第一章 Java入门

运行环境：JRE（java runtime environment）

包括核心API，集成API，用户界面API，核心技术，JVM五个部分。

开发环境：JDK

包括了编译java程序的编译器（即javac命令）。

Java三个部分：

* J2EE(企业级开发应用的解决方案),
* J2SE（java技术的核心和基础）,
* J2ME（用于移动设备和信息家电的有限存储设备）。

高级语言程序执行方式分为两种，编译型和解释型

编译型：

使用专门的编译器、针对特定平台将高级编程语言源代码一次性“翻译”成该平台硬件执行的机器码，并包装成该平台所能识别的可执行程序格式，这个转换过程就叫编译。例如c,c++等语言。移植性差，即跨平台性差。

解释型：

使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行语言，相当于将编译型语言中编译和解释过程混合在一起同时完成。移植性好，效率差，不能脱离解释器独立运行。如RUBY,python。

Java程序运行机制：

Java语言编写的程序需要经过编译步骤，但这个编译步骤不会生成特定平台的机器码，而是生成一种与平台无关的字节码（\*.class文件），这种字节码不是可执行性的，需要通过java解释器来解释执行（即JVM）。

Java源文件（\*.java文件）->使用javac命令编译->字节码文件（\*.class文件）->使用java命令解释->特定平台的机器码。

开发前的准备

安装JDK

JDK(java SE development kit)即java标准版开发包。它提供了编译、运行java程序的所需的各种工具和资源，包括java编译器、java运行环境、常用的java类库。

安装了jdk就包含了jre(开发java程序)，如果只需要运行java程序，就只需要安装jre，JRE包含了JVM和运行java程序的其他环境支持。不提供单独的JVM下载。

JDK安装文件下的部分文件路径：

bin:存放JDK的各种工具命令，常用的如javac,java命令。

jre:运行java程序所必须的JRE环境。

lib:存放的是JDK工具命令的实际执行程序文件，典型的dt.jar和tools.jar

path环境变量：

计算机查找命令，windows操作系统根据Path环境变量来查找命令。

Path:追加：JDK的安装路径下各种工具命令所在的包eg.C:\java\jdk1.7.0\_80\bin

ClassPath环境变量：（用于定位类，即找寻\*.class编译文件，来实现解释）

Jdk1.4版本后，系统会自动查找.class文件位置，完全不用我们手动设置ClassPath变量。

但是能手动配置也没有错，配置格式为：

.;%JAVA\_HOME%\lib\dt.jar; %JAVA\_HOME%\lib\tools.jar;

一定不要忽略前面的（.），点代表当前路径，用以强制java解释器在当前路径下搜索java类。

Java源文件命名规则

后缀名为.java；主文件名可以是任意的。但是有一种情况例外，即java程序源代码中用public修饰了的类名，该文件的主文件名必须与类名称相同。

提示：一个java源文件中可以包涵多个类定义，但最多只能包涵一个public类定义，如果java源文件包涵public类定义，则该源文件的文件名必须与这个public类名称相同。

命名建议：

一个源文件中只定义一个类，不同类使用不同的源文件定义。

将每个源文件中单独定义的类都定义成public。

保持java源文件的主文件名与该源文件中定义的public类同名。

类的加载、连接和初始化

JVM和类

当我们调用java命令运行某个java程序时，该命令将会启动一条java虚拟机进程，不管该程序有多复杂，该程序启动了多少个线程，他们都处于该虚拟机进程里。当java程序运行结束时，jvm进程结束，该进程在内存中状态将会丢失。当系统出现以下几种情况，jvm进程将被终止：

* 程序运行到最后正常结束。
* 程序运行到使用System.exit()或Runtime.getRuntime().exit()代码结束程序。
* 程序执行过程中遇到未捕获的异常或错误而结束。
* 程序所在的平台强制结束了JVM进程。

类的加载

当程序主动使用某个类时，如果该类还未被加载到内存中，系统会通过加载、连接、初始化三个步骤来对该类进行初始化，如果没有意外，JVM将会连续完成这三个步骤，所以有时也把这三个步骤称为类加载或类初始化。

类的连接

当类被加载之后，系统为之生成一个对应的Class对象，接着将会进入连接阶段，连接阶段将会负责把类地二进制数据合并到jre中。

类加载指的是将类的class文件读入内存，并为之创建一个java.lang.Class对象，也就是说当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。

类的加载由类加载器完成，类加载器通常由JVM提供，类加载器是所有java程序运行的基础，JVM提供的这些类加载器通常称为系统加载器。除此之外，开发者可以通过继承ClassLoader基础类来创建自己的类加载器。

第二章 数据类型

一、八种基本数据类型

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 表示符 | 内存 | 范围 |
| 整 型 | byte | 1字节 | -128~127(-2^7~2^7-1) |
| short | 2字节 | -2^15~2^15-1 |
| int | 4字节 | -2^31~2^31-1 |
| long | 8字节 | -2^63~2^63-1 |
| 浮点型 | float | 4字节 | 有效位数为6~7 |
| double | 8字节 | 有效位数为15位 |
| 字符型 | char |  |  |
| 布尔型 | boolean |  |  |

二、变量及常量

1、标识符：

1.由字母、数字、下划线和“$”中任意字符组合而成。

2.首字母必须是字母、下划线和“$”。

3.需具有一定的意义，不能是系统关键字。

4.严格区分大小写，例如Student和student是两个不同的标识符。

2、声明变量及变量初始化：

语法格式1：数据类型 变量名；

语法格式2：数据类型 变量名 = 值；

**float** floatNum; //声明一个变量

floatNum = 18.5f;//为变量赋值即初始化

**int** price = 20;//声明变量并初始话

3、常量

常量通过关键字final来指示，表示该变量只能被赋值一次，一旦赋值之后就不能再更改，习惯上常量名用大写来表示。

语法格式：final 数据类型 变量名 = 值；

**final int** UNITPRICE = 20;//声明常量并赋值

三、运算符

1、位运算符（重点）

位运算包括：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 与 | 或 | 异或 | 非 | 右移 | 左移 | 无符号右移 |
| & | **|** | **^** | **~** | **>>** | **<<** | **>>>** |
| 5&7 | 5|7 | 5^7 | ~4 | 8>>3 | 2<<3 | -2>>>2 |
| 5 | 7 | 2 | -5 | 1 | 16 |  |

运算法则关系表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 第一个数 | 第二个数 | 按位与& | 按位或| | 按位异或^ |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

运算原理（&，|，^，~）

00000101

^ 00001001

00001100

-5的原码：10000000 00000000 00000000 00000101

-5的反码：11111111 11111111 11111111 11111010

-5的补码：11111111 11111111 11111111 11111011

~运算结果：00000000 00000000 00000000 00000100

00000101

& 00001001

00000001

00000101

| 00001001

00001101

**int** numb1 = 5&9; //输出结果为1

**int** numb2 = 5|9; //输出结果为13

5对应的二进制为：00000101;(省略前24位0)

9对应的二进制为：00001001;

numb1,numb2运算过程：

numb1=2^0\*1=1；numb2=2^3\*1+2^2\*1+2^0\*1=13;

所以位运算在进行运算的时候会以二进制的形式对每一位上的数值进行位运算。

**int** numb3 = ~-5; //输出结果为4

**int** numb4 = 5^9; //输出结果为12

numb3,numb4运算过程：

注意：

①、负数在计算机中以其正数的补码形式存储，

②、反码除符号位不变，其它位全部取反，

③、补码是在反码的基础上加1，

④、正数的原码、反码、补码一致。

提示：&与|两个运算符，与&&和||这两个运算符非常相似，&与|在计算结果前，一定要计算两个操作数的值。

**boolean** flag1 = a>b && a>c;

**boolean** flag2 = a>b & a>c;

flag1的计算过程：先比较条件1，也就是a>b是否为真，如果不为真，不会再对条件2进行计算，直接返回结果；

flag2的计算过程：不管条件1是否为真，都会对条件2进行计算，然后再返回结果；

运算原理（<<，>>，>>>）

11111111 11111111 11111111 11111011 ——-5的补码

11111111 11111111 11111111 1111101100 ——左移2位,0填充低位

11111111 11111111 11111111 11101100 ——截断高位,新补码

10000000 00000000 00000000 00010011 ——新反码

10000000 00000000 00000000 00010100 ——新补码,最终结果

-（24+22）=-20 ——换算成十进制最终结果

11111111 11111111 11111111 11110110 ——-11的补码

11111111 11111111 11111111 11110110 ——右移2位

1111111111 11111111 11111111 11110110 ——符号位填充高位

11111111 11111111 11111111 11111101 ——截断低位,新补码

10000000 00000000 00000000 00000010 ——新反码

10000000 00000000 00000000 00000011 ——新补码,最终结果

-（21+1）=-3 ——换算成十进制最终结果

11111111 11111111 11111111 11110110 ——-11的补码

11111111 11111111 11111111 11110110 ——右移2位

0011111111 11111111 11111111 11110110 ——0填充高位

00111111 11111111 11111111 11111101 ——截断低位,新原码

换算成十进制最终结果：230-3

**int** numb1 = 5<<2; //输出结果为20

**int** numb2 = -5<<2; //输出结果为-20

-5对应的二进制补码为：11111111 11111111 11111111 11111011;

numb2运算过程：

**int** numb3 = 8>>2; //输出结果为2

**int** numb4 = -11>>2; //输出结果为-3

numb4运算过程：

**int** numb5 = -11>>>2; //输出结果为230-3

numb5运算过程：

小结：

①、<<，>>，>>>三个位运算符只适合**byte,short,char,int,long**等整数型 数据类型的运算；

②、对于低于**int**型（如**byte,short,char**）的操作数总是先自动类型转换成 **int**类型后再移位；

③、没有无符号左移运算符，对于a<<b运算：

当log2a + b< L-1(L为a的字长)时，SD = a\*2b>0；

当log2a + b= L-1时，SD = a\*2b <0,值为负数(java程序运算结果)；

当log2a + b= L时，SD = a\*2b =0,值为零；

当log2a + b> L(b<L)时，SD = a\*2b >0；

当b>L时，系统先用b对L取余，得到的结果才是真正的移位的位数。例 如对于**int**类型的a<<33与a<<1的结果是一样的。

④、有符号右移，对于a>>b运算，SD=a/2b,当b>L时，系统先用b对L取余， 得到的结果才是真正的移位的位数。例如对于**int**类型的a>>33与a>>1 的结果是一样的。

2、其他运算符

|  |  |
| --- | --- |
| 算术运算符 | + - \* / % ++ -- |
| 关系运算符 | > < == != >= <= |
| 逻辑运算符 | 或（||）与（&&）非（！） |
| 赋值运算符 | = |
| 条件运算符 | 条件表达式？语句1：语句2 |
| 运算符的优先级：  {() []}（结合性：从左到右）  {！ +（正） -（负） ++ --}（结合性：从右到左）  {（\* / % + -）（ == != ）（&& ||）}（结合性：从左到右）  {（？：）（= += -= \*= /= %=）}(结合性：从右向左) | |

四、数据类型转换

1、自动类型转换

Java允许7种基本数值类型的值或变量之间进行相互转换，当把表数范围小的值或变量直接赋予表数范围大的值或变量的时候，java会自动进行类型转换。

任何基本类型的值和字符串值进行连接运算时，基本类型的值将自动转换为字符串类型。

2、强制类型转换

强制类型转换等价于自动类型转换的逆过程，是把表数范围大的值或变量赋予表数范围小的变量，强制类型转的运算符为（()）。强制类型转换的过程中，整数位按字长小的位数取值，字长大的数值的高位全部被截断，如果是将浮点型转换成整型，小数位会直接被截断。所以强制类型转换很容易造成数据丢失。

**public** **class** NarrowConversion {

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

**int** iValue = 233;

**byte** bValue = (**byte**) iValue;

//输出-23

System.*out*.println(bValue);

**double** dValue = 3.98;

**int** tol = (**int**)dValue;

//输出3

System.*out*.println(tol);

}

}

在上述示例中，int类型的233强制类型转换为byte类型的数值就造成了数据丢失，转换过程如下：

**int**: 00000000 00000000 00000000 11101001

**byte**: ~~00000000 00000000 00000000~~ 11101001

反码：10010110

原码：10010111 结果：-(24+22+21+1)=-23

第三章 流程控制语句

一、顺序结构

顺序结构是流程控制语句中最简单的，就是程序从上到下一行一行的执行，期间没有任何判断和跳转。

二、选择结构

1、**if** /**else**结构语句

**import** java.util.Scanner;

**public** **class** OrderConstruct {

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

**int** score;

String section;

Scanner sc = **new** Scanner(System.in);

score = sc.nextInt();

section = sc.next();

**if**(section.equals("理")){

**if**(score>=550){

System.out.println("2012湖北一本理工科分数线");

}**else** **if**(490<=score && score<550){

System.out.println("2012湖北二本理工科分数线");

}**else** **if**(400<=score && score<490){

System.out.println("2012湖北三本理工科分数线");

}**else**{

System.out.println("咋们复读吧");

}

}**else** **if**(section.equals("文")){

**if**(score>=540){

System.out.println("2012湖北一本文史科分数线");

}**else** **if**(480<=score && score<540){

System.out.println("2012湖北二本文史科分数线");

}**else** **if**(360<=score && score<480){

System.out.println("2012湖北三本文史科分数线");

}**else**{

System.out.println("还是复读吧");

}

}**else**{

System.out.println("体育艺术生不在该系统查询范围类!");

}

}

}

2、**switch-case**结构语句

**import** java.util.Scanner;

//问题描述：统计一个字符串中+、-、\*、/、%的个数已经对应的索引位置，例如：1+2+3\*4-5/5+6

**public** **class** OrderConstruct2 {

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

Scanner sc = **new** Scanner(System.*in*);

String expression = sc.next();

**int** length = 0;

**int**[] index = **new** **int**[expression.length()];

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

**char** operatorJudge = expression.charAt(i);

**switch**(operatorJudge){

**case** '+':index[length]=i;

length++;

**break**;

**case** '-':index[length]=i;

length++;

**break**;

**case** '\*':index[length]=i;

length++;

**break**;

**case** '/':index[length]=i;

length++;

**break**;

**case** '%':index[length]=i;

length++;

**break**;

}

}

System.*out*.println("算术运算符个数："+length);

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

System.*out*.println("第"+i+1+"个运算符位置："+index[i]);

}

}

}

三、循环结构

1、**While**语句

语法结构：

**While**（条件表达式）{

循环体;

}

**import** java.util.Scanner;

//题目3：一球从100米高度自由落下，每次落地后反跳回原高度的一半；再落下，那么它第10次落地时，共经过多少米？第10次反弹多高？

**public** **class** algorithm3 {

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

**double** height = 100;

**double** distance = 100;

**int** i=1;

**while**(i<=10){

distance = distance +height;

height = height/2;

i++;

}

System.*out*.println("路程："+distance);

System.*out*.println("高度："+height/2);

}

}

2、**do-while**语句

语法结构：

**do**{

循环体;

}**while**（条件表达式）;

**import** java.util.Scanner;

//问题：求1!+2!+...+n!;用do\_while语句实现

**public** **class** DoWhileTest {

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

Scanner sc = **new** Scanner(System.*in*);

**int** n = sc.nextInt();

**int** count = 1;

**int** sum = 0;

**int** sn =1;

**do**{

sn\*=count;

sum+=sn;

count++;

}**while**(count<=n);

System.*out*.println("总和sum="+sum);

}

}

3、**for**语句

语法结构：

**for**(表达式1;条件表达式;表达式2){

循环体;

}

//题目2：一个数如果恰好等于它的因子数之和，这个数就称为"完数"。例如6=1+2+3，编程找出1000以内的所有完数。

**public** **class** algorithm2 {

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

**for**(**int** i=1;i<=1000;i++){

**int** perfect = 0;

**for**(**int** j =1;j<=i/2;j++){

**if**(i%j == 0){

perfect +=j;

}

}

**if**(perfect == i){

System.*out*.println(i);

}

}

}

}

四、控制循环结构

1、使用**Break**结束循环

break语句用于循环体语句中时，当循环语句中执行至break语句时，将立即结束循环，执行循环体外的语句。

2、使用**continue**结束本次循环

Continue语句用于结束本次循环，在循环语句中执行至continue语句时，循环体中continue语句后面的语句不会被执行，而跳到下一次循环。

3、使用**return**结束方法

五、大数值

如果基本的整数和浮点数精度不能满足需求，那么可以使用java.Math包中的两个很有用的类：BigInteger和BigDecimal。这两个类可以处理包含任意长度的数字序列的数值。BigInteger类实现了任意精度的整数运算，BigDecimal实现了任意精度的浮点数运算。

使用静态的valueof()方法可以将普通的数值转化为大数值：

BigInteger a = BigInteger.*valueOf*(100);

BigDecimal b = BigDecimal.*valueOf*(99.99);

大数值不能用算术运算符处理，需由其提供的相应的方法来完成加减乘除等运算。

|  |  |
| --- | --- |
| BigInteger API | |
| BigInteger add(BigeInteger other) | 加法 |
| BigInteger subtract(BigeInteger other) | 减法 |
| BigInteger multiply(BigeInteger other) | 乘法 |
| BigInteger divide(BigeInteger other) | 除法 |
| BigInteger mod(BigeInteger other) | 余数 |
| BigDecimal API | |
| BigDecimal add(BigDecimal other) | 加法 |
| BigDecimal subtract(BigDecimal other) | 减法 |
| BigDecimal multiply(BigDecimal other) | 乘法 |
| BigDecimal divide(BigDecimal other，roundmode) | 除法，需给出舍入方式 |

第四章 数组

一、数组基础知识

1、定义数组的两种方式

**type**[] arrayName;

**type** arrayName[];

一般使用第一种方式。

2、初始化的两种方式

静态初始化：初始化时由程序员显示指定每个数组元素的值，由系统决定需要的数组长度。

动态初始化：初始化时程序员只指定数组长度，由系统为数组元素分配初始值。

语法格式及示例：

语法格式：

1.静态初始化：arrayName=**new** **type**[]{element1,element2,…}或简写为：

arrayName={element1,element2,…}

2.动态初始化：arrayName= **new** **type**[length]

示例：（以下为定义并初始化）

**int**[] x={1,2,23,4,5,56,6,6,7};

**int**[] x=**new** **int**[]{1,2,4,5,6,7,8,8,9,54};

**int**[] y=**new** **int**[10];

注意：在java中允许数组的长度为0，可以new elementType[0];

3、命令行参数

每一个java程序都会在main方法中带一个**String**[] args的参数，这个参数表明main方法将接收一个字符串数组，也就是命令行参数，命令行参数即java程序在编译之后，解释运行的时候所传入的参数，也称运行时参数。

示例：

**public** **class** argsTest {

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

**if**(args[0].equals("-h")){

System.*out*.print("hello,");

}**else** **if**(args[0].equals("-g")){

System.*out*.print("goodbye,");

}

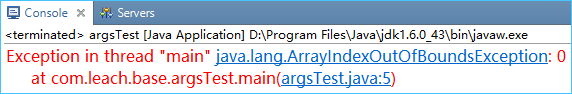
**for**(**int** i=1;i<args.length;i++){

System.*out*.print(" "+args[i]);

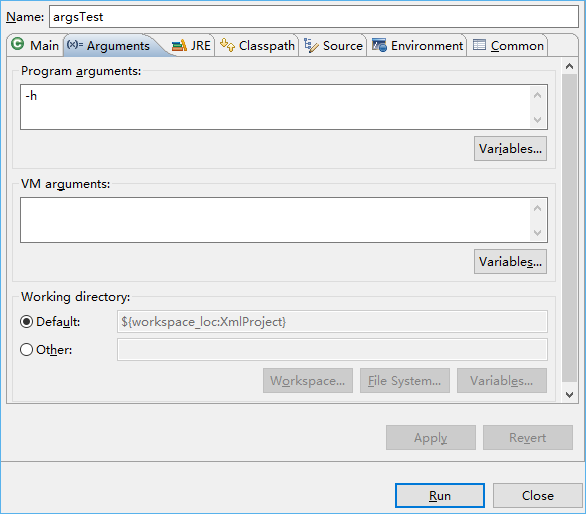
}}

}

首先我们通过myeclipse的Run As->java application命令运行程序会看到：



这是因为运行时，并没有传入运行时参数，所以args没有参数值，为空。接着我们通过myclipse的run as ->run configurations命令，选择arguments,输入-h,如下所示：



单击Run，就能在控制台看到我们代码中编写的结果：hello。

通过命名提示符同样可以进行命令行参数的演示：

1.首先通过cmd命令进入命名提示符控制台

2.选择java文件所在盘符，如D盘，输入D:进入d盘

3.通过cd 文件夹名称进入java文件所在的根目录

4.通过javac java文件名带后缀编译成class文件

5.通过java java文件名不带后缀 参数值运行java程序

如图所示：



4、数组的遍历

数组的遍历都是通过循环的方法来访问数组元素，可以通过for循环，也可以通过foreach循环来访问。

**foreach**语法结构：

**for**(type varribleName:arrays|collection){

//varribleName自动迭代访问每一个元素…

}

例：

**public** **class** argsTest {

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

**int**[] codes = new **int**[]{9,4,3,2,5,6,8,10,12,14};

**for**(**int** code:codes){

System.*out*.print(code);

}

}

}

5、数组排序

对数值型数组排序，可以使用java.util.Arrays工具类的sort()方法，sort方法可以对数组中的部分元素进行排序，需要添加索引参数。

**import** java.util.Arrays;

**public** **class** OrderArrayTest {

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

**int**[] codes = **new** **int**[]{9,8,5,6,7,4,2,1,3};

//Arrays.*sort*(codes,1,5);

Arrays.*sort*(codes);

**for**(**int** code:codes){

System.*out*.print(" "+code);

}

}

}

冒泡排序：其思想就是将相邻的两个数逐次比较，不满足要求就更换顺序，示例：

**public** **class** OrderArrayTest {

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

**int** [] order=**new** **int**[]{1,4,45,6,7,47,61,23,55,33};

**int** temp;

**for**(**int** i=1;i<order.length;i++){

**for**(**int** j=0;j<order.length-i;j++){

**if**(order[j]<order[j+1]){

temp=order[j];

order[j]=order[j+1];

order[j+1]=temp;

}

}

}

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

System.*out*.print(" "+order[i]);

}

}

}

选择排序：主要思想是每一轮循环选出最大和最小的数与对应顺序的值进行交换。示例：

**public** **class** OrderArrayTest {

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

**int** [] order=**new** **int**[]{1,4,56,67,34,23,12,78,97,7};

**int** temp;

**int** k;

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

k=i;

**for**(**int** j=i+1;j<order.length;j++){

**if**(order[k]<order[j]){

k=j;

}

}

**if**(k!=i){

temp=order[k];

order[k]=order[i];

order[i]=temp;

}

}

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

System.*out*.print(" "+order[i]);

}}

}

二、深入数组

1、数组初始化

Java语言是典型的静态语言，因此Java的数组是静态的，即当数组被初始化之后，该数组的长度是不可变的，所谓初始化，就是为数组对象的元素分配内存空间，并为每个数组元素指定初始值。

动态初始化时，系统自动为数组元素分配的默认值：

|  |  |
| --- | --- |
| 数据类型 | 默认值 |
| byte[],short[],int[],long[] | 0 |
| float[],double[] | 0.0 |
| boolean[] | Flase |
| char[] | ‘\u00000’ |
| 引用类型数组 | null |

一旦为数组初始化完成，数组元素的内存空间分配即结束，程序只能改变数组元素的值，而无法改变数组的长度。

Java的数组变量是一种引用类型的变量，数组变量并不是数组本身，它只是指向堆内存中的数组对象。

2、栈内存与堆内存

栈（stack）：栈的优势是，存取速度比堆要快，仅次于直接位于CPU中的寄存器，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。栈数据可以共享。

堆（heap）：堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器。堆数据在没有引用变量指向它的时候，才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

当一个方法执行时，每个方法都会建立自己的内存栈，在这个方法内定义的变量将逐个放入这块栈内存中，随着方法的执行结束，这个方法的内存栈也将自然销毁。因此，所有在方法中定义的变量（非静态变量）都是放在栈内存中的。当创建一个对象时，这个对象将保存到运行时数据区中，以便反复利用，这个运行时数据区就是堆内存。

所有局部变量都是放在栈内存里保存的，不管其是基本类型的变量，还是引用类型的变量，都是存储在各自的方法栈区中；但引用类型变量所引用的对象（包括数组、普通Java对象）则总是存储在堆内存中。

堆内存中的对象（不管是数组对象，还是普通的Java对象）通常不允许直接访问，为了访问堆内存中的对象，通常只能通过引用变量。

引用变量本质上只是一个指针，只要程序通过引用变量访问属性，或者通过引用变量来调用方法，该引用变量将会由它所引用的对象代替。

main方法声明的变量都属于局部变量，因此它们都被保存在main方法栈中；但数组元素则作为数组对象的一部分，总是保存在堆内存中，不管它们是基本类型的数组元素，还是引用类型的数组元素。

3、多维数组

所谓多维数组，其实只是数组元素依然是数组的1维数组，2维数组是数组元素是1维数组的数组，3维数组是数组元素是2维数组的数组，4维数组是数组元素是3维数组的数组……N维数组是数组元素是N−1维数组的数组。

数组的length属性应该返回系统为该数组所分配的连续内存空间的长度。

以下示例可以很好的理解数组在内存中的分配机制

**public** **class** ObjectArrayTest {

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

//定义并初始化一个Object数组

Object[] objArr=**new** Object[3];

//让objArr所引用数组的第2个元素再次指向一个长度为2的Object[]数组

objArr[1]=**new** Object[2];//①

//将objArr[1]的值赋给objArr2，即让objArr2和objArr[1]指向同一个数组对象

Object[] objArr2=(Object[])objArr[1]; //②

//让objArr2所引用数组的第2个元素再次指向一个长度为3的Object[]数组

objArr2[1]=**new** Object[3]; //③

//将objArr2[1]的值赋给objArr3，即让objArr3和objArr2[1]指向同一个数组对象

Object[] objArr3 = (Object[])objArr2[1];//④

//让objArr2所引用数组的第2个元素再次指向一个长度为3的int[]数组

objArr3[1] = **new** **int**[5]; //⑤

//将objArr3[1]的值赋给iArr，即让iArr和objArr3[1]指向同一个数组对象

**int**[] iArr = (**int**[])objArr3[1]; //⑥

//依次为iArr数组的每个元素赋值

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

iArr[i] = i \* 3 + 1;

}

//直接通过objArr访问iArr数组的第3个元素

System.*out*.println(

((**int**[])((Object[])((Object[])objArr[1])[1])[1])[2]);//⑦

}

}

第五章 类与对象

一、面向对象的基本特征

三个基本特征为：封装，继承，多态。封装是将对象的属性即实现细节隐藏起来，然后通过相关的公用方法来暴露该对象的功能；继承是面向对象实现软件复用的重要手段，当子类继承父类后，子类作为一种特殊的父类，将直接获得父类的属性和方法；多态是子类对象可以直接赋给父类变量，但运行时仍然表现出子类的行为特征，这意味着同一个类型的对象在运行时可能表现出不同的行为特征。

二、类与对象之间的关系

抽象的说，类是对象的抽象，对象是类的实现。类是构造对象的模版或蓝图，由类构造对象的过程称为创建类的实例。

三、类与类之间存在的关系

1、依赖：一个类的方法操作另一个类的对象，就说一个类依赖与另一个类。

2、聚合：一个类对象包含另一个类对象，比如A类的一个属性是B类。

3、继承：父子类关系。

四、类的定义

1、定义语法

**[**修饰符①**]** **class** 类名 {

//类属性的声明或定义;

[修饰符②] **属性类型** 属性名 [= 默认值];

//构造器的声明定义；

[修饰符③] 构造器名(形参列表){

执行体…

}

//类方法的声明与定义;

[修饰符④] **返回类型** 方法名(形参列表){

方法执行体…

}

}

2、访问修饰符说明

修饰符①：可以是public、final，或者完全省略这两个修饰符；

修饰符②：可以是public、protect、private、final、static;public、protect、private三者中只能出现其一，可与static、final组合修饰属性；

修饰符③：可省略，可以是public、protect、private其中之一；

修饰符④：可省略，可以是public、protect、private、static、final、abstract;public、protect、private三者中只能出现其一，final、abstract二者只能出现其一，它们可与static组合修饰方法；

3、修饰符关键字说明

|  |  |
| --- | --- |
| public | 最宽松级别，public修饰的成员（成员变量或方法）可以被所以类访问 |
| protect | protect修饰的成员（成员变量或方法）可被同一个包中其他类访问，也可被不同包中子类访问，非子类不可访问，如果用protect修饰一个方法，通常是希望子类重写该方法 |
| default | defalut修饰的成员（成员变量或方法）只能被相同包下的其他类访问 |
| private | private修饰的成员（成员变量或方法）只能在当前类内部访问 |

五、对象的产生与使用

1、语法

创建对象的根本途径是通过构造器，通过new关键字来调用某个类的构造器即可创建该类的实例。Eg.Person p = new Person();

类的属性和方法的调用可以通过类.属性|方法，实例.属性|方法来调用；使用static修饰的方法或属性（注意static不可修饰类），既可使用类.属性|方法来调用，也可使用实例.属性|方法来调用；没有使用static修饰的方法或属性只能通过实例.属性|方法来调用。

2、对象、引用和指针

当一个对象创建成功后，该对象保存在堆内存中，java程序不允许直接访问堆内存中的对象，只能通过该对象的引用来操作该对象。堆内存中的对象可以有多个引用，多个引用可以指向同一个对象。

3、this关键字

this关键字是java对象的默认引用，this关键字总是指向调用该方法的对象。根据this出现的位置的不同，this做为对象的默认引用有两种情况：

* 构造器中引用该构造器执行初始化的对象；
* 在方法中引用调用该方法的对象。

如果在static中直接使用this关键字，则this关键字无法指向合适的对象。所以static修饰的方法不能使用this引用。由于static修饰的方法不能使用this引用，所以static修饰的方法不能访问非static修饰的成员。即静态成员不能访问非静态成员。

4、形参长度可变的方法

从jdk1.5以后，java允许定义形参长度可变的参数，从而允许为方法指定数量不确定的形参。在最后一个形参后加三点（…），表明该形参可以接受多个参数值，多个参数值被当成数组传入。即便是采用形参长度可变形式来定义的方法，调用该方法时依然可以为该方法传入一个数组。数组形式的参数可以位于形参列表任何位置，但个数可变的形参只能处于形参列表的最后。也就是说，一个方法中最多只能有一个可变长度的形参。

示例如下：

**public** **class** VarArgs {

//定义形参长度可变方法

**public** **static** **void** test(**int** a,String... books){

**for**(String temp:books){

System.*out*.println(temp);

}

}

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

*test*(3,"xiaomi","huawei","meizu");

}

}

5、递归方法

一个方法体内调用它自身，被称为方法的递归。定义递归方法的一条最重要规定：递归一定要向已知方向进行。否则会造成死循环。示例：

//问题描述：汉诺塔问题是指有三根杆子A,B,C。C杆上有若干碟子，把所有碟子从A杆上移到C杆上，每次只能移动一个碟子，大的碟子不能叠在小的碟子上面。求最少要移动多少次,且打印出每一步的移动位置？

2:A-B,A-C,B-C

3:A-C,A-B,C-B,A-C,B-A,B-C,A-C

4:A-B,A-C,B-C,A-B,C-A,C-B,A-B,A-C,B-C,B-A,C-A,B-C,A-B,A-C,B-C

**public** **class** DiguiTest {

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

System.*out*.println(*totalFn*(3));

*moveDisk*(4,"A","B","C");

}

//只求移动次数总和

**public** **static** **int** totalFn(**int** n){

**int** sum = 0;

sum = n==1?1:2\**totalFn*(n-1)+1;

**return** sum;

}

//打印移动步骤

**public** **static** **void** moveDisk(**int** diskQuantity,String A,String B,String C){

**if**(diskQuantity == 1){

System.*out*.println("Move disk from "+A+" to "+C);

}**else**{

*moveDisk*(diskQuantity-1,A,C,B);

*moveDisk*(1,A,B,C);

*moveDisk*(diskQuantity-1,B,A,C);

}

}

}

6、方法重载

如果同一个类中包含两个或两个以上的方法的方法名相同，但形参列表不同，则被称为方法重载。至于其他的，如返回值类型、修饰符等与方法重载没有关系。注意，不推荐重载具有形参长度可变的方法。java中最常见的例子就是构造方法的重载。

7、成员变量和局部变量

成员变量是在类范围内定义的变量，即类的属性；局部变量指的是在一个方法中定义的变量。以下是所有变量分类：

|  |  |  |
| --- | --- | --- |
| 所有变量 | 成员变量 | 实例属性（不以static修饰） |
| 类属性（static修饰） |
| 局部变量 | 形参（方法签名中定义的变量） |
| 方法局部变量（在方法内定义的变量） |
| 代码块局部变量（在代码块内定义） |

成员变量可以不用显式的初始化，当系统加载类或创建类实例的时候，系统自动为成员变量分配内存空间，并在分配内存空间后，自动为成员变量指定初始值。

局部变量定义后，必须经过显式初始化才能使用，系统不会为局部变量执行初始化。局部变量不属于任何类或实例，它总是保存在其所在方法的栈内存中。如果局部变量是基本类型，则直接把这个变量的值保存在该变量对应的内存中，如过是引用类型变量，则这个变量存放的是地址，通过该地址引用到该变量实际引用的对象或数组。

栈内存中的变量无需进行系统垃圾回收，栈内存中的变量往往是随方法或代码块的运行的结束而结束的。

六、封装

将对象内部的状态信息隐藏在对象内部，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问。例：

**public** **class** Token {

**private** String accessToken; //接口访问凭证

**private** **int** expiresIn; //凭证有效期，单位:秒

**public** String getAccessToken() {

**return** accessToken;

}

**public** **void** setAccessToken(String accessToken) {

**this**.accessToken = accessToken;

}

**public** **int** getExpiresIn() {

**return** expiresIn;

}

**public** **void** setExpiresIn(**int** expiresIn) {

**this**.expiresIn = expiresIn;

}

}

七、继承

1、继承的特点

Java继承通过extends关键字来实现，实现继承的类被称为子类，被继承的类称为父类。语法格式如下：

**[**修饰符**]** **class** subClass **extends** superClass {

//类定义部分

}

2、重写父类方法

子类包含与父类重名的方法的现象称为方法重写（Override）。重写遵循“两同两小一大”原则，两同即方法名、形参列表相同；两小即子类方法返回值类型应比父类方法返回值类型的范围更小或相同，子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相同；一大即子类方法的访问权限应比父类方法更大或相同。特别注意的是，覆盖方法和被覆盖方法要么都是类方法，要么都是实例方法，不能一个是类方法，一个是实例方法。

当子类覆盖了父类方法后，依然可以在子类的方法中调用父类中被覆盖的方法，使用super关键字调用。如果父类方法具有private权限，则该方法对子类也是隐藏的，子类无法访问该方法，也无法重写该方法。即便子类中有与父类同名的方法也不是重写。

如果子类定义了和父类同名的属性，也会发生子类属性覆盖父类属性的的情形。在子类定义的实例方法中可以通过super来访问父类被覆盖的属性。

3、调用父类构造器

子类不会获得父类的构造器，但有时候子类构造器需要调用父类构造器的初始化代码。示例：

**class** base{

**private** String name;

**private** **int** age;

**public** base(String name,**int** age){

**this**.name = name;

**this**.age = age;

}

}

**class** subBase **extends** base{

**private** String sex;

**public** subBase(String name, **int** age,String sex) {

**super**(name, age);

**this**.sex = sex;

}

}

在上述代码中，如果我们不为subBase创建构造器，系统会提示我们编译错误，因为直接父类具有显示构造器，系统此时并不会再为我们创建无参的隐式构造器，所以此时子类可以调用父类构造器的初始化代码。我们所见的哪些没有显示构造器而没有编译错误的类，其实都会有系统为它们创建了隐式的无参构造器，这个过程就是通过super关键字来隐式调用它们父类的构造器。

子类构造器调用父类构造器分以下几种情况：

* 子类构造器执行体中第一行使用super显式调用父类构造器，系统将根据super调用传入的实参列表调用父类对应的构造器。
* 子类构造器执行体中第一行代码使用this显式调用本类中重载的构造器，系统将根据this调用里传入的实参列表调用本类另一个构造器。执行本类中另一个构造器时即会调用父类的构造器。
* 子类构造器既没有super调用，也没有this调用，系统将会在执行子类构造器之前，隐式的调用父类无参的构造器。

八、多态

Java引用变量有两个类型：一个是编译时的类型，一个是运行时的类型，编译时的类型由声明该变量时使用的类型决定，运行时类型由实际赋给该变量的对象决定。如果编译时类型和运行时类型不一直，就会出现所谓的多态。

Java允许把一个子类对象直接赋值给一个父类引用变量，无须任何类型转换，该过程由系统自动完成。

九、代码块与静态代码块

1、基本概念

代码块也称初始化块，与构造器相似都是对对像或类进行初始化操作，它是java对象中的第四种成员，一个类里可以有多个初始化块，相同类型的初始化块之间有顺序：前面定义的初始化块先执行，后面定义的初始化块后执行。语法格式：

**[**修饰符**]** {

//初始化块的可执行代码

}

初始化块的修饰符只能是static,static修饰的代码块称为静态代码块。初始化块只在创建java对象时隐式执行，而且在构造器之前执行。

2、初始化块与构造器

初始化块总是在构造器之前执行，系统同样可以使用初始化块来进行对象的初始化操作。与构造器不同的是，初始化块不接受任何参数。

与构造器相似，创建一个java对象时，不仅会执行该类的初始化块和构造器，系统会一直上溯到java.lang.Object类，先执行Object类的初始化块，执行object类的构造器，依次向下执行其父类的的初始化块和构造器，最后才执行该类的初始化块和构造器。

3、静态初始化块

静态初始化块是类相关的，系统将在类初始化阶段执行静态初始化块，而不是在创建对象时才执行。因此静态代码块总是比普通初始化块先执行。

静态初始化块不能访问非静态成员，不能对实例属性进行初始化处理。

//代码块测试

**public** **class** CodeBlockTest {

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

**new** chlidClass("O","jamas");

}

}

**class** ParentClass{

**private** String bloodType;

**private** **static** String *surname*;

//构造器

**public** ParentClass(String bloodType,String surname){

**this**.bloodType = bloodType;

**this**.*surname* =surname;

System.*out*.println("parent的构造器");

}

//父类普通代码块

{

System.*out*.println("parent的普通代码块");

}

//父类静态代码块

**static**{

*surname* = "Thomas";

System.*out*.println("parent的静态初始化块,surname="+*surname*);

}

}

**class** chlidClass **extends** ParentClass{

//构造器

**public** chlidClass(String bloodType, String surname) {

**super**(bloodType, surname);

System.*out*.println("chlid的构造器");

}

//初始化块

{

System.*out*.println("child的初始化块");

}

//静态代码块

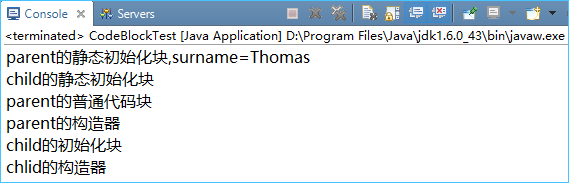
**static**{

System.*out*.println("child的静态初始化块");

}

}

运行结果：



九、内存中的对象

1、对象与垃圾回收

垃圾回收只负责回收堆内存中的对象，不会回收任何物理资源（例如数据库连接、网络io等资源）。

程序无法精确控制垃圾回收的运行，垃圾回收会在何时的时候运行。当对象永久性的失去后、系统就会在合适的时候回收它所占有的内存。

垃圾回收回收任何对象之前，总会调用它的finalize方法，该方法可以是对象重新复合（让一个引用变量重新引用该对象），从而导致垃圾回收机制取消回收。

2、对象在内存中的状态

激活状态：

当一个对象被创建后，有一个以上的引用变量引用它，则这个对象在程序中处于激活状态，程序可通过引用变量来调用该对象的属性和方法。

去活状态：

如果程序中某个对象不再有任何引用变量引用它，它就进入了去活状态。在这个状态下，系统的垃圾回收机制准备回收该对象所占用的内存，在回收该对象之前，系统会调用所有去活状态对象的finalize方法进行资源清理，如果系统在调用finalize方法重新让一个引用变量引用该对象，则这个对象会再次变为激活状态，否则进入死亡状态。

死亡状态：

当对象的所有引用变量的关联被切断，且系统已经调用所有对象的finalize方法依然没有使该对象变成激活状态，那这个对象将永久性失去引用，最后变为死亡状态。只有当一个对象处于死亡状态时，系统才会真正回收该对象占有的资源。

强制垃圾回收：

程序无法精确的控制垃圾回收的时机，但我们依然可以强制系统进行垃圾回收——只是这种强制是通知系统进行垃圾回收，但系统是否进行垃圾回收依然不确定。两种方法：

System类的gc()静态方法：System.gc();

Runtime对象的gc()实例方法：Runtime.getRuntime().gc()

第六章 面向对象（下）

一、基本数据类型的包装类

1、对应包装类

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| byte | short | int | long | float | double | boolean | char |
| Byte | Short | Integer | Long | Float | Double | Boolean | Character |

2、基本类型到包装类的转换

**public** **void** baseType2packType(){

**char** ch = 'A';

**int** i = 20;

**float** f = 20.55f;

**boolean** bl= **true**;

//通过new关键字转化

Character cter = **new** Character(ch);

Integer iter = **new** Integer(i);

Float fter = **new** Float(f);

Boolean blter = **new** Boolean(bl);

//自动装箱

Character cterAuto = ch;

Integer iterAuto = i;

Float flterAuto = f;

Boolean blterAuto = bl;

}

3、包装类到基本类型的转换

**public** **void** packType2baseType(){

Character cterAuto = 'B';

Integer iterAuto = 20;

Float flterAuto = 19.55f;

Boolean blterAuto = **true**;

//通过对象的xxxValue方法转换

**char** ch = cterAuto.charValue();

**int** i = iterAuto.intValue();

**float** f = flterAuto.floatValue();

**boolean** bl = blterAuto.booleanValue();

//自动拆箱

**char** chAuto = cterAuto;

**int** iAuto = iterAuto;

**float** fAuto = flterAuto;

**boolean** blAuto = blterAuto;

}

4、包装类实现基本数据类型与字符串之间的转换

除了Character类外，所有包装类都提供的一个parseXXX(String s)静态方法，用于将字符串转换成基本数据类型；String类也提供了大量重载的valueOf（）方法，用于将基本类型转换成字符串。

**public** **static** **void** baseType1String(){

String si = "110";

String sf = "14.55";

String sb = "true";

//转换成基本类型

**int** ii = Integer.*parseInt*(si);

**float** ff = Float.*parseFloat*(sf);

**boolean** bb = Boolean.*parseBoolean*(sb);

System.*out*.println("ii="+ii+"\nff="+ff+"\nbb="+bb);

//转化成字符串

String is = String.*valueOf*(ii);

String fs = String.*valueOf*(ff);

String bs = String.*valueOf*(bb);

System.*out*.println("is="+is+"\nfs="+fs+"\nbs="+bs);

}

二、单例类

如果一个类始终只能创建一个实例，这个类被称为单例类。

实现原理：第一，将类的构造器用private修饰，这样其他类就不能通过该类的构造器来创建该类的实例；第二，需要提供一个public方法做为该类的访问点，用于创建该类的唯一对象，且该方法必须用static修饰（因为调用该方法时，该类的实例并不存在，因此调用该方法的不可能是对象，只能是类）；第三，该类还必须缓存已经创建的对象，否则该类无法知道曾经是否已经创建过对象，也就无法保证只创建一个对象。

懒汉模式（不考虑线程安全）：

**public** **class** Singleton {

//缓存曾经创建过实例的变量

**private** **static** Singleton *instance*;

**private** Singleton(){}

//保证能创建实例的静态方法

**public** **static** Singleton getInstance(){

**if**(*instance* == **null**){

*instance* = **new** Singleton();

}

**return** *instance*;

}

}

保证线程安全的懒汉模式：

//懒汉模式，考虑线程安全，方法加synchronized

**class** Singleton1{

**private** **static** Singleton1 *instance*;

**private** Singleton1(){}

//同步方法

**public** **static** **synchronized** Singleton1 getInstance(){

**if**(*instance* == **null**){

*instance* = **new** Singleton1();

}

**return** *instance*;

}

}

//懒汉模式，考虑线程安全，双重检查锁定

**class** Singleton2{

**private** **static** Singleton2 *instance*;

**private** Singleton2(){}

**public** **static** Singleton2 getInstance(){

**if**(*instance* == **null**){

//同步代码块

**synchronized**(Singleton2.**class**){

**if**(*instance* ==**null**){

*instance* = **new** Singleton2();

}

}

}

**return** *instance*;

}

}

//懒汉模式，考虑线程安全，静态内部类

**class** Singleton3{

**private** Singleton3(){}

//静态内部类

**private** **static** **class** LazyHolder {

**private** **static** **final** Singleton3 *INSTANCE*=**new** Singleton3();

}

**public** **static** Singleton3 getInstance(){

**return** LazyHolder.*INSTANCE*;

}

}

饿汉模式（天生线程安全）：

**public** **class** Singleton {

//缓存曾经创建过实例的变量

**private** **static** Singleton *instance =* **new** Singleton();

**private** Singleton(){}

//保证能创建实例的静态方法

**public** **static** Singleton getInstance(){

**return** *instance*;

}

}

三、final修饰符

final关键字可用于修饰类、变量和方法。

1、final修饰变量

final修饰变量时，表示该变量一旦获得了初始值之后就不可被改变。final既可修饰成员变量，也可修饰局部变量、形参。

final修饰成员变量

成员变量是随类初始化或对象初始化而初始化的。因此，成员变量的值可以在定义该变量时指定，也可以在初始化块和构造方法中指定。所以对final修饰的成员变量不可在普通方法中对成员变量赋值。当使用final修饰成员变量时，要么在定义成员变量的时候指定初始值，要么在初始化块、构造器中为成员变量赋初始值。final修饰的类属性、实例属性赋初始值的地方：

* 类属性：静态代码块中、声明该属性时指定；
* 实例属性：非静态代码块，声明该属性、构造器中指定初始值。

final修饰的成员变量，必须由程序员显式初始化，系统不会对final成员变量进行隐式初始化。

示例：

**public** **class** FinalPropertyTest {

**final** **int** a = 6;

**final** String str;

**final** **int** c;

**final** **static** **double** *d*;

{c = 9;}

**public** FinalPropertyTest(String str){

**this**.str = str;

}

**static**{*d* = 20.5;}

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

FinalPropertyTest ft = **new** FinalPropertyTest("hello");

System.*out*.println(ft.a+"\n"+ft.str+"\n"+ft.c+"\n"+ft.*d*);

}

}

Final修饰局部变量

系统不会对局部变量进行初始化，必须有程序员显式初始化。因此使用final修饰局部变量既可以在定义是指定默认值，也可以不指定默认值。如果定义是没有指定默认值，后续代码中可以对其初始化，但只能赋值一次。final修饰的形参不能被赋值，由方法调用时传入指定值。

Final修饰基本类型和引用类型的区别

Final修饰基本类型时，不能对基本类型变量重新赋值，因此基本类型变量不能被改变。对于引用类型的变量而言，它保存的仅仅是一个引用，final只保证这个引用引用的地址不会发生改变，即一直引用同一个对象，但这个对象完全可以发生改变。

2、final方法

final方法不可被重写，javat提供的Object类就有一个final方法：getClass();

3、final类

final修饰的类不可以有子类，例如java.lang.Math类就是一个final类。

4、不可变类

不可变类是值创建该类的实例后，该实例的属性是不可改变的。Java提供的八个包装类和java.lang.String类都是不可变类。

创建自定义类遵循规则：

* 使用private和final修饰属性；
* 提供带参构造器，用于传入参数来初始化类里的属性；
* 提供getter方法，不提供setter方法；
* 如果有必要，重新Object类中hashCode和equals方法。

四、抽象类

1、概念及语法

抽象方法是抽象类的重点，抽象方法和抽象类都必须用abstract修饰，有抽象方法的类只能被定义成抽象类，抽象类中可以没有抽象方法，但是这样抽象类将毫无意义。

语法格式：

**[**修饰符**] abstract** **class** 类名 {

//类属性的声明或定义;

//构造器的声明定义；

//抽象方法：只有方法签名，没有执行体；

[修饰符] **abstract 返回类型** 方法名(形参列表)；

}

2、抽象类、抽象方法规则说明

* 抽象类不能被实例化，无法使用new关键字来调用抽象类的构造器创建抽象类实例；
* 可以包含属性，构造器，方法（普通方法和抽象方法），初始化块，内部类，枚举类；抽象类的构造器不能用于创建实例，主要是用于被其子类调用；
* 含有抽象方法的类只能被定义成抽象类，即含有抽象方法的类一定不是普通类，因此普通类也一定不会有抽象方法；
* 由于final修饰的方法不能被重写，final修饰的类不能有子类，所以final与abstrat不能共存；
* 由于static修饰的方法是类方法，抽象类是不能被实例化的，所以static与abstract不能同时修饰同一个方法；
* abstract不能用于修饰属性；

五、接口

1、接口的定义

**[**修饰符**] interface** 接口名 **extends** 父接口1,父接口2…{

//零到多个常量定义…

//零到多个抽象方法定义…

[修饰符] **abstract 返回类型** 方法名(形参列表)；

}

2、接口规则说明

* 接口中的方法只能是抽象方法；接口中的属性只能是常量；
* 接口中方法可以不显式使用abstract修饰，系统会自动使用abstract修饰；
* 接口里的属性系统默认使用public static final修饰的；可以不显式使用public static final;
* 接口中没有构造器和初始化块；因此属性只能在定义时初始化，即所有属性都该是常量;

3、使用接口

接口不能用于创建实例，但是可以用于声明引用类型变量。当使用接口声明引用类型变量时，这个引用变量必须引用到其实现类的对象。实现类使用implements关键字，一个类可以实现多个接口。

语法格式：

**[**修饰符**] class** 类名 **extends** 父类 **implements** 接口1，接口2…{

……

}

六、内部类

1、概念阐述

定义在其他类内部的类被称为内部类；内部类作用：

* 内部类提供更好的封装，可以把内部类隐藏在外部类之内，不允许同一个包中的其他类访问该类；
* 内部类成员可以直接访问外部类的私有数据，外部内不能访问内部类的实现细节；
* 匿名内部类可用于仅需要一次调用的类；

2、内部类分类

内部类可以定义在类中的任何位置，甚至可以定义在方法中，通常内部类都作为成员内部类定义，而不是作为局部内部类。

|  |  |  |
| --- | --- | --- |
| 内部类 | 成员内部类 | 静态内部类 |
| 非静态内部类 |
| 局部内部类 | |
| 匿名内部类 | |

3、非静态内部类

语法即示例：

**public** **class** NSInnerClass{

**private** String name;

**private** String age;

//非静态内部类

**class** Secret{

**private** String name;

**private** String date;

{name = "小明";}

**public** Secret(String name,String date){

**this**.name =name;

**this**.date =date;

}

**private** **void** doSomething(){

System.*out*.println(NSInnerClass.**this**.name+"在"+date+name+"嘿嘿嘿，你懂的...");

}

}

**public** **void** test(){

Secret s = **new** Secret("seeMovie","2007-10-09");

s.doSomething();

}

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

NSInnerClass ts = **new** NSInnerClass();

ts.test();

}

}

注意事项：

* 编译包含内部类文件会生成两个.class文件，成员内部类的编译文件总是这种形式：OuterClass$InnerClass.class;
* 非静态内部类可以直接访问外部类的private成员；
* 非静态内部类的方法访问某个变量时，先查找当前方法是否存在该名字的局部变量，如果存在则使用；如果不存在，则到该方法所在内部类中查找是否存在该名字的属性，如果存在则使用；如果不存在，则到该内部类所在的外部类查找是否存在该名字的属性，如果存在则使用；如果不存在，系统将出现编译错误：提示找不到该变量；
* 如果外部类属性、内部类属性、内部类方法的局部变量同名，可通过this、外部类类名.this作为限定区分；
* 非静态内部类中不能有静态方法、静态属性、静态代码块；

4、静态内部类

使用static修饰的成员内部类为静态内部类，是与类相关的。静态内部类可以包含非静态成员，静态内部类不能访问外部类的实例成员，只能访问外部内的类成员。

接口中允许定义内部类，但是只能是静态内部类；

5、局部内部类

定义在方法中的内部类称为局部内部类，局部内部类仅在该方法中有效。局部内部类不能在外部类以外的地方使用，那么局部内部类也无需使用访问控制符和static修饰。

局部内部类的编译文件命名规则为：outerClass$NinnerClass.class,N为随机数（1，2，3…）;

6、匿名内部类

匿名内部类语法格式：

**new** 父类构造器（实参列表）|实现接口（）{

//匿名内部类类体部分

}

**interface** Product{

**public** **double** getPrice();

**public** String getName();

}

**public** **class** NoneInnerClass {

**public** **void** test(Product p){

System.*out*.println("购买一个"+p.getName()+"花掉了"+p.getPrice());

}

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

NoneInnerClass nic = **new** NoneInnerClass();

//匿名内部类类体部分

nic.test(**new** Product(){

**public** **double** getPrice(){

**return** 1466;

}

**public** String getName(){

**return** "Inter core i5 4460 CPU";

}

});

}}

匿名内部类必须继承一个父类，或实现一个接口，但最多只能继承一个父类，或实现一个接口。

匿名内部类不能是抽象类，因为系统创建匿名内部类的时候会立即创建匿名内部类的对象。

匿名内部类不能定义构造器，因为匿名内部类没有类名，但匿名内部类可以定义实例初始化块。

七、枚举类

1、概念

把实例有限且固定的类称之为枚举类。枚举类用关键字enum定义，枚举类是一种的特殊的类，但一样可以有自己的属性和方法，一样可以实现一个或多个接口，也可以定义自己的构造器。

枚举类与普通类的区别：

* 可以实现一个或多个接口，使用enum定义的枚举类默认继承了java.lang.Enum类，而不是Object类，Enum类实现了java.lang.serializable和java.lang.Comparable类。
* 枚举类的构造器只能用private访问控制符，若省略，默认为private,显式指定，只能用private。
* 枚举类的实例必须在枚举类中显式列出，否则这个枚举类永远不能产生实例。列出的实例，系统会自动添加public static final 修饰。
* 所有枚举类都提供了一个values方法，该方法可方便遍历枚举值。

2、枚举类的属性、方法、构造器

枚举类通常设计为不可变类，也就说明它的属性值不应该允许改变，这样会更安全。所以枚举类的属性都应该用private final修饰。

**public** **enum** EnumGendar {

//枚举类的实例

*MALE*("男"),*FAMALE*("女");

**private** **final** String name;

//带参的构造函数

**private** EnumGendarTest(String name){

**this**.name = name;

}

**public** String getName(){

**return** **this**.name;

}

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

//枚举类的调用

EnumGendar egt = Enum.*valueOf*(EnumGendar.**class**,"MALE");

System.*out*.println(egt.getName());

}

}

一旦属性使用final修饰，属性的初始化就只能由带参的构造器来完成。

3、实现接口的枚举类

如果由枚举类实现接口里的方法，则每个枚举值在调用该方法时，都有相同的行为方法，如果需要每个枚举值在调用该方法时呈现不同的行为方式，则可以让每个枚举值来实现该方法。例：

**interface** Season{

**public** **void** info();

}

//枚举类实现接口

**public** **enum** EnumSeasonImpl **implements** Season {

//匿名内部类为每个实例实现方法，定义不同的行为

*SPRING*{

**public** **void** info(){

System.*out*.println("春暖花开，蝴蝶自来");

}

},

*SUMMER*{

**public** **void** info(){

System.*out*.println("炎炎夏日，烈日 灼灼");

}

},

*FALL*{

**public** **void** info(){

System.*out*.println("一叶知秋，金风送爽");

}

},

*WINTER*{

**public** **void** info(){

System.*out*.println("寒冬腊月，傲雪凌霜");

}

};

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

//枚举类单个实例调用

EnumSeasonImpl esi = Enum.*valueOf*(EnumSeasonImpl.**class**, "SPRING");

esi.info();

//枚举类所有实例遍历

**for**(EnumSeasonImpl es:EnumSeasonImpl.*values*()){

es.info();

}

}

}

4、包含抽象方法的枚举类

枚举类里定义抽象方法时无需显式使用abstract关键字将枚举类定义成抽象类，但因枚举类需显式创建枚举值，而不是作为父类，所以定义每个枚举值时必须为抽象方法提供实现，否则将出现编译错误。例：

**public** **enum** EnumOperation {

*PLUS*{

**public** **double** eval(**double** x,**double** y){

**return** x+y;

}

},*MINUS*{

**public** **double** eval(**double** x,**double** y){

**return** x-y;

}

},*TIMES*{

**public** **double** eval(**double** x,**double** y){

**return** x\*y;

}

},*DIVIDE*{

**public** **double** eval(**double** x,**double** y){

**return** x/y;

}

};

//枚举类中抽象方法

**public** **abstract** **double** eval(**double** x,**double** y);

//测试方法

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

**for**(EnumOperation eo:EnumOperation.*values*()){

**double** result = eo.eval(9, 3);

System.*out*.println(result);

}

}

}

第七章 泛型

一、泛型入门

1、为什么需要泛型

在没有泛型之前，一旦把一个对象保存到集合中，集合就会忘记对象的类型，把所有的对象当成Object处理。当程序从集合中取出对象后，需要进行强制类型转换，这种强制类型转换不仅代码臃肿，而且很容易造成ClassCastException异常。

2、泛型的好处

增加泛型支持后的集合，完全可以记住集合中元素的类型，并可以在编译时检查集合中元素的类型，如果试图向集合中添加不满足类型需求的对象，编译器就会提示错误。增加泛型支持后的集合，让代码更简洁，程序更健壮。除此外java泛型还增加了枚举类、反射等方面的功能。

3、泛型的概念

把java的参数化类型称之为泛型，所谓参数化类型即创建集合时允许为之指定集合元素的类型。

二、定义泛型接口、类

1、java集合使用泛型的部分代码示例：

//定义接口时指定一个类型形参，形参名为E

**public** **interface** List<E> **extends** Collection<E> {

//E可以作为返回类型，E可以作为形参类型

Iterator<E> iterator();

**boolean** add(E e);

List<E> subList(**int** fromIndex, **int** toIndex);

}

//定义接口时指定两个类型形参，形参名为K,V

**public** **interface** Map<K,V> {

//K, V可以作为返回类型，K,V可以作为形参类型

Set<K> keySet();

V put(K key, V value);

}

泛型实质：允许在定义接口、类时指定类型形参，类型形参在整个接口、类体内中可以当成类型使用，几乎所有可使用其他普通类型的地方都可以使用这种类型形参。

2、自定义泛型类

我们可以为任何类增加泛型声明，并不是只有集合类才可以使用泛型声明，虽然泛型是集合类的重要使用场所。

当创建带泛型声明的自定义类，为该类定义构造器时，构造器名还是原来的类名，不要增加泛型声明。通过构造器实例化对象的时候可以使用Apple<T>形式。

示例：

**public** **class** Apple<T> {

**private** T info;

**public** Apple(T info){

**this**.info = info;

}

**public** T getInfo(){

**return** **this**.info;

}

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

Apple<String> ap1 = **new** Apple<String>("apple");

Apple<Integer> ap2 = **new** Apple<Integer>(2);

}

}

3、从泛型类派生子类

当创建了带泛型声明的接口、父类之后，可以为该接口创建实现类，或从该父类派生子类，父类或接口的类型参数应特别注意，如下：

//父类不带类型参数

**class** AppleA **extends** Apple{

//……

}

}

//父类带类型形参，子类也要带类型形参，否则编译出错

**class** AppleB<T> **extends** Apple<T>{

**public** AppleB(T info) {

**super**(info);

}

}

//父类带类型实参，即指定具体的类型参数

**class** AppleC **extends** Apple<String>{

**public** AppleC(String info) {

**super**(info);

}

}

4、实际并不存在泛型类

系统并不会为ArrayList<String>生成新的class文件，也不会把ArrayList<String>当成新类来处理。不管泛型类型的实际类型参数是什么，它们在运行时总有同样的类。

类的静态方法、静态初始化块和静态变量的声明和初始化都不允许使用类型参数。

instanceof运算符后不能使用泛型类。

三、类型通配符

1、使用类型通配符

类型通配符是一个问号（？），是为了表示某一个类的各种泛型类的父类，例如：List<String>,List<Integer>,都是类型参数类型已知的，而List的所有泛型类可以用List<?>，List<?>可以代表List<String>,List<Integer>中的任意。

带通配符的List近代表它是各种泛型List的父类，并不能将元素加入其中。

2、使用类型通配符上限

被限制的泛型通配符表示：

//它表示所有sharp泛型List的父类

List<? **extends** sharp>

3、设定类型形参的上限

Java泛型不仅允许在使用通配符形参时设定类型上限，也可以在定义类型形参时设定上限。用于表示传入该类型形参的实际类型必须是该上限类型，或该上限类型的子类。

**class** Apple1<T **extends** Number>{

T col;

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

Apple1<Integer> ai =**new** Apple1<Integer>();

Apple1<Double> ad =**new** Apple1<Double>();

//编译错误

Apple1<String> ac =**new** Apple1<String>();

}

}

四、泛型方法

有这样一种情况，我们在定义类，接口时没有使用泛型，但定义方法时想自己定义类型形参，也是可以的。

1、定义泛型方法

泛型方法用法格式：

修饰符 <T,S> 返回值类型 方法名(形参列表){

//方法体

}

**public** **class** MethodGeneric {

//声明一个泛型方法

**public** **static** <T> **void** fromArrayToCollection(T[] a,Collection<T> c){

**for**(T o:a){

c.add(o);

}

}

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

Object[] oa = **new** Object[20];

Collection<Object> co = **new** ArrayList<Object>();

*fromArrayToCollection*(oa,co);

String[] sa = **new** String[20];

Collection<String> cs = **new** ArrayList<String>();

*fromArrayToCollection*(sa,cs);

*fromArrayToCollection*(sa,co);

Integer[] ia = **new** Integer[20];

Double[] da = **new** Double[20];

Collection<Number> cn = **new** ArrayList<Number>();

*fromArrayToCollection*(ia,cn);

*fromArrayToCollection*(da,cn);

//编译出错，

*fromArrayToCollection*(da,cs);

}

}

2、泛型方法与类型通配符的区别

通配符是用来支持灵活的子类化的。

泛型方法允许类型形参被用来表示方法的一个或多个参数之间的类型依赖关系，或者方法返回值与参数之间的类型依赖关系。如果没有这种依赖关系，不应该使用泛型。

3、设定通配符下限

**public** **static** <T> T copy(Collection<T> dest,Collection<? **extends** T> src){

T last = **null**;

**for**(T el:src){

last = el;

dest.add(el);

}

**return** last;

}

//设定通配符下限

**public** **static**<T> T copy(Collection<? **super** T> dest,Collection<T> src){

T last = **null**;

**for**(T el:src){

last = el;

dest.add(el);

}

**return** last;

}