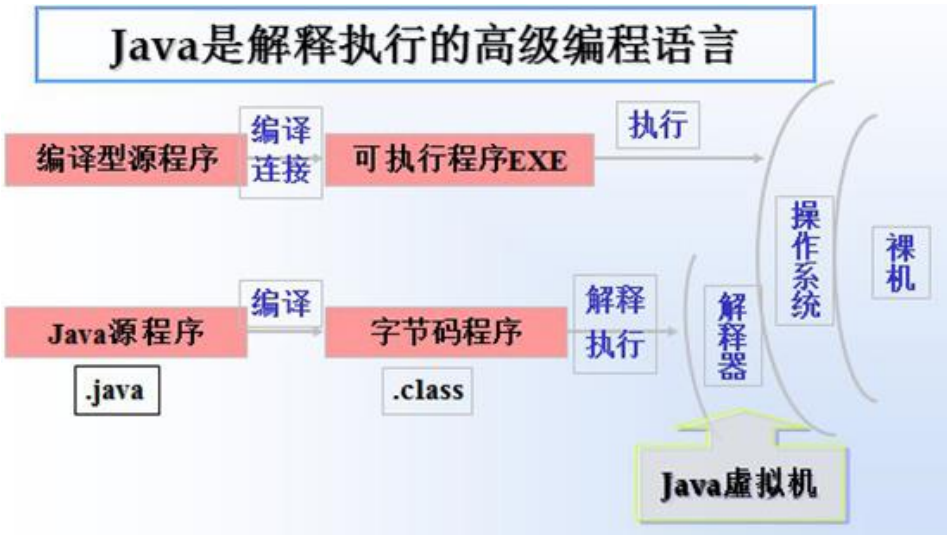
Java 学习笔记

目　录

Java 技术基础

1.1 编程语言

机器语言：0　　1　　在硬件直接执行
汇编语言：助记符
高级语言：　　（Java 运行比 C/C++慢）
1）面向过程的高级语言：程序设计的基本单位为函数，如：C/C++语言。
2）面向对象的高级语言：程序设计的基本单位为类，如：Java、C#。



1.2 Java 的特点

平台无关性、简单性、面向对象、健壮性、多线程、自动内存管理。

平台无关性：指 Java 语言平台无关，而 Java 的虚拟机却不是，需要下载对应平台 JVM 虚拟机的。

自动内存管理：对临时存储的数据自动进行回收，释放内存。如：引用类型的变量没有指向时，被回收；程序执行完后，局部变量被回收。

1.3 Java 开发环境

Java　Developement　Kit——Java 开发工具包，简称 JDK，是由 Sun 公司提供的一个免费的 Java 开发工具，编程人员和最终用户可以利用这个工具来编译、运行 Java 程序。目前版本有 JDK1.0、JDK1.1、JDK1.2、JDK1.3、JDK1.4、JDK1.5（J2SE5.0）、JDK1.6（J2SE6.0）、JDK1.7（J2SE7.0）。

JDK 结构：JDK
　　|--开发工具（Tools）命令：java、javac、jar、rmic ...
　　|-- JRE（Java 基本运行环境）
　　　　|--系统 API 库，系统类库
　　　　|　　系统带来的标准程序库，标准 API
　　　　|-- J VM java 虚拟机
　　　　　　java　语言的执行环境

1.4 Java 开发环境配置

安装完 JDK 之后，不能立刻使用，需要设置环境变量：

1）设置 PATH：D:\Java\jdk1.6.0\bin（指向 JDK 中 bin 文件夹，有各种编译命令）。

2）CLASSPATH：告诉 Java 程序去哪里查找第三方和自定义类，如果 .class 文件和类源文件在同一文件夹内，则不需要配置 classpath，后续有包，则需要。

A．Windows：在命令行执行

set　CLASSPATH=E:\workspace\1304\bin　（临时环境配置）

java　day02.Demo1

- 注意事项：
  - E：\ set classpath　= c:\　（不加分号就不找当前路径）
    = . ; c:\ ; d:\ ;（先找 classpath，若无，再找当前路径）
  - C、D 两盘有同名 . class 文件，classpath 设置为 D 盘，而命令行窗口当前盘符为 C 盘，则 JVM 现找 classpath 路径，后找当前路径。

B．Linux：在控制台执行

①设置 CLASSPATH 环境变量，指向 package 所在的目录，一般是项目文件夹中的 bin 目录。

②执行 java　package.ClassName　（包名必须写）。

export　CLASSPATH=/home/soft01/workspace/1304/bin　（临时环境配置）

java　day01.HelloWorld

java　-cp　/home/soft01/workspace/1304/bin　day01.HelloWorld（二合一）

- 注意事项：
  - Windows 根目录是反斜线：\
  - Linux 根目录是斜线：/

1.5 Linux 命令与相关知识

1）Linux 无盘符，只有一个根目录（root）

2）终端 == 控制台 == 命令行窗口

3）pwd：打印当前工作目录，显示当前工作目录的位置

4）ls：列表显示目录内容，默认显示当前目录内容

5）cd：改变当前工作目录；cd 后不加参数=返回 home 文件夹；cd　~：返回 home；
cd　/：切换到根目录；cd　.. ：返回上一层目录（相对的）；

6）mkdir：创建文件夹（目录）　注意：目录 == 文件夹

7）rm：删除文件；rm　xx　xx：可删多个文件；
rm　–rf　xx：-为减号，r 表递归，f 表强制

8）cat　xx：显示文本文件内容

9）启动 Java 开发工具：cd/opt/eclipse　→　./eclipse　. 表当前目录下

10）绝对路径：　/home　（以 / 开始为绝对路径，相对于根目录）
相对路径：home　（相对于当前工作目录）

11）home（用户主目录，用户的家）：/home/username　如：/home/soft01

12）主目录（home）：有最大访问权限：什么都能干，增删改查、建目录等
其他地方：一般只能查看，不能增删改查、创建目录等

1.6 Eclipse/Myeclipse 程序结构

```
Project 项目文件
 |-- src 源文件
 |     |-- Package 包
 |               |--   .java 源文件
 |-- bin
       |-- Package 包
               |-- .class 字节码程序
```

- 注意事项：
  - Myeclipse5.5 消耗少，Myeclipse6.5 最稳定

Java 语言基础

2.1 基础语言要素

1）标识符：给类、方法、变量起的名字

A．必须以字母或下划线或 $ 符号开始，其余字符可以是字母、数字、$ 符号和下划线。

B．只能包含两个特殊字符，即下划线 _ 和美元符号 $ 。不允许有任何其他特殊字符。

C．标识符不能包含空格。

D．区分大小写。

2）关键字：只有系统才能用的标识符

- 注意事项：
  - true、false、null 不是关键字！是字面量。
  - main 不是关键字！但是是一个特殊单词，可以被 JVM 识别，主函数是固定格式，作为程序的入口。

3）注释：单行注释：//　多行注释：/* ……*/　　文档注释：/**……*/

- 注意事项：开发中类前、属性前、方法前，必须有文档注视。

2.2 八种基本数据类型

1）四种整数类型(byte、short、int、long)：

byte：8 位，用于表示最小数据单位，如文件中数据，-128~127

short：16 位，很少用，-32768 ~ 32767

int：32 位、最常用，-2^31-1~2^31 （21 亿）

long：64 位、次常用

- 注意事项：
  - int i=5; // 5 叫直接量（或字面量），即直接写出的常数。
  - 整数字面量默认都为 int 类型，所以在定义的 long 型数据后面加 L 或 l。
  - 小于 32 位数的变量，都按 int 结果计算。
  - 强转符比数学运算符优先级高。见常量与变量中的例子。

2）两种浮点数类型(float、double)：

float：32 位，后缀 F 或 f，1 位符号位，8 位指数，23 位有效尾数。

double：64 位，最常用，后缀 D 或 d，1 位符号位，11 位指数，52 位有效尾数。

- 注意事项：
  - 二进制浮点数：1010100010=101010001.0*2=10101000.10*2^10(2 次方)=1010100.010*2^11(3 次方)= . 1010100010*2^1010(10 次方)
  - 尾数：　. 1010100010　指数：1010　基数：2
  - 浮点数字面量默认都为 double 类型，所以在定义的 float 型数据后面加 F 或 f；double 类型可不写后缀，但在小数计算中一定要写 D 或 X.X。
  - float 的精度没有 long 高，有效位数（尾数）短。
  - float 的范围大于 long 指数可以很大。
  - 浮点数是不精确的，不能对浮点数进行精确比较。

3）一种字符类型(char)：

char：16 位，是整数类型，用**单引号**括起来的 **1** 个字符（可以是一个中文字符），使用 Unicode 码代表字符，0~2^16-1（65535）。

- 注意事项：
  - 不能为 0 个字符。
  - 转义字符：\n 换行 \r 回车 \t Tab 字符 \" 双引号 \\ 表示一个\
  - 两字符 char 中间用“+”连接，内部先把字符转成 int 类型，再进行加法运算，char 本质就是个数！二进制的，显示的时候，经过“处理”显示为字符。

4）一种布尔类型(boolean)：true 真 和 false 假。

5）类型转换： char-->

自动转换：byte-->short-->int-->long-->float-->double

强制转换：①会损失精度，产生误差，小数点以后的数字全部舍弃。

②容易超过取值范围。

2.3 常量和变量

变量：内存中一块存储空间，可保存当前数据。在程序运行过程中，其值是可以改变的量。

1）必须声明并且初始化以后使用（在同一个作用域中不能重复声明变量）!

2）变量必须有明确类型（Java 是强类型语言）。

3）变量有作用域（变量在声明的地方开始，到块｛｝结束）。变量作用域越小越好。

4）局部变量在使用前一定要初始化！成员变量在对象被创建后有默认值，可直接用。

5）在方法中定义的局部变量在该方法被加载时创建。

常量：在程序运行过程中，其值不可以改变的量。

- 注意事项：
  - 字面量、常量和变量的运算机制不同，字面量、常量由编译器计算，变量由运算器处理，目的是为了提高效率。

    eg：小于 32 位数的字面量处理

```
byte b1 = 1; byte b2 = 3;
//byte b3 = b1+b2;//编译错误，按照 int 结果，需要强制转换
byte b3 = (byte)(b1+b2);
//byte b3 = (byte)b1+(byte)b2;//编译错误！两个 byte、short、char 相加还是按 int 算
System.out.println(b3); //选择结果：A 编译错误 B 运行异常 C 4 D b3
byte b4 = 1+3;//字面量运算，编译期间替换为 4，字面量 4
//byte b4 = 4; 不超过 byte 就可以赋值
```

  - 不管是常量还是变量，必须先定义，才能够使用。即先在内存中开辟存储空间，才能够往里面放入数据。
  - 不管是常量还是变量，其存储空间是有数据类型的差别的，即有些变量的存储空间用于存储整数，有些变量的存储空间用于存储小数。

2.4 运算符与表达式

1）数学运算：+ - * / % ++ --

- 注意事项：
  - + - * / 两端的变量必须是同种类型，并返回同种类型。
  - % 取余运算，负数的余数符号与被模数符号相同， -1%5=-1，1%-5=

1；Num % n，n>0，结果范围[0,n)，是周期函数。

- ❖ 注意整除问题：1 / 2 = 0（整数的除法是整除）1.0 / 2 = 0.5　1D / 2 = 0.5
- ❖ 单独的前、后自增或自减是没区别的，有了赋值语句或返回值，则值不同！

eg1：自增自减

```
int a = 1;  a = a++;  System.out.println("a 的值："+a);
第 1 步：后++，先确定表达式 a++的值（当前 a 的值）  a++ ---->1
第 2 步：++，给 a 加 1                              a  ---->2
第 3 步：最后赋值运算,把 a++整个表达式的值赋值给 a  a  ---->1
a 被赋值两次，第 1 次 a = 2，第 2 次把 1 赋值给 1
```

eg2：自增自减

```
x，y，z 分别为 5，6，7  计算 z + = -- y * z++ ；// x = 5，y = 5，z = 42
        z = z + -- y * z++  → 42 = 7 + 5 * 7  从左到右入栈，入的是值
```

eg3：取出数字的每一位

```
d = num%10;//获取 num 的最后一位数    num/=10; //消除 num 的最后一位
```

2）位运算：&　|　~（取反）　^（异或）>>　<<　>>>

- ◆ 注意事项：
  - ❖ 一个数异或同一个数两次，结果还是那个数。
  - ❖ |：上下对齐，有 1 个 1 则为 1；&：上下对齐，有 1 个 0 则为 0；（都为二进制）
  - ❖ &相当于乘法，| 相当于加法；&：有 0 则为 0，| ：有 1 则为 1，^：两数相同为 0，不同为 1。

3）关系运算符：>　<　>=　<=　==　!=

4）逻辑运算符：&&　||（短路）　！　&　|

eg：短路运算：&&：前为 flase，则后面不计算；|| ：前为 true，则后面不计算

```
int x=1,y=1,z=1;
if(x--==1 && y++==1 || z++==1) //  || 短路运算后面的不执行了！
System.out.println("x="+x+",y="+y+",z="+z);//  0 , 2, 1
```

5）赋值运算符：=　+=　-=　*=　/=　%=

eg：正负 1 交替

```
int flag= -1;  System.out.println(flag *= -1); ……
```

6）条件（三目）运算符：表达式 1 ？ 表达式 2 ：表达式 3

- ◆ 注意事项：
  - ❖ 右结合性：a > b ? a : i > j ? i : j 相当于 a > b ? a : (i > j ? i : j)
  - ❖ 三目运算符中：第二个表达式和第三个表达式中如果都为基本数据类型，整个表达式的运算结果由容量高的决定。如：int x = 4; x > 4 ? 99.9 : 9; 99.9 是 double 类型，而 9 是 int 类型，double 容量高，所以最后结果为 9.9。

7）运算符优先级：括号 > 自增自减 > ~！ > 算数运算符 > 位移运算 > 关系运算 > 逻辑运算 > 条件运算 > 赋值运算

2.5 编程风格

MyEclipse/Eclipse 中出现的红色叉叉：编译错误

编译错误：java 编译器在将 Java 源代码编译为 class 文件的过程出现的错误，一般是语

法使用错误！当有编译错误时候，是没有 class 文件产生，也就不能运行程序。

Java 程序结构：

```
package demo.day01; //必须是小写字母，多个单词用.隔开
import java.util.Scanner;//Java API 一定在当前库中存在

public class HelloWorld {//类名每个单词首字母要大写
   //类体中的成员，要缩进1个Tab 宽
   public static void main(String[] args){
       //方法中成员也要缩进1个Tab 宽
       //Java的语句以英文分号 ； 为结尾,不是中文的分号！
       //括号要配对使用：先写出成对的括号，然后在中间填代码
       Scanner console = new Scanner(System.in);
   }//方法体的结束也要和方法的声明位置对齐
}//类体 class body
//括号要配对，声明开始的位置和结束的位置要对齐
```

2.6 流程控制语句

1）选择控制语句

if 语句：if 、if-else、if-else-if：可以处理一切分支判断。

格式：if(判断){…}、if(判断){…}else{…}、if(判断){…}else if(判断){…}

switch 语句：switch(必须整数类型){case 常量 1：…; case 常量 2：… ; ….}

- 注意事项：
  - int 类型指：byte、short、int，只能写 long 类型，要写也必须强转成 int 类型；而 byte、short 为自动转换成 int。
  - swtich-case：若 case 中无符合的数，并且 default 写在最前（无 break 时），则为顺序执行，有 break 或 } 则退出。
  - swtich-case：若 case 中无符合的数，并且 default 写在最后，则执行 default。
  - swtich-case：若 case 中有符合的数，并且 default 写在最后，并且 default 前面的 case 没有 break 时，default 也会执行。

2）循环控制语句

①for：最常用，用在与次数有关的循环处理，甚至只用 for 可以解决任何循环问题。

- 注意事项：for 中定义的用于控制次数的循环变量，只在 for 中有效，for 结束则循环变量被释放（回收）。

②while：很常用，用在循环时候要先检查循环条件再处理循环体，用在与次数无关的情况。如果不能明确结束条件的时候，先使用 while(true)，在适当条件使用 if 语句加 break 结束循环。

③do-while：在循环最后判断是否结束的循环。如：使用 while(true) 实现循环的时候，结束条件 break 在 while 循环体的最后，就可以使用 do-while 。do-while 的结束条件经常是“否定逻辑条件”，不便于思考业务逻辑，使用的时候需要注意。可以利用 while（true）+ break 替换。

④循环三要素：A. 循环变量初值 B. 循环条件 C. 循环变量增量（是循环趋于结束的表达式）

⑤for 和 while 循环体中仅一条语句，也要补全{ }，当有多条语句，且不写{ }时，它们只执行紧跟着的第一条语句。

⑥循环的替换：

while(布尔表达式){} 等价 for(;布尔表达式;){}

while(true){}　等价　for(;;)

while(true){} + break　替换　do{}while(布尔表达式);

for(;;) + break　替换　do{}while(布尔表达式);

3）跳转控制语句

continue：退出本次循环，直接执行下一次循环

break：退出所有循环

2.7 数组

类型一致的一组数据，相当于集合概念，在软件中解决一组，一堆 XX 数据时候使用数组。

1）数组变量：是引用类型变量（不是基本变量）引用变量通过数组的内存地址位置引用了一个数组（数组对象），即栓到数组对象的绳子。

eg：数组变量的赋值

```
int[] ary = new int[3];// ary----->{0,0,0}<----ary1
int[] ary1 = ary;// ary 的地址赋值给 ary1，ary 与 ary1 绑定了同一个数组
//ary[1] 与 ary1[1] 是同一个元素，数组变量不是数组（数组对象）
```

2）数组（数组对象）有 3 种创建（初始化）方式：①new int[10000] 给元素数量，适合不知道具体元素，或元素数量较多时 ②new int[]{3,4,5} 不需要给出数量，直接初始化具体元素适合知道数组的元素。③ {2,3,4} 静态初始化，是②简化版，只能用在声明数组变量的时候直接初始化，不能用于赋值等情况。

eg：数组初始化

```
int[] ary1 = new int[]{2,3,4};//创建数组时候直接初始化元素
int[] ary2 = {2,3,4};//数组静态初始化，只能在声明变量的同时直接赋值
//ary2 = {4,5,6};//编译错误，不能用于赋值等情况
ary2 = new int[]{4,5,6};
```

3）数组元素的访问：①数组长度：长度使用属性访问，ary.length 获取数组下标。②数组下标：范围：0 ~ length-1 就是[0,length)，超范围访问会出现下标越界异常。③使用[index]访问数组元素：ary[2]。④迭代（遍历）：就是将数组元素逐一处理一遍的方法。

4）数组默认初始化值：根据数组类型的不同，默认初始化值为：0（整数）、0.0（浮点数）、false（布尔类型）、\u0000（char 字符类型，显示无效果，相当于空格，编码为 0 的字符，是控制字符，强转为 int 时显示 0）、null（string 类型，什么都没有，空值的意思）。

5）数组的复制：数组变量的赋值，是并不会复制数组对象，是两个变量引用了同一个数组对象。数组复制的本质是创建了新数组，将原数组的内容复制过来。

6）数组的扩容：创建新数组，新数组容量大于原数组，将原数组内容复制到新数组，并且丢弃原数组，简单说：就是更换更大的数组对象。System.arraycopy() 用于复制数组内容，简化版的数组复制方法：Arrays.copyOf()方法，但需 JKD1.5+。

2.8 字符串

字符串(string)：永远用“”双引号（英文状态下），用字符串连接任何数据（整数），都会默认的转化为字符串类型。

字符串与基本数据类型链接的问题：如果第一个是字符串那么后续就都按字符串处理，如 System.out.println("(Result)"+6 + 6);那么结果就是(Result)66，如果第一个和第二个…第 n 个都是基本数据，第 n+1 是字符串类型，那么前 n 个都按加法计算出结果在与字符串连接。

如下例中的 System.out.println(1+2+”java”+3+4);结果为 3java34。

eg：字符串前后的“+”都是连接符！不是加法运算符！

```
System.out.println("A"+'B');//AB
System.out.println('A'+'B');//131
System.out.println(1+2+”java”+3+4);//3java34
```

- 注意事项：比较字符串是否相等必须使用 equals 方法！不能使用==。"1".equals(cmd) 比 cmd.equals("1") 要好。

2.9 方法三要素

方法：method（函数 function = 功能） y=f(x)

1）方法的主要三要素：方法名、参数列表、返回值。

2）什么是方法：一个算法逻辑功能的封装，是一般完成一个业务功能，如：登录系统，创建联系人，简单说：方法是动作，是动词。

3）方法名：一般按照方法实现的功能定名，一般使用动词定义，一般使用小写字母开头，第二个单词开始，单词首字母大写。如：createContact() 。

4）参数列表：是方法的前提条件，是方法执行依据，是数据。如：

login(String id, String pwd) ，参数的传递看定义的类型及顺序，不看参数名。

5）方法返回值：功能执行的结果，方法必须定义返回值，并且方法中必须使用 return 语句返回数据；如果无返回值则定义为 void，此时 return 语句可写可不写；返回结果只能有一个，若返回多个结果，要用数组返回（返回多个值）。

- 注意事项：递归调用：方法中调用了方法本身，用递归解决问题比较简练，只需考虑一层逻辑即可！但是需要有经验。一定要有结束条件！如：f(1)=1; 递归层次不能太深。总之：慎用递归！

2.10 插入排序

将数组中每个元素与第一个元素比较，如果这个元素小于第一个元素，则交换这两个元素循环第 1 条规则，找出最小元素，放于第 1 个位置经过 n-1 轮比较完成排序。

```
for(int i = 1; i < arr.length; i++) {
    int k = arr[i];// 取出待插入元素
    int j;// 找到插入位置
    for (j = i - 1; j >= 0 && k < arr[j]; j--) {
        arr[j + 1] = arr[j];// 移动元素
    }
    arr[j + 1] = k;// 插入元素
    System.out.println(Arrays.toString(arr));
}
```

2.11 冒泡排序

比较相邻的元素，将小的放到前面。

```
for(int i = 0; i < arr.length - 1; i++) {
    boolean isSwap = false;
    for (int j = 0; j < arr.length - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
            int t = arr[j];
```

```
                    arr[j] = arr[j + 1];
                    arr[j + 1] = t;
                    isSwap = true;
                }
            }
            if (!isSwap){     break;     }
            System.out.println(Arrays.toString(arr));
        }
```

2.12 冒泡排序：轻气泡上浮的方式

冒泡排序法可以使用大气泡沉底的方式，也可以使用轻气泡上浮的方式实现。如下为使用轻气泡上浮的方式实现冒泡排序算法。

```
        for (int i = 0; i < arr.length - 1; i++) {
            boolean isSwap = false;
            for (int j = arr.length - 1; j > i; j--) {
                if (arr[j] < arr[j - 1]) {
                    int t = arr[j];
                    arr[j] = arr[j - 1];
                    arr[j - 1] = t;
                    sSwap = true;
                }
            }
            if (!isSwap){     break;     }
            System.out.println(Arrays.toString(arr));
        }
```

2.13 二分法查找

```
        intlow = 0;             inthigh = arr.length - 1;          intmid = -1;
        while(low <= high) {
            mid = (low + high) / 2;
            if (arr[mid] < value){       low = mid + 1;        }
            else if (arr[mid] > value){     high = mid - 1;     }
            else{     break;}
        }
        if (low <= high) {            System.out.println("可以找到：index = " + mid + "。");
        } else {       System.out.println("无法找到！");          }
```

二分法思想是取中，比较 ：

1）求有序序列 arr 的中间位置 mid。　　2）k 为要查找的数字。

若 arr[mid] == k，查找成功；

若 arr[mid] > k，在前半段中继续进行二分查找；

若 arr[mid] < k，则在后半段中继续进行二分查找。

假如有一组数为 3、12、24、36、55、68、75、88 要查给定的值 k=24。可设三个变量 low、mid、high 分别指向数据的上界，中间和下界，mid=（low+high）/2.

1）开始令 low=0（指向 3），high=7（指向 88），则 mid=3（指向 36）。因为 k<mid，故

应在前半段中查找。

2）令新的 high=mid-1=2（指向 24），而 low=0（指向 3）不变，则新的 mid=1（指向 12）。此时 k>mid，故确定应在后半段中查找。

3）令新的 low=mid+1=2（指向 24），而 high=2（指向 24）不变，则新的 mid=2，此时 k=arr[mid]，查找成功。

如果要查找的数不是数列中的数，例如 k=25，当第四次判断时，k>mid[2]，在后边半段查找，令 low=mid+1，即 low=3(指向 36)，high=2（指向 24）出现 low>high 的情况，表示查找不成功。

2.14 Java 系统 API 方法调用

Arrays 类，是数组的工具类，包含很多数组有关的工具方法。如：

1）toString() 连接数组元素为字符串，方便数组内容输出。

2）equals 比较两个数组序列是否相等。

3）sort() 对数组进行排序，小到大排序。

4）binarySearch(names, "Tom") 二分查找，必须在有序序列上使用。

2.15 二进制基础

1）计算机中一切数据都是 2 进制的！基本类型，对象，音频，视频。

2）10 进制是人类习惯，计算按照人类习惯利用算法输入输出。

"10" -算法转化-> 1010(2)　　1010 -算法转化-> "10"

3）16 进制是 2 进制的简写，16 进制就是 2 进制！

4）计算机硬件不支持正负号，为了解决符号问题，使用补码算法，补码规定高位为 1 则为负数，每位都为 1 则为-1，如 1111 1111 = -1 = 0xff

5）二进制数右移>>：相当于数学 / 2(基数)，且正数高位补 0，负数高位补 1；二进制字左移<<：相当于数学 * 2(基数)，且低位补 0；二进制数无符号右移>>>：相当于数学 / 2(基数)，且不论正负，高位都补 0。

6）注意掩码运算：把扩展后前面为 1 的情况除去，与 0xff 做与运算。

eg1：二进制计算

```
int max = 0x7fffffff;        long l = max + max + 2;     System.out.println( l );// 0
```

eg2：二进制运算（拼接与拆分）

```
int b1 = 192; int b2 = 168; int b3 = 1; int b4 = 10; int color = 0xD87455;
int ip = (b1<<24) + (b2<<16) + (b3<<8) + b4; //  或者 ip = (b1<<24) | (b2<<16) | (b3<<8) | b4;
int b = color&0xff; // 85    int g = (color >>> 8)&0xff; // 116    int r = (color >>> 16)&0xff;// 216
```

2.16 Java 基础其他注意事项

- Java 程序严格区分大小写。
- 类名，每个单词首字母必须大写（公司规范！）。
- 一个 Java 应用程序，有且只有一个 main 方法，作为程序的入口点。
- 每一条 Java 语句必须以分号结束。
- 类定义关键字 class 前面可以有修饰符（如 public），如果前面的修饰符是 public，该类的类名必须要与这个类所在的源文件名称相同。
- 注意程序的缩进。
- double a[] = new double[2]; //语法可以，但企业中一定不要这么写，Java 中[]建议放前面。

- **Java 中所有范围参数都是包含 0，不包含结束**，如 int n = random.nextInt(26); //生成 0 到 26 范围内的随机数，不包括 26。
- 任何数据在内存中都是 2 进制的数据，内存中没有 10 进制 16 进制。
- int n = Integer.parseInt(str);//将字符串--> int 整数。
- System.out.println(Long.toBinaryString(maxL)); Long 类型用 Long.XXXX 。
- 程序：数据+算法 数据即为变量，算法为数据的操作步骤，如：顺序、选择、循环。
- 字符串按编码大小排序。

面向对象

Object：对象，东西，一切皆对象 == 啥都是东西

面向对象核心：封装、继承、多态。

3.1 类

1）是同类型东西的概念，是对现实生活中事物的描述，映射到 Java 中描述就是 class 定义的类。类是对象的模板、图纸，是对象的数据结构定义。简单说就是“名词”。

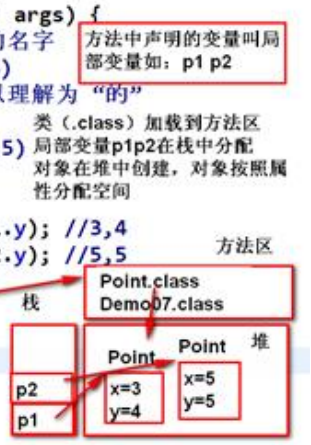
2）其实定义类，就是在描述事物，就是在定义属性（变量）和方法（函数）。

3）类中可以声明：属性，方法，构造器；属性（变量）分为：实例变量，局部变量；实例变量：用于声明对象的结构的，在创建对象时候分配内存，每个对象有一份！实例变量（对象属性）在堆中分配，并作用于整个类中，实例变量有默认值，不初始化也能参与运算；局部变量在栈中分配，作用于方法或语句中，必须初始化，有值才能运算。

4）类与类之间的关系：①关联：一个类作为另一个类的成员变量：需要另一个类来共同完成。class A { pulic B b } class B {} ②继承：class B extends A {} class A {} ③依赖：个别方法和另一个类相关。class A { public void f(B b) {}//参数里有 B public B g() {}// 返回值里有 B } class B {}

```
package day01;

public class Demo07 {
  public static void main(String[] args) {
    Point p1;//引用“变量”，是对象的名字
    p1 = new Point();//对象 p1(3,4)
    p1.x = 3;//. 叫取成员运算，可以理解为“的”
    p1.y = 4;
    Point p2 = new Point();//p2(5,5)
    p2.x = 5;
    p2.y = 5;
    System.out.println(p1.x+","+p1.y); //3,4
    System.out.println(p2.x+","+p2.y); //5,5
  }
}
class Point{//类
  int x;
  int y;
}
```



5）**null 与空指针异常：**引用类型变量用于存放对象的地址，可以给引用类型赋值为 null，表示不指向任何对象。当某个引用类型变量为 null 时无法对对象实施访问（因为它没有指向任何对象）。此时，如果通过引用访问成员变量或调用方法，会产生 NullPointerException 空指针异常。

◆ 注意事项：除了 8 中基本类型，其他都是引用类型变量（也叫句柄）。

3.2 对象

是这类事物实实在在存在的个体！利用类做为模板创建的个体实例，本质是数据。

匿名对象：使用方式一：当对对象的方法只调用一次时，可用匿名对象来完成，这样比较简化。如果对一个对象进行多个成员调用，则必须给这个对象起个名字。

使用方式二：可以将匿名对象作为实际参数进行传递。

3.3 包

1）包名必须是小写，多个单词用“.”隔开。在同一个包中，不能有同名的类！

2）只要在同一个包中，则可直接用 extends（类型互知道在哪），若不在同一个包中，则用 import 导入。

3.4 方法及其调用

是用于对当前对象数据进行算法计算，实现业务功能。方法是对象的功能，对象的动作，对象的行为。总之是动词！方法名没有规定，建议首单词为小写动词，其他单词首字母大写。必须定义返回值！可有无参数方法。**方法调用只有两种方式：**①对象引用调用②类名调用（即静态类时）。

3.5 引用

是对个体的标识名称。

1）是代词，是对象的引用，就像拴着对象的绳子。

2）引用本身不是对象！引用指代了对象！

3）引用的值是对象的地址值（或叫句柄），通过地址值引用了对象。

4）引用的值不是对象！

◆ 注意事项：“.”叫取成员运算，可以理解为“的”。

3.6 访问控制（封装）

封装：将数据封装到类的内部，将算法封装到方法中。

1）封装原则：将不需要对外提供的内容都隐藏起来，把属性都隐藏，提供公共方法对其访问，通常有两种访问方式：set 设置，get 获取。

2）封装结果：存在但是不可见。

3）public：任何位置可见，可以修饰：类、成员属性、成员方法、内部类、跨包访问类（需要使用 import 语句导入），成员属性 == 成员变量。

4）protected：当前包中可见，子类中可见。可以修饰：成员属性、成员方法、内部类（只能在类体中使用，不能修饰类）。

5）默认的：当前包内部可见，就是没有任何修饰词，可以修饰：类、成员属性、成员方法、内部类，但在实际项目中很少使用。默认类（包内类）的访问范围：当前包内部可见，不能在其他包中访问类，访问受限！main 方法若定在默认类中 JVM 将找不到，无法执行，因此必定在 public 类中。

6）private：仅仅在类内部可见。可以修饰：成员属性、成员方法、内部类（只能在类体中使用，不能修饰类）。私有的方法不能继承，也不能重写。

◆ 注意事项：在企业项目中建议：所有类都是公用类。封装的类使用内部类！

3.7 构造器

用于创建对象并初始化对象属性的方法，叫“构造方法”，也叫“构造器”；构造器在类中定义。

1）构造器的名称必须与类名同名，包括大小写。

2）构造器没有返回值，但也不能写 void，也不能写 return。

3）构造器的参数：一般是初始化对象的前提条件。

4）用 new 调用！且对象一建立，构造器就运行且仅运行一次。一般方法可被调用多次。

5）类一定有构造器！这是真的，不需要质疑！

6）如果类没有声明（定义）任何的构造器，Java编译器会自动插入默认构造器！

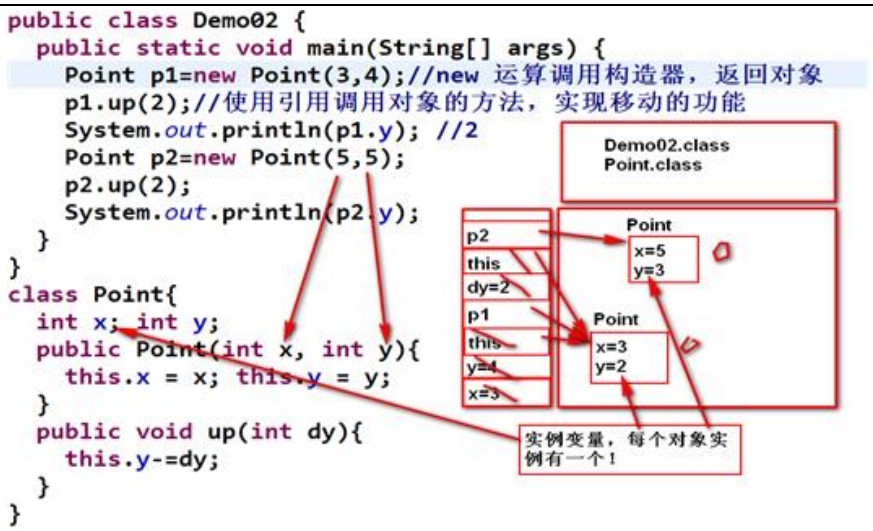
7）**默认构造是无参数，方法体是空的构造器**，且默认构造器的访问权限随着所属类的访问权限变化而变化。如，若类被 public 修饰，则默认构造器也带 public 修饰符。

8）默认构造器是看不到的，一旦自己写上构造器则默认构造器就没有了，自己写的叫自定义构造器，即便自己写的是空参数的构造器，也是自定义构造器，而不是默认构造器。

9）**如果类声明了构造器，Java编译器将不再提供默认构造器**。若没手动写出无参构造器，但却调用了无参构造器，将会报错！

eg：默认构造器

```
public class Demo {     public static void main(String[] args) {
                Foo foo = new Foo();//调用了 javac 自动添加的默认构造器！
                //Koo koo = new Koo();//编译错误，没有 Koo()构造器
                Koo koo = new Koo(8);           }           }
class Foo { } //Foo 有构造器，有无参数的默认构造器！
class Koo {       public Koo(int a) {   //声明了有参数构造器
                        System.out.println("Call Koo(int)");   }         }
```



10）构造器是可以重载的，重载的目的是为了使用方便，重载规则与方法重载规则相同。

11）构造器是不能继承的！虽说是叫构造方法，但实际上它不是常说的一般方法。

12）子类继承父类，那么子类型构造器默认调用父类型的无参数构造器。

13）子类构造器一定要调用父类构造器，如果父类没有无参数构造器，则必须使用 super(有参数的)，来调用父类有参的构造器。 那么，为什么子类一定要访问父类的构造器？因为父类中的数据子类可以直接获取。所以子类对象在建立时，需要先查看父类是如何对这些数据进行初始化的，所以子类在对象初始化时，要先访问一下父类中的构造器。

总之，子类中至少会有一个构造器会访问父类中的构造器，且子类中每一个构造函数内的第一行都有一句隐式 super()。

3.8 super()、super．和 this()、this．

1）this：在运行期间，哪个对象在调用 this 所在的方法，this 就代表哪个对象，隐含绑定到当前“这个对象”。

2）super()：调用父类无参构造器，一定在子类构造器第一行使用！如果没有则是默认存在 super()的！这是 Java 默认添加的 super()。

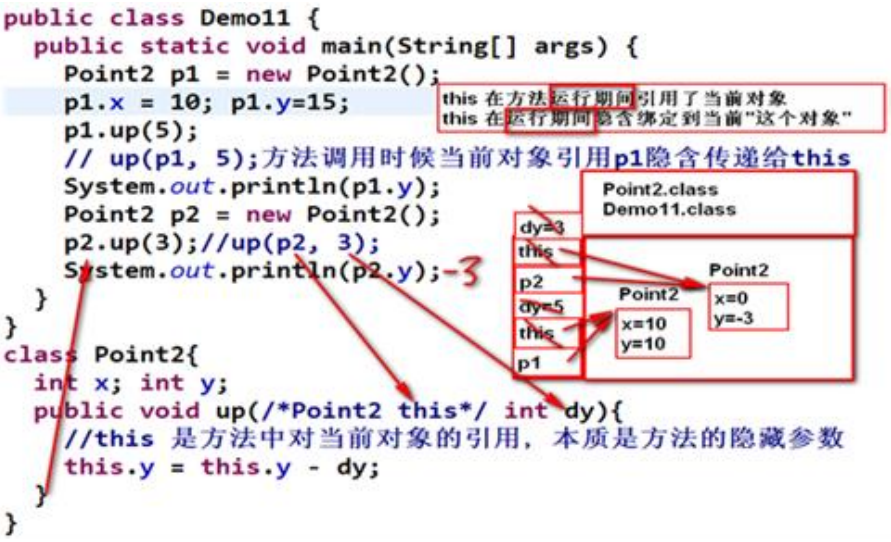
3）super．是访问父类对象，父类对象的引用，与 this．用法一致

4）this()：调用本类的其他构造器，按照参数调用构造器，必须在构造器中使用，必须在第一行使用，this() 与 super() 互斥，不能同时存在

5）this．是访问当前对象，本类对象的引用，在能区别实例变量和局部变量时，this 可省略，否则一定不能省！

6）如果子父类中出现非私有的同名成员变量时，子类要访问本类中的变量用 this． ；子类要访问父类中的同名变量用 super． 。

eg1：方法参数传递原理 与 this 关键字



eg2：this. 和 this()

```
Cell c = new Cell();       System.out.println(c.x + ","+c.y);
class Cell {       int x; int y;
        public Cell() {    this(1,1);//调用本类的其他构造器       }
        public Cell( int x, int y) {       this.x = x ; this.y = y;     }              }
```

eg3：super()

```
class Xoo{       public Xoo(int s) {     System.out.println("Call Xoo(int)");      }              }
        //super()用于在子类构造器中调用父类的构造器
class Yoo extends Xoo{
    //public Yoo() {}//编译错误，子类调用不到父类型无参数构造器
public Yoo(){//super();//编译错误，子类调用不到父类型无参数构造器
                super(100);//super(100) 调用了父类 Xoo(int) 构造器 }          }
```

3.9 重载和重写

1）**重写**：通过类的继承关系，由于父类中的方法不能满足新的要求，因此需要在子类中修改从父类中继承的方法叫重写（覆盖）。

①方法名、参数列表、返回值类型与父类的一模一样，但方法的实现不同。若方法名、参数列表相同，但返回值类型不同会有变异错误！若方法名、返回值类型相同，参数列表不同，则不叫重写了。

②子类若继承了抽象类或实现了接口，则必须重写全部的抽象方法。若没有全部实现抽象方法，则子类仍是一个抽象类！

③子类重写抽象类中的抽象方法或接口的方法时，访问权限修饰符一定要大于或等于被重写的抽象方法的访问权限修饰符！

④静态方法只能重写静态方法！

2）**重载**：方法名一样，参数列表不同的方法构成重载的方法（多态的一种形式）。

①调用方法：根据参数列表和方法名调用不同方法。

②与返回值类型无关。

③重载遵循所谓“编译期绑定”，即在编译时根据**参数变量的类型**判断应调用哪个方法。 eg：重载

```
int[] ary1 = {'A','B','C'};              char[] ary2 = {'A', 'B', 'C'};
System.out.println(ary1);//println(Object)
                        //按对象调用，结果为地址值，没有 println(int[])
System.out.println(ary2);//println(char[])   ABC
System.out.println('中');//println(char)      中
System.out.println((int)'中');//println(int)   20013
```

3.10 继承

父子概念的继承：圆继承于图形，圆是子概念（子类型 Sub class）图形是父类型（Super Class 也叫超类），继承在语法方面的好处：子类共享了父类的属性和方法的定义，子类复用了父类的属性和方法，节省了代码。

1）继承是 is a ：“是”我中的一种，一种所属关系。

2）子类型对象可以赋值给父类型变量（多态的一种形式），变量是代词，父类型代词可以引用子类型东西。

3）继承只能是单继承，即直接继承，而非间接继承。因为多继承容易带来安全隐患，当多个父类中定义了相同功能，当功能内容不同时，子类无法确定要运行哪一个。

4）父类不能强转成子类，会造型异常！子类向父类转化是隐式的。

5）只有变量的类型定义的属性和方法才能被访问！见下例。

6）重写遵循所谓“运行期绑定”，即在运行的时候根据**引用变量指向的实际对象类型**调用方法。

eg：Shape s，s 只能访问 Shape 上声明的属性和方法

```
Circle c = new Circle(3,4,5);
Shape s = c;//父类型变量 s 引用了子类型实例
//s 和 c 引用了同一个对象 new Circle(3,4,5)
s.up();          System.out.println(c.r);
System.out.println(c.area());
//System.out.println(s.area());//编译错误
//System.out.println(s.r);//在 Shape 上没有定义 r 属性！
```

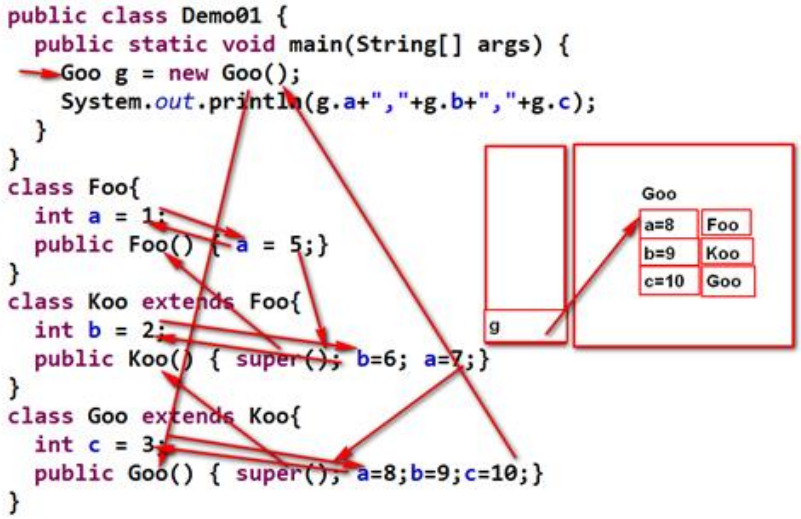
7）引用类型变量的类型转换 instanceof

```
public static void main(String[] args) {
    Circle c = new Circle(3,4,5);          Rect r = new Rect(3,4,5,6);
    Shape s = c;          Shape s1 = r;
    //Circle x = s;//编译错误，父类型变量不能赋值给子类型
    Circle x = (Circle)s;//正常执行
    //Circle y = (Circle)s1;//运行异常,类型转换异常
    //instaceof instace：实例 of:的
    //instaceof 运算 检查变量引用的对象的类型是否兼容
    //s 引用的是圆对象，s instanceof Circle 检查 s 引用的对象是否是 Circle 类型的实例！
```

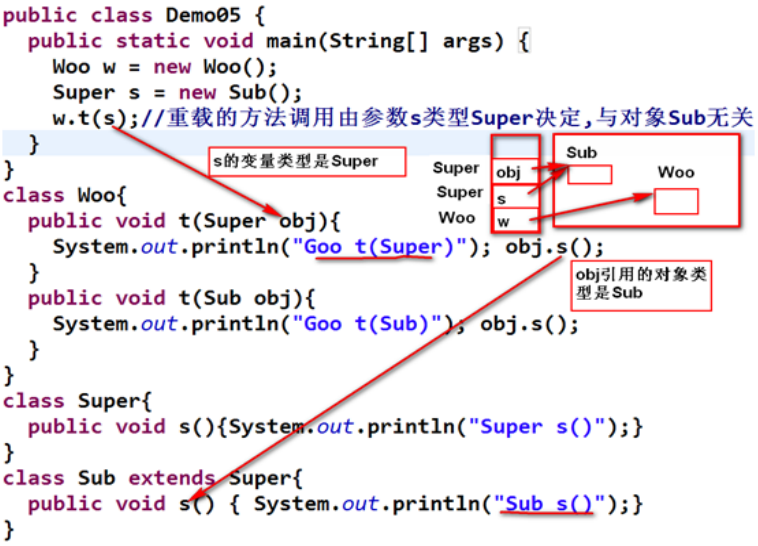
```
        System.out.println(s instanceof Circle);//true
        System.out.println(s1 instanceof Circle);//false
        test(c);        test(r);            }
    public static void test(Shape s){//多态的参数
        //if(s instanceof Circle)保护了(Circle)s 不会出现异常
        if(s instanceof Circle){//实现了安全的类型转换
            Circle c = (Circle) s;     System.out.println("这是一个圆，面积"+c.area()); }
        if(s instanceof Rect){
            Rect r = (Rect) s;     System.out.println("这是一个矩形，面积"+r.area());  }     }
```

8）继承时候对象的创建过程

①Java 首先递归加载所有类搭配方法区。②分配父子类型的内存（实例变量）。③递归调用构造器。

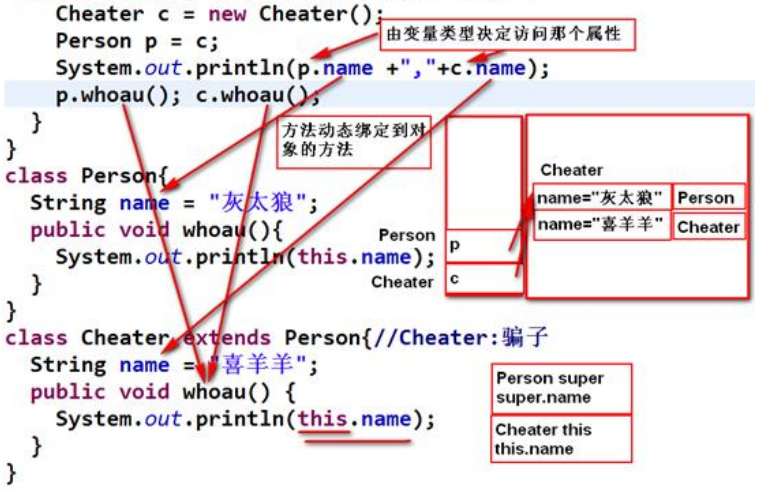


9）重写方法与重载方法的调用规则



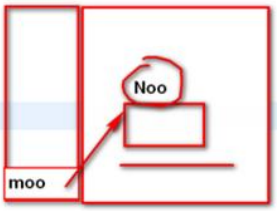
10）属性绑定到变量的类型，由变量类型决定访问哪个属性；方法动态绑定到对象，由对象的类型决定访问哪个方法。（强转对方法动态绑定到对象无影响，因为强转的是父类的引用，而实例是没变的，只是把实例当作另一个状态去看而已。但是强转对属性动态绑定到变量类型有影响。）其他解释请看多态部分！

```
public class Demo04 {
  public static void main(String[] args) {
    Cheater c = new Cheater();
    Person p = c;
    System.out.println(p.name +","+c.name);
    p.whoau(); c.whoau();
  }
}
class Person{
  String name = "灰太狼";
  public void whoau(){
    System.out.println(this.name);
  }
}
class Cheater extends Person{//Cheater:骗子
  String name = "喜羊羊";
  public void whoau() {
    System.out.println(this.name);
  }
}
```



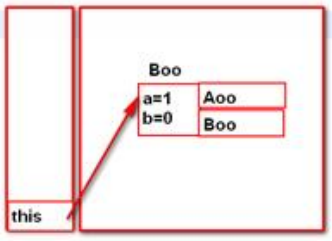
eg1：方法动态绑定到运行期间对象的方法 实例 1

```
public class Demo02 {
  public static void main(String[] args) {
    Moo moo = new Noo();//父类型变量引用了子类对象
    moo.test();//父类型Moo上声明的方法，子类型重写的方法
    //动态绑定到Noo对象，执行Noo对象的方法
  }
}
class Moo{
  public void test(){
    System.out.println("Moo test()");
  }
}
class Noo extends Moo{
  public void test() {
    System.out.println("Noo test()");
  }
}
```



eg2：方法动态绑定到运行期间对象的方法 实例 2

```
public class Demo03 {
  public static void main(String[] args) {
    Boo b = new Boo();
  }
}
class Aoo{
  int a=1;
  public Aoo() { this.test(); }
  public void test(){
    System.out.println("Aoo "+a);
  }
}
class Boo extends Aoo{
  int b=2;
  public Boo() { super(); }//默认构造器
  public void test() {
    System.out.println("Boo "+a+","+b);
  }
}
```



11）为何查阅父类功能，创建子类对象使用功能？

Java 中支持多层继承，也就是一个继承体系。想要使用体系，先查阅父类的描述，因为父类中定义的是该体系中共性的功能，通过了共性功能，就可以知道该体系的基本功能，那么这个体系已经可以基本使用了，然而在具体调用时，要创建最（低）子类的对象，原因

如下：①因为父类有可能不能创建对象②创建子类对象，可以使用更多的功能，包括基本的也包括特有的。

12）属性无继承概念，所以你有你的，我有我的，各自调用各自的，不影响，即使子父类中有同名属性也无影响。

eg：子父类同名属性无影响

```
class Base {    public static final String FOO="foo";    public static void main(String[] args){
                    Base b=new Base();          Sub s=new Sub();
                    Base.FOO;//foo        b.Foo; //foo       Sub.Foo;//bar        s.Foo;//bar
                  ( (Base) s . Foo);// foo                                                  }          }
class Sub extends Base{    public static final String FOO="bar";    }
```

3.11 static

静态的，只能在类内部使用，可以修饰：属性，方法，内部类。在类加载期间初始化，存在方法区中。

1）静态成员随着类的加载而加载，加载于方法区中，且优先于对象存在。

2）静态修饰的成员：属于类级别的成员，是全体类实例（所有对象）所共享。

3）静态属性：只有一份（而实例变量是每个对象有一份），全体实例共享，类似于全局变量。

4）使用类名访问静态变量，以及类名直接调用方法，不需要创建对象。

5）静态方法只能访问静态成员（静态属性和静态方法），非静态方法既可访问静态，也可访问非静态。

6）静态方法中没有隐含参数 this，因此不能访问当前对象资源。也不能定义 this 和 super 关键字，因为静态优于对象先存在。

7）非静态方法省略的是 this，静态方法省略的是类名（在同一类中），即直接使用属性和方法。

8）静态方法一般用于与当前对象无关工具方法，工厂方法。如：Math.sqrt() Arrays.sort()

9）静态代码块：随着类的加载而执行（用到类的内容才叫加载，只有引用是不加载的），且只执行一次，且优先于主函数，用于给类初始化。

10）代码块（构造代码块）：给所有对象进行统一初始化，且优先于构造器执行；而构造器是给对应的对象进行初始化。

eg：静态代码块与代码块（构造代码块）

```
class Goo{
    {//代码块（构造代码块），在创建对象时候执行！类似于构造器的作用
        System.out.println("HI");
     }
    static{//静态代码块，在类的加载期间执行，只执行一次
        System.out.println("Loading Goo.class");
    }
}
public static void main(String[] args) {
    Point p1 = new Point(3,4);          Point p2 = new Point(6,8);
    //在对象上调用方法，当前对象隐含传递给隐含参数 this
    System.out.println(p1.distance(p2));//distance(p1,p2)
    double d = Point.distance(p1, p2);        System.out.println(d); //5
```

```
    //静态方法调用时候不传递隐含的当前对象参数          }
class Point{          int x; int y;
    public Point(int x, int y) {      this.x = x; this.y = y;   }
 //静态方法中没有隐含参数 this！在静态方法中不能访问 this 的属性和方法！
    public static double distance(Point p1, Point p2){
    int a = p1.x -p2.x;   int b = p1.y -p2.y;     return Math.sqrt(a*a + b*b);          }
/** 计算当前点（this）到另外一个点（other）的距离 */
    public double distance(/*Point this*/ Point other){
        int a = this.x - other.x;          int b = this.y - other.y;
      double c = Math.sqrt(a*a + b*b);      return   c;          }          }
```

11）对象的创建过程及顺序：

Person P = new Person(“chang” , 23) ; 这句话都做了什么事情？

①因为 new 用到了 Person.class，所以会先找到 Person.class 文件加载到内存中。

②执行该类中的 static 代码块（如果有的话），给 Person 类.class 类进行初始化。

③在堆内存中开辟空间，分配内存地址，栈内存中开辟空间也就有了。

④在堆内存中建立对象的特有属性，并进行默认（隐式）初始化。

⑤对属性进行显式初始化。

⑥对对象进行构造代码块初始化。

⑦对对象进行对应的构造器初始化。

⑧将内存地址赋给栈内存中的 P 变量。

3.12 final

最终的，可以修饰：类、方法、变量（成员变量和局部变量）。

1）final 修饰的类：不能再继承。

2）final 修饰的方法：不能再重写。

3）final 的方法和类，阻止了动态代理模式！动态代理模式广泛的应用在： Spring Hibernate Struts2

4）企业编程规范：不允许使用 final 的方法和类！

5）final 的变量：final 变量只能初始化一次（赋值一次，且方法中不能有给 final 变量赋值的语句！因为方法可被调用多次！），不能再修改！也可在方法的参数列表中添加 final。

eg1：final 的局部变量

```
final int a;      a = 5;//第一次叫初始化！不是赋值   //a = 8;//编译错误
public static void test(final int a, int b){
        //a++;//编译错误，不能再修改
        System.out.println(a);                          }
```

eg2：final 的数组

```
final String[] ary={"A","B"};          //ary:数组变量，ary[0]数组元素
ary[0]="Tom";//数组元素可以修改
//ary=null;//数组变量不能修改
```

eg3：final 的实例变量

```
public static void main(String[] args) {
    Dog d1 = new Dog();          Dog d2 = new Dog();
    //d1.id = 8;//每个实例的 id 不可以再修改
    System.out.println(d1.id+","+d2.id+","+Dog.numOfDogs);        }
```

```
class Dog{      final int id;//实例变量，每个对象一份，不能再次修改
                static int numOfDogs=0;//静态，只有一份
                public Dog() {          id = numOfDogs++;       }               }
```

6）static final 共同修饰的叫常量，常量：public static final double PI = 3.14; PI 是直接数的代名词，是名字。字面量(==直接量)：直接写出数值 3.1415926535897 宏观说：字面量和常量都称为常量！

3.13 多态

继承体现了多态：父类型变量可以引用各种各样的子类型实例，也可接收子类对象。

个体的多态：父类型的子类型实例是多种多样的。

行为的多态：父类型定义方法被子类重写为多种多样的，重载也是多态的方法。

1）千万不能出现将父类对象转成子类类型，会造型异常！

2）多态前提：必须是类与类之间有关系。要么继承，要么实现。通常还有一个前提：存在覆盖。

3）多态的好处：多态的出现大大的提高程序的扩展性。

4）多态的弊端：虽然提高了扩展性，但是只能使用父类的引用访问父类中的成员。

5）在多态中成员函数的特点：

①在编译时期:参阅引用型变量所属的类中是否有调用的方法。如果有,编译通过,如果没有编译失败。

②在运行时期：参阅对象所属的类中是否有调用的方法。

③简单总结就是：成员方法在多态调用时，编译看左边，运行看右边。

6）在多态中，成员变量的特点：无论编译和运行，都参考左边(引用型变量所属的类)。

7）在多态中，静态成员方法和属性的特点：无论编译和运行，都参考做左边。

8）父类引用指向子类对象，当父类想使用子类中特有属性、方法时，要向下转型。

3.14 抽象类

抽象就是将拥有共同方法和属性的对象提取出来，提取后，重新设计一个更加通用、更加大众化的类，就叫抽象类。

1）abstract 关键字可以修饰类、方法，即抽象类和抽象方法。

2）抽象类可以有具体的方法，或者全部都是具体方法，但一个类中只要有一个抽象方法，那么这个类就是抽象类，并且必须用 abstract 修饰类。

3）抽象类可以被继承，则子类必须实现抽象类中的全部抽象方法，否则子类也将是抽象类。抽象类也可主动继承实体类。

4）抽象类不能实例化，即不能用 new 生成实例。

5）可以声明一个抽象类型的变量并指向具体子类的对象。

6）抽象类可以实现接口中的方法。

7）抽象类中可以不定义抽象方法，这样做仅仅是不让该类建立对象。

3.15 接口

interface 差不多 = = abstract class

1）接口是 like a ：“像”我中的一种，是继承体系之外的，用于功能扩展！想扩展就实现，不想就不用实现。

2）接口中只能声明抽象方法和常量且声明格式都是固定的，只不过可以省略。

eg：接口中声明常量和抽象方法的格式是固定的

```
interface Runner {
    /*public abstract final*/int SPEED=100;//声明常量
    /*public abstract  省略了，写也对*/void run();//声明抽象方法
}
```

3）接口中的成员不写修饰符时，默认都是 public。

4）接口不能有构造器，因为不能实例化何以初始化，接口只能被“实现”。

5）具体类实现了一个接口，则必须实现全部的抽象方法，若没有全部实现，则该类为抽象类。所以说，接口约定了具体类的方法，约定了类的外部行为。

6）具体类可以同时实现多个接口，就是多继承现象。

7）多重继承：class Cat implements Hunter , Runner Cat 即是 Hunter 也是 Runner。

8）接口用 implements 表实现，实际是继承关系，可有多个接口（实现），继承用 extends 只能有一个继承关系。

9）一个类既可以继承的同时，又“实现”接口：class A extends B implements C , D

10）类与类之间是继承关系，类与接口之间是实现关系，接口与接口之间是继承关系，且只有接口之间可以多继承，即：interface A{}，interface B{}，interface C extends A , B 但接口多继承时要注意，要避免 A、B 接口中有方法名相同、参数列表相同，但返回值类型不相同的情况，因为被具体类实现时，不确定调用哪个方法。

11）abstract class 和 interface 有什么区别。

①从语法角度：abstract class 方法中可以有自己的数据成员，也可以有非 abstract 的成员方法，并赋予方法的默认行为，而在 interface 方式中一般不定义成员数据变量，所有的方法都是 abstract，方法不能拥有默认的行为。

②从编程的角度：abstract class 在 java 语言中表示的是一种继承关系，一个类只能使用一次继承关系。而一个类可以实现多个 interface。

③从问题域角度：abstract class 在 Java 语言中体现了一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is a"关系，即父类和派生类在概念本质上应该是相同的。对于 interface 来说则不然，并不要求 interface 的实现者和 interface 定义在概念本质上是一致的，仅仅是实现了 interface 定义的契约而已。

3.16 内部类

当描述事物时，事物的内部还有事物，该事物用内部类来描述。因为内部事物在使用外部事物的内容。

在类内部定义的类为成员内部类，在方法里定义的类为局部内部类，被 static 修饰的为静态内部类。一个类中可有多个内部类。

1）内部类主要用于，封装一个类的声明在类的内部，减少类的暴露。

2）内部类的实例化：实例化时不需要出写对象，非要写的话为：new 外部类名．内部类名()；而不是外部类名．new 内部类名()。

3）内部类的访问规则：内部类可以直接访问外部类中的成员，包括私有。之所以可以直接访问外部类中的成员，是因为内部类中持有了一个外部类的引用，格式：外部类名．This 即下面第 4 条。外部类要访问内部类，必须建立内部类对象。

4）当内部类定义在外部类的成员位置上，而且非私有，则在外部其他类中可以直接建立内部类对象。格式：外部类名．内部类名 变量名 = 外部类对象．内部类对象；

```
Outer.Inner in = new Outer().new Inner();
```

5）当内部类在成员位置上，就可以被成员修饰符所修饰。比如 private：将内部类在外部类中进行封装。

6）静态内部类：被 static 修饰后就具备了静态的特性。当内部类被 static 修饰后，只能直接访问外部类中的 static 成员，出现了访问局限。

①在外部其他类中，如何直接访问 static 内部类的非静态成员呢？

new Outer.Inner().function();

②在外部其他类中，如何直接访问 static 内部类的静态成员呢？

Outer.Inner.function();

- 注意事项：
  - 当内部类中定义了静态成员，该内部类必须是 static 的。
  - 当外部类中的静态方法访问内部类时，内部类也必须是 static 的。

7）内部类想调用外部类的成员，需要使用：外部类名.this.成员，即 OutterClassName.this 表示外部类的对象。如果写 this．成员= =成员，调用的还是内部类的成员（属性或方法）。

8）Timer 和 TimerTask：继承 TimerTask 重写 run()方法，再用 Timer 类中的 schedule 方法定时调用，就能自动启用 run()（不像以前似的要用 ．XXX 调用）。

eg：内部类

```
class Xoo{              Timer timer = new Timer();
    public void start(){
        timer.schedule(new MyTask(), 0, 1000);//0 表示立即开始，无延迟
        timer.schedule(new StopTask(), 1000*10);//在 10 秒以后执行一次        }
class StopTask extends TimerTask{
        public void run()  {   timer.cancel();    }//取消 timer 上的任务        }
class MyTask extends TimerTask {
        int i=10;     public void run()  {   System.out.println(i--);    }      }
```

3.17 匿名类

匿名内部类＝＝匿名类

1）匿名内部类的格式： new 父类或者接口(){定义子类的内容}；如 new Uoo(){}就叫匿名内部类！是继承于 Uoo 类的子类或实现 Uoo 接口的子类，并且同时创建了子类型实例，其中{}是子类的类体，可以写类体中的成员。

2）定义匿名内部类的前提：内部类必须是继承一个类或者实现接口。

3）匿名内部类没有类名，其实匿名内部类就是一个匿名子类对象。而且这个对象有点胖。可以理解为带内容的对象。

4）在匿名内部类中只能访问 final 局部变量。

5）匿名内部类中定义的方法最好不要超过 3 个。

eg1：匿名内部类的创建

```
public static void main(String[] args) {
  Uoo u = new Uoo();//创建 Uoo 实例 Uoo u1 = new Uoo(){};//创建匿名内部类实例
  Uoo u2 = new Uoo(){
        public void test() {//方法的重写   System.out.println("u2.test()"); }        };
  u2.test();//调用在匿名内部类中重写的方法。
```

```
        // new Doo();编译错误，不能创建接口实例
        Doo doo = new Doo(){//实现接口，创建匿名内部类实例
        public void test() {//实现接口中声明的抽象方法
                System.out.println("实现 test");   }          };
        doo.test();//调用方法
    }
    interface Doo{ void test();      }
    class Uoo{      public void test(){}   }
```

eg2：匿名内部类中只能访问 final 局部变量

```
    final Timer timer=new Timer();
    timer.schedule(new TimerTask(){
            public void run(){   timer.cancel();//在匿名内部类中只能访问 final 局部变量
            }
    }, 1000*10);
```

6）nonymous Inner Class (匿名内部类) 是否可以 extends(继承)其它类？是否可以 implements(实现)interface(接口)？

匿名内部类是可以继承其它类，同样也可以去实现接口的，用法为：

这样的用法在 swing 编程中是经常使用的，就是因为它需要用到注册监听器机制,而该监听类如果只服务于一个组件,那么将该类设置成内部类/匿名类是最方便的。

3.18 二维数组和对象数组

二维数组（假二维数组），Java 中没有真正的二维数组！Java 二维数组是元素为数组的数组。

对象数组：元素是对象（元素是对象的引用）的数组。

```
Point[] ary;// 声明了数组变量 ary
    ary = new Point[3];// 创建了数组对象
    // new Point[3]实际情况：{null,null,null}
    // 数组元素自动初始化为 null，并不创建元素对象！
    System.out.println(ary[1]);// null
    ary[0] = new Point(3, 4);       ary[1] = new Point(5, 6);       ary[2] = new Point(1, 2);
    System.out.println(ary[1]);// 默认调用了对象的 toString()
    System.out.println(ary[1].toString());//结果上面的一样
//toString 是 Object 类定义，子类继承的方法
//在输出打印对象的时候，会默认调用，重写这个方法可以打印的更好看！
    System.out.println(Arrays.toString(ary));//输出 3 个对象
    System.out.println(ary.toString());//地址值
    int[] c={1,3,5,7};
    System.out.println(c[1]);
    //System.out.println(c[1].toString());//错误，不能在 int 类型调用 toString()
```

3.19 其他注意事项

1）Java 文件规则：

一个 Java 源文件中可以有多个类，但只能有一个公有类！其他类只能是默认类（包中类）而且 Java 的文件夹一定与公有类类名一致！如果没有公有类，可以和任何一个文件名一致。

◆ 一般建议：一个文件一个公有类！一般不在一个文件中写多个类

2）JVM 内存结构堆、栈和方法区分别存储的内容：

JVM 会在其内存空间中开辟一个称为“堆”的存储空间，这部分空间用于存储使用 new 关键字创建的对象。

栈用于存放程序运行过程当中所有的局部变量。一个运行的 Java 程序从开始到结束会有多次方法的调用。JVM 会为每一个方法的调用在栈中分配一个对应的空间，这个空间称为该方法的栈帧。一个栈帧对应一个正在调用中的方法，栈帧中存储了该方法的参数、局部变量等数据。当某一个方法调用完成后，其对应的栈帧将被清除。

方法区该空间用于存放类的信息。Java 程序运行时，首先会通过类装载器载入类文件的字节码信息，经过解析后将其装入方法区。类的各种信息都在方法区保存。

Java SE 核心 I

4.1 Object 类

在 Java 继承体系中，java.lang.Object 类位于顶端（是所有对象的直接或间接父类）。如果一个类没有写 extends 关键字声明其父类，则该类默认继承 java.lang.Object 类。Object 类定义了“对象”的基本行为，被子类默认继承。

1）toString 方法：返回一个可以表示该对象属性内容的字符串。

```
MyObject obj=new MyObject();        String info=obj.toString();        System.out.println(info);
```

A．上例为什么我有 toString 方法？

因为所有的类都继承自 Object，而 toString 方法是 Ojbect 定义的，我们直接继承了这个方法。Object 的 toString 方法帮我们返回一个字符串，这个字符串的格式是固定的：类型@hashcode，这个 hashcode 是一串数字，在 java 中叫句柄，或叫地址（但不是真实的物理地址，是 java 自己的一套虚拟地址，防止直接操作内存的）。

```
public String toString(){//只能用 public，重写的方法访问权限要大于等于父类中方法的权限
        return "这个是我们自己定义的 toString 方法的返回值 MyObject!";      }
```

B．上例为什么要重写 toString 方法？

toString 定义的原意是返回能够描述当前这个类的实例的一串文字，我们看一串 hashcode 没意义，所以几乎是要重写的。

```
public static void main(String[] args){  //System.out.println(toString());//不行！编译错误！
    Point p=new Point(1,2);   System.out.println(p);//输出 p 对象的 toString 方法返回值   }
```

C．上例为何有编译错误？

不能直接使用 toString 方法，因为该方法不是静态的。ava 语法规定：静态方法中不能直接引用非静态的属性和方法，想引用必需创建对象。非静态方法中可以直接引用静态属性和方法。

2）equals 方法：用于对象的“相等”逻辑。

A. 在 Object 中的定义：public boolean equals(Object obj){ return (this==obj); }

由此可见，this==obj 与直接的==（双等于）效果一样，仅仅是根据对象的地址（句柄，那个 hashcode 值）来判断对象是否相等。因此想比较对象与给定对象**内容**是否一致，则必须重写 equals 方法。

B．“==”与 equals 的区别：

用“==”比较对象时，描述的是两个对象是否为同一个对象！**根据地址值**判断。而 equals 方法力图去描述两个**对象的内容**是否相等，内容相等取决于业务逻辑需要，可以自行定义比较规则。

C．equals 方法的重写：如，判断两点是否相等。

```
public boolean equals(Object obj){//注意参数
        /**若给定的对象 obj 的地址和当前对象地址一致，那么他们是同一个对象，equals 方法中有大量的内容比较逻辑时，加上这个判断会节省性能的开销！ */
    if(this == obj){ return true;    }
        /** equals 比较前要进行安全验证，确保给定的对象不是 null！若 obj 是 null，说明该引用变量没有指向任何对象，那么就不能引用 ojb 所指象的对象（因为对象不存在）的属性和方法！若这么做就会引发 NullPointException，空指针异常！ */
    if(obj == null){ return false;    }
```

```
        /**直接将 Object 转为子类是存在风险的！我们不能保证 Object 和我们要比较的对象是同一个类型的这会引发 ClassCastException！我们称为：类造型异常。*/
        /**重写 equals 时第一件要做的事情就是判断给定的对象是否和当前对象为同一个类型，不是同类型直接返回 false，因为不具备可比性！*/
    if(!(obj instanceof Point)){     return false;   }
    Point p=(Point)obj;
        /**不能随便把父类转成子类，因为 Object 是所有类的父类，任何类型都可以传给它所以不能保证 obj 传进来的就是相同类型（点类型）*/
        return this.x==p.x && this.y==p.y;//内容比较逻辑定义                    }
```

4.2 String 类

是字符串类型，是引用类型，是“不可变”字符串，无线程安全问题。在 java.lang.String 中。

◆ 注意事项：String str =“abc”;和 String str=new String(“abc”);的区别！

1）String 在设计之初，虚拟机就对他做了特殊的优化，将字符串保存在虚拟机内部的字符串常量池中。一旦我们要创建一个字符串，虚拟机先去常量池中检查是否创建过这个字符串，如有则直接引用。String 对象因为有了上述的优化，就要保证该对象的内容自创建开始就不能改变！所以对字符串的任何变化都会创建新的对象，而不是影响以前的对象！

2）String 的 equals 方法：两个字符串进行比较的时候，我们通常使用 equals 方法进行比较，字符串重写了 Object 的 equals 方法，用于比较字符串内容是否一致。虽然 java 虚拟机对字符串进行了优化，但是我们不能保证任何时候“==”都成立！

3）编程习惯：当一个字符串变量和一个字面量进行比较的时候，用字面量.equals 方法去和变量进行比较，即：if("Hello".equals(str))因为这样不会产生空指针异常。而反过来用，即：if(str.equals("Hello"))则我们不能保证变量不是 null，若变量是 null，我们在调用其 equals 方法时会引发空指针异常，导致程序退出。若都为变量则 if(str!=null&&str.equals(str1))也可。

4）String 另一个特有的 equals 方法：euqalsIgnoreCase，该方法的作用是忽略大小写比较字符串内容，常用环境：验证码。if("hello".equalsIgnoreCase(str))。

5）String 的基本方法：

①String toLowerCase()：返回字符串的小写形式。如：str.toLowerCase()

②String toUpperCase()：返回字符串的大写形式。如：str.toUpperCase()

③String trim()：去掉字符串两边的空白（空格\t\n\r），中间的不去。如：str.trim()

④boolean startsWith()：判断字符串是否以参数字符串开头。如：str.startsWith("s")

⑤boolean endsWith()：判断字符串是否以参数字符串结尾。如：str.endsWith("s")

⑥int length()：返回字符串字符序列的长度。如：str.length()

eg：如何让 HelloWorld 这个字符串以 hello 开头成立

```
if(str.toLowerCase().startsWith("hello"))//有返回值的才能继续 .  先转成小写再判断
```

6）indexOf 方法（检索）：位置都是从 0 开始的。

①int indexOf(String str)：在给定的字符串中检索 str，返回其第一次出现的位置，找不到则返回-1。

②int indexOf(String str,int from)：在给定的字符串中从 from 位置开始检索 str，返回其第一次出现的位置，找不到则返回-1（包含 from 位置，from 之前的不看）。

eg：查找 Think in Java 中 in 后第一个 i 的位置

```
index=str.indexOf("in");          index=str.indexOf("i",index+"in".length());
//这里对 from 参数加 in 的长度的目的是 从 in 之后的位置开始查找
```

③int lastIndexOf(String str)：在给定的字符串中检索 str，返回其最后一次 出 现 的位置，找不到则返回-1（也可认为从右往左找，第一次出现的位置）。

④int lastIndexOf(String str,int from)：在给定的字符串中从 from 位置开始检索 str，返回其最后一次出现的位置，找不到则返回-1（包含 from 位置，from 之后的不看）。

7）charAt 方法：char charAt(int index)：返回字符串指定位置（index）的字符。

eg：判断是否是回文：上海自来水来自海上

```
boolean tf=true;     for(int i=0;i<str2.length()/2;i++){
//char first=str2.charAt(i);      //char last=str2.charAt(str2.length()-i-1);
if(str2.charAt(i)!=str2.charAt(str2.length()-i-1)){//优化
tf = false;        break;//已经不是回文了，就没有必要再继续检查了   }        }
```

8）substring 方法（子串）：字符串的截取，下标从 0 开始的。

①String substring(int start,int end)：返回下标从 start 开始（包含）到 end 结束的字符串（不包含）。

②String substring(int start)：返回下标从 start 开始（包含）到结尾的字符串。

9）getBytes 方法（编码）：将字符串转换为相应的字节。

①byte[] getBytes()：以当前系统默认的字符串编码集，返回字符串所对应的二进制序列。如：byte[] array=str.getBytes(); System.out.println(Arrays.toString(array));

②byte[] getBytes(String charsetName)：以指定的字符串编码集，返回字符串所对应的二进制序列。这个重载方法需要捕获异常，这里可能引发没有这个编码集的异常，UnsupportedEncodingException，如：str="常"; byte[] bs=info.getBytes("UTF-8");

- 注意事项：
  - Windows 的默认编码集 GBK：英文用 1 个字节描述，汉字用 2 个字节描述；ISO-8859-1 欧洲常用编码集：汉字用 3 个字节描述；GBK 国标；GB2312 国标；UTF-8 编码集是最常用的：汉字用 3 个字节描述。
  - 编码：将数据以特定格式转换为字节；解码：将字节以特定格式转换为数据。
  - String(byte[] bytes, String charsetName) ：通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。如：String str=new String(bs,"UTF-8");

10）split 方法（拆分）：字符串的拆分。

String[] split(String regex)：参数 regex 为正则表达式，以 regex 所表示的字符串为分隔符，将字符串拆分成字符串数组。其中，regex 所表示的字符串不被保留，即不会存到字符串数组中，可理解为被一刀切，消失！

eg：对图片名重新定义，保留图片原来后缀

```
String name="me.jpg";                    String[] nameArray=name.split("\\.");
    //以正则表达式拆分 .有特殊含义，所以用\\. 转义
System.out.println("数组长度："+nameArray.length);//如果不用\\.则长度为 0
System.out.println(Arrays.toString(nameArray));//任意字符都切一刀，都被切没了
String newName="123497643."+nameArray[1];
System.out.println("新图片名："+newName);
```

- 注意事项：分隔符放前、中都没事，放最后将把无效内容都忽略。

```
String str="123,456,789,456,,,";
String[] array=str.split(",");//分隔符放前、中都没事，放最后将把无效内容都忽略
System.out.println(Arrays.toString(array));//[123, 456, 789, 456]
```

11）replace 方法：字符串的替换。

String replaceAll(String regex,String replacement)：将字符串中匹配正则表达式 regex 的字符串替换成 replacement。如：String str1=str.replaceAll("[0-9]+", "chang");

12）String.valueOf()方法：重载的静态方法，用于返回各类型的字符串形式。

```
String.valueOf(1);//整数，返回字符串 1 String.valueOf(2.1);//浮点数，返回字符串 1.2
```

4.3 StringUtils 类

针对字符串操作的工具类，提供了一系列静态方法，在 Apache 阿帕奇 Commons-lang 包下中，需下载。

StringUtils 常用方法：

1）String repeat(String str,int repeat)：重复字符串 repeat 次后返回。

2）String join(Object[] array,String)：将一个数组中的元素连接成字符串。

3）String leftPad(String str,int size,char padChar)：向左边填充指定字符 padChar，以达到指定长度 size。

4）String rightPad(String str,int size,char padChar)：向右边填充指定字符 padChar，以达到指定长度 size。

4.4 StringBuilder 类

与 String 对象不同，StringBuilder 封装“可变”的字符串，有线程安全问题。对象创建后，可通过调用方法改变其封装的字符序列。

StringBuilder 常用方法：

1）追加字符串：StringBuilder append(String str)：

2）插入字符串：StringBuilder insert(int index,String str)：插入后，原内容依次后移

3）删除字符串：StringBuilder delete(int start,int end)：

4）替换字符串：StringBuilder replace(int start,int end,String str)：含头不含尾

5）字符串反转：StringBuilder reverse()：

eg：各类操作

```
StringBuilder builder=new StringBuilder();
builder.append("大家好！") .append("好好学习") .append("天天向上");
                                    //返回的还是自己：builder，所以可以再 .
System.out.println(builder.toString());
builder.insert(4, "！");          System.out.println(builder.toString());
builder.replace(5,9,"Good Good Study!");          System.out.println(builder.toString());
builder.delete(9, builder.length());        System.out.println(builder.toString());
```

- 注意事项：
  - 该类用于对某个字符串频繁的编辑操作，使用 StringBuilder 可以在大规模修改字符串时，不开辟新的字符串对象，从而节约内存资源，所以，对有着大量操作字符串的逻辑中，不应使用 String 而应该使用 StringBuilder。
  - append 是有返回值的，返回类型是 StringBuilder，而返回的 StringBuilder 其实就是自己（this），append 方法的最后一句是 return this；
  - StringBuilder 与 StringBuffer 区别：效果是一样的。
    StringBuilder 是线程不安全的，效率高，需 JDK1.5+。
    StringBuffer 是线程安全的，效率低，“可变”字符串。
    在多线程操作的情况下应使用 StringBuffer，因为 StringBuffer 是线程安全

的，他难免要顾及安全问题，而进行必要的安全验证操作。所以效率上要比 StringBuilder 低，根据实际情况选择。

4.5 正则表达式

实际开发中，经常需要对字符串数据进行一些复杂的匹配、查找、替换等操作，通过正则表达式，可以方便的实现字符串的复杂操作。

正则表达式是一串特定字符，组成一个“规则字符串”，这个“规则字符串”是描述文本规则的工具，正则表达式就是记录文本规则的代码。

[]	表示一个字符
[abc]	表示 a、b、c 中任意一个字符
[^abc]	除了 a、b、c 的任意一个字符
[a-z]	表示 a 到 z 中的任意一个字符
[a-zA-Z0-9_]	表示 a 到 z、A 到 Z、0 到 9 以及下滑线中的任意一个字符
[a-z&&[^bc]]	表示 a 到 z 中除了 b、c 之外的任意一个字符，&&表示“与”的关系
.	表示任意一个字符
\d	任意一个数字字符，相当于[0-9]
\D	任意一个非数字字符，相当于[^0-9]
\s	空白字符，相当于[\t\n\f\r\x0B]
\S	非空白字符，相当于[^\s]
\w	任意一个单词字符，相当于[a-zA-Z0-9_]
\W	任意一个非单词字符，相当于[^\w]
^	表示字符串必须以其后面约束的内容开始
$	表示字符串必须以其前面约束的内容结尾
?	表示前面的内容出现 0 到 1 次
*	表示前面的内容出现 0 到多次
+	表示前面的内容出现 1 到多次
{n}	表示前面的字符重复 n 次
{n,}	表示前面的字符至少重复 n 次
{n,m}	表示前面的字符至少重复 n 次，并且小于 m 次　X>=n && X<m

- 注意事项：
  - 邮箱格式的正则表达式 @无特殊含义，可直接写，也可[@]
  - 使用 Java 字符串去描述正则表达式的时候，会出现一个冲突，即如何正确描述正则表达式的“.”。

    起因：在正则表达式中我们想描述一个“.”，但“.”在正则表达式中有特殊含义，他代表任意字符，所以我们在正则表达式中想描述“.”的愿义就要写成“\.”但是我们用 java 字符串去描述正则表达式的时候，因为“.”在 java 字符串中没有特殊意义，所以 java 认为我们书写 String s="\. ";是有语法错误的，因为“.”不需要转义，这就产生了冲突。

    处理：我们实际的目的很简单，就是要让 java 的字符串描述"\. "又因为在 java 中"\"是有特殊含义的，代表转义字符我们只需要将"\"转义为单纯的斜杠，即可描述"\. "了所以我们用 java 描述“\.”的正确写法是 String s="\\. ";
  - 若正则表达式不书写^或$，正则表达式代表匹配部分内容，都加上则表示权匹配

eg：测试邮箱正则表达式：Pattern 的作用是描述正则表达式的格式支持，使用静态方

法 compile 注册正则表达式，生成实例。

```
String regStr="^[a-zA-Z0-9_]+@[a-zA-Z0-9]+(\\.com|\\.cn|\\.net)+$";
    Pattern pattern=Pattern.compile(regStr);//注册正则表达式
    String mailStr="chang_2013@chang.com.cn";
    //匹配字符串，返回描述匹配结果的 Matcher 实例
    Matcher matcher=pattern.matcher(mailStr);
    //通过调用 Matcher 的 find 方法得知是否匹配成功
    if(matcher.find()){  System.out.println("邮箱匹配成功！");          }
    else{System.out.println("邮箱格式错误！");                  }
```

4.6 Date 类

java.util.Date 类用于封装日期及时间信息，一般仅用它显示某个日期，不对他作任何操作处理，作处理用 Calendar 类，计算方便。

```
Date date=new Date();//创建一个 Date 实例，默认的构造方法创建的日期代表当前系统时间
System.out.println(date);//只要输出的不是类名@hashcode 值，就说明它重写过 toString()
long time=date.getTime();//查看 date 内部的毫秒值
date.setTime(time+1000*60*60*24);//设置毫秒数让一个时间 Date 表示一天后的当前时间
int year=date.getYear();//画横线的方法不建议再使用：1、有千年虫问题。2、不方便计算
```

4.7 Calendar 类

java.util.Calendar 类用于封装日历信息，其主作用在于其方法可以对时间分量进行运算。

1）通过 Calendar 的静态方法获取一个实例该方法会根据当前系统所在地区来自行决定时区，帮我们创建 Calendar 实例，这里要注意，实际上根据不同的地区，Calendar 有若干个子类实现。而 Calendar 本身是抽象类，不能被实例化！我们不需要关心创建的具体实例为哪个子类，我们只需要根据 Calendar 规定的方法来使用就可以了。

2）日历类所解决的根本问题是简化日期的计算，要想表示某个日期还应该使用 Date 类描述。Calendar 是可以将其描述的时间转化为 Date 的，我们只需要调用其 getTime()方法就可以获取描述的日期的 Date 对象了。

3）通过日历类计算时间：为日历类设置时间，日历类设置时间使用通用方法 set。

set(int field,int value)，field 为时间分量，Calendar 提供了相应的常量值，value 为对应的值。

4）只有月份从 0 开始：0 为 1 月，以此类推，11 为 12 月，其他时间是正常的从 1 开始。也可以使用 Calendar 的常量 calendar.NOVEMBER……等.

5）Calendar.DAY_OF_MONTH　　月里边的天---号；

Calendar.DAY_OF_WEEK　星期里的天---星期几

Calendar.DAY_OF_YEAR　年里的天

```
Calendar calendar=Calendar.getInstance();//构造出来表示当前时间的日历类
Date now=calendar.getTime();//获取日历所描述的日期
calendar.set(Calendar.YEAR, 2012);//设置日历表示 2012 年
calendar.set(Calendar.DAY_OF_MONTH,15);//设置日历表示 15 号
calendar.add(Calendar.DAY_OF_YEAR, 22);//想得到 22 天以后是哪天
calendar.add(Calendar.DAY_OF_YEAR, -5);//5 天以前是哪天
calendar.add(Calendar.MONTH, 1);得到 1 个月后是哪天
System.out.println(calendar.getTime());
```

6）获取当前日历表示的日期中的某个时间单位可以使用 get 方法.

```
int year=calendar.get(Calendar.YEAR);
int month=calendar.get(Calendar.MONTH);
int day=calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(year+"年"+(month+1)+"月"+day+"日");//month 要处理
```

4.8 SimpleDateFormat 类

java.text.SimpleDateFormat 类，日期转换类，该类的作用是可以很方便的在字符串和日期类之间相互转换。

1）这里我们在字符串与日期类间相互转换是需要一些约束的，"2012-02-02"这个字符串如何转换为 Date 对象？Date 对象又如何转为字符串？

parse 方法用于按照特定格式将表示时间的字符串转化成 Date 对象。

format 方法用于将日期数据（对象）按照指定格式转为字符串。

2）常用格式字符串

字符	含义	示例
y	年	yyyy 年-2013 年；yy 年-13 年
M	月	MM 月-01 月；M 月-1 月
d	日	dd 日-06 日；d 日-6 日-
E	星期	E-星期日（Sun）
a	AM 或 PM	a-下午（PM）
H	24 小时制	a h 时-小午 12 时 HH：mm：ss-12：46：33 hh(a)：mm：ss-12（下午）：47：48
h	12 小时制	
m	分钟	
s	秒	

eg：字符串转成 Date 对象

```
//创建一个 SimpleDateFormat 并且告知它要读取的字符串格式
SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd");
String dateFormat="2013-05-14";//创建一个日期格式字符串
//将一个字符串转换为相应的 Date 对象
Date date=sdf.parse(dateFormat);//要先捕获异常
System.out.println(date);//输出这个 Date 对象
```

eg：Date 对象转成字符串

```
SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd");
Date now=new Date();        String nowStr=sdf.format(now);//把日期对象传进去
```

3）在日期格式中 - 和 空格 无特殊意义，无特殊含义的都将原样输出。

eg：将时间转为特定格式

```
//将当前系统时间转换为 2012/05/14 17:05:22 的效果
SimpleDateFormat format1=new SimpleDateFormat("yyyy/MM/dd hh:mm:ss");
nowStr=format1.format(now);            System.out.println(nowStr);
```

4.9 DateFormat 类

java.text.DateFormat 类（抽象类）是 SimpleDateFormat 类的父类，用的少，没 SimpleDateFormat 灵活。

创建用于将 Date 对象转换为日期格式的字符串的 DateFormat，创建 DateFormat 对象的实例，使用静态方法 getDateInstance(style,aLocale)，style 为输出日期格式的样式：DateFormat

有对应的常量；aLocale 为输出的地区信息，影响字符串的语言和表现形式。

```
Date now=new Date();      DateFormat format=DateFormat.getDateInstance(
                                    DateFormat.MEDIUM,Locale.CHINA);
```

4.10 包装类

Java 语言的 8 种基本类型分别对应了 8 种“包装类”。每一种包装类都封装了一个对应的基本类型成员变量，同时还提供了针对该数据类型的实用方法。

1）包装类的目的：用于将基本类型数据当作引用类型看待。

2）包装类的名字：除了 Integer(int)，Character(char)外，其余包装类名字都是基本类型名首字母大写。

3）拆、装箱：Integer i=new Integer(1);创建一个以对象形式存在的整数 1，这种从基本类型转为引用类型的过程称之为“装箱”，反之叫“拆箱”。

4）装箱：方式一：Double d=new Double(2.2);//装箱

方式二：Double d=Double.valueOf(2.2);//基本类型都有 valueOf 方法

5）拆箱：double num=d.doubleValue();//拆箱

6）包装类使用前提：JDK1.5+

```
public static void say(Object obj){      System.out.println(obj);      }
int a=1;//基本类型，不是 Object 子类！
say(a);//在 java 1.4 版本的时候，这里还是语法错误的！因为 int 是基本类型，不是 Object 对象，要自己写 8 种基本类型对应的方法
```

7）包装类的使用：实例化一个对象，该对象代表整数 1；Integer 的作用是让基本类型 int 作为一个引用类型去看待。这样就可以参与到面向对象的编程方式了。由此我们可以将一个 int 当作一个 Object 去看待了，也成为了 Object 的子类。

```
Integer i=new Integer(a);//装箱，或者写 Integer i=new Integer(1);
Integer ii=Integer.valueOf(a);//装箱另一种方式
int num=i.intValue();//拆箱      say(i);//Integer 是 Object 的子类，可以调用！
```

8）JDK1.5 包装类自动拆装箱（原理）：在编译源程序的时候，编译器会预处理，将未作拆箱和装箱工作的语句自动拆箱和装箱。可通过反编译器发现。

```
say(Integer.valueOf(a));自动装箱      num=i;//引用类型变量怎么能复制给基本类型呢?
                                    //num=i.intValuse();//自动拆箱
```

9）包装类的一些常用功能：将字符串转换为其类型，方法是：parseXXX，XXX 代表其类型。这里要特别注意！一定要保证待转换的字符串描述的确实是或者兼容要转换的数据类型！否则会抛出异常！

```
String numStr="123";      System.out.println(numStr+1);//1231
int num=Integer.parseInt(numStr);      System.out.println(num+1)//124
long longNum=Long.parseLong(numStr);      System.out.println(longNum);//123
double doubleNum=Double.parseDouble(numStr);
System.out.println(doubleNum);//123.0
```

10）Integer 提供了几个有趣的方法：将一个整数转换为 16 进制的形式，并以字符串返回；将一个整数转换为 2 进制的形式，并以字符串返回。

```
String bStr=Integer.toBinaryString(num);      String hStr=Integer.toHexString(num);
```

11）所有包装类都有几个共同的常：获取最大、最小值。

```
int max=Integer.MAX_VALUE;//int 最大值      int min=Integer.MIN_VALUE;//int 最小值
System.out.println(Integer.toBinaryString(max));  System.out.println(Integer.toBinaryString(min));
```

4.11 BigDecimal 类

表示精度更高的浮点型，在 java.math.BigDecimal 包下，该类可以进行更高精度的浮点运算。需要注意的是，BigDecimal 可以描述比 Double 还要高的精度，所以在转换为基本类型时，可能会丢失精度！

1）BigDecimal 的使用：创建一个 BigDecimal 实例，可以使用构造方法 BigDecimal（String numberFormatString）用字符串描述一个浮点数作为参数传入。

```
BigDecimal num1=new BigDecimal("3.0");
BigDecimal num2=new BigDecimal("2.9");  //运算结果依然为 BigDecimal 表示的结果
BigDecimal result=num1.subtract(num2);//num1-num2     System.out.println(result);
float f=result.floatValue();//将输出结果转换为基本类型 float
int i=result.intValue();//将输出结果转换为基本类型 int
```

2）BigDecimal 可以作加 add、减 subtract、乘 multiply、除 divide 等运算：这里需要注意除法，由于除法存在结果为无限不循环小数，所以对于除法而言，我们要制定取舍模式，否则会一直计算下去，直到报错（内存溢出）。

```
result=num1.divide(num2,8,BigDecimal.ROUND_HALF_UP);
                                                    //小数保留 8 位，舍去方式为四舍五入
```

4.12 BigInteger 类

使用描述更长位数的整数“字符串”，来表示、保存更长位数的整数，在 java.math.BigInteger 包下。

1）BigInteger 的使用：创建 BigInteger

```
BigInteger num=new BigInteger("1");//num=new BigInteger(1);不可以，没有这样的构造器
//这种方式我们可以将一个整数的基本类型转换为 BigInteger 的实例
num=BigInteger.valueOf(1);
```

2）理论上：BigInteger 存放的整数位数只受内存容量影响。

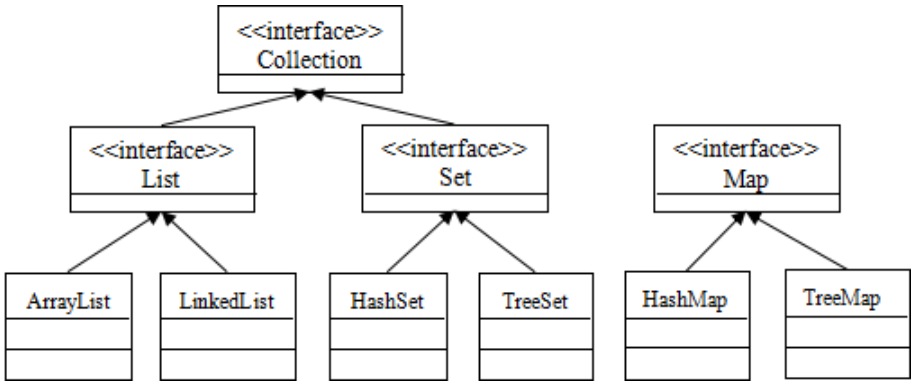
3）BigInteger 同样支持加 add、减 subtract、乘 multiply、除 divide 等运算。

eg：1-200 的阶乘

```
for(int i=1;i<=200;i++){        num=num.multiply(BigInteger.valueOf(i));    }
System.out.println("结果"+num.toString().length()+"位");    System.out.println(num);
```

4.13 Collection 集合框架

在实际开发中，需要将使用的对象存储于特定数据结构的容器中。而 JDK 提供了这样的容器——集合框架，集合框架中包含了一系列不同数据结构（线性表、查找表）的实现类。



1）Collection 常用方法：

①int size()：返回包含对象个数。　　②boolean isEmpty()：返回是否为空。

③boolean contains(Object o)：判断是否包含指定对象。

④void clear()：清空集合。　　⑤boolean add(E e)：向集合中添加对象。

⑥boolean remove(Object o)：从集合中删除对象。

⑦boolean addAll(Collection<? extends E > c)：另一个集合中的所有元素添加到集合

⑧boolean removeAll(Collection<?> c)：删除集合中与另外一个集合中相同的原素

⑨Iterator<E> iterator()：返回该集合的对应的迭代器

2）Collection 和 Collentions 的区别

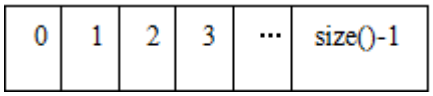
Collection 是 java.util 下的接口，它是各种集合的父接口，继承于它的接口主要有 Set 和 List；Collections 是个 java.util 下的类，是针对集合的帮助类，提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

4.14 List 集合的实现类 ArrayList 和 LinkedList

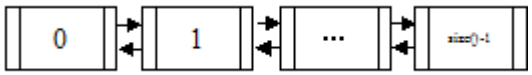
List 接口是 Collection 的子接口，用于定义线性表数据结构，元素可重复、有序的；可以将 List 理解为存放对象的数组，只不过其元素个数可以动态的增加或减少。

1）List 接口的两个常见的实现类：ArrayList 和 LinkedList，分别用动态数组和链表的方式实现了 List 接口。List、ArrayList 和 LinkedList 均处于 java.util 包下。

2）可以认为 ArrayList 和 LinkedList 的方法在逻辑上完全一样，只是在性能上有一定的差别，ArrayList 更适合于随机访问，而 LinkedList 更适合于插入和删除，在性能要求不是特别苛刻的情形下可以忽略这个差别。



ArrayList



LinkedList

3）使用 List 我们不需要在创建的时候考虑容量集合的容量是根据其所保存的元素决定的换句话说，集合的容量是可以自动扩充的。

4）List 的实现类会重写 toString 方法，依次调用所包含对象的 toString 方法，返回集合中所包含对象的字符串表现。

5）常用方法：

①add(Object obj)：向想集合末尾追加一个新元素，从该方法的参数定义不难看出，集合可以存放任意类型的元素，但在实际编程中我们发现，几乎不会向集合中存放一种以上的不同类型的元素。

②size()方法：返回当前集合中存放对象的数量。

③clear()方法：用于清空集合。

④isEmpty()方法：用于返回集合是否为空。

```
List list=new ArrayList();  list.add("One");  list.add("Two");  list.add("Three");
//list.add(1);//不建议这样的操作！尽量不在同一个集合中存放不用类型元素
System.out.println("集合中元素的数量："+list.size());
System.out.println(list);//System.out.println(list.toString());
//ArrayList 重写了 toString()方法返回的字符串是每个元素的 toString()返回值的序列
list.clear();//清空  System.out.println("清空后元素的数量："+list.size());
System.out.println("集合是否为空？："+list.isEmpty());
```

⑤contains(Object obj)方法：检查给定对象是否被包含在集合中，检查规则是将 obj 对象与集合中每个元素进行 equals 比较，若比对了所有元素均没有 equals 为 true 的则返回

false。

- 注意事项：根据情况重写 equals：若比较是否是同一个对象，则不需要重写，直接用 contains 里的 equals 比较即可。若重写 equals 为内容是否相同，则按内容比较，不管是否同一个对象。是否重写元素的 equals 方法对集合的操作结果有很大的效果不同！

⑥boolean remove(Object obj)方法：删除一个元素，不重写 equals，不会有元素被删除（因为比较的是对象的地址，都不相同），重写 equals 为按内容比较，则删除第一个匹配的就退出，其他即使内容相同也不会被删除。

```
List list=new ArrayList();//多态的写法  //ArrayList arrayList=new ArrayList();//正常的写法
list.add(new Point(1,2)); //Point point =new Point(1,2);     list.add(point);等量代换
list.add(new Point(3,4));        list.add(new Point(5,6));
System.out.println("集合中元素的数量："+list.size());       System.out.println(list);
Point p=new Point(1,2);//创建一个 Point 对象
System.out.println("p 在集合中存在么？"+list.contains(p));//不重写为 false    重写为 true
System.out.println("删前元素："+list.size());
list.remove(p);//将 p 对象删除，不重写 equals，不会有元素被删除
System.out.println("删后元素："+list.size());              System.out.println(list);
```

⑦E remove(int index)方法：移除此列表中指定位置上的元素。向左移动所有后续元素（将其索引减 1）。因此在做删除操作时集合的大小为动态变化的，为了防止漏删，必须从后往前删！

```
ArrayList list=new ArrayList(); list.add("java");      list.add("aaa");
        list.add("java");      list.add("java");      list.add("bbb");
//相邻的元素删不掉！
for(int i=0;i<list.size();i++){   if( "java".equals(list.get(i)));     list.remove(i);          }
//可以删干净！
for(int i=list.size()-1;i>=0;i--){ if("java".equals(list.get(i))){    list.remove(i);        }
```

⑧addAll(Collection c)方法：允许将 c 对应的集合中所有元素存入该集合，即并集。注意，这里的参数为 Collection，所以换句话说，任何集合类型都可以将其元素存入其他集合中！

⑨removeAll(Collection c)方法：删除与另一个集合中相同的元素。它的“相同”逻辑通过 equals 方法来判断。

⑩retainAll(Collection c)方法：保留与另一个集合中相同的元素，即交集。它的“相同”逻辑通过 equals 方法来判断。

```
list1.addAll(list2);//并集
list1.removeAll(list3);//从 list1 中删除 list3 中相同(equals 为 true)的元素
list1.retainAll(list2);//保留 list1 中删除 list2 中相同(equals 为 true)的元素
```

⑪Object get(int index)方法：根据元素下标获取对应位置的元素并返回，这里元素的下标和数组相似。

⑫Object set(int index,Object newElement)方法：将 index 位置的元素修改为 newElement 修改后会将被修改的元素返回。因此，可实现将 List 中第 i 个和第 j 个元素交换的功能：list.set (i , list.set (j , list.get (i))) ;

⑬add(int index,Object newElement)方法：使用 add 的重载方法，我们可以向 index 指定位置插入 newElement，原位置的元素自动向后移动，即所谓的“插队”。

⑭Object remove(int index)方法：将集合中下标为 index 的元素删除，并将被删除的

元素返回（不根据 equals，根据下标删除元素）。

```
List list=new ArrayList();  list.add("One");  list.add("Two");  list.add("Three");
//因为 get 方法是以 Object 类型返回的元素，所以需要造型，默认泛型 Object
String element=(String)list.get(2);//获取第三个元素  System.out.println(element);
for(int i=0;i<list.size();i++){//遍历集合  System.out.println(list.get(i));  }
Object old=list.set(2, "三");
System.out.println("被替换的元素："+old);  System.out.println(list);
list.add(2, "二");//在 Two 与“三”之间插入一个“二”  System.out.println(list);
Object obj=list.remove(1);  System.out.println("被删除的元素："+obj);
```

⑮indexOf(Object obj)方法：用于在集合中检索对象，返回对象第一次出现的下标。

⑯lastIndexOf(Object obj)方法：用于在集合中检索对象，返回对象最后一次出现的下标。

⑰Object[] toArray()方法：该方法继承自 Collection 的方法，该方法会将集合以对象数组的形式返回。

⑱toArray()的重载方法，T[] toArray(T[] a)：可以很方便的让我们转换出实际的数组类型，如下例，参数 new Point[0]的作用是作为返回值数组的类型，所以参数传入的数组不需要有任何长度，因为用不到，就没有必要浪费空间。

```
Object[] array=list.toArray();//将集合以对象数组的形式返回
for(int i=0;i<array.length;i++){  Point p=(Point)array[i];  System.out.println(p.getX()); }
Point[] array1=(Point[])list.toArray(new Point[0]);//toArray()的重载方法
for(int i=0;i<array1.length;i++){  Point p=array1[i];//不需要每次都强转了
                                   System.out.println(p.getX());  }
```

⑲List<E> subList(int fromIndex, int toIndex)方法：获取子集合，但在获取子集后，若对子集合的元素进行修改，则会影响原来的集合。

```
List<Integer> list=new ArrayList<Integer>();
for(int i=0;i<10;i++){  list.add(i);  }
List<Integer> subList=list.subList(3, 8);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  取子集(3-7)
System.out.println(subList);//[3, 4, 5, 6, 7]
//我们在获取子集后，若对自己元素进行修改，会影响原来的集合
for(int i=0;i<subList.size();i++){  int element=subList.get(i);
        element*=10;  subList.set(i, element);
        //subList.set(i, subList.get(i)*10);同上三步  }
System.out.println(list);//原集合内容也被修改了
```

⑳Comparable 接口：针对对象数组或者集合中的元素进行排序时，首选需要确定对象元素的“比较”逻辑（即哪个大哪个小）。Comparable 接口用于表示对象间的大小关系，我们需要实现 Comparable 接口，并重写 compareTo()方法定义比较规则。

```
public int compareTo(ComparablePoint o){  int r=x*x+y*y;//自身点到原点的距离
    int other=o.x*o.x+o.y*o.y;//参数点到原点的距离
//返回结果大于 0，自身比参数大；小于 0，自身比参数小；等于 0，自身和参数相等；
    //equals 返回 true 的时候，comparTo 的返回值应该为 0
        return r-other;  }
```

㉑Collections.sort()方法：需要集合中的对象实现 Comparable 接口，从而可以调用其 compareTo 方法判断对象的大小，否则 sort 将无法判断。该方法会依次调用集合中每个元素的 compareTo 方法，并进行自然排序（从小到大）。

```
List<ComparablePoint> list=new ArrayList<ComparablePoint>();
list.add(new ComparablePoint(1,5));      list.add(new ComparablePoint(3,4));
list.add(new ComparablePoint(2,2));   System.out.println(list);//输出顺序与存方时一致
            Collections.sort(list);            System.out.println(list);
```

㉒Comparator 接口：比较器。一旦 Java 类实现了 Comparable，其比较逻辑就已经确定了，如果希望在排序中的操作按照“临时指定规则”，即自定义比较规则。可以采用 Comparator 接口回调方式。

Comparator 比较器创建步骤：A.定义一个类并实现 Comparator 接口。B.实现接口中的抽象方法 compare(E o1,E o2)。C.实例化这个比较器 D.调用 Collections 的重载方法：sort(Collection c,Comparator comparator)进行排序。通常使用匿名类方式创建一个实例来定义比较器。

```
Comparator<ComparablePoint> c=new Comparator<ComparablePoint>(){
      public int compare(ComparablePoint   o1,ComparablePoint o2){
            return o1.getX()-o2.getX();//两个点的 X 值大的大          }          };
Collections.sort(list, c);          System.out.println(list);
```

4.15 Iterator 迭代器

所有 Collection 的实现类都实现了 iterator 方法，该方法返回一个 Iterator 接口类型的对象，用于实现对集合元素迭代的便利。在 java.util 包下。

1）Iterator 定义有三个方法：

①boolean hasNext()方法：判断指针后面是否有元素。

②E next()方法：指针后移，并返回当前元素。E 代表泛型，默认为 Object 类型。

③void remove()方法：在原集合中删除刚刚返回的元素。

2）对于 List 集合而言，可以通过基于下标的 get 方法进行遍历；而 iterator 方法是针对 Collection 接口设计的，所以，所有实现了 Collection 接口的类，都可以使用 Iterator 实现迭代遍历。

3）迭代器的使用方式：先问后拿。问：boolean hasNext()该方法询问迭代器当前集合是否还有元素；拿：E next()该方法会获取当前元素。迭代器的迭代方法是 while 循环量身定制的。

```
List list=new ArrayList(); list.add("One");          list.add("#");
Iterator it=list.iterator();
while(it.hasNext()){//集合中是否还有下一个元素
            Object element=it.next();//有就将其取出
            System.out.println(element);          }
```

4）迭代器中的删除问题：在迭代器迭代的过程中，我们不能通过“集合”的增删等操作，来改变该集合的元素数量！否则会引发迭代异常！若想删除迭代出来的元素，只能通过 Iterator。迭代器在使用自己的 remove()方法时，可以将刚刚获取的元素从集合中删除，但是不能重复调用两次！即在不迭代的情况下，不能在一个位置删两次。

```
while(it.hasNext()){//集合中是否还有下一个元素
String element=(String)it.next();//有就将其取出，next 返回值为 E（泛型）默认为 Object 所以需要强转
if("#".equals(element)){ //list.remove(element);不可以！
            it.remove();//删除当前位置元素          }          }
```

4.16 泛型

1）泛型是 JDK1.5 引入的新特性，泛型的本质是参数化类型。在类、接口、方法的定义过程中，所操作的数据类型为传入的指定参数类型。所有的集合类型都带有泛型参数，这样在创建集合时可以指定放入集合中的对象类型。同时，编译器会以此类型进行检查。

2）ArrayList 支持泛型，泛型尖括号里的符号可随便些，但通常大写 E。

3)迭代器也支持泛型,但是迭代器使用的泛型应该和它所迭代的集合的泛型类型一致！

4）泛型只支持引用类型，不支持基本类型，但可以使用对应的包装类

5）如果泛型不指定类型的话，默认为 Object 类型。

```
ArrayList<Point> list=new ArrayList<Point>();
list.add(new Point(1,2)); list.add(new Point(3,4));
//list.add("哈哈");//定义泛型后，只运行 Point 类型，否则造型异常
for(int i=0;i<list.size();i++){    Point p=/*(Point)也不需要强转造型了*/list.get(i);
                                   System.out.println(p.getX());                    }
Iterator<Point> it=list.iterator();    while(it.hasNext()){ Point p=it.next();
                          //也不需要强转了          System.out.println(p);          }
```

6）自定义泛型

```
Point p=new Point(1,2);//只能保存整数
//把 Point 类的 int 都改成泛型 E，或者也可设置多个泛型 Point<E,Z>
Point<Double> p1=new Point<Double>(1.0,2.3);//设置一个泛型
Point<Double,Long> p2=new Point<Double,Long>(2.3,3L);//设置多个泛型
```

4.17 增强型 for 循环

JDK 在 1.5 版本推出了增强型 for 循环，可以用于数组和集合的遍历。

◆ 注意事项：集合中要有值，否则直接退出（不执行循环）。

1）老循环：自己维护循环次数， 循环体自行维护获取元素的方法。

```
int[] array=new int[]{1,2,3,4,5,6,7};
for(int i=0;i<array.length;i++){//维护循环次数
    int element=array[i];//获取数组元素          System.out.print(element);          }
```

2）新循环：自动维护循环次数（由遍历的数组或集合的长度决定），自动获取每次迭代的元素。

```
int[] array=new int[]{1,2,3,4,5,6,7};
for(int element:array){          System.out.print(element);                }
```

3）新循环执行流程：遍历数组 array 中的每个元素，将元素一次赋值给 element 后进入循环体，直到所有元素均被迭代完毕后退出循环。

◆ 注意事项：使用新循环，element 的类型应与循环迭代的数组或集合中的元素类型一致！至少要是兼容类型！

4）新循环的内部实现是使用迭代器完成的 Iterator。

5）使用新循环遍历集合：集合若使用新循环，应该为其定义泛型，否则我们只能使用 Object 作为被接收元素时的类型。通常情况下，集合都要加泛型，要明确集合中的类型，集合默认是 Object。

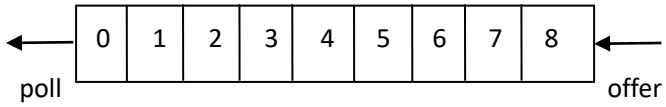
```
ArrayList<String> list=new ArrayList<String>();
list.add("张三");   list.add("李四");   list.add("王五");
for(String str:list){//默认是 Object 自己加泛型 String      System.out.println(str);      }
```

4.18 List 高级－数据结构：Queue 队列

队列（Queue）是常用的数据结构，可以将队列看成特殊的线性表，队列限制了对线性表的访问方式：只能从线性表的一端添加（offer）元素，从另一端取出（poll）元素。Queue 接口：在包 java.util.Queue。

1）队列遵循先进先出原则：FIFO（First Input First Output）队列不支持插队，插队是不道德的。

2）JDK 中提供了 Queue 接口，同时使得 LinkedList 实现了该接口（选择 LinkedList 实现 Queue 的原因在于 Queue 经常要进行插入和删除的操作，而 LinkedList 在这方面效率较高）。



3）常用方法：

①boolean offer(E e)：将一个对象添加至队尾，如果添加成功则返回 true。

②poll()：从队列中取出元素，取得的是最早的 offer 元素，从队列中取出元素后，该元素会从队列中删除。若方法返回 null 说明 队列中没有元素了。

③peek()：获取队首的元素（不删除该元素!）

eg：队列相关操作

```
Queue<String> queue=new LinkedList<String>();
queue.offer("A");       queue.offer("B");       queue.offer("C");
System.out.println(queue);//[A, B, C]
System.out.println("队首："+queue.peek());//获取队首元素，但不令其出队
String   element=null;   while((element=queue.poll())!=null){
                                        System.out.println(element);          }
```

4.19 List 高级－数据结构：Deque 栈

栈（Deque）是常用的数据结构，是 Queue 队列的子接口，因此 LinkedList 也实现了 Deque 接口。栈将双端队列限制为只能从一端入队和出队，对栈而言即是入栈和出栈。子弹夹就是一种栈结构。在包 java.util.Deque 下。

1）栈遵循先进后出的原则：FILO(First Input Last Output)。

2）常用方法：

①push:压入，向栈中存入数据。

②pop:弹出，从栈中取出数据。

③peek：获取栈顶位置的元素，但不取出。

- 注意事项：我们在使用 pop 获取栈顶元素之前，应现使用 peek 方法获取该元素，确定该元素不为 null 的情况下才应该将该元素从栈中弹出”，否则若栈中没有元素后，我们调用 pop 会抛出异常“NoSuchElementException”。

eg：栈相关操作

```
Deque<Character> deque=new LinkedList<Character>();
for(int i=0;i<5;i++){ deque.push((char)('A'+i));          }
System.out.println(deque);
//注意使用 peek 判断栈顶是否有元素
while(deque.peek()!=null){          System.out.print(deque.pop()+" ");
```

```
}
```

4.20 Set 集合的实现类 HashSet

Set 是无序，用于存储不重复的对象集合。在 Set 集合中存储的对象中，不存在两个对象 equals 比较为 true 的情况。

1）HashSet 和 TreeSet 是 Set 集合的两个常见的实现类，分别用 hash 表和排序二叉树的方式实现了 Set 集合。HashSet 是使用散列算法实现 Set 的。

2）Set 集合没有 get(int index)方法，我们不能像使用 List 那样，根据下标获取元素。想获取元素需要使用 Iterator。

3）向集合添加元素也使用 add 方法，但是 add 方法不是向集合末尾追加元素，因为无序。

4）宏观上讲：元素的顺序和存放顺序是不同的，但是在内容不变的前提下，存放顺序是相同的，但在我们使用的时候，要当作是无序的使用。

```
Set<String> set=new HashSet<String>();//多态
//也可 HashSet<String> set=new HashSet<String>();
set.add("One");    set.add("Two");    set.add("Three");    Iterator<String>
it=set.iterator();
while(it.hasNext()){ String element=it.next();    System.out.print(element+" ");    }
for(String element:set){    System.out.print(element+" ");    }//新循环遍历 Set 集合
```

5）hashCode 对 HashSet 的影响：若我们不重写 hashCode，那么使用的就是 Object 提供的，而该方法是返回地址（句柄）！换句话说，就是不同的对象，hashCode 不同。

6）对于重写了 equals 方法的对象，强烈要求重写继承自 Object 类的 hashCode 方法的，因为重写 hashCode 方法与否会对集合操作有影响！

7）重写 hashCode 方法需要注意两点：

①与 equals 方法的一致性，即 equals 比较返回为 true 的对象其 hashCode 方法返回值应该相同。

②hashCode 返回的数值应该符合 hash 算法要求，如果有很多对象的 hashCode 方法返回值都相同，则会大大降低 hash 表的效率。一般情况下，可以使用 IDE（如 Eclipse）提供的工具自动生成 hashCode 方法。

8）boolean contains(Object o)方法：查看对象是否在 set 中被包含。下例虽然有新创建的对象，但是通过散列算法找到了位置后，和里面存放的元素进行 equals 比较为 true，所以依然认为是被包含的（重写 equals 了时）。

```
Set<Point> set=new HashSet<Point>();    set.add(new Point(1,2));
set.add(new Point(3,4));    System.out.println(set.contains(new Point(1,2)));
```

9）HashCode 方法和 equals 方法都重写时对 hashSet 的影响：将两个对象同时放入 HashSet 集合，发现存在，不再放入（不重复集）。当我们重写了 Point 的 equals 方法和 hashCode 方法后，我们发现虽然 p1 和 p2 是两个对象，但是当我们将他们同时放入集合时，p2 对象并没有被添加进集合。因为 p1 在放入后，p2 放入时根据 p2 的 hashCode 计算的位置相同，且 p2 与该位置的 p1 的 equals 比较为 true， hashSet 认为该对象已经存在，所以拒绝将 p2 存入集合。

```
Set<Point> set=new HashSet<Point>();
Point p1=new Point(1,2);    Point p2=new Point(1,2);
System.out.println("两者是否同一对象："+(p1==p2));
System.out.println("两者内容是否一样："+p1.equals(p2));
```

```
System.out.println("两者 HashCode 是否一样："+  (p1.hashCode()==p2.hashCode()));
set.add(p1);   set.add(p2);   System.out.println("hashset 集合的元素数"+set.size());
for(Point p:set){          System.out.println(p);          }
```

10）不重写 hashCode 方法，但是重写了 equals 方法对 hashSet 的影响：两个对象都可以放入 HashStet 集合中，因为两个对象具有不用的 hashCode 值，那么当他们在放入集合时，通过 hashCode 值进行的散列算法结果就不同。那么他们会被放入集合的不同位置，位置不相同，HashSet 则认为它们不同，所以他们可以全部被放入集合。

11）重写了 hashCode 方法，但是不重写 equals 方法对 hashSet 的影响：在 hashCode 相同的情况下，在存放元素时，他们会在相同的位置，hashSet 会在相同位置上将后放入的对象与该位置其他对象一次进行 equals 比较，若不相同，则将其存入在同一个位置存入若干元素，这些元素会被放入一个链表中。由此可以看出，我们应该尽量使得多种类的不同对象的 hashcode 值不同，这样才可以提高 HashSet 在检索元素时的效率，否则可能检索效率还不如 List。

12）结论：不同对象存放时，不会保存 hashCode 相同并且 equals 相同的对象，缺一不可。否则 HashSet 不认为他们是重复对象。

4.21 Map 集合的实现类 HashMap

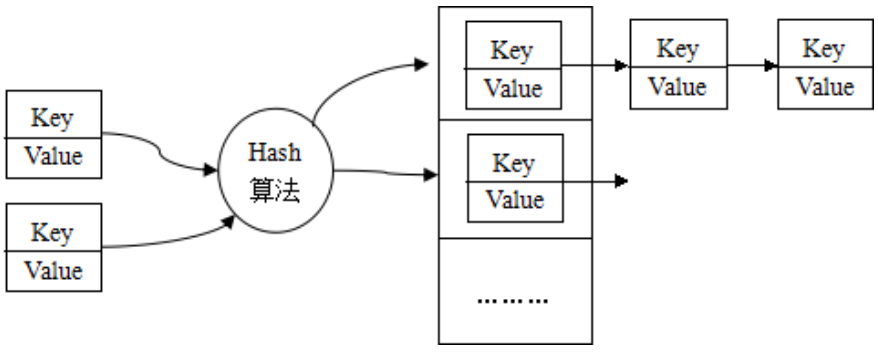
Map 接口定义的集合又称为查找表，用于存储所谓“Key-Value”键值对。Key 可以看成是 Value 的索引。而往往 Key 是 Value 的一部分内容。

1）Key 不可以重复，但所保存的 Value 可以重复。

2）根据内部结构的不同，Map 接口有多种实现类，其中常用的有内部为 hash 表实现的 HashMap 和内部为排序二叉树实现的 TreeMap。同样这样的数据结构在存放数据时，也不建议存放两种以上的数据类型，所以，通常我们在使用 Map 时也要使用泛型约束存储内容的类型。

3）创建 Map 时使用泛型，这里要约束两个类型，一个是 key 的类型，一个是 value 的类型。

4）基本原理图：



5）HashMap 集合中常用的方法：

①V put(K Key,V value)：将元素以 Key—Value 的形式放入 map。若重复保存相同的 key 时，实际的操作是替换 Key 所对应的 value 值。

②V get(Object key)：返回 key 所对应的 value 值。如果不存在则返回 null。

③boolean containsKey(Object Key)：判断集合中是否包含指定的 Key。

④boolean containsValue(Object value)：判断集合中是否包含指定的 Value。

6）若给定的 key 在 map 中不存在则返回 null，所以，原则上在从 map 中获取元素时要先判断是否有该元素，之后再使用，避免空指针异常的出现。Map 在获取元素时非常有针

对性，集合想获取元素需要遍历集合内容，而 Map 不需要，你只要给他特定的 key 就可以获取该元素。

```
Map<String,Point> map=new HashMap<String,Point>();
map.put("1,2", new Point(1,2));    map.put("3,4", new Point(3,4));
Point p=map.get("1,2");    System.out.println("x="+p.getX()+",y="+p.getY());
map.put("1,2", new Point(5,6));//会替换之前的
p=map.get("1,2");    System.out.println("x="+p.getX()+",y="+p.getY());
p=map.get("haha");    System.out.println("x="+p.getX()+",y="+p.getY());// 会报空指异常
```

eg：统计每个数字出现的次数。步骤：①将字符串 str 根据“，”拆分。②创建 map。③循环拆分后的字符串数组。④蒋梅一个数字作为 key 在 map 中检查是否包含。⑤包含则对 value 值累加 1。⑥不包含则使用该数字作为 key，value 为 1 存入 map。

```
String str="123,456,789,456,789,225,698,759,456";
String[] array=str.split(",");
Map<String,Integer> map=new HashMap<String,Integer>();
for(String number:array){    if(map.containsKey(number)){
    int sum=map.get(number);//将原来统计的数字取出 sum++;  //对统计数字加 1
    map.put(number, sum); //放回 map.put(number, map.get(number)+1);等同上三部
    }else{  map.put(number, 1);//第一次出现 value 为 1    }    }
System.out.println(map);//HashMap 也重写了 toString()
```

7）计算机中有这么一句话：越灵活的程序性能越差，顾及的多了。

8）遍历 HashMap 方式一：获取所有的 key 并根据 key 获取 value 从而达到遍历的效果（即迭代 Key）。keySet()方法：是 HashMap 获取所有 key 的方法，该方法可以获取保存在 map 下所有的 key 并以 Set 集合的形式返回。

```
Map<String,Point> map=new HashMap<String,Point>();
map.put("1,2", new Point(1,2));    map.put("2,3", new Point(2,3));
map.put("3,4", new Point(3,4));    map.put("4,5", new Point(4,5));
/** 因为 key 在 HashMap 的泛型中规定了类型为 String，所以返回的 Set 中的元素也是 String，为了更好的使用，我们在定义 Set 类型变量时也应该加上泛型 */
Set<String> keyset=map.keySet();
for(String key:keyset){  Point p=map.get(key);//根据 key 获取 value
    System.out.println(key+":"+p.getX()+","+p.getY());    }
for(Iterator<String> it=keyset.iterator() ;  it.hasNext()  ;  ){//普通 for 循环
    String key=it.next();Point p=map.get(key);
    System.out.println(key+":"+p.getX()+","+p.getY());    }
```

9）LinkedHashMap：用法和 HashMap 相同，内部维护着一个链表，可以使其存放元素时的顺序与迭代时一致。

10）Entry 类，遍历 HashMap 方式二：以“键值对”的形式迭代。Map 支持另一个方法 entrySet()：该方法返回一个 Set 集合，里面的元素是 map 中的每一组键值对，Map 以 Entry 类的实例来描述每一个键值对。其有两个方法：getKey()获取 key 值；getValue()获取 value 值。Entry 也需要泛型的约束，其约束的泛型应该和 Map 相同！Entry 所在位置：java.util.Map.Entry。

```
Map<String,Point> map=new LinkedHashMap<String,Point>();
```

```
        map.put("1,2", new Point(1,2));          map.put("2,3", new Point(2,3));
        map.put("3,4", new Point(3,4));          map.put("4,5", new Point(4,5));    //泛型套泛型
        Set<Entry<String,Point>> entrySet=map.entrySet();//Set 的泛型不会变，就是 Entry
        for(Entry<String,Point> entry:entrySet){
            String key=entry.getKey();//获取 key    Point p=entry.getValue();//获取 value
            System.out.println(key+","+p.getX()+","+p.getY());          }
```

11）List、Map、Set 三个接口存储元素时各有什么特点：

①List：是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素，这类似于 Java 的数组。

②Set：是一种不包含重复的元素的 Collection，即任意的两个元素 e1 和 e2 都有 e1.equals(e2)=false，Set 最多有一个 null 元素。

③Map：请注意，Map 没有继承 Collection 接口，Map 提供 key 到 value 的映射。

4.22 单例模式和模版方法模式

设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。

1）使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。简单的说：设计模式是经典问题的模式化解决方法。

2）经典设计模式分为三种类型，共 23 类。

创建模型式：单例模式、工厂模式等

结构型模式：装饰模式、代理模式等

行为型模式：模版方法模式、迭代器模式等

3）单例设计模式：意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。适用性：当类只能有一个实例而且客户可以从一个众所周知的访问点访问它。任何情况下，该类只能创建一个实例！

4）单例设计模式创建步骤：①定义一个私有的静态的当前类型的属性。②私有化构造方法。③定义一个静态的可以获取当前类实例的方法。这个方法中我们可以判断是否创建过实例，创建过就直接返回，从而达到单例的效果。

```
        private static DemoSingleton obj;
        //或 private static DemoSingleton obj=new DemoSingleton();
        private DemoSingleton(){ }
        public static DemoSingleton getInstance(){
            if(obj==null){      obj= new DemoSingleton();          }
            return obj;                                                    }
```

5）模版方法模式：意图：定义一个操作中的算法过程的框架，而将一些步骤延迟到子类中实现。类似于定义接口或抽象类，子类去实现抽象方法。

Java SE 核心 II

5.1 Java 异常处理机制

异常结构中的父类 Throwable 类，其下子类 Exceptionlei 类和 Error 类。我们在程序中可以捕获的是 Exception 的子类异常。

Error 系统级别的错误：Java 运行时环境出现的错误，我们不可控。

Exception 是程序级别的错误：我们可控。

1）异常处理语句：try-catch，如果 try 块捕获到异常，则到 catch 块中处理，否则跳过忽略 catch 块（开发中，一定有解决的办法才写，无法解决就向上抛 throws）。

```
try{//关键字，只能有一个 try 语句
        可能发生异常的代码片段
}catch(Exception e){//列举代码中可能出现的异常类型，可有多个 catch 语句
    当出现了列举的异常类型后，在这里处理，并有针对性的处理
}
```

2）良好的编程习惯，在异常捕获机制的最后书写 catch(Exception e)（父类，顶极异常）捕获未知的错误（或不需要针对处理的错误）。

3）catch 的捕获是由上至下的，所以不要把父类异常写在子类异常的上面，否则子类异常永远没有机会处理！在 catch 块中可以使用方法获取异常信息：

①getMessage()方法：用来得到有关异常事件的信息。

②printStackTrace()方法：用来跟踪异常事件发生时执行堆栈的内容。

4）throw 关键字：用于主动抛出一个异常

当我们的方法出现错误时（不一定是真实异常），这个错误我们不应该去解决，而是通知调用方法去解决时，会将这个错误告知外界，而告知外界的方式就是 throw 异常（抛出异常）catch 语句中也可抛出异常。虽然不解决，但要捕获，然后抛出去。

使用环境：

我们常在方法中主动抛出异常，但不是什么情况下我们都应该抛出异常。原则上，自身决定不了的应该抛出。那么方法中什么时候该自己处理异常什么时候抛出？

方法通常有参数，调用者在调用我们的方法帮助解决问题时，通常会传入参数，若我们方法的逻辑是因为参数的错误而引发的异常，应该抛出，若是我们自身的原因应该自己处理。

```
public static void main(String[] args) {
    try{/**通常我们调用方法时需要传入参数的话，那么这些方法，JVM 都不会自动处理异常，而是将错误抛给我们解决*/
        String result=getGirlFirend("女神");  System.out.println("追到女神了么？"+result);
    }catch(Exception e){
        System.out.println("没追到");//我们应该在这里捕获异常并处理。
    }
}
public static String getGirlFirend(String name){
    try{  if("春哥".equals(name)){      return "行";
        }else if("曾哥".equals(name)){      return "行";
        }else if("我女朋友".equals(name)){      return "不行";
        }else{/**当出现了错误(不一定是真实异常)可以主动向外界抛出一个异常！ */
```

```
                throw new RuntimeException("人家不干！");
            }
        }catch(NullPointerException e){
            throw e;//出了错不解决，抛给调用者解决
        }
    }
```

5）throws 关键字：不希望直接在某个方法中处理异常，而是希望调用者统一处理该异常。声明方法的时候，我们可以同时声明可能抛出的异常种类，通知调用者强制捕获。就是所谓的“丑话说前面”。原则上 throws 声明的异常，一定要在该方法中抛出。否则没有意义。相反的，若方法中我们主动通过 throw 抛出一个异常，应该在 throws 中声明该种类异常，通知外界捕获。

- 注意事项：
  - 注意 throw 和 throws 关键字的区别：抛出异常和声明抛出异常。
  - 不能在 main 方法上 throws，因为调用者 JVM 直接关闭程序。

```
    public static void main(String[] args) {
        try{  Date today=stringToDate("2013-05-20");   } catch (ParseException e){
            //catch 中必须含有有效的捕获 stringToDate 方法 throws 的异常
            // 输出这次错误的栈信息可以直观的查看方法调用过程和出错的根源
            e.printStackTrace();      }          }
```

eg：将一个字符串转换为一个 Date 对象，抛出的异常是字符格式错误 java.text.ParseException

```
    public static Date stringToDate(String str) throws ParseException{
        SimpleDateFormat format=new SimpleDateFormat("yyyy-MM-DD");
        Date date=format.parse(str);       return date;                    }
```

6）捕获异常两种方式：上例 SimpleDataFormat 的 parse 方法在声明的时候就是用了 throws，强制我们调用 parse 方法时必须捕获 ParseException，我们的做法有两种：一是添加 try-catch 捕获该异常，二是在我们的方法中声明出也追加这种异常的抛出（继续往外抛）。

7）java 中抛出异常过程：java 虚拟机在运行程序时，一但在某行代码运行时出现了错误，JVM 会创建这个错误的实例，并抛出。这时 JVM 会检查出错代码所在的方法是否有 try 捕获，若有，则检查 catch 块是否有可以处理该异常的能力（看能否把异常实例作为参数传进去，看有没有匹配的异常类型）。若没有，则将该异常抛给该方法的调用者（向上抛）。以此类推，直到抛至 main 方法外仍没有解决（即抛给了 JVM 处理）。那么 JVM 会终止该程序。

8）java 中的异常 Exception 分为：

①非检测异常（RuntimeException 子类）：编译时不检查异常。若方法中抛出该类异常或其子类，那么声明方法时可以不在 throws 中列举该类抛出的异常。常见的运行时异常有：NullPointerException、 IllegalArgumentException、ClassCastException、NumberFormatException、ArrayIndexOutOfBoundsException、ArithmeticException

②可检测异常（非 RuntimeException 子类）：编译时检查，除了运行时异常之外的异常，都是可检查异常，则必须在声明方法时用 throws 声明出可能抛出的异常种类！

9）finally 块：finally 块定义在 catch 块的最后（所有 catch 最后），且只能出现一次（0－1 次）， 无论程序是否出错都会执行的块！ 无条件执行！通常在 finally 语句中进行资源的消除工作，如关闭打开的文件，删除临时文件等。

```
public static void main(String[] args) {
    System.out.println(  test(null)+","+test("0")+","+test("")    );              }
    /**输出结果？1,0,2 ？  4,4,4 为正确结果 */
public static int test(String str){
    try{  return    str.charAt(0)-'0';
    }catch(NullPointerException e){          return 1;
    }catch(RuntimeException e){       return 2;
    }catch(Exception e){      return 3;
    }finally{//无条件执行    return 4;       }                                  }
```

10）重写方法时的异常处理

如果使用继承时，在父类别的某个地方上宣告了 throws 某些异常，而在子类别中重新定义该方法时，可以：①不处理异常（重新定义时不设定 throws）。②可仅 throws 父类别中被重新定义的方法上的某些异常（抛出一个或几个）。③可 throws 被重新定义的方法上的异常之子类别（抛出异常的子类）。

但不可以：①throws 出额外的异常。　　②throws 被重新定义的方法上的异常之父类别（抛出了异常的父类）。

5.2 File 文件类

java 使用 File 类（java.io.File）表示操作系统上文件系统中的文件或目录。换句话说，我们可以使用 File 操作硬盘上的文件或目录进行创建或删除。

File 可以描述文件或目录的名字，大小等信息，但不能对文件的内容操作！File 类的构造器都是有参的。

1）关于路径的描述：不同的文件系统差异较大，Linux 和 Windows 就不同！最好使用相对路径，不要用绝对路径。

2）“.”代表的路径：当前目录（项目所处的目录），在 eclipse_workspace/project_name 下，File.separator：常量，目录分隔符，推荐使用！根据系统自动识别用哪种分割符，windows 中为/，Linux 中为\。

3）创建该对象并不意味着硬盘上对应路径上就有该文件了，只是在内存中创建了该对象去代表路径指定的文件。当然这个路径对应的文件可能根本不存在！

```
File file=new File("."+File.separator+"data.dat");//   效果为./data.dat
//File file=new File("e:/XX/XXX.txt");不建议使用
```

4）createNewFile()中有 throws 声明，要求强制捕获异常！

5）新建文件或目录：

①boolean mkdir()：只能在已有的目录基础上创建目录。

②boolean mkdirs()：会创建所有必要的父目录（不存在的自动创建）并创建该目录。

③boolean createNewFile()：创建一个空的新文件。

6）创建目录中文件的两种方式：

①直接指定 data.dat 需要创建的位置，并调用 createNewFile()，前提是目录都要存在！

②先创建一个 File 实例指定 data.dat 即将存放的目录，若该目录不存在，则创建所有不存在的目录，再创建一个 File 实例，代表 data.dat 文件，创建是基于上一个代表目录的 File 实例的。使用 File(File dir,String fileName)构造方法创建 File 实例，然后再调用 createNewFile()：在 dir 所代表的目录中表示 fileName 指定的文件

```
File dir=new File("."+File.separator+"demo"+File.separator+"A");
if(!dir.exists()){ dir.mkdirs();//不存在则创建所有必须的父目录和当亲目录          }
File file=new File(dir,"data.dat");
if(!file.exists()){file.createNewFile();System.out.println("文件创建完毕！");          }
```

7）查看文件或目录属性常用方法

①long length()：返回文件的长度。

②long lastModified()：返回文件最后一次被修改的时间。

③String getName()：返回文件或目录名。　　⑧String getPath()：返回路径字符串。

④boolean exists()：是否存在。　　⑨boolean isFile()：是否是标准文件。

⑤boolean isDirectory()：是否是目录。　　⑩boolean canRead()：是否可以读取。

⑥boolean canWrite()：是否可以写入、修改。

⑦File[] listFiles()：获取当亲目录的子项（文件或目录）

eg1：File 类相关操作

```
File dir=new File(".");     if(dir.exists()&&dir.isDirectory()){//是否为一个目录
    File[] files=dir.listFiles();//获取当前目录的子项（文件或目录）
    for(File file:files){//循环子项
        if(file.isFile()){//若这个子项是一个文件
            System.out.println("文件："+file.getName());
        }else{      System.out.println("目录："+file.getName());     }     }     }
```

eg2：递归遍历出所有子项

```
    File dir=new File(".");              File[] files=dir.listFiles();
if(files!=null&&files.length>0){//判断子项数组有项
    for(File file:files){//遍历该目录下的所有子项
        if(file.isDirectory()){//若子项是目录
            listDirectory(file);//不到万不得已，不要使用递归，非常消耗资源
        }else{System.out.println("文件："+file);//有路径显示，输出 File 的 toString()
            //file.getName()无路径显示，只获取文件名     }     }     }
```

8）删除一个文件：boolean delete()：①直接写文件名作为路径和"./data.dat"代表相同文件，也可直接写目录名，但要注意第 2 条。②删除目录时：要确保该目录下没有任何子项后才可以将该目录删除，否则删除失败！

```
File dir=new File(".");       File[] files=dir.listFiles();
if(files!=null&&files.length>0){      for(File file:files){          if(file.isDirectory()){
    deleteDirectory(file);//递归删除子目录下的所有子项      }else{
        if(!file.delete()){      throw new IOException("无法删除文件："+file);}
            System.out.println("文件："+file+"已被删除！");          }          }
```

9）FileFilter：文件过滤器。FileFilter 是一个接口，不可实例化，可以规定过滤条件，在获取某个目录时可以通过给定的删选条件来获取满足要求的子项。accept()方法是用来定义过滤条件的参数 pathname 是将被过滤的目录中的每个子项一次传入进行匹配，若我们认为该子项满足条件则返回 true。如下重写 accept 方法。

```
FileFilter filter=new FileFilter(){
    public boolean accept(File pathname){
        return pathname.getName().endsWith(".java");//保留文件名以.java 结尾的
        //return pathname.length()>1700;按大小过滤          }          };
File dir=new File(".");//创建一个目录
```

```
File[] sub=dir.listFiles(filter);//获取过滤器中满足条件的子项，回调模式
for(File file:sub){        System.out.println(file);      }
```

10）回调模式：我们定义一段逻辑，在调用其他方法时，将该逻辑通过参数传入。这个方法在执行过程中会调用我们传入的逻辑来达成目的。这种现象就是回调模式。最常见的应用环境：按钮监听器，过滤器的应用。

5.3 RandomAccessFile 类

可以方便的读写文件内容，但只能一个字节一个字节（byte）的读写 8 位。

1）计算机的硬盘在保存数据时都是 byte by byte 的，字节埃着字节。

2）RandomAccessFile 打开文件模式：rw：打开文件后可进行读写操作；r：打开文件后只读。

3）RandomAccessFile 是基于指针进行读写操作的，指针在哪里就从哪里读写。

①void seek(long pos)方法：从文件开头到设置位置的指针偏移量，在该位置发生下一次读写操作。

②getFilePointer()方法：获取指针当前位置，而 seek(0)则将指针移动到文件开始的位置。

③int skipBytes(int n)方法：尝试跳过输入的 n 个字节。

4）RandomAccessFile 类的构造器都是有参的。

①RandomAccessFile 构造方法 1：

```
RandomAccessFile raf=new    RandomAccessFile(file,"rw");
```

②RandomAccessFile 构造方法 2：

```
RandomAccessFile raf=new RandomAccessFile("data.dat","rw");
```

直接根据文件路径指定，前提是确保其存在！

5）读写操作完了，不再写了就关闭：close();

6）读写操作：

```
File file=new File("data.dat");//创建一个 File 对象用于描述该文件
if(!file.exists()){//不存在则创建该文件
    file.createNewFile();//创建该文件，应捕获异常，仅为演示所以抛给 main 了 }
RandomAccessFile raf=new RandomAccessFile(file,"rw");//创建 RandomAccessFile，并将 File 传入，RandomAccessFile 对 File 表示的文件进行读写操作。
/**1 位 16 进制代表 4 位 2 进制；2 位 16 进制代表一个字节8 位 2 进制；
 * 4 字节代表 32 位 2 进制；write(int) 写一个字节，且是从低 8 位写*/
int i=0x7fffffff;//写 int 值最高的 8 位   raf.write(i>>>24);//00 00 00 7f
raf.write(i>>>16);//00 00 7f ff       raf.write(i>>>8);// 00 7f ff ff
raf.write(i);//        7f ff ff ff
byte[]  data=new  byte[]{0,1,2,3,4,5,6,7,8,9};//定义一个 10 字节的数组并全部写入文件
raf.write(data);//写到这里，当前文件应该有 14 个字节了
/**写字节数组的重载方法：write(byte[] data.int offset,int length)，从 data 数组的offset 位置开始写，连续写 length 个字节到文件中 */
raf.write(data, 2, 5);// {2，3，4，5，6}
System.out.println("当前指针的位置："+raf.getFilePointer());
raf.seek(0);//将指针移动到文件开始的位置
int num=0;//准备读取的 int 值
```

```
int b=raf.read();//读取第一个字节 7f 也从低 8 位开始
num=num | (b<<24);//01111111 00000000 00000000 00000000
b=raf.read();//读取第二个字节 ff
num=num| (b<<16);//01111111 11111111 00000000 00000000
b=raf.read();//读取第三个字节 ff
num=num| (b<<8);//01111111 11111111 11111111 00000000
b=raf.read();//读取第四个字节 ff
num=num| b;//01111111 11111111 11111111 11111111
System.out.println("int 最大值："+num);          raf.close();//写完了不再写了就关了
```

7）常用方法：

①write(int data)：写入第一个字节，且是从低 8 位写。

②write(byte[] data)：将一组字节写入。

③write(byte[] data.int offset,int length)：从 data 数组的 offset 位置开始写，连续写 length 个字节到文件中。

④writeInt(int)：一次写 4 个字节，写 int 值。

⑤writeLong(long)：一次写 8 个字节，写 long 值。

⑥writeUTF(String)：以 UTF-8 编码将字符串连续写入文件。

write……

①int read()：读一个字节，若已经读取到文件末尾，则返回-1。

②int read(byte[] buf)：尝试读取 buf.length 个字节。并将读取的字节存入 buf 数组。返回值为实际读取的字节数。

③int readInt()：连续读取 4 字节，返回该 int 值

④long readLong()：连续读取 8 字节，返回该 long 值

⑤String readUTF()：以 UTF-8 编码将字符串连续读出文件，返回该字符串值

read……

```
byte[] buf=new byte[1024];//1k 容量     int sum=raf.read(buf);//尝试读取 1k 的数据
System.out.println("总共读取了："+sum+"个字节");
System.out.println(Arrays.toString(buf)); raf.close();//写完了不再写了就关了
```

8）复制操作：读取一个文件，将这个文件中的每一个字节写到另一个文件中就完成了复制功能。

```
try { File srcFile=new File("chang.txt");
    RandomAccessFile src=new RandomAccessFile(srcFile,"r");//创建一个用于读取文件的 RandomAccessFile 用于读取被拷贝的文件
    File desFile=new File("chang_copy.txt");      desFile.createNewFile();//创建复制文件
    RandomAccessFile  des=new  RandomAccessFile(desFile,"rw");//创建一个用于写入文件的 RandomAccessFile 用于写入拷贝的文件
//使用字节数组作为缓冲，批量读写进行复制操作比一个字节一个字节读写效率高的多！
    byte[] buff=new byte[1024*100];//100k 创建一个字节数组，读取被拷贝文件的所有字节并写道拷贝文件中
    int sum=0;//每次读取的字节数
    while((sum=src.read(buff))>0){   des.write(buff,0,sum);//注意！读到多少写多少！}
        src.close();          des.close();    System.out.println("复制完毕！");
    } catch (FileNotFoundException e) {      e.printStackTrace();
```

```
} catch (IOException e) { e.printStackTrace();      }
//int data=0;//用于保存每一个读取的字节
//读取一个字节，只要不是－1（文件末尾），就进行复制工作
//while((data=src.read())!=-1){          des.write(data);//将读取的字符写入          }
```

9）基本类型序列化：将基本类型数据转换为字节数组的过程。writeInt(111):将 int 值 111 转换为字节并写入磁盘；持久化：将数据写入磁盘的过程。

5.4 基本流：FIS 和 FOS

Java I/O 输入/输出

流：根据方向分为：输入流和输出流。方向的定了是基于我们的程序的。流向我们程序的流叫做：输入流；从程序向外流的叫做：输出流

我们可以把流想象为管道，管道里流动的水，而 java 中的流，流动的是字节。

1）输入流是用于获取（读取）数据的，输出流是用于向外输出（写出）数据的。

InputStream：该接口定义了输入流的特征

OutputStream：该接口定义了输出流的特征

2）流根据源头分为：

基本流（节点流）：从特定的地方读写的流类，如磁盘或一块内存区域。即有来源。

处理流（高级流、过滤流）：没有数据来源，不能独立存在，它的存在是用于处理基本流的。是使用一个已经存在的输入流或输出流连接创建的。

3）流根据处理的数据单位不同划分为：

字节流：以一个“字节”为单位，以 Stream 结尾

字符流：以一个“字符”为单位，以 Reader/Writer 结尾

4）close()方法：流用完一定要关闭！流关闭后，不能再通过其读、写数据

5）用于读写文件的字节流 FIS/FOS（基本流）

①FileInputStream：文件字节输入流。　　②FileOutputStream：文件字节输出流。

6）FileInputStream常用构造方法：

①FileInputStream(File file)：通过打开一个到实际文件的连接来创建一个 FileInputStream，该文件通过文件系统中的 File 对象 file 指定。即向 file 文件中写入数据。

②FileInputStream(String filePath)：通过打开一个到实际文件的连接来创建一个 FileInputStream，该文件通过文件系统中的文件路径名指定。也可直接写当前项目下文件名。

常用方法：

①int read(int d):读取 int 值的低 8 位。

②int read(byte[] b):将 b 数组中所有字节读出，返回读取的字节个数。

③int read(byte[] b,int offset,int length):将 b 数组中 offset 位置开始读出 length 个字节。

④available()方法：返回当前字节输入流 可读取的总字节数。

7）FileOutputStream 常用构造方法：

①FileOutputStream(File File)：创建一个向指定 File 对象表示的文件中写入数据的文件输出流。会重写以前的内容，向 file 文件中写入数据时，若该文件不存在，则会自动创建该文件。

②FileOubputStream(File file,boolean append)：append 为 true 则对当前文件末尾进行写操作（追加，但不重写以前的）。

③FileOubputStream(String filePath)：创建一个向具有指定名称的文件中写入数据的

文件输出流。前提路径存在，写当前目录下的文件名或者全路径。

④FileOubputStream(String filePath,boolean append)：append 为 true 则对当前文件末尾进行写操作（追加，但不重写以前的）。

常用方法：

①void write(int d):写入 int 值的低 8 位。

②void write(byte[] d):将 d 数组中所有字节写入。

③void write(byte[] d,int offset,int length):将 d 数组中 offset 位置开始写入 length 个字节。

5.5 缓冲字节高级流：BIS 和 BOS

对传入的流进行处理加工，可以嵌套使用。

1）BufferedInputStream：缓冲字节输入流

A．构造方法：BufferedInputStream(InputStream in)

BufferedInputStream(InputStream in, int size)

B．常用方法：

①int read()：从输入流中读取一个字节。

②int read(byte[] b,int offset,int length)：从此字节输入流中给定偏移量 offset 处开始将各字节读取到指定的 byte 数组中。

2）BufferedOutputStream：缓冲字节输出流

A．构造方法：BufferedOutputStream(OutputStream out)

BufferedOutputStream(OutputStream out, int size)

B．常用方法：

①void write(int d)：将指定的字节写入此缓冲的输出流。

②void write(byte[] d,int offset,int length)：将指定 byte 数组中从偏移量 offset 开始的 length 个字节写入此缓冲的输出流。

③void flush()：将缓冲区中的数据一次性写出，“清空”缓冲区。

C．内部维护着一个缓冲区，每次都尽可能的读取更多的字节放入到缓冲区，再将缓冲区中的内容部分或全部返回给用户，因此可以提高读写效率。

3）辨别高级流的简单方法：看构造方法，若构造方法要求传入另一个流，那么这个流就是高级流。所以高级流是没有空参数的构造器的，都需要传入一个流。

4）有缓冲效果的流，一般为写入操作的流，在数据都写完后一定要 flush,flush 的作用是将缓冲区中未写出的数据一次性写出：bos.flush();即不论缓存区有多少数据，先写过去，缓冲区再下班～确保所有字符都写出

5）使用 JDK 的话，通常情况下，我们只需要关闭最外层的流。第三方流可能需要一层一层关。

5.6 基本数据类型高级流：DIS 和 DOS

是对“流”功能的扩展，简化了对基本类型数据的读写操作。

1）DataInputStream(InputStream in)：可以直接读取基本数据类型的流

常用方法：

①int readInt()：连续读取 4 个字节（一个 int 值），返回该 int 值

②double readDouble()：连续读取 8 个字节（一个 double 值），返回 double 值

③String readUTF()：连续读取字符串

……

2）DataOutputStream(OutputStream out)：可以直接写基本数据类型的流

常用方法：

①void writeInt(int i)：连续写入 4 个字节（一个 int 值）

②void writeLong(long l)：连续写入 8 个字节（一个 long 值）

③void writeUTF(String s)：连续写入字符串

④void flush()：将缓冲区中的数据一次性写出，“清空”缓冲区。

……

5.7 字符高级流：ISR 和 OSW

以“单个”“字符”为单位读写数据，一次处理一个字符(unicode)。

字符流底层还是基于字节形式读写的。

在字符输入输出流阶段，进行编码修改与设置。

所有字符流都是高级流。

1） OutputStreamWriter：字符输出流。

A．常用构造方法：

OutputStreamWriter(OutputStream out)：创建一个字符集的输出流。

OutputStreamWriter(OutputStream out, String charsetName)：创建一个使用指定字符集的输出流。

B．常用方法：

①void write(int c)：写入单个字符。

②void write(char c[], int off, int len)：写入从字符数组 off 开头到 len 长度的部分

③void write(String str, int off, int len)：写入从字符串 off 开头到 len 长度的部分。

④void flush()：将缓冲区中的数据一次性写出，“清空”缓冲区。

⑤void close()：关闭流。

eg：向文件中写入字符：①创建文件输出流（字节流）。②创建字符输出流（高级流），处理文件输出流，目的是我们可以以字节为单位写数据。③写入字符。④写完后关闭流。

```
OutputStreamWriter  writer=null;//不写 try-catch 外的话 finally 找不到流，就无法关闭
    try{  FileOutputStream fos=new FileOutputStream("writer.txt");
        //  writer=new  OutputStreamWriter(fos);//默认构造方法使用系统默认的编码集
          writer=new OutputStreamWriter(fos,"UTF-8");//最好指定字符集输出
          writer.write("你好！ ");        writer.flush();//将缓冲区数据一次性写出
          }catch(IOException e){          throw e;              }
           finally{        if(writer!=null){      writer.close();       }          }
```

2）InputStreamReader：字符输入流。

A．常用构造方法：

InputStreamReader(InputStream in)：创建一个字符集的输入流。

InputStreamReader(InputStream in, String charsetName)：创建一个使用指定字符集的输入流。

B．常用方法：

①int read()：读取单个字符。

②int read(char cbuf[], int offset, int length)：读入字符数组中从 offset 开始的

length 长度的字符。

③void close()：关闭流。

eg：读取文件中的字符

```
InputStreamReader reader=null;
    try{//创建用于读取文件的字节出入流
      FileInputStream fis=new FileInputStream("writer.txt");
      //创建用于以字符为单位读取数据的高级流
      reader=new InputStreamReader(fis,"UTF-8");      int c=-1;//读取数据
     while((c=reader.read())!=-1){ //InputStreamReader 只能一个字符一个字符的读
                  System.out.println((char)c);              }
     }catch(IOException e){        throw e;       }
     finally{   if(reader!=null){    reader.close();          }              }
```

5.8 缓冲字符高级流：BR 和 BW

可以以“行”为单位读写“字符”，高级流。

在字符输入输出流修改编码。

1）BufferedWriter：缓冲字符输出流，以行为单位写字符

A．常用构造方法：

BufferedWriter(Writer out)：创建一个使用默认大小的缓冲字符输出流。

BufferedWriter(Writer out,int size)：创建一个使用给定大小的缓冲字符输出流。

B．常用方法：

①void write(int c)：写入单个字符。

②void write(char[] c,int off,int len)：写入字符数组从 off 开始的 len 长度的字符。

③void write(String s,int off,int len)：写入字符串中从 off 开始的 len 长度的字符。

④void newLine()：写入一个行分隔符。

⑤flush()：将缓冲区中的数据一次性写出，“清空”缓冲区。

⑥close()：关闭流。

◆ 注意事项：BufferedWriter 的构造方法中不支持给定一个字节输出流，只能给定一个字符输出流 Writer 的子类，Writer 是字符输出流的父类。

```
//创建用于写文件的输出流
FileOutputStream fos=new FileOutputStream("buffered.txt");
//创建一个字符输出流，在字符输入输出流修改编码
OutputStreamWriter osw=new OutputStreamWriter(fos,"UTF-8");
BufferedWriter writer=new BufferedWriter(osw);
writer.write("你好啊！！ ");          writer.newLine();//输出一个换行
writer.write("我是第二行！！ ");   writer.newLine();//输出一个换行
writer.write("我是第三行！！ ");   writer.close();//输出流关闭后，不能再通过其写数
据
```

2）BufferedReader：缓冲字符输入流，以行为单位读字符

A．常用构造方法：

BufferedReader(Reader in)：创建一个使用默认大小的缓冲字符输入流。

BufferedReader(Reader in,int size)：创建一个使用指定大小的缓冲字符输入流。

B．常用方法：

①int read()：读取单个字符。如果已到达流末尾，则返回-1。

②int read(char cbuf[], int off, int len)：从字符数组中读取从 off 开始的 len 长度的字符。返回读取的字符数，如果已到达流末尾，则返回-1。

③String readLine()：读取一个文本行。通过下列字符之一即可认为某行已终止：换行 ('\n')、回车 ('\r') 或回车后直接跟着换行。如果已到达流末尾，则返回 null。EOF：end of file 文件末尾。

④void close()：关闭流。

eg：读取指定文件中的数据，并显示在控制台

```
FileInputStream fis=new FileInputStream(
        "src"+File.separator+"day08"+File.separator+"DemoBufferedReader.java");
InputStreamReader isr=new InputStreamReader(fis);
BufferedReader reader=new BufferedReader(isr);          String str=null;
if((str=reader.readLine())!=null){//readLine()读取一行字符并以字符串形式返回
        System.out.println(str);          }          reader.close();
```

eg：读取控制台输入的每以行信息，直到在控制台输入 exit 退出程序

```
//1 将键盘的字节输入流转换为字符输入流
InputStreamReader isr=new InputStreamReader(System.in);
//2 将字符输入流转换为缓冲字符输入流，按行读取信息
BufferedReader reader=new BufferedReader(isr);
// 循环获取用户输入的信息并输出到控制台
String info=null;    while(true){    info=reader.readLine();
if("exit".equals(info.trim())){  break;          }
        System.out.println(info);//输出到控制台          }          reader.close();
```

5.9 文件字符高级流：FR 和 FW

用于读写“文本文件”的“字符”输入流和输出流。

1）FileWriter 写入：继承 OutputStreamWriter

A．常用构造方法

FileWriter(File file) 、FileWriter(File file, boolean append)

FileWriter(String filePath)、FileWriter(String fileName, boolean append)

意思和 FileOutputStream 的四个同类型参数的构造方法一致。

◆ 注意事项：FileWriter 的效果等同于：FileOutputStream + OutputStreamWriter。

B．常用方法：

①void write(int c)：写入单个字符。

②void write(char c[], int off, int len)：写入字符数组从 off 到 len 长度的部分

③void write(String str, int off, int len)：写入字符串从 off 到 len 长度的部分。

④void flush()：将缓冲区中的数据一次性写出，“清空”缓冲区。

⑤void close()：关闭流。

```
FileWriter writer=new FileWriter("filewriter.txt");
//File file=new File("filewriter.txt");
//FileWriter writer=new FileWriter(file);
writer.write("hello!FileWriter!");                writer.close();
```

2） FileReader 读取：继承 InputStreamReader

A．“只能”以“字符”为单位读取文件，所以效率低

B．常用构造方法

FileReader(File file)、FileReader(String filePath)

意思和 FileInputStream 的两个同类型参数的构造方法一致。

C．常用方法：

①int read()：读取单个字符。

②int read(char cbuf[], int offset, int length)：读入字符数组中从 offset 开始的 length 长度的字符。

③void close()：关闭流。

```
FileReader reader=new FileReader("filewriter.txt");
//int c=-1;      //只能以字符为单位读取文件
//while((c=reader.read())!=-1){      System.out.println((char)c);          }
//将文件字符输入流转换为缓冲字符输入流便可以行为单位读取
BufferedReader br=new BufferedReader(reader);
String info=null;      while((info=br.readLine())!=null){      System.out.println(info);    }
br.close();
```

5.10 PrintWriter

另一种缓冲“字符”输出流，以“行”为单位，常用它作输出，BufferedWriter 用的少。

1)Servlet:运行在服务器端的小程序,给客户端发送相应使用的输出流就是 PrintWriter。

2）写方法：println(String data)：带换行符输出一个字符串，不用手动换行了。

println……

3）构造方式：

PrintWriter(File file)：以行为单位向文件写数据

PrintWriter(OutputStream out)：以行为单位向字节输出流写数据

PrintWriter(Writer writer)：以行为单位向字符输出流写数据

PrintWriter(String fileName)：以行为单位向指定路径的文件写数据

```
PrintWriter writer=new PrintWriter("printwriter.txt"); //向文件写入一个字符串
writer.println("你好！ PrintWriter");//自动加换行符
/**我们要在确定做写操作的时候调用 flush()方法，否则数据可能还在输出流的缓冲区中，没有作真实的写操作！ */
writer.flush();        writer.close();
```

eg：将输出流写入文件

```
System.out.println("你好!！ ");                PrintStream out=System.out;
PrintStream fileOut=new PrintStream(    new FileOutputStream("SystemOut.txt")    );
System.setOut(fileOut);//将我们给定的输出流赋值到 System.out 上
System.out.println("你好！我是输出到控制台的!");              System.setOut(out);
System.out.println("我是输出到控制台的!");                    fileOut.close();
```

5.11 对象序列化

将一个对象转换为字节形式的过程就是对象序列化。序列化还有个名称为串行化，序列化后的对象再被反序列化后得到的对象，与之前的对象不再是同一个对象。

1)对象序列化必须实现 Serializable 接口，但该接口无任何抽象方法，不需要重写方法，只为了标注该类可序列化。

2）且同时建议最好添加版本号（编号随便写）：serialVersionUID。版本号，用于匹配当前类与其被反序列化的对象是否处于同样的特征（属性列表一致等）。反序列化时，

ObjectInputStream 会根据被反序列化对象的版本与当前版本进行匹配，来决定是否反序列化。不加版本号可以，但是可能存在反序列化失败的风险。

3）JDK 提供的大多数 java bean 都实现了该接口

4）transient 关键字：序列化时忽略被它修饰的属性。

5）对象的序列化使用的类：ObjectOutputStream

writeObject(Object obj)：①将给定对象序列化。②然后写出。

6）对象的反序列化使用的类：ObjectInputStream

Object readObject()：将读取的字节序列还原为对象

7）对于 HTTP 协议：通信一次后，必须断开连接，想再次通信要再次连接。

8）想要实现断点续传，我们必须告诉服务器我们当前读取文件的开始位置。相当于我们本地调用的 seek()，因为我们不可能直接调用服务器的对象的方法，所以我们只能通过某种方式告诉服务器我们要干什么。让它自行调用自己流对象的 seek()到我们想读取的位置。bytes=0- 的意思是告诉服务器从第一个字节开始读，即 seek(0)从头到尾；bytes=128- 的意思是告诉服务器从地 129 个字节开始读，即 seek(128)。String prop="bytes="+info.getPos()+"-";

eg：序列化和反序列化

```
        try{  DownloadInfo                                                    info=new
DownloadInfo("http://www.baidu.com/download/xxx.zip"
                    ,     "xxx.zip"         );
              info.setPos(12587);       info.setFileSize(5566987);
              File file=new File("obj.tmp");//将对象序列化以后写到文件中
              FileOutputStream fos=new FileOutputStream(file);
              //通过 oos 可以将对象序列化后写入 obj.tmp 文件中
              ObjectOutputStream oos=new ObjectOutputStream(fos);
              oos.writeObject(info);//将 info 序列化后写出          oos.close();
              ///反序列化操作
              FileInputStream fis=new FileInputStream(file);
              ObjectInputStream ois=new ObjectInputStream(fis);
              DownloadInfo obj=(DownloadInfo)ois.readObject();//反序列化
              System.out.println(obj.getUrl());          System.out.println(obj.getFileName());
              System.out.println(obj.getFileSize());    System.out.println(obj.getPos());
              System.out.println(info==obj);             ois.close();
        }catch(Exception e){    e.printStackTrace();     System.out.println("非常 sorry！");    }
```

5.12 Thread 线程类及多线程

进程：一个操作系统中可以同时运行多个任务（程序），每个运行的任务（程序）被称为一个进程。即系统级别上的多线程（多个任务）。

线程：一个程序同时可能运行多个任务（顺序执行流），那么每个任务（顺序执行流）就叫做一个线程。即在进程内部。

并发：线程是并发运行的。操作系统将时间化分为若干个片段（时间片），尽可能的均匀分配给每一个任务，被分配时间片后，任务就有机会被 cpu 所执行。微观上看，每个任务都是走走停停的。但随着 cpu 高效的运行，宏观上看所有任务都在运行。这种都运行的现象称之为并发，但不是绝对意义上的“同时发生”。

1）Thread 类的实例代表一个并发任务。任何线程对象都是 Thread 类的（子类）实例。Thread 类是线程的模版，它封装了复杂的线程开启等操作，封装了操作系统的差异性。因

此并发的任务逻辑实现只要重写 Thread 的 run 方法即可。

2）线程调度：线程调度机制会将所有并发任务做统一的调度工作，划分时间片（可以被 cup 执行的时间）给每一个任务，时间片尽可能的均匀，但做不到绝对均匀。同样，被分配时间片后，该任务被 cpu 执行，但调度的过程中不能保证所有任务都是平均的获取时间片的次数。只能做到尽可能平均。这两个都是程序不可控的。

3）线程的启动和停止：void start()：想并发操作不要直接调用 run 方法！而是调用线程的 start()方法启动线程！void stop()：不要使用 stop()方法来停止线程的运行，这是不安全的操作，想让线程停止，应该通过 run 方法的执行完毕来进行自然的结束。

4）**线程的创建方式一：**1：继承自 Thread。2：重写 run 方法：run 方法中应该定义我们需要并发执行的任务逻辑代码。

5）**线程的创建方式二：**将线程与执行的逻辑分离开，即实现 Runnalbe 接口。因为有了这样的设计，才有了线程池。关注点在于要执行的逻辑。

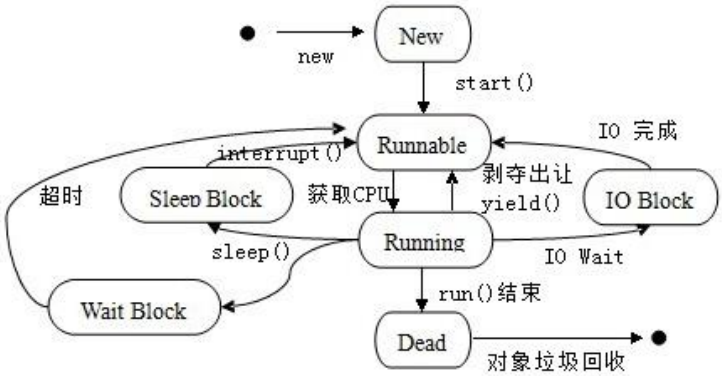
6）Runnable 接口：用于定义线程要执行的任务逻辑。我们定一个类实现 Runnable 接口，这时我们必须重写 run 方法，在其中定义我们要执行的逻辑。之后将 Runnable 交给线程去执行。从而实现了线程与其执行的任务分离开。将任务分别交给不同的线程并发处理，可以使用线程的重载构造方法：Thread(Runnable runnable)。解藕：线程与线程体解藕，即打断依赖关系。Spring 的 ioc 就是干这个的。

```
/**创建两个需要并发的任务，MyFirstRunnable 和 MySecRunnable 都继承了 Runnable 接口并重写了 run()方法 */
        Runnable r1=new MyFirstRunnable();          Runnable r2=new MySecRunnable();
        Thread t1=new Thread(r1);           Thread t2=new Thread(r2);
        t1.start();       t2.start();
```

7）**线程的创建方式三：**使用匿名内部类方式创建线程

```
        new Thread(){public void run(){…}}.start();   /** * 匿名类实现继承 Thread 形式*/
                        Thread t1=new Thread(){
                            public void run(){
                                for(int i=0;i<1000;i++){
                                    System.out.println(i);
                                }
                            }
                        };
        new Thread(new Runnable(){public void run(){…}}).start();
            /**匿名类实现 Runnable 接口的形式 */
                        Thread t2=new Thread(new Runnable(){
                            public void run(){
                                for(int i=0;i<1000;i++){
                                    System.out.println("你好"+i+"次");
                                }
                            }
                        });
```

8）线程生命周期：



9）线程睡眠阻塞：使当前线程放弃 cpu 时间，进入阻塞状态。在阻塞状态的线程不会分配时间片。直到该线程结束阻塞状态回到 Runnable 状态，方可再次获得时间片来让 cpu 运行（进入 Running 状态）。

①static void sleep(times)方法：让当前线程主动进入 Block 阻塞状态，并在 time 毫秒后回到 Runnalbe 状态。

- 注意事项：使用 Thread.sleep()方法阻塞线程时，强制让我们必须捕获“中断异常”。 引发情况：当前线程处于 Sleep 阻塞期间，被另一个线程中断阻塞状态时，当前线程会抛出该异常。

```
int i=0;          while(true){    System.out.println(i+"秒");    i++;
try {       Thread.sleep(1000);
} catch (InterruptedException e) {       e.printStackTrace();       }          }
```

10）void interrupt()方法：打断/唤醒线程。一个线程可以提前唤醒另外一个 sleep Block 的线程。

- 注意事项：方法中定义的类叫局部内部类：局部内部类中，若想引用当前方法的其他局部变量，那么该变量必须是 final 的。

```
final Thread lin=new Thread(){
    public void run(){          System.out.println("林：睡觉了……");
    try {   Thread.sleep(1000000);   } catch (InterruptedException e) {
        System.out.println("林：干嘛呢！干嘛呢！干嘛呢！");
        System.out.println("林：都破了相了！");       }          }          };
        lin.start();//启动第一个线程
Thread huang=new Thread(){
    public void run(){   System.out.println("80 一锤子，您说咂哪儿？");
        for(int i=0;i<5;i++){ System.out.println("80!");
            try {   Thread.sleep(1000);    } catch (InterruptedException e) {
                                                e.printStackTrace();       }          }
            System.out.println("咣当！");
            System.out.println("黄：搞定！");
            lin.interrupt();//中断第一个线程的阻塞状态          }          };
            huang.start();//启动第二个线程
```

11）线程的其他方法：

①static void yield()：当前线程让出处理器（离开 Running 状态）即放弃当前时间片，主动进入 Runnable 状态等待。

②final void setPriority(int)：设置线程优先级；优先级越高的线程，理论上获取 cpu

的次数就越多。但理想与现实是有差距的……设置线程优先级一定要在线程启动前设置！

③final void join()：等待该线程终止。

```
Thread t1=new Thread(){
    public void run(){  for(int i=0;i<100;i++){   System.out.println("我是谁啊？ ");
            Thread.yield();          }          }
};
Thread t2=new Thread(){
    public void run(){  for(int i=0;i<100;i++){   System.out.println("我是修水管的");
            Thread.yield();          }          }
};
Thread t3=new Thread(){
    public void run(){  for(int i=0;i<100;i++){   System.out.println("我是打酱油的");
            Thread.yield();          }          }
};
t1.setPriority(Thread.MAX_PRIORITY);    t2.setPriority(Thread.MIN_PRIORITY);
t1.start();     t2.start();     t3.start();
```

12）线程并发安全问题：synchronized 关键字，线程安全锁、同步监视器。

多线程在访问同一个数据时（写操作），可能会引发不安全操作。

①哪个线程报错不捕获，则线程死，不影响主程序。

②同步：同一时刻只能有一个执行，A 和 B 配合工作，步调一致的处理（B 得到 A 的执行结果才能继续）。如一群人上公交车。

异步：同一时刻能有多个执行，并发，各自干各自的。如一群人上卡车。

③synchronized 可以修饰方法也可以单独作为语句块存在（同步块）。作用是限制多线程并发时同时访问该作用域。

④synchronized 修饰方法后，会为方法上锁。方法就不是异步的了，而是同步的。锁的是当前对象。

⑤synchronized 同步块：分析出只有一段代码需要上锁，则使用。效率比直接修饰方法要高。

⑥线程安全的效率低，如 Vector、Hashtable。线程不安全的效率高，如 ArrayList、HashMap

```
synchronized void getMoney(int money){          if(count==0){
            throw new RuntimeException("余额为 0");      }
            Thread.yield();count-=money;                 }
void getMoney(int money){
    synchronized(this){//synchronized(Object){需要同步的代码片段}
    if(count==0){  throw new RuntimeException("余额为 0");      }
    Thread.yield();count-=money;                 }
```

13）Daemon 后台线程也称为守护线程：当当前进程中“所有”“前台”线程死亡后，后台线程将被强制死亡（非自然死亡），无论是否还在运行。

①守护线程，必须在启动线程前调用。

②main 方法也是靠线程运行的，且是一个前台线程。

③正在运行的线程都是守护线程时，JVM 退出。

14）wait/notify 方法

这两个方法不是在线程 Thread 中定义的方法，这两个方法定义在 Object 中。两个

方法的作用是用于协调线程工作的。

①等待机制与锁机制密切关联：wait/notify 方法必须与 synchronized 同时使用，谁调用 wait 或 otify 方法，就锁谁！

②wait()方法：当条将不满足时，则等待。当条件满足时，等待该条件的线程将被唤醒。如：浏览器显示一个图片，displayThread 要想显示图片，则必须等代下载线程 downloadThread 将该图片下载完毕。如果图片没有下杂完成，则 dialpayThread 可以暂停。当 downloadThread 下载完成后，再通知 displayThread 可以显示了，此时 displayThread 继续执行。

③notify()方法：随机通知、唤醒一个在当前对象身上等待的线程。

④notifyAll 方法：通知、唤醒所有在当前对象身上等待的线程。

5.13 Socket 网络编程

Socket 套接字。在 java.net.Socket 包下。

1）网络通信模型：C/S：client/server，客户端/服务器端；B/S：browser/server，浏览器端/服务器端；C/S 结构的优点：应用的针对性强，画面绚丽，应用功能复杂。缺点：不易维护。B/S 结构的优点：易于维护。缺点：效果差，交互性不强。

2）Socket：封装着本地的地址，服务端口等信息。ServerSocket：服务端的套接字。

服务器：使用 ServerSocket 监听指定的端口，端口可以随意指定（由于 1024 以下的端口通常属于保留端口，在一些操作系统中不可以随意使用，所以建议使用大于 1024 的端口），等待客户连接请求，客户连接后，会话产生；在完成会话后，关闭连接。

客户端：使用 Socket 对网络上某一个服务器的某一个端口发出连接请求，一旦连接成功，打开会话；会话完成后，关闭 Socket。客户端不需要指定打开的端口，通常临时的、动态的分配一个 1024 以上的端口。

3）永远都是 Socket 去主动连接 ServerSocket。一个 ServerSocket 可以接收若干个 Socket 的连接。网络通信的前提：一定要捕获异常。

4）Socket 连接基于 TCP/IP 协议，是一种长连接（长时间连着）。

5）读取服务器信息会阻塞，写操作不会。

6）建立连接并向服务器发送信息步骤：①通过服务器的地址及端口与服务器连接，而创建 Socket 时需要以上两个数据。②连接成功后可以通过 Socket 获取输入流和输出流，使用输入流接收服务端发送过来的信息。③关闭连接。

7）连接服务器：一旦 Socket 被实例化，那么它就开始通过给定的地址和端口号去尝试与服务器进行连接（自动的）。这里的地址"localhost"是服务器的地址，8088 端口是服务器对外的端口。我们自身的端口是系统分配的，我们无需知道。

8）和服务器通信（读写数据）：使用 Socket 中的 getInputStream()获取输入流，使用 getOutputStream()获取输出流。

9）ServerSocket 构造方法要求我们传入打开的端口号，ServerSocket 对象在创建的时候就向操作系统申请打开这个端口。

10）通过调用 ServerSocket 的 accept 方法，使服务器端开始等待接收客户端的连接。该方法是一个阻塞方法，监听指定的端口是否有客户端连接。直到有客户端与其连接并接收客户端套接字，否则该方法不会结束。

eg1.1：客户端 ClientDemo 类

```
private Socket socket;
```

```
    public void send(){
        try{  System.out.println("开始连接服务器");      socket=new Socket("localhost",8088);
        InputStream in=socket.getInputStream();//获取输入流
        OutputStream out=socket.getOutputStream();//获取输出流
        /**将输出流变成处理字符的缓冲字符输出流*/
        PrintWriter writer=new PrintWriter(out);            writer.println("你好！服务器！");
        /**注意，写到输出流的缓冲区里了，并没有真的发给服务器。想真的发送就要作真实的写操作，清空缓冲区*/
        writer.flush();
        /**将输入流转换为缓冲字符输入流*/
        BufferedReader reader=new BufferedReader(new InputStreamReader(in));
        /**读取服务器发送过来的信息*/
        String info=reader.readLine();//读取服务器信息会阻塞     System.out.println(info);
        writer.println("再见！服务器！");       writer.flush();
        info=reader.readLine();          System.out.println(info);
        }catch(Exception e){       e.printStackTrace();       }                    }
    public static void main(String[] args){
        ClientDemo demo=new ClientDemo();          demo.send();//连接服务器并通信
    }
```

eg1.2：服务器端 ServerDemo 类（不使用线程）

```
    private ServerSocket socket=null;          private int port=8088;
    /**构建 ServerDemo 对象时就打开服务端口*/
    public ServerDemo(){
        try{  socket=new ServerSocket(port); }catch(Exception e){   e.printStackTrace();}      }
    /**开始服务，等待收受客户端的请求并与其通信*/
    public void start(){
        try{  System.out.println("等待客户端连接……");       Socket s=socket.accept();
            //获取与客户端通信的输入输出流
            InputStream in=s.getInputStream();      OutputStream out=s.getOutputStream();
            //包装为缓冲字符流
            PrintWriter writer=new PrintWriter(out);
            BufferedReader reader=new BufferedReader(new InputStreamReader(in));
            //先听客户端发送的信息
            String info=reader.readLine();//这里同样会阻塞         System.out.println(info);
            //发送信息给客户端
            writer.println("你好！客户端");          writer.flush();
            info=reader.readLine();          System.out.println(info);
            writer.println("再见！客户端");          writer.flush();
            socket.close();//关闭与客户端的连接
        }catch(Exception e){       e.printStackTrace();       }                  }
    public static void main(String[] args){          System.out.println("服务器启动中……");
        ServerDemo demo=new ServerDemo();       demo.start();                }
```

eg2：服务器端 ServerDemo 类（使用线程），start()方法的修改以及 Handler 类

```
    public void start(){
```

```
        try{while(true){ System.out.println("等待客户端连接……"); Socket s=socket.accept();
            /** 当一个客户端连接了，就启动一个线程去接待它 */
            Thread clientThread=new Thread(new Handler(s));     clientThread.start();      }
        }catch(Exception e){      e.printStackTrace();      }          }
    /** 定义线程体，该线程的作用是与连接到服务器端的客户端进行交互操作 */
    class Handler implements Runnable{
        private Socket socket;//当前线程要进行通信的客户端 Socket
        public Handler(Socket socket){//通过构造方法将客户端的 Socket 传入
                this.socket=socket;       }
        public void run(){
        try{ //获取与客户端通信的输入输出流
          InputStream                                  in=socket.getInputStream();OutputStream
out=socket.getOutputStream();
          PrintWriter writer=new PrintWriter(out);//包装为缓冲字符流
          BufferedReader reader=new BufferedReader(new InputStreamReader(in));
          String info=reader.readLine();//先听客户端发送的信息，这里同样会阻塞
          System.out.println(info);
          //发送信息给客户端
          writer.println("你好！客户端");          writer.flush();
          info=reader.readLine();            System.out.println(info);
          writer.println("再见！客户端");          writer.flush();
          socket.close();//关闭与客户端的连接
        }catch(Exception e){      e.printStackTrace();      }                }                }
    public static void main(String[] args){          System.out.println("服务器启动中……");
        ServerDemo demo=new ServerDemo();       demo.start();              }
```

5.14 线程池

线程若想启动需要调用 start()方法。这个方法要做很多操作。要和操作系统打交道。注册线程等工作，等待线程调度。ExecutorService 提供了管理终止线程池的方法。

1）线程池的概念：首先创建一些线程，它们的集合称为线程池，当服务器接受到一个客户请求后，就从线程池中取出一个空闲的线程为之服务，服务完后不关闭该线程，而是将该线程还回到线程池中。在线程池的编程模式下，任务是提交给整个线程池，而不是直接交给某个线程，线程池在拿到任务后，它就在内部找有无空闲的线程，再把任务交给内部某个空闲的线程，一个线程同时只能执行一个任务，但可以同时向一个线程池提交多个任务。

2）线程池的创建都是工厂方法。我们不要直接去 new 线程池，因为线程池的创建还要作很多的准备工作。

3）常见构造方法：

①Executors.newCachedThreadPool():可根据任务需要动态创建线程，来执行任务。若线程池中有空闲的线程将重用该线程来执行任务。没有空闲的则创建新线程来完成任务。理论上池子里可以放 int 最大值个线程。缓存线程生命周期 1 分钟，得不到任务直解 kill

②Executors.newFixedThreadPool(int threads):创建固定大小的线程池。池中的线程数是固定的。若所有线程处于饱和状态，新任务将排队等待。

③Executors.newScheduledThreadPool():创建具有延迟效果的线程池。可将带运行的任务延迟指定时长后再运行。

④Executors.newSingleThreadExecutor():创建单线程的线程池。池中仅有一个线程。所有未运行的任务排队等待。

5.15 双缓冲队列

BlockingQueue：解决了读写数据阻塞问题，但是同时写或读还是同步的。

1）双缓冲队列加快了读写数据操作，双缓冲对列可以规定队列存储元素的大小，一旦队列中的元素达到最大值，待插入的元素将等。等待时间是给定的，当给定时间到了元素还没有机会被放入队列那么会抛出超时异常。

2）LinkedBlockingQueue 是一个可以不指定队列大小的双缓冲队列。若指定大小，当达到峰值后，待入队的将等待。理论上最大值为 int 最大值。

eg1.1：log 服务器写日志文件，客户端 ClientDemo 类，try 语句块中修改如下

```
try{        System.out.println("开始连接服务器");
            socket=new Socket("localhost",8088);
            OutputStream out=socket.getOutputStream();
            PrintWriter writer=new PrintWriter(out);
            while(true){
                writer.println("你好！服务器！");
                writer.flush();
                Thread.sleep(500);                    }                }
```

eg1.2：log 服务器写日志文件，服务器端 ServerDemo 类，增加线程池和双缓冲队列两个属性，删掉与原客户端的输出流

```
private ExecutorService threadPool;//线程池
private BlockingQueue<String> msgQueue;  //双缓冲队列
public ServerDemo(){
    try{  socket=new ServerSocket(port);
        //创建 50 个线程的固定大小的线程池
        threadPool=Executors.newFixedThreadPool(50);
        msgQueue=new LinkedBlockingQueue<String>(10000);
        /**创建定时器，周期性的将队列中的数据写入文件*/
        Timer timer=new Timer();
        timer.schedule(new TimerTask(){
            public void run(){
                try{  //创建用于向文件写信息的输出流
                    PrintWriter writer=new PrintWriter(new FileWriter("log.txt",true));
                    //从队列中获取所有元素，作写出操作
                    String msg=null;
                    for(int i=0;i<msgQueue.size();i++){
                        /**参数  0：时间量 TimeUnit.MILLISECONDS：时间单位*/
                        msg=msgQueue.poll(0,TimeUnit.MILLISECONDS);
                        if(msg==null){    break;   }
                        writer.println(msg);//通过输出流写出数据
                    }
                    writer.close();
                }catch(Exception e){       e.printStackTrace();       }
```

```
                }
            }, 0,500);
        }catch(Exception e){       e.printStackTrace();       }            }
    public void start(){
        try{  while(true){    System.out.println("等待客户端连接……");
                Socket s=socket.accept();
    /**将线程体（并发的任务）交给线程池，线程池会自动将该任务分配给一个空闲线程去执行。*/                threadPool.execute(new Handler(s));
                System.out.println("一个客户端连接了，分配线程");                }
        }catch(Exception e){            e.printStackTrace();       }            }
    /**定义线程体，该线程的作用是与连接到服务器端的客户端进行交互操作*/
    class Handler implements Runnable{
        private Socket socket;//当前线程要进行通信的客户端 Socket
        public Handler(Socket socket){//通过构造方法将客户端的 Socket 传入
            this.socket=socket;        }
        public void run(){
            try{  //获取与客户端通信的输入输出流
                InputStream in=socket.getInputStream();
                //包装为缓冲字符流
                BufferedReader reader=new BufferedReader(new InputStreamReader(in));
                String info=null;
                while(true){//循环读取客户端发送过来的信息
                    info=reader.readLine();
                    if(info!=null){  //插入对列成功返回 true，失败返回 false
                        //该方法会阻塞线程，若中断会报错!
                        boolean b=msgQueue.offer(info, 5, TimeUnit.SECONDS);          }
                }
            }catch(Exception e){       e.printStackTrace();       }
        }
    }
```