***JAVA在线笔记***

***——比兴哥***

**在Eclipse中格式化代码 ：Shift+Tab 导包：Ctrl+1**

***标识符：***

**类名和接口的规范：**

首字母要大写，如果由多个单词构成每个单词的首字符都大写。

例：XxxYyy

如：PersonService

**变量名和方法名：**

我们遵守单词首字符不要大写，如果由单词构成，从第二个单词首字符都大写。 这叫驼峰模式。

例：xxxYyyZzz

如：personWangXing

**包名：** 所有字符都小写

例：xxx.yyy.zzz

**常量：** 所有字符都大写，多个单词之间使用下划线分隔

例：xxx\_yyy\_zzz

***关键词：*这些关键词都是不能用来自定义的**



位运算符合移位运算

按位与 & : 两位全为1,结果才为1(按补码做与运算再转换成原码,最后得到有符号数)

按位或 | : 两位全为0,结果才为0(按补码做或运算再转换成原码,最后得到有符号数)

按位异或 ^ : 两位不一样,结果才为1(按补码做异或运算再转换成原码,最后得到有符号数)

按位取反 ~ :0 -> 1 , 1 -> 0 (按补码取反再转换成原码,最后得到有符号数)

~2 = -3 2&3 = 2 2|3 = 3 ~-5 = 4 -3^3 = -2

>> 算术右移 : 低位溢出,符号为不变,并用符号为补溢出的高位 1>>2 = 0

<< 算术左移 : 符号为不变,低位补0

>>>逻辑右移 : 低位溢出,高位补0

***数据类型之间转换：***

byte b =127;

short s =50;

int i=100;

long l=10000l;

//常量5默认是int类型，b+5可能超出byte的最大值，所以jvm定义byte和short和char变量参与运算的时候数据类型会提升到int

int aa=b+5;

//short bb =s+1; 错误

int bb =s+1;

//定义一个a字符，每一个字符对应以个ASCII码，在字符参与运算时是用ASCII码参与运算

char c =’a’;

//char变量参与运算的时候数据类型会提升到int

int cc =c+3;

//byte和short和char之间运算也会转换成int

int ss =b+s;

**常量的计算**可以使用byte，short，char接受（常量的计算结果不能溢出）

byte ii=4+3; //正确

int ii=4+3; //正确

int i1=2147483647;

//如果int计算时发生溢出就会变为负值，不会自动转换成long，

int i2=i1+10;

***break：***可以省略不会报错，如果省略就会穿透执行语句（不管是否匹配上），知道遇到一个break才会跳出，所以不建议省略break。

在jdk1.7之前，switch的表达式只能是byte，short，char，int，但是1.7之后可以使用string。

在switch中，default是否则的意思。

在循环中， break是终止循环，彻底跳出循环；

continue是中断本次循环，重新下次循环；

***输入字符型数据：***

import java.util.Scanner;

public class Java001 {

public static void main(String[] args) {

Scanner sc=new Scanner(System.in);

System.out.println("请输入：");

char c=sc.next().charAt(0);

}

}

***输入数值型数据：***

import java.util.Scanner;

public class Java002 {

public static void main(String[] args) {

Scanner sc=new Scanner(System.in);

System.out.println("请输入：");

int c=sc.nextInt();

}

}

## ***使用 concat() 方法：***

在Java中，String类的concat()方法实现了将一个字符串连接到另一个字符串的后面。

格式： 字符串1**.**concat(字符串2);

例子：

public static void main(String[] args){

String info="三国演义、";/\*info此时值为：三国演义、\*/

info=info.concat("西游记、");/\*info此时值为：三国演义、西游记、\*/

info=info.concat("水漭传、");/\*info此时值为：三国演义、西游记、水漭传、\*/

info=info.concat("红楼梦");/\*info此时值为：三国演义、西游记、水漭传、红楼梦\*/

System.out.println(info); //info变量后接的所有字符都显示出来！！！

String cn="中国";

System.out.println(cn.concat("北京").concat("海淀区").concat("人民公园"))

显示： 三国演义、西游记、水浒传、红楼梦

中国北京海淀区人民公园

***Java获取字符串长度（length()）***

格式： 字符串.length();

例子：import java.util.Scanner;

public class Test05{

public static void main(String[] args) {

String sys="学生信息管理"; //字义一个字符串表示系统名称

System.out.println("欢迎进入《"+sys+"》系统"); //输出系统名称

System.out.println("请设置一个管理员密码：");

Scanner input=new Scanner(System.in);

String pass=input.next(); //获取用户输入的密码

int length=pass.length(); //获取密码的长度

if(length>6&&length<12){

System.out.println("密码长度符合规定。");

System.out.println("已生效，请牢记密码："+pass);

}else if(length>=12) System.out.println("密码过长。");

else System.out.println("密码过短。");

}

}

显示：

运行程序，当输入的密码过短时，运行结果如下所示：

欢迎进入《学生信息管理》系统

请设置一个管理员密码：

123456

密码过短。

当输入的密码符合规定时，运行结果如下所示：

欢迎进入《学生信息管理》系统

请设置一个管理员密码：

abc12345678

密码长度符合规定。

已生效，请牢记密码：abc12345678

***将字符串都转换成大写（或者都转换成小写）***

**public** **class** Java01 {

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

String str = "anzhiyuan HENHAO 哟";

System.***out***.println(str.toLowerCase()); /\*转换成小写\*/

System.***out***.println(str.toUpperCase()); /\*转换成大写\*/

}

}

***显示：***

anzhiyuan henhao 哟

ANZHIYUAN HENHAO 哟

Java去除空格（trim()）【只去除字符串前后的空格】

**格式：**字符串名.trim()

String str = “ an hui zhi ye ji shu xue yuan “;

System.out.println(str.length()); /\*输出31\*/

Systrm.out.prnitln(str.trim().length()); /\*输出29\*/

注意：str.trim().length() 先执行str.trim(),再执行后面的length()

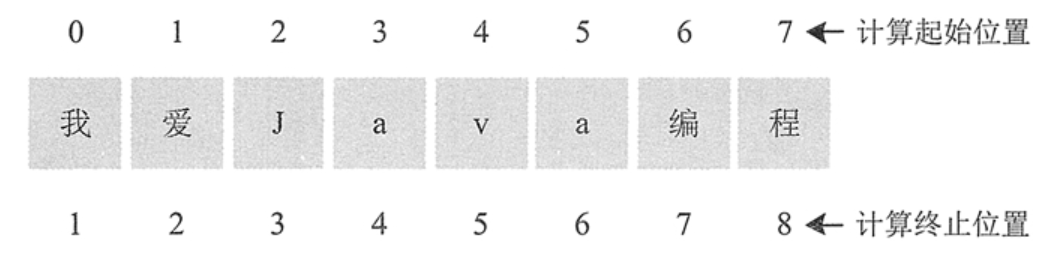
Java提取子字符串(substring())【提取单位是字符而不是字节】

**格式：**substring(int beginIndex)【此方式用于提取从索引位置至结尾的字符串部分。调用时，括号是需要提取字符串的开始位置，方法的返回值是提取的字符串。】

String str = “我爱 Java 编程”;

System.out.println(str.substring(3)); //输出：Java编程

**格式：**substring(int beginIndex,int endIndex)



String str = “我爱 Java 编程”;

System.out.println(str.substring(3,7)); //输出：Java

字符串的替换

**格式：**字符串变量名. replease(“要替换的字符串”,”想替换的字符串”)

**例子：**

String str = ”an zhi yuan”;

System.out.print(str.replease(“yuan”,”xueyuan”));

**格式：**字符串变量名. replease(“要替换的字符串”,”想替换的字符串”)

**例子：**

String str = ”an zhi yuan”;

System.out.print(str.replease(“yuan”,”xueyuan”));

/\*replaceFirst : 将想要替换的字符只替换第一个\*/

String str4 = "hello java , hello c语言 , hello python …";

String firststr4 = str4.replaceFirst("hello" , "你好");

System.***out***.println("\n" + firststr4);

**显示：**你好 java , hello c语言 , hello python

/\*rplaceAll : 将想要替换的字符全替换掉\*/

String allstr4 = str4.replaceAll("hello" , "你好");

System.***out***.println(allstr4);

**显示：**你好 java , 你好 c语言 , 你好 python

/\*equals : 两个字符串比较，完全相同输出true，否则输出false\*/

/\*equalsIgnoreCase : 两个字符串比较(不区分大小写)，完全相同输出true，否则输出false\*/

/\*compareTo : 按字典顺序比较两字符串大小（该比较是基于字符串各个字符的Unicode值），结果为数值型，完全相同输出 0 ，否则输出 ASCII码的差值 \*/

String str5="abc";String str6=**new** String("abc");String str7="ABC";

System.***out***.println(str5.equals(str6)); //输出 true

System.***out***.println(str5.equals(str7)); //输出 false

System.***out***.println(str5.equalsIgnoreCase(str7)); //输出 true

查找字符（串）

/\*indexOf : 用于返回字符（串）在指定的字符中首次出现的索引位置，如果能找到，返回索引位置的值，否则返回 - 1\*/

例子： String st = "Hello Java";

**int** size1 = st.indexOf('H'); //size的结果为0

**int** size2 = st.indexOf('v'); //size的结果为8 （第8个）

**int size3 =** st.indexOf("Ja" , 2)); //size的结果为6

**int size4 =** st.indexOf("Ja" , 7)); //size的结果为-1

/\*lastindexOf : 查找策略是从右往左查找，如果不能指定起始索引位置，则从字符串的末尾开始查找\*/

/\*charAt : 在字符串内根据指定的索引查找字符\*/

String words="today,monday,sunday";

System.out.println(words.charAt(0)); //结果：t

System.out.println(words.charAt(8)); //结果：n

替换字符（串）

//replace("旧","新"):新替换旧，替换字符串中所有的

//replaceAll("旧","新"):新替换旧，替换字符串中所有的；replaceAll("[i s]","or"):分别将“i”,“ ”,“s”替换成"or"；replaceAll("[时 使]","是"):只分别将“时”,“使”替换成"是",中间的空格讲不会被替换

//replaceFirst("旧","新"):新替换旧，替换字符串首次出现的；

//replaceFirst("[is]","or"):只替换首次出现的“i”，则不替换首次出现的“is”

String s1 = "this is my original string,it is very good!";

String s2 = s1.replace("is", "or"); System.***out***.println("s2="+s2);

String s3 = s1.replaceFirst("[is]", "or"); System.***out***.println("s3="+s3);

String s4 = s1.replaceAll("[h i]", "or"); System.***out***.println("s4="+s4);

String s5="今天时星其天，外面时下雨天。妈 米 去 买 菜 了，漏网在家写作业。"+"语文作业时”其”写 5 行，数学使第 10 页。";

String s6=s5.replaceAll("[时使]","是"); System.***out***.println("s6="+s6);

Math 类常用方法大全

E 常量的值：2.718281828459045

PI 常量的值：3.141592653589793

Math.sqrt() —> 计算平方根

Math.cbrt() —> 计算立方根

Math.hypot(x,y) —> 计算(x的平方+y的平方)的平方根

Math.pow(a,b) —> 计算a的b次方

Math.exp(x) —> 计算e^x的值

Math.max(多个参数) —> 计算最大值

Math.min(多个参数) —> 计算最小值

Math.abs() —> 求绝对值

Math.ceil() —> 返回大的值 Math.ceil(-10.1) //10.0 Math.ceil(10.7) //11.0

Math.floor() —> 返回小的值 Math.ceil(-10.1) //-11.0 Math.ceil(10.7) //10.0

Math.round() —> 四舍五入（float时返回int值，double时返回long）

Math.nextUp(a) —> 返回比a大一点点的浮点数 Math.nextUp(1.2));//1.2000000000000002

Math.nextDown(a) —> 返回比a小一点点的浮点数 Math.nextDown(1.2)); //1.1999999999999997

Math.nextAfter(a,b) —> 返回(a,b)或(b,a)间与a 相邻的浮点数 b可以比a小Math.nextAfter(1.2,2.7)); //1.2000000000000002

Math.nextAfter(1.2, -1)); //1.1999999999999997

StringBuffer类

StringBuffer类与String类区别：

StringBuffer默认分配16字节长度的缓冲区，当字符串超过该大小时，会自动增加缓冲区长度，而不是生成新的对象，

StringBuffer不像String，只能通过new来创建对象，布置成简写形式

格式： StringBuffer str = new StringBuffer（x）；

str ：为变量名.

x ：若是数值，则是分配缓冲区的长度，默认为16字节；也可以是字符串[“x”]

Capacity()方法返回字符串的容量大小

StringBuffer str = new StringBuffer();

System.out.print(str.capacity()); //输出str字符串的容量长度 16

**追加字符：append()方法用于向原有StringBuffer对象中追加字符串。**

**格式： StringBuffer str.append(String str1)；**

**str1字符串追加到str后面**

**替换字符：**teCharAt()**方法用于在字符串的指定索引位置替换一个字符**

**格式： StringBuffer str.setCharAt(索引值，要替换的字符)；**

**StringBuffer str = new StringBuffer(“hello”);**

**str.setCharAt(1,’E’); //此时str的值为 hEllo**

**反转字符串：reverse()方法对字符串进行反转【字符倒述，字符反过来输出】**

**格式： StringBuffer str.reverse()；**

**StringBuffer str = new StringBuffer(“java”);**

**str.reverse(); //此时str的值为 avaj**

**删除字符串：deleteCharAt()和delete()两个删除字符串的方法**

**格式： StringBuffer str.deleteCharAt(索引值);**

**StringBuffer str = new StringBuffer(“She”);**

**str.deleteCharAt(2); 此时str的值为Sh**

**格式： StringBuffer str.delete(int start,int end);**

**StringBuffer str = new StringBuffer(“hello java”);**

**str.delete(2,4); 此时str的值为heo java**

**Date**类：**（import java.Util.Date;）**

**Date() 表示时间默认顺序是 星期、月、日、小时、分、秒、年**

**Date(long date) 此种形式表示从GMT时间(格林尼治时间)**

**方法：**

**after(Date when)判断此日期是否在指定日期之后**

**before(Date when)判断此日期是否在指定日期之前**

**int compareTo(Date antherDate)比较两个日期的顺序**

**equals(Object obj)比较两个日期的相等性**

**long getTime()返回自1970年1月1日GMT以来，此Date对象表示的毫秒数**

**String toString()把此Date对象转换为以下形式的**

**String：星期、月、日、小时、分、秒、年**

**Scanner类：import java.Util.Scanner;**

**next()与nextline()区别：**

**next(): 1、一定要读取到有效字符后才可以结束输入。**

**2、对输入有效字符之前遇到的空白，next()方法会自动将其去掉。**

**3、只有输入有小字符后将其后面输入的空白作为分隔符或者结束符。**

**4、next()不能得到带有空格的字符串。**

**nextLine(): 1、以Enter为结束符，也就是说nextLine()方法 返回的是输入回车之前的所有字符。**

**2、可以获得空白。**

**数组： Array***类*：**import java.util.Arrays;**

|  |
| --- |
| **public static int binarySearch(Object[] a, Object key)** **用二分查找算法在给定数组中搜索给定值的对象(Byte,Int,double等)。数组在调用前必须排序好的。如果查找值包含在数组中，则返回搜索键的索引；否则返回 (-(插入点) - 1)。** |
| **public static boolean equals(long[] a, long[] a2) 如果两个指定的 long 型数组彼此相等，则返回 true。如果两个数组包含相同数量的元素，并且两个数组中的所有相应元素对都是相等的，则认为这两个数组是相等的。换句话说，如果两个数组以相同顺序包含相同的元素，则两个数组是相等的。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。** |
| **public static void fill(int[] a, int val) vals 数组a的下标 将指定的 int 值分配给指定 int 型数组指定范围中的每个元素。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。** |
| **public static void sort(Object[] a) 对指定对象数组根据其元素的自然顺序进行升序排列。同样的方法适用于所有的其 他基本数据类型（Byte，short，Int等）。** |

**数组比较equals() Arrays equals(数组A，数组B)**

**数组填充fill() Arrays.fill(数组名，数组下标)**

**数组查找binarySearch()**

**Arrays.****binarySearch(要查找的数组名，检索数组的元素)**

**Arrays.binarySearch(要查找的数组名,索引的起始值(包括开始处),索引的终止值(不包括终止值),检索数组的元素)**

**注：若在此范围找不到检索的元素，则返回-1或“-插入点”**

**数组复制copyOf() Arrays.copyOf(****strArray,length) //strArray要复制的数组；length新数组的长度 【 int[] newScores=(int[])Arrays.copyOf(scores,8); 】**

**CopyOfRange() Arrays.copyOfRange(strArray,int starIndex,int endIndex) strArray要复制的数组；starIndex表示开始索引位置（包括索引值），endIndex表示终止索引位置（不包括索引值）**

**注：目标数组如果已经存在，将会被重构**

**使用arraycopy()方法 java.lang.System**

**System.arraycopy( srcArray , srcIndex , destArray , destIndex , length)**

**srcArray表示原数组名；sraIndex表示原数组中起始索引位置；destArray表示新数组名；destIndex表示新数组中的起始索引位置；length表示要复制的数组（原）的长度**

**使用此方法复制数组时，length+srcIndex 必须小于等于 srcArray.length**

**同时 length+destIndex 必须小于等于 destArray.length**

**注意：目标数组必须已经存在，且不会被重构，相当于替换目标数组中的部分元素。**

**package** Java;

**import** java.util.Arrays;

**public** **class** Java02 {

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

//定义源数组，长度为 8

**int** scores[]=**new** **int**[]{100,81,68,75,91,66,75,100};

//定义目标数组

**int** newScores[]=**new** **int**[]{80,82,71,92,68,71,87,88,81,79,90,77};

System.***out***.println("源数组中的内容如下：");

//遍历源数组

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

System.***out***.print(scores[i]+"\t");

}

System.***out***.println("\n目标数组中的内容如下：");

//遍历目标数组

**for**(**int** j=0;j<newScores.length;j++){

System.***out***.print(newScores[j]+"\t");

}

System.*arraycopy*(scores,0,newScores,2,8);

//复制源数组中的一部分到目标数组中

System.***out***.println("\n替换元素后的目标数组内容如下：");

//循环遍历替换后的数组

**for**(**int** k=0;k<newScores.length;k++){

System.***out***.print(newScores[k]+"\t");

}

}

}

显示：

源数组中的内容如下：

100 81 68 75 91 66 75 100

目标数组中的内容如下：

80 82 71 92 68 71 87 88 81 79 90 77

替换元素后的目标数组内容如下：

80 82 100 81 68 75 91 66 75 100 90 77

**使用clone()方法复制数组**

**该方法是类Object中的方法，可以创建一个有单独内存空间的对象。因为数组也是一个Object类，因此也可以使用数组对象的clone()方法来复制数组。**

**clone()方法返回值是Object类型，要使用强制类型转换为恰当的类型。**

**格式： array\_name.clone()**

**实例：int[] targetArray=(int[])sourceArray.clone()**

**注：新数组如果已经存在，将会被※重构※**

**targeArray:旧数组 sourceArray:新数组**

**使用sort()方法数组排序**

**格式： Array.sort(数组名)**

怎样在Java中实现基本数据类型与字符之间的转换  
　　　摘要：在我们对Java的学习当中数据类型之间的转换，是我们常见的事，我们也都知道基本数据类型之间有自动转换和强制转换，在int   、 short  、 long 、float 、double之间的转换我们都比较清楚的吧，但我们也应该知道它们与字符串及字符型之间的类型转换又是怎样对，让我们一起来探讨一下它们的转换：  
1.整形与字符型之间的数据类型转换：  
　　一.int转换成char有两种方法：  
　　① 是利用char的unicode编码  
　　例：int num1 = 8;  
　　char ch1 = (char) (num1 + 48);  
　　二. char转换成int型  
　　① 使用int的parseint方法  
　　例： Character ch2 = '8'; （char是基本数据类型，Character是其包装类型。）  
　　int num2 = Integer.parseInt(ch2.toString());  
　　②是利用char的unicode编码  
　　例：char ch3 = '8';  
　　int num3 = ch3 - 48;  
2.基本数据类型与字符串之间的转换：  
　　一. int转换成String  
　　①.整型转换成字符型： String num = Integer.toString(int n);  
　　②.Long型转换成字符型： String num = Long.toString(long n);  
　　③.Short型转换成字符型： String num = Short.toString(Short n);  
　　④.Float型转换成字符型： String num = Float.toString(Float n);  
　　⑤.Double型转换成字符型： String num = Double.toString(Double n);  
　　二. String转换成int  
　　①.转换成Int型： int/Integer num = Integer.parseInt(String str);  
　　②.转换成long型： Long/long num = Long.parseLong(String str);  
　　③.转换成short型： short/Short num = Short.parseShort(String str);  
　　④.转换成float型： float/Float num = Float.parseFloat(String str);  
　　⑤.转换成double型： double/Double num = Double.parseDouble(String str);

## String转换为char

在Java中将String转换为char是非常简单的。   
1. 使用String.charAt(index)（返回值为char）可以得到String中某一指定位置的char。   
2. 使用String.toCharArray()（返回值为char[]）可以得到将包含整个String的char数组。这样我们就能够使用从0开始的位置索引来访问string中的任意位置的元素。

## char转换为String

将char转换为String大致有6种方法。总结如下：

**1. String s = String.valueOf('c'); //效率最高的方法**

**2. String s = String.valueOf(new char[]{'c'}); //将一个char数组转换成String**

**3. String s = Character.toString('c');**

**// Character.toString(char)方法实际上直接返回String.valueOf(char)**

**4. String s = new Character('c').toString();**

**5. String s = "" + 'c';**

**// 虽然这个方法很简单，但这是效率最低的方法**

**// Java中的String Object的值实际上是不可变的，是一个final的变量。**

**// 所以我们每次对String做出任何改变，都是初始化了一个全新的String Object并将原来的变量指向了这个新String。**

**// 而Java对使用+运算符处理String相加进行了方法重载。**

**// 字符串直接相加连接实际上调用了如下方法：**

**// new StringBuilder().append("").append('c').toString();**

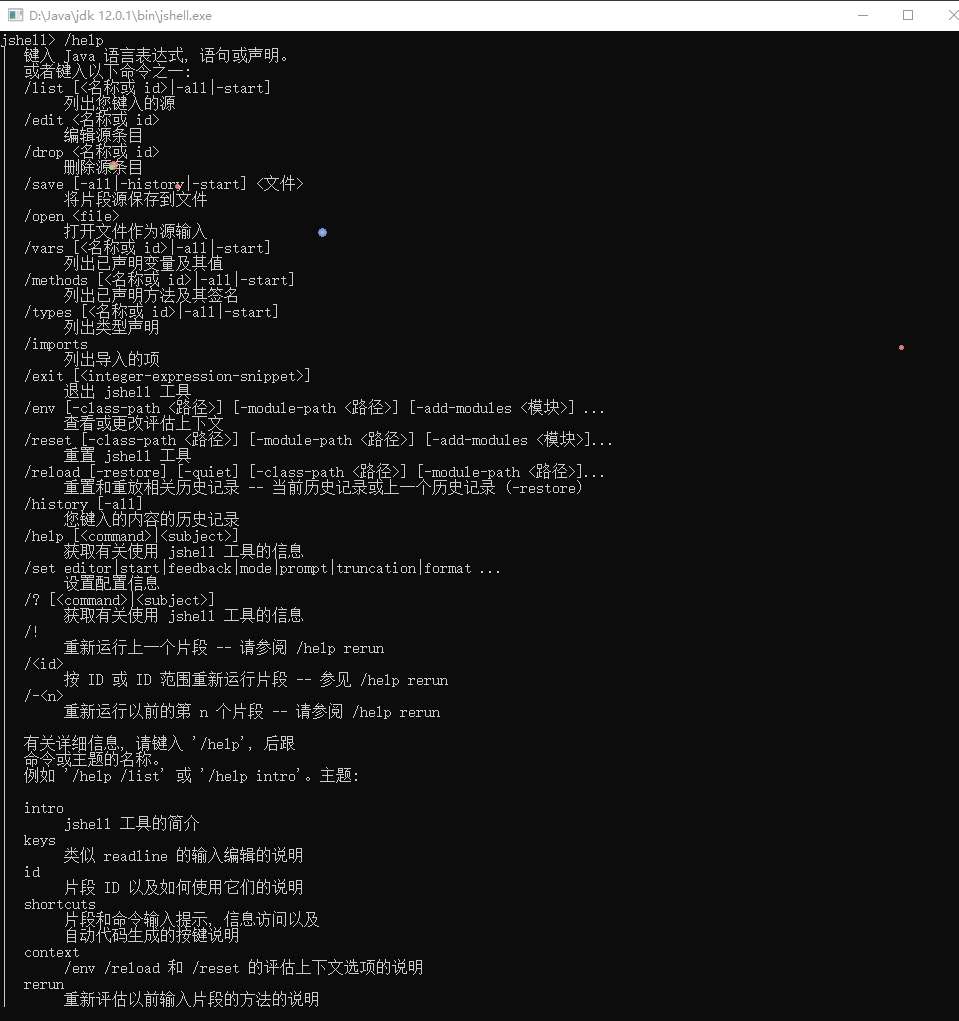
1. **String s = new String(new char[]{'c'})；**

**JDk 9 新增的**jshell工具

**启动jshell非常简单，只要在命令窗口输入jshell命令，即可进入jshell交互模式**

**进入交互模式后，可执行 /help 来查看帮助信息，也可执行 /exit 退出jshell**

**帮助信息入下图：**

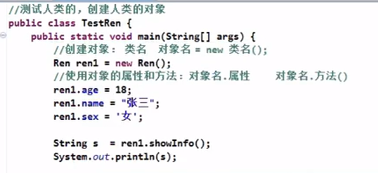
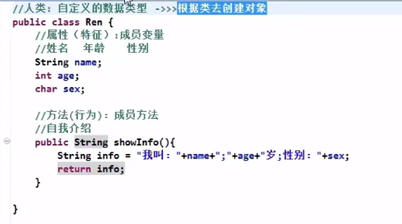


**类和对象**

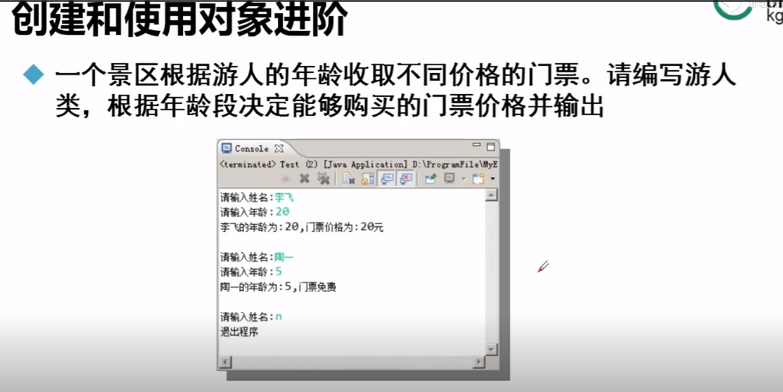
**万物皆对象**

**特性—>属性—>属性值 行为—>方法—>**

**类图工具：Visio Rose**



info：请你在命令行里，把当前信息打印出来



方法的形参和实参具有以下特点：

1、形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只有在方法内部有效，方法调用结束返回主调 方法后则不能再使用该形参变量。

2、实参可以是常量、变量、表达式、方法等，无论实参是何种类型的量，在进行方法调用时，它们都必须具有确定的值，以便把这些值传送给形参。因此应预先用赋值、输入等办法使实参获得确定值。

3、实参和形参在数量、类型和顺序上应严格一致，否则会发生“类型不匹配” 的错误。

4、方法调用中发生的数据传送是单向的，即只能把实参的值传送绐形参，而不能把形参的值反向地传送给实参。因此在方法调用过程中，形参的值发生改变，而实参中的值不会变化。

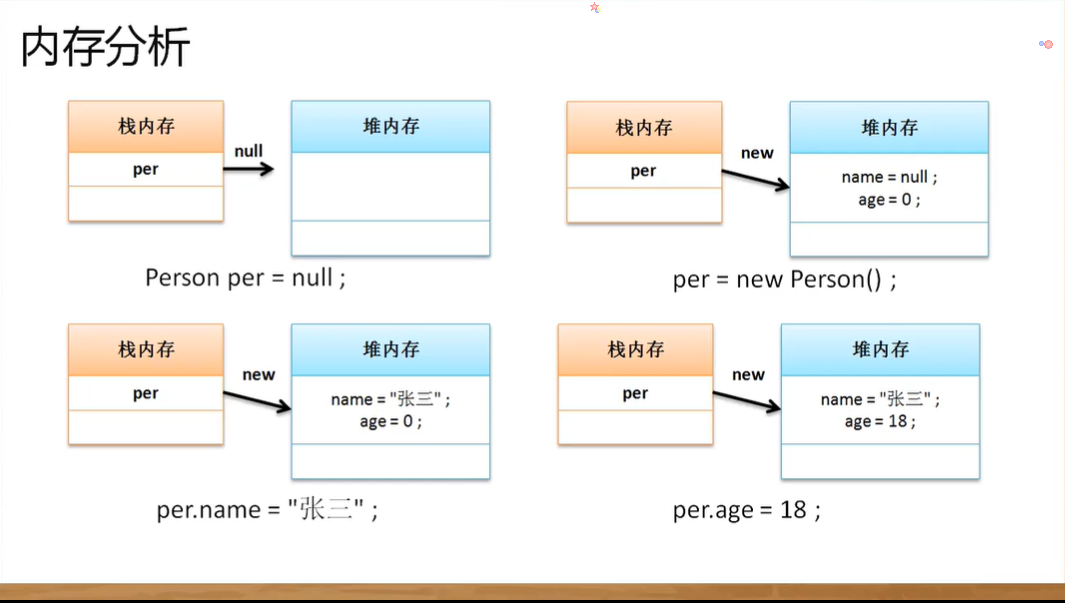
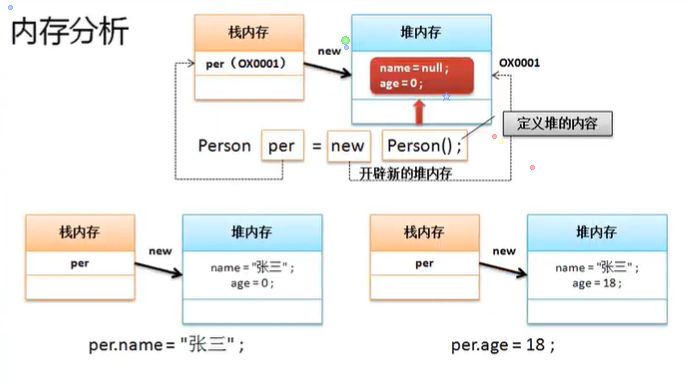
在 Java 虚拟机的堆区，每个对象都可能处于以下三种状态之一。  
 (1) 可触及状态：当一个对象被创建后，只要程序中还有引用变量引用它，那么它就始终处于可触及状态。  
 (2) 可复活状态：当程序不再有任何引用变量引用该对象时，该对象就进入可复活状态。在这个状态下，垃圾回收器会准备释放它所占用的内存，在释放之前，会调用它及其他处于可复活状态的对象的 finalize() 方法，这些 finalize() 方法有可能使该对象重新转到可触及状态。  
 (3) 不可触及状态：当 Java 虚拟机执行完所有可复活对象的 finalize() 方法后，如果这些方法都没有使该对象转到可触及状态，垃圾回收器才会真正回收它占用的内存。  
 注意：调用 System.gc() 或者 Runtime.gc() 方法也不能保证回收操作一定执行，它只是提高了 Java 垃圾回收器尽快回收垃圾的可能性。

内存分析：（最常用的内存空间）

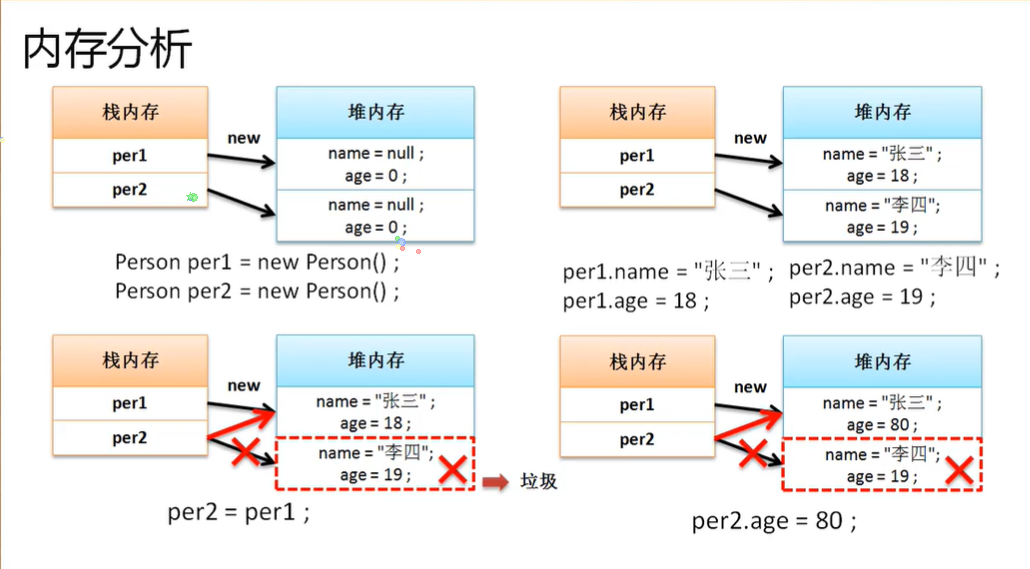
堆内存：保存的是对象的具体信息，在程序之中堆内存空间的开辟是通过new完成的；

栈内存：保存的是一块堆内存的地址，即：通过地址找到堆内存，而后找到对象内容，但是为了分析简化起见可以简单的理解为：将对象名称保存在栈内存之中。

**一个栈内存只能够保存有一个内存的地址数据，如果发生更改，则之前的地址数据将从此栈内存中彻底消失。**



**垃圾处理：**所谓的垃圾空间指的就是没有任何栈内存所指向的栈内存空间，所有的垃圾将被GC（Garbage Collection、垃圾收集器）不定期进行回收并且释放无用内存空间，但是如果垃圾过多，一定将影响到GC的处理性能，从而降低整体的程序性能，那么在时间的开发之中，对于垃圾的产生应该越少越好。



匿名对象：没有名字的对象 只能被使用一次自动变成垃圾

因为匿名对象只能使用一次所以一般我们都用作方法调用，调用一次 对象编程垃圾，完成一个功能就可以，这样节省了内存空间，减小了过多无用对象空占内存

作用：调用一个类的方法只使用一次的情况

作为参数传递

**构造方法的定义要求如下：**

构造方法名称必须与类名称保持一致；

构造方法不允许设置任何的返回值类型，即：没有返回值定义；

构造方法是在关键字new实例化对象的时候自动调用的。

**构造方法是在类对象实例化的时候调用的，而普通方法是在类对象实例化产生之后调用的。**

final关键字

1、final 关键字表示对象是最终形态的，对象是不可改变的意思。final 应用于类、方法和变量时意义是不同的，但本质是一样的：final 表示不可改变。

2、final 用在变量的前面表示变量的值不可以改变，此时该变量可以被称为常量；final 用在方法的前面表示方法不可以被重写；final 用在类的前面表示类不可以被继承，即该类是最终形态，如常见的 java.lang.String 类。

final 修饰符使用在如下方面：

#### 1、final 修饰类中的属性（变量）

表示该属性一旦被初始化便不可改变，这里不可改变的意思对基本类型来说是其值不可变，而对对象属性来说其引用不可再变。其初始化可以在两个地方：一是其定义处，也就是说在 final 属性定义时直接给其赋值；二是在构造函数中。这两个地方只能选其一，要么在定义时给值，要么在构造函数中给值，不能同时既在定义时赋值，又在构造函数中赋予另外的值。

#### 2、final 修饰类中的方法

说明这种方法提供的功能已经满足当前要求，不需要进行扩展，并且也不允许任何从此类继承的类来重写这种方法，但是继承仍然可以继承这个方法，也就是说可以直接使用。在声明类中，一个 final 方法只被实现一次。

#### 3、final 修饰类

表示该类是无法被任何其他类继承的，意味着此类在一个继承树中是一个叶子类，并且此类的设计已被认为很完美而不需要进行修改或扩展。

对于 final 类中的成员，可以定义其为 final，也可以不是 final。而对于方法，由于所属类为 final 的关系，自然也就成了 final 型。也可以明确地给 final 类中的方法加上一个final，这显然没有意义。

使用 main() 方法时应该注意如下几点：

访问控制权限必须是公有的（public）。

main() 方法是静态的。如果要在 main() 方法中调用本类中的其他方法，则该方法也必须是静态的，否则需要先创建本类的实例对象，然后再通过对象调用成员方法。

main() 方法没有返回值，只能使用 void。

main() 方法具有一个字符串数组参数，用来接收执行 [Java](http://c.biancheng.net/java/" \t "_blank) 程序的命令行参数。命令行参数作为字符串，按照顺序依次对应字符串数组中的元素。

除了形参变量名可以任意设置以外，main() 方法中的其他内容都是固定不变的。

# **Java方法的可变参数**

声明可变参数的语法格式：

methodName({paramList},paramType…paramName)

methodName表示方法名称； paramList表示方法的固定参数列表； paramType表示可变参数的类型； … 是表示可变参数的标识； paramName表示可变参数名称

注意：可变参数必须定义在参数列表的最后

static 修饰符

每次参加考试的人数是不固定的，但是每次考试完之后都需要打印出本次考试的总人数以及参加考试的学生名单。下面编写程序，使用方法的可变参数实现该功能，具体的代码如下：

public class StudentTestMethod{

//定义输出考试学生的人数及姓名的方法

public void print(String...names){

int count = names.length; //获取总个数

System.out.println("本次参加考试的有"+count+"人，名单如下：");

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

System.out.println(names[i]);

}

}

public static void main(String[] args){

//TODO Auto-generated method stub

StudentTestMethod student = new StudentTestMethod();

student.print("张强","李成","王勇"); //传入3个值

student.print("马丽","陈玲");

}

}

在 Student TestMethod 类中定义了 print() 方法和 main() 方法。print() 方法声明了一个 String 类型的可变参数，方法体打印可变参数的总个数以及参数值。在 main() 方法中创建了 StudentTestMethod 类的实例，然后分别传入不同个数的参数调用 print() 方法。

运行结果如下：

本次参加考试的有3人，名单如下：

张强

李成

王勇

本次参加考试的有2人，名单如下：

马丽

陈玲

static 可以用来修饰

成员变量

静态变量，可以直接通过类名访问

成员方法

静态方法，可以直接通过类名访问

代码块

静态代码块，当Java虚拟机加载类时，就会执行该代码块

static变量

类的成员变量包括

类变量（静态变量）

1、被static修饰的变量

2、在内存中只有一个拷贝（也就是只拷贝一次）

3、类内部，可在任何方法内直接访问静态变量

4、其他类中，可以直接通过类名访问

实例变量

1、没有被static修饰的变量

2、每创建一个实例，就会为实例变量分配一次内存，（创建多个，互不影响）

实例变量可以在内存中有多个拷贝，互不影响

static变量的作用：

（1）能被类的所有实例共享，可作为实例之间进行交流的共享数据

（2）如果类的所有实例都包含一个相同的常量属性，可把这个属性定义为静态常量类型，从而节省内存空间

Static方法

静态方法：可直接同哦过类名访问

静态方法中不能使用this和super

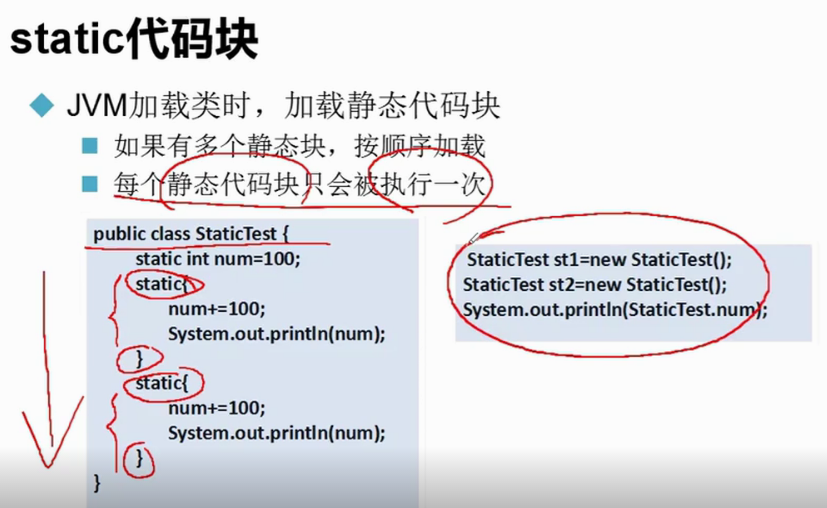
不能直接访问所属类的实例变量和实例方法

可直接访问类的静态变量和静态方法

实例方法：通过实例访问

可直接访问所属类的静态变量、静态方法、实例变量和实例方法

静态方法必须被实现



被static 修饰的变量和方法都能用类直接调用，

而不被static修饰的是实例方法和变量，则要通过对象来调用

普通方法必须要用创建对象来调用

在方法里不可以定义static变量



内部类

内部类的特点如下：

1. 内部类仍然的一个独立的类，在编辑之后内部类会编译成独立的.class文件，但是前面冠以外部类的类名和$符号
2. 内部类不能用普通的方式访问。内部类是外部类的一个成员，因此内部类可以自由地访问外部类的成员变量无论是否为private的
3. 内部类声明成静态的，就不能随便访问外部类的成员变量，仍然是只能访问外部类的静态成员变量（任何情况下静态只能访问静态的）

|  |
| --- |
| public class Test{  public class InnerClass{  public int getSum(int x,int y){  return x+y;  }  }  public static void main(String[] args){  Test.InnerClass ti=new Test().new InnerClass();  int i=ti.getSum(2/3);  System.out.println(i); //输出5  }  } |

|  |
| --- |
| 有关内部类的说明有如下几点：   * 外部类只有两种访问级别：public和默认；内部类则有4种访问级别：public、protected、private和默认 * 在外部类中可以直接通过内部类的类名访问内部类 * InnerClass ic = new InnerClass(); //InnerClass为内部类的类名 * 在外部类以外的其他类中则需要通过内部类的完整类名访问内部类 * Test.InnerClass ti = new Test().new InnerClass(); //Test.InnerClass是内部类的完整类名 * 内部类与外部类不能重名   提示： 内部类的很多访问规则可以参考变量和方法 |

实例内部类 是指没有用 static 修饰的内部类

|  |
| --- |
| 示例代码如下：  public class Outer{  class Inner  {  //实例内部类  }  } |

实例内部类有如下特点：

1. 在外部类的静态方法和外部类以外的其他类中，必须通过外部类的实例创建内部类的实例

|  |
| --- |
| public class Outer{  class Inner1{}  Inner1 i=new Inner1(); //在外部类里，不需要创建外部类实例  public void method1(){  Inner1 i=new Inner1(); //在外部类的实例方法里，不需要创建外部类实例  }  public static void method2(){  Inner1 i=new Outer().new inner1(); //在外部类的静态方法里，需要创建外部类实例  }  class Inner2{  Inner1 i=new Inner1(); //在外部类的内部类中，不需要创建外部类实例  }  }  class OtherClass{  Outer.Inner i=new Outer().new Inner(); //在其他类中，需要创建外部类实例  } |

1. 在实例内部类中，可以访问外部类的所有成员

|  |
| --- |
| public class Outer{  public int a=100;  static int b=100;  final int c=100;  private int d=100;  public String method1(){  return "实例方法1";  }  public static String method2(){  return "静态方法2";  }  class Inner{  int a2=a+1; //访问public的a  int b2=b+1; //访问static的b  int c2=c+1; //访问final的c  int d2=d+1; //访问private的d  String str1=method1(); //访问实例方法method1  String str2=method2(); //访问静态方法method2  }  public static void main(String[] args){  Inner i=new Outer().new Inner(); //在外部类静态方法里，创建内部类实例  System.out.println(i.a2); //输出：101  System.out.println(i.b2); //输出：101  System.out.println(i.c2); //输出：101  System.out.println(i.d2); //输出：101  System.out.println(i.strl); //输出：实例方法1  System.out.println(i.str2); //输出：静态方法2  }  } |

提示：如果有多层嵌套，则内部类可以访问所有外部类的成员

1. 在外部类中不能直接访问内部类的成员，而必须通过内部类的实例去访问。如果类A包含内部类B，类B中包含内部类C，则在类A中不能直接访问类C，而应该通过类B的实例去访问类C

|  |
| --- |
| **public class** A{  **int** a=100;  **int** b=*100*;  **class** B{  **int** a2=a+1;  **int** b2=b+1;  **class** C{  **int** a3=a+1;  **int** b3=b+1;  }  }  **public** **static** **void** main(String[] args) {  A k = **new** A();  System.***out***.println(k.a); //输出100  System.***out***.println(k.b); //输出100  B i=**new** A().**new** B();  System.***out***.println(i.a2); //输出101  System.***out***.println(i.b2); //输出101  B.C j = **new** A().**new** B().**new** C();  System.***out***.println(j.a3); //输出102  System.***out***.println(j.b3); //输出102  }  } |

1. 外部类实例与内部类实例是一对多的关系，也就是说一个内部类实例只对应一个外部类实例，而一个外部类实例可以对应多个内部类实例

如果实例内部类B与外部类A包含有同名的成员t，则在类B中t和this.t都表示B中的成员t，而A.this.t表示A中的成员t

|  |
| --- |
| **public** **class** AA{  **int** a=100;  **int** b=100;  **class** BB{  **int** a=AA.**this**.a+1;  **int** b=AA.**this**.b+1;  **class** CC{  **int** a=AA.BB.**this**.a+1;  **int** b=AA.BB.**this**.b+1;  }  }  **public** **static** **void** main(String[] args) {  AA k = **new** AA();  System.***out***.println(k.a); //输出100  System.***out***.println(k.b); //输出100  BB i = **new** AA().**new** BB();  System.***out***.println(i.a); //输出101  System.***out***.println(i.b); //输出101  BB.CC j = **new** AA().**new** BB().**new** CC();  System.***out***.println(j.a); //输出102  System.***out***.println(j.b); //输出102  }  } |

1. 在实例内部类中不能定义static成员，除非同时使用final(和static)修饰 【static貌似都不能定义】

静态内部类 是指使用 static 修饰的内部类

|  |
| --- |
| public class Outer  {  static class Inner  {  //静态内部类  }  } |

静态内部类有如下几个特点：

1. 在创建静态内部类的实例时，不需要创建外部类的实例

|  |
| --- |
| **public** **class** Dome02{  **static** **class** Inner{}  }  **class** OtherClass{  Dome02.Inner oi = **new** Dome02.Inner(); //外部类以外的其他类来调用创建实例时【外部类.内部类】  } |

1. 静态内部类中可以定义静态成员和实例成员。外部类以外的其他类需要通过完整的类名访问静态内部类中的静态成员，如果要访问静态内部类中的实例成员，则需要通过静态内部类的实例

|  |
| --- |
| **public** **class** Outer{  **static** **class** Inner{  **int** a=0; //实例变量a  **static** **int** *b* = 0; //静态变量 b  }  }  **class** OtherClass{  Outer.Inner oi = **new** Outer.Inner();  **int** a2 = oi.a; //访问实例成员【**new** Outer.Inner().a】创建对象才能完整的类名访问  **int** b2 = Outer.Inner.*b*; //访问静态成员：外部类名.内部类名.静态变量  } |

1. 静态内部类可以直接访问外部类的静态成员，如果要访问外部类的实例成员，则需要通过外部类的实例去访问。静态内部类可以直接访问外部类的静态成员，如果要访问外部类的实例成员，则需要通过外部类的实例去访问。

|  |
| --- |
| **public** **class** Outer{  **int** a = 0; //实例变量  **static** **int** *b* = 0; //静态变量  **static** **class** Inner{  Outer o = **new** Outer;  **int** a2 = o.a; //访问实例变量 创建实例再访问  **int** b2 = *b*; //访问静态变量  }  } |

# 局部内部类 局部内部类是指在一个方法中定义的内部类

|  |
| --- |
| **public** **class** Test{  **public** **void** method(){  **class** Inner{  //局部内部类  }  }  } |

局部内部类有如下特点：

1. 局部内部类与局部变量一样，不能使用访问控制修饰符（public、private 和 protected）和 static 修饰符修饰
2. 局部内部类只在当前反腐中有效

|  |
| --- |
| **public** **class** Test{  Inner i = **new** Inner(); //编译出错  Test.Inner ti = **new** Test.Inner(); //编译出错  Test.Inner ti2 = **new** Test().**new** Inner(); //编译出错  **public** **void** method(){  **class** Inner{}  Inner i = **new** Inner();  }  } |

(3) 局部内部类中不能定义 static 成员  
(4) 局部内部类中还可以包含内部类，但是这些内部类也不能使用访问控制修饰符（public、 private 和 protected）和 static 修饰符修饰  
(5) 在局部内部类中可以访问外部类的所有成员  
(6) 在局部内部类中只可以访问当前方法中 final 类型的参数与变量。如果方法中的成员与外部类中的成员同名，则可以使用<OuterClassName>.this.<MemberName> 的形式访问外部类中的成员 [外部类.this.外部类的成员变量名]

|  |
| --- |
| **public** **class** Test{  **int** a=0;  **int** d=0;  **public** **void** method(){  **int** b=0;  **final** **int** c=0;  **final** **int** d=10;  **class** Inner{  a2=a; //访问外部类中的成员  //int b2=b; //编译出错  **int** c2=c; //访问方法中的成员  **int** d2=d; //访问方法中的成员  **int** d3=Test.**this**.d; //访问外部类中的成员[外部类.this.外部类的成员变量名]  }  Inner i=**new** Inner();  System.***out***.println(i.d2); //输出10  System.***out***.printIn(i.d3); //输出0  }  **public** **static** **void** main(String[] args){  Test t=**new** Test();  t.method();  }  } |

匿名内部类是指没有类名的内部类，必须在创建是使用new语句来声明类

|  |
| --- |
| new<类或接口>()  {  //类的主体  }; |

这种形式的 new 语句声明一个新的匿名类，它对一个给定的类进行扩展，或者实现一个给定的接口。使用匿名类可使代码更加简洁、紧凑，模块化程度更高

匿名类有两种实现方式：①继承一个类，重写其方法②实现一个接口（可以是多个），实现其方法

|  |
| --- |
| class Out{  void show(){  System.***out***.println("调用 Out 类的 show() 方法");  }  }  public class TestAnonymousInterClass{  //在这个方法中构造一个匿名内部类  private void show(){  Out anonyInter = new Out()  {  //获取匿名内部类的实例  void show(){  System.***out***.println("调用匿名类中的 show() 方法");  }  };  anonyInter.show(); //调用上面创建的Out对象里面的show()方法  }  public static void main(String[] args){  TestAnonymousInterClass test = new TestAnonymousInterClass();  test.show();  }  } |
| 程序的输出结果如下：  调用匿名类中的 show() 方法 |

提示：匿名内部类实现一个接口的方式与实现一个类的方法相同

1. 匿名类和局部内部类一样，可以访问外部类的所有成员 。如果匿名类位于一个方法中，则匿名类只能访问方法中 final类型的局部变量和参数

|  |
| --- |
| public static void main(String[] args){  int a=10;  final int b=10;  Out anonyInter=new Out(){  void show(){  //System.out.println("调用了匿名类的 show() 方法"+a); //编译出错  System.out.println("调用了匿名类的 show() 方法"+b); //编译通过  }  };  anonyInter.show(); //调用上面创建的Out对象里面的show()方法  } |

1. 匿名类中允许使用非静态代码块进行成员初始化操作

|  |
| --- |
| Out anonyInter=new Out(){  int i;  { //非静态代码块  i=10; //成员初始化  }  public void show(){  System.out.println("调用了匿名类的 show() 方法"+i);  }  }; |

1. 匿名类的非静态代码块会在父类的构造方法之后被执行

# Java内部类的作用：Java利用内部类实现多重继承

**继承**

继承关键字

继承

使用继承

编写父类

[访问修饰符]class Pet {

//公共的属性和方法

}

编写子类，继承父类

[访问修饰符]class Dog **extends** Pet {

//子类特有的属性和方法

}

继承是Java中实现代码重用手段之一。Java中只支持单根继承，即一个类只能有一个直接父类

在何处使用使用继承（何时使用继承？）

继承与真实世界类似

只要说“猫是哺乳动物”，猫的很多属性、行为就不言自明了 藏獒是一种狗

符合is-a关系的设计使用继承

继承是代码重用的一种方式

将子类共有的属性和行为放到父类中

子类继承父类的什么？

继承public和protected修饰的属性和方法，不管子类和父类是否在同一个包里

继承默认权限修饰符修饰的属性和方法，但子类和父类必须在同一个包里

不能被继承的父类成员

1. private成员
2. 子类与父类不在同包，使用默认访问权限的成员
3. 构造方法

子类访问父类成员

访问父类构造方法

* super();
* super(name);

访问父类属性

* super.name;

访问父类方法

* super.print()

1. 使用super关键字，super代表父类对象
2. 在子类构造方法中调用且必须是第一句
3. 不可以访问父类中定义为private的属性和方法

小结

super **关键字来访问父类的成员**

* super 只能出现子类的方法和构造方法
* super调用构造方法时，只能是第一句
* super 不能访问父类的private成员

什么的方法的重写

* 方法的重写或方法的覆盖（overrding）
* 子类根据需求对父类的继承的方法进行重新编写
* 重写时，可以用super.方法的方式来保留父类的方法
* 构造方法不能被重写

继承条件下的构造方法

* 继承条件下构造方法的调用规则
* 子类构造方法没有通过super显式调用父类的有参构造方法，

也没通过this显式自身调用自身其他构造方法

* + 系统默认调用父类的无参构造方法
* 子类构造方法通过super显式调用父类的有参构造方法
  + 执行父类相应构造方法，而不执行父类无参构造方法
* 子类构造方法通过this显式调用自身的其他构造方法，在相应构造方法中应用以上两条规则

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 比较项 | 位置 | 方法名 | 参数表 | 返回值 | 访问修饰符 |
| 方法重写 | 子类 | 相同 | 相同 | 相同或是其子类 | 不能比父类更严格 |
| 方法重载 | 同类 | 相同 | 不相同 | 无关 | 无关 |

方法重写规则

* 方法名相同
* 参数列表相同
* 返回值类型相同 或者是其子类
* 访问权限不能严于父类
* 父类的静态方法不能被子类覆盖为非静态方法，父类的非静态方法不能被子类覆盖为静态方法
* 子类可以定义与父类同名的静态方法，  
  以便在子类中隐藏父类的静态方法
* 父类的私有方法不能被子类覆盖
* 不能抛出比父类方法更多的异常

方法覆盖（方法重写）的原则：

覆盖方法的返回类型、方法名称、参数列表必须与原方法的相同。

覆盖方法不能比原方法访问性差（即访问权限不允许缩小）。

覆盖方法不能比原方法抛出更多的异常。

被覆盖的方法不能是final类型，因为final修饰的方法是无法覆盖的。

被覆盖的方法不能为private，否则在其子类中只是新定义了一个方法，并没有对其进行覆盖。

被覆盖的方法不能为static。如果父类中的方法为静态的，而子类中的方法不是静态的，但是两个方法除了这一点外其他都满足覆盖条件，那么会发生编译错误；反之亦然。即使父类和子类中的方法都是静态的，并且满足覆盖条件，但是仍然不会发生覆盖，因为静态方法是在编译的时候把静态方法和类的引用类型进行匹配。

方法的重载：

前面已经对Java方法重载进行了说明，这里再强调一下，Java父类和子类中的方法都会参与重载，

例如，父类中有一个方法是 func(){ ... }，子类中有一个方法是 func(int i){ ... }，就构成了方法的重载。

覆盖和重载的不同：

方法覆盖要求参数列表必须一致，而方法重载要求参数列表必须不一致。

方法覆盖要求返回类型必须一致，方法重载对此没有要求。

方法覆盖只能用于子类覆盖父类的方法，方法重载用于同一个类中的所有方法（包括从父类中继承而来的方法）。

方法覆盖对方法的访问权限和抛出的异常有特殊的要求，而方法重载在这方面没有任何限制。

父类的一个方法只能被子类覆盖一次，而一个方法可以在所有的类中可以被重载多次。

Object类

Object类被子类经常重写的方法

|  |  |
| --- | --- |
| 方 法 | 说 明 |
| toString() | 返回当前对象本事的有关信息，按字符串对象返回 |
| equals() | 比较两个对象是否是同一个对象，是则返回true |
| heshCode() | 返回该对象的哈希代码值 |
| getClass() | 获取当前对象所属的类信息，返回Class对象 |

1. Object类的equals()方法与==没区别
2. 当有特殊需求，如认为属性相同即为同一对象时，需要重写equals()
3. Java.lang.String重写了equals()方法，把equals()方法的判断变为了判断值

equals()

Object类的equals()方法

—比较两个对象是否是同一个对象，是则返回true

—操作符 ==

* 简单数据类型，直接比较值。如：1 == 2
* 引用类型，比较两者是否为同意对象

hashCode()  
 散列码(hashCode)是按照一定的算法由对象得到的一个数值，散列码没有规律。如果 x 和 y 是不同的对象，x.hashCode() 与 y.hashCode() 基本上不会相同。  
 hashCode() 方法主要用来在集合中实现快速查找等操作，也可以用于对象的比较。  
在 Java 中，对 hashCode 的规定如下：  
 在同一个应用程序执行期间，对同一个对象调用 hashCode()，必须返回相同的整数结果——前提是 equals() 所比较的信息都不曾被改动过。至于同一个应用程序在不同执行期所得的调用结果，无需一致。  
 如果两个对象被 equals() 方法视为相等，那么对这两个对象调用 hashCode() 必须获得相同的整数结果。  
 如果两个对象被 equals() 方法视为不相等，那么对这两个对象调用 hashCode() 不必产生不同的整数结果。然而程序员应该意识到，对不同对象产生不同的整数结果，有可能提升hashTable（后面会学到，集合框架中的一个类）的效率。

简单地说：如果两个对象相同，那么它们的 hashCode 值一定要相同；如果两个对象的 hashCode 值相同，它们并不一定相同。在 Java 规范里面规定，一般是覆盖 equals() 方法应该连带覆盖 hashCode() 方法。

//重写equals()：告诉计算机一个新的比较规则：如果学号和名字都相同，我们认为就是同一个对象

//把你传过来的obj对象和this（当前学员对象）比较

//instanceof操作符的作用：判断某一个对象是否属于某一种类型

public boolean equals(Object obj) {

if (this == obj){

return true;

}

if (!(obj instanceof Student)){

return false

}

Student s = (Student)obj;

if (this.sid == s.sid && this.name.equals(s.name)){

return true;

}else{

return false;

}

}

instanceof用于判断一个引用类型所引用的对象是否是一个类的实例

toString()  
 toString()方法是 Object 类中定义的另一个重要方法，是对象的字符串表现形式，语法为：  
    public String toString()  
 返回值是 String 类型，用于描述当前对象的有关信息。Object 类中实现的 toString() 方法是返回当前对象的类型和内存地址信息，但在一些子类（如 String、Date 等）中进行了 重写，也可以根据需要在用户自定义类型中重写 toString() 方法，以返回更适用的信息。  
 除显式调用对象的 toString() 方法外，在进行 String 与其它类型数据的连接操作时，会自动调用 toString() 方法。

多态

多态性是面向对象编程的又一个重要特征，它是指在父类中定义的属性和方法被子类继承之后，可以具有不同的数据类型或表现出不同的行为，这使得同一个属性或方法在父类及其各个子类中具有不同的含义。  
  
对面向对象来说，多态分为编译时多态和运行时多态。其中编译时多态是静态的，主要是指方法的重载，它是根据参数列表的不同来区分不同的方法。通过编译之后会变成两个不同的方法，在运行时谈不上多态。而运行时多态是动态的，它是通过动态绑定来实现的，也就是大家通常所说的多态性。

[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 实现多态有 3 个必要条件：继承、重写和向上转型。只有满足这 3 个条件，开发人员才能够在同一个继承结构中使用统一的逻辑实现代码处理不同的对象，从而执行不同的行为。

* 继承：在多态中必须存在有继承关系的子类和父类
* 重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法
* 向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用既能调用父类的方法，又能调用子类的方法

方法重写是实现多态的基础

多态—1

同一种事物，由于条件不同，产生的结果也不同

同一个引用类型，使用不同的实例而执行不同操作

多态—2

又要给xxx类继承Pet类（旧方案也需要）

创建xxx类对象（旧方案也需要）

其他代码不变（不用修改Master类）

多态—3

Pet类的toHospital()如何实现呢？

— toHospital()不需要有具体的实现

— 抽象方法（abstract）

— Pet类声明为抽象类

— 实例化Pet毫无意义

抽象方法

.

public abstract class Pet{

public void toHospital(){

}

}

每个子类的实现不同

public abstract void print() ;

抽象方法

* 以下代码有什么问题？
* 抽象方法

— 抽象方法没有方法体

— 抽象方法必须在抽象类里

— 抽象方法必须在子类中被实现，除非子类是抽象类；

* 有抽象方法的类一定是抽象类
* 抽象类不一定有抽象方法
* 抽象类不能被实例化（Pet pet = new Pet()）
* 抽象类的子类可以是普通类，该普通类要实现抽象类中的抽象方法
* 抽象类的子类也可以是抽象类，此时，父类中抽象方法可以在子类中不被实现

向上转型

—<父类型><引用变量名> = new <子类型>();

—此时通过父类引用变量调用的方法是子类覆盖或继承父类的方法，不是父类的方法

—此时通过父类引用变量无法调用子类特有的方法

向下转型

—<子类型><引用变量名> = (<子类型>)<父类型的引用变量>;

—在乡下转型的过程中，如果没有转换为真实子类类型，会出现类型转换异常

instanceof 如何减少在向下转型的过程中，没有转换为真实子类类型的类型转换异常？

Java中提供了instanceof运算符来进行类型的判断

——使用instanceof时，对象的类型必须和instanceof后面的参数所指定的类在继承上有上下级关系

抽象构造方法

抽象方法

抽象类

public abstract class Pet{

public abstract void toHospital();

}

抽象类与抽象方法

回顾如下代码

* 抽象类VS普通类
* 抽象类不能被实例化
* 但可以创建一个引用变量，其类型是一个抽象类，指向非抽象的子类实例
* 抽象方法VS普通方法
* 有无方法体
* 抽象类与抽象方法的使用
* 抽象类中可以没有抽象方法，但包含抽象方法的类必须被定义为抽象类
* 如果子类没有实现父类的所有抽象方法，子类必须被定义为抽象类
* 没有抽象构造方法，也没有抽象静态方法 ~~( public abstract 类名( ){ } ; )~~
* 抽象类中可以有非抽象的构造方法，创建子类的实例时可能调用

public 类名( ){ } √

接口（比抽象还抽象）

接口的特性

* 接口不可以被实例化（常作为类型使用）
* 实现类必须实现接口的所有方法
* 实现类可以实现多个接口（Java中的多继承）
* implements、多个接口使用逗号隔开
* 接口的变量都是静态常量（public static final）
* 所有方法默认的都是：public abstract

概念性的接口，系统对外提供的所有服务

实在的接口，被interface修饰

接口表示一种能力

* “做这项工作需要一个钳工（木匠/程序员）”
* 钳工是一种“能力”，不关心具体 是谁
* 接口是一种能力
* 体现在接口的方法上
* 面向接口的编程
* 程序设计时
* 关心实现类有何能力，而不关心实现细节
* 面向接口的约定而 不考虑接口的具体实现过程

接口是一种约定

* 生活中，我们使用的两相电源插座，规定了：
* 两个接头见的额定电压
* 两个接头见的距离
* 接头的形状
* 接口是一种约定

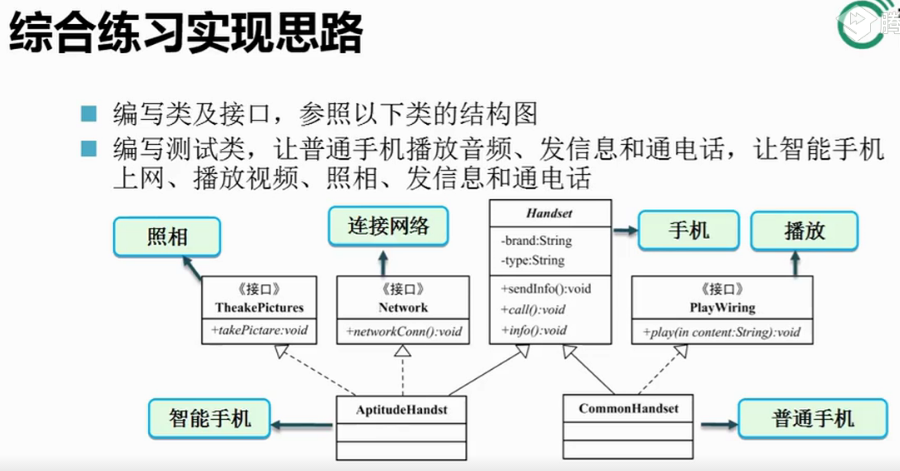
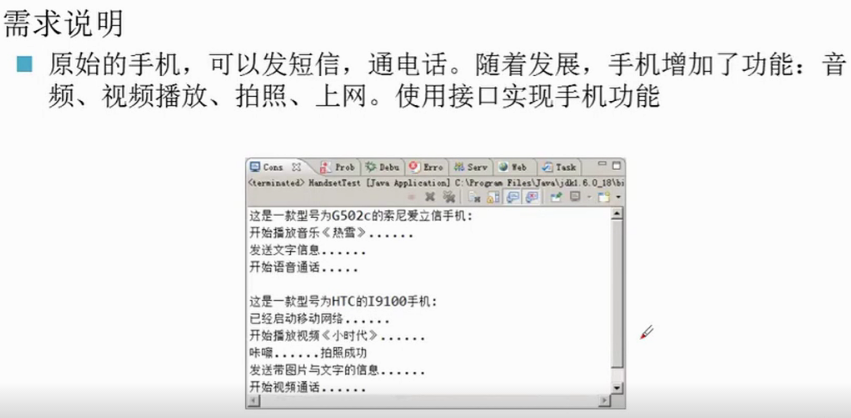
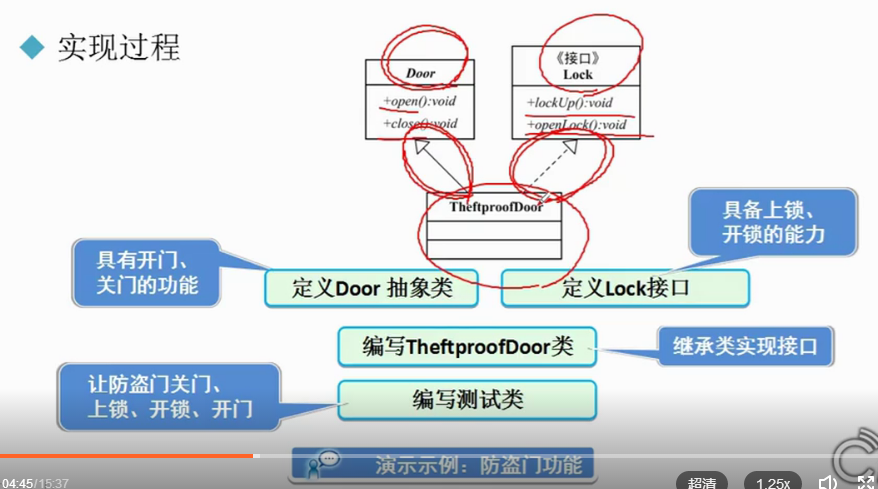
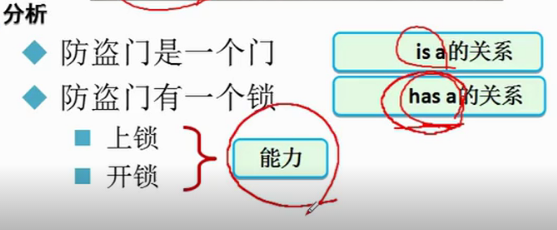
有些接口只有名称

* 面向接口编程

程序设计时面向接口的约定而不考虑具体实现

接口使用

* 接口中的成员（全局）变量
* 默认都是public static final 的，必须显式初始化
* 接口中的方法Motorola
* 默认都是public abstract 的
* 除了成员变量和方法，不能包含其他
* 接口没有构造方法，不能被实例化
* 一个接口不能实现另一个接口，但可以继承多个其他接口（接口可相互继承）
* 一个类必须实现接口抽象方法（implements），除非这个类也是抽象类



抽象类VS接口

* 相同点
* 代表系统的抽象层
* 都不能被实例化
* 都能包含抽象方法
* 用于描述系统提供的服务，不必提供具体实现
* 异同点
* 在抽象类中可以为部分方法提供默认实现，而接口中只能包含抽象方法
* 一个类只能坚持以个直接的父类，但可以实现多个接口
* 已存在的继承树，可以方便的抽取接口，但是抽取抽象类不容易
* 使用原则
* 接口做系统与外界交互的窗口
* 接口提供服务
* 接口本身一旦制定，就不允许随意修改
* 抽象类可完成部分功能实现，还有部分功能可作为系统的扩展点

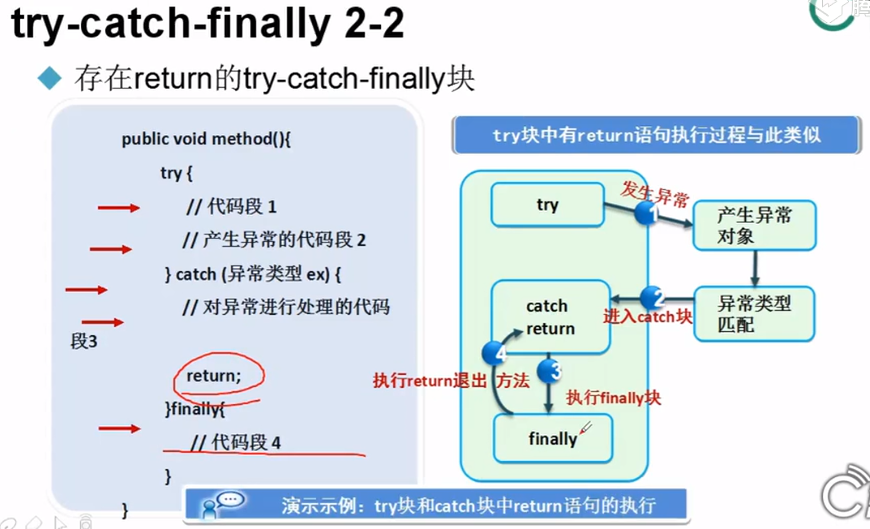
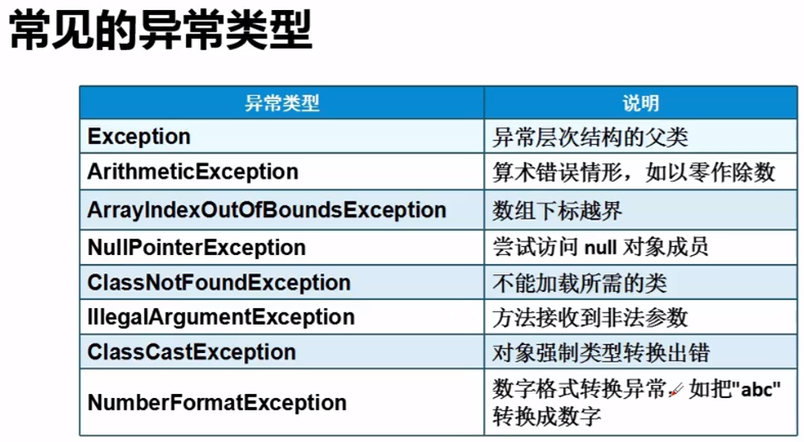
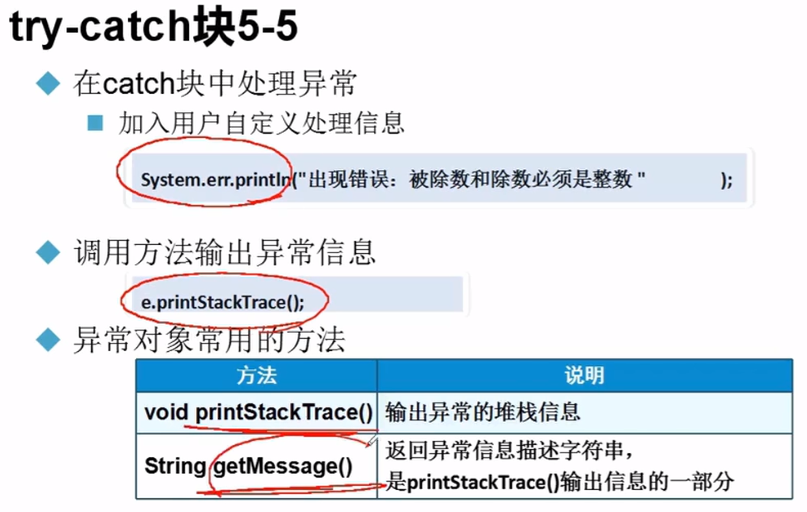
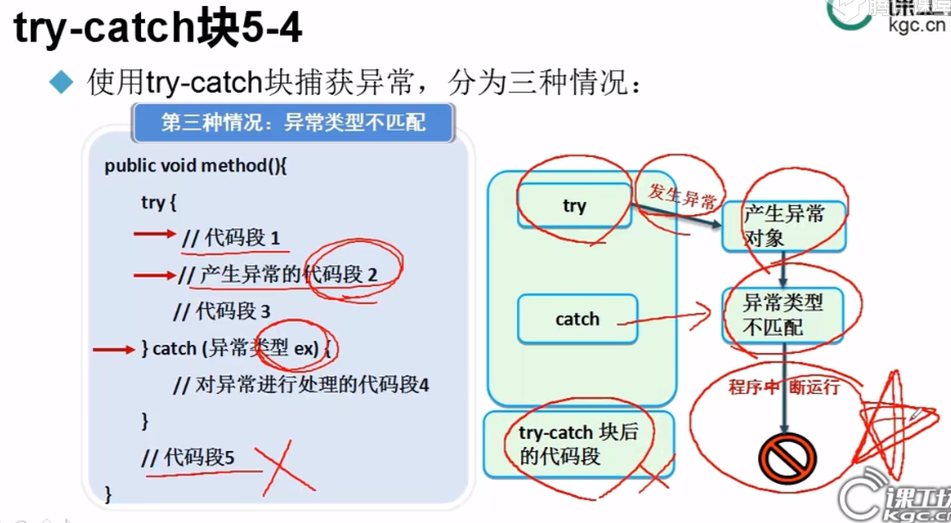
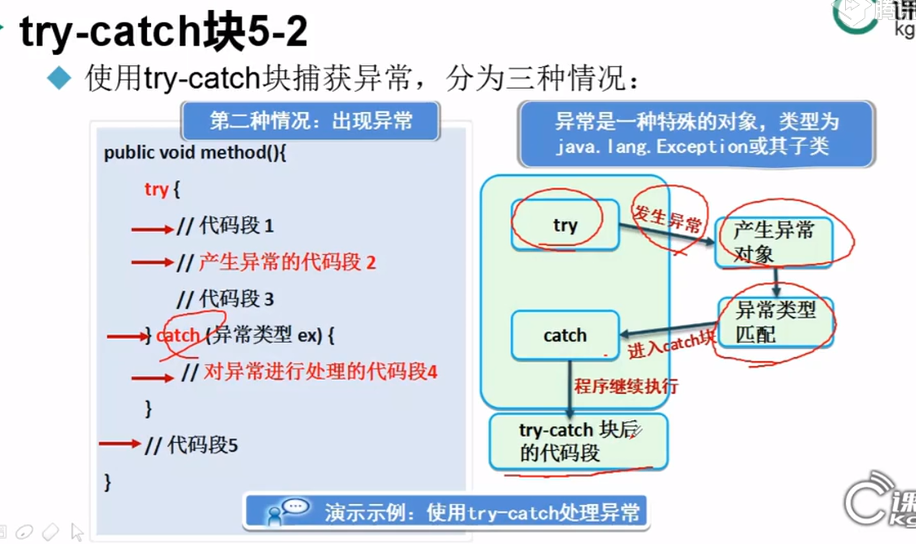
异常：指在程序的运行过程中所发生的不正常的所发生的不正常的事件，它会中断正在运行的程序

异常处理：Java编程语言使用异常处理机制为程序提供了错误处理的能力

Java的异常处理是通过5个关键字来实现的：try、catch、finally、throw、throws

|  |  |
| --- | --- |
| 捕  获  异  常 | try : 执行可能发生异常的代码 |
| catch : 捕获异常 |
| finally : 无论是否发生异常，代码总能执行 |
| 声明异常 | throws : 声明方法可能要抛出的各种异常 |
| 抛出异常 | throw : 手动抛出异常 |

|  |
| --- |
| try{  可能发生异常的代码  }catch(Exception 变量名){  System.err.println(“出现了错误操作！”);  //异常对象：变量名；printStackTrace() : 打印异常堆栈信息  变量名.printStackTrace();  }  printStackTrace()的堆栈跟踪功能显示出程序运行到当前类的执行流程 |



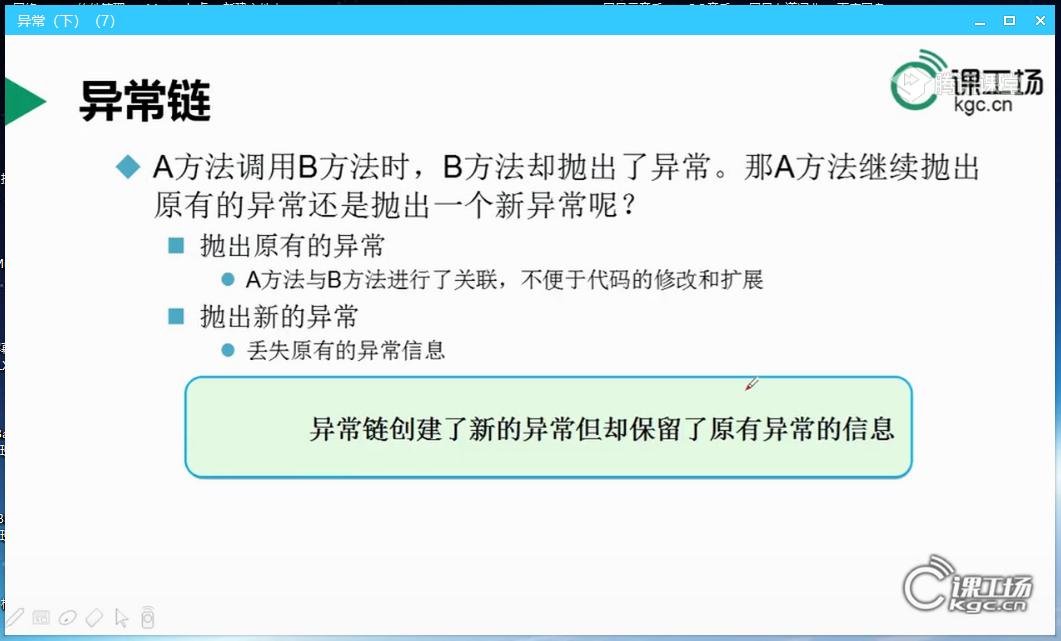
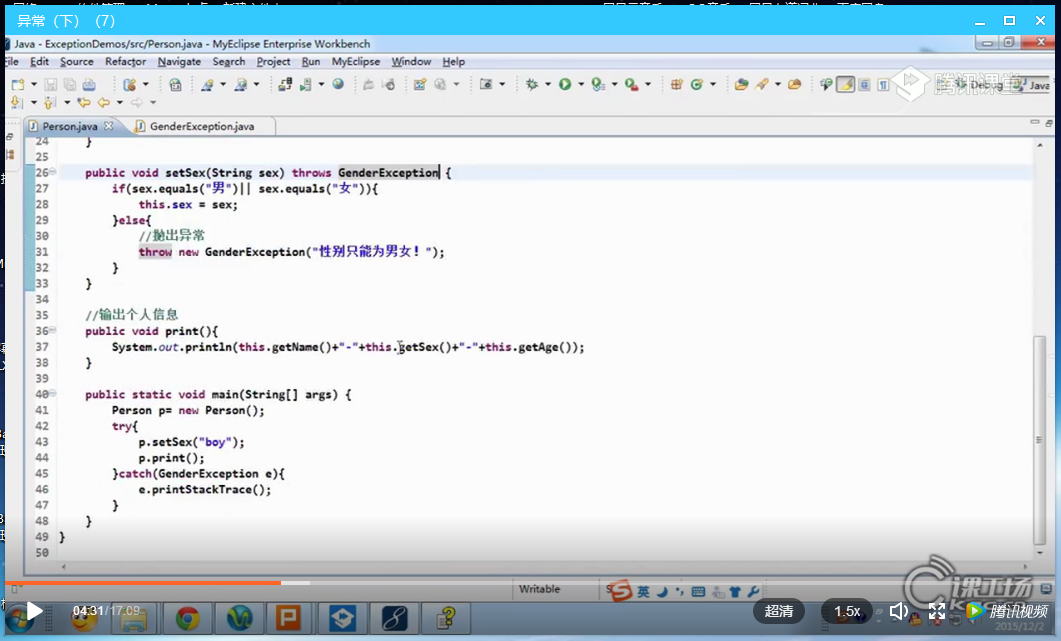
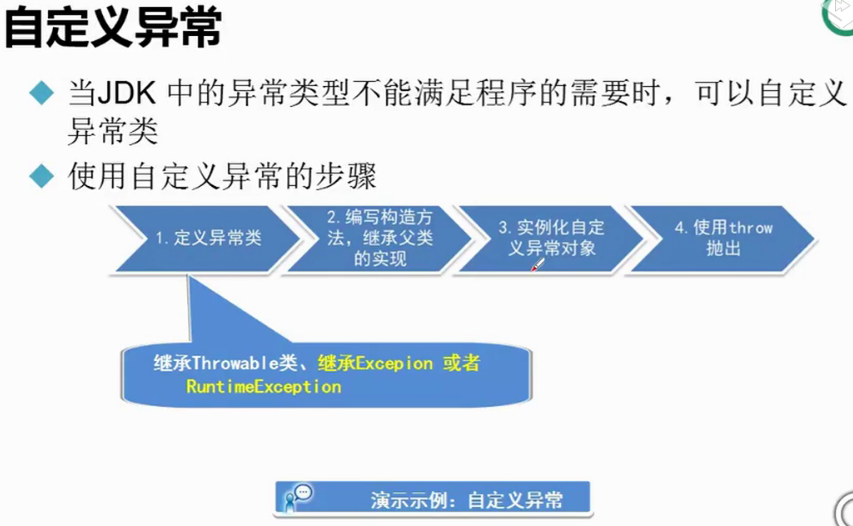
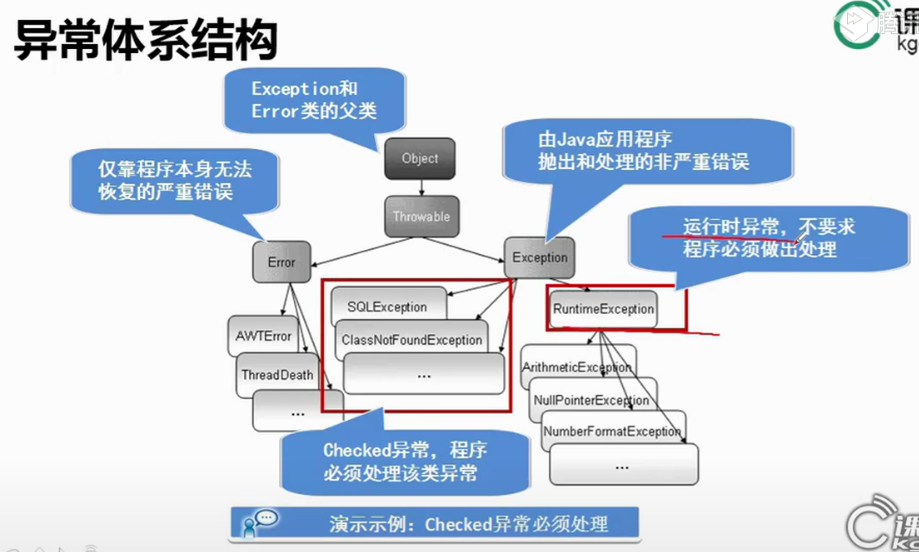
|  |
| --- |
| try{  }catch(异常Exception或者Exception的子类){  }   1. 无异常。try -> catch后面的代码 2. 有异常，而且恰好和catch后面的异常类型匹配：try --> catch里面的代码 --> catch后面的代码   （3）有异常，但是和catch后面的异常类型不匹配：try --> 中断 |

|  |
| --- |
| 多重catch块   * 引发多种类型的异常 * 排列catch语句的顺序：先子类后父类 * 发生异常时按顺序逐个匹配 * 只执行第一个与异常类型匹配的catch语句 |

|  |
| --- |
| 小结：   * try— catch— finally结构中try语句块是必须的，catch、finally语句块均可选，但两者至少出现之一（try不能单独使用） * 面试题：try— catch块中存在return语句，是否还执行finally块，如果执行，说出执行顺序 * try— catch— finally块中，finally块唯一不执行的情况是什么？（exit(1) 直接退出JVM） |

|  |
| --- |
| 声明异常   * 如果在一个方法体重抛出了异常，如何通知调用者？   throws 声明某个方法可能抛出的各种异常，多个异常用逗号隔开   * 方式1：调用者处理异常 * 方式2：调用者继续声明异常 * main()方法声明的异常由Java虚拟机处理 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 抛出异常   * 除了系统自动抛出异常外，有些问题需要程序员自行抛出异常 * 使用throw抛出异常 * throw与throws  |  |  | | --- | --- | | throw | throws | | 生成并抛出异常 | 声明方法内抛出了异常 | | 位于方法体内部，可作为单独语句使用 | 必须跟在方法参数列表后面，不能单独使用 | | 抛出一个异常对象，且只能是一个 | 声明抛出异常类型，可以跟多个异常 |   public void setSex(String sex) throws Exception {  if(sex.equals(‘男’) || sex.equals(‘女’)){  this.sex = sex;  }else{  //抛出异常  throw new Exception(“性别只能为男/女！”);  }  } |



文件处理

|  |
| --- |
| 使用Try-with-resources自动关闭资源  程序员经常会忘记关闭文件。JDK 7 提供了下面的新的try-with-resources语法来自动关闭文件。  try (声明和创建资源) {  使用资源来处理文件;  } |

|  |
| --- |
| 以下情形，finally块将不会被执行：   1. finally块中发生了异常 2. 程序所在线程死亡 3. 在前面的代码中用了System.exit(0) 4. 关闭CPU |

集合 简称集，是用来存储多个元素的容器

集合和数组的区别

* 元素类型

集合：引用类型（存储基本类型时自动装箱）

数组：基本类型、引用类型

* 元素个数

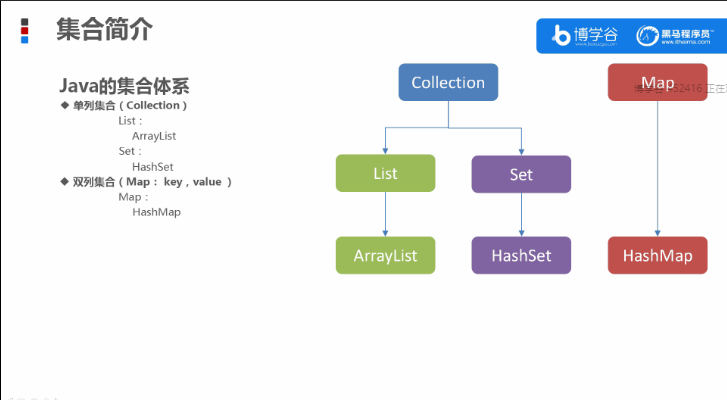
集合：不固定，可任扩展

数组：固定，不能改变容量

* 集合的好处

不受容器大小限制，可以随时添加、删除元素

提供了大量操作的方法（判断、获取等）



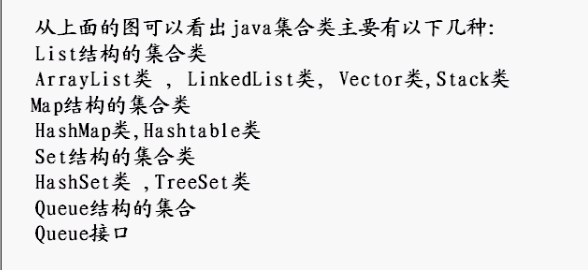
List集合特点

特点：可重复、有序（存取顺序相同）

应用：List +变量名+ = new ArrayList();

集合的步骤：

1. 创建集合对象
2. 创建元素对象
3. 将元素对象添加到集合中
4. 遍历集合



|  |  |
| --- | --- |
| 集合的步骤：   1. 创建集合对象   List list = new ArrayList()；   1. 创建元素对象   Student s1 = new Student(“张三”,24);  Student s2 = new Student(“张三”,24);  Student s3 = new Student(“李四”,26);  Student s4 = new Student(“王五”,22);   1. 将元素对象添加到集合中   list.add(s1);  list.add(s2);  list.add(s3);  list.add(s4);  //直接打印集合  System.out.println(list);  //获取索引为2的元素  Object obj = list.get(2);  System.out.println(obj);  //获取集合中元素的个数  list.size(); | 显示：  [张三24, 张三24, 李四26, 王五22]﻿  李四26 |
| 1. 遍历集合   for (int i = 0; i < list.size(); i++){  //i表示的就是集合中每个元素的索引  //获取元素  Object object = list.get(i);  System.out.println(object);  } | 显示：  张三24  张三24  李四26  王五22 |

增强for格式：

for（元素的数据类型 变量名：要遍历的数组或者集合对象）{

//循环体，变量也就是元素

}

快捷方式： iter --> 回车

注意： 增强for的底层 依赖的是迭代器（Iterator）;

大白话解释：增强for就是迭代器的简写形式

|  |
| --- |
| 迭代器：概述：  对过程的重复，称为迭代。  迭代器是遍历Collection集合的通用方式  迭代器的常用方法  E next（）；返回迭代的下一个元素对象  Boolean hasNext（）；如果仍有元素可以迭代，则返回true  注意：  列表迭代器是List体系独有的遍历方式，可以在对集合遍历的同事进行添加、删除等操作。  但是必须通过调用列表迭代器的方法来实现。  使用步骤：   1. 根据集合对象获取器对象的迭代器对象 2. 判断迭代器中是否有元素 3. 如果有就获取元素   总结：  普通的迭代器在遍历集合的同时不能添加或者删除元素，否则会报：并发修改异常  列表迭代器在遍历集合的同时可以修改集合中的元素（添加，删除等），必须使用列表迭代器中的方法 |

迭代器的具体步骤：

|  |  |  |
| --- | --- | --- |
| 1. 创建集合对象   List list = new ArrayList();   1. 创建元素对象 2. 将元素对象添加到集合对象中   list.add(“a”);  list.add(“b”);  list.add(“c”); | | 显示：  a  b  c  [a,b,java,c] |
| 1. 遍历集合   //迭代器的用法   1. 根据集合对象获取器对象的迭代器对象   Iterator it = list.iterator();   1. 判断迭代器中是否有元素   while(it.hasNext()){ //如果迭代器中有元素，就一直迭代  hasNext():判断是否还有下一个元素,若有,则返回true.  String s = (String)it.next();  if(“b”.equals(s)){  //list.add(“java”);  //这样写行，必须调用列表迭代器的方法来实现  it.add(“java”);  }  System.out.println(s);  } | 4.遍历集合  //迭代器的用法  1.根据集合对象获取器对象的迭代器对象  ListIterator it = list.ListIterator();  2.判断迭代器中是否有元素  while(it.hasNext()){ //如果迭代器中有元素，就一直迭代  3.如果有就获取元素  String s = (String)it.next();  if(“b”.equals(s)){  //list.add(“java”);  //这样写行，必须调用列表迭代器的方法来实现  it.add(“java”);  }  System.out.println(s);  }  System.out.println(-------);  System.out.println(list); |

[LinkedList类](file://E:\\Technological%20Learning\\Java\\JDK_api\\JDk-12.0.1_doc-all\\docs\\api\\java.base\\java\\util\\LinkedList.html)

|  |  |  |
| --- | --- | --- |
| 方法名 | 描述 | 备注 |
| add(int index,E e) | 将指定元素插入此列表的指定位置,若不明确位置,则追加到列表的末尾 |  |
| [addAll](" \l "addAll(int,java.util.Collection))(int index,[Collection](Collection.html" \o "interface in java.util)<? extends [E](LinkedList.html" \o "type parameter in LinkedList)> c) | 从指定位置开始,将指定集合中的所有元素插入此列表若不明确位置,则追加到列表的末尾 |  |
| addFirst() | 在此列表的开头插入指定的元素 |  |
| addLast() | 将指定的元素追加到此列表的末尾 |  |
| clear() | 从此列表中删除所有元素 |  |
| clone() | 返回此LinkedList的浅表副本 |  |
| element() | 获取但不移除此列表的头(第一个元素) |  |
| get(int index) | 返回此列表中指定位置的元素 |  |
| getFirst() | 返回此列表中第一个元素 |  |
| getLast() | 返回此列表中最后一个元素 |  |
| indexOf(Object o) | 返回此列表中首次出现的指定元素的索引,如果此列表中不包含该元素,则返回-1 | o : 要搜索的元素 |
| lastIndexOf(Object o) | 返回此列表中指定元素最后一个出现的索引,如果此列表不包含该元素,则返回-1 | o : 要搜索的元素 |
| offer(E e) | 将指定的元素添加为此列表的尾部(最后一个元素) | e : 要添加的元素 |
| offerFirst(E e) | 在指定列表的前面插入指定的元素 | e : 要添加的元素 |
| offerLast(E e) | 在此列表的末尾插入指定的元素 | e : 要添加的元素 |
| peek() | 获取但不移除此列表的头(第一个元素) |  |
| peekFirst() | 获取但不移除此列表的第一个元素;如果此列表为空,则返回null |  |
| peekLast() | 获取但不移除此列表的最后一个元素;如果此列表为空 ,则返回null |  |
| poll() | 获取并移除此列表的头(第一个元素) |  |
| pollFirst() | 获取并移除此列表的第一个元素;如果此列表为空,则返回 null |  |
| pollLast() | 获取并移除此列表的最后一个元素;如果此列表为空,则返回null |  |
| pop() | 从此列表所有表示的堆栈出弹出一个元素 |  |
| push() | 将元素推入此列表所表示的堆栈 |  |
| remove() | 获取并移除此列表的头(第一个元素) |  |
| remove(int index) | 移除此列表中指定位置处的元素 |  |
| remove(Object o) | 从此列表中移除首次出现的指定的元素(如果存在) |  |
| removeFirst() | 移除并返回此列表的第一个元素 |  |
| removeFirstOccurrence(Object o) | 从此列表中移除第一个出现的指定元素(从头部到尾部遍历列表时) |  |
| removeLast() | 移除并返回此列表的最后一个元素 |  |
| removeLastOccurrence(Object o) | 从此列表中移除最后一次出现的指定的元素(从头部到尾部遍历列表时) |  |
| set(int index, E element) | 将此列表中指定位置的元素替换 为指定的元素 |  |
| size() | 返回此列表的元素数 |  |
| toArray() | 返回以适当顺序(从第一个元素到最后一个元素)包含此列表中所有元素的数组.  此方法充当基于的API与基于collection的API之间的桥梁. | 由于此列表不维护对返回数组的任何引用,因而它将是”安全的”(换句话说,此方法必须分配一个新数组.)因此,调用者可以随意修改返回的数组. |
| toArray(T[] a) | 返回以适当顺序(从第一个元素到最后一个元素)包含此列表中所有元素的数组;返回数组的运行时类型为指定数组的类型.如果指定数组能容纳列表,则在其中返回该列表哦.否则,分配具有指定数组的运行时类型和此列表大小的新数组.  如果指定数组能容纳列表,并有剩余空间(即数组比列表元素多),则紧跟在列表末尾的数组元素会被设置为null.(只有在调用者知道列表不包含任何null元素时,才可以使用此方法来确定类表的长度.) | 像toArray()方法一样,此方法充当基于数组的API与基于collection的API之间的桥梁.更进一步说,此方法允许对输出数组的运行是类型 上进行精确控制,在某些情况下,可以用来节省分配开销.  假设x是只包含字符串的一个已知列表.以下代码可以将该列表转储到一个新分配的Sting数组:  String[] y = x.toArray(new Sting[0])  toArray(new Object[0]和toArray()在功能上是相同的. |

Vector类

|  |  |
| --- | --- |
| **构造方法** | |
| **[Vector](../../java/util/Vector.html" \l "Vector())**()            构造一个空向量，使其内部数据数组的大小为 10，其标准容量增量为零。 |  |
| **[Vector](../../java/util/Vector.html" \l "Vector(java.util.Collection))**([Collection](../../java/util/Collection.html" \o "java.util 中的接口)<? extends [E](../../java/util/Vector.html" \o "Vector 中的类型参数)> c)            构造一个包含指定 collection 中的元素的向量，这些元素按其 collection 的迭代器返回元素的顺序排列。 |  |
| **[Vector](../../java/util/Vector.html" \l "Vector(int))**(int initialCapacity)            使用指定的初始容量和等于零的容量增量构造一个空向量。 |  |
| **[Vector](../../java/util/Vector.html" \l "Vector(int, int))**(int initialCapacity, int capacityIncrement)            使用指定的初始容量和容量增量构造一个空的向量。 |  |

|  |  |
| --- | --- |
| **方法** | |
| boolean | **[add](../../java/util/Vector.html" \l "add(E))**([E](../../java/util/Vector.html" \o "Vector 中的类型参数) e)            将指定元素添加到此向量的末尾。 |
| void | **[add](../../java/util/Vector.html" \l "add(int, E))**(int index, [E](../../java/util/Vector.html" \o "Vector 中的类型参数) element)            在此向量的指定位置插入指定的元素。 |
| boolean | **[addAll](../../java/util/Vector.html" \l "addAll(java.util.Collection))**([Collection](../../java/util/Collection.html" \o "java.util 中的接口)<? extends [E](../../java/util/Vector.html" \o "Vector 中的类型参数)> c)            将指定 Collection 中的所有元素添加到此向量的末尾，按照指定 collection 的迭代器所返回的顺序添加这些元素。 |
| boolean | **[addAll](../../java/util/Vector.html" \l "addAll(int, java.util.Collection))**(int index, [Collection](../../java/util/Collection.html" \o "java.util 中的接口)<? extends [E](../../java/util/Vector.html" \o "Vector 中的类型参数)> c)            在指定位置将指定 Collection 中的所有元素插入到此向量中。 |
| void | **[addElement](../../java/util/Vector.html" \l "addElement(E))**([E](../../java/util/Vector.html" \o "Vector 中的类型参数) obj)            将指定的组件添加到此向量的末尾，将其大小增加 1。 |
| int | **[capacity](../../java/util/Vector.html" \l "capacity())**()            返回此向量的当前容量。 |
| void | **[clear](../../java/util/Vector.html" \l "clear())**()            从此向量中移除所有元素。 |
| [Object](../../java/lang/Object.html" \o "java.lang 中的类) | **[clone](../../java/util/Vector.html" \l "clone())**()            返回向量的一个副本。 |
| boolean | **[contains](../../java/util/Vector.html" \l "contains(java.lang.Object))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o)            如果此向量包含指定的元素，则返回 true。 |
| boolean | **[containsAll](../../java/util/Vector.html" \l "containsAll(java.util.Collection))**([Collection](../../java/util/Collection.html" \o "java.util 中的接口)<?> c)            如果此向量包含指定 Collection 中的所有元素，则返回 true。 |
| void | **[copyInto](../../java/util/Vector.html" \l "copyInto(java.lang.Object[]))**([Object](../../java/lang/Object.html" \o "java.lang 中的类)[] anArray)            将此向量的组件复制到指定的数组中。 |
| [E](../../java/util/Vector.html" \o "Vector 中的类型参数) | **[elementAt](../../java/util/Vector.html" \l "elementAt(int))**(int index)            返回指定索引处的组件。 |
| [Enumeration](../../java/util/Enumeration.html" \o "java.util 中的接口)<[E](../../java/util/Vector.html" \o "Vector 中的类型参数)> | **[elements](../../java/util/Vector.html" \l "elements())**()            返回此向量的组件的枚举。 |
| void | **[ensureCapacity](../../java/util/Vector.html" \l "ensureCapacity(int))**(int minCapacity)            增加此向量的容量（如有必要），以确保其至少能够保存最小容量参数指定的组件数。 |
| boolean | **[equals](../../java/util/Vector.html" \l "equals(java.lang.Object))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o)            比较指定对象与此向量的相等性。 |
| [E](../../java/util/Vector.html" \o "Vector 中的类型参数) | **[firstElement](../../java/util/Vector.html" \l "firstElement())**()            返回此向量的第一个组件（位于索引 0) 处的项）。 |
| [E](../../java/util/Vector.html" \o "Vector 中的类型参数) | **[get](../../java/util/Vector.html" \l "get(int))**(int index)            返回向量中指定位置的元素。 |
| int | **[hashCode](../../java/util/Vector.html" \l "hashCode())**()            返回此向量的哈希码值。 |
| int | **[indexOf](../../java/util/Vector.html" \l "indexOf(java.lang.Object))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o)            返回此向量中第一次出现的指定元素的索引，如果此向量不包含该元素，则返回 -1。 |
| int | **[indexOf](../../java/util/Vector.html" \l "indexOf(java.lang.Object, int))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o, int index)            返回此向量中第一次出现的指定元素的索引，从 index 处正向搜索，如果未找到该元素，则返回 -1。 |
| void | **[insertElementAt](../../java/util/Vector.html" \l "insertElementAt(E, int))**([E](../../java/util/Vector.html" \o "Vector 中的类型参数) obj, int index)            将指定对象作为此向量中的组件插入到指定的 index 处。 |
| boolean | **[isEmpty](../../java/util/Vector.html" \l "isEmpty())**()            测试此向量是否不包含组件。 |
| [E](../../java/util/Vector.html" \o "Vector 中的类型参数) | **[lastElement](../../java/util/Vector.html" \l "lastElement())**()            返回此向量的最后一个组件。 |
| int | **[lastIndexOf](../../java/util/Vector.html" \l "lastIndexOf(java.lang.Object))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o)            返回此向量中最后一次出现的指定元素的索引；如果此向量不包含该元素，则返回 -1。 |
| int | **[lastIndexOf](../../java/util/Vector.html" \l "lastIndexOf(java.lang.Object, int))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o, int index)            返回此向量中最后一次出现的指定元素的索引，从 index 处逆向搜索，如果未找到该元素，则返回 -1。 |
| [E](../../java/util/Vector.html" \o "Vector 中的类型参数) | **[remove](../../java/util/Vector.html" \l "remove(int))**(int index)            移除此向量中指定位置的元素。 |
| boolean | **[remove](../../java/util/Vector.html" \l "remove(java.lang.Object))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o)            移除此向量中指定元素的第一个匹配项，如果向量不包含该元素，则元素保持不变。 |
| boolean | **[removeAll](../../java/util/Vector.html" \l "removeAll(java.util.Collection))**([Collection](../../java/util/Collection.html" \o "java.util 中的接口)<?> c)            从此向量中移除包含在指定 Collection 中的所有元素。 |
| void | **[removeAllElements](../../java/util/Vector.html" \l "removeAllElements())**()            从此向量中移除全部组件，并将其大小设置为零。 |
| boolean | **[removeElement](../../java/util/Vector.html" \l "removeElement(java.lang.Object))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) obj)            从此向量中移除变量的第一个（索引最小的）匹配项。 |
| void | **[removeElementAt](../../java/util/Vector.html" \l "removeElementAt(int))**(int index)            删除指定索引处的组件。 |
| protected  void | **[removeRange](../../java/util/Vector.html" \l "removeRange(int, int))**(int fromIndex, int toIndex)            从此 List 中移除其索引位于 fromIndex（包括）与 toIndex（不包括）之间的所有元素。 |
| boolean | **[retainAll](../../java/util/Vector.html" \l "retainAll(java.util.Collection))**([Collection](../../java/util/Collection.html" \o "java.util 中的接口)<?> c)            在此向量中仅保留包含在指定 Collection 中的元素。 |
| [E](../../java/util/Vector.html" \o "Vector 中的类型参数) | **[set](../../java/util/Vector.html" \l "set(int, E))**(int index, [E](../../java/util/Vector.html" \o "Vector 中的类型参数) element)            用指定的元素替换此向量中指定位置处的元素。 |
| void | **[setElementAt](../../java/util/Vector.html" \l "setElementAt(E, int))**([E](../../java/util/Vector.html" \o "Vector 中的类型参数) obj, int index)            将此向量指定 index 处的组件设置为指定的对象。 |
| void | **[setSize](../../java/util/Vector.html" \l "setSize(int))**(int newSize)            设置此向量的大小。 |
| int | **[size](../../java/util/Vector.html" \l "size())**()            返回此向量中的组件数。 |
| [List](../../java/util/List.html" \o "java.util 中的接口)<[E](../../java/util/Vector.html" \o "Vector 中的类型参数)> | **[subList](../../java/util/Vector.html" \l "subList(int, int))**(int fromIndex, int toIndex)            返回此 List 的部分视图，元素范围为从 fromIndex（包括）到 toIndex（不包括）。 |
| [Object](../../java/lang/Object.html" \o "java.lang 中的类)[] | **[toArray](../../java/util/Vector.html" \l "toArray())**()            返回一个数组，包含此向量中以恰当顺序存放的所有元素。 |
| |  | | --- | | <T> T[] | | **[toArray](../../java/util/Vector.html" \l "toArray(T[]))**(T[] a)            返回一个数组，包含此向量中以恰当顺序存放的所有元素；返回数组的运行时类型为指定数组的类型。 |
| [String](../../java/lang/String.html" \o "java.lang 中的类) | **[toString](../../java/util/Vector.html" \l "toString())**()            返回此向量的字符串表示形式，其中包含每个元素的 String 表示形式。 |
| void | **[trimToSize](../../java/util/Vector.html" \l "trimToSize())**()            对此向量的容量进行微调，使其等于向量的当前大小。 |

Stack类

构造方法:

Stack()创建一个空堆栈

|  |  |  |
| --- | --- | --- |
| **方法** | | |
| boolean | **[empty](../../java/util/Stack.html" \l "empty())**()          测试堆栈是否为空。 | **返回 : 当且仅当堆栈中不含任何项时返回true;否则返回false** |
| [E](../../java/util/Stack.html" \o "Stack 中的类型参数) | **[peek](../../java/util/Stack.html" \l "peek())**()          查看堆栈顶部的对象，但不从堆栈中移除它。 | **返回 : 堆栈顶部的对象(Vector对象中的最后一项)** |
| [E](../../java/util/Stack.html" \o "Stack 中的类型参数) | **[pop](../../java/util/Stack.html" \l "pop())**()          移除堆栈顶部的对象，并作为此函数的值返回该对象。 | **返回 : 堆栈顶部的对象(Vector对象中的最后一项)** |
| [E](../../java/util/Stack.html" \o "Stack 中的类型参数) | **[push](../../java/util/Stack.html" \l "push(E))**([E](../../java/util/Stack.html" \o "Stack 中的类型参数) item)          把项压入堆栈顶部。 | **item : 压入堆栈的项**  **返回: item参数** |
| int | **[search](../../java/util/Stack.html" \l "search(java.lang.Object))**([Object](../../java/lang/Object.html" \o "java.lang 中的类) o)          返回对象在堆栈中的位置，以 1 为基数。 | **o : 目标对象**  **返回 : 对象到堆栈顶部的位置,以1为基数,返回值-1表示此对象不再堆栈中.** |

|  |
| --- |
| ArrayList和Vector的区别  ArrayList与Vector都是Java的集合类,都可以用来存放java对象,这是他们的相同点,  异同点:  一、同步性  Vector是同步的,这个类中的一些方法保证了Vector中的对象是线程安全的.而ArrayList则是异步的,因此ArrayList中的对象并不是线程安全的.因为同步的要求会影响执行的效率,所以如果你不需要线程安全的集合那么使用ArrayList,这样可以避免由于同步带来的不必要的性能开销.   1. 数据增长   从内部实现机制来讲ArrayList和Vector都是使用数组(Array)来控制集合中的对象.当你向这两种类型中增加元素的时候,如果元素的数目超出了内部数组目前的长度他们都需要扩展内部数组的长度,Vector缺省情况下自动增长原来的一倍的数组长度,ArrayList是原来的50%,所以最后你获得的这个集合所占的空间总是比你实际需要的要大.所以如果你要在集合中保存大量的数据那么使用Vector有一些优势,因为你可以通过设置集合的初始大小来避免不必要的资源开销. |

泛型 即泛指任意类型，又叫参数化类型，对具体类型的使用骑到辅助作用，类似于方法的参数

集合类泛型的解释

表示该集合中存放指定类型的元素

给List集合加上泛型String

List<String> list = new Arraylist<>();

泛型的好处

类型安全 避免了类型转换

|  |  |
| --- | --- |
| //不使用泛型的集合  创建集合对象  List list1 = new ArrayList();  创建元素对象  将元素对象添加到集合对象中  list1.add(“a”);  list1.add(“b”);  list1.add(“c”);  //ist1.add(10); //编译无错，但运行时出错  遍历集合  for（Object obj : list1）{  String s = (String)obj;  System.out.println(s);  }  //使用泛型  创建集合对象  List<String> list2 = new ArrayList</\*String （可省略）\*/>();  创建元素对象  将元素对象添加到集合对象中  list2.add(“abc”);  list2.add(“bcd”);  list2.add(“cde”);  //list2.add(10);//编译出错  遍历集合  for（String s : list2）{  System.out.println(s);  } | 显示：  a  b  c  显示：  abc  bcd  cde |

总结：

泛型一般只和集合类相结合使用；泛型是JDK5的新特性，但是从JDK7开始，后边的泛型可以不用谢具体的数据类型了（菱形泛型）

泛型的好处是在编译的时候检查类型安全,并且所有的强制转换都是自动和隐式的,提高代码的重用率。

泛型的优点：

1. 类型安全；
2. 向后兼容；
3. 层次清晰；

4、性能较高，用泛型Java编写的代码可以为Java编译器和虚拟机带来更多的类型信息，这些信息对java程序做进一步优化提供条件；

Collections简介针对集合进行的工具类

//创建集合对象

List<Integer> list = new ArrayList<>();

//往集合中添加数据

list.add(1);

list.add(3);

list.add(3);

list.add(5);

list.add(2);

list.add(2);

list.add(4);

System.out.println(list);//显示 [1,3,3,5,2,2,4]

成员方法

sort（List<T>）根据元素的自然顺序，讲指定列表按升序排序

Collections.sort(list);//现在为： [1,2,2,3,3,4,5]

max（Collections<T>）返回集合的祖达元素

Integer max = new Collections.max(list); //max值为5

reverse（List<T>）反转List集合元素[反向]

Collections.reverse(list);//现在为： [4,2,2,5,3,3,1]

shuffle（List<T>）使用默认的随机源随机置换指定的列表

Collections.shuffle(list); //;相当于洗牌

Set集合的特点 不可重复，无序

应用：Set<T> set = new HashSet<>();

步骤： 创建集合对象

Set<Student> set = new HashSet<>();

创建元素对象

Student s1 = new Student(“张三”,24);

Student s2 = new Student(“张三”,24);

Student s3 = new Student(“李四”,26);

Student s4 = new Stud0ent(“王五”,22);

将集合对象添加到元素对象中

list.add(s1);

list.add(s2);

list.add(s3);

list.add(s4);

遍历集合

System.out.println(set);// 显示：[张三24, 张三24, 李四26, 王五22]

为什么Set集合没有“去重”？

因为Set集合保证元素的唯一性依赖：equals()和hashCode()两个方法

你没有在Student类中重写这两个方法 。默认调用的是Object类中的两个方法

而Object类中的equals()方法默认比较的是地址值是否相同

解决方案：在Student类中重写equals()和hashCode()方法

|  |
| --- |
| Set和List区别：   1. Set接口实例存储的是无序的，不重复的数据。List接口实例存储的是有序的，可以重复的元素。 2. Set检索效率低下，删除和插入效率高，插入和删除不会引起元素位置改变<实现类有HashSet,TreeSet>。 3. List和数组类似，可以动态增长，根据实际存储的数据的长度自动增长List的长度。查找元素效率高，插入删除效率低，因为会引起其他元素位置改变<实现类有ArrayList,LinkedList,Vector> |

Map集合的特点双列集合，元素有键值对（Entry）构成：key--value；key不可以重复，value可以重复

|  |
| --- |
| 应用： Map<T1,T2> map = new HashMap<>();  T1：表示键的数据类型 T2：表示值的数据类型 |
| 需求：向Map集合中添加三个元素，并遍历打印 |
| 分析：A：向map集合中添加到元素的方法为：put()  B:遍历集合的方式：  获取所有的key：keySet()  遍历keySet，通过key获取value：get()  C:遍历keySet的方法：iterator() |
| 步骤：   1. 创建集合对象：   Map<Integer,Student> map = new HashMap<>();   1. 分别创建三个Student对象 2. 使用put方法将Student对象添加到集合中 3. 获取所有的key，并使用迭代器遍历 4. 通过key分别获取对应的value并打印 |
| 成员方法：  V put(K key，V value); 添加元素（键值对的形式）；  元素第一次添加，返回null；重复添加会用新值覆盖旧值，并返回旧值  V get(Object key); 根据键获取其对应的值  Set<K> keySet(); 获取所有键的集合 |
| 遍历步骤：  获取所有键的集合 keySet()  遍历所有的键，获取到每一个键 迭代器，增强for  根据键，获取指定的值 get() |
| 创建集合对象：  Map<Integer,Student> map = new HashMap<>();  创建元素对象  Student s1 = new Student(“张三”,23);  Student s2 = new Student(“李四”,25);  Student s3 = new Student(“张三”,23);  将元素对象添加到集合对象中  Student stu1 = map.put(1,s1); //stu1值： null  System.out.println(map); // map 值：{1=name:张三 age:23}  --------------------------------------------------------  Student stu1 = map.put(1,s1); //stu1值： null  Student stu2 = map.put(1,s2); //stu2值： name:张三 age:23  System.out.println(map); // map 值：{1=name:李四 age:25} |
| 遍历集合  //4.1获取所有键的集合 keySet()  Set<Integer> keys = map.keySet();  //4.2遍历所有的键 ，获取到每一个键、 迭代器，增强for  //获取迭代器对象  Iterator<Integer> it = keys.iterator();  **while**(it.hasNext()) {  //如果迭代器中有数据，就获取  Integer key = it.next();  //4.3根据键，获取指定的值 get()  //根据键获取键  Student value = map.get(key);  System.***out***.println("key:" + key + "\tvalue:" + value);  }  System.***out***.println("---------------------------------");  //通过增强for实现  **for**(Integer key : keys) {  Student value = map.get(key);  System.***out***.println("key:" + key + "\tvalue:" + value);  } |

|  |
| --- |
| HashMap和Hashtable的区别  HashMap与Hashtable都是java的集合类,都可以用来存放java对象,这是他们的相同点  异同点:   1. 历史原因   Hashtable是基于陈旧的Dictionary类，HashMap是Java1.2引进的Map接口的一个实现。   1. 同步性   Hashtable是同步的.这个类中的一些保证了Hashtable中的对象是线程安全的.而HashMap则是异步的,因此HashMap中的对象并不是线程安全的.因为同步的要去也会影响执行的效率,所以如果你不需要线程安全的集合那么使用HashMap,这样可以避免由于同步带来的不必要的内存开销,从而提高效率。   1. 值   HashMap可以让你将空值作为一个表的条目的key或value，但是Hashtable是不能放入空值的(null) |

|  |
| --- |
| (1)如果要求线程安全,使用vector和Hashtable  (2)如果不要求线程安全,应使用ArrayList,LinkedList,HashMap  (3)如果要求键-值对,则使用HashMap,Hashtable  (4)如果数据量很大,又要线程安全考虑Vector |

I/O流

概述：I(Input，输入)/O(Output，输出)流，是Java中用来传输数据方式

System.in(标准的输入)：通常代表键盘输入

System.out(标准的输出)：通常写往显示器

System.err(标准的错误输出)：通常写往显示器

colse()方法是关闭流

划分：

按照流向分：输入流：读数据；输出流：写数据

按照操作分：

字节流：以字节为单位来操作数据

InputStream：字节输入流的顶层抽象类

FileInputStream： 普通的字节输入流

BufferedInputStream: 高效的字节输入流（也叫：字节缓冲输入流）

OutputStream：字节输出流的顶层抽象类

FileOutputStream： 普通的字节输出流

BufferedOutputStream: 高效的字节输出流（也叫：字节缓冲输出流）

字符流：以字符为单位来操作数据

Reader：字符输入流的顶层抽象类

FileReader： 普通的字符输入流

BufferedReader: 高效的字符输入流（也叫：字符缓冲输入流）

Writer：字符输出流的顶层抽象类

FileWriter： 普通的字符输出流

BufferedWriter: 高效的字符输出流（也叫：字符缓冲输出流）

File类：

概述：文件，文件夹，一个File对象代表磁盘上的某个文件或文件夹，也就是用来操作问价（夹）路径的

构造方法：

File(String 文件（夹）路径) 根据给定的字符串路径创建其对应File对象

File(String 父目录路径，String 子目录路径) 根据给定的字符串形式父目录对象和子文件（夹）名创建File对象

File(String 父目录路径，String 子目录路径) 根据给定的父目录对象和子文件（夹）名创建File对象

成员方法：

创建功能：如果不存在就创建，返回true，否则就不创建返回false.

createNewFile();创建TXT文件

mkdir(); 创建单级目录

mkdirs(); 创建单/多级目录

判断功能： isDirectory(); 判断File对象是否为文件夹

isFile();判断File对象是否为文件

exists(); 判断File对象是否存在

需求：将D:\abc\1.txt封装成File对象

方式一：根据给定的字符串路径创建其对应File对象

File file1 = new File(“D:\\abc\\1.txt”);

或者 File file1 = new File(“D:/abc/1.txt”);

方式二：根据给定的字符串形式父目录对象和子文件（夹）名创建File对象

File file2 = new File(“D:/abc/”,“1.txt”);

方式三：根据给定的父目录对象和子文件（夹）名创建File对象

File file3 = new File(“D:/abc/”);

File file4 = new File(file3,“/1.txt”);

File类获取功能：

getAbsolutePath() : 获取文件绝对路径

getPath() : 获取文件相对路径

getName() : 获取文件名

list() : 获取指定目录下所有文件（夹）名称数组

listFiles() : 获取指定目录下所有文件（夹）File数组

已知文件路径为D:\abc\lib\1.txt

则：相对路径为：lib\1.txt

则：绝对路径为：D:\abc\lib\1.txt

[File文件操作类](http://c.biancheng.net/view/1133.html)

|  |  |
| --- | --- |
| 方法名称 | 说明 |
| boolean canRead() | 测试应用程序是否能从指定的文件中进行读取 |
| boolean canWrite() | 测试应用程序是否能写当前文件 |
| boolean delete() | 删除当前对象指定的文件 . |
| boolean exists() | 测试当前 File 是否存在 |
| boolean createNewFile() | 测试该文件是否存在，若已存在，则不创建，返回false，若没有，则返回true |
| String getAbsolutePath() | 返回由该对象表示的文件的绝对路径名 |
| String getName() | 返回表示当前对象的文件名 |
| String getParent() | 返回当前 File 对象路径名的父路径名，如果此名没有父路径则为 null |
| boolean isAbsolute() | 测试当前 File 对象表示的文件是否为一个绝对路径名 |
| boolean isDirectory() | 测试当前 File 对象表示的文件是否为一个路径 |
| boolean isFile() | 测试当前 File 对象表示的文件是否为一个“普通”文件 |
| long lastModified() | 返回当前 File 对象表示的文件最后修改的时间 |
| long length() | 返回当前 File 对象表示的文件长度 |
| String[] list() | 返回当前 File 对象指定的路径文件列表 |
| String[] list(FilenameFilter) | 返回当前 File 对象指定的目录中满足指定过滤器的文件列表 |
| boolean mkdir() | 创建一个目录，它的路径名由当前 File 对象指定 |
| boolean mkdirs() | 创建一个目录，它的路径名由当前 File 对象指定 |
| boolean renameTo(File) | 将当前 File 对象指定的文件更名为给定参数 File 指定的路径名 |

# [RandomAccessFile类：随机访问文件类](http://c.biancheng.net/view/1137.html)

RandomAccessFile类的常用方法

|  |  |
| --- | --- |
| **方法名及返回值类型** | **说明** |
| boolean readBoolean() | 从文件中读取一个 boolean 值 |
| byte readByte() | 从文件中读取一个带符号位的字节 |
| char readChar() | 从文件中读取一个字符 |
| int readlnt() | 从文件中读取一个带符号位的整数 |
| long readLong() | 从文件中读取一个带符号位的 long 值 |
| String readLine() | 从文件中读取下一行文本 |
| void seek(long pos) | 指定从文件起始位置开始的指针偏移量 |
| void writeBoolean(boolean v) | 以字节的形式向文件中写入一个 boolean 值 |
| void writeByte(int v) | 以单字节的形式向文件中写入一个 byte 值 |
| void writeChar(int v) | 以双字节的形式向文件中写入一个 char 值 |
| void writelnt(int v) | 以4字节的形式向文件中写入一个整数 |
| writeLong(long v) | 以8字节的形式向文件中写入一个 long 值 |
| void writeBytes(String s) | 以字节序列的形式向文件中写入一个字符串 |
| void skipBytes(int n) | 以当前文件指针位置为起始点，跳过 n 字节 |

反射机制

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **访问方法** | **返回值类型** | **说明** |
| 包路径 | getPackage() | Package 对象 | 获取该类的存放路径 |
| 类名称 | getName() | String 对象 | 获取该类的名称 |
| 继承类 | getSuperclass() | Class 对象 | 获取该类继承的类 |
| 实现接口 | getlnterfaces() | Class 型数组 | 获取该类实现的所有接口 |
| 构造方法 | getConstructors() | Constructor 型数组 | 获取所有权限为 public 的构造方法 |
| getDeclaredContxuectors() | Constructor 对象 | 获取当前对象的所有构造方法 |
| 方法 | getMethods() | Methods 型数组 | 获取所有权限为 public 的方法 |
| getDeclaredMethods() | Methods 对象 | 获取当前对象的所有方法 |
| 成员变量 | getFields() | Field 型数组 | 获取所有权限为 public 的成员变量 |
| getDeclareFileds() | Field 对象 | 获取当前对象的所有成员变量 |
| 内部类 | getClasses() | Class 型数组 | 获取所有权限为 public 的内部类 |
| getDeclaredClasses() | Class 型数组 | 获取所有内部类 |
| 内部类的声明类 | getDeclaringClass() | Class 对象 | 如果该类为内部类，则返回它的成员类，否则返回 null |

如表 1 所示，在调用getFields() 和 getMethods()方法时将会依次获取权限为 public 的字段和变量，然后将包含从父类中继承到的成员实量和方法。而通过getDeclareFields() 和 getDeclareMethod()只是获取在本类中定义的成员变量和方法。

线程

一、线程的概念

程序是一段静态的代码,它是应用线程执行的蓝本。

进程是程序的一次动态过程，它对应了从代码加载、执行到执行完毕的一个完整过程。程序可以被多次加载到系统的不同内存区域分别执行，形成不同的进程。

线程是进程内部的一个顺序执行控制流。一个进程在执行过程中，可以产生多个线程同时执行。

每个线程有自己产生、存在和消亡的过程。

Java应用程序总是从主类的main方法开始执行。当JVM加载代码，发现main方法之后，就会启动一个线程，这个线程称作“主线程”，该线程负责执行main方法。在main方法中再创建的线程，就称为主线程中的线程。

如果main方法中没有创建其他的线程，那么当main方法执行完最后一个语句，即main方法返回时JVM就会结束Java应用程序。

如果main方法中又创建了其他线程，那么JVM就要在主线程和其他线程之间轮流切换，保证没个线程都有机会使用CPU资源，main方法即使执行完最后的语句，JVM也不会结束程序，JVM一直要等到主线程中的所有线程丢结束之后，才结束Java应用程序。

在Java程序中，特别饿是网络编程中常利用多线程技术来实现。

Java中的线程模型

对多线程的综合支持是Java语言的一个重要特色，它提供了Thread类来实现多线程。在Java中，线程可分为三部分组成：Code，Data，Virtual CPU(虚拟CPU)

虚拟CPU,封装在Java.lang.Thread类中,它控制着整个线程的运行；

执行代码,传递给Thread类,由Thread类控制顺序执行；

处理的数据,传递给Thread类,是在代码执行中所要的数据。

创建线程：

Java中编程实现多线程有两种方式：

1. 由Thread类派生子类创建线程类；2、通过实现Runnable接口创建线程类。

Thread类本身只是线程的虚拟CPU，线程所执行的代码（或者说线程所要完成的功能）是通过方法run()来完成的,run()方法称为线程体.在一个线程被建立并初始化以后，Java的运行时系统就自动调用run()方法,完成线程要达到的目标。

例1: 在程序中通过继承Thread类创建一个线程子类testThread,通过Thread1主类创建并运行两个线程对象t1和t2。

|  |  |
| --- | --- |
| class testThread extends Thread {  public testThread(String str){  super(str);//调用父类的构造方法为线程对象命名  }  public void run () {  for (int i = 0; i< 3; i++){  System.out.println(getName()+”在运行”);  try{  sleep(1000);//用休眠1000毫秒来区分哪个线程在运行  System.out.println(getName()+”在休眠”);  }catch(InterruptedException e ){}  }  System.out.println(getName()+”已结束”);  }  }  public class Thread1 {  public static void main(String args[]){  testThread t1 = new testThread(“thread1”);  testThread t2 = new testThread(“thread2”);  t1.start();  t2.start();  }  } | 显示:(运行顺序随机)  thread2在运行  thread1在运行  thread2在休眠  thread1在休眠  thread1在运行  thread2在运行  thread2在休眠  thread2在运行  thread1在休眠  thread1在运行  thread1在休眠  thread1已结束  thread2在休眠  thread2已结束 |
| 以实现Runnable接口的类为参数创建对象 |  |
| import java.util.Date;  public class Thread02 {  public static void main(String[] args) {  Thread clocker = new Thread(new ShowSeconds());  clocker.start();  }  }  class ShowSeconds implements Runnable {  private Date now;  @Override  public void run() {  while (true) {  now = new Date();  System.out.println(now);  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  System.out.println("Thread error:" + e);  }  }  }  } | 显示:  Mon Aug 19 17:56:44 CST 2019  Mon Aug 19 17:56:45 CST 2019  Mon Aug 19 17:56:46 CST 2019 |
| 继承 Thread 类的优缺点当一个 run() 方法体现在继承 Thread 类中时，可以用 this 指向实际控制运行的 Thread 实例。因此，代码不需要使用以下控制语句: Thread.currenThread().sleep(); |  |
| 实现 Runnable 接口的优缺点 从面向对象的角度来看，Thread 类是一个虚拟处理机严格的封装，因此只有当处理机模型修改或扩展时，才应该继承该类。由于 Java 技术只允许单一继承，因此如果已经继承了 Thread 类，就不能再继承其他任何类，这会使用户只能采用实现 Runnable 接口的方式创建线程。 |  |

二、线程的调度与控制

同一优先级的线程，采取先进先出的原则由操作系统按时间轮转方式或独占方式来分配线程的执行时间。

线程的优先级用数字来表示，范围从1—10，Thread类有三个关于优先级的静态常量：

MIN\_PRIORITY = 1；MAX\_PRIORITY = 10；NORM\_PRIORITY = 5

对应于一个新线程，系统会遵循以下原则：

新线程讲继承创建它的父线程的优先级；

一般情况下，线程具有普通优先级；

用户可以根据需要，通过setPriority（）方法来修改优先级。

1. yield()[礼让]: 将执行的权利交给其他优先级相同的线程，自己到可运行线程队列的最后等待，若队列空,该方法无效，即该线程继续运行
2. sleep():休眠一段时间(给定一个时间段)
3. join(): 是当前的线程等待,直到t结束位置,线程恢复到可运行状态.有三中调用格式:
4. join(): 如当前线程发出调用t.join()则当前线程讲等待线程t结束在继续执行;
5. join(long millis): 如当前新出发出调用t.join()，则当前线程讲等待线程t结束或最多等待millis毫秒后，在继续执行；
6. join(long millis, int nanos): 如当前线程发出调用t.join()，则当前线程将等待线程结束或最多等待millis毫秒 + nanos纳秒后，再继续执行
7. interrupt() [休眠打断]: 经常用来“吵醒”休眠的线程。当一个线程调用sleep、join、wait方法处于等待状态是，另外一个线程在执行过程中，可以让休眠的线程分别调用interrupt方法“吵醒”（激活）自己，即导致休眠的线程发生InterruptedException异常，从而结束休眠进入可进行状态，重新排队等待CPU资源
8. currentThread(): 该方法是Thread类中的静态方法，可以用类名调用，该方法返回当前正在执行的的线程（应该是线程的名称）
9. isAlive(): 测试线程是否处于正在执行的状态。若线程已被启动或未被终止，则返回true；若线程是新创建或已被终止，则返回false
10. stop(): 强行终止线程
11. suspend(): 将线程暂停执行，若想要恢复线程，必须由其他线程调用resume()来恢复该线程的执行

三、线程同步与线程安全问题

1、多线程并发操作中的问题

1. 线程间的资源互斥共享
2. 线程间的同步

除了要处理多线程间共享数据操作的互斥问题之外，在进行多线程程序设计时，还会遇到另一类问题，这就是如何控制相互交互的线程之间的运行进度，即多线程的同步

2、对象锁及其操作

为了解决所出现的问题，在Java语言中，引入了“对象互斥所”的概念（又称为管程）来实现不同 线程对共享数据操作的同步。“对象互斥锁”阻止多个线程同时访问同一个方法，保护线程共享数据，控制和切换线程的执行，保证内存的一致性。用关键字synchronized来声明一个操作共享数据的方法或一段代码

1. 声明synchronized()方法的一般格式为：(在方法的前面加上 synchronized 关键字)

(public) synchronized returnType methodName（[parameterList]）{/\*方法体\*/}

1. synchronized关键字除了可以放在方法声明中表示正个方法外,还可以放在一段代码的前面限制它的执行,格式如下:

synchronized (this) {/\*some codes\*/}

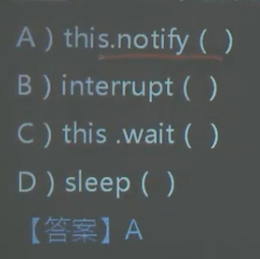
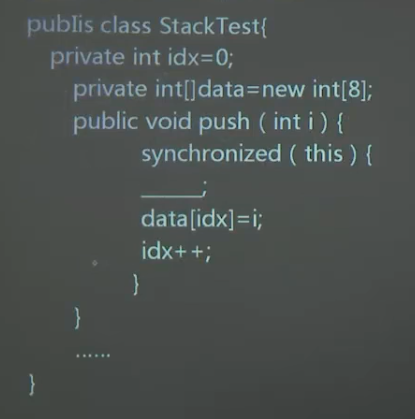
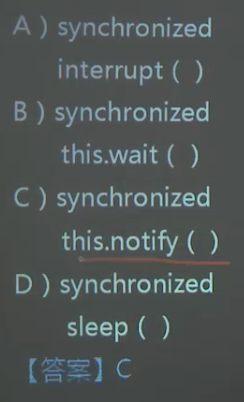
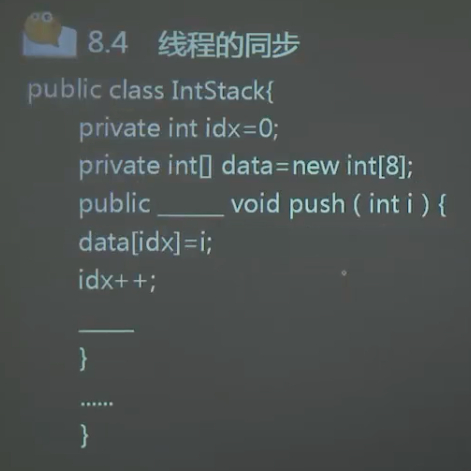
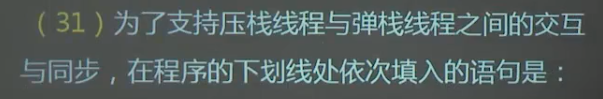
1. 另外，如果synchronized用在类声明中，则表明该类中的所有方法都是synchronized的，该类是同步类，该类中的所有方法都是同步方法

3、线程间的交互： wait() 和 notify() : 只能在被声明为synchronized的方法或代码段中调用

在Java云中还可以用wait（）和notify（）/notifyAll（）方法用来协调线程间的读取关系。wait()、notify（）和notifyAll（）都是Object类中的final方法，被所有的类继承，且不允许重写的方法

wait（）方法的作用让当前线程释放其所持有的“对象互斥锁”，进入wait队列（等待队列）；

notify（）/notifyAll（）方法的作用是唤醒一个或所有正在等待队列中等待的线程，并将它（们）移入等待同一个“对象互斥锁”的队列



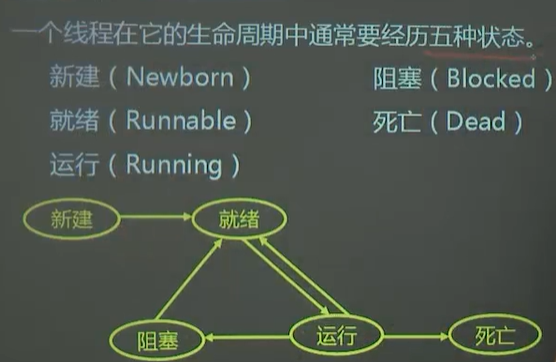
|  |  |
| --- | --- |
| public class Thread3 {  public static void main(String[] args) {  Thread thread01 = new Thread(new TicketSystem());  Thread thread02 = new Thread(new TicketSystem());  Thread thread03 = new Thread(new TicketSystem());  thread01.start();  thread02.start();  thread03.start();  }  }  /\*\*  \* 买票系统  \* \*/  class TicketSystem implements Runnable {  private static int ticketNumber = 100;  @Override  public void run() {  try {  for (int i = 0; i <= 2000; i++) {  if (ticketNumber > 0) {  ticketNumber--;  System.out.println("恭喜你，成功买到票，还剩" + ticketNumber + "张票");  Thread.sleep(50);  }  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } | 显示:  恭喜你，成功买到票，还剩97张票  恭喜你，成功买到票，还剩97张票  恭喜你，成功买到票，还剩97张票  恭喜你，成功买到票，还剩96张票  恭喜你，成功买到票，还剩94张票  恭喜你，成功买到票，还剩94张票  ……  恭喜你，成功买到票，还剩3张票  恭喜你，成功买到票，还剩2张票  恭喜你，成功买到票，还剩2张票  恭喜你，成功买到票，还剩2张票  恭喜你，成功买到票，还剩1张票  恭喜你，成功买到票，还剩1张票  恭喜你，成功买到票，还剩1张票  恭喜你，成功买到票，还剩0张票  恭喜你，成功买到票，还剩0张票 |
| 什么是线程安全问题：当多个线程操作同一共享数据时，引发的一种不正常的结果  如何解决线程安全问题：给公共代码块上锁,通过synchronized()来解决 | |
| class TicketSystem implements Runnable {  private static int ticketNumber = 100;  @Override  public void run() {  try {  for (int i = 0; i <= 2000; i++) {  synchronized (TicketSystem.class) { //上锁[执行完锁起来的部分,再出去]  if (ticketNumber > 0) {  ticketNumber--;  System.out.println("恭喜你，成功买到票，还剩" + ticketNumber + "张票");  Thread.sleep(50);  }  }  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } | |

|  |
| --- |
| String 、 StringBuilder 与 StringBuffer区别  1、String比StringBuilder和StringBuffer 耗内存  若向String对象里添加则会重新开辟一个空间，然后再进行向后添加；  2、StringBuilder与StringBuffer（存在线程安全问题）  StringBuilder；单线程（线程非安全）  StringBuffer：多线程（线程安全） |

四、线程的状态

一个线程在它的生命周期中通常要经历五种状态: 新建（Newborn）、阻塞（Blocked）、就绪（Runnable）、死亡（Dead）、运行（Running）

下面是这五种状态之间转换关系:



新建状态

当创建了一个新的线程对象时,它就处于新建状态，此时它仅仅是一个空的线程对象，系统不为它分配资源。处于这种状态时只能启动start（）或终止stop（）该线程，调用除这两种以外的其他方法都会失败并且会引起非法状态异常IllegalThreadStatException（对于其它状态，若所调用的方法与状态不符，都会引起非法状态异常）。

就绪状态

当线程出于新建状态是，可以调用start()方法来启动它，产生运行这个线程所需的系统资源，安排其运行，并调用线程体run（）方法[,并没有运行]，这样该线程就处于就绪（Runnable）状态。

这一状态并不是运行状态（Running），因为线程也许实际并未真正运行。由于很多计算机都是单处理器的，所以要在同一时刻运行所有的处于就绪状态的线程是不可能的，Java运行系统必须通过调度来保证这些线程共享处理器。

运行状态

运行状态是线程占有CPU并实际运行的状态。此时线程状态的变迁有三种情况：

1、如果线程正常执行结果或应用程序停止运行时，线程将进入终止状态

2、如果当前线程执行了yield（），或者当前线程因调度策略而进入就绪状态

3、如果发生下面几种情况时，线程就进入堵塞状态

（1）线程调用了sleep（）方法、join（）方法时；

（2）线程调用wait（）方法时；

（3）如果线程中使用synchronized来请求对象的锁未获得时；

（4）如果线程中输入输出操作，也将进入阻塞状态，待输入输出操作结束后，线程进入就绪状态

阻塞状态（Blocked）

线程处于运行状态时，当下面几种情况发生，线程就进入阻塞状态：

调用了sleep（）方法或join（）方法。sleep（）方法中设定休眠时间过去后，线程变为可运行的；当调用t.join（）方法的进程t结束或等待时间到时，线程也为可运行的；

调用等待方法wait（）。可被notify（）和notifyAll（）唤醒，或被interrupt（）中断；

使用synchronized来请求对象的锁但未获得时。当线程获得对象锁后，将进入就绪状态。

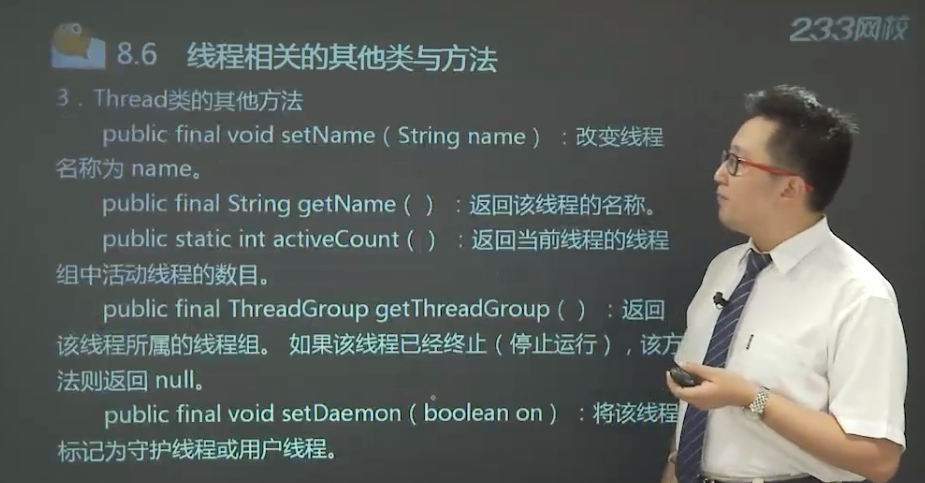
需要注意的是每种方法都仅仅相对的状态才有作用，例如当一个线程休眠并且休眠时间还有结束时，调用其他方法是无效的，并且还会引起非法状态异常

死亡状态（Dead）

线程的终止一般可通过两种方法实现：自然撤销或强行终止

自然撤销是指从线程的run（）方法正常结束退出，如果希望线程正常终止，一般可使用某种手段来使线程中的run（）方法结束执行

调用线程的实例方法stop（）则可以强制停止当前线程。既可以在其他线程中调用该线程的stop（）方法来终止，也可以由线程自己调用stop（）方法，自我终止。但这个方法已建议不再使用，应当避免使用。



GUI图形用户界面

Layout（布局管理器）

| **所属类包** | **布局管理器名称** | **说明** |
| --- | --- | --- |
| Java.awt | FlowLayout（流式布局） | 组件按照加入的先后顺序按照设置的对齐方式从左向右排列，一行排满到下一行开始继续排列 |
| Java.awt | BorderLayout（边界布局） | 容器划分为东、西、南、北、中五个区域，每个区域只能放置一个组件。 |
| Java.awt | GridLayout（网格布局） | 容器的空间划分成M×N列的网格区域, 每个区域只能放置一个组件。 |
| Java.awt | CardLayout（卡片布局） | 如同一叠牌，每个牌对应一个组件，但每次只能显示其中的一张牌。适用于在一个空间中防止多个组件的情况 |
| Java.awt | GridBagLayout（网格包布局） | GridLayout的升级版，组件仍然是按照行、列放置，但是每个组件可以占据多个网格 |
| Java.swing | BoxLayout（箱式布局） | 允许在容器中纵向或者横向防止多个控件 |
| Java.swing | SprigLayout（弹簧布局） | 根据一组约束条件放置控件 |
| 无 | 空布局 | 不使用布局管理器，按照控件自身提供的大小、位置信息放置控件 |

FlowLayout（流式布局）

*流式布局管理器要注意的事项：*

*1.流式布局管理器默认情况是居中对齐的。*

*FlowLayout(FlowLayout.align,int x, int y);*

*FlowLayout.align：指定对齐的方式 x:水平间距 y:垂直间距*

*LEFT 每一行组件都应该是左对齐的， 或者0*

*RIGHT 每一行组件都应该是右对齐的， 或者2*

*CENTER 每一行组件都应该是居中的， 或者1*

*2. panel默认的局部管理器就是FlowLayout.*

*方法：*

*int getRows() 获取此布局中的行数。 默认值是1*

*int getColumns() 获取此布局中的列数。 默认值是1*

*int getHgap() 获取组件之间的水平间距。 默认值是0*

*int getVgap() 获取组件之间的垂直间距。 默认值是0*

*设置*

*void setRows(int rows) 将此布局中的行数设置为指定值。 默认值是1*

*void setColumns(int cols) 将此布局中的列数设置为指定值。 默认值是1*

*void setHgap(int hgap) 将组件之间的水平间距设置为指定值。 默认值是0*

表格布局管理器(GridLayout)

*注意的事项： 如果表格数量不够使用时，默认会多加一列。*

*GridLayout(4,1,5,10)*

*前面两个参数（4，1）是指网格的行、列数*

*后面两个参数（5，10）是指组件的水平、垂直间隔*

*（一个组件占一个网格）*

*方法：*

*int getRows() 获取此布局中的行数。 默认值是1*

*int getColumns() 获取此布局中的列数。 默认值是1*

*int getHgap() 获取组件之间的水平间距。 默认值是0*

*int getVgap() 获取组件之间的垂直间距。 默认值是0*

*设置*

*void setRows(int rows) 将此布局中的行数设置为指定值。 默认值是1*

*void setColumns(int cols) 将此布局中的列数设置为指定值。 默认值是1*

*void setHgap(int hgap) 将组件之间的水平间距设置为指定值。 默认值是0*

AWT的UI元素

|  |  |
| --- | --- |
| Label | [标签](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Label.html) |
| Button | [按钮](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Button.html) |
| CheckBox | [复选框](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Checkbox.html) |
| CheckBoxGroup | 分组复选框 |
| List | 滚动的文本项 列表 |
| TextField | 文本框，编辑单行文本 |
| TextArea | 文本框，编辑多行文本 |
| Choice | 选择控制用于显示弹出菜单选择。所选选项将显示在顶部的菜单 |
| Image | 图像控制是表示图形图像的所有图像类的超类 |
| ScrollBar | 代表一个滚动条的滚动条控制组件以让用户选择的值来自范围（下拉列表框） |
| Dialog | 一个对话框控制是一个顶层窗口，采取某种形式的用户输入的标题和边框 |
| FileDialog | 一个FileDialog的控制代表一个对话框窗口，用户可以选择一个文件 |

Label（[标签](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Label.html)）

|  |  |  |
| --- | --- | --- |
| **修饰符和类型** | **领域** | **描述** |
| static int | **[CENTER](" \l "CENTER)** | 表示标签应居中。 |
| static int | **[LEFT](" \l "LEFT)** | 表示标签应左对齐。 |
| static int | **[RIGHT](" \l "RIGHT)** | 表示标签应该是右对齐的 |

构造函数（[具体细节](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Label.html)）

|  |  |
| --- | --- |
| **[Label](" \l "<init>())**() | 构造一个空标签。 |
| **[Label](" \l "<init>(java.lang.String))**(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** text) | 使用指定的文本字符串构造一个新标签，左对齐。 |
| **[Label](" \l "<init>(java.lang.String,int))**(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** text, int alignment) | 构造一个新标签，该标签显示具有指定对齐方式的指定文本字符串 |

方法（[具体细节](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Label.html)）

|  |  |  |
| --- | --- | --- |
| **修饰符和类型** | **方法** | **描述** |
| void | **[addNotify](" \l "addNotify())**() | 为此标签创建对等方。 |
| **[AccessibleContext](../../javax/accessibility/AccessibleContext.html" \o "class in javax.accessibility)** | **[getAccessibleContext](" \l "getAccessibleContext())**() | 获取与此Label关联的AccessibleContext。 |
| int | **[getAlignment](" \l "getAlignment())**() | 获取此标签的当前对齐方式。 |
| **[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** | **[getText](" \l "getText())**() | 获取此标签的文本。 |
| protected **[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** | **[paramString](" \l "paramString())**() | 返回表示此状态的字符串Label。 |
| void | **[setAlignment](" \l "setAlignment(int))**​(int alignment) | 将此标签的对齐方式设置为指定的对齐方式。 |
| void | **[setText](" \l "setText(java.lang.String))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** text) | 将此标签的文本设置为指定的文本 |

Button（[按钮](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Button.html)）

构造函数（[具体细节](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Button.html)）

|  |  |
| --- | --- |
| **[Button](" \l "<init>())**() | 构造一个带有空字符串的按钮作为其标签。 |
| **[Button](" \l "<init>(java.lang.String))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** label) | 构造具有指定标签的按钮。 |

方法（[具体细节](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Button.html)）

|  |  |  |
| --- | --- | --- |
| **修饰符和类型** | **方法** | **描述** |
| void | **[addActionListener](" \l "addActionListener(java.awt.event.ActionListener))**​(**[ActionListener](event/ActionListener.html" \o "interface in java.awt.event)** l) | 添加指定的动作侦听器以从此按钮接收动作事件。 |
| void | **[addNotify](" \l "addNotify())**() | 创建按钮的对等方。 |
| **[AccessibleContext](../../javax/accessibility/AccessibleContext.html" \o "class in javax.accessibility)** | **[getAccessibleContext](" \l "getAccessibleContext())**() | 获取AccessibleContext与此关联Button。 |
| **[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** | **[getActionCommand](" \l "getActionCommand())**() | 返回此按钮触发的动作事件的命令名称。 |
| **[ActionListener](event/ActionListener.html" \o "interface in java.awt.event)**[] | **[getActionListeners](" \l "getActionListeners())**() | 返回在此按钮上注册的所有动作侦听器的数组。 |
| **[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** | **[getLabel](" \l "getLabel())**() | 获取此按钮的标签。 |
| <T extends**[EventListener](../../../java.base/java/util/EventListener.html" \o "interface in java.util)**> T[] | **[getListeners](" \l "getListeners(java.lang.Class))**​(**[Class](../../../java.base/java/lang/Class.html" \o "class in java.lang)**<T> listenerType) | 返回当前FooListener在此注册为s 的所有对象的数组Button。 |
| protected **[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** | **[paramString](" \l "paramString())**() | 返回表示此状态的字符串Button。 |
| protected void | **[processActionEvent](" \l "processActionEvent(java.awt.event.ActionEvent))**​(**[ActionEvent](event/ActionEvent.html" \o "class in java.awt.event)** e) | 处理在此按钮上发生的操作事件，方法是将它们分派给任何已注册的ActionListener对象。 |
| protected void | **[processEvent](" \l "processEvent(java.awt.AWTEvent))**​(**[AWTEvent](AWTEvent.html" \o "class in java.awt)** e) | 处理此按钮上的事件。 |
| void | **[removeActionListener](" \l "removeActionListener(java.awt.event.ActionListener))**​(**[ActionListener](event/ActionListener.html" \o "interface in java.awt.event)** l) | 删除指定的操作侦听器，以便它不再从此按钮接收操作事件。 |
| void | **[setActionCommand](" \l "setActionCommand(java.lang.String))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** command) | 设置此按钮触发的动作事件的命令名称。 |
| void | **[setLabel](" \l "setLabel(java.lang.String))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** label) | 将按钮的标签设置为指定的字符串。 |

CheckBox（[复选框](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Checkbox.html)）

构造方法（[具体细节](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Checkbox.html)）

|  |  |
| --- | --- |
| **[Checkbox](" \l "<init>())**() | 创建一个复选框，其标签为空字符串。 |
| **[Checkbox](" \l "<init>(java.lang.String))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** label) | 创建具有指定标签的复选框。 |
| **[Checkbox](" \l "<init>(java.lang.String,boolean))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** label, **boolean** state) | 创建具有指定标签的复选框并设置指定的状态。 |
| **[Checkbox](" \l "<init>(java.lang.String,boolean,java.awt.CheckboxGroup))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** label, **boolean** state,**[CheckboxGroup](CheckboxGroup.html" \o "class in java.awt)** group) | 构造一个具有指定标签，设置为指定状态和指定复选框组的复选框。 |
| **[Checkbox](" \l "<init>(java.lang.String,java.awt.CheckboxGroup,boolean))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** label, **[CheckboxGroup](CheckboxGroup.html" \o "class in java.awt)** group, **boolean** state) | 在指定的复选框组中创建具有指定标签的复选框，并将其设置为指定的状态。 |

方法（[具体细节](file://E:\\Technological%20Learning\\Java\\JDK\\JDk-12.0.1_doc-all\\docs\\api\\java.desktop\\java\\awt\\Checkbox.html)）

|  |  |  |
| --- | --- | --- |
| **修饰符和类型** | **方法** | **描述** |
| void | **[addItemListener](" \l "addItemListener(java.awt.event.ItemListener))**​(**[ItemListener](event/ItemListener.html" \o "interface in java.awt.event)** l) | 添加指定的项侦听器以从此复选框接收项事件。 |
| void | **[addNotify](" \l "addNotify())**() | 创建Checkbox的对等方。 |
| **[AccessibleContext](../../javax/accessibility/AccessibleContext.html" \o "class in javax.accessibility)** | **[getAccessibleContext](" \l "getAccessibleContext())**() | 获取与此Checkbox关联的AccessibleContext。 |
| **[CheckboxGroup](CheckboxGroup.html" \o "class in java.awt)** | **[getCheckboxGroup](" \l "getCheckboxGroup())**() | 确定此复选框的组。 |
| **[ItemListener](event/ItemListener.html" \o "interface in java.awt.event)**[] | **[getItemListeners](" \l "getItemListeners())**() | 返回在此复选框上注册的所有项侦听器的数组。 |
| **[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** | **[getLabel](" \l "getLabel())**() | 获取此复选框的标签。 |
| <T extends**[EventListener](../../../java.base/java/util/EventListener.html" \o "interface in java.util)**> T[] | **[getListeners](" \l "getListeners(java.lang.Class))**​(**[Class](../../../java.base/java/lang/Class.html" \o "class in java.lang)**<T> listenerType) | 返回当前FooListener在此注册为s 的所有对象的数组Checkbox。 |
| **[Object](../../../java.base/java/lang/Object.html" \o "class in java.lang)**[] | **[getSelectedObjects](" \l "getSelectedObjects())**() | 返回包含复选框标签的数组（长度为1），如果未选中该复选框，则返回null。 |
| **boolean** | **[getState](" \l "getState())**() | 确定此复选框是处于“开”还是“关”状态。 |
| protected **[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** | **[paramString](" \l "paramString())**() | 返回表示此状态的字符串Checkbox。 |
| protected void | **[processEvent](" \l "processEvent(java.awt.AWTEvent))**​(**[AWTEvent](AWTEvent.html" \o "class in java.awt)** e) | 处理此复选框上的事件。 |
| protected void | **[processItemEvent](" \l "processItemEvent(java.awt.event.ItemEvent))**​(**[ItemEvent](event/ItemEvent.html" \o "class in java.awt.event)** e) | 处理在此复选框上发生的项事件，方法是将它们分派给任何已注册的ItemListener对象。 |
| void | **[removeItemListener](" \l "removeItemListener(java.awt.event.ItemListener))**​(**[ItemListener](event/ItemListener.html" \o "interface in java.awt.event)** l) | 删除指定的项侦听器，以便项侦听器不再从此复选框接收项事件。 |
| void | **[setCheckboxGroup](" \l "setCheckboxGroup(java.awt.CheckboxGroup))**​(**[CheckboxGroup](CheckboxGroup.html" \o "class in java.awt)** g) | 将此复选框的组设置为指定的复选框组。 |
| void | **[setLabel](" \l "setLabel(java.lang.String))**​(**[String](../../../java.base/java/lang/String.html" \o "class in java.lang)** label) | 将此复选框的标签设置为字符串参数。 |
| void | **[setState](" \l "setState(boolean))**​(**boolean** state) | 将此复选框的状态设置为指定的状态。 |

Swing

/\*\*

    \* ****1、借助java.awt.Desktop打开****

    \* @see 打开的目录或文件名中允许包含空格

    \*/

    private static void useAWTDesktop() throws IOException{

    Desktop.getDesktop().open(new File("D:/my local/测试用例.xls"));

    }

    /\*\*

    \*****2、 借助cmd命令打开****

    \* @see WPS文字--------Runtime.getRuntime().exec("cmd /c start wps")

    \* @see WPS表格--------Runtime.getRuntime().exec("cmd /c start et")

    \* @see WPS演示--------Runtime.getRuntime().exec("cmd /c start wpp")

    \* @see Office Word---Runtime.getRuntime().exec("cmd /c start winword")

    \* @see Office Excel--Runtime.getRuntime().exec("cmd /c start excel")

    \*/

    private static void useCMDCommand() throws IOException{

    //若打开的目录或文件名中不包含空格,就用下面的方式

    //Runtime.getRuntime().exec("cmd /c start D:/mylocal/测试用例.xls");

    //(可以'运行'或'Win+R',然后输入'cmd /?'查看帮助信息)

    Runtime.getRuntime().exec(new String[]{"cmd.exe", "/c", "D:/my local/测试用例.xls"});

    }

    /\*\*

    \* ****3、借助本地安装程序打开****

    \* @see 若打开的目录或文件名中包含空格,它就无能为力了..不过本地程序的安装目录允许含空格

    \*/

    private static void useLocalCommand() throws IOException{

    String etCommand = "D:/Program Files/WPS/8.1.0.3526/office6/et.exe";

    String filePath = "D:/mylocal/测试用例.xls";

    Runtime.getRuntime().exec(etCommand + " " + filePath);

    }

}

打开 “兴旺斗地主\_\_帮助文档.txt” 文件

File file=new File("C:/Users/汪兴/Desktop/兴旺斗地主\_\_帮助文档.txt");

InputStreamReader input=new InputStreamReader(new FileInputStream(file),"UTF-8");

BufferedReader reader=new BufferedReader(input);

StringBuilder sb=new StringBuilder();

while (reader.readLine()!=null){

sb.append(reader.readLine());

}