person.class;
    Annotation[] annotations = clazz.getAnnotations();
    for (Annotation anno : annotations) {
      System.out.println(anno);
    }
}
```

调用运行时类的指定结构

- 调用运行时类中的指定属性

```
// 如何操作运行时类中的指定属性 -- 需要掌握
@Test
public void testFiled1() throws Exception {
    Class<Person> clazz = Person.class;

    // 创建运行时类的对象
    Person instance = clazz.getDeclaredConstructor().newInstance();

    // getDeclaredField(String FieldName)：获取运行时类中指定变量名的属性
    Field name = clazz.getDeclaredField("name");

    // 保证当前属性是可访问的
    name.setAccessible(true);

    // 获取、设置指定对象的此属性值
    name.set(instance, "Tom");

    System.out.println(name.get(instance));
```

```
}
```

- 调用运行时类中的指定方法 -- 需要掌握

```
// 调用运行时类中的指定方法 -- 需要掌握
@Test
public void testMethod() throws Exception {
    Class<Person> clazz = Person.class;
    // 创建运行时类的对象
    Person p = clazz.getDeclaredConstructor().newInstance();

    /*
    1、获取指定的某个方法
    getDeclaredMethod()：参数一：指明获取的方法的名称 参数2：指明获取的方法的形参列表
     */
    Method show = clazz.getDeclaredMethod("show", String.class);

    // 2、保证当前方法是可访问的
    show.setAccessible(true);

    /*
    3、调用方法的invoke()：参数一：方法的调用者 参数二：给方法形参赋值的实参
    invoke的返回值即为对应类中调用的方法的返回值
     */
    Object returnValue = show.invoke(p, "CHN");// String value = p.show("CHN");
    System.out.println(returnValue);

    System.out.println("******************如何调用静态方法*******************");

    // private static void showDesc()
    Method showDesc = clazz.getDeclaredMethod("showDesc");
    showDesc.setAccessible(true);
    // 如果调用的运行时类中的方法没有返回值，则此invoke返回null
//        Object invoke = showDesc.invoke(Person.class);// 写法1
    Object invoke = showDesc.invoke(null);// 写法2
    System.out.println(invoke); // null
}
```

- 调用运行时类中的指定构造器

```
// 调用运行时类中的指定构造器
@Test
public void testConstructor() throws Exception {
    Class<Person> clazz = Person.class;

    /*
    1、获取指定的构造器
    getDeclaredConstructor()：参数：指明构造器的参数列表
```

```
     */
    Constructor<Person> constructor =
clazz.getDeclaredConstructor(String.class);

    // 2、保证此构造器是可访问的
    constructor.setAccessible(true);

    // 3、调用此构造器创建运行时类的对象
    Person person = constructor.newInstance("Tom");
    System.out.println(person);
}
```

反射的应用 - 动态代理

- 静态代理举例

```
package com.test.java;

/**
 * 静态代理举例
 * 特点：代理类和被代理类在编译期间，就被确定下来了
 */

interface ClothFactory {
    void produceCloth();
}

// 代理类
class ProxyClothFactory implements ClothFactory{
    private ClothFactory factory;

    public ProxyClothFactory(ClothFactory factory) {
        this.factory = factory;
    }

    @Override
    public void produceCloth() {
        System.out.println("代理模式开始");

        factory.produceCloth();

        System.out.println("代理模式结束");
    }
}

// 被代理类
class NikeClothFactory implements ClothFactory{

    @Override
```

```
    public void produceCloth() {
        System.out.println("这里是Nike生产的衣服");
    }
}

public class StaticProxyTest {
    public static void main(String[] args) {
        // 创建被代理类的对象
        NikeClothFactory nikeClothFactory = new NikeClothFactory();
        // 创建代理类的对象
        ProxyClothFactory proxyClothFactory = new
ProxyClothFactory(nikeClothFactory);
        proxyClothFactory.produceCloth();
    }
}
```

- 动态代理的举例

```
package com.test.java;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * 动态代理的举例
 */
interface Human {
    String getBelief();
    void eat(String food);
}

// 被代理类
class SuperMan implements Human {
    @Override
    public String getBelief() {
        return "I like Java";
    }

    @Override
    public void eat(String food) {
        System.out.println("我喜欢吃" + food);
    }
}

/*
要想实现动态代理，需要解决的问题?
问题一：如何根据加载到内存中的被代理类，动态的创建一个代理类及其对象
```

```
问题二：当通过代理类的对象调用方法a时，如何动态的去调用被代理类中的同名方法a
 */
class ProxyFactory {
    // 调用此方法，返回一个代理类的对象。解决问题一
    public static Object getProxyInstance(Object obj) { // obj:被代理类的对象
        MyInvocationHandler handler = new MyInvocationHandler();

        handler.bind(obj);

        return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
obj.getClass().getInterfaces(), handler);
    }
}

class MyInvocationHandler implements InvocationHandler {
    private Object obj; // 需要使用被代理类的对象进行赋值

    public void bind(Object obj) {
        this.obj = obj;
    }

    // 当我们通过代理类的对象调用方法a时，就会自动的调用如下的方法：invoke()
    // 将被代理类要执行的方法a的功能就声明在invoke()中
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

        // Method:即为代理类对象调用的方法，此方法也就作为了被代理类对象要调用的方法
        // obj:被代理类的对象
         Object returnValue = method.invoke(obj, args);

         // 上述方法的返回值就作为当前类中的invoke()的返回值。
         return returnValue;
    }
}

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

        SuperMan superMan = new SuperMan();

        // proxyInstance：代理类的对象
        Human proxyInstance = (Human) ProxyFactory.getProxyInstance(superMan);

        // 当通过代理类对象调用方法时，会自动的调用被代理类中同名的方法
        String belief = proxyInstance.getBelief();
        System.out.println(belief);

        proxyInstance.eat("米饭");
```

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

        NikeClothFactory clothFactory = new NikeClothFactory();
        ClothFactory proxyInstance1 = (ClothFactory)
ProxyFactory.getProxyInstance(clothFactory);
        proxyInstance1.produceCloth();
    }

}
```

- 动态代理与AOP(面向切面编程)

```
package com.test.java;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * 动态代理的举例
 */
interface Human {
    String getBelief();

    void eat(String food);
}

// 工具类
class HumanUtils {
    public void method1() {
        System.out.println("**************通用方法一**************");
    }

    public void method2() {
        System.out.println("**************通用方法二**************");
    }
}

// 被代理类
class SuperMan implements Human {
    @Override
    public String getBelief() {
        return "I like Java";
    }

    @Override
    public void eat(String food) {
```

```
        System.out.println("我喜欢吃" + food);
    }
}

/*
要想实现动态代理，需要解决的问题?
问题一：如何根据加载到内存中的被代理类，动态的创建一个代理类及其对象
问题二：当通过代理类的对象调用方法a时，如何动态的去调用被代理类中的同名方法a
 */
class ProxyFactory {
    // 调用此方法，返回一个代理类的对象。解决问题一
    public static Object getProxyInstance(Object obj) { // obj:被代理类的对象
        MyInvocationHandler handler = new MyInvocationHandler();

        handler.bind(obj);

        return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
obj.getClass().getInterfaces(), handler);
    }
}

class MyInvocationHandler implements InvocationHandler {
    private Object obj; // 需要使用被代理类的对象进行赋值

    public void bind(Object obj) {
        this.obj = obj;
    }

    // 当我们通过代理类的对象调用方法a时，就会自动的调用如下的方法：invoke()
    // 将被代理类要执行的方法a的功能就声明在invoke()中
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

        HumanUtils utils = new HumanUtils();
        utils.method1();

        // Method:即为代理类对象调用的方法，此方法也就作为了被代理类对象要调用的方法
        // obj:被代理类的对象
        Object returnValue = method.invoke(obj, args);

        utils.method2();
        // 上述方法的返回值就作为当前类中的invoke()的返回值。
        return returnValue;
    }
}

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

```
        SuperMan superMan = new SuperMan();

        // proxyInstance：代理类的对象
        Human proxyInstance = (Human) ProxyFactory.getProxyInstance(superMan);

        // 当通过代理类对象调用方法时，会自动的调用被代理类中同名的方法
        String belief = proxyInstance.getBelief();
        System.out.println(belief);

        proxyInstance.eat("米饭");

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

        NikeClothFactory clothFactory = new NikeClothFactory();
        ClothFactory proxyInstance1 = (ClothFactory)
ProxyFactory.getProxyInstance(clothFactory);
        proxyInstance1.produceCloth();
    }
}
```

Java8新特性

Lambda表达式

- Lambda表达式的使用举例

```
@Test
public void test1() {
    Runnable r1 = new Runnable() {
        @Override
        public void run() {
            System.out.println("我爱Java");
        }
    };
    r1.run();

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

    // Lambda表达式的写法
    Runnable r2 = () -> System.out.println("我爱PHP");
    r2.run();
}

@Test
public void test2() {
    Comparator<Integer> com1 = new Comparator<>() {
        @Override
```

```
        public int compare(Integer o1, Integer o2) {
            return Integer.compare(o1, o2);
        }
    };

    int compare = com1.compare(21, 12);
    System.out.println(compare);

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

    // Lambda表达式的写法
    Comparator<Integer> com2 = (o1, o2) -> Integer.compare(o1, o2);
    int compare2 = com2.compare(12, 21);
    System.out.println(compare2);

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

    // 方法引用
    Comparator<Integer> com3 = Integer :: compare;
    int compare3 = com3.compare(13, 13);
    System.out.println(compare3);
}
```

- Lambda表达式的使用

```
package com.test.java1;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.function.Consumer;

/**
 * Lambda表达式的使用
 *
 * 1、举例：(o1, o2) -> Integer.compare(o1, o2);
 *
 * 2、格式：
 *      “->” 该操作符被称为 Lambda 操作符 或 箭头操作符
 *      左侧：Lambda形参列表 （其实就是接口中的抽象方法的形参列表）
 *      右侧：Lambda 体，是抽象方法的实现逻辑，也即Lambda 表达式要执行的功能。（其实就是重写的抽象方法的方法体）
 *
 * 3、Lambda表达式的使用：（分6种情况介绍）
 *      总结：
 *      ->左边：Lambda形参列表的参数类型可以省略(类型推断)；如果Lambda形参列表只有一个参数，其一对()也可以省略
```

```
 *      ->右边：Lambda体应该使用一对{}包裹；如果Lambda体只有一条执行语句(可能是return语句)，可以省略这一对{}和return关键字
 *
 * 4、Lambda表达式的本质：作为函数式接口的实例
 */
public class LambdaTest1 {

    // 语法格式一：无参，无返回值
    @Test
    public void test1() {
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("我爱Java");
            }
        };
        r1.run();

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

        // Lambda表达式的写法
        Runnable r2 = () -> {
            System.out.println("我爱PHP");
        };
        r2.run();
    }

    // 语法格式二：Lambda 需要一个参数，但是没有返回值。
    @Test
    public void test2() {
        Consumer<String> consumer = new Consumer<>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        };
        consumer.accept("我爱北京天安门");

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

        Consumer<String> consumer1 = (String s) -> {
            System.out.println(s);
        };
        consumer1.accept("我爱打游戏");
    }

    // 语法格式三：数据类型可以省略，因为可由编译器推断得出，称为“类型推断”
    @Test
    public void test3() {
```

```
        Consumer<String> consumer1 = (String s) -> {
            System.out.println(s);
        };
        consumer1.accept("abcdefg");

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

        Consumer<String> consumer2 = (s) -> {
            System.out.println(s);
        };
        consumer2.accept("abcdefg123");

        // 其他类型推断举例
        ArrayList<String> list = new ArrayList<>(); // 类型推断
        int[] arr = {1,2,3};// 类型推断
    }

    // 语法格式四：Lambda 若只需要一个参数时，参数的小括号可以省略
    @Test
    public void test4() {
        Consumer<String> consumer1 = (s) -> {
            System.out.println(s);
        };
        consumer1.accept("abcdefg");

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

        Consumer<String> consumer2 = s -> {
            System.out.println(s);
        };
        consumer2.accept("abcdefg123");
    }

    // 语法格式五：Lambda 需要两个或以上的参数，多条执行语句，并且可以有返回值
    @Test
    public void test5() {
        Comparator<Integer> comparator = new Comparator<>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                System.out.println(o1);
                System.out.println(o2);
                return o1.compareTo(o2);
            }
        };
        System.out.println(comparator.compare(12, 21));

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

        Comparator<Integer> comparator1 = (o1, o2) -> {
```

```
            System.out.println(o1);
            System.out.println(o2);
            return o1.compareTo(o2);
        };
        System.out.println(comparator1.compare(21, 12));

    }

    // 语法格式六：当 Lambda 体只有一条语句时，return 与大括号若有，都可以省略
    @Test
    public void test6() {
        Comparator<Integer> comparator1 = (o1, o2) -> {
            return o1.compareTo(o2);
        };
        System.out.println(comparator1.compare(21, 12));

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

        Comparator<Integer> comparator2 = (o1, o2) -> o1.compareTo(o2);
        System.out.println(comparator2.compare(12, 21));
    }
}
```

函数式(Functional)接口

- 定义

> 只包含一个抽象方法的接口，称为函数式接口。
>
> 我们可以在一个接口上使用 @FunctionalInterface 注解，这样做可以检
>
> 查它是否是一个函数式接口。

> 所以以前用匿名实现类表示的现在都可以用Lambda表达式来写。

- Java内置四大核心函数式接口介绍及使用举例

```
package com.test.java1;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

/**
 * Java内置的4大核心函数式接口
```

```
 *
 * 消费型接口：  Consumer<T>   void accept(T t)
 * 供给型接口：  Supplier<T>   T get()
 * 函数型接口：  Function<T,R> R apply(T t)
 * 断定型接口：  Predicate<T>  boolean test(T t)
 *
 */
public class LambdaTest2 {

    // Consumer举例
    @Test
    public void test1() {
        happyTime(500, new Consumer<Double>() {
            @Override
            public void accept(Double aDouble) {
                System.out.println("买了一本书，价格为：" + aDouble);
            }
        });

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

        // Lambda表达式
        happyTime(1000, money -> System.out.println("买个电脑，价格为：" +
money));

    }

    public void happyTime(double money, Consumer<Double> con) {
        con.accept(money);
    }

    @Test
    public void test2() {
        List<String> list = Arrays.asList("北京", "天津", "南京");
        List<String> stringList = filterString(list, new Predicate<String>() {
            @Override
            public boolean test(String s) {
                return s.contains("京");
            }
        });
        System.out.println(stringList);

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

        // Lambda表达式
        List<String> stringList1 = filterString(list, s -> s.contains("京"));
        System.out.println(stringList1);
    }
```

```
    // 根据给定的规则，过滤集合中的字符串。此规则由Predicate的方法决定
    public List<String> filterString(List<String> list, Predicate<String>
predicate) {
        ArrayList<String> filterList = new ArrayList<>();

        for (String s : list) {
            if (predicate.test(s)) {
                filterList.add(s);
            }
        }

        return  filterList;
    }
}
```

方法引用与构造器引用

- Employee类

```
package com.test.java2;

public class Employee {

  private int id;
  private String name;
  private int age;
  private double salary;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
```

```
    this.age = age;
  }

  public double getSalary() {
    return salary;
  }

  public void setSalary(double salary) {
    this.salary = salary;
  }

  public Employee() {

  }

  public Employee(int id) {

    this.id = id;
  }

  public Employee(int id, String name) {
    this.id = id;
    this.name = name;
  }

  public Employee(int id, String name, int age, double salary) {

    this.id = id;
    this.name = name;
    this.age = age;
    this.salary = salary;
  }

  @Override
  public String toString() {
    return "Employee{" + "id=" + id + ", name='" + name + '\'' + ", age=" + age
+ ", salary=" + salary + '}';
  }

  @Override
  public boolean equals(Object o) {
    if (this == o)
      return true;
    if (o == null || getClass() != o.getClass())
      return false;

    Employee employee = (Employee) o;

    if (id != employee.id)
```

```
      return false;
    if (age != employee.age)
      return false;
    if (Double.compare(employee.salary, salary) != 0)
      return false;
    return name != null ? name.equals(employee.name) : employee.name == null;
  }

  @Override
  public int hashCode() {
    int result;
    long temp;
    result = id;
    result = 31 * result + (name != null ? name.hashCode() : 0);
    result = 31 * result + age;
    temp = Double.doubleToLongBits(salary);
    result = 31 * result + (int) (temp ^ (temp >>> 32));
    return result;
  }
}
```

- EmployeeData类

```
package com.test.java2;

import java.util.ArrayList;
import java.util.List;
/**
 * 提供用于测试的数据
 *
 * @author shkstart 邮箱：shkstart@126.com
 *
 */
public class EmployeeData {

  public static List<Employee> getEmployees(){
    List<Employee> list = new ArrayList<>();

    list.add(new Employee(1001, "马化腾", 34, 6000.38));
    list.add(new Employee(1002, "马云", 12, 9876.12));
    list.add(new Employee(1003, "刘强东", 33, 3000.82));
    list.add(new Employee(1004, "雷军", 26, 7657.37));
    list.add(new Employee(1005, "李彦宏", 65, 5555.32));
    list.add(new Employee(1006, "比尔盖茨", 42, 9500.43));
    list.add(new Employee(1007, "任正非", 26, 4333.32));
    list.add(new Employee(1008, "扎克伯格", 35, 2500.32));

    return list;
```

```
    }
}
```

- MethodRefTest类：方法引用的使用

```
package com.test.java2;

import org.junit.Test;

import java.io.PrintStream;
import java.util.Comparator;
import java.util.function.BiPredicate;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

/**
 * 方法引用的使用
 *
 * 1、使用情境：当要传递给Lambda体的操作，已经有实现的方法了，可以使用方法引用！
 *
 * 2、方法引用，本质上就是Lambda表达式，而Lambda表达式作为函数式接口的实例，所以，方法引用，也是函数式接口的实例
 *
 * 3、使用格式：(类)对象 :: 方法名
 *
 * 4、具体分为如下的三种情况
 *     情况一：  对象 :: 非静态方法
 *     情况二：  类 :: 静态方法
 *
 *     情况三：  类 :: 非静态方法
 *
 * 5、方法引用使用的要求：要求接口中的抽象方法的形参列表与和返回值类型与方法引用的方法的形参列表和返回值类型相同！（针对于情况一和情况二）
 *
 * Created by shkstart.
 */
public class MethodRefTest {

    // 情况一：对象 :: 实例方法
    //Consumer中的void accept(T t)
    //PrintStream中的void println(T t)
    @Test
    public void test1() {
        Consumer<String> consumer = str -> System.out.println(str);
        consumer.accept("北京");

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

```
    // 方法引用
    PrintStream ps = System.out;
    Consumer<String> consumer1 = ps::println;
    consumer1.accept("beijing");

  }

  //Supplier中的T get()
  //Employee中的String getName()
  @Test
  public void test2() {
    Employee employee = new Employee(10, "TOM", 20, 5000);

    Supplier<String> sup = () -> employee.getName();
    System.out.println(sup.get());

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

    // 方法引用
    Supplier<String> sup1 = employee::getName;
    System.out.println(sup1.get());
  }

  // 情况二：类 :: 静态方法
  //Comparator中的int compare(T t1,T t2)
  //Integer中的int compare(T t1,T t2)
  @Test
  public void test3() {
    Comparator<Integer> com = (o1, o2) -> Integer.compare(o1, o2);
    System.out.println(com.compare(12,21));

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

    Comparator<Integer> com1 = Integer::compare;
    System.out.println(com1.compare(21, 12));
  }

  //Function中的R apply(T t)
  //Math中的Long round(Double d)
  @Test
  public void test4() {
    Function<Double, Long> fun = new Function<>() {
      @Override
      public Long apply(Double d) {
        return Math.round(d);
      }
    };
```

```
    Function<Double, Long> fun1 = d -> Math.round(d);
    System.out.println(fun1.apply(6.3));

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

    Function<Double, Long> fun2 = Math::round;
    System.out.println(fun2.apply(6.6));
  }

  // 情况三：类 :: 实例方法
  // Comparator中的int comapre(T t1,T t2)
  // String中的int t1.compareTo(t2)
  @Test
  public void test5() {
    Comparator<String> com = (t1, t2) -> t1.compareTo(t2);
    System.out.println(com.compare("abc", "abd"));

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

    Comparator<String> com1 = String::compareTo;
    System.out.println(com1.compare("abc", "abd"));
  }

  //BiPredicate中的boolean test(T t1, T t2);
  //String中的boolean t1.equals(t2)
  @Test
  public void test6() {
    BiPredicate<String,String> bi = (s1, s2) -> s1.equals(s2);
    System.out.println(bi.test("abc", "abc"));

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

    BiPredicate<String,String> bi1 = String::equals;
    System.out.println(bi1.test("abc", "abd"));
  }

  // Function中的R apply(T t)
  // Employee中的String getName();
  @Test
  public void test7() {
    Employee employee = new Employee(1001, "Jack", 22, 6000);

    Function<Employee, String> fun1 = e -> e.getName();
    System.out.println(fun1.apply(employee));

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

    Function<Employee, String> fun2 = Employee::getName;
    System.out.println(fun2.apply(employee));
```

```
  }
}
```

- ConstructorRefTest类：构造器引用的使用

```
package com.test.java2;

import org.junit.Test;

import java.util.Arrays;
import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.function.Supplier;

/**
 * 一、构造器引用
 *      和方法引用类似，函数式接口的抽象方法的形参列表和构造器的形参列表一致。
 *      抽象方法的返回值类型即为构造器所属的类的类型
 *
 * 二、数组引用
 *      可以把数组看做是一个特殊的类，则写法与构造器引用一致。
 *
 * Created by shkstart
 */
public class ConstructorRefTest {
  //构造器引用
    //Supplier中的T get()
    // Employee的空参构造器：Employee()
    @Test
    public void test1(){
        Supplier<Employee> sup = new Supplier<Employee>() {
            @Override
            public Employee get() {
                return new Employee();
            }
        };
        System.out.println(sup.get());

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

        Supplier<Employee> sup1 = () -> new Employee();
        System.out.println(sup1.get());

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

        Supplier<Employee> sup2 = Employee::new;
        System.out.println(sup2.get());
  }
```

```
  //Function中的R apply(T t)
    @Test
    public void test2(){
        Function<Integer, Employee> fun = id -> new Employee(id);
        Employee employee = fun.apply(1001);
        System.out.println(employee);

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

        Function<Integer, Employee> fun1 = Employee::new;
        Employee employee1 = fun1.apply(1002);
        System.out.println(employee1);
    }

  //BiFunction中的R apply(T t,U u)
    @Test
    public void test3(){
        BiFunction<Integer, String, Employee> fun1 = (id, name) -> new
Employee(id, name);
        System.out.println(fun1.apply(1001, "TOM"));

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

        BiFunction<Integer, String, Employee> fun2 = Employee::new;
        System.out.println(fun2.apply(1002, "Jack"));
  }

  //数组引用
    //Function中的R apply(T t)
    @Test
    public void test4(){
        Function<Integer, String[]> fun  = length -> new String[length];
        String[] apply = fun.apply(5);
        System.out.println(Arrays.toString(apply));

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

        Function<Integer, String[]> fun1 = String[]::new;
        String[] apply1 = fun1.apply(10);
        System.out.println(Arrays.toString(apply1));
    }
}
```

Stream API

概述

1. Stream关注的是对数据的运算，与CPU打交道；集合关注的是数据的存储，与内存打交道
2. > Stream 自己不会存储元素
   >
   > Stream 不会改变源对象，相反，它们会返回一个持有结果的新Stream。
   >
   > Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行
3. Stream 执行流程

   > Stream的实例化
   >
   > 一系列的中间操作（过滤、映射、...）
   >
   > 终止操作
4. 说明：

   > 一个中间操作链，对数据源的数据进行处理
   >
   > 一旦执行终止操作，就执行中间操作链，并产生结果。之后，不会再被使用

Stream的实例化

```
 // 创建 Stream 方式一：通过集合
@Test
public void test1() {
  List<Employee> employees = EmployeeData.getEmployees();

  // default Stream<E> stream()：返回一个顺序流
  Stream<Employee> stream = employees.stream();

  // default Stream<E> parallelStream()：返回一个并行流
  Stream<Employee> employeeStream = employees.parallelStream();
}

// 创建 Stream 方式二：通过数组
@Test
public void test2() {
  int[] arr = new int[]{1,2,3,4,5,6};
  // 调用Arrays类的static <T> Stream<T> stream(T[] array)：返回一个流
  IntStream stream = Arrays.stream(arr);

  Employee e1 = new Employee(1001, "Tom");
  Employee e2 = new Employee(1002, "Jack");
  Employee[] arr1 = new Employee[]{e1, e2};
  Stream<Employee> stream1 = Arrays.stream(arr1);
}

// 创建 Stream 方式三：通过  Stream 的 of()
@Test
```

```
public void test3() {
  Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);
}

// 创建 Stream 方式四：创建无限流
@Test
public void test4() {
  /*
  迭代
  public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
  遍历10个偶数
  */
  Stream.iterate(0, t -> t + 2).limit(10).forEach(System.out::println);

  /*
  生成
  public static<T> Stream<T> generate(Supplier<T> s)
  */
  Stream.generate(Math::random).limit(10).forEach(System.out::println);
}
```

Stream的中间操作

- 筛选与切片

```
@Test
public void test1() {
  List<Employee> list = EmployeeData.getEmployees();
  // filter(Predicate p)---接收Lambda，从流中排除某些元素
  Stream<Employee> stream = list.stream();
  // 练习：查询员工表中薪资大于7000的员工信息
  stream.filter(e -> e.getSalary() > 7000).forEach(System.out::println);

  System.out.println();
  // limit(n)---截断流，使其元素不超过给定数量。
  list.stream().limit(3).forEach(System.out::println);

  System.out.println();
  // skip(n)---跳过元素，返回一个扔掉了前n个元素的流。若流中元素不足n个，则返回一个空流。与limit(n)互补
  list.stream().skip(3).forEach(System.out::println);

  System.out.println();
  // distinct()---筛选，通过流所生成元素的hashCode() 和 equals() 去除重复元素
  list.add(new Employee(1010, "马云", 50, 6000));
  list.add(new Employee(1010, "马云", 51, 6000));
  list.add(new Employee(1010, "马云", 50, 6000));
  list.add(new Employee(1010, "马云", 50, 6000));
```

```
  list.stream().distinct().forEach(System.out::println);
}
```

- 映射

```
@Test
public void test2() {
  // map(Function f)--接收一个函数作为参数，将元素转换成其他形式或提取信息，该函数会被应用到每个元素上，并将其映射称一个新的元素。
  List<String> list = Arrays.asList("aa", "bb", "cc");
  list.stream().map(str -> str.toUpperCase()).forEach(System.out::println);

  // 练习：获取员工姓名长度大于3的员工的姓名
  EmployeeData.getEmployees().stream().map(Employee::getName).filter(name -> name.length() > 3).forEach(System.out::println);

  System.out.println();

  // 练习2：
  Stream<Stream<Character>> streamStream = list.stream().map(StreamAPITest1::fromStringToStream);
  streamStream.forEach(s->{
    s.forEach(System.out::println);
  });

  System.out.println();

  // flatMap(Function f)--接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。
  list.stream().flatMap(StreamAPITest1::fromStringToStream).forEach(System.out::println);


}

// 将字符串中的多个字符构成的集合转换为对应的Stream的实例
public static Stream<Character> fromStringToStream(String str) {
  ArrayList<Character> list = new ArrayList<>();
  for (Character c : str.toCharArray()) {
    list.add(c);
  }
  return list.stream();
}
```

- 排序

```
@Test
public void test3() {
```

```
  // sorted() -- 自然排序
  List<Integer> list = Arrays.asList(8, 4, 5, 7, 13, -5, 0, 6);
  list.stream().sorted().forEach(System.out::println);

  // 抛异常, 原因：Employee类没有实现comparable接口
  //        List<Employee> employees = EmployeeData.getEmployees();
  //        employees.stream().sorted().forEach(System.out::println);

  // sorted(Comparator com) -- 定制排序
  List<Employee> employees = EmployeeData.getEmployees();
  employees.stream().sorted( (e1, e2) -> {
    int i = Integer.compare(e1.getAge(), e2.getAge());
    if (i != 0) {
      return  i;
    } else {
      return Double.compare(e1.getSalary(), e2.getSalary());
    }

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

Stream的终止操作

- 匹配与查找

```
@Test
public void test1() {
    List<Employee> list = EmployeeData.getEmployees();

    /*
     allMatch(Predicate p) 检查是否匹配所有元素
     练习：是否所有的员工的年龄都大于18
     */
    boolean allMatch = list.stream().allMatch(e -> e.getAge() > 18);
    System.out.println(allMatch);

    /*
    anyMatch(Predicate p) 检查是否至少匹配一个元素
    练习：是否存在员工的工资大于10000
     */
    boolean anyMatch = list.stream().anyMatch(e -> e.getSalary() > 10000);
    System.out.println(anyMatch);

    /*
    noneMatch(Predicate p) 检查是否没有匹配所有元素
    练习：是否存在员工姓“雷”
     */
    boolean noneMatch = list.stream().noneMatch(e ->
e.getName().startsWith("雷"));
```

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

    /*
    findFirst() 返回第一个元素
     */
    Optional<Employee> first = list.stream().findFirst();
    System.out.println(first);

    /*
     findAny() 返回当前流中的任意元素
     */
    Optional<Employee> any = list.parallelStream().findAny();
    System.out.println(any);

    /*
     count() 返回流中元素总数
     */
    long count = list.stream().count();
    System.out.println(count);

    /*
     max(Comparator c) 返回流中最大值
     练习：返回最高的工资。
     */
    Optional<Double> max = list.stream().map(e ->
e.getSalary()).max(Double::compare);
    System.out.println(max);

     /*
     min(Comparator c) 返回流中最小值
     练习：返回最低工资的员工。
     */
    Optional<Employee> min = list.stream().min((e1, e2) ->
Double.compare(e1.getSalary(), e2.getSalary()));
    System.out.println(min);

    // forEach(Consumer c) 内部迭代(使用 Collection 接口需要用户去做迭代，称为外部迭
代。相反，Stream API 使用内部迭代——它帮你把迭代做了)
    list.stream().forEach(System.out::println);

    // 使用集合的遍历操作
    list.forEach(System.out::println);
}
```

- 归约

```
@Test
public void test1() {
  //     reduce(T iden, BinaryOperator b) 可以将流中元素反复结合起来，得到一个值。返回
T
  List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
  Integer sum = list.stream().reduce(0, Integer::sum);
  System.out.println(sum);

  //     reduce(BinaryOperator b) 可以将流中元素反复结合起来，得到一个值。返回
Optional<T>
  List<Employee> employees = EmployeeData.getEmployees();
  Stream<Double> doubleStream = employees.stream().map(Employee::getSalary);
  Optional<Double> sumMoney = doubleStream.reduce((d1, d2) -> d1 + d2);
  System.out.println(sumMoney);
}
```

- 收集

```
 @Test
public void test2() {
  // collect(Collector c) 将流转换为其他形式。接收一个 Collector接口的实现，用于给
Stream中元素做汇总的方法
  // 练习：查找工资大于6000的员工，结果返回为一个List或Set

  List<Employee> employees = EmployeeData.getEmployees();
  List<Employee> list = employees.stream().filter(e -> e.getSalary() >
6000).collect(Collectors.toList());
  list.forEach(System.out::println);

  System.out.println();

  Set<Employee> set = employees.stream().filter(e -> e.getSalary() >
6000).collect(Collectors.toSet());
  set.forEach(System.out::println);
}
```