必须养成优秀程序员的编写习惯：缩进(用空格)、注释、命名约定。

大小写敏感。

单独的“；”代表一条空语句。

main函数是我们整个程序的执行入口所以必须是静态公开的。

必须写成这样： publicstaticvoidmain(String[]args){……}

生成jar包：

在eclipse里，选中要打包的几个文件，右键-Export-写文件名-Next-Next-选main方法的class-finish

在jar包的同一文件夹下，新建一个空文档，写“java -jar ./文件名.jar”，再把这文档改成“文件名.sh”

把这sh的属性－权限 改成“允许以程序执行文件”。以后双击这个sh即可运行

文本注释 Comments：

注释必须写上，以便其它人阅读、引用和维护。

单行注释 //……

多行注释 /* ……*/

文档注释 /** …… */

文档注释，可以使用JDK的javadoc工具从原文件中抽取这种注释形成程序的帮助文档。

使用javadoc命令建立HTML格式的程序文档：

javadoc[options][packagenames][sourcefiles][@files]

标示符：

用来给一个类、变量或方法命名的符号

标示符命名规则：

1．以字母，“_”和“$”开头。可以包含字母、数字、“_”和“$”。

在实际应用中，应避免定义“$”开头的变量，因为这是表示特殊类型的。

2．大小写敏感

3．不能与保留关键字冲突

4．没有长度限制(暗示使用长的标示符，以便阅读。长名字可使用工具输入)

5.建议使用JavaBeans规则命名，并根据方法的目的，以 set、get、is、add 或 remove 开头。

标示符命名约定：

1．类名、接口名：每个单词的首字母应该大写，尤其第一个单词的首字母应该大写。(驼峰规则)

```
class MyFirstClass
interface Weapon
```

2．字段、方法以及对象：第一个单词首字母应小写，其它单词首字母大写。(以便跟上面的有所区别)

boolean isWoman

void setName(String name)

3．常量：全部用大写字母表示。如果由几个单词组成，则由下画线连接。

public final int GREEN

public final int HEAD_ COUNT

4．Java包(Package)：全部用小写字母。

package java.awt.event

java.lang.System.gc(); / java.lang.Runtime.gc();

垃圾回收的建议语句,只能建议而不能强制回收

注意： System.gc(); 是静态方法，可直接调用。

java.lang.Runtime.gc(); 不是静态方法，不能直接在main方法里调用

package 包

目的：命名冲突，便于管理类

运行时，先找到包所在目录，再执行“ 包名.类名”

import 导入。导入包内的类

定义包之后，执行时：javac -d 包的路径 类名.java

java 包名.类名

import java.util.*; //表示导入java.util里面的所有类；但import java.*; 则什么类都导不进

用“*”表示导入当前包的类，不包括子包的类(可把包看作目录)。

声明规则

* 一个源代码文件最多只能有一个公共(public)类。
* 如果源文件包含公共类，则该文件名称应该与公共类名称相同。
* 一个文件只能有一个包语句，但是，可以有多个导入语句。
* 包语句（如果有的话）必须位于源文件的第一行。
* 导入语句（如果有的话）必须位于包之后，并且在类声明之前。
* 如果没有包语句，则导入语句必须是源文件最前面的语句。
* 包和导入语句应用于该文件中的所有类。
* 一个文件能够拥有多个非公共类。
* 没有公共类的文件没有任何命名限制。

输入：使用 Scanner 获取输入

在J2SE 5.0中，可以使用java.util.Scanner类别取得使用者的输入

可以使用这个工具的 next() 功能，来获取用户的输入

```
import java.util.Scanner;
Scanner s = new Scanner(System.in);
System.out.println("请输入文字");
System.out.printf("您输入了字符： %s \r\n", s.next());
System.out.println("请输入数字");
System.out.printf("您输入了数字： %d \r\n",
s.nextInt());
```

输入：使用 BufferedReader 取得输入 // 5.0之前的读取键盘的方法

BufferedReader建构时接受java.io.Reader物件

可使用java.io.InputStreamReader

例：

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
class test{
public static void main(String args[]){
System.out.println("请输入一列文字");
BufferedReader s = new BufferedReader(new
InputStreamReader(System.in));
String next;
try{ next = s.readLine(); // 此语句会抛异常，需处理
System.out.println("您输入了文字：" + next);
}catch(Exception e){}
}}
```

数值保存方式：

正数＝ 二进制

负数＝ 补码

补码＝ 反码 +1 正数＝负数的补码(反码+1)

反码＝ 非(二进制数)

八进制数，零开头 011(八进制)＝9(十进制)

十六进制数，零x开头 0x55(十六进制)＝5*16+5(十进制)

类型：数据都必须有类型

boolean (8bit,不定的)只有true和false两个值

char 16bit, 0~2^16-1 (2^16=6万6)

byte 8bit, -2^7~2^7-1 (2^7=128； 注意：两个 byte 数相加，变 int 型)

short 16bit, -2^15~2^15-1 (2^15=32768)

int 32bit, -2^31~2^31-1 (2147483648,20亿,10位有效数字)

long 64bit, -2^63~2^63-1 (900亿亿,20位有效数字)

float 32bit, 9位有效数字，含小数(四舍五入)(小数点算一位，正

负号不算)

double 64bit, 18位有效数字(最大: 1.7e308)

注：float 和 double 的小数部分不可能精确，只能近似。

比较小数时，用 double i=0.01; if (i - 0.01 < 1E-6) ……

不能直接 if (i==0.01)……

默认，整数是int类型，小数是double类型

long类型值，需跟L或l在数据后；float类型要跟f或F；或强制类型转换

科学计数法：12.5E3

类型转换默认序列：

byte > short > int > long > float > double

char 」

注意：默认类型转换(自动类型提升)会丢失精度，但只有三种情况：

int>float; long>float; long>double. 看一下他们的有效位就明白。

二进制是无法精确的表示 0.1 的。

进行高精度运算可以用java.math包中BigDecimal类中的方法。

自动类型提升又称作隐式类型转换。

强制类型转换：int ti; (byte) ti ;

强制转换，丢弃高位

宣告变量名称的同时，加上“final”关键词来限定，这个变量一但指定了值，就不可以再改变它的值

如：final int n1= 10; n1=20; 这就会报错

输出命令：

System.out.println() 会自动换行的打印

System.out.print() 直接打印，不会自动换行

System.out.printf() 可插入带 % 的输入类型,前两种只可以插入转义符, 不能插入 % 的数据或字符串

在 printf 里面,输出有5个部分 %[argument_index$][flags][width][.precision]conversion

以“%”开头，[第几个数值$][flags][宽度][.精确度][格式]

printf()的引入是为了照顾c语言程序员的感情需要

格式化输出 Formatter；格式化输入 Scanner；正则表达式

输出格式控制：

转义符 (Escape Sequence)：

\ddd 1到3位8进制数指定Unicode字符输出(ddd)

\uxxxx 1到4位16进制数指定Unicode字符输出(xxxx)
\\ \
\' '
\" "
\b 退格(光标向左走一格)
\f 走纸转页,换页
\n 换行
\r 光标回到行首，不换行
\t 跳格
%% %
%d 输出10进制整数，只能输出Byte、Short、 Integer、Long、或BigInteger类型。(输出其它类型会抛异常)
%f 以10进制输出浮点数，提供的数必须是Float、Double或BigDecimal (输出Integer类型也抛异常)
%e,%E 以10进制输出浮点数，并使用科学记号，提供的数必须是Float、 Double或BigDecimal
%a,%A 用科学记号输出浮点数,以16进制输出整数部份,以10进制输出指数部份,数据类型要求同上。
%o (字母o)以8进制整数方式输出，限数据类型:Byte,Short,Integer,Long或BigInteger
%x,%X 将浮点数以16进制方式输出，数据类型要求同上
%s,%S 将字符串格式化输出(可输出任何类型)
%c,%C以字符方式输出，提供的数必须是Byte、Short、Character或 Integer
%b,%B 输出"true"或"false"(%B输出"TRUE"或"FALSE");另外,非空值输出true,空值输出 false
%t,%T 输出日期/时间的前置，详请看API文档

```
/********找出各字符的Unicode值*******************/
class Test{
public static void main(String[] args) {
String s= ""+0+'a'; //0=48,9=57
//A=65,Z=90;a=97,z=122;空格=32
int i = s.codePointAt(0);
int j = s.codePointAt(1);
//利用这codePointAt(int index)方法
System.out.printf("%d %d",i,j);
}}
/***********************************************/
```

java.lang.Math:
求绝对值： Math.abs(数值);

求平方根： Math.sqrt(数值);
求最大值： Math.max(数值1,数值2);
求最小值： Math.min(数值1,数值2);
字符串的拼接：
字符串+数值＝字符串
数值+字符串＝字符串
如：str+10+20 ＝＝str1020 而 10+20+str ＝＝30str
str1 = str1.concat(str2); 与这语句同效： str1 = str1 + str2;
"+" 和 "+=" 都被重载了，具有合并字符串的能力，相当于 String 类里的 concat()；
字符串的长度：length()方法获取
StringBuffer 字符串：
注：位置从0开始
1) charAt方法 可取字符串中的一个元素。 // char charAt(int index)
2) setCharAt方法 可更改字符串中的某个元素。 // char setCharAt(int index,char c)
例： StringBuffer sb = new StringBuffer("abcd");
sb.setCharAt(2,'j'); //把'c'改成'j'
3) length方法，取得长度。 //int length()
4) setLength方法，更改长度。//void setLength(int newLength)
当增加字符串长度时，会在原字符串后加上null；缩短时则删除多出部分
5)capacity方法，取得字符串占用空间的大小。一般比length方法大一些，用法一样。 // int capacity()
6) append方法，在后面加入字符，如果加入其它类型会自动转换成字符串。加入后，缓冲区的长度会增加。
// StringBuffer append(String str)
7) insert方法，在指定位置插入字符串，原位置的元素往后移。
//StringBuffer insert(int offset, String str)
//StringBuffer insert(int offset, char c)
8) delete方法，删除指定字符。
//StringBuffer delete(int start,int end); 从start开始删除到end-1位置
9) replace方法，代替指定位置的字符串。若指定位置不存在，则加到后面；字符串会因代替的内容而改变长度。

//StringBuffer replace(int start, int end, String str) 从start开始到end-1位置，替换为 str

例： StringBuffer sb = new StringBuffer("abcd");

sb.append("efg"); //用法跟 String的concat 不同，它不需返回

sb.insert(3," java "); //结果：abc java defg

sb.delete(6,10); //结果：abc jaefg

10) 小积累：

在 String 中想加入双引号，可用单引号括起一个双引号，再加进去。如：str += '"'; // char 类型

也可以用转义符，如: str += "dd\"dd";

String 与 StringBuffer 的转换：

```
//String 转成 StringBuffer
String s1 = "String1";
StringBuffer sb1 = new StringBuffer(s1);
//StringBuffer 转成 String
StringBuffer sb2 = new StringBuffer("String2");
String str2 = sb2.toString();
```

运算：

算术运算： 加(+) 减(－) 乘(*) 除(/) 取余(%)

% 取余运算： 2%3=2 100%3=1

赋值运算符：

= += -= *= /= %=

(先运行完右边的，再跟左边的进行赋值运算；如 int i=10;i-=5-3; 结果8)

<<= >>=

比较、条件运算：

大于> 不小于>= 小于< 不大于<= 等于== 不等于 !=

逻辑运算：

短路运算(且 && 或 ||) 非短路运算(& |) 反相 !

短路运算：当前面一个表达式可以决定结果时，后面的语句不用再判断。非短路运算时，还照样判断后面的

位运算：

&(AND) |(OR) ^(XOR异或) ~(补码)按位取反 = 加1再取反(全 1 的补码是-1)

移位运算：

>> << >>>

>>右移:全部向右移动,移到右段的低位被舍弃,最高位则移入原来最

高位的值。右移一位相当于除2取商。

>>>同上，只是最高位移入0(不带符号)。因为最高位是符号位，所以负数跟 >> 有区别，正数没区别。

12>>>33 为12>>(33%32) = 12>>1 =6；因为int 型只有32位，认为全移走后就没意义

1 <<32 为1

“==”双等于号，比较数值是否相等。还可以用于比较两个引用，看他们引用的地址是否相等。

在 Object 类里 equals() 跟“==”功能一样；但可以重载定义一个比较两者意义是否相等的方法。

在java里可以把赋值语句连在一起写，如： x=y=z=5;这样就x,y,z都得到同样的数值 5

关系运算符：instanceof

用户判断某一个对象是否属于某一个类的实例。

1、boolean c = a instanceof B； //a是某类的实例，而B是一个类。

2、一般用在强制类型转换之前判断对象变量是否可以强制转换为指定类型。

两个数相运算时，默认是 int 类型

如果有更高级的，就按高级的那个类型

if(其中一个是double型)double型；

else if(其中一个是float型)float型；

else if(其中一个是long型)long型；

else int 型。

选择：

```
if(……){……}else{……}
if(……){……}else if(……){……}else{……}
if(……){…… if(……){……}else{……}}else{……}
```

三重以上的选择，建议使用 switch

```
switch(char c){
case c1: ……; break;
case c2: ……; break;
……
default :……;
} /*switch的括号里只能用 int 和 枚举类型，以及能隐式转换为 int 的。
```

如：byte,short,char,Integer,Byte,Short,Character。

不能用 long、小数类型(float,double) 和 String。

case后的值必须是常量。而包装类变量(Integer,Character)不会被视作常量。*/

循环：

for(初始表达式; 布尔表达式 ; 增量) { 循环语句; }

跟C的 for 一样，for 的初始化条件、结束条件、增量都可以不写。

但条件判断部分只能是boolean值，所以只能是一条条件判断语句。

for 循环一般用在循环次数已知的情况。

while () { ……; }

do { ……; } while (<condition>); 注意：do 后最好用“{}”，while 后的分号不可忘。

break 和 continue

break 退出当前的循环体，在嵌套循环中，只退出当前的一层循环。

continue 结束当前本次循环，继续进行下一轮的循环。可以说，只是本次忽略循环内后面的语句。

continue 只能在循环体内用。break 可以用在任意代码块中，表示退出当前程序块(配合标签使用，很好用)

这两个相当于JAVA里的 goto 语句。

注意：(个人归结的)

循环体内申明的变量，在循环体结束后立即释放，循环体外无法使用。

但在另外一个循环体内可以再次申明一个跟前面同名的变量，互相不影响。

如for内定义的 i： for(int i=0;i<10;i++){……}

则在上式 for 循环结束后无法再调用 i 值，还会报错。

for(int i=0;i<10;i++){……} 和后面的 for(int i=0;i<3;i++){……} 互不影响

若想循环体外还可以调用 for 循环体内的值，应先在体外定义。

如： int i; for (i=0; i<10; i++){……} 则for 循环后再调用 i 值，其值为10

关键字列表：

abstract boolean break byte case catch char class
continue default do double else extends enum false
final finally float for if implements import instanceof
int interface long native new null package private
protected public return short static super switch synchronized
this throw throws transient true try void volatile while

Java 中 true、false不是关键字,而是boolean类型的字面量。但也不能当作变量用。

所有的关键字都是小写，friendly，sizeof不是java的关键字

保留字：const，goto ：这两个已经削去意义，但同样不能用作变量名。

取得系统时间：Calendar

```
java.util.Calendar
Calendar cal = Calendar.getInstance();
int year = cal.get(Calendar.YEAR);
int month = cal.get(Calendar.MONTH) + 1; //注：月份从0开始
int date = cal.get(Calendar.DATE); //取得当前的日期
int day = cal.get(Calendar.DAY_OF_WEEK); //取得当前是星期几。周日是1，周六是7
int hour = cal.get(Calendar.HOUR_OF_DAY);
int min = cal.get(Calendar.MINUTE);
int sec = cal.get(Calendar.SECOND);
System.out.println("今天是："+year+"年"+month+"月"+date+"日");
System.out.println("现在是:"+hour+"时"+min+"分"+sec+"秒");
```

Calendar 的 after、before、equals方法，比较两个时间

after(): Calendar现在的时刻比参数的Calendar时刻晚，则true；否则false

before(): Calendar现在的时刻比参数的Calendar时刻早，则true；否则false

equals(): Calendar现在的时刻跟参数的Calendar时刻相等，则true；否则false

例：

```
Calendar Date1 = Calendar.getInstance();
Date1.set(2001,11,10);
Calendar Date2 = Calendar.getInstance();
Date2.set(2001,11,11);
System.out.println(Date1.after(Date2)); //结果false
```

Calendar 的 Setlenient方法，检查日期是否存在

当Setlenient方法的参数设置成false时，会进行检查。抛出illegalArgumentException异常

求程序执行时间：

```
java.util.Date
long sTime = new Date().getTime(); //当前时间的毫秒数，最小毫秒
long bs=Calendar.getInstance().getTimeInMillis(); //
```

很精确

第三章 对象(物件)

名词

对象：

类： 一类属性相同的对象

属性：是什么样

方法：能做什么(C 中叫作函数)

对象：

声明：Student s ;

这时我们只是说明s是一个能够指向Student类型的引用(相当于C++中的指针),并没有创建一个对象。

所以我们此时不能对s做任何操作。

初始化：s = new Student();

向系统申请一块存储空间(地址空间)，该地址空间保存的是一个Student类型的数据。

而s中保存的就是该地址空间的首地址。

变量：内存空间中一块具有固定长度的，用来保存数据的地址空间。(s也是一个变量)

一个对象可以有多个引用指向。

Student[] s = new Student[3] 只是相当于声明一个长度为 3 的Student类型的数组。

实例变量和局部变量

实例变量：

1、在一个类中，任何方法之外定义的变量；

2、从面向对象的思想来说我们又把实例变量看成一个类的属性。

3、实例变量在没有符初值时系统会自动帮我们做初始化：

整型数据初始化为 0，布尔型数据初始化为 false，对象类型初始化为 null。

实例变量的作用域在本类中完全有效，当被其它的类调用的时候也可能有效。

局部变量：

1、在方法内定义的变量叫局部变量。

2、局部变量使用前必须初始化，系统不会自动给局部变量做初始化。

3、局部变量的生命范围在他所在的代码块，在重合的作用域范围内不允许两个局部变量命名冲突。

注：局部变量与实例变量允许同名，在局部变量的作用域内，其优先级高于实例变量。

我们可以用 this.实例变量名 以区分局部变量。

第四章 数组

数组：

数组也是对象

数组中保存着多个相同类型的元素

数组中的每一个元素都是变量

可以创建数组对象，但数组里只能放对象的引用，不能直接放对象进去

数组的创建：

1. 声明一个int数组变量，数组变量是数组对象的遥控器

```
int[] nums; //这数组不可以赋值，因为还没为它分配空间。
```

注意： int[] nums = new nums[]{}; 这种创建方法创出来的数组也不可以赋值。因为它的长度为零。

上面两种方法创建的数组，使用时都需要分配空间： nums = new int[]{1,25,4};或 nums=new int[4]再赋值。

2. 创建大小为7的数组，并将它赋值给变量nums

```
nums = new int[7];
```

3. 赋于int数组每一个元素一个int值

```
nums[0] = 6; nums[1] = 34; nums[2] = 23; nums[3] = 4;
```

多维数组：

1. 定义方式：type 维数 arrayName;

如： int[][] b = new int [2] [1];

2. 分配内存空间,有两种方法：

直接为每一维分配空间: int[][] a = new int[2][3];

分别为每一维分配空间 int[][] a = new int[2][ ]; //列数可以没有，行数则一定要有

```
a[0] = new int[3]; a[1] = new int[5]; //a[][] 看成一维数组
```

可以为每行设置为空间大小不同的数组。

3. 初始化,有两种方式：

先定义数组，分配空间，然后直接对每个元素进行赋值(一个个写，或用for函数)

在定义数组的同时进行初始化。

如：int a[][] = {{2,3}, {1,5}, {3,4}};

java实质上把多维数组看作一维数组，但数组里的元素也是一个数组，即数组的数组

多维数组的长度 = 行数； (a.length=行数； a[0].length=列数)

创建数组对象的另外几种方式：

Int[] nums = {6,34,23,4,15,0, 57}; (java 形式)

这方法只能在初始化定义的时候可以，以后再想定义nums={……}就不行了

Int[] nums = new int[] {6,34,23,4,15,0, 57};

这句的后一个 int[] 内不能填数字，怕人弄错数目；

这句可以先 int[] nums;以后再另外定义 nums = new int[]{……}

[]可以换换位置，如：

Int nums[]; (C 和 C++ 形式)

注意： short [] z [] []; //这是合法的，定义一个三维数组

声明数组时，不能定义其大小；只有 new 数组时可以定大小。

数组元素的默认值：

byte short int long 为 0

float double 为 0.0

char 为 '\0'

boolean 为 false

引用类型为null

数组的 length 属性：

表示数组的长度，是指这个数组最多能保存的元素个数

length属性只能被读取，不能被修改

java.lang.ArrayIndexOutOfBoundsException: (这是数组下标越界的报错)

随机数：

Math.random(); //可以产生随机的0~1 的小数,不需导包

java.util.Random; //可以产生更加多种的随机数

0~100的一个随机整数(包括0，但不包括100)：

Double d = 100*Math.random(); int r = d.intValue(); //方法一

Random r = new Random(); int num = r.nextInt(100); //方法二；需要 import java.util.Random;

可以直接在程序中写这句，而临时导入 int i = new java.util.Random().nextInt(100);

java.lang.Math:

求绝对值： Math.abs(数值);

求平方根： Math.sqrt(数值);

求最大值： Math.max(数值1,数值2);

求最小值： Math.min(数值1,数值2);

Arrays.sort(数组名)

排序算法。需导入 impor java.util.Arrays;

数组的拷贝:

1. 用 for 语句，将数组的元素逐个赋值。直接如果直接将数组 a = 数组b;则是将b的指针赋给a

2. 用System.arraycopy();

arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

src - 源数组。

srcPos - 源数组中的起始位置。

dest - 目标数组。

destPos - 目标数据中的起始位置。

length - 要复制的数组元素的数量。

如：System.arraycopy(a, 0, b, 0, a.length); //把数组 a 全部复制到数组 b 中

在java中对面向对象(OO)的要求

1. 对象是客观存在的，万物皆对象。

(注：看不见的对象并不表示该对象不存在，比如说事件)；

2. 简单性：采用面向对象方法可以使系统各部分各司其职各尽所能。

3. 复用性：对象的功能越简单其可重用性越高。

4. 弱耦合性：各司其职各尽所能。

5. 高内聚性：一个对象独立完成一个功能的能力

6. 类是一类事务的共性，是人类主观认识的一种抽象，是对象的模板。

面向过程与面向对象的对比

面向过程：先有算法，后有数据结构。先考虑怎么做。

面向对象：先有数据结构，后有算法。先考虑用什么做。

第六章 构造方法

方法的声明(分为五个部分)

1. 方法的修饰符(可以有多个，且顺序无关)

2. 方法的返回值类型

3. 方法名

4. 方法的参数列表

如果方法有参数,一定要以正确的数量、类型、和顺序传递参数;可以将变量当作参数传入,但要类型相符

5. 方法允许抛出的例外(异常)

注：编译器只能做语法上的检查，而不能进行逻辑上的检查。

Java中不允许有废话，永远不会执行的语句不允许写。

1. 声明格式:

<modifiers><return_type><name>([argument_list>])[throws <exception>]{}

例如：public String getName(){return name;}

2．当没有返回值时，返回类型必须被定义为 void。

3．构造方法没有返回类型。

4．返回类型必须与方法名相邻，其它修饰符号可以调换位置。

参数传递

在java方法传参过程中简单类型是按值传递，对象类型是按引用传递。

按值传递传递的是数据的副本。

按引用传递 传递的是保存该数据的地址

Java语言总是使用传值调用,这意味着方法得到的只是所有参数值的拷贝。

因此，方法不能修改传递给它的任何参数变量的内容。

对象类型的参数传递的也是该对象的引用值

方法中并不能改变对象变量，但能通过该变量调用对象的方法或修改对象的成员。

方法的参数基本上与局部变量相同，但你不需要直接初始化它

编译器会确保方法调用时会有与声明相符的参数传进来，且参数会自动被赋值

形参 VS 实参：

形参(形式参数)：相当于函数(Java中也把函数称之为方法)中的局部变量

在函数被调用时创建，并以传入的实参作为起始值，函数调用结束时被释放

不会影响主程序中其它的变量(即使有变量跟他们同名),因为他们是不同作用域的变量，互不干扰。

实参：调用函数时，实际传给函数形式参数的数据。

重载(Overload)

在同一个类中，允许同时存在一个以上的同名函数，只要他们的参数列表不同即可。

参数列表不同，可以是参数的类型或个数不同，也可以是不同类型参数的顺序不同。

1、相同方法名，不同参数表。

2、方法重载时，对于参数的匹配有个向上就近原则。(这样可以节省栈空间资源)；

3、为什么面向对象中要有方法重载？

方法的重载使同一类方法由于参数造成的差异对于对象的使用者是透明的。

对象的使用者只负责把参数交给对象，而具体怎么实现由对象内部决定。

4、Java中的运算符重载

java中唯一重载的运算符是String类型的“+”号，任何类型+String类型结果都为Stirng类型。

5、注意点：重载不仅出现在同一个类中，也可以出现在父子类中。

重载的方法只是刚好有相同名字的不同方法

方法的覆盖 (Override) 重写

继承之后，想改变由父类继承下来的方法。

1. 同样的方法名、参数列表、返回类型(从Java 5 起，返回类型可以是子类型)

2. 访问权限不能更小

3. 异常不能更宽 (可以抛出更少或是更窄的检查异常，或者任何非检查异常)

重构 (extract Method)

消除代码的重复，提高代码的可维护性。整理凌乱的代码，把可重用的代码块包装起来。

常用重复的方法封装成工具类(工具类太多则做成工具箱),一般都是静态方法和常量，没有属性。

在eclipse里,选中要重构的代码,右键Refactor-Extract Mathod或(Shift+Alt+M)

创建对象的步骤

1、分配空间

2、初始化属性

3、调用构造方法

注:构造方法不能手工调用,在对象的生命周期内构造方法只调用一次。

构造方法 (参考day05的 TestCat.java)

构造方法是在生成对象的过程中调用的方法，但构造方法并不能创建对象。

new 对象的时候需要调用构造方法。

1、特点：没有返回值(连void也没有)，方法名与类名相同。(如果加上 void 会变成普通方法。)

2、在不写构造方法时，系统会自动生成一个无参的构造方法。

3、请养成在每个类中自己加上无参构造方法的习惯。

格式为：public ClassName(){}

构造方法也可以是其它的限制符--private protected default

private 一般用在 singleton 模式中。

在一个对象的生成周期中构造方法只用一次，一旦这个对象生成，那么这个构造方法失效。

* 接口不能创建实例，因为没有构造方法

可以构造多个构造方法，但多个构造方法的参数表一定不同，或参数顺序不同

即属于不同的构造方法：> 构造方法的重载

使用构造方法来初始化对象的状态：把初始化代码放到构造方法中，并且把构造方法设定成需要参数的

编译器一定会帮你写出没有参数的构造方法吗？不会

如果你已经写了一个有参数的构造方法，并且你需要一个没有参数的构造方法，则你必须自己动手写

如果类有一个以上的构造方法,则参数列表一定要不一样，我们可以认为这几个构造方法形成重载关系

如果我们提供了有参的构造方法，那么系统不会再提供无参的构造方法了。

这样当被子类继承时，如果子类构造方法不人为调用父类的有参构造方法就会出现异常。

构造方法可以通过 this 调用另外一个构造方法(this 此时必须在第一行语句)

匿名对象：

创建对象时，直接调用对象的方法而不定义对象的句柄。

如： person p1 = new person; p1.shout();

改写成： new person.shout(); //此方法执行完，此匿名对象也就变成了垃圾。

使用匿名对象的情况：

1. 此对象只需要一次方法调用。

2. 此对象作为实参传给一个函数调用。

this 当前对象 (参考day05 的TestThis.java)

谁调用该方法，在这一时刻谁就是该方法的当前对象；是个隐式参数，代表被构造的对象。

用this来区分实例变量和局部变量。

this.实例变量名 = 局部变量名 (将局部变量赋值给实例变量)

this()表示调用本类的其它构造方法，且只能放在一个方法中的第一行第一句。

构造方法可以通过this调用另外一个构造方法(this此时必须在第一

行语句)

*super 关键字也是个隐形参数，代表被构造对象的父类。

同样也必须在构造方法的第一行

对象和对象引用的区别

对象好比一台电视机，对象引用好比电视机遥控。对象引用 中存的是对象的地址。

多个对象引用中存放的是同一个地址，表示该对象被多个对象引用所引用。

面向对象的三大特性：

封装(Encapsulation)、继承(Inheritance)、多态 polymiorphism

封装：

1. 定义：指一个对象的内部状态对外界是透明的，对象与对象之间只关心对方有什么方法，而不关心属性。

封装使实现的改变对架构的影响最小化。封装后的代码更具有安全性、可扩展性和可维护性。

2. 原则：封装使对象的属性尽可能的私有，根据需要配上相应的get/set方法，对象的方法尽可能的公开。

该隐藏的一定要隐藏，该公开的一定要公开。

3. 方法公开的是声明而不是实现。使方法实现的改变对架构的影响最小化。

4. 访问权限控制从严到宽

private :仅本类成员可见

default :本类＋同包类可见(默认)

protected:本类＋同包＋不同包的子类

public :公开

注：这里的同包指的是跟父类所在的包相同。

5、完全封装：属性全部私有，并提供相应的get/set方法。

优点：

1. 事物的内部实现细节隐藏起来

2. 对外提供一致的公共的接口——间接访问隐藏数据

3. 可维护性

一、继承：

1 定义：基于一个已存在的类构造一个新类。

继承已存在的类就是复用这些类的方法和属性，在此基础上，还可以在新类中添加一些新的方法和属性。

2 父类到子类是从一般到特殊的关系。

3 继承用关键字extends

dog extends Animal :表示狗类继承了动物类

4 Java中只允许单继承(java简单性的体现)

父子类之间的关系是树状关系。(而多继承是网状关系)

5 父类中的私有属性可以继承但是不能访问。

也可以说父类中的私有属性子类不能继承。

6 原则：父类放共性，子类放个性。

7 构造方法不能被子类继承。

父类的成员能否继承到子类?

private：本类内部可以访问 不能继承到子类

(default)：本类内部可以访问，同包其它类也可以访问

能否继承到子类？ 不一定：同包的可继承，不同包则不可继承。

protected：本类内部可以访问，不同包的子类也可以访问， 同包其它类也可以访问

能继承到子类

public：任何地方都可以访问 能继承到子类

继承的意义：

1. 避免了重复的程序代码，提高了程序的可重用性

2. 定义出共同的协议

二、带继承关系的对象创建的过程

1. 递归的构造父类对象

2. 分配空间

3. 初始化属性

4. 调用本类的构造方法

三、super 关键字

1. super()表示调用父类的构造方法

2. super()也和this一样必须放在构造方法的第一行第一句。不是构造方法则不用第一行。

3. super.表示调用父类的方法或属性。例：super.m();

4. super 可以屏蔽子类属性和父类属性重名时带来的冲突

5. 在子类的构造函数中如果没有指定调用父类的哪一个构造方法，就会调用父类的无参构造方法，即super()

指向父类的引用

super.age

super.addAge()

调用父类的构造方法

super();

super(“wangcai”,8);

一个对象的创建过程

1．当构造一个对象的时候，系统先构造父类对象，再构造子类对象。

2．构造一个对象的顺序：(注意：构造父类对象的时候也是这几步)

递归地创建父类的 static 成员(即使只调用其子类静态成员，也会先创建父类静态成员)；

顺序地创建本类的 static 成员(只要调用这个类的属性或方法都需创建一次)；

递归地创建父类对象(先创建父类非静态成员，再调用父类构造方法)；

顺序地创建本类非静态成员(包括属性和方法)；

调用本类的构造方法(它可调用本类或父类的成员，也可调用本类的其它构造方法)。

创建完成了(更多详情参看下面：类加载的顺序)

四、白箱复用和黑箱复用

1．白箱复用：又叫继承复用，子类会继承父类所有的东西，

从某种程度上说白箱复用破坏了封装。是一种 is a 的关系。

```
例：class Liucy{
public void teachCpp(){System.out.println("Teach Cpp");}
public void chimogu(){ }
}
class Huxy extends Liucy{}
```

2、黑箱复用：又叫组合复用，是一种 has a 的关系。

```
例：class Liucy{
public void teachCpp(){System.out.println("Teach Cpp");}
public void chimogu(){ }
}
class Huxy {
private Liucy liucy = new Liucy();
public void teachCpp(){liucy.teachCpp();}
}
```

原则：组合复用取代继承复用原则。

使我们可以有机会选择该复用的功能。

多态

1．定义：是指一个对象可以有多种形态，换句话说多态使我们可以把一个子类对象看作是一个父类对象类型

(例：father A = new child())。

多态指的是编译时的类型变化，而运行时类型不变。

2．多态分为两种：编译时多态和运行时多态。

编译时类型：定义时类型(主观概念)把它看作什么。

运行时类型：真实类型(客观概念) 实际上他是什么。

重载是编译时多态，覆盖是运行时多态。在方法重载的情况下，参数类型决定于编译时类型。

3．作用：在需要一类对象的共性时，可以很容易的抽取。并且可以屏蔽不同子类对象之间所不关心的差异。

多态方便写出更通用的代码，以适应需求的不断变化

4．多态常见的用法：

(1)、多态用在方法的参数上

(2)、多态用在方法的返回类型上

5．运行时多态的三原则：

(1)、对象不变(改变的是主观认识)

(2)、对于对象的调用只能限于编译时类型的方法。

(3)、在程序的运行时，动态类型判定。运行时调用运行时类型，即他调用覆盖后的方法。

注意：多态时，只有一般方法是动态调用的；而 static 方法和 final 方法依然是静态调用的；属性全是静态调用的。

如：father A = new child(); 则A的一般方法是child的方法；但A的属性是father的属性。

同样：child C = new child(); 则 (father)C 的一般方法还是child的方法，但属性是father的属性。

if(cat instanceof Animal){ …… } //如果cat属于Animal类型则执行

强制类型转换，在迫不得已的时候再用。因为很容易出错。

第八章：高级语言特性

静态变量 static

一个类只有一个静态变量，跟对象没有关系。被类的所有实例共享；如果子类没有覆盖，也共享父类的静态成员。

一般直接使用类名来访问 “类名.静态变量名”。可以在没有任何实例时调用。

在某种意义上类似于全局变量(Java里没有全局变量，这只是C和C++的说法)

不能在 static 方法或代码块里访问非 static 成员(变量或方法)

能继承和覆盖，但覆盖 static 方法必须也是 static 的方法。

1．可以修饰属性、方法、初始代码块，成为类变量、静态方法、静态初始化代码块。

注：初始代码块是在类中而不是在任何方法之内的代码块。

2．类变量、静态方法、静态初始化代码块与具体的某个对象无关，只与类相关，是全类公有的。 在类加载时初始化。

3．类加载：JVM通过CLASSPATH找到字节码文件，并将字节码文件中的内容通过I/O流读到JVM并保存的过程

在虚拟机的生命周期中一个类只被加载一次。

注：Java命令的作用是启动JVM (Java Virtual Mechine)。

4．tatic定义的是一块为整个类共有的一块存储区域，其发生变化时访问到的数据都是经过变化的。

5．为什么主方法必须是静态的？

主方法是整个应用程序的入口，JVM只能通过类名去调用主方法。

6．类变量和静态方法可以在没有对象的情况下用：类名.方法名(或属性名)来访问。

7．静态方法不可被覆盖(允许在子类中定义同名的静态方法,但是没有多态)

父类如果是静态方法,子类不能覆盖为非静态方法。父类如果是非静态方法，子类不能覆盖为静态方法。

争论：静态方法可以覆盖但是没有多态。

思考：没有多态的覆盖叫覆盖吗？

在静态方法中不允许调用本类中的非静态成员。

8．静态初始化代码块只在类加载的时候运行一次，以再也不执行了。所以静态代码块一般被用来初始化静态成员。

9．不加static为动态初始化代码块，在创建对象时被调用(在构造函数之前)。

10．最后要注意的一点就是 static 不能修饰局部变量。

什么时候类加载

第一次需要使用类信息时加载。

类加载的原则：延迟加载，能不加载就不加载。

触发类加载的几种情况：

(1)、调用静态成员时，会加载静态成员真正所在的类及其父类。

通过子类调用父类的静态成员时，只会加载父类而不会加载子类。

(2)、第一次 new 对象的时候 加载(第二次再 new 同一个类时，不需再加载)。

(3)、加载子类会先加载父类。

注：如果静态属性有 final 修饰时，则不会加载，当成常量使用。

例：public static final int a =123;

但是如果上面的等式右值改成表达式(且该表达式在编译时不能确定其值)时则会加载类。

例：public static final int a = math.PI

如果访问的是类的公开静态常量，那么如果编译器在编译的时候能确定这个常量的值，就不会被加载；

如果编译时不能确定其值的话，则运行时加载

类加载的顺序：

1.加载静态成员/代码块：

先递归地加载父类的静态成员/代码块(Object的最先)；再依次加载到本类的静态成员。

同一个类里的静态成员/代码块，按写代码的顺序加载。

如果其间调用静态方法，则调用时会先运行静态方法，再继续加载。同一个类里调用静态方法时，可以不理会写代码的顺序。

调用父类的静态成员，可以像调用自己的一样；但调用其子类的静态成员，必须使用“子类名.成员名”来调用。

2.加载非静态成员/代码块：

先递归地加载父类的非静态成员/代码块(Object的最先)；再依次加载到本类的非静态成员。

同一个类里的非静态成员/代码块，按写代码的顺序加载。同一个类里调用方法时，可以不理会写代码的顺序。

但调用属性时，必须注意加载顺序。一般编译不通过，如果能在加载前调用，值为默认初始值(如：null 或者 0)。

调用父类的非静态成员(private 除外)，也可以像调用自己的一样。

3.调用构造方法：

先递归地调用父类的构造方法(Object的最先)；默认调用父类空参的，也可在第一行写明调用父类某个带参的。

再依次到本类的构造方法；构造方法内，也可在第一行写明调用某个本类其它的构造方法。

注意：如果加载时遇到 override 的成员，可看作是所需创建的类型赋值给当前类型。

其调用按多态用法：只有非静态方法有多态；而静态方法、静态属性、非静态属性都没有多态。

假设子类override父类的所有成员，包括静态成员、非静态属性和非静态方法。

由于构造子类时会先构造父类；而构造父类时，其所用的静态成员和非静态属性是父类的，但非静态方法却是子类的；

由于构造父类时，子类并未加载；如果此时所调用的非静态方法里有成员，则这个成员是子类的，且非静态属性是默认初始值的。

成员，包括变量和方法

成员变量，包括实例变量和静态变量

final 修饰符

(最终的、最后的)当final修饰时，不能被改变，不能被继承

1. final 可以用来修饰类、属性和方法、局部变量。

2. final 修饰一个属性时，该属性成为常量。

(1)对于在构造方法中利用final进行赋值时，此时在构造之前系统设置的默认值相对于构造方法失效。

(2)对于实例常量的赋值有两次机会

在初始化的时候通过声明赋值

在构造的时候(构造方法里)赋值

注：不能在声明时赋值一次，在构造时再赋值一次。

注意：当final修饰实例变量时，实例变量不会自动初始化为0；但必须给他赋值才能通过编译。

3. final修饰方法时，该方法成为一个不可覆盖的方法。这样可以保持方法的稳定性。

如果一个方法前有修饰词private或static，则系统会自动在前面加上final。

即 private 和 static 方法默认均为 final 方法。

4. final 常常和 static、public 配合来修饰一个实例变量，表示为一个全类公有的公开静态常量。

例： public static final int a = 33;

在这种情况下属性虽然公开了，但由于是一个静态常量所以并不算破坏类的封装。

5. final 修饰类时，此类不可被继承，即final类没有子类。

一个 final 类中的所有方法默认全是final方法。

final 不能修饰构造方法，构造方法不能被继承更谈不上被子类方法覆盖。

关于 final 的设计模式：不变模式

1、不变模式：一个对象一旦产生就不可能再修改(string 就是典型的不变模式)；

通过不变模式可以做到对象共享；

2、池化思想：用一个存储区域来存放一些公用资源以减少存储空间的开销。

有池的类型:boolean,byte,int,short,long,char,(池范围在-127~128之间)

(float,double 等小数没有池)

例：在String类中有个串池(在代码区)。

池：堆里的一片独立空间。目的是拿空间换时间，让运算效率更高。

(1)如果用Stirng str = “abc” 来创建一个对象时，则系统会先在“串池”中寻找有没有“abc”这个字符串

如果有则直接将对象指向串池中对应的地址，如果没有则在串池中创建一个“abc”字符串。

所以：String str1 = “abc”;

String str2 = “abc”;

Str1 == str2 返回值是ture;他们的地址是一样的。

也就是说str1和str2都指向了代码空间中相同的一个地址，而这个地址空间保存就是是字符串"abc"

字符串是不可改变的类型，所以可以共享。所以串池里不会有相同的两个字符串。

(2)如果用String str = new String("abc")则直接开辟一块内存放"abc"这个字符串。

所以上面这语句，创建两个"abc"，一个在池，一个是对象

String str2 = new String("abc");

Str == str2 返回值是false;他们的地址是不一样的。

即是说str和str2分别指向了堆空间中不同的两个地址，而这两个地址空间保存的都是字符串"abc"

StringBuffer 类(java.lang下的)。

对于字符串连接

String str=”1”+”2”+”3”＋”4”;

产生：

12 123 1234

这在串池中产生多余对象，而我们真正需要的只有最后一个对象，这种方式在时间和空间上都造成相当大的浪费。

所以我们应该使用 StringBuffer(线程安全的) 或者 StringBuilder(线程不安全的)来解决

解决方案：

```
StringBuffer sb = new StringBuffer(“1”);
Sb.append(“2”);
Sb.append(“3”);
Sb.append(“4”);
S = sb.toString();
```

解决后的方案比解决前在运行的时间上快2个数量级。

StringBuilder (1.5版本后出现的)

线程不安全的，在多线程并发时会出现问题。但仍是字符串合并的首选。

运行效率比 StringBuffer 快一倍。

abstract 抽象

1. 可用来修饰类、方法

2. abstract 修饰类时，则该类成为一个抽象类。

抽象类不可生成对象(但可以有构造方法留给子类使用),必须被继承使用。

抽象类可以声明，作为编译时类型，但不能作为运行时类型。

abstract 永远不会和 private,static,final 同时出现。(因为抽象必须被继承。)

3. abstract 修饰方法时，则该方法成为一个抽象方法，抽象方法不能有实现；由子类覆盖后实现。

比较：private void print(){};表示方法的空实现

abstract void print();表示方法为抽象方法，没有实现

4. 抽象方法从某中意义上来说是制定了一个标准，父类并不实现，留给子类去实现。

注：抽象类中不一定要有抽象方法，但有抽象方法的类一定是抽象类。

抽象类可以有抽象方法和非抽象方法。实现抽象类的第一个具体类必须实现其所有抽象方法。

5. 关于抽象类的设计模式：模板方法

灵活性和不变性

interface 接口

1、定义：接口不是类，而是一组对类需求的描述，这些类要遵从接口描述的统一格式进行定义。

定义一个接口用关键字 interface。

例：public interface a{……}

2、接口是一种特殊的抽象类。

在一个接口中，所有的方法为公开、抽象的方法，所有的属性都是公开、静态、常量。

所以接口中的所有属性可省略修饰符：public static final。也只能用这三个修饰符。

接口中所有的方法可省略修饰符：public abstract。但这些都是默认存在的。

3、一个类实现一个接口必须实现接口中所有的方法，否则其为一抽象类。而且实现类的方法需要 public

实现接口用关键字 implements.

所谓实现一个接口就是实现接口中所有的方法。

例：class Aimple implements A{……..};

4、一个类除了继承另一个类外(且只能继承一个类),还可以实现多个

接口(接口之间用逗号分割)。

接口可以实现变相的多继承。

例：class Aimple extends Arrylist implements A,B,C{…}

5、不能用“new 接口名”来实例化一个接口，但可以声明一个接口。

6、接口与接口之间可以多继承。

例：interface face1 extends face2,face3{}

接口的继承相当于接口的合并

7、接口的作用

(1)、间接实现多继承。

用接口来实现多继承并不会增加类关系的复杂度。因为接口不是类,是在类的基础上的再次抽象。

父类：主类型 接口：副类型

典例：父亲(主) 和 老师(副)

(2)、允许我们为一个类定义出混合类型。

(3)、通过接口制定标准

接 口：标准的定义 定义标准

接口的调用者：标准的使用 使用标准

接口的实现类：标准的实现 实现标准

接口的回调:先有接口的使用者，再有接口的实现者，最后把接口的实现者的对象传到接口的使用者中，

并由接口的使用者通过接口来调用接口实现者的方法。

例：sun公司提供一套访问数据库的接口(标准)，java程序员访问数据库时针对数据库接口编程。

接口由各个数据库厂商负责实现。

(4)、解耦合作用：采用接口可以最大限度的做到弱耦合，将标准的实现者与标准的制定者隔离

(例：我们通过JDBC接口来屏蔽底层数据库的差异)

8、接口的编程设计原则

(1)、尽量针对接口编程(能用接口就尽量用接口)

(2)、接口隔离原则(用若干个小接口取代一个大接口)

这样可以只暴露想暴露的方法，实现一个更高层次的封装。

9、注意点：

(1)、一个文件只能有一个 public 接口，且与文件名相同。

(2)、在一个文件中不能同时定义一个 public 接口和一个 public 类。

(3)、接口与实体类之间只有实现关系，没有继承关系；

抽象类与类之间只有继承关系没有实现关系。接口与接口之间只有继承

关系，且允许多继承。

(4)、接口中可以不写 public,但在子类实现接口的过程中 public 不可省略。

接口 VS 抽象类

1、接口中不能有具体的实现，但抽象类可以。

2、一个类要实现一个接口必须实现其里面所有的方法，而抽象类不必。

3、通过接口可以实现多继承，而抽象类做不到。

4、接口不能有构造方法，而抽象类可以。

5、实体类与接口之间只有实现关系，而实体类与抽象类只有继承关系

抽象类与接口之间既有实现关系又有继承关系。

6、接口中的方法默认都是公开抽象方法，属性默认都是公开静态常量，而抽象类不是。

修饰符的使用情况：

(Y表可用；不写表示不可用)

修饰符 类 属性 方法 局部变量(所有局部变量都不能用修饰符)

public Y Y Y

protected Y Y

(default) Y Y Y

private Y Y

static Y Y

final Y Y Y Y

abstract Y Y

访问权限控制：

修饰符 同一个类 同一个包 (不同包)子类 其它包

public Y Y Y Y

protected Y Y Y

(default) Y Y

private Y

Object类

1、object类是类层次结构的根类，他是所有类默认的父类。

2、object类中的其中三个方法。

(1)、finalize()

当一个对象被垃圾收集的时候，最后会由JVM调用这个对象的finalize方法；

注意：这个方法一般不用，也不能将释放资源的代码放在这个方法里；只有调用C程序时，才可能要用到

(2)、toString()

存放对象地址的哈希码值。

返回一个对象的字符串表示形式。打印一个对象其实就是打印这个对象

toString方法的返回值。

可以覆盖类的toString()方法，从而打印我们需要的数据。 Public String toString(){……}

(3)、equals(Object obj)

用来判断对象的值是否相等。前提是覆盖了equals方法。Object类中的equals方法判断的依然是地址

注意：String类已经覆盖了equals方法，所以能用equals来判断String对象的值是否相等。

下面是覆盖equals方法的标准流程：

```
/**********************************************************
*********************/
    public boolean equals(Object obj){
    //第一步：现判断两个对象地址是否相等
    if(this == obj) return true;
    //第二步：如果参数是null的话直接返回false;
    if(obj == null) return false;
    //第三步：如果两个对象不是同一个类型直接返回false
    if (getClass() != obj.getClass()) return false;
    //??
if(!(this.getClass.getName().equals(o.getClass.getName
())) return false;
    //第四步：将待比较对象强转成指定类型，然后自定义比较规则
    Student s = (Student) obj;
    if(s.name.equals(this.name)&&s.age==this.age)
return true;
    else return false;
    }

/**********************************************************
*********************/
```

覆盖equals的原则： 1.自反性(自己＝自己)、 2.对称性(y=x则x=y)、

3.一致性(多次调用，结果一致)、 4.传递性(A=B,B=C则A=C)。

非空原则： t1.equals(Null)返回False；(如果t1不等于空)

(4)、clone 克隆，拷贝

一个对象参与序列化过程，那么对象流会记录该对象的状态，当再次序列化时，

会重复序列化前面记录的对象初始状态，我们可以用对象克隆技术来解决这个问题

1 类中覆盖父类的clone方法，提升protected为public

2 类实现Cloneable接口

浅拷贝：只简单复制对象的属性

深拷贝：如果被复制对象的属性也是一个对象,则还会复制这个属性对象

这种复制是一个递归的过程,通过对象序列化实现

《Java5.0新特性》

四大点(枚举、泛型、注释、..)；5小点(包装类、静态应用、可变长参数、for-each、..)

一、自动装箱 和 自动解箱技术

装箱Autoboxing，也翻译作 封箱；解箱Unautoboxing(也译作 解封)

1、自动装箱技术：编译器会自动将简单类型转换成封装类型。

2、编译器会自动将封装类型转换成简单类型

3、注意：自动装箱和自动解箱只会在必要的情况下执行。

int 能隐式提升成 long；但Integer不能隐式提升成Long，只能提升成Number

封装之后就成类，只能由子类转成父类；而Integer和Long是Number的不同子类。

如： int i; short s; Integer II; Short SS;

可以 i=SS; 但不可以 II=s; //赋值时，右边的数先转成左边数的对应类型，再进行隐式类型提升

二、静态引用概念：

用 import static 节省以后的书写。

引入静态属性 import static java.lang.System.out;

引入静态方法 import static java.lang.Math.random;

import static 只能引入静态的方法或属性；不能只引入类或非静态的方法。

如：import static java.lang.System.*;

out.println(“a”); //等于System.out.println("a"); 由于out是一个字段，所以不能更节省了

如果 import static java.lang.System.gc; 则可以直接在程序中用 gc(); //等于System.gc();

三、可变长参数

一个方法的参数列表中最多只能有一个可变长参数,而且这个变长参数必须是最后一个参数

方法调用时只在必要时去匹配变长参数。

/**********变长参数的例子

```
**************************************/
    importstaticjava.lang.System.*;//节省书写,System.out直接写out
    public class TestVararg {
    public static void main(String…… args){
    m();
    m("Liucy");
    m("Liucy","Hiloo");
    }
    static void m(String……
s){out.println("m(String……)");}
    //s可以看作是一个字符串数组String[] s
    static void m(){out.println("m()");}
    static void m(String s){out.println("m(String)");}
    } //m(String…… s) 是最后匹配的

/**************************************************************
*****/
```

四、枚举 enum

1、定义：枚举是一个具有特定值的类型，对用户来说只能任取其一。

对于面向对象来说时一个类的对象已经创建好，用户不能新生枚举对象，只能选择一个已经生成的对象。

2、枚举本质上也是一个类。枚举值之间用逗号分开，以分号结束(如果后面没有其它语句，分号可不写)。

3、枚举分为两种：类型安全的枚举模式和类型不安全的枚举模式

4、枚举的超类(父类)是:Java.lang.Enum。枚举是 final 类所以不能继承或被继承。但可以实现接口。

枚举中可以写构造方法，但构造方法必需是私有的，而且默认也是 私有的 private

5、一个枚举值实际上是一个公开静态的常量，也是这个类的一个对象。

6、枚举中可以定义抽象方法，但实现在各个枚举值中(匿名内部类的方式隐含继承)

由于枚举默认是 final型，不能被继承，所以不能直接用抽象方法(抽象方法必须被继承)

在枚举中定义抽象方法后，需要在自己的每个枚举值中实现抽象方法。

枚举是编译期语法，编译后生成类型安全的普通类

values()静态方法，返回枚举的元素数组

name方法

```
/**************************************************************
```

```
*****/
    final class Season1{ //用 final 不让人继承
    private Season1(){} //用 private 构造方法，不让人 new 出
来
    public static final Season1 SPRING=new Season1("春");
    public static final Season1 SUMMER=new Season1("夏");
    public static final Season1 AUTUMN=new Season1("秋");
    public static final Season1 WINTER=new Season1("冬");
    String name; //将"春夏秋冬"设为本类型，而不是24种基本类型，
为防止值被更改
    private Season1(String name){
    this.name=name;
    }
    public String getName(){
    return this.name;
    }}
    /********上面是以前版本时自定义的枚举，下面是新版的枚举写法
********/
    enum Season2{
    SPRING("春"), SUMMER("夏"), AUTUMN("秋"), WINTER("冬
");
    final String name;
    Season2(String name){ this.name=name; }
    public String getName(){return this.name;}
    }
    //注意：枚举类是有序的；如：Season2.SPRING.ordinal(),从0
开始
    enum Season3{
    SPRING, SUMMER, AUTUMN, WINTER;
    }

/**********************************************************
*****/
    /*******关于枚举的例子
**************************************/
    import static java.lang.System.*;
    public class TestTeacher {
    public static void main(String[] args) {
    for(TarenaTeacher t:TarenaTeacher.values()){
    t.teach();
    }}}
    enum TarenaTeacher{
    LIUCY("liuchunyang"){void
```

```
teach(){out.println(name+" teach UC");}},
    CHENZQ("chenzongquan"){void
teach(){out.println(name+" teach C++");}},
    HAIGE("wanghaige"){void teach(){out.println(name+"
teach OOAD");}};
    String name;
    TarenaTeacher(String name){this.name=name;}
    abstract void teach();
    }

/**********************************************************
*****/
    enum Animals {
    DOG ("WangWang") , CAT("meow") , FISH("burble");
    String sound;
    Animals ( String s ) { sound = s; }
    }
    class TestEnum {
    static Animals a;
    public static void main ( String[] args ) {
    System.out.println ( a.DOG.sound + " " + a.FISH.sound );
    }}

/**********************************************************
*****/
```

五、新型 for 循环 for-each，用于追求数组与集合的遍历方式统一

1、数组举例：

```
int[] ss = {1,2,3,4,5,6};
for(int i=0; i l = new ArrayList();
```

:表示该集合中只能存放String类型对象。

3、使用了泛型技术的集合在编译时会有类型检查，不再需要强制类型转换。

```
String str = l.get(2); //因为List l, 所以 Error
```

注：一个集合所允许的类型就是这个泛型的类型或这个泛型的子类型。

4、List<Number> l = new ArrayList //Error

```
List l = new ArrayList //Right
```

必须类型一致，泛型没有多态

5、泛型的通配符<?>

泛型的通配符表示该集合可以存放任意类型的对象。但只有访问，不可以修改。

```
    static void print( Cllection<?> c ){
    for( Object o : c )
    out.println(o);
    }
```

6、带范围的泛型通配符

泛型的声明约定T表示类型，E表示元素

(1)、上界通配符，向下匹配：<?extendsNumber>表明“extends”或“implements”，认为是 final 的

表示该集合元素可以为Number类型及其子类型(包括接口)，例如Number,Integer,Double

此时集合可以进行访问但不能修改。即不允许调用此对象的add,set等方法；但可以使用 for-each 或 get.

(2)、下界通配符，向上匹配：<? super Number>

表示该集合元素可以为Number类型及其父类型，直至 Object。

可以使用 for-each,add,addAll,set,get等方法

(3)、接口实现：<? extends Comparable>

表示该集合元素可以为实现了Comparable接口的类

7、泛型方法

在返回类型与修饰符之间可以定义一个泛型方法，令后面的泛型统一

这里只能用 extends 定义，不能用 super ；后面可以跟类(但只能有一个，且要放在首位)其余是接口

符号只有 & //“&”表示“与”；逗号表示后面的另一部分

静态方法里面，不能使用类定义的泛型，只能用自己定义的；因为静态方法可以直接调用；

所以普通方法可以使用类定义的及自己定义的泛型

```
public static <T> void copy(T[] array,Stack<T> sta){……}
public static <T,E extends T> void copy (T[] array,Stack<E> sta){…..}
public static <T extends Number&Comparable> void copy(List<T> list,T[] t);
```

8、不能使用泛型的情况：

(1)、带泛型的类不能成为 Throwable 类和 Exception 类的子类

因为cathc()中不能出现泛型。

(2)、不能用泛型来 new 一个对象

如：T t = new T();

(3)、静态方法不能使用类的泛型，因为静态方法中没有对象的概念。

9、在使用接口的时候指明泛型。

```
class Student implements Comparable{…….}
```

10、泛型类

```
/**********************************************************************/
class MyClass<T>{
```

```
    public void m1(T t){}
    public T m2(){
    return null;
    }}

/**********************************************************
***************/
```

第九章：

内部类(nested classes) (非重点)

1. 定义：定义在其它类中的类，叫内部类(内置类)。内部类是一种编译时的语法,编译后生成

的两个类是独立的两个类。内部类配合接口使用,来强制做到弱耦合(局部内部类，或私有成员内部类)。

2.内部类存在的意义在于可以自由的访问外部类的任何成员(包括私有成员),但外部类不能直接访问内部类的

成员。所有使用内部类的地方都可以不使用内部类;使用内部类可以使程序更加的简洁(但牺牲可读性),

便于命名规范和划分层次结构。

3. 内部类和外部类在编译时是不同的两个类，内部类对外部类没有任何依赖。

4. 内部类可用 static,protected 和 private 修饰。(而外部类只能使用 public 和 default)。

5. 内部类的分类：成员内部类、局部内部类、静态内部类、匿名内部类。

(注意：前三种内部类与变量类似，可以对照参考变量)

① 成员内部类(实例内部类)：作为外部类的一个成员存在，与外部类的属性、方法并列。

成员内部类可看作外部类的实例变量。

在内部类中访问实例变量：this.属性

在内部类访问外部类的实例变量：外部类名.this.属性。

对于一个名为outer 的外部类和其内部定义的名为inner 的内部类。

编译完成后出现outer.class 和outer$inner.class 两类。

不可以有静态属性和方法(final 的除外),因为 static 在加载类的时候创建,这时内部类还没被创建

如果在外部类的外部访问内部类，使用out.inner.***

建立内部类对象时应注意：

在创建成员内部类的实例时，外部类的实例必须存在：

在外部类的内部可以直接使用inner s=new inner(); 因为外部类

知道 inner 是哪个类。

而在外部类的外部，要生成一个内部类对象，需要通过外部类对象生成。

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

相当于：Outer out = new Outer(); Outer.Inner in = out.new Inner();

错误的定义方式：Outer.Inner in=new Outer.Inner()。

② 局部内部类：在方法中定义的内部类称为局部内部类。

类似局部变量，不可加修饰符 public、protected 和 private，其范围为定义它的代码块。

可以访问外部类的所有成员，此外，还可以访问所在方法中的 final 类型的参数和变量。

在类外不可直接生成局部内部类(保证局部内部类对外是不可见的)。

要想使用局部内部类时需要生成对象，对象调用方法，在方法中才能调用其局部内部类。

局部内部类不能声明接口和枚举。

③ 静态内部类：(也叫嵌套类)

静态内部类定义在类中，在任何方法外，用 static 定义。

静态内部类能直接访问外部类的静态成员；

不能直接访问外部类的实例成员，但可通过外部类的实例(new 对象)来访问。

静态内部类里面可以定义静态成员(其它内部类不可以)。

生成(new)一个静态内部类不需要外部类成员，这是静态内部类和成员内部类的区别。

静态内部类的对象可以直接生成： Outer.Inner in=new Outer.Inner();

对比成员内部类：Outer.Inner in = Outer.new Inner();

而不需要通过生成外部类对象来生成。这样实际上使静态内部类成为了一个顶级类。

静态内部类不可用 private 来进行定义。例子：

对于两个类，拥有相同的方法：

```
/**********************************************************
**********************/
    /*class People{void run();}
    interface Machine{void run();}
    此时有一个robot类：
    class Robot extends People implement Machine.
    此时run()不可直接实现。*/
    interface Machine{ void run();}
```

```
class Person{ void run(){System.out.println("run");}}
class Robot extends Person{
private class MachineHeart implements Machine{
public void run(){System.out.println("heart run");}
}
public void run(){System.out.println("Robot run");}
Machine getMachine(){return new MachineHeart();}
}
class Test{
public static void main(String[] args){
Robot robot=new Robot();
Machine m=robot.getMachine();
m.run();
robot.run();
}}
```

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

注意：当类与接口(或者是接口与接口)发生方法命名冲突的时候，此时必须使用内部类来实现。

这是唯一一种必须使用内部类的情况。

用接口不能完全地实现多继承，用接口配合内部类才能实现真正的多继承。

④ 匿名内部类：

【1】匿名内部类是一种特殊的局部内部类，它是通过匿名类实现接口。

【2】不同的是他是用一种隐含的方式实现一个接口或继承一个类，而且他只需要一个对象

【3】在继承这个类时，根本就没有打算添加任何方法。

【4】匿名内部类大部分情况都是为了实现接口的回调。

注：匿名内部类一定是在 new 的后面

其隐含实现一个接口或实现一个类,没有类名,根据多态,我们使用其父类名。

因其为局部内部类，那么局部内部类的所有限制都对其生效。

匿名内部类是唯一一种无构造方法类。

注：这是因为构造器的名字必须和类名相同，而匿名内部类没有类名。

匿名内部类在编译的时候由系统自动起名Out$1.class。

因匿名内部类无构造方法，所以其使用范围非常的有限。

结尾需加上分号。

匿名内部类的例子：

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

```
**********************/
    public class test{
    public static void main(String[] args){
    B.print(new A(){
    public void
getConnection(){ System.out.println("Connection……");}
    });
    }}
    interface A{ void getConnection();}
    class B{
    public static void print(A a){ a.getConnection();}
    }

/**************************************************
**********************/
```

枚举和接口可以在类的内部定义，但不能在方法内部定义。

接口里面还可以定义多重接口和类。

类放在什么位置，就相当于什么成员。

内部类的用途：

封装类型：把标准公开，把标准的实现者作为内部类隐藏起来，

强制要求使用者通过标准访问标准的实现者，从而强制做到弱耦合!

直接访问外部类的成员

配合接口，实现多继承，当父类和接口方法定义发生冲突的时候，就必须借助内部类来区分

模板回调

从内部类继承：

由于直接构造实例内部类时，JVM会自动使内部类实例引用它的外部类实例。

但如果下面Sample类通过以下方式构造对象时：Sample s = new Sample()；

JVM无法决定Sample实例引用哪个Outer实例，为了避免这种错误的发生，在编译阶段，java编译器会要求Sample类的构造方法必须通过参数传递一个Outer实例的引用，然后在构造方法中调用super语句来建立Sample实例与Outer实例的关联关系。

```
/**************************************************
**********************/
    public class Sample extends Outer.Inner{
    //public Sample(){} //编译错误
    public Sample(Outer o){ o.super(); }
    public static void main(String args[]){
    Outer outer1=new Outer(1);
```

```
    Outer outer2=new Outer(2);
    Outer.Inner in=outer1.new Inner();
    in.print();
    Sample s1=new Sample(outer1);
    Sample s2=new Sample(outer2);
    s1.print(); //打印a=1
    s2.print(); //打印a=2
    }}
/**********************************************************
**********************/
```

内部接口：

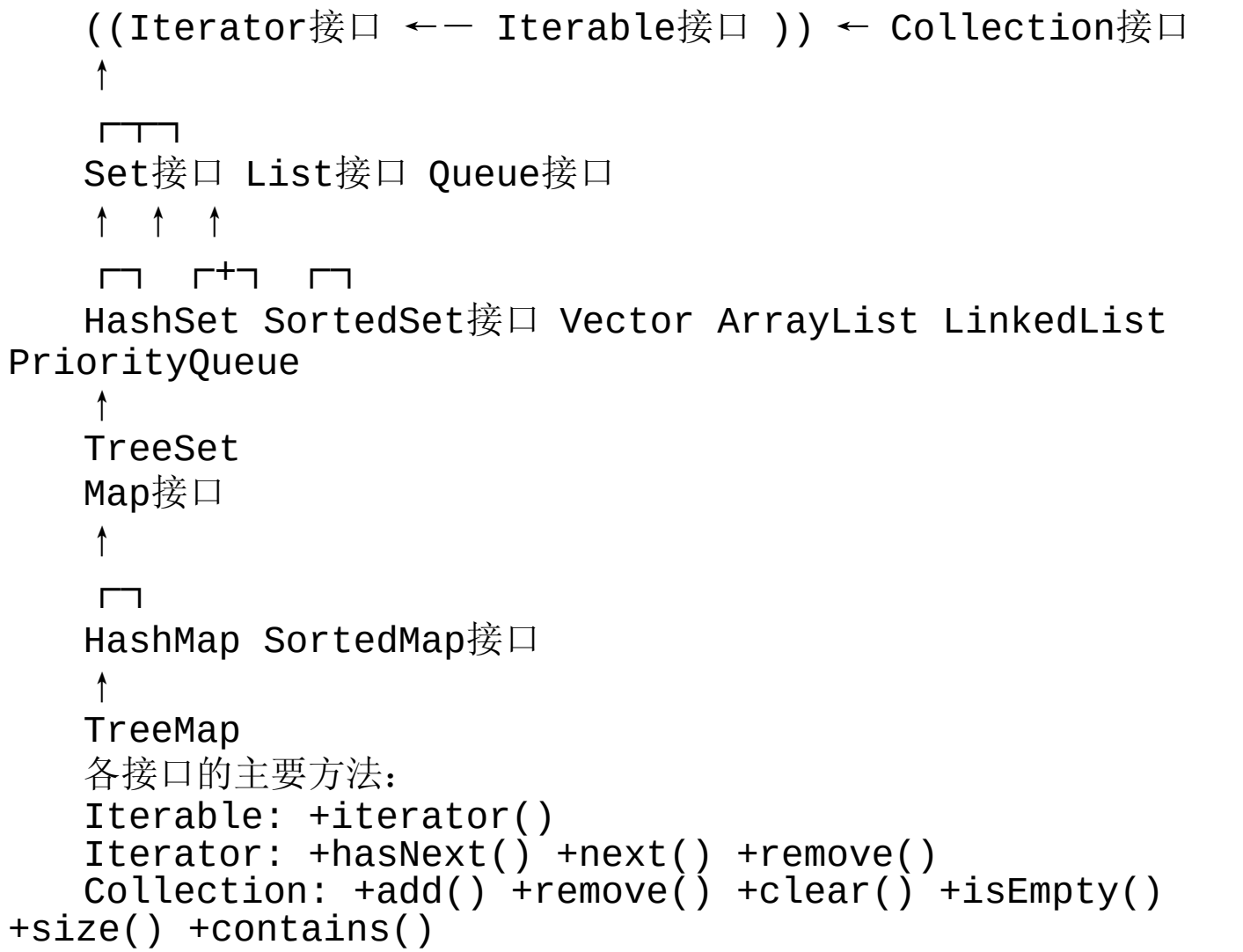
在一个类中也可以定义内部接口

在接口中可以定义静态内部类，此时静态内部类位于接口的命名空间中。

在接口中还可以定义接口，这种接口默认也是public static 的，如Map.Entry就是这种接口

第八章 Collection FrameWork

《集合框架》



各接口的主要方法：

Iterable: +iterator()

Iterator: +hasNext() +next() +remove()

Collection: +add() +remove() +clear() +isEmpty() +size() +contains()

List: +get() +set() +remove()
Queue: +element() +offer() +peek() +poll()
Set:
SortedSet: +comparator() +first() +last() +headSet() +tailSet()
Map: +clear() +containsKey() +containsValue() +get() +keySet()
+isEmpty() +remove() +put()会替换重复键 +size() +values()
SortedMap: +comparator() +firstKey() +lastKey() +headMap() +tailMap()

一、集合(容器，持有对象)：是一个用于管理其它多个对象的对象，且只能保存对象的引用，不是放对象。

1、Collection： 集合中每一个元素为一个对象，这个接口将这些对象组织在一起，形成一维结构。

2、List： 有序、可重复。

ArrayList： 数组。查询快，增删慢。(List是链表)

Vector： 线程安全，但效率很差(现实中基本不用)

3、Set： 无序，且不可重复(不是意义上的重复)。(正好与List 对应)

HashSet： 用 hashCode() 加 equals() 比较是否重复

SortedSet： 会按照数字将元素排列，为“可排序集合”默认升序。

TreeSet： 按二叉树排序(效率非常高)； 按Comparable接口的compareTo() 比较是否重复

4、Map： 其中每一个元素都是一个键值对(Key-Value)。键不能重复。可有一个空键。

SortedMap： 根据 key 值排序的 Map。

HashMap： 用 hashCode() 加 equals() 比较是否重复

5、Queue： 队列：先进先出。

PriorityQueue： 优先队列：元素按照其自然顺序进行排序，或者根据构造队列时提供的 Comparator 进行排序

注意：在“集合框架”中，Map 和Collection 没有任何亲缘关系。

Map的典型应用是访问按关键字存储的值。它支持一系列集合操作，但操作的是键-值对，而不是独立的元素

因此 Map 需要支持 get() 和 put() 的基本操作，而 Set 不需要。

《常用集合列表》

' 存放元素 存放顺序 元素可否重复 遍历方式 排序方式 各自实现类

List Object 有序可 迭代 (2) ArrayList, TreeSet

Set Object 无序 不可 迭代 SortedSet HashSet

SortedSet Object 无序 不可 迭代 已排序 TreeSet

Map (1) Key无序 Key不可,value可 对Key迭代 SortedMap HashMap

SortedMap (1) 无序,有排序 Key不可,value可 对Key迭代 已对键值排序 TreeMap

(1)Object(key)—Object(value);

(2)Collections.sort();

注：以上有序的意思是指输出的顺序与输入元素的顺序一致

HashSet、HashMap通过hashCode()，equals()来判断重复元素

在java中指定排序规则的方式只有两种：1、实现java.util包下的Comparator接口

2、实现java.lang包下的Comparable接口

二、迭代器：Iterator

1、使用Iterator接口方法，您可以从头至尾遍历集合，并安全的从底层Collection中除去元素

2、remove() 由底层集合有选择的支持。底层集合支持并调用该方法时，最近一次next()返回的元素将被删

3、Collection 接口的iterator() 方法返回一个Iterator

4、Iterator中的hasNext()用于判断元素右边是否有数据，返回True则有。然后就可以调用next()动作。

5、Iterator中的next()方法会将游标移到下一个元素，并返回它所跨过的元素。(通常这样遍历集合)

6、用于常规Collection 的Iterator 接口代码如下：

```
/**** 迭代遍历 *****/
List l = new ArrayList();
Iterator it = l.iterator();
while(it.hasNext()){
Object obj = it.next();
System.out.println(obj);
}
//遍历Vector
Vector v = new Vector();
for(int index=0; index<v.size(); index++) {
String str = (String) v.elementAt(index); //需知道Vector里包含的类型，否则强转出错
System.out.println(str);
}
```

注：工具类是指所有的方法都是公开静态方法的类。

Java.util.collections就是一个工具类；

三、对集合的排序

1、我们可以用Java.util.collections中的sort(List l)方法对指定的List集合进行排序；

但是如果List中存放的是自定义对象时，这个方法就行不通了，必须实现Comparable接口并且指定排序规则。

这里我们再来看一下sort(List l)方法的内部实现；

```
/**********************************************************
*****/
    class Collections2{
    public static void sort(List l){
    for(int i=0;i<l.size()-1;i++){
    for(int j=i+1;j<l.size();j++){
    Object o1 = l.get(i);
    Object o2 = l.get(j);
    Comparable c1 = (Comparable)o1;
    Comparable c2 = (Comparable)o2;
    if(c1.compareTo(c2)>0){
    Collections.swap(l,i,j);
    }}}}} //其实用的算法就是个冒泡排序。

/**********************************************************
*/
```

2、实现Java.lang.Comparable接口，其实就是实现他的 public int compareTo(Object obj)方法；

比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数。

其规则是当前对象与obj 对象进行比较，其返回一个 int 值，系统根据此值来进行排序。

如当前对象 > obj，则返回值>0；

如当前对象 = obj，则返回值=0；

如当前对象 < obj，则返回值<0。

注意：String类型已经实现了这个接口，所以可以直接排序；

```
/**********************************************************
*/
    class Student implements Comparable{
    private String name;
    private int age;
    public Student(String name, int age) {
    this.name = name;
    this.age = age;
    }
    public int compareTo(Object obj) {
```

```
    Student s = (Student)obj;
    return s.age-this.age;
    }}

/**********************************************************
*/
```

四、ArrayList和LinkedList集合

1、ArrayList 底层是object 数组，所以ArrayList 具有数组的查询速度快的优点以及增删速度慢的缺点。

Vector 底层实现也是数组，但他是一个线程安全的重量级组件。

2、而在LinkedList 的底层是一种双向循环链表。

在此链表上每一个数据节点都由三部分组成：

前指针(指向前面的节点的位置)、 数据、 后指针(指向后面的节点的位置)。

最后一个节点的后指针指向第一个节点的前指针，形成一个循环。

3、双向循环链表的查询效率低但是增删效率高。所以LinkedList 具有查询效率低但增删效率高的特点。

4、ArrayList 和LinkedList 在用法上没有区别，但是在功能上还是有区别的。

LinkedList 经常用在增删操作较多而查询操作很少的情况下：队列和堆栈。

队列：先进先出的数据结构。

堆栈：后进先出的数据结构。

(堆栈就是一种只有增删没有查询的数据结构)

注意：使用堆栈的时候一定不能提供方法让不是最后一个元素的元素获得出栈的机会。

LinkedList 提供以下方法：(ArrayList 无此类方法)

addFirst(); +removeFirst(); +addLast(); +removeLast();

在堆栈中，push 为入栈操作，pop 为出栈操作。

Push 用addFirst()；pop 用removeFirst()，实现后进先出。

用isEmpty()--其父类的方法，来判断栈是否为空。

在队列中，put 为入队列操作，get 为出队列操作。

Put 用addFirst()，get 用removeLast()实现队列。

List 接口的实现类 Vector 与ArrayList 相似，区别是Vector 是重量级组件，消耗的资源较多。

结论：在考虑并发的情况下用Vector(保证线程的安全)。

在不考虑并发的情况下用ArrayList(不能保证线程的安全)。

5、面试经验(知识点):

java.util.stack(stack即为堆栈)的父类为Vector。可是stack的父类是最不应该为Vector 的。

因为Vector的底层是数组,且Vector有get方法(意味着它可能访问任意位置的元素，很不安全)。

对于堆栈和队列只能用push 类和get 类。(这是早期的某个java编写工程师的失误造成)

Stack 类以后不要轻易使用。实现堆栈一定要用LinkedList。

(在JAVA1.5 中，collection 有queue 来实现队列。)

五、HashSet集合

1、HashSet是无序的，没有下标这个概念。HashSet集合中元素不可重复(元素的内容不可重复);

2、HashSet 底层用的也是数组。

3、HashSet如何保证元素不重复？Hash算法和equals方法。

当向数组中利用add(Object obj)添加对象的时候，系统先找对象的hashCode:

int hc=obj.hashCode(); 返回的hashCode 为整数值。

int I=hc%n;(n 为数组的长度)，取得余数后，利用余数向数组中相应的位置添加数据，以n 为6 为例，

如果I=0则放在数组a[0]位置，如果I=1则放在数组a[1]位置。

如果equals()返回true，则说明数据重复。如果equals()返回false，则再找其它的位置进行比较。

这样的机制就导致两个相同的对象有可能重复地添加到数组中，因为他们的hashCode 不同。

如果我们能够使两个相同的对象具有相同hashcode，才能在equals()返回为真。

在实例中，定义student 对象时覆盖它的hashcode。

因为String类会自动覆盖，所以比较String 类的对象时，不会出现相同的string 对象的情况。

现在，在大部分的JDK 中，都已经要求覆盖了hashCode。

结论:如将自定义类用hashSet 来添加对象，一定要覆盖hashcode()和equals(),

覆盖的原则是保证当两个对象hashcode 返回相同的整数，而且equals()返回值为True。

如果偷懒，直接将hashCode方法的返回值设为常量；虽然结果相同，但会多次地调用equals()，影响效率。

我们要保证相同对象的返回的hashCode 一定相同，也要保证不相同

的对象的hashCode 尽可能不同

(因为数组的边界性，hashCode 还是有极微几率相同的)。

六、TreeSet集合

1、TreeSet是SortedSet的实现类TreeSet通过实现Comparable接口的compareTo来实现元素不重复。

2、TreeSet由于每次插入元素时都会进行一次排序，因此效率不高。

3、java.lang.ClassCastException是类型转换异常。

4、在我们给一个类用CompareTo()实现排序规则时

5、从集合中以有序的方式抽取元素时，可用TreeSet，添加到TreeSet的元素必须是可排序的。

“集合框架”添加对Comparable 元素的支持。

一般说来，先把元素添加到HashSet，再把集合转换为TreeSet 来进行有序遍历会更快。

七、Map

1、HashMap集合

(1)HashMap就是用hash算法来实现的Map

(2)在实际开发中一般不会用自定义的类型作为Map的Key。做Key的无非是八中封装类。

(3)HashMap的三组操作：

【1】改变操作，允许从映射中添加和除去键-值对。键和值都可以为null。

不能把Map作为一个键或值添加给自身。

-Object put(Object key, Object value)
-Object remove(Object key)
-void clear()

【2】查询操作允许您检查映射内容：

-Object get(Object key)
-intsize()
-boolean isEmpty()

【3】最后一组方法允许您把键或值的组作为集合来处理。

-public Set KeySet();
-public Collection values()

(4)HashMap和HashTable的区别等同于ArrayList和Vector的区别。

只不过HashTable中的Key和Value不能为空，而HashMap可以。

(5)HashMap底层也是用数组，HashSet底层实际上也是HashMap，HashSet类中有HashMap属性(查API)。

HashSet 实际上为(key.null)类型的HashMap。有key 值而没有

value 值。

2、HashMap 类和TreeMap 类

?集合框架提供两种常规Map 实现：HashMap和TreeMap。

?在Map 中插入、删除和定位元素，HashMap 是最好选择。

?如果要按顺序遍历键，那么TreeMap 会更好。

?根据集合大小，先把元素添加到HashMap，再把这种映射转换成一个用于有序键遍历的TreeMap 可能更快。

?使用HashMap 要求添加的键类明确定义了hashCode() 实现。

?有了TreeMap 实现，添加到映像的元素一定是可排序的

?HashMap和TreeMap 都实现Cloneable 接口。

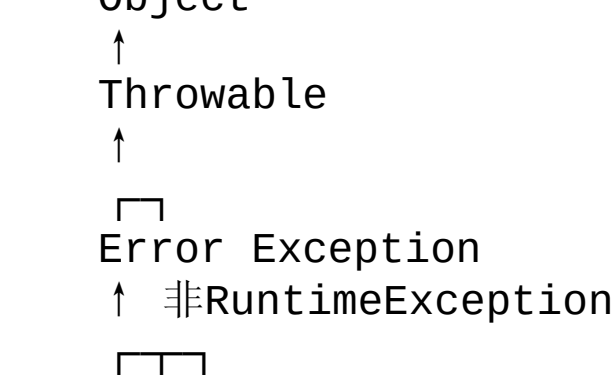
```
/******* 遍历hashMap **********/
Map map = new HashMap();
//方法一： (会比方法二快一倍)
for (Iterator iter = map.entrySet().iterator();
iter.hasNext();) {
Map.Entry entry = (Map.Entry) iter.next();
Object key = entry.getKey();
Object val = entry.getValue();
}
//方法二：
Iterator it = map.keySet().iterator();
while(it.hasNext())
{
Object key =it.next();
Object value =map.get(key);
}
/********** 结束 *****************/
```

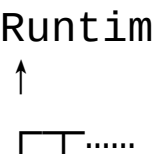
异常 Exception

1. 概念： JAVA将所有的错误封装成为一个对象，其根本父类为Throwable。异常处理可以提高我们系统的容错性。

Throwable 有两个子类：Error 和Exception。

Error:一般是底层的不可恢复的错误。





2. Exception分类：

Runtime exception(未检查异常)和 非Runtime exception(已检查异常)。

未检查异常是因为程序员没有进行必要的检查,因为他的疏忽和错误而引起的异常。可避免。

几个常见的未检查异常：

①java.lang.ArithmeticException //如：分母为0;

②java.lang.NullPointerException //如：空指针操作;

③java.lang.ArrayIndexoutofBoundsExceptio//如：数组越界；数组没有这元素;

④java.lang.ClassCastException //如：类型转换异常;

已检查异常是不可避免的，对于已检查异常必须处理。

3、异常对象的传递。

当一个方法中出现了异常而又没做任何处理，则这个方法会返回该异常对象。

异常依次向上层调用者传递，直到传到JVM，虚拟机退出。（用一句话说就是沿着方法调用链反向传递）

应该在合适的位置处理异常，不让它一直上抛到 main 方法，应遵循的规则：

谁知情谁处理，谁负责谁处理，谁导致谁处理。

4、如何来处理异常(这里主要是针对已检查异常)

【1】 throws 消极处理异常的方式。

方法名(参数表)throws 后面接要往上层抛的异常。

表示该方法对指定的异常不作任何处理，直接抛往上一层。

【2】积极处理方式 try、catch

try {可能出现错误的代码块} catch(exception e){进行处理的代码} ;

一个异常捕获只会匹配一次 try,catch.

一个异常一旦被捕获就不存在了。

catch 中要求必须先捕获子类异常再捕获父类异常。 catch 要求有零到多个，但异常的名字不能重复。

【3】 finally (紧接在 catch 代码块后面)

finally 的代码块是无论如何都会被执行的(除非虚拟机退出)，所以

里面一般写释放资源的代码。

return 也无法阻止 finally,但System.exit(0):退出虚拟机则可以。

有 finally 的 try/catch 流程

如果 try 块失败了，抛出异常，程序马上转移到 catch 块，完成后执行 finally 块，再执行其后程序。

如果 try块成功，程序跳过 catch块并去到 finally块，finally块完成后，继续执行其后程序。

如果 try 或 catch 块有 return 语句, finally 还是会执行；程序会跳去执行 finally 块然后再回到 return 语句

```
/********** 使用try/catch的例子
*****************************************/
class MyException{
void myException(){
System.out.println("MyException");
// if(1==1)return; //测试没有异常时
System.out.println(1/0);
}
public static void main(String[] args) {
MyException me = new MyException();
try{
System.out.println("try");
// if(1==1)return;
me.myException();
}catch(ArithmeticException ae){
System.out.println("cath");
}finally{
System.out.println("finally");
}}}

/*****************************************************
*********************/
```

5、自定义异常(与一般异常的用法没有区别)

```
class MyException extends Exception{
public MyException(String message){ super(message);}
public MyException(){}
}
```

6、如何控制 try 的范围

根据操作的连动性和相关性，如果前面的程序代码块抛出的错误影响了后面程序代码的运行，

那么这个我们就说这两个程序代码存在关联，应该放在同一个 try中。

7、不允许子类比父类抛出更多的异常。

8、断言：只能用于代码调试时用。

一般没什么用，只在测试程序时用

```
/********* 断言的列子
**********************************************************
/
public class TestAssertion {
public static void main(String[] args){
int i = Integer.parseInt(args[0]);
assert i==1:"ABCDEFG"; //格式： assert (布尔表达式/布尔值) : String;
/*断言语句:(表示断言该boolean语句返回值一定为真，如果断言结果为false就会报Error错误)
“ : ”后面跟出现断言错误时要打印的断言信息。 */
System.out.println(i);
}}
//javac -source 1.4 TestAssertion.java //表示用1.4的新特性来编译该程序。
//java -ea TestAssertion 0 //表示运行时要用到断言工具

/**********************************************************
**********************/
```

Module 10 Reflection And Annotation

一、 Reflection 反射

1、反射主要用于工具的开发。所有的重要Java技术底层都会用到反射。反射是一个底层技术。

是在运行时动态分析或使用一个类的工具(是一个能够分析类能力的程序)

2、反射使我们能够在运行时决定对象的生成和对象的调用。

3、Class

(1)定义：在程序运行期间，Java运行时系统始终为所有的对象维护一个被称为运行时的类型标识。

虚拟机利用类型标识选用相应的方法执行。

可以通过专门的类访问这些信息。保存这些信息的类被称为Class(类类)

(2)类对象(类类:用于存储和一个类有关的所有信息),用来描述一个类的类。

类信息通过流读到虚拟机中并以类对象的方式保存。

一个类的类对象在堆里只有一个。

注：简单类型也有类对象。

在反射中凡是有类型的东西，全部用类对象来表示。

4. 获得类对象的3种方式：

(1)通过类名获得类对象 Class c1 = String.class; //类.Class;

(2)通过对象获得类对象 Class c2 = s.getClass(); //类对象.getClass();

(3)通过Class.forName("类的全名")来获得类对象 //Class.forName(包名.类名);

Class c3 = Class.forName("Java.lang.String");//这会强行加载类到内存里，前两种不加载

注：第三种方式是最常用最灵活的方式。第三种方式又叫强制类加载。

5. java.lang.reflect .Field 对象，描述属性信息。

6. java.lang.reflect .Constructor 描述构造方法信息。

7. java.lang.reflect .Method 描述方法信息。

8. 在反射中用什么来表示参数表？

Class[] cs2 = {StringBuffer.class};//表示一个参数表

Constructor c = c1.getConstructor(cs2);//返回一个唯一确定的构造方法。

Class[] cs2 = {String.class,int.class}

Method m = c1.getMethod(methodName,cs3);

9. 可以通过类对象来生成一个类的对象。

Object o = c.newInstance();

10、反射是一个运行时的概念。反射可以大大提高程序的通用性。

一个关于反射的例子：

```
/*************************************************************
****/
    import java.lang.reflect.*;
    public class TestClass2 {
    public static void main(String[] args) throws
Exception{
    //0.获得在命令行输入的类的类对象
    Class c=Class.forName(args[0]);//需处理异常
(ClassNotFoundException),输入类的全名(如: com.Student)
    //Object o=c.newInstance();//空参构造方法
    //1.得到构造方法对象
    Class[] cs1={String.class};
    Constructor con=c.getConstructor(cs1);
    //2.通过构造方法对象去构造对象
```

```
    Object[] os1={args[1]};//输入构造方法
    Object o=con.newInstance(os1);
    //3.得到方法对象
    String methodName=args[2];//输入方法名(如: study)
    Class[] cs2={String.class, int.class};//列表里对应各参数的类型
    Method m=c.getMethod(methodName,cs2);
    //4.调用方法
    Object[]os2={args[3],args[4]};//输入方法的各个参数(如: CoreJava,5)
    m.invoke(o,os2);
    /* 以上相当于知道类的情况时，这样直接用
    Student s=new Student("Liucy");
    s.study("CoreJava", 5); */
    }}

/**********************************************************
*****/
```

下面是用反射调用私有方法的一个例子:

```
/**********************************************************
*****/
    import java.lang.reflect.Method;
    public class TestClass2 {
    public static void main(String[] args) throws Exception{
    Class c = Class.forName("test.AA");// 字符串为类的全名,如 com.my.AA
    Method[] m = c.getDeclaredMethods();// 读取它的全部方法
    //逐个的读取私有方法,由于是私有方法无法以名称读取
    for (int i=0; i<m.length; i++)
    {
    Method m1 = m[i];// 拿其中的第i个方法
    if ( !"print".equals(m1.getName()) ) continue;//相当于按名称来取
    m1.setAccessible(true);// 把private的属性设成可访问，否则不能访问
    Class[] cs2 = {String.class, int.class};// 列表里对应各参数的类型
    Object[] os2 = {"test", 5};// 输入方法的各个参数
```

```
    m1.invoke(c.newInstance(), os2);
    }
    }}
    class AA{
    private void print(String name, int i){
    System.out.println("name:" + name + " i:" + i);
    }}

/**********************************************************
*****/
```

要求学会的内容：

概念：类类，类对象，类的对象，对象类(Object类)

类对象：Class，指向类的对象。

类对象包括：属性对象 Feild，方法对象Method，构造方法对象Constructor。

类对象能做什么：探查类定义的所有信息：父类，实现的接口，所有属性及方法，以及构造方法。

类的修饰符，属性以及方法的修饰符，方法的返回类型，方法的

……

构造一个类的对象(类对象.newInstance())

强制修改和访问一个对象的所有属性(包括私有属性)

调用一个对象的方法(普通方法，静态方法)

Method.invoke(方法所在的对象(类对象，null),给方法传参数

……

构造数组的另一种用法(动态构造数组，不定长度)

注释 Annotation

1、定义：Annotation描述代码的代码(给机器看的)。

区别：描述代码的文字，给人看的，英语里叫Comments。

任何地方都可以使用Annotation注释，它相当于一段代码，可用来作自动检测。

一个注释其实是一种类型(类class，接口interface，枚举enum，注释Annotation)，注释本质上是接口。

定义注释 public @interface Test{}，注释都是Annotation接口的子接口

2、注释的分类：

(1)、标记注释：没有任何属性的注释。@注释名

(2)、单值注释：只有一个属性的注释。@注释名(value="***")

在单值注释中如果只有一个属性且属性名就是value，则"value＝"可以省略。

(3)、多值注释：有多个属性的注释。多值注释又叫普通注释。

@注释名(多个属性附值，中间用逗号隔开)

3、内置注释(java.lang)：

(1)、@Override(只能用来注释方法)

表示一个方法声明打算重写超类中的另一个方法声明。

如果方法利用此注释类型进行注解但没有重写超类方法，则编译器会生成一条错误消息。

(2)、@Deprecated

有 @Deprecated 注释的程序元素，不鼓励程序员使用，通常是因为它很危险或存在更好的选择。

在使用不被赞成的程序元素或在不被赞成的代码中执行重写时，编译器会发出警告。

(3)、@SuppressWarnings(抑制警告，该注释效果与版本相关)

指示应该在注释元素(以及包含在该注释元素中的所有程序元素)中取消显示指定的编译器警告。

4、自定义注释

(1)、定义注释类型

自定义注释默认就是java.lang.annotation.Annotation接口的子接口。注释本质上就是一个接口。

```
public @interface TestAnnotation {}
```

(2)、为注释添加注释

```
import java.lang.annotation.*;
@Documented //能在帮助文档里出现
@Inherited //能否被继承下去
@Retention(value = {RetentionPolicy.RUNTIME}) //注释该注释运行时仍然保留
//@Retention默认是CLASS(保留到编译期)，最短期是SOURCE(原代码级，编译时丢弃)

@Target(value={ElementType.METHOD,ElementType.FIELD})
/*用来注释该注释能用来注释方法和属性,还可以定义它用来注释其它的，如类、注释、构造方法等等*/
/*如果不写Target，默认是可以注释任何东西*/
public @interface TestAnnotation {……}
```

(3)、为注释添加属性方法

```
import java.lang.annotation.*;
@Target(value={ElementType.TYPE})
public @interface TestAnnotation {
```

//如果一个注释不是标记注释，则还要定义属性；这属性同时也是方法，但不可能有参数，只可以有默认值

String parameter() default "liucy";

//给属性方法parameter添加一个默认值"liucy"

//parameter()括号里不能写其它东西，类型只能是24种基本类型之一

//24种类型：8种基本数据类型、String、枚举、注释、Class、以及它们的一维数组

}

@TestAnnotation("haha")

public class MyClass{……}

5、注释的注释：(元注释 meta annotation)

都在 java.lang. annotation 包中

(1)、Target：指示注释类型所适用的程序元素的种类。

一个注释只能出现在其该出现的位置，Target是给注释定位的。

例：@Target(value = {ElementType.METHOD}); //说明该注释用来修饰方法。

(2)、Retention：指示注释类型的注释要保留多久。

如果注释类型声明中不存在 Retention 注释，则保留策略默认为 RetentionPolicy.CLASS。

例：Retention(value = {RetentionPolicy.xxx})

当x为CLASS表示保留到类文件中，运行时抛弃。

当x为RUNTIME表示运行时仍保留(最常用)

当x为SOURCE时表示编译后丢弃。

(3)、Documented：指示某一类型的注释将通过 javadoc 和类似的默认工具进行文档化。

应使用此类型来注释这些类型的声明：其注释会影响由其客户端注释的元素的使用。

(4)、Inherited：指示注释类型被自动继承。

如果在注释类型声明中存在 Inherited 元注释，并且用户在某一类声明中查询该注释类型，

同时该类声明中没有此类型的注释，则将在该类的超类中自动查询该注释类型。

注：在注释中，一个属性既是属性又是方法。

使用注释

/***/

Class c = Class.forName(args[0]);

Object o = c.newInstance();

```
Method[] ms = c.getMethods();
for(Method m:ms){
//判断m方法上有没有Test注释
if (m.isAnnotationPresent(Test.class)){
//得到m之上Test注释parameter属性值
Test t=m.getAnnotation(Test.class);
String parameter=t.parameter();
m.invoke(o,parameter);
}}
/**********************************************/
```

图型界面(非重要：不常用、难学)

1、Awt：抽象窗口工具箱，它由三部分组成：

①组件：界面元素；

②容器：装载组件的容器(例如窗体)；

③布局管理器：负责决定容器中组件的摆放位置。

2、图形界面的应用分四步：

① 选择一个容器：

⑴window:带标题的容器(如Frame)；

⑵Panel:面板通过add()向容器中添加组件。

注：Panel不能作为顶层容器。

Java 的图形界面依然是跨平台的。但是调用了窗体之后只生成窗体；必须有事件的处理，关闭按钮才工作。

②设置一个布局管理器：用setLayout()；

③向容器中添加组件；

jdk1.4用getContentPare()方法添加主件。

③ 添加组件的事务处理。

Panel 也是一种容器：但是不可见的，很容易忘记设置它们的可见性。

```
Panel pan=new Panel;
Fp.setLayout(null);//表示不要布局管理器。
```

3、五种布局管理器：

(1)、Flow Layout(流式布局)：按照组件添加到容器中的顺序，顺序排放组件位置。

默认为水平排列，如果越界那么会向下排列。排列的位置随着容器大小的改变而改变。

```
FlowLayout layout = new FlowLayout(FlowLayout.LEFT);//
```

流式布局，可设对齐方式

Panel 默认的布局管理器为Flow Layout。

(2)、BorderLayout：会将容器分成五个区域：东西南北中。

语句：Button b1=new Botton(“north”);//botton 上的文字

f.add(b1,”North”);//表示b1 这个botton 放在north 位置

f.add(b1, BorderLayout.NORTH);//这句跟上句是一样的效果，不写方位默认放中间，并覆盖

注：一个区域只能放置一个组件，如果想在一个区域放置多个组件就需要使用Panel 来装载。

Frame 和Dialog 的默认布局管理器是Border Layout。

(3)、Grid Layout(网格布局管理器)：将容器生成等长等大的条列格，每个块中放置一个组件。

f.setLayout GridLayout(5,2,10,10)//表示条列格为5 行2 列，后面为格间距

(4)、CardLayout(卡片布局管理器):一个容器可以放置多个组件，但每次只有一个组件可见(组件重叠)。

使用first()，last()，next()可以决定哪个组件可见。可以用于将一系列的面板有顺序地呈现给用户。

(5)、GridBag Layout(复杂的网格布局管理器):

在Grid中可指定一个组件占据多行多列，GridBag的设置非常烦琐。

注：添加滚动条：JScrollPane jsp = new JScrollPane(ll);

4、常用的组件：

(1)、JTextArea：用作多行文本域

(2)、JTextField：作单行文本

(3)、JButton:按钮

(4)、JComboBox：从下拉框中选择记录

(5)、JList：在界面上显示多条记录并可多重选择的列表

(6)、JMenuBar：菜单栏

(7)、JScrollPane：滚动条

```
/**********************************************************
******/
//最简单的图形用户界面，学会其中的四大步骤
import java.awt.*;
import javax.swing.*;
class FirstFrame{
public static void main(String[] args){
//1、选择容器
JFrame f = new JFrame();//在JFrame()的括号里可以填写窗口标题
//2、选择布局管理器
LayoutManager lm = new BorderLayout();
f.setLayout(lm);
```

```
    //3、添加组件
    JButton b = new JButton("确定");
    f.add(b);
    //4、添加事件，显示
    JOptionPane.showMessageDialog(null,"哈哈，你好！");//对话窗口
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//用窗口的开关控制程序的结束
    f.setSize(400, 300);//窗口的大小
    f.setVisible(true); //让窗口可见，默认不可见的
    }}

/*************************************************************/
    /*******************例题 画出计算器的界面*****************************
    界面如下：
    1 2 3 +
    4 5 6 -
    7 8 9 *
    0 . = /
    ********************/
    import java.awt.*;
    import javax.swing.*;
    class Calculator {
    public static void main(String[] args){
    JTextField text = new JTextField();
    JFrame f = new JFrame("计算器");
    Font font = new Font("宋体", Font.BOLD, 25);//"宋体"想写成默认，则写“null”
    text.setFont(font); //定义字体
    text.setHorizontalAlignment(JTextField.RIGHT);//令text的文字从右边起
    text.setEditable(false);//设置文本不可修改，默认可修改(true)
    f.add(text, BorderLayout.NORTH);//Frame和Dialog的默认布局管理器是Border Layout
    JPanel buttonPanel = new JPanel();//设法把计算器键盘放到这个Jpanel按钮上
    String op = "123+456-789*0.=/";
    GridLayout gridlayout = new GridLayout(4,4,10,10);
```

```
    buttonPanel.setLayout(gridlayout);//把计算器键盘放到buttonPanel按钮上
    for(int i=0; i<op.length(); i++){
    char c = op.charAt(i);
    JButton b = new JButton(c+"");
    buttonPanel.add(b);
    }//这个循环很值得学习
    f.add(buttonPanel/*, BorderLayout.CENTER*/); //默认添加到CENTER位置
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//关窗口时结束程序
    f.setSize(300, 250);
    f.setVisible(true);//这句要放到最后，等事件完成后再显示
    }}
    /******学过事件之后，可以实现计算器的具体功能
*******************************/
```

观察者模式：

事件源一旦产生事件，监察者立即作出相应处理。

事件源：产生一个事件的对象

事件对象：由事件源产生的一个对象

事件监听者(观察者)：处理这个事件对象的对象

事件注册：给一个事件源注册一个(或多个)事件监听者

事件模型(重点)

1. 定义： 事件模型指的是对象之间进行通信的设计模式。

事件模型是在观察者模式基础上发展来的。

2. 对象1 给对象2 发送一个信息相当于对象1 引用对象2 的方法。

3. 事件对象分为三种：

(1)事件源：发出事件者；

(2)事件对象：发出的事件本身(事件对象中会包含事件源对象)

事件对象继承:java.util.EventObjct类.

(3)事件监听器：提供处理事件指定的方法。

标记接口：没有任何方法的接口；如EventListene接口

监听器接口必须继承java.util.EventListener接口。

监听接口中每一个方法都会以相应的事件对象作为参数。

4. 授权：Java AWT 事件模型也称为授权事件模型，指事件源可以和监听器之间事先建立一种授权关系：

约定那些事件如何处理，由谁去进行处理。这种约定称为授权。

当事件条件满足时事件源会给事件监听器发送一个事件对象，由事件监

听器去处理。

事件源和事件监听器是完全弱偶合的。

一个事件源可以授权多个监听者(授权也称为监听者的注册);事件源也可以是多个事件的事件源。

监听器可以注册在多个事件源当中。监听者对于事件源的发出的事件作出响应。

在java.util 中有EventListener 接口：所有事件监听者都要实现这个接口。

java.util 中有EventObject 类：所有的事件都为其子类。

注意：接口因对不同的事件监听器对其处理可能不同，所以只能建立监听的功能，而无法实现处理。

```
    //监听器接口要定义监听器所具备的功能，定义方法
    /************下面程序建立监听功能
**************************/
    import java.awt.*;
    import javax.swing.*;
    public class TestEvent {
    public static void main(String[] args) {
    JFrame f = new JFrame("测试事件");
    JButton b = new JButton("点击");//事件源：鼠标点击
    JTextArea textArea = new JTextArea();
    textArea.setFont(new Font(null,
Font.BOLD+Font.ITALIC, 26));
    JScrollPane scrollPane = new JScrollPane(textArea);
    f.add(scrollPane, "Center");
    ButtonActionListener listener = new
ButtonActionListener(textArea);
    b.addActionListener(listener);
    f.add(b, BorderLayout.SOUTH);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setSize(400, 300);
    f.setLocation(250, 250);
    f.setVisible(true);
    }}
    //事件对象，可以直接调用系统写好的
    /*class ActionEvent extends EventObject{
    public ActionEvent(Object source) {
    super(source);
    }}*/
    //监听接口，同样可以调用系统写好的
    /*interface ActionListener extends EventListener{
    public void actionPerformed(ActionEvent event);
    }*/
```

```
    //监听者
    class ButtonActionListener implements ActionListener{
    private JTextArea textArea;
    public ButtonActionListener(JTextArea textArea) {
    this.textArea = textArea;
    }
    public void actionPerformed(ActionEvent e) {//必须覆盖它的actionPerformed()
    //JOptionPane.showMessageDialog(null, "按钮被按了一下");
    textArea.append("哈哈，放了几个字\n");
    textArea.append("哈哈，又放了几个字\n");
    }}

/*************************************************************/
```

注意查看参考书：事件的设置模式，如何实现授权模型。

事件模式的实现步骤：开发事件对象(事件发送者)—接口—接口实现类—设置监听对象

重点：学会处理对一个事件源有多个事件的监听器(在发送消息时监听器收到消息的排名不分先后)。

事件监听的响应顺序是不分先后的，不是谁先注册谁就先响应。

事件监听由两个部分组成(接口和接口的实现类)。

一定要理解透彻Gril.java 程序。

事件源 事件对象 事件监听者

gril EmotinEvent EmotionListener(接口)、Boy(接口的实现类)

鼠标事件：MouseEvent，接口：MouseListener。

注意在写程序的时候：import java.awt.*;以及import java.awt.event.*注意两者的不同。

在生成一个窗体的时候，点击窗体的右上角关闭按钮激发窗体事件的方法：

窗体Frame 为事件源，用WindowsListener 接口调用Windowsclosing()。

为了配合后面的实现，必须实现WindowsListener所有的方法；除了Windowsclosing方法，其余的方法均为空实现。这样的程序中实现了许多不必要的实现类，虽然是空实现。为了避免那些无用的实现，可以利用WindowEvent 的一个WindowEvent 类，还是利用windowsListener。WindowAdapter类,实现于WindowsListener。它给出的全是空实现，

那就可以只覆盖其中想实现的方法，不必再写空实现。

```
    /******练习：写一个带button 窗体，点关闭按钮退出。
*************/
    import java.awt.event.WindowAdapter;
    import java.awt.event.WindowEvent;
    import javax.swing.JFrame;
    public class TestAdapter {
    public static void main(String[] args) {
    JFrame f = new JFrame("测试适配器");
    MyWindowListener listener = new MyWindowListener();
    //f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    //现在有上面这一句话，就可以不必再创建一个类了
    f.addWindowListener(listener);
    f.setSize(400, 300);
    f.setVisible(true);
    }
    private static class MyWindowListener extends
WindowAdapter{
    public void windowClosing(WindowEvent e) {
    System.exit(0);
    }}}

/**********************************************************
****/
```

注意：监听过多，会抛tooManyListener 例外。

缺省试配设计模式：如果一个接口有太多的方法，我们可以为这个接口配上一个对应的抽象类。

《多线程》

一.线程:线程是一个并发执行的顺序流,一个进程包括多个顺序执行流程,这执行流程称为线程。

线程是一个操作系统创建并维护的一个资源,对操作系统来说JVM就是一个进程。

对于单个CPU系统来说，某一个时刻只可能由一个线程在运行。

一个Thread对象就表示一个线程。

线程由三部分组成:

(1).CPU分配给线程的时间片

(2).线程代码(写在run方法中)

(3).线程数据

进程是独立的数据空间,线程是共享的数据空间.

线程对象存在于虚拟机进程空间的一块连续的地址空间(静态)

//main()也是一个线程。

注意:

1.线程是动态的,与线程对象是两回事.

2.线程对象与其它对象不同的是线程对象能够到底层去申请管理一个线程资源。

3.只有对线程对象调用start()方法才是到底层去申请管理一个线程资源。

4.任务并发执行是一个宏观概念，微观上是串行的。

二.进程的调度

进程的调度由OS负责(有的系统为独占式(Windows),有的系统为共享式(Unix)，根据重要性，进程有优先级)

由OS 将时间分为若干个时间片。JAVA 在语言级支持多线程。分配时间的仍然是OS。

三.线程有两种实现方式:

第一种方式:

```
class MyThread extends Thread{
public void run(){
//……需要进行执行的代码，如循环。
}}
class TestThread{
public static void main(String[]args){
Thread t1=new MyThread();
t1.start();
}}
```

只有等到所有的线程全部结束之后，进程才退出。

第二种方式：通过接口实现继承

```
class MyThread implements Runnable{
public void run(){
//……需要进行执行的代码，如循环。
}}
class TestThread{
public static void main(String[]args){
Runnable target=new MyThread();
Thread t3=new Thread(target);
t3.start();//启动线程，这种方式跟前者一样，只是可以继承
}}
/*******************多线程事例
*********************************/
class TestThread{
public static void main(String[] args){
System.out.println("Main Thread Start!");
//Thread t = new MyThread();
Runnable r = new MyRunnable();
```

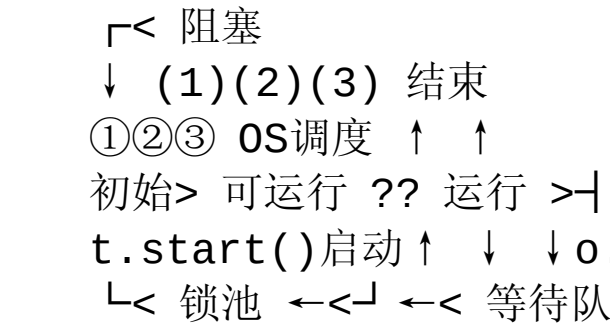
```
    Thread t1 = new Thread(r);//启动另一个带任务的线程
    Thread t = new MyThread(t1);//这个线程需要join(t1)，否则用上面没参的一句
    t.start();//启动一个线程
    t1.setPriority(10);//设置优先级
    t1.start();
    System.out.println("Main Thread End!");
    }}
    class MyThread extends Thread{
    private Thread t;
    public MyThread (){}
    public MyThread (Thread t){this.t = t;}
    public void run(){
    for(int i=0; i<100; i++){
    System.out.println(i+" $$$$");
    if(i==65){ //join()加入其它线程，等其运行完后再运行
    try{t.join();}
    catch(Exception e){e.printStackTrace();}
    }
    //当i=50时，放弃CPU占用，让其它程序或线程使用
    if(i==50){Thread.yield();}//没有sleep睡眠方法时，此句才可看出效果
    //阻塞，睡眠5毫秒，中途会被打断而抛异常
    try{Thread.sleep(5);} catch(Exception e){}
    }}}
    class MyRunnable implements Runnable{
    //Runnable是线程任务接口，让你可以继承其它类；Thread是类，不能继承
    public void run(){
    for(int i=0; i<100; i++){
    System.out.println(i+" ****");
    try{Thread.sleep(5);} catch(Exception e){}
    }}}

/*************************************************************
****/
```

四．线程中的7种非常重要的状态： 初始New、可运行Runnable、运行Running、阻塞Blocked、

锁池lock_pool、等待队列wait_pool、结束Dead

有的书上认为只有五种状态：将“锁池”和“等待队列”都看成是“阻塞”状态的特殊情况：这种认识也是正确的。

将“锁池”和“等待队列”单独分离出来有利于对程序的理解。



获得锁标志 synchronized(o)

注意：图中标记依次为

①输入完毕； ②wake up ③t1 退出

⑴等待输入(输入设备进行处理，而CUP 不处理)，则放入阻塞，直到输入完毕。

⑵线程休眠sleep()

⑶t1.join()将t1 加入运行队列，直到t1 退出，当前线程才继续。

特别注意：①②③与⑴⑵⑶是一一对应的。

进程的休眠：Thread.sleep(1000);//括号中以毫秒为单位

当线程运行完毕，即使在结束时时间片还没有用完，CPU 也放弃此时间片，继续运行其它程序。

T1.join 实际上是把并发的线程编成并行运行。

五．线程的优先级：

设置线程优先级：setPriority(Thread. MAX_PRIORITY);

setPriority(10); //设置优先级，独占式的操作系统按优先级分配CPU，共享式操作系统按等待时长分配

JAVA的优先级可以是 1~10,默认是5，数据越大，优先级越高。

//windows只有6个优先级，会自动转换来分配

为了跨平台，最好不要使用优先级决定线程的执行顺序。

//跨平台性的含义：除了程序能够正常运行，还必须保证运行的结果。

线程对象调用yield()时会马上交出执行权,交由一个高优先级的线程进入可运行状态；自己等待再次调用。

程序员需要关注的线程同步和互斥的问题。

多线程的并发一般不是程序员决定，而是由容器决定。

六．多线程出现故障的原因：

(1).多个线程同时访问一个数据资源(该资源称为临界资源)，形成数据发生不一致和不完整。

(2).数据的不一致往往是因为一个线程中的多个关联的操作(这几个操作合成原子操作)未全部完成。

避免以上的问题可采用对数据进行加锁的方法，如下

七．对象锁 Synchronized

防止打断原子操作，解决并发访问的故障。

//原子操作：不可分割的几个操作，要么一起不做，要么不能被干扰地完成。

1. 互斥锁标记:每个对象除了属性和方法，都有一个monitor(互斥锁标记)，

用来将这个对象交给一个线程，只有拿到monitor 的线程才能够访问这个对象。

2. Synchronized:这个修饰词可以用来修饰方法和代码块

Object obj; Obj.setValue(123);

Synchronized用来修饰代码块时，该代码块成为同步代码块。

Synchronized 用来修饰方法，表示当某个线程调用这个方法之后，其它的事件不能再调用这个方法。

只有拿到obj 标记的线程才能够执行代码块。

注意：

(1)Synchronized 一定使用在一个方法中。

(2)锁标记是对象的概念，加锁是对对象加锁，目的是在线程之间进行协调。

(3)当用Synchronized 修饰某个方法的时候，表示该方法都对当前对象加锁。

(4)给方法加Synchronized 和用Synchronized 修饰对象的效果是一致的。

(5)一个线程可以拿到多个锁标记，一个对象最多只能将monitor 给一个线程。

(6)构造方法和抽象方法不能加synchronized;

(7)一般方法和静态方法可以加synchronized同步。//静态方法把类看作对象时可加锁

(8)Synchronized 是以牺牲程序运行的效率为代价的，因此应该尽量控制互斥代码块的范围。

(9)方法的Synchronized 特性本身不会被继承，只能覆盖。

八. 锁池

1. 定义：线程因为未拿到锁标记而发生的阻塞不同于前面五个基本状态中的阻塞，称为锁池。

2. 每个对象都有自己的锁池的空间，用于放置等待运行的线程。这些线程中哪个线程拿到锁标记由系统决定。

3. 死锁：线程互相等待其它线程释放锁标记，而又不释放自己的；造成无休止地等待。

死锁的问题可通过线程间的通信解决。

九．线程间通信：

1. 线程间通信机制实际上也就是协调机制。

线程间通信使用的空间称之为对象的等待队列，这个队列也属于对象的空间。

注：现在，我们已经知道一个对象除了由属性和方法之外还有互斥锁标记、锁池空间和等待队列空间。

2. wait()

在运行状态中，线程调用wait()，表示这线程将释放自己所有的锁标记，同时进入这个对象的等待队列。

等待队列的状态也是阻塞状态，只不过线程释放自己的锁标记。

用notify()方法叫出之后，紧跟着刚才wait();的位置往下执行。

3. Notify()

如果一个线程调用对象的notify()，就是通知对象等待队列的一个线程出列。进入锁池。

如果使用notifyall()则通知等待队列中所有的线程出列。

注意：只能对加锁的资源(Synchronized方法里)进行wait()和notify()。

我们应该用notifyall取代notify，因为我们用notify释放出的一个线程是不确定的，由OS决定。

释放锁标记只有在Synchronized 代码结束或者调用wait()。

锁标记是自己不会自动释放，必须有通知。

注意：在程序中判定一个条件是否成立时要注意使用WHILE 要比使用IF 更严密。

//WHILE 循环会再次回来判断，避免造成越界异常。

4. 补充知识：

suspend()将运行状态推到阻塞状态(注意不释放锁标记)。恢复状态用resume()。Stop()释放全部。

这几个方法上都有Deprecated 标志，说明这个方法不推荐使用。

一般来说，主方法main()结束的时候线程结束，可是也可能出现需要中断线程的情况。

对于多线程一般每个线程都是一个循环，如果中断线程我们必须想办法使其退出。

如果想结束阻塞中的线程(如sleep或wait)，可以由其它线程对其对象调用interrupt()。

用于对阻塞(或锁池)会抛出例外Interrupted。

5. Exception。

这个例外会使线程中断并执行catch 中代码。

十. 5.0的新方法：

//参看 java.util.concurrent.*;包下的Callable,ExecutorService,Executors;

1. ExecutorService代替Thread，它不会销毁线程，效率更高，用空间换时间，适用于服务器。

ExecutorService exec = Executors.newFixedThreadPool(3);//创建3个等待调用的线程

Callable c1 = new Task();//用Callable的Task子类实现任务，其call()代替run()

Future f1 = exec.submit(c1);//Futrue获得运行线程后的结果，这线程与主程序分开

String s1 = f1.get();//Futrue的获得结果的方法，会返回异常

2. 用Callable接口代替Runnable

因为Runnable不能抛异常，且没有返回值。

3. Lock对象 替代Synchronized标志符，获得的更广泛的锁定操作，允许更灵活的结构。

还有tryLock()尝试加锁。

解锁用unLock()；可以主动控制解锁，方便复杂的方法调用。

//下列方法参考 java.util.concurrent.locks

在这里wait()用await()替代；Notify(),notifyall()用signal(),signalAll()替代。

ReadWriteLock读写锁：

writeLock()写锁，排他的，一旦加上，其它任何人都不能来读写。

readLock()读锁，共享的，加上之后，别人可以读而不能写。可以加多个读锁。

重点： 实现多线程的两种方式， Synchronized, 生产者和消费者问题

练习：① 存车位的停开车的次序输出问题。

② 写两个线程，一个线程打印 1~52，另一个线程打印字母A-Z。打印顺序为12A34B56C……5152Z。要求用线程间的通信。

注：分别给两个对象构造一个对象o，数字每打印两个或字母每打印一个就执行o.wait()。

在o.wait()之前不要忘了写o.notify()。

补充：通过Synchronized，可知Vector与ArrayList的区别就是Vector几乎所有的方法都有Synchronized。

所以Vector 更为安全，但效率非常低下。 同样：Hashtable 与

HashMap 比较也是如此。

```
    /*****************多线程间的锁及通信
*************************/
    //生产者—消费者问题，见P272
    public class TestProducerAndConsumer {
    public static void main(String[] args) {
    MyStack ms = new MyStack();
    Thread producer = new ProducerThread(ms);
    Thread consumer = new ConsumerThread(ms);
    producer.start();
    consumer.start();
    Thread producer2 = new ProducerThread(ms);
    producer2.start();//用第2个生产者，说明while循环判断比if
更严密
    }}
    class MyStack{ //工厂
    private char[] cs = new char[6];//仓库只能容6个产品
    private int index = 0;
    public synchronized void push(char c){
    //用while来循环判断，避免多个push进程时的下标越界异常。如果
用if，只能在一个push进程时能正常
    while(index==6){try{wait();}catch(Exception e){}}
    cs[index] = c;
    System.out.println(cs[index]+ " pushed!");
    index++;
    }
    public synchronized void pop(){
    while(index==0){try{wait();}catch(Exception e){}}
    index--;
    System.out.println(cs[index]+" poped!");
    notify();
    cs[index] = '\0';
    }
    public void print(){for(int i=0; i<index;
i++){System.out.print(cs[i]+"\t");}
    System.out.println();
    }}
    class ProducerThread extends Thread{ //生产者
    private MyStack s;
    public ProducerThread(MyStack s){this.s = s;}
    public void run(){
    for(char c='A'; c<='Z'; c++){
    s.push(c);
    try{Thread.sleep(20);}
    catch(Exception e){e.printStackTrace();}
```

```
    s.print();
    }}}
    class ConsumerThread extends Thread{ //消费者
    private MyStack s;
    public ConsumerThread(MyStack s){
    this.s = s;
    }
    public void run(){
    for(int i=0; i<26*2; i++){
    s.pop();
    try{Thread.sleep(40);}
    catch(Exception e){e.printStackTrace();}
    s.print();
    }}}

/**********************************************************
****/
```

Daemon Threads(daemon 线程)

是服务线程，当其它线程全部结束，只剩下daemon线程时，虚拟机会立即退出。

```
Thread t = new DaemonThread();
t.setDaemon(true);//setDaemon(true)把线程标志为
```

daemon，其它的都跟一般线程一样

```
t.start();//一定要先setDaemon(true)，再启动线程
```

在daemon线程内启动的线程，都定为daemon线程

```
Thread.currentThread() //取得执行中当前线程的信息。
```

Thread t; t.isAlive(); //boolean isAlive() 判断线程是否存活。如果线程正在执行则true。

I/O流

一. I/O 流(java 如何实现与外界数据的交流)

1. Input/Output：指跨越出了JVM 的边界，与外界数据的源头或者目标数据源进行数据交换。

注意：输入/输出是针对JVM 而言。

2. 流的分类：

按流向分为输入流和输出流；

按传输单位分为字节流和字符流；

按功能还可以分为节点流和过滤流。(以Stream结尾的类都是字节流。)

节点流：负责数据源和程序之间建立连接；(相当于电线中的铜线，过滤流相当于电线的塑料皮)

过滤流：用于给节点增加功能。(相当于功能零部件)

过滤流的构造方式是以其它流位参数构造，没有空参构造方法(这样的设计模式称为装饰模式)。

注：I/O流使用完后建议调用close()方法关闭流并释放资源。

在关闭流时只需关闭最外层的流，会自动关闭内层的流。

3. File 类(java.io.*)可表示一个文件，也有可能是一个目录

在JAVA 中文件和目录都属于这个类中，而且区分不是非常的明显。

4. Java.io 下的方法是对磁盘上的文件进行磁盘操作，但是无法读取文件的内容。

注意：创建一个文件对象和创建一个文件在JAVA中是两个不同的概念。前者是在虚拟机中创建了一个文件，但却并没有将它真正地创建到OS 的文件系统中，随着虚拟机的关闭，这个创建的对象也就消失了。而创建一个文件是指在系统中真正地建立一个文件。

例如：File f=new File(“11.txt”);//创建一个名为11.txt 的文件对象

f.CreateNewFile(); //这才真正地创建文件

f.CreateMkdir();//创建目录

f.delete();//删除文件

getAbsolutePath();//打印文件绝对路径

getPath();//打印文件相对路径

f.deleteOnExit();//在进程退出的时候删除文件，这样的操作通常用在临时文件的删除。

5. 对于跨平台：File f2=new file("d:\\abc\\789\\1.txt")

//这文件路径是windows的，"\"有转义功能，所以要两个

这个命令不具备跨平台性，因为不同的OS的文件系统很不相同。

如果想要跨平台，在file 类下有separtor()，返回锁出平台的文件分隔符。

```
File.fdir=new File(File.separator);
String str=”abc”+File.separator+”789”;
```

6. List()：显示文件的名(相对路径)

ListFiles()：返回Files 类型数组，可以用getName()来访问到文件名。

使用isDirectory()和isFile()来判断究竟是文件还是目录。

使用I/O流访问file中的内容。

JVM与外界通过数据通道进行数据交换。

二. 字节流

1. 字节输入流：io包中的InputStream为所有字节输入流的父类。

Int read();//读入一个字节(每次一个)；

可先使用new byte[]=数组，调用read(byte[] b)

read (byte[])返回值可以表示有效数；read (byte[])返回值为-1 表示结束。

2. 字节输出流：io包中的OutputStream为所有字节输入流的父类。

Write和输入流中的read相对应。

3. 在流中close()方法由程序员控制。因为输入输出流已经超越了JVM的边界，所以有时可能无法回收资源。

原则：凡是跨出虚拟机边界的资源都要求程序员自己关闭，不要指望垃圾回收。

4. 以Stream结尾的类都是字节流。

```
FileOutputStream f = new FileOutputStream("1.txt");//如果之前有这文件，将会覆盖
FileOutputStream f = new FileOutputStream("1.txt",true);//如果有这文件，只会追加
```

DataOutputStream,DataInputStream:可以对八种基本类型加上String类型进行写入。

因为每种数据类型的不同，所以可能会输出错误。

所有对于：DataOutputStream DataInputStream两者的输入顺序必须一致。

```
/**输入、输出流；字节流**************************/
public static void main(String[]args) throws IOException{
FileOutputStream fos=null;//输出流，字节流
fos = new FileOutputStream("a.txt");//会抛异常；如果不用绝对路径，将生成所在类的路径中
for(int i = 32; i<=126 ;i++) fos.write(i);//直接写进
// String s = "ABCDE";byte[] ss = s.getBytes();fos.write(ss,0,4);//直观地写进
fos.close();
FileInputStream fis = new FileInputStream("a.txt");//输入流
//for(int i=0;(i=fis.read())!=-1;) System.out.print((char) i);
//上句是直接读取；fis.read()类似next().
byte[] b= new byte[1024];//在内存划分空间以便装载从文件读来的数据
fis.read(b);fis.close();
for(int i=0;i<b.length;i++)
```

```
System.out.print((char)b[i]);
    }//打印出键盘上的所有数字、大小写字母和标准符号
    /*****************************************/
```

5. java.io.inputStream 的主要方法：

int available() throws IOException 取得可从输入流读取的byte数。

void close() throws IOException 关闭输入流，释放输入流相关的所有系统资源。

public void mark(int readlimit) 在输入流当前位置加上标记。

boolean markSupported() 判断输入流是否支持 mark 和 reset 方法。

int read() throws IOException 从输入流读入下一个 byte 数据。

int read(byte[] b) throws IOException 将输入流读入的byte数存入缓冲区数组b。

int read(byte[] b, int off, int len) throws IOException 从输入流off的位置起，读入长度为len的数据存入缓冲区数组b。

void reset() throws IOException 将串流的位置重新移回先前呼叫mark方法时位置。

long skip(long n) throws IOException 在输入流中，跳过n个byte。

6.java.io.OutputStream 的主要方法：

void close() throws IOException 关闭输出流，释放输出流相关的所有系统资源。

void flush() throws IOException 将输出流缓冲区中所有资料强制输入。

void write(byte[] b) throws IOException 从指定的位数组中将b.length位的资料写入到输出串流。

void write(byte[] b, int off, int len) throws IOException 从输出流的off位置起的缓冲区数组b中，输出len位至输出流。

void write(int b) throws IOException 将指定的 byte 输出到输出流。

7.文件输出入时常用的类：

java.io.File 取得文件的资料。

java.io.FileInputStream 可以用byte为单位，读取二进制文件的值。

java.io.FileOutputStream 可以用byte为单位，输出二进制文

件。

java.io.FileRead 可以用byte为基础，读取文件的值。

java.io.FileWriter 可以用byte为基础，输出文件。

过滤流：(装饰模式，油漆工模式，修饰字节流)

bufferedOutputStream

bufferedInputStream

在JVM的内部建立一个缓冲区，数据先写入缓冲区，直到缓冲区写满再一次性写出，效率提高很多。

使用带缓冲区的输入输出流(节点流)会大幅提高速度，缓冲区越大，效率越高。(典型的牺牲空间换时间)

切记：使用带缓冲区的流，如果数据数据输入完毕，使用flush方法将缓冲区中的内容一次性写入到外部数据源。

用close()也可以达到相同的效果，因为每次close前都会使用flush。一定要注意关闭外部的过滤流。

管道流(非重点)：也是一种节点流，用于给两个线程交换数据。

PipedOutputStream

PipedInputStream

输出流： connect(输入流)

RondomAccessFile 类允许随机访问文件同时拥有读和写的功能。

GetFilepoint()可以知道文件中的指针位置，使用seek()定位。

Mode(“r”:随机读；”w”：随机写；”rw”：随机读写)

字符流： reader\write 只能输纯文本文件。

FileReader 类：字符文件的输出

三、字节流的字符编码：

字符编码把字符转换成数字存储到计算机中，按ASCii 将字母映像为整数。

把数字从计算机转换成相应的字符的过程称为解码。

乱码的根源是编码方式不统一。任何一种编码方式中都会向上兼容ASCII码。所以英文没有乱码。

编码方式的分类：

ASCII(数字、英文):1个字符占一个字节(所有的编码集都兼容ASCII)

ISO8859-1(欧洲，拉丁语派)：1个字符占一个字节

GB-2312/GBK：1 个字符占两个字节。GB代表国家标准。

GBK是在GB－2312上增加的一类新的编码方式，也是现在最常用的汉字编码方式。

Unicode: 1 个字符占两个字节(网络传输速度慢)

UTF-8：变长字节，对于英文一个字节，对于汉字两个或三个字节。

原则：保证编解码方式的统一，才能不至于出现错误。

I/O学习种常犯的两个错误：1. 忘了flush 2. 没有加换行。

四、字符流

以reader或write结尾的流为字符流。 Reader和Write是所有字符流的父类。

Io 包的InputStreamReader 称为从字节流到字符流的桥转换类。这个类可以设定字符解码方式。

OutputStreamred:字符到字节

Bufferread 有readline()使得字符输入更加方便。

在I/O 流中，所有输入方法都是阻塞方法。

最常用的读入流是BufferedReader.没有PrintReader。

Bufferwrite给输出字符加缓冲，因为它的方法很少，所以使用父类PrintWriter，它可以使用字节流对象构造，省了桥转换这一步，而且方法很多。注：他是带缓冲的。最常用的输出流。

对象的持久化：把对象保存文件，数据库

对象的序列化：把对象放到流中进行传输

对象的持久化经常需要通过序列化来实现。

四大主流：InputStream,OutputStream,Read,Write

awt:

java.applet.Applet 的 getParameter方法，可以取得HTML里的参数。

HTML文件必须用

标签，如：

```
(Graphics g;)
g.drawString(getParameter("htmlpram"),70,80);
//drawString 是显示文字的方法。
```

IO

java.io.File 类

boolean exists() 检查文件是否存在。

```
File file1 = new File("sample.txt");
if(file1.exists()){……}
```

String getName() 取得文件名。 //file1.getName()返回如：xxx.txt

String getParent() 取得所在目录。 //file1.getParent()返回如：C:\JAVA\265

String getPath() 取得完整路径。 //file1.getPath()返回如：C:\JAVA\265\xxx.txt

String[] list() 返回指定目录下的所有文件名、目录名。

```
//file1.list()
```

long length() 取得指定对象的长度。若对象不存在，不会抛异常，而会返回0L。 //file1.length()

boolean canRead() 检查指定文件是否可读。如果文件存在且可读，返回 true。 //file1.canRead()

boolean canWrite() 检查指定文件是否可写入。如果文件存在且可写入，返回 true。 //file1.canWrite()

long lastModified() 返回最后修改的时刻。当文件不存在，或发生输出入错误时，会返回0L，即1970年1月1日……。需要转成时间格式。

```
import java.io.*; import java.util.*; import java.text.*;
File file1 = new File("sample.txt");
Date date1 = new Date(file1.lastModified());
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.LONG,DateFormat.LONG);
System.out.println(df.format(date1));
```

boolean renameTo(File dest) 给文件改名。改名成功则返回true，否则返回false。 //file1.renameTo(file2)，将files1改名成file2

boolean delete() 删除文件或目录。删除目录时，目录必须是空的。删除成功则true。 //file1.delete()

boolean createNewFile() throws IOException 建立指定的文件。当指定文件不存在而成功建立则true，文件存在则false。 //file1.createNewFile()

boolean mkdir() 建立指定目录。成功建立则true。 //file1.mkdir()

void deleteOnExit() 在程序结束时，自动删除文件或目录。一旦要求后，就无法在程序中取消。//file1.deleteOnExit()

一．对象序列化

1．定义：把一个对象通过I/O流写到文件(持久性介质)上的过程叫做对象的序列化。

2．序列化接口：Serializable

此接口没有任何的方法，这样的接口称为标记接口。

3．不是所有对象都能序列化的，只有实现了Serializable的类，他的实例对象才是可序列化的。

4．Java种定义了一套序列化规范，对象的编码和解码方式都是已经定义好的。

5．class ObjectOutputStream和ObjectInputStream也是带缓冲的过滤流，使节点流直接获得输出对象

可以用来向文件中写入八种基本数据类型和对象类型。

最有用的方法：

(1)writeObject(Object b)

(2)readObject()返回读到的一个对象，但是需要我们注意的是,该方法不会以返回null表示读到文件末尾。

而是当读到文件末尾时会抛出一个IOException;

6. 序列化一个对象并不一定会序列化该对象的父类对象

7. 瞬间属性(临时属性)不参与序列化过程。

8. 所有属性必须都是可序列化的，特别是当有些属性本身也是对象的时候，要尤其注意这一点。

序列化的集合就要求集合中的每一个元素都是可序列化的。

9. 用两次序列化把两个对象写到文件中(以追加的方式)，与用一次序列化把两个对象写进文件的大小是不同的。

因为每次追加时都会在文件中加入开始标记和结束标记。所以对象的序列化不能以追加的方式写到文件中。

二、transient关键字

1. 用transient修饰的属性为临时属性。

三. 分析字符串工具java.util. StringTokenizer;

1.string tokenizer 类允许应用程序将字符串分解为标记

2.可以在创建时指定，也可以根据每个标记来指定分隔符(分隔标记的字符)集合。

3.StringTokenizer(s,":")用“：”隔开字符，s 为String对象。

```
/**********************************************************
****/
    import java.util.StringTokenizer;
    public class TestStringTokenizer {
    public static void main(String[] args) {
    String s = "Hello:Tarena:Chenzq";
    StringTokenizer st = new StringTokenizer(s,":");
    while(st.hasMoreTokens()){
    String str = st.nextToken();
    System.out.println(str);
    }}}

/**********************************************************
***/
```

四、网络的基础知识:

1、ip：主机在网络中的唯一标识，是一个逻辑地址。

127. 0. 0. 1 表示本机地址。(没有网卡该地址仍然可以用)

2、端口：端口是一个软件抽象的概念。如果把Ip地址看作是一个电话号码的话，端口就相当于分机号。

进程一定要和一个端口建立绑定监听关系。端口号占两个字节。

3、协议：通讯双方为了完成预先制定好的功能而达成的约定。

4、TCP/IP网络七层模型：

物理层Physical(硬件)、 数据链路层DataLink(二进制)、网络层Network(IP协议:寻址和路由)

传输层Transport(TCP协议，UDP协议) 、会话层Session(端口)

表示层Presentation、应用层Application(HTTP,FTP,TELET,SMTP,POPS,DNS)

注：层与层之间是单向依赖关系。对等层之间会有一条虚连接。Java中的网络编程就是针对传输层编程

5、网络通信的本质是进程间通信。

6、Tcp协议和UDP协议

TCP：开销大，用于可靠性要求高的场合。TCP的过程相当于打电话的过程。面向连接，可靠，低效

UDP：用在对实时性要求比较高的场合。UDP的过程相当于写信的过程。无连接，不可靠，效率高

五、网络套节字Socket(TCP)

1、一个Socket相当于一个电话机。

OutputStream相当于话筒

InputStream相当于听筒

2、服务器端要创建的对象：java。Net。ServerSocket

3、创建一个TCP服务器端程序的步骤：

1). 创建一个ServerSocket

2). 从ServerSocket接受客户连接请求

3). 创建一个服务线程处理新的连接

4). 在服务线程中，从socket中获得I/O流

5). 对I/O流进行读写操作，完成与客户的交互

6). 关闭I/O流

7). 关闭Socket

```
/**************************************************
******/
    import java.net.*;
    import java.io.*;
    public class TcpServer{//服务器端
    public static void main(String[] args) {
```

```
    ServerSocket ss = null;
    Socket s = null;
    try{ ss= new ServerSocket(10222);
    s = ss.accept();//客户端连上后返回Socket，监听端口
    OutputStream os = s.getOutputStream();
    PrintWriter pw = new PrintWriter(os);
    pw.println("欢迎欢迎！");//要换行，否则不能读取
    pw.flush();//从内存输出去
    }catch(Exception e){}
    finally{if(s!=null )try{s.close(); }catch(Exception
e){}
    if(ss!=null)try{ss.close();}catch(Exception e){}
    }}}
    public class TcpClient {//接受端
    public static void main(String[] args) throws Exception
{
    Socket s = new Socket("10.3.1.79", 10222);
    BufferedReader br = new BufferedReader(new
InputStreamReader
    (s.getInputStream()));
    System.out.println(br.readLine());
    s.close();
    }}

/**********************************************************
******/
```

4、建立TCP客户端

创建一个TCP客户端程序的步骤：

1). 创建Socket

2). 获得I/O流

3). 对I/O流进行读写操作

4). 关闭I/O流

5). 关闭Socket

5、网络套节字Socket(UDP)

1. UDP编程必须先由客户端发出信息。

2. 一个客户端就是一封信，Socket相当于美国式邮筒(信件的收发都在一个邮筒中)。

3. 端口与协议相关，所以TCP的3000端口与UDP的3000端口不是同一个端口

6、URL：统一资源定位器

唯一的定位一个网络上的资源

如:http://www.tarena.com.cn:8088

```
    /***下载程序
**********************************************************
********/
    import java.net.*;
    import java.io.*;
    class TestUrl{
    public static void main(String[] args) throws
Exception{
    String str =
"http://192.168.0.23:8080/project_document.zip";
    URL url = new URL(str);//上句指定下载的地址和文件
    URLConnection urlConn = url.openConnection();
    urlConn.connect();
    InputStream is = urlConn.getInputStream();
    FileOutputStream fos = new
FileOutputStream("/home/sd0807/down.zip");
    byte[] buf = new byte[4096]; //上句指定下载的地址和下载
后的名称
    int length = 0;
    while((length=is.read(buf))!=-1){
    fos.write(buf, 0, length);
    }
    fos.close();
    is.close();
    }}

/**********************************************************
**********************/
    网路:
    java.net.InetAddress类
    InetAddress getLocalHost() throws
UnknownHostException 取得本地端的IP
    InetAddress myaddress = InetAddress.getLocalHost();
System.out.println(myaddress);
    InetAddress getByName(String host) throws
UnknownHostException 籍由DNS以Host名称取得主机的IP地址。
    InetAddress sa =
InetAddress.getByName("www.moug.net");
System.out.println(myaddress); // 打印21.188.250.59
    String getHostName() 取得IP地址对应的名称
    InetAddress[] getAllByName(String host) throws
UnknownHostException 取得该host对应的所有IP地址列表。
    InetAddress sa[] =
InetAddress.getAllByName("www.microsoft.com");
```

for(int i=0; i
用法：javac <选项> <源文件>
其中，可能的选项包括：
-g 生成所有调试信息
-g:none 不生成任何调试信息
-g:{lines,vars,source} 只生成某些调试信息
-nowarn 不生成任何警告
-verbose 输出有关编译器正在执行的操作的消息
-deprecation 输出使用已过时的 API 的源位置
-classpath <路径> 指定查找用户类文件和注释处理程序的位置
-cp <路径> 同上(是 classpath 的缩写)
-sourcepath <路径> 指定查找输入源文件的位置
-bootclasspath <路径> 覆盖引导类文件的位置
-extdirs <目录> 覆盖安装的扩展目录的位置
-endorseddirs <目录> 覆盖签名的标准路径的位置
-proc:{none,only} 控制是否执行注释处理和/或编译。
-processor <class1>[,<class2>,<class3>……]
要运行的注释处理程序的名称；绕过默认的搜索进程
-processorpath <路径> 指定查找注释处理程序的位置
-d <目录> 指定存放生成的类文件的位置，默认与程序同目录
-s <目录> 指定存放生成的源文件的位置
-implicit:{none,class} 指定是否为隐式引用文件生成类文件
-encoding <编码> 指定源文件使用的字符编码，默认用平台预设的转换码
-source <版本> 提供与指定版本的源兼容性
-target <版本> 生成特定 VM 版本的类文件
-version 版本信息
-help 输出标准选项的提要
-Akey[=value] 传递给注释处理程序的选项
-X 输出非标准选项的提要
-J<标志> 直接将 <标志> 传递给运行时系统
-nowarn 设置不警告编译错误
输入[sd0807@localhost ~]$ java
运行(虚拟机)： java ***
Usage: java [-options] class [args……]
(to execute a class)
or java [-options] -jar jarfile [args……]
(to execute a jar file)

```
    where options include:
    -d32 use a 32-bit data model if available
    -d64 use a 64-bit data model if available
    -client to select the "client" VM
    -server to select the "server" VM
    -hotspot is a synonym for the "client" VM [deprecated]
The default VM is client.
    -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar
files>
    A : separated list of directories, JAR archives,
    and ZIP archives to search for class files.
    -D<name>=<value> set a system property
    -verbose[:class|gc|jni] enable verbose output要求编译
器显示正在编译的文件和正在加载的类别等。
    -version print product version and exit
    -version:<value> require the specified version to run
    -showversion print product version and continue
    -jre-restrict-search | -jre-no-restrict-search
    include/exclude user private JREs in the version search
    -? -help print this help message
    -X print help on non-standard options
    -ea[:
    ……|:<classname>]
    -enableassertions[:
    ……|:<classname>] enable assertions
    -da[:
    ……|:<classname>]
    -disableassertions[:
    ……|:<classname>] disable assertions
    -esa | -enablesystemassertions enable system
assertions
    -dsa | -disablesystemassertions disable system
assertions
    -agentlib:
    [=]
    load native agent library
    , e.g. -agentlib:hprof
    see also, -agentlib:jdwp=help and -agentlib:hprof=help
    -agentpath:
    [=] load native agent library by full pathname
    -javaagent:<jarpath>[=]
    load Java programming language agent, see
java.lang.instrument
    -splash: show splash screen with specified image
```

study happy,made by micky