[Java基础 4](#_Toc279574883)

[JKD安装目录： 4](#_Toc279574884)

[标识符命名规则： 4](#_Toc279574885)

[Java符号集 4](#_Toc279574886)

[变量 和 常量 4](#_Toc279574887)

[基本数据类型 4](#_Toc279574888)

[表达式 6](#_Toc279574889)

[运算符 6](#_Toc279574890)

[流程控制 8](#_Toc279574891)

[数组 9](#_Toc279574892)

[访问控制 11](#_Toc279574893)

[内存管理与垃圾收集 13](#_Toc279574894)

[面向对象 13](#_Toc279574895)

[类 和 对象 14](#_Toc279574896)

[继承 和 多态 15](#_Toc279574897)

[继承 17](#_Toc279574898)

[多态 17](#_Toc279574899)

[方法 重写 和 重载 18](#_Toc279574900)

[访问修饰符(final / static) 19](#_Toc279574901)

[final 20](#_Toc279574902)

[static 23](#_Toc279574903)

[abstract 23](#_Toc279574904)

[基于继承的多态 25](#_Toc279574905)

[接口 25](#_Toc279574906)

[静态导入 27](#_Toc279574907)

[构造器 27](#_Toc279574908)

[创建一个对象时, 类的加载过程: 29](#_Toc279574909)

[抽象类 29](#_Toc279574910)

[内部类 29](#_Toc279574911)

[非静态内部类： 30](#_Toc279574912)

[局部内部类： 30](#_Toc279574913)

[静态内部类 （静态嵌套类、顶级嵌套类）： 30](#_Toc279574914)

[匿名内部类： 31](#_Toc279574915)

[内部接口： 31](#_Toc279574916)

[异常 32](#_Toc279574917)

[异常的分类 32](#_Toc279574918)

[异常处理机制 32](#_Toc279574919)

[throw 和 throws的使用 33](#_Toc279574920)

[自定义异常 33](#_Toc279574921)

[异常处理原则 35](#_Toc279574922)

[断言(1.4以后) 35](#_Toc279574923)

[多线程 36](#_Toc279574924)

[16.2 线程的状态 38](#_Toc279574925)

[16.3 线程的调度 39](#_Toc279574926)

[16.4 线程的同步 41](#_Toc279574927)

[16.5 获取当前正在运行的线程 42](#_Toc279574928)

[16.6 volatile关键字的含义与使用 42](#_Toc279574929)

[17.1 线程池的使用 42](#_Toc279574930)

[I / O 42](#_Toc279574931)

[以byte为单位的 字节流类 43](#_Toc279574932)

[InputStream类 43](#_Toc279574933)

[OutputStream类 44](#_Toc279574934)

[以char为单位的 字符流类 45](#_Toc279574935)

[Reader类 45](#_Toc279574936)

[Writer类 46](#_Toc279574937)

[常用I/O处理方式 46](#_Toc279574938)

[文件夹类 46](#_Toc279574939)

[文件的字节输入/输出流 47](#_Toc279574940)

[文件的直符输入/输出流 48](#_Toc279574941)

[对象序列化 48](#_Toc279574942)

[标准I/O 49](#_Toc279574943)

[读取控制台输入 49](#_Toc279574944)

[将System.out转换为PrintWriter 49](#_Toc279574945)

[标准I/O重导向 49](#_Toc279574946)

[Java输入/输出 类 和 接口 49](#_Toc279574947)

[字节流 49](#_Toc279574948)

[字符流 49](#_Toc279574949)

[文件处理 49](#_Toc279574950)

[文件操作 49](#_Toc279574951)

[文件访问 49](#_Toc279574952)

[写文件 50](#_Toc279574953)

[读文件 52](#_Toc279574954)

[随机文件的读写 53](#_Toc279574955)

[对文件 及 文件夹 进行操作 53](#_Toc279574956)

[复制文件 53](#_Toc279574957)

[复制文件夹 54](#_Toc279574958)

[删除文件 56](#_Toc279574959)

[分行写入文件 56](#_Toc279574960)

[对象 序列化 与 反序列化 56](#_Toc279574961)

[二进制文件 的 读写 操作 58](#_Toc279574962)

[I/O streams的运用 59](#_Toc279574963)

[socket 59](#_Toc279574964)

[JDK 5.0 新特性 59](#_Toc279574965)

[1. 泛型 59](#_Toc279574966)

[2. 增强的for循环 61](#_Toc279574967)

[3. 自动 装箱 / 拆箱 61](#_Toc279574968)

[4. 枚举 61](#_Toc279574969)

[5. 静态导入 61](#_Toc279574970)

[6. 可变参数(变长参数) 62](#_Toc279574971)

[注解(Annotation) 62](#_Toc279574972)

[反射 63](#_Toc279574973)

[集合 63](#_Toc279574974)

[Hash表 65](#_Toc279574975)

[常用的集合接口 68](#_Toc279574976)

[Collection 68](#_Toc279574977)

[List 68](#_Toc279574978)

[Set 68](#_Toc279574979)

[Map 68](#_Toc279574980)

[Iterator 68](#_Toc279574981)

[常用集合类 68](#_Toc279574982)

[Iterator 68](#_Toc279574983)

[ArrayList 和 LinkedList 68](#_Toc279574984)

[HashSet 和 TreeSet 68](#_Toc279574985)

[HashMap 和 treeMap 68](#_Toc279574986)

[遍历Enumeration 68](#_Toc279574987)

[遍历map 69](#_Toc279574988)

[遍历List 69](#_Toc279574989)

[集合的简单输出 70](#_Toc279574990)

[遍历Set 70](#_Toc279574991)

[***集合的简单输出*** 70](#_Toc279574992)

[正则表达式 70](#_Toc279574993)

[常用包 71](#_Toc279574994)

[text包 71](#_Toc279574995)

[常用类 71](#_Toc279574996)

[Object类 71](#_Toc279574997)

[Math类 72](#_Toc279574998)

[字符串 相关类 72](#_Toc279574999)

[日期 和 时间 相关类 76](#_Toc279575000)

[Date常用代码 78](#_Toc279575001)

[Java获取各种常用时间方法 78](#_Toc279575002)

[日期 <----转----> 字符串 99](#_Toc279575003)

[基本数据类型 和 封装类 100](#_Toc279575004)

[Random类 103](#_Toc279575005)

[***URL编码 解码*** 103](#_Toc279575006)

[把字符串从一种编码转换成另外一种编码 103](#_Toc279575007)

[Random类： 104](#_Toc279575008)

[事件处理 104](#_Toc279575009)

[Swing 104](#_Toc279575010)

[AWT 105](#_Toc279575011)

[Applet 105](#_Toc279575012)

[\_\_\_\_\_\_\_\_\_未分类\_\_\_\_\_\_\_\_\_ 105](#_Toc279575013)

运行jar包里的类，添加额外jar包

|  |
| --- |
| #!/bin/bash  set -x  hbase=/home/docker/lib  for i in `ls $hbase|grep jar`; do x+="$hbase/"$i":"; done;  jars=$x  java -classpath /home/docker/cdc-audit-package.jar":"$jars com.xjgz.cdc.audit.App |

# Java基础

下载JDK：<http://java.sun.com/javase/downloads/iondex.jsp>

Java API官方网站：<http://java.sun.com/reference/api>

## JKD安装目录：

|  |  |
| --- | --- |
| bin | 开发中需要的工具 |
| demo | 演示程序 |
| db | 使用嵌入式数据库Derby开发所需要的资源以及一些案例 |
| include | 用于编译本地方法的C++头文件 |
| jre | java运行时环境 |
| lib | 开发工具包的类库文件 |
| sample | 开发工具包自带的示例程序。 |
| src.zip | 类库API源代码文件。 |

## 标识符命名规则：

1. 开头：字母、下划线、美元符号。
2. 后面：字母、下划线、美元符号、数字。

提倡的命名习惯：

1. 包名：全部小写。
2. 类名、接口名：由名词组成，所有单词第一个字母大写，其它字母小写。
3. 方法名：第一个单词是动词，第一个字母小写，后面每个单词第一个字母大写，其余小写。
4. 变量名：成员变量名与成员方法名相同；局部变量全部小写。只有临时变量可以使用单字符命名。
5. 常量名：全部大写，用下划线分割单词。

Java中字符编码不是ASCII码（8bit，1字节），而是Unicode码（16bit，2字节）。

## Java符号集

## 变量 和 常量

成员变量系统会自动初始化。

## 基本数据类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据类型 | 关键字 | | 占用内存(位) | 默认值 |
| 整形 | 字节型 | byte | 8 | 0 |
| 短整型 | short | 16 | 0 |
| 整形 | int | 32 | 0 |
| 长整形 | long | 64 | 0 |
| 浮点型 | 单精度 | float | 32 | 0.0f |
|  | 双精度 | double | 64 | 0.0d |
| 字符型 | 字符型 | char | 16 | '\u000' |
| 布尔型 | 布尔型 | boolean | 8 | false |

整形的三种进制表示

|  |  |
| --- | --- |
| 八进制 | 1~9开头的数 |
| 十进制 | 以0开头, 且其后是0~7范围组成的整数 |
| 十六进制 | 以0x或0X开头, 且其后是0~9, a~f组成 |

**基本数据类型 值之间的转换**

1. 自动转换

|  |
| --- |
|  |

2. 手动强制转换:

|  |
| --- |
| 语法格式: 在圆括号中给出想要转换的目标类型, 随后紧跟待转换的变量或字面常量.  例如:  double d = 5.998;  int i = (int)d;  byte b = (byte)5.998; |

3. 隐含强制转换

|  |
| --- |
| Java中有一种特殊情况, 因为没有专用的后缀来表示byte/short类型的字面常量, 所以Java系统允许在编写代码时用int类型的字面常量来表示byte, short类型的字面常量,由编译系统来自动进行转换识别.  byte b = 123;  short s = 123;  char c = 134; //可将0~65535的整数字面常量直接赋值给char型变量.  这其中就包含隐含的强制转换, 编译系统会自动将"123"变为byte型,而不是int型. 但要注意的是,对变量则不行.  如:  int i = 123;  byte b = i; //正确做法为: byte b = (byte) i; |

## 表达式

## 运算符

|  |  |
| --- | --- |
| "+"运算符 | 加法运算符  数值正号  字符串连接符 |
| "-"运算符 | 减法运算  取负运算 |
| "\*"运算符 | 乘法运算(同样遵循自动提升法则) |
| "/"运算符 | 整除运算: 两个整数相除, 即使有结果有小数, 也只保留整数部分. 如果被除数小于除数,结果永远为0  浮点除运算: 如果参与运算的有一个为浮点数,则进行浮点除. 规则如下:  1. 两个浮点数或一个浮点数与一个整数进行除法运算, 结果将得到一个浮点数, 即使结果是一个整数, 也被存储为浮点值.  2. 正浮点数(非零浮点数)除以零, 将得到结果Infinity(正无穷大);  3. 负浮点数(非零浮点数)除以零, 将得到结果-Infinity(负无穷大);  4. 浮点数零(0.0)除以零, 结果为NaN. |
| %运算符 | 求余(取模)运算: 左边的操作数的绝对值除以右边的操作数的绝对值, 结果取余数.  有两种特殊情况:  1. 整数进行求余运算. 如果右边的操作数为零, 则报运行时错误.  2. 浮点数进行求余运算, 如果右边的操作数为零, 则得到结果NaN.   |  |  | | --- | --- | | 参与运算的数据类型： | 前面那个只能是整形 | | 结果类型： | 取决于后面那个 | | 结果符号： | 取决于前面那个 | |
| 自增: ++  自减: -- |  |

关系运算符

|  |  |
| --- | --- |
| 等于/不等于 运算 | ==  !=  无穷大 与 NaN 的比较 |
| 比较大小运算 | >  >=  <  <= |

逻辑运算符

|  |  |
| --- | --- |
| 与 | 短路与 &&  非短路与 & |
| 或 | 短路或  非短路或 |
| 非 | ! |

三元运算符

|  |
| --- |
| 逻辑表达式 ? 结果表达式1 : 结果表达式2 |

位运算符

|  |
| --- |
| 位与 &  位或  异或 ^ |

移位运算符

|  |  |
| --- | --- |
| 左移<< |  |
| 右移>> |  |
| 无符号右移>>> | 不管高位是0还是1, 都补0 |
| 注意:  在进行移位前, Java系统首先把 要移的位数 与 被移数的位数 进行求余, 然后移动余数个位数.  把一个整数移动比他位数还多的位数是没有价值的. | |

赋值运算符

|  |  |
| --- | --- |
| 普通赋值运算符 | = |
| 运算赋值运算符 | += -= \*= /= %= |

括号 及运算符间的优先级

|  |  |  |
| --- | --- | --- |
| 由高到低 | 运算符 | 计算顺序 |
| 14 | [] . () | —> |
| 13 | ! ~ ++ -- +(取正) -（取负） | 🡨 |
| 12 | \* 、 % | —> |
| 11 | +（加） -（减） | —> |
| 10 | << >> >>> | —> |
| 9 | < > <= >= instanceof | —> |
| 8 | == != | —> |
| 7 | & | —> |
| 6 | ^ | —> |
| 5 | | | —> |
| 4 | && | —> |
| 3 | || | —> |
| 2 | ? : | 🡨 |
| 1 | = += -= \*= /= %= ^= |= &= >>= <<= >>>= | 🡨 |

NaN：not a number。

## 流程控制

|  |
| --- |
| switch语句：  表达式类型：  1、基本类型：int兼容的（byte(1字节)、char(2字节)、short(4字节)、int(4字节)）。  2、枚举类型。  case值必须满足：   * 1. case值类型与switch判断表达式相同；   2. case值为常量（可以是含final修饰的常量表达式）；   3. 不能有两个相同的case值。 |
| while语句：  while(条件表达式)  {语句序列} |
| do-while：  do  {语句序列}  while(条件表达式); |
| java中的break和continue都可以有标号  标号下一行必须的循环语句块，是给循环语句块起一个名。  brdak语句的两个用法:  1. 在switch语句中, 表示一个case的结束, 退出switch;  2. 作为循环控制语句, 在循环体中表示退出循环.   |  | | --- | | labell: //声明标号"labell", 标识紧跟的for语句  for(int i=0, j=0; i<10; i++){  ...  break labell; //退出循环;  } |   当continue执行时, 本次循环结束, 进入条件判断.  目的是跳出循环。  标号必须单独占一行。 |

## 数组

|  |
| --- |
| 数组：用来存储同一类型的值。  数组可以是基本数据类型，也可以是对象引用类型。  数组的维数只与方括号的对数有关，与方括号对于数组引用之间位置无关。  1. 声明数组引用：int[] a;  2. 创建数据对象: new int[5]; new int[2][3]; //创建数据对象时, 一定要给出数组第一维所需的长度. 否则系统无法为其分配空间. 从而导致编译失败. 至于第二维以及之后的长度可给可不给.  创建数据对象后, 需要将相应类型的数组引用指向该数组对象. 才能对数组进行操作. 也可把第1. 2 步合并一行.  3. 给已声明的数组引用赋值：a = new int[1]{90};  给数组赋值null：a = null;  4. 匿名数组：new int{2, 3, 4};  多维数组对象不一定一次创建完成，可以逐步创建。  多维数组除了最底层外，其它都是对象数组。  数组创建完成后，系统会自动初始化：除了boolean为false外, 其它都是相应的零. |
| 枚举初始化数组：  1、标准语法：String[] name = new String[] {“jinwanlin”, “wugang”, “luochuan”};  2、省略语法：String[] name = {“jinwanlin”, “wugang”, “luochuan”};  数组用于for循环：  for(type element: array){ //一维数组  System.out.println(element);  }  for(int[] row: arr){ //二维数组  for(int element: row){  System.out.println(element);  }  }  数组赋值: 传递引用  1. 基本类型数组相互赋值规则：  a、维数要相等。  b、类型要完全相同。  2. 引用型数组相互赋值规则:  a. 数组的维数要相等;  b. 数组元素的类型要兼容, 也就是说数组元素可以赋值;  数组的常用操作  1. 数组复制; 传递值   |  |  | | --- | --- | | public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) | src:任意类型的数组引用, 指向源数组  srcPos: 从原数组中拷贝元素的起始下标;  dest: 与src同类型的数组引用, 指向目标数组;  destPos: 目标数组放置拷贝元素的起始下标  length: 拷贝元素的个数, 如果越界, 抛异常. | | public static xxx[] copyOf(xxx[] original, int newLength) | original: 任意类型的数组引用, 指向源数组  newLength: 制定拷贝数组的长度, 如果拷贝数组长度大于源数组长度则会用0或null值对拷贝数组填充, 否则截取源数组的一部分拷贝到拷贝数组中. 返回值为拷贝数组的引用. | | public static xxx[] copyOfRange(xxx[] original, int from, int to) | original: 任意类型的数组引用, 指向源数组  form 和 to为开始和结束的下标位置.  newLength: 制定拷贝数组的长度, 如果拷贝数组长度大于源数组长度则会用0或null值对拷贝数组填充, 否则截取源数组的一部分拷贝到拷贝数组中. 返回值为拷贝数组的引用. | |
| 数组排序   |  |  | | --- | --- | | public static void sort(xxx[] a) | 对指定的数组进行排序, 排序结果还存放在原来数组中 | | public static void sort(xxx[] a, int fromIndex, int toIndex) | 对指定数组的区间进行排序,  fromIndex 与 toIndex用来指定区间, 排序结果还存放在原数组中 | | 说明: xxx代表任意类型, 这两个方法也可以对引用应用类型数组进行排序, 但要求引用所在类的对象必须有特定的自然顺序, 即引用所在的类必须实现了Comparable接口. | | |
| 搜索指定元素   |  |  | | --- | --- | | public static int binarySearch(xxx[] a, xxx key) | 从数组中索引第一个指定值元素的位置, 返回值为搜索到位置的索引, 如果没有找到, 则返回负数. | | public static int binarySearcy(xxx[] a, int fromIndex, int toIndex, xxx key) | 从数组中索引第一个指定值元素的位置, fromIndex和toIndex指出搜索区间, 返回值为搜索到位置的索引, 如果没有找到, 则返回负数. | | 说明:xxx代表任意类型， 这两个方法都是基于二元搜索算法的， 要求被搜索的数组是已经排好序的, 可以在搜索前用sort系列方法排序. | | |
| 比较两个数组中的元素值是否相同.  boolean flag = Arrays.equals(a, b); //a, b 是两个数组的引用. |

## 访问控制

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **类的成员的访问控制** | **同一个类中** | **同一个包中** | **子类中** | **任意范围** |  | | **private(私有)** | Y |  |  |  | 本类 | | **缺省(包级别)** | Y | Y |  |  | 本包类 | | **protected(子类级别)** | Y | Y | Y |  | 本包类、所有子类 | | **public** | Y | Y | Y | Y | 所有类 | |
| 类的访问控制只有两种：  public：公共类：任何包中的类都可以访问公共类。  默认级别：包级： 只能被同一个包中的类访问。 |
| 存储方式修饰符：   1. 静态：static修饰。 2. 非静态：。省略不写。 |
| 与继承有关的关键词：   1. final：不能被重写。 2. abstract：必须被重写。 |
| package：   1. 包语句必须放在源文件最前面，只能有一个包语句。 2. 运行程序时，要将当前目录设置为最外层的包文件夹所在目录。 3. 匿名包：没有名字的包，编译后类文件和源文件在同一个目录中。 |
| import：   1. import 包名.\*； 2. import 包名.类名； 3. import语句只在编译时候起作用。 4. 系统自动加上：import java.lang.\*; 5. 编译系统会优先匹配import语句中明确给出类名的类。 |
| 静态引入（导入静态方法和静态成员变量）：（很少使用）   1. 需要在import后加static关键字： 2. import static 包名.类名.\*； 3. import static 包名.类名.成员方法名/成员变量名； 4. 如果需要使用大量带有繁琐名字的静态成员，可以使用静态引入。 |

|  |
| --- |
| final：  1、final成员变量：在构造器完成前必须进行显示初始化。  初始化时间：  1、声明的同时；  2、非静态语句块中；  3、构造器中。  2、final局部变量：可以只声明，不初始化。初始化在使用前就行。  7.5 static  1、static只能修饰成员（变量、方法、语句块、内部类）。  2、static修饰的成员属于类，不属于任何特定的对象。  3、静态成员为该类所共享。  静态最终成员变量（常量）：  1、static属于类，表示不必创建对象就可以使用。  2、final表示值不能改变。（最终成员，能被继承，不能被覆盖。）  3、一般都设为public。  4、必须显示初始化。常量属于类，等不到构造器运行再初始化，必须在静态语句块结束之前完成初始化。因此类加载完成后必须可以使用。  5、在声明的同时进行初始化；在静态语句块中进行初始化。  6、引用方式：类名.静态成员名；对象名.静态成员名；（本类中直接访问）静态成员名。  静态最终成员变量：必须在静态语句块结束之前对其初始化。  非静态最终成员变量：必须在构造器完成之前对其初始化。 |
|  |

## 内存管理与垃圾收集

18.1 什么是垃圾

垃圾：不再有用的对象。

18.1.1 对象成为垃圾的条件

1、对于非线程对象来说，当所有活动线程都不可能访问到该对象时，该对象变成为“垃圾”。

2、对于线程对象来锁，除了要满足第一条的标准之外，还要求此线程本身已经死亡或者还处于新建状态。

18.1.2 符合条件的几种情况

1、单个对象的情况：将指向该对象的引用不再指向该对象即可。

2、多个对象的孤岛情况：不能被活动线程访问到。

18．2 垃圾收集器

18.2.2 申请垃圾收集器运行

18.3 垃圾收集前的处理工作

18.3.1

# 面向对象

|  |  |
| --- | --- |
| 封装 |  |
| 继承 |  |
| 多态 |  |

## 类 和 对象

对象

类

封装

抽象

抽象 与 封装 的 区别

类的实现

类的定义

类的成员变量

类的成员方法

实例化 和 构造方法

包

创建包

导入包

|  |
| --- |
| **定义类**  类的声明  类体  成员变量  局部变量  成员方法  构造方法 |
| **创建对象**  声明对象  为对象分配内存 |
| **使用对象** |
| **销毁对象** |

## 继承 和 多态

|  |
| --- |
| 类之间的关系：  1、USES-A：类A的方法用到了类B的成员----类A“USES-A”（用到了）类B。  2、HAS-A：类A中有类B类型的成员引用变量----类A“HAS-A”(拥有)类B。  3、IS-A：类A是类B子类的一种。----类A“IS-A”（是一种）类B。  8.3 成员的继承与隐藏  继承与否，完全取决于访问限制。  成员变量的隐藏：  当子类本身具有与继承自父类的成员变量名称相同的成员变量时， 便构成了成员变量的隐藏。  super与this一样：表示一个预定义对象引用。  this表示自己这个对象；super表示父类对象。  8.4 对象引用的使用：  1、任何对象引用，都可指向自身对象实例。  2、父类的引用变量，可以指向子类（不一定是直接子类）的对象实例。  3、对成员变量的访问：引用是什么类型，系统就访问谁的成员。  引用类型转换：只能是父类与子类之间相互转换。  对象引用调用成员的规则：引用只能调用其所在类具有的成员，不能调用子类特有的成员。  强制类型转换，转的只是引用类型，真正指向的对象是不会发生变化的。  对象引用的赋值：  1、可以直接将子类引用赋值给父类引用。  2、父类引用赋值给子类引用，必须强制类型转换。  引用的比较：  1、相同类型的引用可以比较。  2、不同类型的引用要进行比较，其中一个的类型必须派生自另一个。否则编译报错.  方法重写（覆盖）：  父类引用只能调用父类中定义的方法，若只是子类中有的方法，不能通过父类引用调用。  方法的调用按对象的类型。无论使用什么类型的引用，调用的都是具体对象所在类中定义的方法。这与成员变量不同。父类引用指向子类对象时，若访问被重写的方法，则将访问被重新定义的方法。  方法的调用：按对象类型，调用具体对象中的方法。  成员变量的调用：按引用类型，调用引用类型类中的成员变量。  调用隐藏的静态方法：引用类型的。  调用重写的非静态方法：对象类型的。  方法重写条件：  1、返回类型若为基本类型，则返回类型必须完全相同。  2、返回类型若为对象引用类型，必须是相同的对象引用类型，或子类对象引用类型。  3、访问级别限制：一定不能比被重写方法的窄。  4、不能够重写final方法。  5、重写基于继承。  6、静态方法只能被子类静态方法隐藏。用父类调用的是父类的静态方法，用子类调用的是子类的静态方法，与对象类型无关。  7、隐藏：根据引用的类型来调用的。  8、重写：根据对象的类型来调用的。  8.6方法重载：  方法重载条件：   1. 参数表必须不同。 2. 返回值类型不管。 3. 存在于同一类中，方法名相同的函数。 4. 在方法调用进行匹配的时候，首先选择直接匹配的方法。 5. 如果没有可直接匹配的方法，则将参数进行提升转换后再匹配方法。总是匹配最接近的方法。 6. 对于基本类型而言：按照基本类型之间的自动转换规则进行提升。 7. 对于引用型而言：选择最接近的父类类型。 8. 调用：参数类型决定选择哪个重载方法。   8.8 Abstract：  Abstract类（又叫抽象类）：   1. 必须包含抽象方法，可以包含非抽象方法。 2. 非抽象类不能包含抽象方法。 3. 抽象类不能有构造函数，不能实例化（也就是创建对象）。 4. 不能有静态方法。 5. 不能把同一个类标识为abstract和final。因为这两个关键字的意义正好相反。 6. 抽象类中的非抽象方法：是抽象类所有子类都具有的，子类不需要重写的方法。 7. 抽象类的第一个非抽象子类必须实现其父类所有的抽象方法。其中包括父类继承的抽象方法。 8. 抽象类只能继承一个父类，但可以实现多个来自任何地方的接口。   Abstract方法（又叫抽象方法）：   1. 只有方法声明，没有方法体。 2. 用abstract来声明抽象方法。 3. 用分号标识声明结束。 4. 抽象方法只存在于抽象类中。 5. 抽象方法不能是private的。因为抽象方法需要在子类中继承并重写来具体实现。   非静态语句块：直接用大括号括住，创建对象的时候自动执行一次。  8.9多态：  多态性：不同的对象有相同的一般轮廓或形态，但具体执行的过程却大相径庭。  基于继承的多态：对象功能的调用者，用超类的引用来进行方法调用。因为超类的引用能调用各种不同的子类实现。父类的引用可以使用具体子类实现的各种不同功能。特定类型的子类引用只能指向这个类的对象。  什么不能被继承：  1、受访问修饰符限制的成员。  2、构造方法。 |

### 继承

父类的static成员方法能被继承, 若子类定义了同名的方法, 则把父类的方法隐藏. 且父类方法是static的, 子类也必须是static的. 父类方法不是static的, 子类方法也不能是static的.

父类的final成员方法能被继承, 但不能被重写.

父类的成员变量(不管是static的还是final的)都可能被继承, 但有可能被隐藏.

|  |
| --- |
| 1. 如果子类声明了一个与父类成员变量相同的成员变量, 则子类的成员变量隐藏了父类的成员变量.  2. 如果子类声明了一个与父类成员方法相同的成员方法, 则子类的成员方法覆盖了父类的成员方法. |
| 成员变量的访问是按引用的类型, 访问子类变量: ((Person)p).name  成员函数的访问是对象的的类型. |

### 多态

|  |
| --- |
| 基于继承的多态: 拿着父类的引用, 可以根据需要将引用指向不同的子类对象, 来调用不同的子类实现. |
| 基于接口的多态: |
| 基于重载的多态: 系统在调用方法时，会根据参数的类型自动选择调用合适的方法。 |

### 方法 重写 和 重载

**重写/覆盖**

**依据:** 方法名和参数列表 必须完全相同

|  |
| --- |
| 重写方法时，不能抛出更多的异常 |
| 返回类型的规则  基本数据类型: 必须要相同。  对象数据类型: 必须与被重写方法返回类型相同, 或被重写方法的返回值类型的子类型; |
| 访问限制的要求:访问权限只能相同或更大，不能变得更小。 |
| 重写基于继承  只有能被访问的方法，才会被继承。  只有能被继承的实例方法才能被子类重写。  默认访问权限的实例方法，可能会被子类重写，也可能不能被子类重写（这要看该实例方法能否被继承）。 |
| 静态方法的隐藏(没有重写)  当子类中重写的方法也是静态方法时, 实质上是将父类的静态方法隐藏, 而不是将其重写.  方法重写只针对实例方法，对于static方法，只有隐藏。没有重写这一说。  不能将父类的实例方法，在子类中“重写”成static方法。否则编译抛出异常  不能将父类的static方法,在子类中“重写”成非static方法。 |
| 不能重写被表示为final的方法.(final修饰的方法可以被继承, 但不能被重写.) |
| 隐藏是根据引用类型来调用的, 重写是根据对象的类型来进行调用的.  静态方法是根据引用来进行调用的, 所以不会被重写, 只会被隐藏; |

**重载**

**依据:** 相同方法名, 不同参数列表

|  |
| --- |
| 规则  1. 重载的方法参数列表各不相同  2. 重载的方法返回值类型, 访问限制 没有要求, 可以相同,可以不同. |
| 重载方法的匹配: 根据参数类型决定调用哪个方法.  1. 基本类型参数  首先选择直接匹配的方法, 如果没有, 则将参数进行提升转换后再匹配方法, 总是匹配最接近的方法.  对于基本类型而言, 就是按照基本数据类型间的自动转换关系, 将传递的参数进行自动转换, 并寻找最匹配的一个方法.  2. 引用型参数  对于引用型参数一样, 当调用重载的方法时, 若与某个重载方法的参数完全匹配, 则调用该方法.  若没有完全匹配的方法, 则寻找重载方法的参数类型中哪个最能兼容传递的参数, 就选择哪个方法.  引用类型决定调用哪个方法, 而不是对象类型. |

**重写与重载的区别**

|  |  |  |
| --- | --- | --- |
|  | 重载 | 重写 |
| 参数列表 | 必须修改 | 一定不能改 |
| 返回类型 | 可以任意修改 | 对于基本数据类型必须相同, 对于对象引用类型必须能够兼容 |
| 访问限制 | 可以修改 | 一定不能做更严格的限制, 但可以降低访问限制 |
| 调用 | 参数类型决定选择哪个重载方法 | 对象类型决定了调用什么样的方法. |
| 抛出异常 (未总结) |  |  |

### 访问修饰符(final / static)

**成员修饰符**

#### final

|  |
| --- |
| Final类：   1. 使用地点：当类中的所有方法都不能被重写时才使用。 2. 最终类，不能有子类。   Final方法：   1. final方法只被能继承，不能被重写。   final变量  对于原始类型, 该变量一旦被显示初始化后其值就不能被修改。  对于引用类型, 一旦赋值之后, 将不能修改final引用指向哪一个对象, 可以修改final的引用的引用的对象.  final成员变量：  1. 系统不会默认赋初值。而是要求在构造器完成之前必须显示初始化。  2. 初始化时机有以下三个  a 在声明的同时进行初始化;  b 在非静态语句块中进行初始化  c 在构造器中进行初始化  final局部变量：  1、可以只声明不初始化。必须人工初始化，系统不自动初始化。一旦初始化则不能修改.  2、初始化的时间只要在使用其值之前就行。  **static final静态最终成员变量:**  显示赋值要求在静态语句块结束之前必须完成.  1. 声明的同时  2. 静态语句块 |

**final 与 继承**

|  |
| --- |
| 不能继承final修饰的类.  final修饰的方法只能继承, 不能重写(覆盖) |

#### static

|  |
| --- |
| 静态成员变量 |
| 静态成员方法 |
| 静态代码块 |

#### abstract

**abstract 与 继承**

|  |
| --- |
| 抽象类永远不能实例化, 其唯一的用途是用与继承扩展.  抽象方法只是方法声明, 没有方法体.  抽象方法不能是private的.  当某类继承自抽象类时, 如果其本身不是抽象类, 则必须实现所继承抽象类中的所有抽象方法. |

|  |
| --- |
| 获取对象所属类的名字：  String name = 对象名.getClass().getName();  类 的声明格式：  [public]  [abstract/final]  class className  [extends supperClassName]  [implements]  成员变量 的声明格式：  [public/protected/private]  [static]  [final]  [transient]  [volatile]  type variableName  成员方法 的声明格式：  [public/protected/private]  [static]  [abstract/final]  [native]  [synchronized]  return type methordName()  [throws exceptionlist]  方法  返回值：  1、对象引用返回类型的方法中可以返回null，基本数据类型作为返回类型时，则不可以返回null。  2、在基本数据返回类型的方法内，可以返回任何值，只要其能够自动转换为返回值类型。（与C语言相同）  变长参数：  1、一个方法最多只能有一个变长参数。  2、如果方法的参数不止一个，变长参数必须是最后一个。  局部变量：  1、编译器不会自动初始化。  2、不使用的变量，可以不初始化，但它也就没有存在的意义了。  this 和 super  this.成员：调用本类的成员。  super.成员：调用父类被子类覆盖的成员。  this(参数表)：调用兄弟构造方法，只能在构造函数第一行使用。  super(参数表)：调用父类构造方法，只能在构造函数第一行使用。  1、成员方法调用本类成员。  2、this只能在属于对象的方法或语句块中使用，类似main这样的static块中不可使用。  字符串（String）类型的空引用（null）：没有指向任何对象。  空字符串：内容为空的字符串对象。  空引用：编译能通过，运行会出错。  成员引用：系统会自动为其分配初值null，为了确保正确，只要在使用前检查是否为null即可。  局部引用：系统不会自动赋值null，所以检查是否为null是没用的。 |

### 基于继承的多态

|  |
| --- |
|  |

## 接口

只能包含的成员

|  |
| --- |
| 常量 public static final  抽象方法 public abstract  接口中不能有静态语句块等执行部分. 所以常量的赋值只能在声明的时候. |
| 接口的实现类必须重写所有抽象方法. |
| **接口的声明**  接口默认类型：[public] [abstract] interface 接口名{}  永远不能用final来修饰接口。  不能用static, final来修饰接口中的方法.  接口可以继承多个接口。不可以继承类。（其它都是单继承） |
| 接口中的成员变量：   * 1. 默认为public static final的，一般不写出来。   2. 接口中不能有静态语句块等执行代码部分, 所以常量的初始化只能在声明的同时进行.   接口中的成员方法：   1. 默认为public abstract的。也只能是public abstract的，一般都不写出来。 2. 接口中的方法，一定不能是静态的，也不能用public之外的其它限制修饰符进行修饰。   接口的实现：   1. class 类名 implements 接口名列表 2. 实现接口的类之间不能有任何继承关系。   接口引用：   1. 接口不是类，不可以用new实例化。 2. 但某个类实现了接口后，该类的对象可以用其实现的接口或父接口类型的引用来指向。 3. 子接口引用可以直接赋值给父接口引用。 4. 父接口引用需要强制转换后才能赋值给子接口引用。 5. 其它没有继承关系类型的接口引用也需要进行强制类型转换。   类引用 与 接口引用之间的赋值：   1. 类引用可以直接赋值给接口引用。只要类直接或间接地实现了接口就行。 2. 接口引用需要强制类型转换才可以赋值给类引用，而且类必须实现了该接口才行。   9.4 接口 与 抽象类  接口：  1、不能实现其它接口。  抽象类：  1、可以选择性的设计父类或父接口中的抽象方法。  接口的多态：   1. 继承只能让超类引用指向与其属于同一类的对象。 2. 接口引用这能指向任何类型的对象。只要这些对象所在的类实现了该接口即可。 3. 开发中，习惯使用接口来调用，而不是使用实例对象来调用。   接口回调：？？？？  9.7 instanceof  Boolean b=a instanceof B;  1、含义：引用a中的对象类型时候是B类型的。  一个苹果属于苹果类；  一个苹果属于水果类。  4、对基本类型数组，不可看着其它任何类型的数组对象。  5、参与测试的引用类型必须能够强制类型转换成测试的类或接口类型，才能通过编译。  6、在执行强制类型转换前应该首先执行instanceof测试，若通过，再进行强制类型转换。 |

## 静态导入

|  |
| --- |
| import static <包名>.<类名>.\*;  import static <包名>.<类名>.<具体静态方法/成员变量名>;  好处: pow(3, 2) 比 Math.pow(3, 2) 看起来要清晰得多. |

## 构造器

|  |
| --- |
| 构造器：   * 1. 创建对象时自动运行的。   2. 不会被继承的。   3. 不能有返回类型（也无返回值）。若有返回值，则是 方法名 与 类名 相同的 普通成员方法。   4. 不可以被方法调用。   构造器的访问限制修饰符：   1. public （任何地方） 2. protected 3. 缺省类型 （包内） 4. private （本类）   静态工厂：构造函数是private的，成员方法是静态的，而且有创建对象的功能。  创建对象：  new 构造器名\类名（参数表）  自己编写的调用父类构造器的代码必须位于构造器的第一句，否则编译错误。  若不在构造器中编写调用父类构造器的代码，系统则调用父类的无参构造器，若需要调用父类有参构造器，则必须自己编写代码。  调用父类构造器：super(\*\*\*);必须写在第一行。  调用兄弟构造器：this(\*\*\*);必须写在第一行。  调用兄弟构造器后，不会再自动加上父类构造器了，而会调用兄弟构造器中的父类构造器。  单列模式：   1. 某个类在同一时刻只允许有一个对象，这个成之为单列模式。 2. 节约资源。   java程序的加载过程：   1. 加载要创建对象的类，及其 直接或间接父类。 2. 在类被加载的同时，会加载静态成员（静态成员变量的初始化，静态语句块的执行，在加载时按代码的先后顺序执行）。 3. 加载完需要的类后，开始创建对象，首先加载非静态成员（非静态成员变量的初始化，非静态语句块的执行，在加载时按代码的先后顺序执行）。 4. 执行构造器，构造器执行完毕，对象生成。   注意：再次创建同类的对象时，类的加载不再进行，语句块（静态和非静态）只执行一次，加载完成后不再执行。创建子类对象时上述过程要先走父类，一直递归到java.lang.Object类。 |
| 静态工厂方法  **public** **class** Person {    /\*\*  \* 构造方法  \*/  **private** Person(){  }    /\*\*  \* 静态工厂方法, 返回对象  \* **@return**  \*/  **public** **static** Person getInstance() {  **return** **new** Person();  }    } |

### 创建一个对象时, 类的加载过程:

**加载类**

1. 加载要创建对象的类及其直接与间接父类;

2. 在类被加载的同时会将静态成员进行加载, 主要包括静态成员变量的初始化, 静态语句块的执行, 在加载时按代码的先后顺序进行.

**创建对象**

3. 类加载完成后, 开始创建对象, 首先会加载非静态的成员, 主要包括非静态成员变量的初始化, 非静态语句块的执行, 在加载时按代码的先后顺序进行.

4. 最后执行构造器, 构造器执行完毕, 对象生成.

## 抽象类

|  |
| --- |
| 特点:  1. 有抽象方法的类, 一定是抽象类;  2. 抽象类, 不一定要有抽象方法;  3. 抽象类 的 子(非抽象)类: 必须实现所有抽象方法. |

# 内部类

**各种内部类可用的修饰符：**

|  |  |
| --- | --- |
| 内部类 | 可用修饰符 |
| 静态成员内部类 | final、abstract、public、private、protected、static |
| 非静态成员内部类 | final、abstract、public、private、protected |
| 局部内部类 | final、abstract |
| 匿名内部类 | 无修饰符 |

**内部类分类：**

语句块内：局部内部类、匿名内部类。

语句块外：非静态成员内部类、静态成员内部类。

**内部类class文件名：**

外部类名$内部类名.class

## 非静态内部类：

内部类不可以有静态成员

**内部类与外部类之间的成员互访:**

外部类中访问内部类的成员, 需要首先创建内部类对象.

即使是内部类的私有成员, 外部类一样可以进行访问.

修饰成员的访问限制符都可以修饰内部类：private、protected、public。

内部类编译后会单独产生一个class文件。

外部类 之内 创建内部类对象：

内部类名 引用名=new 内部类名(参数表);

外部类 之外 创建内部类对象：

外部类类名.内部类类名 引用变量 = new 外部类构造器.new 内部类构造器;

外部类类名 外部类对象引用 = new 外部类对象引用;

外部类类名.内部类类名 引用变量 = 外部类对象引用.new 内部类构造器;

内部类可以使用外部类的任何成员。

* 1. 成员名 //直接引用。
  2. 外部类名.this.成员名 //某成员在外部类和内部类中均有定义。

内部类中使用this调用的是内部类的成员。

内部类中使用外部类对象的this语法：外部类名.this.外部类中需要被访问的成员名;

## 局部内部类：

作用域与局部变量相同，只在语句块中有效，不需要访问权限修饰符。（只能在其有效位置访问或创建对象）

局部内部类可以访问所有外部类成员，不能访问同在一个局部代码块中的局部变量（final修饰的局部常量是可以访问的）。

静态方法中的局部内部类：只能访问外部类静态成员。

## 静态内部类 （静态嵌套类、顶级嵌套类）：

语法规则：比非静态内部类多个static。

不能访问外部类的非静态成员。

创建内部类的对象：

外部类类名.内部类类名 引用变量 = new 外部类类名.内部类构造器;

## 匿名内部类：

（用于只需使用一次的类，可节约资源。）

无构造方法，声明的同时创建对象。

只能使用外部类中的final常量。

声明：要么基于继承，要么基于实现接口。不能既继承 又 实现接口。

基于继承的匿名内部类：

声明方式：

new父类的构造器{

}

因为没有类名，所以无法编写构造器。初始化, 用非静态语句块实现.

类体中可以重写父类的方法，或创建自己独有的方法（因为没有类名，所以无法在外面调用独有的方法，只有自己用了；外面可以调用重写的方法。）

编译后仍然是一个独立的class文件，文件名为：外部类名$n 说明：n代表第几个匿名内部类。

基于实现接口的匿名内部类：

因为没有类名，所以不能是抽象类，必须实现接口中所有的方法。

对象初始化：因为没有构造函数，所以只能放在非静态语句块中。

## 内部接口：

扮演成员角色的接口。

内部类用class修饰，内部接口用interface来修饰。

无论是否用static来声明，都是静态的。

内部接口可以用修饰成员的修饰符来修饰，而普通接口不可以。

private修饰的内部接口只能由外部类中的某个内部类来实现。

接口中的内部接口：（在接口中定义接口）, 实现方式: class <类名> implements <外部接口>.<内部接口>

接口中的内部接口不能是private的。

外部接口外实现内部接口：

内部类需注意的问题：

* 1. 继承和普通类一样；
  2. 嵌套层次无限制；
  3. 可以实现接口；
  4. 非静态内部类：对这个类的成员有访问权限；不能包含静态成员变量。
  5. 静态内部类：只能声明静态成员变量；只可调用外部类的静态成员。
  6. 局部内部类：可以访问外部类的所有属性；可访问所在局部块中的final变量；不能用任何修饰符修饰；也不能用static修饰。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 可以是public, protected, private, 默认  可以是final, abstract   |  |  | | --- | --- | | s 或 this.s | 内部类对象的属性s | | a.this.s | 外层类对象的属性s | |

# 异常

## 异常的分类

|  |  |
| --- | --- |
| Throwable | |
| Error: 致命错误  系统内部错误或资源耗尽  比较严重的, 仅靠修改程序本身是不能回复执行的. | Exception: 非致命错误 |

**常见系统内置异常类**

|  |  |
| --- | --- |
| RuntimeException | 错误的的类型转换  数组下标越界  空指针访问 |
| IOException | 从一个不存在的文件中读取数据  越过文件结尾继续读  连接一个不存在的URL |

## 异常处理机制

捕获异常(try-catch)

嵌套try-catch块

使用多重catch语句

finally块的使用

异常处理的注意事项

### throw 和 throws的使用

## 自定义异常

|  |  |
| --- | --- |
| 1. 自定义异常类必须继承自Throwable类, 才能被视为异常类. 并且通常是继承Throwable类的子类Exception或Exception的子孙类. 其他地方与普通类语法相同.  2. 使用自定义异常类的步骤:  a. 创建自定义异常类  b.在方法中通过throw关键字抛出异常对象.  c.如果在当前抛出异常的方法中处理异常, 可以使用try-catch语句捕获并处理, 否则在方法的声明处通过throws关键字指明要抛出给方法调用的异常.  d. 在出现异常方法的调用者中捕获并处理异常. |  |
|  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| finall语句块：在离开try—catch之前执行。即使catch中有return语句，也要执行finally语句。  中断finall语句块的情况：   * 1. finally语句块里面本身产生异常；   2. 执行finally语句块的线程死亡；   3. finally语句块中执行了“System.exit(0);方法；     捕获异常一般是外界因素产生的，并且是可以恢复的。  继承自Error的类一般代表由硬件运行失败导致的严重错误，程序不能从Error中恢复。  RuntimeException类的子类通常是一些程序运行时错误引起的异常。  catch语句块：catch(异常类名 对象名)  {  对象名.方法名();  }  再次抛出异常：  显性再抛出：编写代码将catch语句捕获的异常再次抛出。  方法声明中用throws对于可能抛出的捕获异常必须列出，对未捕获异常则可以不列出。  方法中用throw抛出一个异常对象引用（异常类型必须与throws后面的类型相同，或是它的子类。）类型的确定，按引用类型。  何时需要throws来声明：   1. 调用一个会抛出“已检查异常”的方法。 2. 程序运行中发生错误，并且throws了“已检查异常”。   方法声明中定义抛出异常格式：public int 方法名(\*\*\*) throws 异常类名  抛出语句：throw new 异常类名（参数表）；  方法重写对抛出异常声明的约束：和访问限制修饰符一样。  自定义异常类：一般充当捕获异常的角色，故从Exception类或其它捕获异常类继承。（字节或间接继承Exception）  语法：class 类名 extends Exception 或其它捕获异常类{}  通常在自定义异常类中编写两个构造器：一个无参数构造器，另外一个是以字符串做参数的构造器。  字符串做参数的构造器可以调用父类中以字符串为参数的构造器。一般用来在有出错信息的情况下创建异常对象。  Exception类能被继承的常用方法：   |  |  | | --- | --- | | public void printStackTrace() | 打印异常调用栈的信息。 | | public String toString() | 返回该异常对象的字符串表示。 | | public String getMessage() | 返回异常对象中携带的出错信息。 |   异常的匹配：  1. 指定处理的异常类型若没有任何子类，这只能捕获指定的异常类型。  2. 指定处理的异常类型若有子类，则指定类型及其子类的异常都可以捕获。  3. Exception异常类是所有捕获异常类的父类，可以捕获所有异常类。  多个catch语句的顺序：  1. 若多个catch语句块中所指定的异常类型相互有派生关系，纳闷必须将子类型的异常写在前面。  2. 若为级别相同或者没有任何派生关系的异常类型，这顺序无所谓。  使用异常处理语句块的注意事项  1. 在try语句块中声明的变量, 只在当前try语句块中有效, 在其后的catch, finally语句块或其他位置都不能访问该变量. 但在try-catch-finally语句块之外声明的变量, 可以在try, catch 和 finally语句块中访问. |

## 异常处理原则

## 断言(1.4以后)

|  |
| --- |
| 加入检查参数, 用于调试  开启断言(enable assertions): java -ea myAppclass  关闭断言(disable assertions): java -da myAppclass |
| 断言：调试程序用的。  语法规则：   1. assert （逻辑表达式）   逻辑表达式返回boolean值，如果返回True，则继续执行；否则抛出AssertionError异常。   1. assert （逻辑表达式）: “信息表达式”   抛出异常的同时还打印信息表达式的值，信息表达式必须有返回值（任何返回值都可以，不能为void）。  开启断言编译：javac –source 1.4 源文件名 (5.0版本后不需要加参数)  关闭断言编译：javac –source 1.3 源文件名  开启断言执行代码：  默认运行情况下，断言是被禁用的。  手动开启断言执行：java –ea 类名 《或者》 java -enableassertions 类名  判断滥用断言的依据：关闭断言会影响程序的功能。  为避免滥用断言，需注意：   1. 断言表达式在使用前后程序的状态应该是一样的。 2. 断言失败是程序逻辑错误导致的，不应该对其进行捕获处理。而应该修改程序逻辑上的问题。   断言只用于开发阶段，发布后不会用到。 |

# 多线程

**线程的基本知识**

|  |
| --- |
| 不能并发执行的任务是不应该使用多线程来实现的.  一个线程其实就是一条独立的执行线索.  main方法被称作主线程. 主线程可以启动很多子线程. |

**定义自己的线程**

|  |
| --- |
| Java中的线程有两方面的意义:  1. 是一条独立执行的线索  2. 是java.lang.Thread类或其子类的对象(继承Thread类 或实现Runnable接口).  备注: Thread本身也实现了Runnable接口 |

**1. 继承Thread类**

|  |
| --- |
| 若一个类直接或间接继承了java.lang.Thread类, 则该类的对象便具有线程的能力.  采用此方法最重要的是要实现run()方法.  run()方法中的代码就是线程所要执行的描述. 线程一旦启动, run()方法中的代码将成为一条独立的执行线索.  run()方法也可作为一般的方法来被调用, 要注意的是: 直接调用run()方法并不产生新的线程线索.  run()方法可以重载, 但重载后的该方法, 不再具有成为一条执行线索的能力. |

**2. 实现Runnable接口**

|  |
| --- |
| 继承java.lang.Runnable, 并重写此接口中的唯一一个抽象方法: run()  Runnable接口的类的对象可以被同时传递给多个线程对象。 |

**3. 两种方式的比较**

|  |
| --- |
| 这个不用说了, 太简单了. |

**16.1.3 创建线程对象**

|  |
| --- |
| **继承Thread类方式**: Thread mt = new MyThread(); 备注: 更一般类创建对象相同. |
| **实现Runnable接口方式**:  实现Runnable接口的类, 其自身的对象并不是一个线程, 只是在该类中通过实现的run方法指出了线程需要完成的任务.  然而, 若想要得到一个线程, 必须创建Thread类或其之类的对象, 这时就需要使用Thread类的特定构造器来完成这个工作. |
| **常用Thread类的构造器**.   |  |  | | --- | --- | | public Thread() | 该构造器将构造一个新的线程对象, 该对象启动后将运行自身的run方法, 并且该对象具有默认的名称. | | public Thread(Runnable target) | 参数target为指定的Runnable实现类, 该构造器将构造一个新的线程对象, 当该对象启动后将执行指定target中的run方法, 该对象具有默认的名称. | | public Thread(Runnable target, String name) | 参数target为指定的Runnable实现类, 参数name为指定的名称,该构造器将构造一个新的线程对象, 当该对象启动后将执行指定target中的run方法, 该对象具有指定的名称. | | public Thread(String name) | 参数name为指定的名称, 该构造器将构造一个新的线程对象, 该对象启动后将运行自身的run方法, 并且该对象具有指定的名称. |   **实际操作步骤**:  1. 首先创建实现Runnable接口的类的对象,  2. 然后将此对象的引用的对象传递给Thread类构造器即可.  这种方式, 实际上是告诉线程对象要执行的任务(run方法)在哪里.  举例:  MyRunnable mr = new MyRunnable();  Thread t = new Thread(mr);  可以看出, 实现Runnable接口的类的对象只是指出线程需要完成的任务, 其本身并不是线程对象.  当然, 实现Runnable接口的类的对象可以被同时传递给多个线程的对象. 这意味着, 这几个线程对象启动后, 将执行完全相同的任务.  MyRunnable mr = new MyRunnable();  Thread t1 = new Thread(mr);  Thread t2 = new Thread(mr);  Thread t3 = new Thread(mr); |

**16.1.4启动线程**

|  |
| --- |
| 当线程对象创建后, 其还是一个普通的对象, 其还只是一个普通的对象, 并没有成为一条独立的执行线索, 想让其成为一条独立的执行线索就必须进行启动, 在没有启动的情况下, 可以像调用普通对象的方法一样调用线程对象中的任何可见方法. 调用run方法并不代表新建立了执行线索, run方法还是在调用其的线程中执行的. |
| 若要启动一个线程, 只需调用线程对象的start方法即可. eg:  MyRunnable mr = new MyRunnable(); //创建Runnable接口类的对象  Thread t1 = new Thread(mr); //创建Thread对象, 并把上一个对象的引用作为构造参数.  t1.start(); //启动线程 |
| 说明: 启动线程后, 相当于开辟了另一条执行线索, 与main方法对应的执行线索是相互独立的. |
| 注意: 线程在其生命周期中只能被启动一次, 如果线程被启动后再次调用其start方法进行启动, 则会抛出java.lang.IllegalThreadStateException异常.  MyRunnable mr = new MyRunnable();  Thread t1 = new Thread(mr);  t1.start();  t1.start(); //抛出异常  若要再次运行线程, 只需将其作为target传给新的线程对象即可, 并启动即可. |

**16.1.5 同时使用多个线程**

|  |
| --- |
| 两个没有约束逻辑的线程, 执行顺序是无规律的. |

## 16.2 线程的状态

|  |  |
| --- | --- |
| 新建状态 | 线程对象创建后, 即Thread类或其子类对象创建后, 便处于新建状态. 这种状态下, 线程对象还只是一个普通的对象, 没有成为一条独立执行的线索, 也不可能被线程调度程序调度. |
| 准备状态 | 新建状态下的线程被调用了start方法后就进入准备状态. |
| 运行状态 | 一旦处于准备状态的线程获取了CPU时间, 就进入了执行状态. 当在执行状态下, 线程随时都可能被调度程序调度回到准备状态. 另外, 线程在执行过程中, 由于需要等待某些必备的条件, 或由于运行了特定的方法, 可能进入 等待/阻塞 状态 |
| 等待/阻塞状态 | 由于某些原因不能继续执行, 可能包括: 睡眠, 阻塞, 挂起, 等待某些特定条件的满足等.   |  |  | | --- | --- | | 睡眠 | 正在执行的线程被调用了其sleep方法,将进入睡眠状态. | | 阻塞 | 正在执行的线程调用了阻塞的方法, 而正好又满足阻塞的情况, 则停止执行, 进入阻塞状态. 直到阻塞条件解除, 再进入准备状态. | | 挂起 | 正在执行的线程被调用了其suspend方法, 就会处于挂起状态, 直到调用了其resume方法才回到准备状态.  提示: 由于使用suspend方法挂起的线程很容易引发线程的死锁, 因此从J2SE1.2开始建议不要使用suspend, resume方法来手工调度线程, 在实际开发中不应该使用这个两个方法. | | 等待 | 正在运行的线程由于逻辑条件不满足, 自己调用wait方法进入等待状态, 直到收到通知的消息回到准备状态. | |
| 死亡状态 | 当run方法正确执行完毕, 或 由于发生异常而终止执行时, 线程就进入死亡状态.  进入死亡状态的线程可以当做普通的对象来使用, 例如调用其方法的成员变量, 当不能再次启动执行. 否则将像前面介绍的抛出java.lang.IllegalThreadStateException异常. |

## 16.3 线程的调度

|  |  |
| --- | --- |
| sleep(x)是静态方法, 一般用Thread来进行调用. | 直到规定的毫秒数满。 |
| I/O等待 | 等操作完成。 |
| wait() | 另一线程调用notify或notifyAll得到对象锁。 |

**16.3.1 睡眠**

|  |
| --- |
| 在线程执行的过程中, 调用sleep方法可以让线程睡眠一段指定的时间, 等指定时间达到后, 该线程自动苏醒, 并进入准备状态等待执行. |
| public static void sleep(long millis)throws InterruptedException  public static void sleep(long millis, int nanos)throws InterruptedException  说明:  1. millis代表睡眠的毫秒数, nanos代表睡眠的纳秒数, 但纳秒级的计时是十分不准确的, 不能用作时间基准.  2. 调用此方法时需要进行异常处理.  3. 由于是静态方法, 所以这两个方法不是与某个线程对象相关联的, 其可以出现在任何位置, 当执行到该方法时, 让执行此方法的线程进入睡眠状态. 也就是说: 哪个线程执行了sleep方法, 哪个线程去睡眠, 并不是调用特定线程对象的sleep方法.  4. 线程醒来将进入准备状态, 因此指定的时间是线程暂停执行的最小时间.  5. 由于sleep为静态方法，为了增强代码的可读性，一般使用类名Thread来调用：Thread.sleep(\*\*)。 |
| class MyRunnable implements Runnable{  public void run(){  System.out.println("打印完这行就睡觉");  try{Thread.sleep(1000);}catch(InterruptedException ie){}  System.out.println("我睡醒了");  }  } |

**16.3.2 线程的优先级**

|  |
| --- |
| 由于哪个线程先执行是没有保障的, 因此优先级策略不适合对线程进行细节调度, 而只适合进行宏观调控. |
| java中优先级用1-10的整数表示，数值越大，优先级越高；默认优先级为5。  子线程初始优先级与其父线程优先级相同。  在没有特别指定优先级的情况下, 主线程的优先级别为5.  子线程其初始优先级与其父线程优先级别相同. |
| 通过调用public final void setPriority(int newPriority) 方法改变优先级。 //参数newPriority表示要设置的优先级别, 应该是1到10之间的整数. 为便于记忆, Java中提供了3个常量来表示比较常用的优先级别.   |  |  | | --- | --- | | Thread.MAX\_PRIORITY | 最高优先级 | | Thread.NORM\_PRIORITY | 默认优先级 | | Thread.MIN\_PRIORITY | 最低优先级 |   例子:  MyThread1 t1 = new MyThread1();  t1.setPriority(Thread.MAX\_PRIORITY);  t1.start(); |

**16.3.3 线程的让步**

|  |
| --- |
| 使当前运行的线程让出CPU, 使其他线程得以执行, 包括两种方式:  1. 线程只是让出当前CPU资源, 至于将CPU让给谁, 不确定  2. 线程将给指定的线程让步, 指定的线程没有完成, 主动让步的线程绝对不恢复执行. |
| 1. 使用Thread.yield()方法  调用yield方法, 可以使当前正在运行的线程让出CPU, 回到等待状态. 但要注意, 该方法没有保障, 很可能线程回到准备状态后又立刻被调度再次进入运行状态, 也就是说yield让步不一定成功.  因此, 使用yield方法真正保证做到的是使正在运行的线程回到准备状态.  public static void yield() //静态方法, 与某个线程对象无关, 也就是说当正在执行的线程运行了此方法, 其将回到准备状态. |
| 2. 使用join方法  当一个线程必须等待另一个线程执行完毕才恢复执执行, 可以使用join方法. 顾名思义, join方法可以达到将两个线程合并为一个线程的效果.  publc final void join() throws InterruptedException  publc final void join(long millis) throws InterruptedException  publc final void join(long millis, int nanos) throws InterruptedException  1> 对于没有入口参数的join方法: 将使得调用此方法的线程一直等待到此方法所在的线程执行完毕才恢复执行, 效果上就像两个线程合并为一个线程.  2> 对于有参数的join方法: millis, nanos为指定等待的时间, 如果在指定的时间内, join方法所属的线程没有执行完毕, 则解除等待关系. |

**16.3.4 守护线程(后台线程)**

|  |
| --- |
| 比如线程调度, 内存管理等这些在后台运行的线程, 一般被称之为守护线程.  开发守护线程和普通的线程没有多大区别, 只要调用线程对象的setDaemon方法进行设置即可.  public final void setDaemon(boolean on)  参数on用于指定是否将指定的线程设置为守护线程,  若on=true则将其设为守护线程.  若on=false则将其设为前台(普通)线程.  例子:  Runnable mc = new MyCommon();  Runnable md = new MyCommon();  Thread tc = new Thread(mc);  Thread td = new Thread(md);  td.setDaemon(true);  tc.start();  td.start(); |
| 备注:  前台线程是保证执行完毕的, 而后台守护线程并不一定会执行完.  因为: Java运行时环境判断程序是否结束的标准是: "是否所有的前台用户线程都执行完毕了. 如果所有的前台线程执行完毕了, 程序退出." （即使守护线程没执行完）。 |

## 16.4 线程的同步

**16.4.1 同步方法简介**

|  |
| --- |
| 同步方法：synchronized关键字修饰的方法。  进入同步方法执行的线程将获得同步方法所属对象的锁，一旦对象被锁，其它线程就不能执行被锁对象的任何同步方法。  线程在执行同步方法之前，首先试图获得方法所属对象的锁，如果不能获得锁就进入对象的锁等待池等待，直到别的线程释放锁，其获得锁才能执行。  声明同步方法的基本语法：  synchronized 返回类型 方法名(参数表) [throws 异常序列]{  方法体  }  关键字synchronized只能标识方法，不能标识成员变量，不存在同步成员变量。  一个对象可以同时有同步方法和异步方法，只有进入同步方法执行才需要获得锁，每个对象只有一个锁。  若一个线程有多个同步方法，当某线程在访问其中之一时，其它线程不能访问该对象中的任何同步方法，但可以访问异步方法。  若线程获得锁后进入睡眠或进行让步，则将带着锁一起睡眠或让步，这种做法将严重影响等待锁的线程的执行，进而影响程序的整体性能。  同步方法退出时，锁将释放，其它等待的线程可以获得锁。  静态同步方法在执行前，线程要获取的是方法所在类的锁，同一时刻一个类只能有一个静态同步方法被访问。 |

**16.4.2 简单使用同步方法**

**16.4.3 线程同步调度的方法**

|  |  |
| --- | --- |
| wait() |  |
| wait(long timeout) |  |
| wait(long timeout, int nanos) |  |
| nofity() |  |
| notifyAll() |  |

**16.4.4 "生产者--消费者"案例的框架**

**16.4.5 "生产者--消费者"案例的实际运行**

**16.4.6 notity方法的使用**

|  |
| --- |
| 资源等待池中等待的线程是没有顺序的，notify方法知识从等待池中任意的选择一个线程唤醒恢复运行，具体选择哪个线程是没有保障的。  当有很多线程访问同一资源时，建议使用notifyAll方法来唤醒全部的等待线程，这样可以让所有的线程恢复工作。 |

**16.4.7 同步的语句块**

|  |
| --- |
| 基本语法：  synchronized(资源对象引用){  需要进行同步的代码  }  这里需要获得的是圆括号中资源对象引用指向对象的锁。  java的线程等待池：  1、调用wait方法进入的wait资源等待池；  2、等待获得锁的锁等待池。 |

**16.4.8 线程的死锁**

|  |
| --- |
| 当有多个访问者并发访问多个互斥的资源时，就有可能造成死锁。  java中的死锁：线程之间互相等待对方释放资源对象的锁，而每个线程有持有其它线程需要的锁，造成几个线程处在没有必要的永久等待当中，程序无法继续运行。  java中没有自动检测死锁并排除死锁的功能，开发人员设计中要执行注意避免。 |

**16.4.9 防止错误的使用wait, notify, notifyAll方法**

## 16.5 获取当前正在运行的线程

## 16.6 volatile关键字的含义与使用

## 17.1 线程池的使用

|  |
| --- |
| 结束线程的方法：   * 1. 自然消亡，从run()结尾处返回。   2. 调用stop()，强制退出线程，只用于Thread类。   线程检查：isAlive()  已开始运行，但还未完成的线程返回true。  join()：使当前执行的线程停下来，直至join所调用的那个线程结束，再恢复。 |

内存管理 与 垃圾收集

# I / O

## 以byte为单位的 字节流类

先转换成字节数组, 然后以字节为单位进行输入输出.

### InputStream类

|  |  |  |
| --- | --- | --- |
| 返回类型 | 方法名称 | 方法说明 |
| int | available() | 返回当前输入流的数据读取方法可以读取的有效字节数量 |
| Abasract int | read() | 从当前数据流中读取一个字节, 若已到达结尾, 则返回-1. |
| int | read(byte[] bytes) | 从当前输入流读取一定的byte数据, 并存放在数组中. 然后返回读取的byte数据的数量, 若已到达流结尾, 则返回-1. |
| int | read(byte[] bytes, int off, int len) | 从当前输入流读取一定的byte数据, 并存放在数组中指定位置, 然后返回读取的byte数据的数量, 若一到达流结尾, 则返回-1. |
| void | reset() | 将当前输入流重新定位到最后一次调用mark()方法时的位置. |
| void | mark(int readlimit) | 在当前输入流中做标记位置, 当调用reset()方法时将返回到该位置, 从标记位置开始, 再读入readLimit个字符为止, 这个标记都维持有效. |
| boolean | markSupported() | 测试当前输入流是否支持mark()和reset()方法. 只要其中一个不支持, 则返回false. |
| long | skip(long n) | 跳过和丢弃当前输入流中的n个字节数据. |
| void | close() | 关闭当前输入流, 并释放任何与之关联的系统资源. |

例子

|  |
| --- |
| public static void main(String[] args){  InputStream is = System.in; //控制台的输入流  try{  byte[] bs = new byte[50];  int i = is.read(bs);  System.out.println(new String(bs).trim());  is.close();  }catch(IOException e){  e.printStackTrace();  }  } |

#### InputStream 🡪 byte[]

|  |
| --- |
| /\*\*  \* **@param** inStream 输入流  \* **@return** 字节流  \* **@throws** Exception  \*/  **public** **static** **byte**[] readStream(InputStream inStream) **throws** Exception{  **byte**[] buffer = **new** **byte**[1024];  **int** len = -1;  ByteArrayOutputStream outStream = **new** ByteArrayOutputStream();  **while**((len = inStream.read(buffer))!=-1){  outStream.write(buffer, 0, len);  }  **byte**[] data = outStream.toByteArray();  outStream.close();  inStream.close();  **return** data;  } |

#### String与InputStream相互转换

|  |
| --- |
| 1.String to InputStream  String str = "String与InputStream相互转换";  InputStream in\_nocode = new ByteArrayInputStream(str.getBytes());  InputStream in\_withcode = new ByteArrayInputStream(str.getBytes("UTF-8")); |
| 2.InputStream to String  这里提供几个方法。  方法1：  **public** String convertStreamToString(InputStream is) {  BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(is));  StringBuilder sb = **new** StringBuilder();  String line = **null**;  **try** {  **while** ((line = reader.readLine()) != **null**) {  sb.append(line + "/n");  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **try** {  is.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **return** sb.toString();  }  方法2：  public String inputStream2String (InputStream in) throws IOException {  StringBuffer out = new StringBuffer();  byte[] b = new byte[4096];  for (int n; (n = in.read(b)) != -1;) {  out.append(new String(b, 0, n));  }  return out.toString();  }  方法3：  public static String inputStream2String(InputStream is) throws IOException{  ByteArrayOutputStream baos = new ByteArrayOutputStream();  int i=-1;  while((i=is.read())!=-1){  baos.write(i);  }  return baos.toString();  } |

### OutputStream类

|  |  |  |
| --- | --- | --- |
| 返回类型 | 方法名称 | 方法说明 |
| void | write(byte[] b) | 将byte[]数组中的数据写入到当前输入流 |
| void | write(byte[] b, int off, int len) | 将byte[]数组下标off开始的len长度的数据写入到当前输入流. |
| Abstract void | write(int b) | 写入一个byte数据到当前数量路 |
| void | flush() | 刷新当前输出流, 并强制写入所有缓存的字节数据. |
| void | close() | 关闭当前输出流, 并释放所有与当前输出流有关的系统资源 |

例子

|  |
| --- |
| public static void main(String[] args){  OutputStream out = System.out; //控制台的输入流  try{  byte[] bs = "love you Loli".getBytes();  out.write(bs);  out.close();  }catch(IOException e){  e.printStackTrace();  }  } |

## 以char为单位的 字符流类

如果只是简单的字符操作, 用char为单位的 字符流类 更加方便快捷.

### Reader类

|  |  |  |
| --- | --- | --- |
| 返回类型 | 方法名称 | 方法说明 |
| boolean | ready() | 判断此数据流是否准备好. |
| int | read() | 读入一个字符, 若已读到流结尾, 则返回值为-1. |
| int | read(char[]) | 读取一些字符到char[]数组内, 并返回所读入的字符的数量. 若已到达流结尾, 则返回-1. |
| Abstract int | read(char[] chars, int off, int len) | 读取一些字符到char[]数组下标off开始到off+len的位置, 并返回所读入的字符的数量. 若已读到流jiewie,则返回-1. |
| void | reset() | 将当前输入流重新定位到最后一次调用mark()方法的位置. |
| void | mark(int readLimit) | 在当前输入流中做标记位置, 当调用reset()方法时将返回到该位置. 从标记位置开始, 到再读入readLimit个字符为止, 这个标记都维持有效. |
| boolean | markSupported() | 测试当前输入流是否支持mark()和reset()方法, 只要有一个方法不支持, 就返回false(); |
| long | skip(long n) | 跳过参数n指定的字符数量, 并返回所跳过字符的数量 |
| Abstract int | close() | 关闭该流并释放与之关联的所有资源, 在关闭该流后, 再调用read(), ready(), mark(), reset()或skip()方法将抛出异常, 关闭已经关闭的流无效. |

### Writer类

|  |  |  |
| --- | --- | --- |
| 返回类型 | 方法名称 | 方法说明 |
| void | write(char[] cbuf) | 将字符数组的数据写入到字符输出流 |
| Abstract void | write(char[] cbuf, int off, int len) | 将字符数组从下标off开始向输出流写入len长度的数据. |
| void | write(int c) | 向输出流写入一个字符数据. |
| void | write(String str) | 向输出流写入一个字符串数据. |
| void | write(String str, int off, int len) | 向输出流写入一个字符串, 从off开始长度为len的长度. |
| Abstract void | flush() | 刷新当前输出流, 并强制写入所有缓冲区的字节数据. |
| Abstract void | close() | 向输出流写入缓冲区的数据, 然后关闭当前输出流, 并释放所有与当前输出流有关的系统资源. |

## 常用I/O处理方式

### 文件夹类

|  |
| --- |
| 概述:  File类用于封装系统的文件和目录的相关信息, 例如文件大小, 修改时间, 文件路径等. |
| 创建文件类   |  |  | | --- | --- | | File(String pathname);  eg: new File("D:/doc/help.txt"); | 通过指定的文件路径字符串来创建一个新的File实例对象 | | File(String parent, String child)  eg: new File("D:/doc/", "help.txt") | 根据指定的父路径字符串和子路径字符串创建File类的实例对象. | | File(File parent, String child)  new File(new File("D:/doc/"), "help.txt") | 根据指定的File类的父路径和字符串类型的子路径创建File类的实例对象. |   File类的常用方法   |  |  |  | | --- | --- | --- | | 返回 | 方法 | 方法说明 | | String | getName()9 | 获取文件名 | | Sting | getParent() | 获取文件的父路径字符串 | |  |  |  | |

### 文件的字节输入/输出流

创建文件字节输入流

|  |  |
| --- | --- |
| FileInputStream(File file) |  |
| FileInputStream(String name) | 根据文件的绝对路径或相对路径, 创建FileInputStream的实例对象 |

创建文件直接输出流

|  |  |
| --- | --- |
| FileOutputStream(File file) |  |
| FileOutputStream(String name) | 根据文件的绝对路径或相对路径, 创建FileOutputStream的实例对象 |

### 文件的直符输入/输出流

### 对象序列化

|  |
| --- |
| 概述:  基本类型的数据可以直接保存.  对象类型的数据必须实现Serializable接口, 然后在保存. |
| 序列化:  Person person = new Person();  FileOutputStream fout = new FileOutputStream("./person.txt");  ObjectOutputStream oout = new ObjectOutputStream(fout);  oout.writeObject(person); |
| 反序列化  FileInputStream fis = new FileInputStream("./person");  ObjectInputStream oi = new ObjectInputStream(fis);  Person person = (Person)oi.readObject(); |

## 标准I/O

### 读取控制台输入

### 将System.out转换为PrintWriter

### 标准I/O重导向

## Java输入/输出 类 和 接口

### 字节流

### 字符流

## 文件处理

### 创建文件和文件夹

|  |
| --- |
| **创建文件夹**  File dirPath = **new** File(localPath);  **if** (!dirPath.exists()) {  dirPath.mkdirs();  } |
| **创建文件**  File qcFile = **new** File(receiptFileName);  **if** (!qcFile.exists()) {  **try** {  qcFile.createNewFile();  } **catch** (IOException e) {  }  } |
| /\*\*  \* **文件重命名**  \*  \* **@param** fileName : 原始文件名(相对路径或绝对路径  \* **@param** newFileName : 新文件名(相对路径或绝对路径  \*/  **public** **void** renameFile(String fileName, String newFileName) {  File file = **new** File(fileName);  File newFile = **new** File(newFileName);  **if** (file.exists() && !newFile.exists()) {  file.renameTo(newFile);  }  } |
| /\*\*  \* 根据路径和文件名，创建新的文件。  \* **@param** path 文件所在目录绝对路径  \* **@param** fileName 文件名。  \* **@return**  \* **@throws** Exception  \*/  **public** File createFile(String path , String fileName) **throws** Exception {  File file = **new** File(path);    **if** (!file.exists()) {  file.mkdirs();  }  **if** (!file.isDirectory()) {  **throw** **new** Exception("Directory : " + path + " not exist.");  }    **if** (!file.canWrite()) {  **throw** **new** Exception("Directory : " + path + " can not write.");  }    String filePullPath = "";  **if** (path.endsWith(File.*separator*)) {  filePullPath = path + fileName;  }  **else** {  filePullPath = path + File.*separator* + fileName;  }    File report = **new** File(filePullPath);  **if** (report.exists()) {  **throw** **new** Exception("there is been a file name : " + fileName + " .");  }**else**{  **try** {  report.createNewFile();  } **catch** (IOException e) {  }  }  **return** report;  } |

### 文件操作

### 文件访问

File类

|  |
| --- |
| 一个File类的对象表示磁盘上的文件或目录.  File类的实例是不可变的, 也就是说, 一旦创建, File对象表示的抽象路径名将永不改变. |

文件 输入 / 输出

创建文件

|  |
| --- |
| File f1 = **new** File("C:\\test");  **if**(f1.exists()){  System.*out*.println("该文件夹已经存在");  }**else**{  f1.mkdir(); //创建test目录  }  File f2 = **new** File("C:\\test\\a.txt");  **try** {  f2.createNewFile();  } **catch** (IOException e) {  e.printStackTrace();  } |

#### 遍历文件夹

|  |
| --- |
| **import** java.io.File;  **import** java.util.ArrayList;  **public** **class** Test{  /\*\*  \* 是否测试, 值为true, 调用测试方法构造数据; 值为false, 调用srvice获取真实数据  \*/  **public** **static** **boolean** *isTest* = **true**;    **private** **static** ArrayList *filelist* = **new** ArrayList();    **public** **static** **void** main(String[] args) {    **long** a = System.*currentTimeMillis*();  *refreshFileList*("c:\\pic");  System.*out*.println(System.*currentTimeMillis*() - a);  }  **public** **static** **void** refreshFileList(String strPath) {  File dir = **new** File(strPath);  File[] files = dir.listFiles();    **if** (files == **null**)  **return**;  **for** (**int** i = 0; i < files.length; i++) {  **if** (files[i].isDirectory()) {  *refreshFileList*(files[i].getAbsolutePath());  } **else** {  String strFileName = files[i].getAbsolutePath().toLowerCase();  System.*out*.println(strFileName);  *filelist*.add(files[i].getAbsolutePath());  }  }  }  } |

### 写文件

|  |
| --- |
| 在java.io包中定义的输出流操作类都适合对磁盘文件内容的读取操作.  例如: 直接用于磁盘文件写操作的类有FileOutputStream, FileWriter等  他们可以通过(绑定)一个文件对象创建输出流对象, 也可以直接指定磁盘文件名创建输出流对象, 通过输出流对象来实现文件的写操作. |
| **FileOutputStream 字节输出流** 写文件  用于写入字节数据流到文件中的类  在java.io包中还定义了一些用于创建顺序字节数据输出流对象的类, 如:ByteArrayOutputStream, BufferedOutputStream, DataOutputStream, ObjectOutputStream, PipedOutputStream等.   |  | | --- | | **public** **static** **void** main(String[] args) {  **try** {  OutputStream fos = **new** FileOutputStream("C:\\test\\a.txt");  String str = "sdaffdasf";  **byte**[] words = str.getBytes();  fos.write(words, 0, words.length);  fos.close();  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  } | |
| **FileWriter 字符输出流** 写文件  FileWriter是用于 字符数据流到文件中的类, FileWriter是OutputStreamWriter类的直接子类  在java.io包中还定义了一些用于创建顺序字符数据输出流对象的类, 如:ByteArrayWriter, BufferedWriter, StringWriter, PipedWriter等.   |  | | --- | | **public** **static** **void** main(String[] args) {  **try** {  FileWriter fw = **new** FileWriter("C:\\test\\a.txt");  BufferedWriter bw = **new** BufferedWriter(fw);  bw.write("我要认真学习, 呵呵");  bw.flush();  bw.close();  } **catch** (IOException e) {  e.printStackTrace();  }  } | |

### 读文件

|  |  |
| --- | --- |
| **FileInputStream 字节输入流** 读文件  在java.io包中还定义了一些用于创建字节数据输出流对象的类, 如:ByteArrayInputStream, BufferedInputStream,DataInputStream等.   |  | | --- | | **public** **static** **void** main(String[] args) {  String path = "C:\\test\\a.txt";  **try** {  File file = **new** File(path);    //判断文件是否存在, 或是文件还是文件夹  **if**(!file.exists() || file.isDirectory()){  System.*out*.print("该文件不存在, 或是文件夹");  }    //根据文件流对象, 创建输入流  FileInputStream fis = **new** FileInputStream(file);  **byte**[] buf = **new** **byte**[1024];  StringBuffer sb = **new** StringBuffer(); //创建一个字符串    //按 单个字节方式循环从输入流fis中读文件数据,读到文件末尾(返回-1)结束读取操作.  **while**((fis.read(buf))!= -1){  sb.append(**new** String(buf));  buf = **new** **byte**[1024];// 重新生成, 避免和上次读取的数据重复  }    System.*out*.println(sb.toString()); //打印输出文件内容  fis.close();//关闭流    } **catch** (IOException e) {  e.printStackTrace();  }  } | |
| **FileReader 字符输入流**  读文件  在java.io包中还定义了一些用于创建顺序字符数据输出流对象的类, 如:LineNumberReader, CharArrayReader, BufferedReader, StringReader, PipedReader等. |

#### 随机文件的读写

## 对文件 及 文件夹 进行操作

## 根据文件目录和文件名, 返回文件全路径

|  |
| --- |
| /\*\*  \* 根据文件目录和文件名, 返回文件全路径  \*  \* **@param** path: 文件目录  \* **@param** separator: 文件分隔符  \* **@param** fileName: 文件名  \* **@return**  \*/  **public** **static** String fileFullPath(String path, String separator, String fileName){    String fileFullPath = **null**;  **if** (path.endsWith(separator)) {  fileFullPath = path + fileName;  }  **else** {  fileFullPath = path + separator + fileName;  }  **return** fileFullPath;  } |

### 复制文件

|  |
| --- |
| Stream sfpath = "./sfile.txt";  Stream dfpath = "./dfile.txt";  File sFile = new File(sfpath);  File dFile = new File(dfpath);  try{  FileInputStream fis = new FileInputStream(sFile);  FileInputStream fout = new FileInputStream(dFile);  }catch(FileNotFoundException e){  e.printStacTrace();  }  byte[] date new byte[512];  int rs = -1;  while((rs=fis.read(date))>0){  fout.write(date, 0, rs);  }  fout.close();  fis.close(); |

### 移动文件(删除原文件)

|  |
| --- |
| /\*\*  \* 移动文件(删除原文件)，移动成功返回true，否则返回false  \* **@param** srcFileDir 源文件夹路径  \* **@param** srcFileName 源文件  \* **@param** destFileDir 目标文件夹路径  \* **@param** destFileName 目标文件  \* **@return**  \*/  **public** **static** **boolean** moveFile(String srcFileDir, String srcFileName, String destFileDir, String destFileName) {  **boolean** flag = **false**;  File srcFile = **new** File(srcFileDir + File.*separator* + srcFileName);  **if**(!srcFile.exists()) { //如果源文件不存在，直接返回false  **return** flag;  }    File destFolderFile = **new** File(destFileDir);//目标文件夹  **if**(!destFolderFile.exists()) {//如果文件夹不存在，则创建  destFolderFile.mkdirs();  }  File destFile = **new** File(destFileDir + File.*separator* + destFileName);  **try** {  FileCopyUtils.*copy*(srcFile, destFile);  //删除源文件  *deleteFile*(srcFileDir, srcFileName);  flag = **true**;  } **catch** (IOException e) {  **return** **false**;  }  **return** flag;  } |

### 复制文件夹

|  |
| --- |
| private static void copy(File[] s, File d){  if(!d.exists()){ //如果文件夹不存在  d.mkdir(); //建立新的文件夹  }  for(int i=0; i<s.length; i++){  if(s[i].isFile()){ //如果是文件类型, 就复制文件  try{  FileInputStream fis = new FileInputStream(s[i]);  FileOutputStream out = new FileOutputStream(new File(d.getPath() + File.separator + s[i].getName()));  int count = fis.available();  byte[] data = new byte[count];  if((fis.read(data))!=-1){  out.write(data); //复制文件内容  }  out.close(); //关闭输出流  fis.close(); //关闭输入流  }catch(Exception e){  e.printStackTrace();  }  }  if(s[i].isDirectory()){ //如果是文件类型  File des = new File(d.getPath() + File.separator + s[i].getName());  des.mkdir(); //在目标文件夹中创建相同的文件夹  copy(s[i].listFiles(), des); //递归调用方法本身.  }  }  } |

### 删除单个文件

|  |
| --- |
| /\*\*  \* 删除存在的文件  \* **@param** fileDir fileName文件所在的文件夹  \* **@param** fileName 文件名  \*/  **public** **static** **void** deleteFile(String fileDir, String fileName) {  **if**(fileName != **null**) {  File file = **new** File(fileDir + File.*separator* + fileName);  **if**(file.exists()) {//如果文件存在则删除  file.delete();  }  }  } |

### 递归删除文件夹

|  |
| --- |
| /\*\*  \* 删除目录, 无论目录下有多少子目录都会递归删除  \* **@param** delDir  \* **@return**  \*/  **public** **static** **boolean** deleteDir(String delDir) {  **if**(delDir==**null**) **return** **false**;    File file = **new** File(delDir);  **if** (!file.isDirectory()) {  **return** file.delete();  }    String[] filelist = file.list();  **for** (**int** i = 0; i < filelist.length; i++) {  File delfile = **new** File(*fileFullPath*(delDir, File.*separator*, filelist[i]));  **if** (!delfile.isDirectory()) {  delfile.delete();  } **else** **if** (delfile.isDirectory()) {  *deleteDir*(*fileFullPath*(delDir, File.*separator*, filelist[i]));  }  }  **return** file.delete();  } |

### 分行写入文件

## 对象 序列化 与 反序列化

|  |
| --- |
| 对象序列化是: 把对象写到一个输出流中  对象反序列化: 从一个输入流中读取一个对象. |
| 对象序列化的用途:  1. 把 对象的字节序列 永久地保存到硬盘上, 通常存放在一个文件中.  2. 在网络上传送对象的字节序列. |
| java.io.ObjectOutputStream代表 对象输出流, 它的writeObject(Object obj)方法 可以对参数指定的obj对象进行序列化, 把得到的字节序列写到一个目标输出流中.  java.io.ObjectInputStream代表 对象输入流, 它的readObject()方法从一个源输入流中读取字节序列, 再把它们反序列化为一个对象, 并将其返回.  只有实现了Serializable 或 Externalizable接口的类的对象才能被序列化.  Externalizable接口继承自Serializable接口, 实现Externalizable接口的类完全由自身来控制序列化的行为.  而仅实现Serializable接口的类可以采用默认的序列化方式. |
| 对象序列化的步骤:  1. 创建一个对象输出流, 它可以包装一个其他类型的目标输出流, 如文件输出流;  2. 通过对象输出流的writeObject()方法写对象.  对象反序列化的步骤:  1. 创建一个对象输入流, 它可以包装一个其他类型的源输入流, 如文件输入流.  2. 通过对象输入流的readObject()方法读取对象. |
| ObjectOutputStream只能对实现了Serializable接口的类的对象进行序列化.  默认情况下, ObjectOutputStream按照默认方式序列化, 这种序列化方式仅仅对对象的非rtansient的实例变量进行序列化, 而不会序列化对象的transient的实例变量, 也不会序列化静态变量.  但ObjectOutputStream那招默认方式反序列化时, 具有如下特点:  1. 如果  **TBD** |
| 例子:   |  | | --- | | **class** Student **implements** Serializable{  **private** String name;  **private** **int** age;    **public** Student(String name, **int** age){  **this**.name = name;  **this**.age = age;  }  **public** String toString(){  **return** name + ", " + age;  }  }  **public** **class** SavaObject {  **public** **static** **void** main(String[] args) **throws** Exception {  FileOutputStream fos = **new** FileOutputStream("c:\\test\\student.txt");  ObjectOutputStream out = **new** ObjectOutputStream(fos);    //序列法对象  Student stu = **new** Student("Loli", 23);  System.*out*.print(stu.toString());  out.writeObject(stu);  out.close();    //反序列化  FileInputStream fis = **new** FileInputStream("c:\\test\\student.txt");  ObjectInputStream in = **new** ObjectInputStream(fis);  Student stu1 = (Student)in.readObject();  System.*out*.print(stu1);  in.close();    } | |

## 二进制文件 的 读写 操作

|  |
| --- |
| java.io.DataInputStream 和 java.io.DataOutputStream 可提供一些对java基本数据类读写的方法, 像读写int, double和boolean等的方法, 由于java的数据类型大小是规定要的, 所以在读写基本数据类型时, 就不用担心平台间数据大小不同的问题. |

## I/O streams的运用

# 文件读写例子

## 以行为单位读取文件

|  |
| --- |
| /\*\*  \* 以行为单位读取文件  \*  \* **@param** file  \*/  **public** **static** **void** readline(String file) {  **try** {  FileReader fr = **new** FileReader(file);  BufferedReader br = **new** BufferedReader(fr);  **while** ((br.read()) != -1) {  String readoneline = br.readLine();  System.*out*.println(readoneline);  }  br.close();  fr.close();  } **catch** (IOException e) {  e.printStackTrace();  }  } |

## 以字符为单位读取文件(当文件中有中文会乱码)

|  |
| --- |
| /\*\*  \* 以字符为单位读取文件(当文件中有中文会乱码)  \*  \* **@param** file  \*/  **public** **static** **void** readfile(String file) {  **try** {  FileReader fr = **new** FileReader(file);  **int** ch = 0;  **while** ((ch = fr.read()) != -1) {  System.*out*.println((**char**) ch);  }  } **catch** (IOException e) {  e.printStackTrace();  }  } |

## 批量修改文件名

|  |
| --- |
| import java.io.File;  import java.io.IOException;  /\*\*  \* 假设文件名为"C123\_the title of article.pdf"，我们要把这类文件批量更改为  \* "the title of article.pdf"的形式。  \*  \* @author huzhyi21 {huzhyi21@163.com}  \*  \*/  public class FileRename {  public static void main(String[] args) throws IOException {  java.io.File file = new java.io.File("G:/Test");  String dirPath = file.getAbsolutePath();//目录路径  if (file.isDirectory()) {  File[] files = file.listFiles();//获取此目录下的文件列表  long starttime = System.currentTimeMillis();  for (File fileFrom : files) {  String fromFile = fileFrom.getName();//文件名  String toFileName;  int index;  index = fromFile.indexOf("\_");  fromFile = fromFile.substring(index + 1);//截取\_后面的String  if (index != -1)//防止有的文件名没有\_  {  toFileName = dirPath + "\\" + fromFile;  File toFile = new File(toFileName);  if (fileFrom.exists() && !toFile.exists()) {  //开始更名  fileFrom.renameTo(toFile);  }  }  }  long endtime = System.currentTimeMillis();  System.out.println("Time:" + new Long(endtime - starttime));//耗时  }  }  } |

## 各种删除文件(未测试)

|  |
| --- |
| 删除文件夹包括以下几种情况：  文件夹里的文件，文件夹里面的子文件夹（有文件），文件夹里面的子文件夹（空文件夹）。所以程序要针对这三种情况写。  import java.io.File;  public class Test {  public static void main(String args[]) {  Test t = new Test();  delFolder("c:/bb");  System.out.println("deleted");  }  // 删除文件夹  // param folderPath 文件夹完整绝对路径  public static void delFolder(String folderPath) {  try {  delAllFile(folderPath); // 删除完里面所有内容  String filePath = folderPath;  filePath = filePath.toString();  java.io.File myFilePath = new java.io.File(filePath);  myFilePath.delete(); // 删除空文件夹  } catch (Exception e) {  e.printStackTrace();  }  }  // 删除指定文件夹下所有文件  // param path 文件夹完整绝对路径  public static boolean delAllFile(String path) {  boolean flag = false;  File file = new File(path);  if (!file.exists()) {  return flag;  }  if (!file.isDirectory()) {  return flag;  }  String[] tempList = file.list();  File temp = null;  for (int i = 0; i < tempList.length; i++) {  if (path.endsWith(File.separator)) {  temp = new File(path + tempList[i]);  } else {  temp = new File(path + File.separator + tempList[i]);  }  if (temp.isFile()) {  temp.delete();  }  if (temp.isDirectory()) {  delAllFile(path + "/" + tempList[i]);// 先删除文件夹里面的文件  delFolder(path + "/" + tempList[i]);// 再删除空文件夹  flag = true;  }  }  return flag;  }  } |

# socket

|  |
| --- |
| Server  **package** com.cdc.util.socket;  **import** java.io.BufferedReader;  **import** java.io.IOException;  **import** java.io.InputStreamReader;  **import** java.io.OutputStreamWriter;  **import** java.io.Reader;  **import** java.io.Writer;  **import** java.net.ServerSocket;  **import** java.net.Socket;  **public** **class** SocketServer {  **private** **static** **final** **int** ***port*** = 8899;  **public** **static** **void** main(String args[]) **throws** IOException {  // 为了简单起见，所有的异常信息都往外抛  **int** port = 8899;  // 定义一个ServerSocket监听在端口8899上  ServerSocket server = **new** ServerSocket(port);  **while** (**true**) {  // server尝试接收其他Socket的连接请求，server的accept方法是阻塞式的  Socket socket = server.accept();  // 每接收到一个Socket就建立一个新的线程来处理它  **new** Thread(**new** Task(socket)).start();  }  }  /\*\*  \* 用来处理Socket请求的  \*/  **static** **class** Task **implements** Runnable {  **private** Socket socket;  **public** Task(Socket socket) {  **this**.socket = socket;  }  **public** **void** run() {  **try** {  handleSocket();  } **catch** (Exception e) {  e.printStackTrace();  }  }  /\*\*  \* 跟客户端Socket进行通信  \*  \* **@throws** Exception  \*/  **private** **void** handleSocket() **throws** Exception {  Reader br = **new** InputStreamReader(socket.getInputStream(), "GBK");  **char** chars[] = **new** **char**[64];  **int** len;  StringBuilder sb = **new** StringBuilder();  **while** ((len = br.read(chars)) != -1) {  sb.append(**new** String(chars, 0, len));  }  System.***out***.println("服务端输出: " + sb);  // 读完后写一句  Writer writer = **new** OutputStreamWriter(socket.getOutputStream(), "UTF-8");  writer.write("你好，客户端。");  writer.flush();  writer.close();  br.close();  socket.close();  }  }  } |
| Client  **package** com.cdc.util.socket;  **import** java.io.InputStreamReader;  **import** java.io.OutputStreamWriter;  **import** java.io.Reader;  **import** java.io.Writer;  **import** java.net.Socket;  **public** **class** SocketClient {  **public** **static** **void** main(String args[]) **throws** Exception {  // 为了简单起见，所有的异常都直接往外抛  String host = "127.0.0.1"; // 要连接的服务端IP地址  **int** port = 8899; // 要连接的服务端对应的监听端口  // 与服务端建立连接  Socket client = **new** Socket(host, port);  // 建立连接后就可以往服务端写数据了  Writer writer = **new** OutputStreamWriter(client.getOutputStream(), "GBK");  writer.write("你好，服务端。");  writer.flush();  client.shutdownOutput(); //下面有解释  // 写完以后进行读操作  // BufferedReader br = new BufferedReader(new  // InputStreamReader(client.getInputStream(), "UTF-8"));  // //设置超时间为10秒  // client.setSoTimeout(10\*1000);  // StringBuffer sb = new StringBuffer();  // String temp;  // int index;  // try {  // while ((temp=br.readLine()) != null) {  // if ((index = temp.indexOf("eof")) != -1) {  // sb.append(temp.substring(0, index));  // break;  // }  // sb.append(temp);  // }  // } catch (SocketTimeoutException e) {  // System.out.println("数据读取超时。");  // }  Reader br = **new** InputStreamReader(client.getInputStream());  **char** chars[] = **new** **char**[64];  **int** len;  StringBuilder sb = **new** StringBuilder();  **while** ((len = br.read(chars)) != -1) {  sb.append(**new** String(chars, 0, len));  }  System.***out***.println("客户端输出: " + sb);  writer.close();  br.close();  client.close();  }  } |
| 客户端程序在打印出最有一个字节后不再往下执行，没有执行其后面的System.out.println("Received: " + new String(data));这行代码，这是因为read（）方法已经将数据读完，没有数据可读，但又没有返回-1，因此在此处产生了阻塞。这便造成了TCP Socket 通信的死锁问题。  问题就出现在read（）方法上，这里的重点是read（）方法何时返回-1，在一般的文件读取中，这代表流的结束，亦即读取到了文件的末尾，但是在Socket套接字中，这样的概念很模糊，因为套接字中数据的末尾并没有所谓的结束标记，无法通过其自身表示传输的数据已经结束，那么究竟什么时候read（）会返回-1呢？答案是：当TCP通信连接的一方关闭了套接字时。  再次分析改过后的代码，客户端用到了read（）返回-1这个条件，而服务端也用到了，只有二者有一方关闭了Socket，另一方的read（）方法才会返回-1，而在客户端打印输出前，二者都没有关闭Socket，因此，二者的read（）方法都不会返回-1，程序便阻塞在此处，都不往下执行，这便造成了死锁。  反过来，再看书上的给出的代码，在客户端代码的while循环中，我们的条件是totalBytesRcvd < data.length，而不是(bytesRcvd = in.read())!= -1，这样，客户端在收到与其发送相同的字节数之后便会退出while循环，再往下执行，便是关闭套接字，此时服务端的read（）方法检测到客户端的关闭，便会返回-1，从而继续往下执行，也将套接字关闭。因此，不会产生死锁。  那么，如果在客户端不知道反馈回来的数据的情况下，该如何避免死锁呢？Java的Socket类提供了shutdownOutput（）和shutdownInput（）另个方法，用来分别只关闭Socket的输出流和输入流，而不影响其对应的输入流和输出流，那么我们便可以在客户端发送完数据后，调用shutdownOutput（）方法将套接字的输出流关闭，这样，服务端的read（）方法便会返回-1，继续往下执行，最后关闭服务端的套接字，而后客户端的read（）方法也会返回-1，继续往下执行，直到关闭套接字。  客户端改变后的代码部分如下：  out.write(data); // Send the encoded string to the server  socket.shutdownOutput(); |

# JDK 5.0 新特性

## 1. 泛型

|  |
| --- |
| 泛型的本质: 参数化类型, 也就是说操作的数据类型被指定为一个参数. 这种参数类型可以用在类, 接口和方法的创建中, 分别成为泛型类, 泛型接口, 和 泛型方法.  没有泛型的情况下, 通过对类型Object的引用来实现参数的"任意化", "任意化"带来的缺点是要做显示的强制类型转换, 而这种转换是要求开发者对实际参数类型可以预知的情况下进行的.  泛型的好处是在编译时检查类型安全, 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.  泛型是具有占位符(类型参数)的类, 结构, 接口, 方法. |
| 泛型的规则限制:  1. 泛型的类型参数只能是类类型(包括自定义类), 不能是简单类型.  2. 泛型的参数类型可以使用extends语句, 例如<T extends superclass>, 习惯上称为"有界类型". |
| 限制泛型的可用类型 |
| 类型通配符   |  | | --- | | **public** **class** Gen<T> {  **private** T foo;  **public** T getFoo() {  **return** foo;  }  **public** **void** setFoo(T foo) {  **this**.foo = foo;  }  **public** **static** **void** main(String[] args) {  //使用类型通配符  //使用extends关键字, 表示向下限制, 表示?只能接受StringBuffer及的子类类型  Gen <? **extends** List> foo = **null**;  foo = **new** Gen<ArrayList>();  foo = **new** Gen<LinkedList>();    //也可以用super关键字, 表示向上限制, 表示?只能接受StringBuffer及上层的父类类型  Gen<? **super** List> foo2 = **new** Gen<Object>();    //也可以指明类型  Gen<String> ge3 = **new** Gen<String>();  }  } |   注意: Gen<?> 等价于 Gen<? extends Object> |
| 使用泛型的优点  1. 提高性能: 在运行的时候不用再进行类型转换;  2. 安全性: 通过使用泛型, 可以创建类型安全的数据类型, 只要编译的时候不出错就能保证运行的时候不会出错. |

## 2. 增强的for循环

|  |
| --- |
| for(Object o: objects) |
| 例子:  for(String str : array){  System.out.println(str);  } |

## 3. 自动 装箱 / 拆箱

## 4. 枚举

|  |  |
| --- | --- |
| 语法 | [访问限制修饰符] enum <枚举类型名称> {<枚举选项列表>}  1. 每个枚举类型相当于一个类, 因此枚举类型的声明应该放在类声明的位置,使用类的访问限制修饰符. 如果把枚举类型当做成员内部类来声明, 就可以使用成员的访问修饰符.  2. 枚举类型的名称应该满足Java中类名的规则, 每个单词的第一个字母大写, 其它字母小写;  3. 枚举的选项由于表示的是常量的含义, 因此应该满足常量的命名规则, 所有字母大写., |
| 定义 | public enum Week{MON, TUE, WED, THU, FRI, SAI, SWU} |
| 使用 | Week.SUN  String[] week = Week.values(); |
| 枚举类 | 枚举值都是所属枚举类的一个实力. |
|  |  |

## 5. 静态导入

|  |
| --- |
| 没使用静态导入: 要使用静态成员(方法和变量), 我们必须给出提供这个静态成员的类,  使用静态导入:　可以使被导入类的所有静态变量和静态方法在当前类直接可见, 使用这些静态成员无需再给出他们的类名. |

## 6. 可变参数(变长参数)

|  |
| --- |
| 注意:  1. 一个方法只能有一个变长参数  2. 如果方法的参数不止一个, 变长参数必须是最后一个. |
| public static void sortAndPrint(int... entrys){ //对收到的整型数组进行排序  Array.sort(entrys);  } |

## 注解(Annotation)

|  |  |  |
| --- | --- | --- |
| Annotation对程序运行没有影响, 它的目的在于对编译器或或分析工具说明程序的某些信息, 我们可以在包, 类 或 方法 等 上加上Annotation, 每一个Annotation对应于一个实际的Annotation类型.  Annotation可以包含有任意的参数:   |  | | --- | | @Author("MyName")  class myClass(){} | | @SuppressWarnings("unchecked")  voidmyMethod(){} | |
| @Deprecated  表示想编译器说明某个方法已经不建议使用,  如果开发人员在一个程序中使用了被@Deprecated标示的方法, 变量, 类 的时候, 编译器会产生警告信息(warning) |
| @Override  表示向编译器说明某个方法必须是重写父类中的方法.  在编译程序时, 如果发现被@Override标示的方法并非重写父类中的方法, 就会报告错误.  作用: 如果一个方法被标示了@Override, 但是其父类中没有这个方法时, 编译器将会报错. |
| @SuppressWarnings  表示: 对编译器说明某个方法中若有警告信息(warnings), 则加以抑制, 不用在编译完成后出现警告. |
| java Language Sprcification列出了两种类型: "deprecation" 和 "unchecked", 每个编译器的警告都属于穷在两个一个类型.  unchecked警告发生在使用非泛型的旧代码交付的泛型集合类时.  但需要忽略警告时, 可使用@SuppressWarnings("unchecked", "deprecation") |

## 反射

# 集合

|  |
| --- |
| **集合 概述**  数组的长度是固定的, 集合的长度是可变的.  数组用来存放基本类型的数据, 集合用来存放类对象的引用.  容器中的元素都是Object, 从容器中取得元素时, 必须把它转换成原来的类型. 用泛型就不用转  Collection接口是List接口和Set接口的父接口 |
| Collection: 是List和Set的父接口, 通常情况下不能直接使用. |
| **List 列表** : 允许存放重复对象, 按照对象插入顺序排序(有序)  以线性方式存储, 灭有特定的顺序, 只有一个开头和结尾. 它与根本没有顺序的数学中的集合是不同的.   |  | | --- | | **ArrayList**: 实现了List接口, 采用数组结构保存对象,  数组结构的特点: 根据索引随机访问快, 插入删除慢 | | **LinkedList**: 实现了List接口, 采用链表结构保存对象,  链表结构的特点: 根据索引随机访问慢, 插入删除快  适合于对 堆栈 队列 双头队列 进行操作. |   客户化排序List集合   |  | | --- | | 通过实现java.lang.Comparable接口 实现客户化排序  通过实现java.util.Comparator接口 实现客户化排序 | |
| **Set**: 不允许存放重复对象, 类似于向口袋中放东西, 不按特定的顺序排序.(无序)  每个具体的Set实现类依赖添加的equals()方法来检查独一性.  Set接口没有引入新方法, 所以Set接口就是一个Collection, 只不过其行为不同.   |  | | --- | | **HashSet类**  即散列表. 按照法希算法计算出法系码作为存取和比较的依据. 具有很好的存取和查询功能.  当向集合中存储对象时, 先生成法系吗, 如果生成的法系吗与已有对象的哈希码相同, 则调用equals()方法对两个对象进行比较; 如果哈希码不相等, 则根据哈希码在集合中寻找相应的位置来存储对象. | | **TreeSet类**  当要从集合中以有序的方式插入 和 抽取元素时, 就可以使用TreeSet实现. 添加到TreeSet的元素必须是可排序的. | |
| **Map**: 以键值对的形式存放对象, 其中键对象不可重复, 值对象可以重复. 按照自身内部的排序规则排序  Map不是继承自collection.  在Map中插入 删除 定位元素, HashMap是最好的选择, 但如果要按自然顺序 或 自定义顺序遍历键, 那么TreeMap是更好.  使用HashMap要求添加的键类 明确定义了hashCode()和equals()的实现.   |  | | --- | | **HashMap类**  是非线程安全的容器, 允许null的键值对,  key是无序存放且不能重复的, 它也是通过哈希码值来保证key不重复的. 如果要加入的键值对和HashMa中健值对的key是相同的, 就会将这个集合中的key所对应的value值覆盖. | | **TreeMap类** | |

## Hash表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| System.out.println()方法在打印引用时，若引用不为空，则首先调用引用指向对象的toString方法获取字符串，然后打印字符串内容。  哈希码（散列码）：  hashCode方法：用来提供所在对象的哈希码。根据对象的不同，哈希码的值也有所不同。  一般每定义一个新的类，都要为其重写一个适合的hashCode方法。  Hash是一种数据结构, 用来查找对象.  哈希的工作原理：   1. 进入哈希存储前，首先调用对象的hashCode方法获取哈希码，定位对象所在的哈希桶。 2. 在哈希桶内部，所有的哈希码相同的不同对象是按照线性表的方式存储的。   重写equals方法：  任何引用，只要不为null，那么对其自身使用equals方法永远返回True。  equals方法工作原理：   1. 测试两个对象是否指向同一对象；如果指向同一对象，直接返回True。 2. 测试传递进来的引用是否为null；如果为null，直接返回False。 3. 测试传递进来的引用是否通过自己类型的instanceof方法；如果不通过，直接返回False。 4. 将传递进来的引用强制类型转换为自己的类型，测试相关的成员是否等价；成立则返回True，否则返回False。   重写equals方法注意：   1. 必须为public类型。 2. 方法的入口参数必须为object类型。 3. 重写equals方法，就必须重写hashCode方法。否则两个等价的对象可能得不到相同的哈希码。   重写hashCode方法：   1. java应用程序执行期间，同一对象多次使用hashCode方法时，在对象上参与equals比较的成员没有被修改的情况下，必须一致地返回相同的整数。 2. 某一引用程序的一次执行到同一引用程序的另外一次执行，该整数无需保持一致。 3. 如果两个对象各自调用hashCode方法产生不同那个的结果，对这两个对象进行equals比较一定法那会False。 4. 如果根据equals方法的比较，两个对象是等价的，那么在两个对象中各自调用hashCode方法必须产生相同的整数结果。（如果equals测试成立，则两个对象中返回的哈希码必然相同。如果两个对象中返回的哈希码不同，则equals测试一定不成立。） 5. 如果根据equals方法的比较，两个对象不等价，那么在两个对象中各自调用hashCode方法不一定会生成不同的整数结果。   equals方法 与 hashCode方法 满足的关系：   |  |  | | --- | --- | | 条件 | 结果 | | a.equals(b)==True | 必须a.hashCode()==b.hashCode() | | a.hashCode()==b.hashCode() | 对equals方法无要求 | | a.equals(b)==False | 对hashCode方法无要求 | | a.hashCode()!=b.hashCode() | 必须a.equals(b)==False |   实际开发中，要么同时重写equals和hashCode方法，要么同时都不重写，不允许只重写一个。  从Collection接口继承而来的一般成为：聚类（或 集合）；  从Map接口继承而来的一般称之为：映射。      Ordered的排列：按某种由具体情况决定的顺序排列。（后天决定的）  Sorted排列：按照天然的顺序排列，是由先天指定的，自身条件影响的。（不受外界影响，反应本质顺序。） |

## 常用的集合接口

### Collection

### List

空集合

Collections.EMPTY\_LIST

### Set

### Map

### Iterator

## 常用集合类

### Iterator

|  |
| --- |
| 用于对集合容器进行向前的单向遍历, 通常称为迭代器. |

### ArrayList 和 LinkedList

### HashSet 和 TreeSet

### HashMap 和 treeMap

## 遍历Enumeration

|  |
| --- |
| **while**(names.hasMoreElements()){  String name = names.nextElement();  String value = **this**.getInitParameter(name);  System.*out*.println(name + ":" + value);  } |

## 遍历map

|  |
| --- |
| 准备工作: 建立Map对象  Map<String,String> map=**new** HashMap<String,String>();  map.put("k1", "v1");  map.put("k2", "v2");  map.put("k3", "v3");  map迭代中不能用remove(key)删除  迭代中删除只能用Iterator的it.remove()方法 |
| // 第1种方法: 用for循环  **for**(Map.Entry<String, String> entry : map.entrySet()){  System.*out*.println(entry.getKey()+"--->"+entry.getValue());  } |
| // 第2种方法: 用entrySet()转换为set, 再用迭代  Iterator i = map.entrySet().iterator();  **while**(i.hasNext()){  Map.Entry<String, String> entry1=(Map.Entry<String, String>)i.next();  System.*out*.println(entry1.getKey()+"=="+entry1.getValue());  } |
| // 第3种方法: 用keySet()转换为set,再用迭代  Iterator it=map.keySet().iterator();  **while**(it.hasNext()){  String key=it.next().toString();  String value=map.get(key);  System.*out*.println(key+"--"+value);  } |

## 遍历List

|  |
| --- |
| List list = new ArrayList();  **for** (Iterator iter=userList.iterator(); iter.hasNext();) {  User user = (User)iter.next();  System.*out*.println(user.getId());  System.*out*.println(user.getName());  } |

## 集合的简单输出

## 遍历Set

|  |
| --- |
| Set<Professional> pro = project.getProfessional();  **for** (Iterator<Professional> it = pro.iterator(); it.hasNext();) {  Professional p = it.next();  System.*out*.println(p.getName());  } |

***集合的简单输出***

|  |  |  |
| --- | --- | --- |
| 打印Array | 类名@哈希码 | List list = Array.asList(array1); 然后就可以直接打印了  数组--转换-- >一个List, 因为list可以直接打印出来. |
| 打印List | [list1, list2, list3] |  |

# 正则表达式

|  |  |  |
| --- | --- | --- |
| 符号 | 描述 | 举例 |
| . | 匹配任何一个字符 | 正则表达式g.f 会匹配glf, g\*f, g#f等 |
| [] | 只有[]里指定的字符才能匹配 | 正则表达式g[abc]f 只能匹配gaf, gbf, gcf |
| \* | 表示紧靠该符号左边的符号出现的次数, 可以是任意次 | 正则表达式g.\*f 它能匹配gf, gaf, gaaf, g\*f等 |
| ? | 表示紧靠该符号左边的符号出现的次数, 可以是0次或1次. | 正则表达式g.?f 它能匹配gaf, g\*f等 |
| \ | 是正则表达式的连接符, 相当于转义字符, 表示. \* ? [] | 正则表达式g.\-f 它能匹配g-f, ga-f, g\*-f等 |

java用正则表达式替换字符串

|  |
| --- |
| /\*\*  \* (已修改好)  \* 过滤HTML文本  \*/  **private** String htmlFilter(String text) {  String htmlText = text;  htmlText = htmlText.replace("&", "&amp;");  htmlText = htmlText.replace("<", "&lt;");  htmlText = htmlText.replace(">", "&gt;");  htmlText = htmlText.replace(" ", "&nbsp;");  htmlText = htmlText.replace("'", "&#39;");  htmlText = htmlText.replace("\"", "&quot;");  htmlText = htmlText.replace("\n", "<br />");  **return** htmlText;  } |

# 常用包

|  |  |
| --- | --- |
| java.applet | 提供创建applet小程序所需要的类 |
| java.lang | 提供java语言程序设计的基础类 |
| java.io | 提供输入输出相关类 |
| java.util | 提供处理日期, 时间, 随机数生成等各种实用工具的类 |
| java.net | 提供用于网络应用程序的类 |
| java.text | 提供处理文本, 日期, 数值和消息的类和接口. |

## text包

SimpleDateFormat类

DecimalFormat类

## util包

### 用java.util.Properties读取properties(国际化资源文件)文件

|  |
| --- |
| 1. 在classpath中新建userInfo.properties文件 |
| 2. 测试类:读写properties  **package** com.ismp.web.task;  **import** java.io.BufferedInputStream;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.util.Enumeration;  **import** java.util.Properties;  **public** **class** TestMain {  /\*\*  \* 根据key读取value  \*  \* **@param** filePath  \* **@param** key  \* **@return**  \*/  **public** **static** String readValue(String filePath, String key) {  Properties props = **new** Properties();  String value = **null**;  **try** {  InputStream in = **new** BufferedInputStream(**new** FileInputStream(filePath));  props.load(in);  value = props.getProperty(key);  } **catch** (Exception e) {  e.printStackTrace();  }  **return** value;  }  /\*\*  \* 读取properties的全部信息  \*  \* **@param** filePath  \*/  **public** **static** **void** readProperties(String filePath) {  Properties props = **new** Properties();  **try** {  InputStream in = **new** BufferedInputStream(**new** FileInputStream(filePath));  props.load(in);  Enumeration en = props.propertyNames();  **while** (en.hasMoreElements()) {  String key = (String) en.nextElement();  String Property = props.getProperty(key);  System.*out*.println(key + Property);  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  /\*\*  \* 写入properties信息  \*  \* **@param** filePath  \* **@param** parameterName  \* **@param** parameterValue  \*/  **public** **static** **void** writeProperties(String filePath, String parameterName, String parameterValue) {  Properties prop = **new** Properties();  **try** {  InputStream fis = **new** FileInputStream(filePath);  // 从输入流中读取属性列表（键和元素对）  prop.load(fis);  // 调用 Hashtable 的方法 put。使用 getProperty 方法提供并行性。  // 强制要求为属性的键和值使用字符串。返回值是 Hashtable 调用 put 的结果。  OutputStream fos = **new** FileOutputStream(filePath);  prop.setProperty(parameterName, parameterValue);  // 以适合使用 load 方法加载到 Properties 表中的格式，  // 将此 Properties 表中的属性列表（键和元素对）写入输出流  prop.store(fos, "Update '" + parameterName + "' value");  } **catch** (IOException e) {  System.*err*.println("Visit " + filePath + " for updating " + parameterName + " value error");  }  }  **public** **static** **void** main(String[] args) {  *readValue*("src/main/resources/ftp.properties", "url");  *writeProperties*("src/main/resources/ftp.properties", "age", "21");  *readProperties*("src/main/resources/ftp.properties");  System.*out*.println("OK");  }  } |

### net包

#### 字符串编码解码(用于处理地址栏传中文乱码问题)

地址栏中文参数是编码后的, 到服务器后解码. 就解决了乱码

|  |  |
| --- | --- |
| 字符串编码 | URLEncoder.*encode*(message, "UTF-8")); |
| 字符串解码 | message = URLDecoder.*decode*(message, "UTF-8"); |

# 常用类

## Object类

|  |  |
| --- | --- |
| hashCode() | 返回该对象的哈希码值。 |
| toString() |  |
| equals() |  |
| finalize() |  |
| clone() |  |
| wait() |  |
| notify()/notifyAll() |  |

## Math类

|  |
| --- |
| java.lang.Math类：  1、Math类的属性和方法都是静态的（不能创建对象）。  2、Math中的常量都是double类型的。 |

|  |  |
| --- | --- |
| Math.PI  Math.E | 圆周率常量  自然常量 |
| abs() | 返回被操作数的绝对值 |
| ceil() | 返回一个大于等于操作数的最近整数值 |
| floor() | 返回一个小于等于操作数的最近整数值 |
| max() | 返回两个操作数中较大的一个 |
| min() | 返回两个操作数中较小的一个 |
| random() | 返回大于等于0.0并且小于等于1.0的随机double值, 此方法不带任何参数. |
| round() | 返回最接近操作数的整数(四舍五入), 起算法是将操作数加上0.5, 并截断. |
| sin() | 返回给定弧度的正弦, 其操作数是double型 |
| cos() | 返回给定弧度的余弦, 其操作数是double型 |
| tan() | 返回给定弧度的正切, 其操作数是double型 |
| sqrt() | 返回给定值的平方根, 其操作数是double型. |
| toDegrees() | 返回给定弧度的角度值 |

## 字符串 相关类

|  |
| --- |
| 创建字符串  1. String name = new String(); //构造一个空字符串  2. String name = new String(bute[] bytes); //将用当前平台的默认字符集解码和指定的byte数组(即字节数组), 构造一个新的字符串. 新字符串的长度不一定等于byte数组的长度. |
| 字符串操作   |  |  | | --- | --- | | 比较字符串  equals()  equalslgnoreCase()  regionMatches(toffset, other,ooffset, len)  regionMatches(ignoreCase, toffset, other, ooffset, len)  startWith(str)  endsWith(str)  int compareTo(String str) |  | | 获取字符串长度  str.length(); |  | | 字符串大小写转换  toLowerCase()  toUpperCase() |  | | 查找字符串  indexOf(int ch)  lastIndexOf(int ch)  indexOf(String str)  lastIndexOf(tring str)  indexOf(int ch, int startIndex)  lastIndexOf(ch, startIndex)  indexOf(String str, int startIndex)  lastIndexOf(String str, int startIndex) |  | | 获取字符串的子串  str.substring(str, end); |  | | 替换 或 去除字符串中的空格  str.replace(oldstr, newstr)  str.trim(); |  | | 分割字符串  str.split(regex) |  |   格式化字符串   |  | | --- | | format(String format, Object... args) | |

StringBuilder类

|  |
| --- |
| 创建字符串生成器  StringBuilder()  StringBuilder(capacity)  StringBuilder(String str) |
| 字符串生成器的应用  append()  delete(int start, int end)  deleteCharAt(int index)  insert(int offset, arg)  length() |

字符串类——java.lang.String

对象创建：

1. String 引用变量=“字符串内容”;
2. String 引用变量= new String(参数表);

字符串搜应是从0开始的。

取字符串方法subString，取到的字符串为索引中间的内容，包括起始索引的字符，不包括结束索引字符。

堆中--字符串常量池

不用new创建字符串对象的过程：

1. 查看字符串常量池中是否有内容相同的字符串对象。
2. 若没有，则创建一个包含该内容的字符串对象，并让引用变量指向该对象。
3. 若已经存在包含该内容的字符串对象，则让字符串引用直接指向该对象。

用new创建字符串对象的过程：

1. 在堆（不是字符串常量池）创建一个包含指定内容的字符串对象。并将字符串引用指向该对象。
2. 去字符串常量池中查看，是否有包含该内容的对象。
3. 若有，则将new出来的字符串对象与字符串常量池中内容相同的对象联系起来。
4. 若没有，则在字符串常量池中再创建一个包含该内容的字符串对象，并将堆中的对象与字符串常量池中新创建出来的对象联系起来。

public String intern()

若本身在堆中，返回字符串常量池中其联系的引用。

若本身在字符串常量池中，则直接将返回自己的引用。

String类的equals方法就是这样实现的，这样就不必自己去调用intern方法了。

实际开发中应尽量少使用new来创建字符串，可以节约内存。

java中创建一个对象要比修改一个已存在的对象耗时大得多。

不管字符串多长，其比较速度都是一样的。

String类是final的。

StringBuffer类：允许对其内容进行修改，而不产生冗余的中间对象。

创建对象只有一种方法：StringBuffer 对象名 = new StringBuffer(参数表);

方法链：在遗憾类完成许多操作。

引用变量.方法1(\*\*\*).方法2(\*\*\*).方法3(\*\*\*);

非最后的方必须有返回值类型（不能是void），且返回值类型必须为对象引用类型。

提示：StringBuffer类也有equals方法，但是其功能相当于两个引用进行“==”比较，并没有String类equals方法的功能。若需要比较两个StringBuffer对象内容是否相同，首先调用toString方法，再进行比较。

StringBuilder类：用于单线程。

StringBuffer类：用于多线程。

正则表达式：指一个特殊的字符串，可以用来表示一种或几种特定的匹配模式。

在复杂匹配、替换时使用Pattern类、Matcher类是很好的。但在简单情况下就没必要了。因为String类也提供了很多基本正则式操作的方法。

Pattern类­——java.util.regex.Pattern类

Pattern类的对象表示通过编译的正则式。可以与任意字符串进行模式匹配。

Pattern类的构造函数是private的，因此不能通过调用构造函数来创建对象。而需要调用静态工厂来获得对象。

Matcher类：

Matcher类的对象表示将要进行模式匹配的字符串或字符序列。

Matcher类的构造函数是private的，因此不能通过调用构造函数来创建对象。一般是通过Pattern对象中的matcher方法得到。

|  |  |
| --- | --- |
| java.lang.String类 | 表示不可修改的Unicode编码字符串  字符串连接, 转换, 截断: concat(), replace(), substring(), toLowerCase(), toUpperCase(), trim()  检索和查找功能: charAt(), startsWith(), indexOf(), lastIndexOf(), length()  内容比较: equals(), equalsIgnoreCase(), compareTo() |
| java.lang.StringBuffer类 | 保存可修改的Unicode编码字符串; 线程安全的,适合多线程 |
| java.lang.StringBuilder类 | 不是线程安全的, 但性能好一些, 适合单线程 |
| java.util.StringTokenizer类 | 将当前字符串按默认或指定的分隔符分解为多个片段. |

## 日期 和 时间 相关类

|  |  |
| --- | --- |
| java.util.Date类 | 对日期时间处理功能并不强大，其记录时间是距离一个固定时间点的ms数。  用于表示特定的时间点, 精确到毫秒, 但不支持日期的国际化和分时区显示 |
| java.util.Calendar类 | 抽象类, 提供常用的日期修改和国际化支持. |
| java.util.GregorianCalendar类 | 是Calendar的子类, 该类提供了世界上大多数国家/地区使用的标准日历系统, 添加了判断润年的功能. |
| Locale类 | 主要封装了"地区"和"语言种类"两方面的信息.  通常用于在国际化/本地化程序中以地区/语言相关的方式显示日期, 数值或文本信息等. |
| java.util.TimeZone类 | 描述时区信息.  抽象类, 调用其静态方法getDefault()以获得该类的实例. |
| java.text.DateFormat类 | 提供了将日期/时间信息进行格式化处理的功能. |
| SimpleDateFormat类 | 是DateFormat的子类, 它用于用户自定义日期/时间信息的表现格式, 并提供了灵活的Date<--->String信息转换和解析的功能. |

**Date类**

|  |
| --- |
| 创建Date对象  new Date(); //创建当前系统时间的Date对象  new Date(106, 11, 8) //106表示2006年, 11表示12月, 8表示8号. |
| 常用方法  boolean after(Date when) //用于测试此日期是否在指定日期之后, 参数为一个Date对象. 当且仅当此Date对象表示的瞬间比when表示的瞬间晚, 返回true; 否则返回false.  boolean before(Date when) ;;此方法与上述方法对应. |

**GregorianCalendar类**

|  |  |
| --- | --- |
| void add(int field, int amount) | field为需要修改的日历字段, 可以使用系统提供的常量, amount为字段添加的日期或时间量 |
| int get(int field) | field为需要修改的日历字段, 可以使用系统提供的常量, 将按照所给的字段返回当前的日期或时间. |
| void setIint field, int value)` | field为需要修改的日历字段, 可以使用系统提供的常量. value为给定日历字段将要设置的值. |
| Date getTime() | 该方法将返回当前对象所描述的时间点. |
| void setTime(Date date) | date为需要设置的Date时间对象, 将时间点设置到当前对象中. |

## 时区

|  |
| --- |
| SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  System.***out***.println(sdf.format(**new** Date())); //假设系统时区选的美国时间, 这里打印的就是美国时间    TimeZone timeZone = TimeZone.*getDefault*();  **int** newTime = (**int**) (8 \* 60 \* 60 \* 1000);    String[] ids = TimeZone.*getAvailableIDs*(newTime);  timeZone = **new** SimpleTimeZone(newTime, ids[15]);  sdf.setTimeZone(timeZone);  System.***out***.println(sdf.format(**new** Date()));//调整时区后, 这里打印的就是中国时间 |

## Date常用代码

|  |
| --- |
| Java获取各种常用时间方法 **import** java.text.DateFormat;  **import** java.text.ParsePosition;  **import** java.text.SimpleDateFormat;  **import** java.util.Calendar;  **import** java.util.Date;  **import** java.util.GregorianCalendar;  **public** **class** TimeTest {  // 用来全局控制 上一周，本周，下一周的周数变化  **private** **int** weeks = 0;  **private** **int** MaxDate;// 一月最大天数  **private** **int** MaxYear;// 一年最大天数  /\*\*  \* **@param** args  \*/  **public** **static** **void** main(String[] args) {  TimeTest tt = **new** TimeTest();  System.*out*.println("获取当天日期:" + tt.getNowTime("yyyy-MM-dd"));  System.*out*.println("获取本周一日期:" + tt.getMondayOFWeek());  System.*out*.println("获取本周日的日期~:" + tt.getCurrentWeekday());  System.*out*.println("获取上周一日期:" + tt.getPreviousWeekday());  System.*out*.println("获取上周日日期:" + tt.getPreviousWeekSunday());  System.*out*.println("获取下周一日期:" + tt.getNextMonday());  System.*out*.println("获取下周日日期:" + tt.getNextSunday());  System.*out*.println("获得相应周的周六的日期:" + tt.getNowTime("yyyy-MM-dd"));  System.*out*.println("获取本月第一天日期:" + tt.getFirstDayOfMonth());  System.*out*.println("获取本月最后一天日期:" + tt.getDefaultDay());  System.*out*.println("获取上月第一天日期:" + tt.getPreviousMonthFirst());  System.*out*.println("获取上月最后一天的日期:" + tt.getPreviousMonthEnd());  System.*out*.println("获取下月第一天日期:" + tt.getNextMonthFirst());  System.*out*.println("获取下月最后一天日期:" + tt.getNextMonthEnd());  System.*out*.println("获取本年的第一天日期:" + tt.getCurrentYearFirst());  System.*out*.println("获取本年最后一天日期:" + tt.getCurrentYearEnd());  System.*out*.println("获取去年的第一天日期:" + tt.getPreviousYearFirst());  System.*out*.println("获取去年的最后一天日期:" + tt.getPreviousYearEnd());  System.*out*.println("获取明年第一天日期:" + tt.getNextYearFirst());  System.*out*.println("获取明年最后一天日期:" + tt.getNextYearEnd());  System.*out*.println("获取本季度第一天到最后一天:" + tt.getThisSeasonTime(11));  System.*out*.println("获取两个日期之间间隔天数2008-12-1~2008-9.29:"  + TimeTest.*getTwoDay*("2008-12-1", "2008-9-29"));  }  /\*\*  \* 得到二个日期间的间隔天数  \*/  **public** **static** String getTwoDay(String sj1, String sj2) {  SimpleDateFormat myFormatter = **new** SimpleDateFormat("yyyy-MM-dd");  **long** day = 0;  **try** {  java.util.Date date = myFormatter.parse(sj1);  java.util.Date mydate = myFormatter.parse(sj2);  day = (date.getTime() - mydate.getTime()) / (24 \* 60 \* 60 \* 1000);  } **catch** (Exception e) {  **return** "";  }  **return** day + "";  }  /\*\*  \* 根据一个日期，返回是星期几的字符串  \*  \* **@param** sdate  \* **@return**  \*/  **public** **static** String getWeek(String sdate) {  // 再转换为时间  Date date = TimeTest.*strToDate*(sdate);  Calendar c = Calendar.*getInstance*();  c.setTime(date);  // int hour=c.get(Calendar.DAY\_OF\_WEEK);  // hour中存的就是星期几了，其范围 1~7  // 1=星期日 7=星期六，其他类推  **return** **new** SimpleDateFormat("EEEE").format(c.getTime());  }  /\*\*  \* 将短时间格式字符串转换为时间 yyyy-MM-dd  \*  \* **@param** strDate  \* **@return**  \*/  **public** **static** Date strToDate(String strDate) {  SimpleDateFormat formatter = **new** SimpleDateFormat("yyyy-MM-dd");  ParsePosition pos = **new** ParsePosition(0);  Date strtodate = formatter.parse(strDate, pos);  **return** strtodate;  }  /\*\*  \* 两个时间之间的天数  \*  \* **@param** date1  \* **@param** date2  \* **@return**  \*/  **public** **static** **long** getDays(String date1, String date2) {  **if** (date1 == **null** || date1.equals(""))  **return** 0;  **if** (date2 == **null** || date2.equals(""))  **return** 0;  // 转换为标准时间  SimpleDateFormat myFormatter = **new** SimpleDateFormat("yyyy-MM-dd");  java.util.Date date = **null**;  java.util.Date mydate = **null**;  **try** {  date = myFormatter.parse(date1);  mydate = myFormatter.parse(date2);  } **catch** (Exception e) {  }  **long** day = (date.getTime() - mydate.getTime()) / (24 \* 60 \* 60 \* 1000);  **return** day;  }  // 计算当月最后一天,返回字符串  **public** String getDefaultDay() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.set(Calendar.*DATE*, 1);// 设为当前月的1号  lastDate.add(Calendar.*MONTH*, 1);// 加一个月，变为下月的1号  lastDate.add(Calendar.*DATE*, -1);// 减去一天，变为当月最后一天  str = sdf.format(lastDate.getTime());  **return** str;  }  // 上月第一天  **public** String getPreviousMonthFirst() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.set(Calendar.*DATE*, 1);// 设为当前月的1号  lastDate.add(Calendar.*MONTH*, -1);// 减一个月，变为下月的1号  // lastDate.add(Calendar.DATE,-1);//减去一天，变为当月最后一天  str = sdf.format(lastDate.getTime());  **return** str;  }  // 获取当月第一天  **public** String getFirstDayOfMonth() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.set(Calendar.*DATE*, 1);// 设为当前月的1号  str = sdf.format(lastDate.getTime());  **return** str;  }  // 获得本周星期日的日期  **public** String getCurrentWeekday() {  weeks = 0;  **int** mondayPlus = **this**.getMondayPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, mondayPlus + 6);  Date monday = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preMonday = df.format(monday);  **return** preMonday;  }  // 获取当天时间  **public** String getNowTime(String dateformat) {  Date now = **new** Date();  SimpleDateFormat dateFormat = **new** SimpleDateFormat(dateformat);// 可以方便地修改日期格式  String hehe = dateFormat.format(now);  **return** hehe;  }  // 获得当前日期与本周日相差的天数  **private** **int** getMondayPlus() {  Calendar cd = Calendar.*getInstance*();  // 获得今天是一周的第几天，星期日是第一天，星期二是第二天......  **int** dayOfWeek = cd.get(Calendar.*DAY\_OF\_WEEK*) - 1; // 因为按中国礼拜一作为第一天所以这里减1  **if** (dayOfWeek == 1) {  **return** 0;  } **else** {  **return** 1 - dayOfWeek;  }  }  // 获得本周一的日期  **public** String getMondayOFWeek() {  weeks = 0;  **int** mondayPlus = **this**.getMondayPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, mondayPlus);  Date monday = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preMonday = df.format(monday);  **return** preMonday;  }  // 获得相应周的周六的日期  **public** String getSaturday() {  **int** mondayPlus = **this**.getMondayPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, mondayPlus + 7 \* weeks + 6);  Date monday = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preMonday = df.format(monday);  **return** preMonday;  }  // 获得上周星期日的日期  **public** String getPreviousWeekSunday() {  weeks = 0;  weeks--;  **int** mondayPlus = **this**.getMondayPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, mondayPlus + weeks);  Date monday = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preMonday = df.format(monday);  **return** preMonday;  }  // 获得上周星期一的日期  **public** String getPreviousWeekday() {  weeks--;  **int** mondayPlus = **this**.getMondayPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, mondayPlus + 7 \* weeks);  Date monday = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preMonday = df.format(monday);  **return** preMonday;  }  // 获得下周星期一的日期  **public** String getNextMonday() {  weeks++;  **int** mondayPlus = **this**.getMondayPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, mondayPlus + 7);  Date monday = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preMonday = df.format(monday);  **return** preMonday;  }  // 获得下周星期日的日期  **public** String getNextSunday() {  **int** mondayPlus = **this**.getMondayPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, mondayPlus + 7 + 6);  Date monday = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preMonday = df.format(monday);  **return** preMonday;  }  **private** **int** getMonthPlus() {  Calendar cd = Calendar.*getInstance*();  **int** monthOfNumber = cd.get(Calendar.*DAY\_OF\_MONTH*);  cd.set(Calendar.*DATE*, 1);// 把日期设置为当月第一天  cd.roll(Calendar.*DATE*, -1);// 日期回滚一天，也就是最后一天  MaxDate = cd.get(Calendar.*DATE*);  **if** (monthOfNumber == 1) {  **return** -MaxDate;  } **else** {  **return** 1 - monthOfNumber;  }  }  // 获得上月最后一天的日期  **public** String getPreviousMonthEnd() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.add(Calendar.*MONTH*, -1);// 减一个月  lastDate.set(Calendar.*DATE*, 1);// 把日期设置为当月第一天  lastDate.roll(Calendar.*DATE*, -1);// 日期回滚一天，也就是本月最后一天  str = sdf.format(lastDate.getTime());  **return** str;  }  // 获得下个月第一天的日期  **public** String getNextMonthFirst() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.add(Calendar.*MONTH*, 1);// 减一个月  lastDate.set(Calendar.*DATE*, 1);// 把日期设置为当月第一天  str = sdf.format(lastDate.getTime());  **return** str;  }  // 获得下个月最后一天的日期  **public** String getNextMonthEnd() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.add(Calendar.*MONTH*, 1);// 加一个月  lastDate.set(Calendar.*DATE*, 1);// 把日期设置为当月第一天  lastDate.roll(Calendar.*DATE*, -1);// 日期回滚一天，也就是本月最后一天  str = sdf.format(lastDate.getTime());  **return** str;  }  // 获得明年最后一天的日期  **public** String getNextYearEnd() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.add(Calendar.*YEAR*, 1);// 加一个年  lastDate.set(Calendar.*DAY\_OF\_YEAR*, 1);  lastDate.roll(Calendar.*DAY\_OF\_YEAR*, -1);  str = sdf.format(lastDate.getTime());  **return** str;  }  // 获得明年第一天的日期  **public** String getNextYearFirst() {  String str = "";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  Calendar lastDate = Calendar.*getInstance*();  lastDate.add(Calendar.*YEAR*, 1);// 加一个年  lastDate.set(Calendar.*DAY\_OF\_YEAR*, 1);  str = sdf.format(lastDate.getTime());  **return** str;  }  // 获得本年有多少天  **private** **int** getMaxYear() {  Calendar cd = Calendar.*getInstance*();  cd.set(Calendar.*DAY\_OF\_YEAR*, 1);// 把日期设为当年第一天  cd.roll(Calendar.*DAY\_OF\_YEAR*, -1);// 把日期回滚一天。  **int** MaxYear = cd.get(Calendar.*DAY\_OF\_YEAR*);  **return** MaxYear;  }  **private** **int** getYearPlus() {  Calendar cd = Calendar.*getInstance*();  **int** yearOfNumber = cd.get(Calendar.*DAY\_OF\_YEAR*);// 获得当天是一年中的第几天  cd.set(Calendar.*DAY\_OF\_YEAR*, 1);// 把日期设为当年第一天  cd.roll(Calendar.*DAY\_OF\_YEAR*, -1);// 把日期回滚一天。  **int** MaxYear = cd.get(Calendar.*DAY\_OF\_YEAR*);  **if** (yearOfNumber == 1) {  **return** -MaxYear;  } **else** {  **return** 1 - yearOfNumber;  }  }  // 获得本年第一天的日期  **public** String getCurrentYearFirst() {  **int** yearPlus = **this**.getYearPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, yearPlus);  Date yearDay = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preYearDay = df.format(yearDay);  **return** preYearDay;  }  // 获得本年最后一天的日期 \*  **public** String getCurrentYearEnd() {  Date date = **new** Date();  SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy");// 可以方便地修改日期格式  String years = dateFormat.format(date);  **return** years + "-12-31";  }  // 获得上年第一天的日期 \*  **public** String getPreviousYearFirst() {  Date date = **new** Date();  SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy");// 可以方便地修改日期格式  String years = dateFormat.format(date);  **int** years\_value = Integer.*parseInt*(years);  years\_value--;  **return** years\_value + "-1-1";  }  // 获得上年最后一天的日期  **public** String getPreviousYearEnd() {  weeks--;  **int** yearPlus = **this**.getYearPlus();  GregorianCalendar currentDate = **new** GregorianCalendar();  currentDate.add(GregorianCalendar.*DATE*, yearPlus + MaxYear \* weeks  + (MaxYear - 1));  Date yearDay = currentDate.getTime();  DateFormat df = DateFormat.*getDateInstance*();  String preYearDay = df.format(yearDay);  getThisSeasonTime(11);  **return** preYearDay;  }  // 获得本季度  **public** String getThisSeasonTime(**int** month) {  **int** array[][] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };  **int** season = 1;  **if** (month >= 1 && month <= 3) {  season = 1;  }  **if** (month >= 4 && month <= 6) {  season = 2;  }  **if** (month >= 7 && month <= 9) {  season = 3;  }  **if** (month >= 10 && month <= 12) {  season = 4;  }  **int** start\_month = array[season - 1][0];  **int** end\_month = array[season - 1][2];  Date date = **new** Date();  SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy");// 可以方便地修改日期格式  String years = dateFormat.format(date);  **int** years\_value = Integer.*parseInt*(years);  **int** start\_days = 1;// years+"-"+String.valueOf(start\_month)+"-1";//getLastDayOfMonth(years\_value,start\_month);  **int** end\_days = getLastDayOfMonth(years\_value, end\_month);  String seasonDate = years\_value + "-" + start\_month + "-" + start\_days  + ";" + years\_value + "-" + end\_month + "-" + end\_days;  **return** seasonDate;  }  /\*\*  \* 获取某年某月的最后一天  \*  \* **@param** year  \* 年  \* **@param** month  \* 月  \* **@return** 最后一天  \*/  **private** **int** getLastDayOfMonth(**int** year, **int** month) {  **if** (month == 1 || month == 3 || month == 5 || month == 7 || month == 8  || month == 10 || month == 12) {  **return** 31;  }  **if** (month == 4 || month == 6 || month == 9 || month == 11) {  **return** 30;  }  **if** (month == 2) {  **if** (isLeapYear(year)) {  **return** 29;  } **else** {  **return** 28;  }  }  **return** 0;  }  /\*\*  \* 是否闰年  \*  \* **@param** year  \* 年  \* **@return**  \*/  **public** **boolean** isLeapYear(**int** year) {  **return** (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);  }  } |

## 日期转换

|  |
| --- |
| 日期 <----转----> 字符串 用的API是SimpleDateFormat，它是属于java.text.SimpleDateFormat，所以请记得import进来。  用法：SimpleDateFormat sdf = new SimpleDateFormat（ " yyyy-MM-dd HH：mm：ss " ）；这一行最重要，它确立了转换的格式，yyyy是完整的公元年，MM是月份，dd是日期，至于HH：mm：ss就不需要我再解释了吧！  ps：为什么有的格式大写，有的格式小写，那是怕避免混淆，例如MM是月份，mm是分；HH是24小时制，而hh是12小时制  这个字符串的内容便类似2002-10-08 14：55：38  透过这个API我们便可以随心所欲的将日期转成我们想要的字符串格式，例如希望将日期输出成2002年10月08日，  我们可以这么写：  SimpleDateFormat sdf = new SimpleDateFormat（ " yyyy年MM月dd日 " ）；  String datestr = sdf.format（ new Date（））；  datestr便会依照我们设定的格式输出  附  SimpleDateFormat parser = new SimpleDateFormat（"EEEE， MMMM dd， yyyy"）；  SimpleDateFormat formatter = new SimpleDateFormat（"EEE. MM/dd"）； |

### 1.字符串转日期：

2002-10-8 15：30：22要把它转成日期，可以用

SimpleDateFormat sdf = new SimpleDateFormat( " yyyy-MM-dd HH：mm：ss " )；

Date date = sdf.parse( " 2002-10-8 15：30：22 " )；

|  |
| --- |
| SimpleDateFormat sdf\_1 = **new** SimpleDateFormat("EEE MMM dd HH:mm:ss z yyyy", Locale.*US*);  Date date1 = sdf\_1.parse("Wed Apr 21 15:03:20 CST 2010"); |
| SimpleDateFormat sdf\_2 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  Date date2 = sdf\_2.parse("2010-08-20 10:10:10"); |

### 2.日期转字符串

假如把今天的日期转成字符串可用

|  |
| --- |
| SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  String datestr = sdf.format(**new** Date()); |
| **简洁版(和上面功能一样)**  String datestr = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(**new** Date()); |

### 3.Long --> Date

|  |
| --- |
| **public** **static** **void** longToDate() {  **long** sd=1412068518365L;  Date dat=**new** Date(sd);  GregorianCalendar gc = **new** GregorianCalendar();  gc.setTime(dat);  java.text.SimpleDateFormat format = **new** java.text.SimpleDateFormat("yyyy-MM-dd hh:mm:ss");  String sb=format.format(gc.getTime());  System.***out***.println(sb);  } |

## Calendar类

|  |
| --- |
| Calendar calendar = Calendar.*getInstance*();  calendar.set(Calendar.*HOUR\_OF\_DAY*, 0);//设置小时为0  calendar.set(Calendar.*MINUTE*, 0); //设置分钟为0  calendar.set(Calendar.*SECOND*, 0); //设置秒为0 |

## 基本数据类型 和 封装类

|  |  |
| --- | --- |
| 基本数据类型 | 封装类 |
| boolean | Boolean |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| char | Character |
| float | Float |
| double | Double |

**封装类的使用：将基本数据类型当作对象使用。**

**封装类的功能：**

1. **将基本数据类型封装后当作对象进行操作。**
2. **为基本数据类型提供各种转换功能。**

**除了Character、Float类之外，所有封装类都提供了两个构造器，一个是参数为对应基本数据类型的；另一个是参数为字符串类型的。**

**Character类只提供了一个参数为char类型的构造器。**

**Float类有3个构造器，一个参数为字符串类型，还有两个参数为原始数据类型，分别是float型和double型。**

**Boolean封装构造器只要接受的字符串不是“True”（不区分大小写），一律认为表示“false”，不会报错。**

**除了Boolean封装类之外，其它带有字符串参数的封装类构造器都只接受对应基本类型数据的字符串，否则报异常。**

**封装类对象一旦创建，其内容永不改变。**

**封装类都是final的，不允许有子类。**

**封装类 数据转换功能：**

**基本数据类型—>字符串类型**

1. **非静态转换方法：**

**首先创建封装对象；然后调用该封装类对象的非静态无参数toString()方法。（任何封装类都提供一个toString方法）**

**对象引用调用toString()方法，将返回：引用变量类型名@地址**

1. **静态转换方法：有参数**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **静态方法** | **Boolean** | **Byte** | **Character** | **Short** | **Integer** | **Long** | **Float** | **Double** |
| **转为字符串** | **toString(XXX)** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **转为指定进制** | **toString(xxx,sss)** |  |  |  |  | **\*** | **\*** |  |  |
| **转为二进制** | **toBinaryString(xxx)** |  |  |  |  | **\*** | **\*** |  |  |
| **转为八进制** | **toOctalString(xxx)** |  |  |  |  | **\*** | **\*** |  |  |
| **转为十六进制** | **toHexString(xxx)** |  |  |  |  | **\*** | **\*** | **\*** | **\*** |

**字符串—>基本数据类型：**

**无参数的非静态（需要创建对象）方法：**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **方法** | **Boolean** | **Byte** | **Character** | **Short** | **Integer** | **Long** | **Float** | **Double** |
| **转为byte** | **byteValue** |  | **\*** |  | **\*** | **\*** | **\*** | **\*** | **\*** |
| **转为short** | **shortValue** |  | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **转为int** | **intValue** |  | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **转为long** | **longValue** |  | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **转为float** | **floatValue** |  | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **转为double** | **doubleValue** |  | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **转为boolean** | **booleanValue** | **\*** | **\*** | **\*** |  |  |  |  |  |
| **转为char** | **charValue** |  |  | **\*** |  |  |  |  |  |

**静态方法：只能以字符串为参数。返回的是基本数据类型的值。**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **方法** | **Boolean** | **Byte** | **Character** | **Short** | **Integer** | **Long** | **Float** | **Double** |
| **parseXxx(String s)** | **\*** | **\*** |  | **\*** | **\*** | **\*** | **\*** | **\*** |
| **parseXxx(String s, int i)** |  | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |

1. **parseXxx方法以字符串为参数，返回对应基本类型，若字符参数形式不正确，则抛出NumberFormatException。**
2. **parseXxx方法以字符串为参数，返回对应指定类型**

**静态工厂方法：可以以对应的基本数据类型为参数。返回的是封装类对象引用。**

**创建封装类对象的时候，除了可以使用new操作符调用构造器外，还可以使用静态工厂方法valueOf：**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **方法** | **Boolean** | **Byte** | **Character** | **Short** | **Integer** | **Long** | **Float** | **Double** |
| **valueOf(xxx)** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **valueOf(String s)** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** | **\*** |
| **valueOf(String s, int i)** |  | **\*** | **\*** | **\*** | **\*** | **\*** |  |  |

**若字符参数形式不正确，则抛出NumberFormatException异常。**

**isNaN方法：只有浮点数对应的封装类具有该方法。**

**不可以用等于（不等于）运算来判断运算结果是否为NaN。**

|  |  |  |
| --- | --- | --- |
| **方法** | **Float** | **Double** |
| **非静态的boolean isNaN()** | **\*** | **\*** |
| **static Boolean isNaN(xxx)** | **\*** | **\*** |

**这两个方法均返回boolean的值。**

**比较任何两个对象是否相等：用public boolean equals(Object o)非静态方法。**

**注意：不要比较两个不同类型的对象，否则有可能得出错误的结果。**

**自动打包/解包：java SE 5.0后开始使用**

**直接将基本数据类型的值当作封装类对象使用，系统会自动封装。**

**特殊的数值计算：**

**特大整数的计算：java中提供不限制大小的整数计算类——java.math.BigInteger**

**浮点数的精确计算：使用java.math.BigDecimal类。**

## Random类

|  |
| --- |
| new Random().nextInt(1000000)+"";//返回六位随机值(字符串类型) |

***URL编码 解码***

|  |  |
| --- | --- |
| URL编码 | msg = URLEncoder.encode(“被编码内容”, “UTF-8”); |
| URL解码 |  |

### 把字符串从一种编码转换成另外一种编码

|  |
| --- |
| String newName = new String(name.getBytes("ISO-8859-1"), "GBK"); |

### Random类：

|  |
| --- |
| 生成1--100的随机数  Random r = new Random();  int t = r.nextInt(100) + 1; |

# 事件处理

# Swing

JFC:java基础类库。

javax中的x是Extension的缩写，表示其是Java核心类库的扩展包。

AWT是：Abstract Window Toolkit（抽象窗体工具包）

AWT是Swing大厦的基石。

AWT中提供的都是重量级控件。是依赖本地系统的对等体实现的。如果编写的程序希望运行在不同的平台上，必须在每一个平台上单独测试，无法真正实现“一次编写，到处运行。”

Swing是轻量级的，是由纯java编写的。

当AWT控件与Swing控件重合时，AWT控件的显示优先级高。

20.1 窗体——javax.swing.JFrame类

窗体：GUI程序最外层的容器。

20.1.2

java中支持的图片格式一般为jpg、gif、png。

20.2 AWT 1.1事件处理模式

事件源：出发时间的控件。

事件监听器：实现了专门监听接口的类的对象，每种事件都有其对应的监听接口，在监听接口中给出了专门处理特定事件的方法。

进行事件处理之前要将监听器注册给事件源。当指定事件被触发时系统会通过监听接口回调对应的事件处理方法。

不论事件源是何种具体类型，返回的都是Object类型的引用，开发人员需要自己编写代码进行引用的强制类型转换。

# AWT

# Applet

# \_\_\_\_\_\_\_\_\_未分类\_\_\_\_\_\_\_\_\_

# 密码不可逆MD5加密

|  |
| --- |
| **import** java.security.MessageDigest;  **import** java.security.NoSuchAlgorithmException;  **public** **class** PasswordHelper {  **public** **static** String encrypt(String password) {  **if**(password == **null**) {  **return** **null**;  } **else** **if**(password == "") {  **return** "";  }    String newPassword = password;  MessageDigest md5;  **try** {  md5 = MessageDigest.*getInstance*("MD5");  } **catch** (NoSuchAlgorithmException e) {  // ignore  **return** newPassword;  }    **return** Base64.*encode*(md5.digest(password.getBytes()));  }  } |

java运行linux命令, 返回运行结果字符串

|  |
| --- |
| /\*\*  \* 运行linux shell  \*  \* **@param** sh  \* **@return**  \*/  **public** **static** String runLinuxShell(String sh){  StringBuffer sb = **new** StringBuffer();  **try** {  Process process = Runtime.*getRuntime*().exec(sh);  InputStreamReader ir = **new** InputStreamReader(process.getInputStream());  LineNumberReader input = **new** LineNumberReader(ir);    String line;  **while** ((line = input.readLine()) != **null**) {  sb.append(line);  sb.append("----");  }  } **catch** (java.io.IOException e) {  System.*err*.println("IOException " + e.getMessage());  }  //System.out.println("shell:"+sh+"; result: "+sb.toString());  **return** sb.toString();  } |

文件大小单位换算

|  |
| --- |
| /\*\*  \* 单位换算  \*  \* **@param** dbs 默认单位为K  \* **@return**  \* **@throws** Exception  \*/  **private** **static** String unitConversion(Long size){  **if** (size==**null** || size<=0) {  **return** "0";  }**else** **if**(size<1024){  **return** size+"k";  }**else** **if**(size<1024\*1024){  **return** size/1024+"M";  }**else** **if**(size<1024\*1024\*1024){  **return** size/1024/1024+"G";  }**else**{  **return** "0";  }  } |