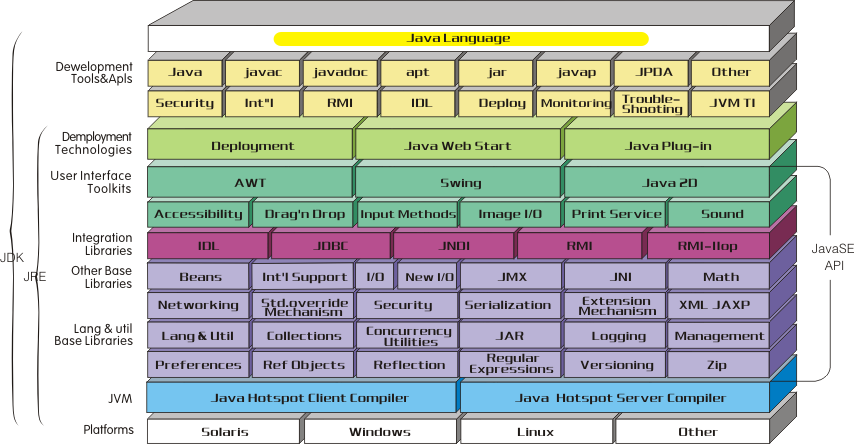
# Information

## Java语言概述



## Java的特点

**Java是简单易用的**

Java中不使用指针，没有多继承，并且提供了自动垃圾回收功能，可以使程序员不必为内存管理而操心。

Java是面向对象的

面向对象指Java以对象为基本粒度，对象中包含属性和方法，对象的说明用属性表达，然后使用方法来操作对象。

Java提供类、接口和继承等面向对象的特性，只支持类之间的单继承，但支持接口之间的多继承，并支持类与接口之间的实现。Java还全面支持动态绑定。

Java是分布式的

Java语言支持Internet应用的开发，在基本的Java应用编程接口中有一个网络应用编程接口（java net），它提供了用于网络应用编程的类库，包括URL、URLConnection、Socket、ServerSocket等。Java的RMI（远程方法激活）机制也是开发分布式应用的重要手段。

**Java是健壮的**

Java的强类型机制、异常处理、垃圾的自动收集等是Java程序健壮性的重要保证。对指针的丢弃是Java的明智选择。

**Java是安全的**

Java通常被用在网络环境中，为此，Java提供了一个安全机制以防恶意代码的攻击。如：安全防范机制（类ClassLoader），如分配不同的名字空间以防替代本地的同名类、字节代码检查。

Java安全性分为四个层面，即语言级安全性、编译时安全性、运行时安全性和可执行代码安全性。

语言级安全性指Java的数据结构是完整的对象，这些封装过的数据类型具有安全性；

编译时安全性指编译时要进行Java语言和语义的检查，保证每个变量对应一个相应的值；

运行时安全性指Java类运行需要由类加载器加载，并由字节码校验器校验；

可执行代码安全性指Java在网络上使用时，对其权限进行了设置，从而保证了用户的安全。

**Java是跨平台性的**

Java文件被编译为字节码文件，然后在任何实现这个Java平台的系统中运行。不同平台上的Java编写内容是相同的，但是不同平台上的jvm是不同的。

**Java是解释型的**

Java程序被编译为字节码格式，然后在实现这个Java平台的任何系统的解释器中运行。

js文件：浏览器直接对js 源码进行解释执行，浏览器会对js 的每一句话进行翻译，然后执行js-> 解释->执行。从理论上说，解释型语言执行速度较慢。

c文件：得到一个二进制的目标文件xx.obj，这种文件可以直接调用内存执行，执行速度快。

java文件：得到一个字节码文件.class，该文件不是一个完全的二进制文件。先JVM 加载.class，再执行.class 文件，速度在编译和解释之间。

**Java是性能略高的**

与那些解释型的高级脚本语言相比，Java的确是高性能的。事实上，Java的运行速度随着JIT(Just-In-Time）编译器技术的发展越来越接近于C++。

**Java是原生支持多线程的**

线程是一种特殊的对象，它必须由Thread类或其子（孙）类来创建或者使用接口Runnable。

**Java是强数据类型的**

强数据类型：在编译期间就要确定数据类型，而弱数据类型是指在编译期间不知道数据类型，在运行时才知道数据类型。

## JDK,JRE,JVM

java develoment kit：Java开发工具集，包括javac\java\javadoc\javap。

java runtime enviroment：Java运行环境，只要装有jre的OS就可以运行Java程序。

java virtual machine：Java虚拟机，用于加载class（字节码）文件并执行代码，同时和操作系统交互。

## Java API文档

应用程序编程接口API （Application Programming Interface）是 Java 提供的基本编程接口。

Java语言提供了大量的基础类，因此 Oracle公司 也为这些基础类提供了相应的API文档，用于告诉开发者如何使用这些类，以及这些类里包含的方法。

## 注释Javadoc

注释（comment）分为行注释，段注释和文档注释（Java特有），注释内容可以被JDK提供的工具 javadoc 所解析，生成一套以网页文件形式体现的该程序的说明文档。eclipse中行注释快捷键为ctrl+/；段注释快捷键为ctrl+shift+/，取消段注释快捷键为ctrl+shift+\。

**使用细节**

对于单行和多行注释，被注释的文字，不会被JVM（java虚拟机）解释执行。

多行注释里面不允许有多行注释嵌套

## Java程序开发

### 注意事项

Java源文件以“java”为扩展名。源文件的基本组成部分是类（class），如本类中的Hello类。

Java应用程序的执行入口是main()方法。它有固定的书写格式：public static void main(String[] args) {...}

Java语言严格区分大小写。

Java方法由一条条语句构成，每个语句以“;”结束。

大括号都是成对出现的，缺一不可。可通过 ctrl + } 来快速配对.

一个源文件中最多只能有一个public类。其它类的个数不限。

如果源文件包含一个public类，则文件名必须按该类名命名!

一个源文件中最多只能有一个public类。其它类的个数不限，也可以将main方法写在非public类中，然后指定运行非public 类，这样入口方法就是非public 的main方法。

### 转义字符

\u表示Unicode字符 \n换行 \t表示tab键（制表） \b退格

\r回车 \”双引号 \’单引号 \\反斜杠

示例：System.out.println("张无忌赵敏周\r芷若小昭");

cmd中输出结果是：芷若小昭敏周

eclipse中输出结果是：张无忌赵敏周

芷若小昭

### 标识符

Java 对各种**变量**、**方法**和**类**等要素命名时使用的字符序列称为标识符，凡是自己可以起名字的都叫标识符

命名规则

由26个英文字母大小写，0-9 ，\_或 $ 组成

数字不可以开头。

不可以使用关键字和保留字，但能包含关键字和保留字。

Java中严格区分大小写，长度无限制。

标识符不能包含空格。

**大小写规则**

项目名、包名全部小写。

类名、接口首字母大写，如果类名由多个单词组成，每个单词的首字母都要大写。

变量名、方法名首字母小写，如果名称由多个单词组成，第二个单词开始每个单词的首字母都要大写。

常量名全部大写，多个单词时每个单词用下划线连接。

### 变量

变量 = 数据类型 + 名称 + 值（变量3要素）

变量(variable)使用常规步骤：

声明变量；

赋值；

使用。

也可以将声明和赋值合并为一步。

变量相当于内存中一个数据存取的空间的表示；变量可以在同一类型范围类任意变化，类型不匹配的本质是内存空间不一样；变量在同一个作用域中不能同名。

基本数据类型的变量存放在栈空间，引用数据类型变量存放在堆变量。

Java中**数值型（包括整数型和浮点型）**变量赋值时可以使用“\_”作为分隔符，虚拟机在编译时会自动忽略“\_”，例如：

int i = 100\_000\_000;

虚拟机编译的结果为：

int i = 100000000；

注意“\_”不能出现在进制标识和数值之间，不能出现在数值开头和结尾，不能出现在小数点旁边。

### 常量

（1）常量值（字面量），比如1、2、4.5、“abc”

（2）有一种变量，一旦给了初始值，就不能改变，这种变量就是常量，用final修饰符修饰。

**编译器常量的定义及风险**

公共静态不可变（**public static final**）变量也就是我们所说的编译期常量，这里的 public 可选的。实际上这些变量在编译时会被替换掉，因为编译器知道这些变量的值，并且知道这些变量在运行时不能改变。这种方式存在的一个问题是你使用了一个内部的或第三方库中的公有编译时常量，但是这个值后面被其他人改变了，但是你的客户端仍然在使用老的值，甚至你已经部署了一个新jar。为了避免这种情况，当你在更新依赖jar文件时，确保重新编译你的程序。

## assert关键字

assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。在实现中，assertion 就是在程序中的一条语句，它对一个 boolean 表达式进行检查，一个正确程序必须保证这个 boolean 表达式的值为 true；如果该值为 false，说明程序已经处于不正确的状态下，assert 将给出警告或退出。

Java 的 assert 是关键字。assert 的两个基本用法如下：

assert logicExp; 直接进行断言；

assert logicExp : expr;断言失败时显示特定信息。

虽然assert是JDK1.4新增的关键字，需要说明的是，Java命令默认不启动断言，为了启动用户断言，应该在运行java命令时增加-ea（Enable Assert）选项。为了启动系统断言，应该在运行 java 命令时增加-esa（Enable System Assert）选项。

# DateType

Java数据类型分两类：基本数据类型（primitive type）和引用数据类型。

基本数据类型变量的长度是一个固定值，与平台无关，不论在32位还是64位的Java虚拟机中，基本数据类型的长度都是不变的。

## 基本数据类型

### 整数类型

（Java中整数默认为int类型）(bit:计算机中的最小存储单位,byte:计算机中基本存储单元)

byte 1字节 8bit -128~127(-2^7~2^7-1) 0

11111111~01111111 0

short 2字节 16bit (-2^15~2^15-1) 0

int 4字节 32bit (-2^31~2^31-1) 0

long 8字节 64bit (-2^63~2^63-1) 0

定义Long时后面需要加上‘l’或‘L’，否则编译时会将数据默认为int类型，从而导致内存溢出。

### 浮点类型

单精度float 4字节 32bit -3.403E38~3.403E38(2^32-1) 0.0F

双精度double(default) 8字节 64bit -1.798E308~1.798E308(2^64-1) 0.0D

十进制数形式：如：5.12 512.0f .512 (必须有小数点）

科学计数法形式:如：5.12E2 5.12E-2

浮点类型数据存储形式：浮点型=符号位+指数位+尾数位；尾数部分可能丢失，造成精度损失；浮点型的存储分为三部分:符号位+指数位+尾数位，在存储过程中，精度会有丢失。

**两个小数比较是否相等**

如果是float直接赋值的，可以比较是否相等；

如果是double类型的，无论是否直接赋值，都不可以比较是否相等。

**注意：**

3\*0.1 == 0.3返回的结果为false，因为有些浮点数不能完全精确的表示出来。

double类型精度更高，一般都会使用double数据类型，float一般保存到小数点后6位，double一般保存到小数点后15位。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //6.3f/3结果是一个近似值，不等于2.1。  **float** a = 6.3f/3;  System.***out***.println(a);  //定义float时后面需要加上‘f’或‘F’，否则编译时会将数据默认为double类型，从而导致内存溢出。  **float** b = 2.1f;  **float** c = 2.1f;  //如果两个小数比较，如果是float直接赋值的，可以比较，如果是double类型的，无论是否直接赋值，都不可以比较是否相等  System.***out***.println(c==b);  //如果是运算结果，可能会出现丢失精度导致不等的问题，因此需要尽量避免  System.***out***.println(a==b);  }  } |
| 2.1000001  true  false |

Java整数类型和浮点类型有固定的表数范围和字段长度，不受OS（operating system）的影响，以保证Java程序的可移植性。

**关于运算中保留小数的实例**

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {    **double** a = 12.34564123;  **double** b = (**double**)Math.*round*(a\*100)/100;  System.***out***.println(b+"\n");    //BigDecimal.ROUND\_HALF\_UP表示四舍五入，BigDecimal.ROUND\_HALF\_DOWN也是五舍六入  //BigDecimal.ROUND\_UP表示进位处理（就是直接加1），BigDecimal.ROUND\_DOWN表示直接去掉尾数  BigDecimal c = **new** BigDecimal(a);  **double** d1 = c.setScale(2, BigDecimal.***ROUND\_HALF\_UP***).doubleValue();  System.***out***.println(d1);  **double** d2 = c.setScale(2, BigDecimal.***ROUND\_HALF\_DOWN***).doubleValue();  System.***out***.println(d2);  **double** d3 = c.setScale(2, BigDecimal.***ROUND\_UP***).doubleValue();  System.***out***.println(d3);  **double** d4 = c.setScale(2, BigDecimal.***ROUND\_DOWN***).doubleValue();  System.***out***.println(d4+"\n");    //#.00表示保留后两位，它的处理方式是直接截掉不要的尾数，不四舍五入  DecimalFormat df = **new** DecimalFormat("#.00");  String str = df.format(a);  System.***out***.println(str+"\n");    //%.2f表示保留后两位，能四舍五入。  System.***out***.println(Double.*parseDouble*(String.*format*("%.2f", a))+"\n");    //RoundingMode.HALF\_DOWN表示 五舍六入，负数先取绝对值再五舍六入再负数，RoundingMode.HALF\_UP:表示四舍五入，负数先取绝对值再五舍六入再负数。  NumberFormat nf = NumberFormat.*getNumberInstance*();  // 保留两位小数  nf.setMaximumFractionDigits(2);  // 如果不需要四舍五入，可以使用RoundingMode.DOWN  nf.setRoundingMode(RoundingMode.***UP***);  System.***out***.println(nf.format(a));  }  } |
| 12.35  12.35 12.35 12.35 12.34  12.35  12.35  12.35 |

### **字符型**

char 2字节 16bit ‘\u0000’

存放单个字符，汉字（如果是一个特殊的没有被包含在Unicode编码字符集中，那么就无法使用char表示）和转义字符

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //在java中，char的本质是一个整数，在输出时，是unicode码对应的字符  **char** c = 65;  System.***out***.println(c);  //char类型可以保存 int的常量值，但不能保存int的变量值，需要强转，本质原因是无法把4个字节的数据赋给2个字节的空间。  **int** e = 65;  **char** d = (**char**)e;  System.***out***.println(d);  //char类型是可以进行运算的，相当于一个整数，因为它都对应有Unicode码  **int** a = c+1;  System.***out***.println(a);  //char型数字0和int型数字0所表示的整数并不相等，因为char型的字符0会根据编码表的不同而不同  //反之int型数字转换为char型不是其数字的字符型，即数字1转换为char不是char型字符1  **char** b = '0';  **int** f = 0;  System.***out***.println(b == f);  }  } |
| A  A  66  false |

char字符型存储到计算机中，需要将字符对应的码值（整数）找出来，因此char型本质上是一个整数。  
 存储：字符'a' => 码值(97) => 转成进制(1100001) => 存储  
 读取：二进制(1100001) => 码值(97)=> 找到对应字符('a') => 显示

### 布尔型

boolean 1个字节 只有true和false false

不要用0或者非0整数来表示false或true

## 引用数据类型

### 类

详细请见面向对象。

### 接口

详细请见面向对象。

### 数组

**数组的特点**：相同数据类型的集合；元素类型可以是基本数据类型和引用数据类型；数组属于引用数据类型，引用存储在栈中，值（对象）存储在堆中；数组一旦创建成功，数组长度就不可变；数组声明时的类型就决定了数组存储元素的类型；数组的四要素：数据类型、标识符、元素和下标。

**数组的优点**：提高代码的简洁性和扩展性，且同时开辟了多个空间，提高了效率；分类存储，且空间是连续的，易于查找。

**数组的缺点：**长度不可变，不能进行扩容；数组中的属性和方法较少，缺少如增删改查等方法；数组中只能用来存储的元素是有序的，且可重复，那么对无序的不可重复的无法达到要求。

**数组的常见异常：**下角标异常和空指针异常

**一维数组**

**一维数组的声明**：

数据类型[] 数组名;（Java独有的声明方式，主要使用这种数组声明方式）

数据类型 数组名[];（c语言的声明方式，Java中也可以使用）

**一维数组的创建**：

因为数组是一个对象，因此可以使用new来创建一个数组。

**静态初始化**:（创建和赋值是同时进行的,即创建时就知道值是什么）

数据类型 数组名[] = new 数据类型[]{数据1,数据2,数据3...};(声明和创建可以分开)

数据类型[] 数据名 = {数据1,数据2,数据3...};(声明和创建不可以分开);

**动态初始化**:（创建和赋值是分开进行的，即创建时不确定值是什么）

数据类型[] 数组名 = new 数据类型[数值];(数值表示数组的长度，表示存放几个元素,不可省略)

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //动态初始化，确定元素的个数，但是不确定元素的值，因此创建元素的个数不能省略  String[] str3 = **new** String[3];  //数组创建后进行循环赋值，str3.length可以获取数组的长度  **for**(**int** i = 0;i < str3.length;i++) {  str3[i] = i+"";  System.***out***.println(str3[i]);  }    //静态初始化，确定元素的个数和值，因此创建赋值时不能写元素的个数  String[] str4 = **new** String[] {"a","b"};  str4 = **new** String[] {"c","d","e"};  //这种数组的声明和创建赋值不能分开  String[] str5 = {"1","2"};  System.***out***.println(str4[0]+str4[1]+str4[2]);  System.***out***.println(str5[0]+str5[1]);  }  } |

**数组元素的默认值**：

数组创建时，每个元素都会按照其数据类型而有一个默认值。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //byte、short、int、long默认值为0  **int**[] num = **new** **int**[1];  //float、double默认值为0.0  **double**[] doub = **new** **double**[1];  //char默认值为\u0000  **char**[] ch = **new** **char**[1];  //布尔型默认值为false  **boolean**[] boo = **new** **boolean**[1];  //引用数据类型的默认值为null  String[] str = **new** String[1];  }  } |

**给数组赋值并且获取数组中的元素**：

赋值：

数组名[0] = "数据"；

数组中的下角标（索引值）是从0开始的到（数组的长度-1）

特点：

基本类型变量的赋值：赋的是值，其中一个更改不影响另外一个

引用类型变量的赋值：赋的是地址，二者共同引用一个空间，其中一个更改影响另外一个

获取：

变量 = 数组名[0];（变量类型必须和数组元素类型相同，否则就要使用强制转换）

**数组的内存存放**

|  |
| --- |
| public static void main(String[] args) {  //每一次new新的数组就会在堆中开辟新的内存，然后将其地址值传递给str  String[] str = new String[]{"hello", "world"};  //此处是将str在栈中的地址值传递给了str2  String[] str2 = str;  //str和str2所表示的地址值指向堆中的相同位置，因此两者相同  System.*out*.println(str == str2);//true  //str和str3堆内存中地址不同，内容也不同，一个是堆中String类对象，一个是常量池中字符串对象  // 但是其中的对应位置元素相同  String[] str3 = {"hello", "world"};  System.*out*.println(str == str3);//false  System.*out*.println(str.equals(str3));//false  System.*out*.println(str[0].equals(str3[0]));//true  //str和str2两者相同，因此两者的改变是同步的  str[0] = "hi";  System.*out*.println(str[0] + "===" + str2[0]);//hi===hi  str2[1] = "earth";  System.*out*.println(str[1] + "===" + str2[1]);//earth===earth } |

**二维数组**

数组中的元素是一维数组。String表示的是二维数组元素的类型，即一维数组的元素类型。

二维数组的声明

String[][] str;

String[] str[];(不建议使用

String str[][];(不建议使用

二维数组的创建

静态初始化：

String[][] str = new String[][]{{元素1,元素2},{元素3,元素4}};

String[][] str = {{元素1,元素2},{元素3,元素4}};

动态初始化：

String[][] str = new String[3][4];

String[][] str = new String[3][0];

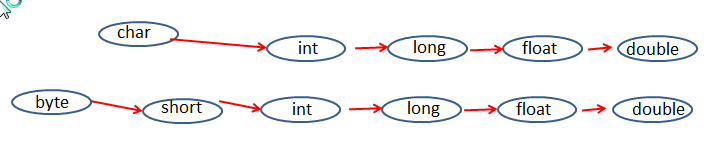
下面这种方法是不被允许的，会编译错误

String[][] str = new String[3][4];

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String []args) {  //静态初始化方式1  String[][] str1 = {{"hello","hi"},{"my","your"}};//这种静态初始化方式数组的声明和赋值不能分开。  //静态初始化方式2，如果str1之前没有声明就需要先声明  str1 = **new** String[][]{{"hello","hi"},{"my","your"}};//堆中开辟新的二维数组并将地址值传递给str1  //遍历二维数组，str1.length表示二维数组的长度，str1[i].length表示二维数组元素（一维数组）的长度  **for**(**int** i =0;i<str1.length;i++) {  **for**(**int** j=0;j<str1[i].length;j++) {  System.***out***.print(str1[i][j]+"\t");  }  System.***out***.println();  }  //动态初始化（两种方式，使用哪种根据实际情况选择）：  //1)这种方式二维数组元素的长度都是一样的，3表示二维数组的长度，4表示二维数组元素（一维数组）的长度。  String[] str2[] = **new** String[3][4];  //2）这种方式二维数组元素的长度可以是不一样的。  String str3[][] = **new** String[3][];  //下列表示二元数组的元素长度分别为2、3和4  str3[0] = **new** String[2];  str3[1] = **new** String[3];  str3[2] = **new** String[4];  //赋值：  str2[0][0] = "one";  //获值：  String str5 = str2[0][0];  System.***out***.println(str5);  System.***out***.println(str2[0][1]);  //二维数组元素的默认值是null。  }  } |
| hello hi  my your  one  null |

## 基本数据类型转换

### 自动类型转换



1）char类型和byte、short之间不能进行自动数据类型转换（可以强制类型转换），但是可以计算，在计算时首先转换为int类型。

2）boolean类型不参与数据类型转换。

3）整体来看，表示范围较小的数据可以自动转换为表示范围较大的数据类型

有多种类型的数据混合运算时，系统首先自动将所有数据转换成精度最大的那种数据类型，然后再进行计算。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //byte和short、char类型只要在进行运算时，当做int类型处理。  **byte** c = 10;  **short** d = 20;  **char** a = 'A';  **int** e = c+d+a;  System.***out***.println(e);  //java中任意一个数值，如果没有类型声明，则默认为int或double型，此时20则默认为int型  **int** f = c + 20;  System.***out***.println(f);  }  } |
| 95  30 |

### 强制类型转换

自动类型转换的逆过程，将精度大的转换为精度小的，使用时要加上强制转换符()，但是可能造成精度降低或溢出（溢出是指保留低位，舍弃高位），必须谨慎使用。

比如将int类型的数据强制转换为byte类型，Java中int是32位的而byte是8 位的，所以强制转化int类型的数据会使高24位被丢弃。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //强转符号只针对于最近的操作数有效，往往会使用小括号提升优先级  **int** a = (**int**)(10\*3.5 + 6\*1.4);  //此时将double类型转换为int型，造成了精度降低，有时也会造成溢出  System.***out***.println(a);  //int a = (int)10\*3.5 + 6\*1.5;此时会报错  //因为只将10强制转换了，最后的运算结果是double型  **double** b = (**int**)10\*3.5 + 6\*1.4;  System.***out***.println(b);  }  } |
| 43  43.4 |

### 基本数据类型和字符串的转换

基本数据类型转换为字符串：字符串 = 基本数据类型 + “” ；

字符串转换为基本数据类型：通过基本类型的包装类调用parseXXX方法。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //将String类型转成基本数据类型时，要确保能够转换。  System.***out***.println(Integer.*parseInt*("25"));  System.***out***.println(Float.*parseFloat*("3.14"));  System.***out***.println(Byte.*parseByte*("25"));  System.***out***.println(Short.*parseShort*("25"));  System.***out***.println(Long.*parseLong*("33"));  System.***out***.println(Boolean.*parseBoolean*("fals"));  }  } |
| 25  3.14  25  25  33  false |

# Operator

## 算术运算符

（+、-、\*、/、%、++、--）

算术运算符是对数值类型的变量进行运算的，在Java程序中使用的非常多。

++、--运算符**不是线程安全**的，涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //整数相除只会得到整数，会将小数部分舍去  **int** a = 9/4;  System.***out***.println(a);  //要想结果是浮点型，则在任意整数上添加小数使之变为浮点型，则结果是浮点型  **double** b = 9.0/4;  System.***out***.println(b);  //String的format可以设置保存几位小数  System.***out***.println(String.*format*("%.4f", b));  }  } |
| 2  2.25  2.2500 |

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //取模（取余）%表示求出余数，取模的公式a%b = a-a/b\*b，取余的结果正负符号和除数保持一致。  System.***out***.println(10%3);  System.***out***.println(-10%3);  System.***out***.println(10%-3);  System.***out***.println(-10%-3);  //小数参与取余运算时，规则是先将小数强制转换成int类型，再进行运算。a%b即为a-(int)a/(int)b\*b。  System.***out***.println(-10.5%3);  //因为结果是个近似值，所以不一定等于准确值。  System.***out***.println(10%3.3);  }  } |
| 1  -1  1  -1  -1.5  0.10000000000000053 |

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int** m = 5,n = 5;  //++运算符独立使用，结果是一样的，++a和a++一样，都是增加1；  m++;  ++n;  System.***out***.println(m);  //打印语句中的表达式会产生结果并影响到下面的语句。  System.***out***.print(n++);  System.***out***.print(n++);  System.***out***.println(n++);  //如果是和赋值或者运算相关，则不一样，++a是先加1再用a，a++是先用a再加1。  **int** a = m++;  System.***out***.println(a);  **int** b = ++m;  System.***out***.println(b);  }  } |
| 6  678  6  8 |

## 赋值运算符

（=、+=、-=、\*=、/=、%=等）

a += b等价于a = a + b,但是前者效率更高。

赋值运算符顺序从右往左，左边只能是变量，右边可以是变量、表达式、常量值；byte b = 1；b += 1；赋值会**自动进行类型转换**，在运行结果上强制转换，使b最后仍为byte类型，而b = b + 1会报错。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **byte** a = 10;  //a += 10;虽然等价于a = a + 10，但是其底层实现了数据类型的自动转换a = (byte)(a + 10);最后结果仍为byte型  a += 10;  System.***out***.println(a);  //a = a + 10;就会报错，必须使用强制转换  a = (**byte**)(a + 10);  System.***out***.println(a);    }  } |
| 20  30 |

## 关系运算符

（比较运算符 >、 >= 、< 、<= 、==、 !=、instanceof）

instanceof：检查是否是类的对象

小结一下

关系运算符的结果都是boolean型，也就是要么是true，要么是false。

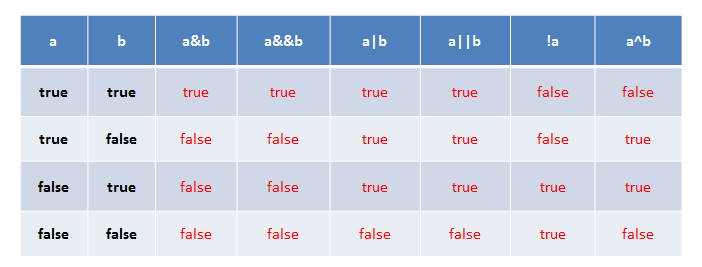
关系运算符组成的表达式，我们称为关系表达式： a > b

比较运算符“==”[表示比较]不能误写成“=”[表示赋值]

## 逻辑运算符

（两大类 短路与&&，短路非||，非！，逻辑与&，逻辑或|，逻辑异或^）

用于连接多个条件（一般来讲就是关系表达式），最终的结果也是一个boolean值。



短路与和逻辑与的区别：短路与按顺序执行，只要执行到一个false，就不用继续执行后面的表达式判断是否为false，逻辑与则需要全部判断。因此短路与的效率更高，实际开发中基本上使用短路与。

短路或和逻辑或的区别与短语与和逻辑与的区别相同，短路或只要执行到一个true，就不用执行后面的表达式，开发中基本上使用短路或。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int** a = 10;  **int** b = 20;  **if** (a++ == 10 && ++ b == 21) {  System.***out***.println(1+"a = "+a+"b = "+b);  }  //同打印语句一样，if语句中的表达式结果会影响到后面的语句  System.***out***.println(2+"a = "+a+"b = "+b);  //逻辑与：++a==11为假，所以后面的表达式不执行，因此b为21  **if** (++a == 11 && b++ == 21) {  System.***out***.println(3+"a = "+a+"b = "+b);  }  System.***out***.println(4+"a = "+a+"b = "+b);  **if** (a++ == 12 & ++ b == 22) {  System.***out***.println(5+"a = "+a+"b = "+b);  }  System.***out***.println(6+"a = "+a+"b = "+b);  //++a==11为假，所以后面的表达式依然要执行判断，因此b为23  **if** (++a == 11 & b++ == 22) {  System.***out***.println(7+"a = "+a+"b = "+b);  }  System.***out***.println(8+"a = "+a+"b = "+b);  }  } |
| 1a = 11b = 21  2a = 11b = 21  4a = 12b = 21  5a = 13b = 22  6a = 13b = 22  8a = 14b = 23 |

## 位运算符

（&，|，^，~，<<，>>，>>>）

位运算是对二进制的运算，按byte型为例，其占8位，最高位0表示正数，1表示负数。

整数向右移一次相当于除以2，向左移相当于乘以2，位运算符的运算效率要高于算数运算符，因此要高效运算2的倍数的乘除运算的话就可以使用位运算符，比如说要求2\*8的值，可以写成2<<3。

判断一个数的奇偶性最快的方式也是使用位运算符，将该数与1进行按位与，再与0（判断奇数偶数不要将结果与1进行比较，会出现负数无法判断出奇偶性的问题）进行判断，若相等则表示该数为偶数，若不为0则表示该数为奇数。

**原码、反码、补码**

（对于有符号的而言）

1) 二进制的最高位是符号位: 0表示正数,1表示负数

2) 正数的原码，反码，补码都一样

3) 负数的反码=它的原码符号位不变，其它位取反

4) 负数的补码=它的反码+1

5) 0的反码，补码都是0

6) java没有无符号数，换言之，java中的数都是有符号的

7) 在计算机运算的时候，都是以补码的方式来运算的

计算机中的运算本质是先转换成二进制，再换成计算机计算的补码，再进行运算得到新的补码，再根据新的补码换成原码，最后根据原码得到最终的结果。

**EXAMPLE**

byte型-3的原码、反码、补码分别为1000 0011、1111 1100、1111 1101

**位运算**

按位与&　 ：　 两位全为１，结果为1，否则为0

按位或| : 两位有一个为1，结果为1，否则为0

按位异或^ : 两位一个为0,一个为1，结果为1，否则为0

按位取反~ : 0->1 ,1->0；取反得到的并不是反码，因为符号位也变了。

**移位运算**

>>、<< 算术右移和算术左移,运算规则:

算术右移 >>：低位溢出,符号位不变,并用符号位补溢出的高位

算术左移 <<: 符号位不变,低位补0

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String []args)  {  **int** a=1>>2;  // 0000 0001 0000 0000  **int** b=-1>>2;  // 1000 0001 1000 0001  // 1111 1110 1111 1110  // 1111 1111 1111 1111  **int** c=1<<2;  // 0000 0001 0000 0100 4  **int** d=-1<<2;  // 1000 0001 1000 0100  // 1111 1110 1111 1011  // 1111 1111 1111 1100  **int** e=3>>>2;  // 0000 0011 0000 0000  System.***out***.println("a="+a);  System.***out***.println("b="+b);  System.***out***.println("c="+c);  System.***out***.println("d="+d);  System.***out***.println("e="+e);  }  } |
| a=0  b=-1  c=4  d=-4  e=0 |

>>> 逻辑右移也叫无符号右移,运算规则是：低位溢出，高位补 0

例：-2>>>1

原码 1000 0010 0111 1111

反码 1111 1101 0111 1111

补码 1111 1110 0111 1111

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String []args) {  **byte** b = -2;  //byte型的b在运算时会自动转换为int型，因此需要将其和255（0xff）进行按位与运算  //使其保留最后8位需要的数，再进行无符号右移操作，最后将结果强制转换为byte型  **byte** a =(**byte**)((b & 255)>>> 1);  System.***out***.println(a);  **short** f = -2;  //short型无符号右移的原理和byte型相同，需要将其和65535（0xffff)进行按位与运算  //使其保留最后16位二进制数，再进行无符号右移，最后将结果强制转换为short型  f = (**short** )((f&65535) >>> 1);  System.***out***.println(f);  **int** c = 256;  //c转换成d是会发生溢出，只保留最后8位  **byte** d = (**byte**)c;  **short** e = (**short**)c;  System.***out***.println(d);  System.***out***.println(e);  }  } |
| 127  32767  0  256 |

特别说明：没有 <<< 运算符

## 三元运算符

条件表达式 ? 表达式1: 表达式2;条件表达式结果为true，执行表达式1并返回结果值；否则执行表达式2并返回结果值。

表达式1和表达式2若其中有一个是变量，按照自动类型转换规则处理成一致的数据类型；

若都是常量，如果一个是char，另一个是[0~65535]之间的整数按char处理；如果一个是char，另一个是其他类型数据，按照自动类型转换规则处理成一致的类型。

表达式1或2返回的类型要为可以接受的数据类型，或者可以自动转换或强制转换的数据类型。

|  |
| --- |
| @SuppressWarnings("unused")  **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int** a = 10;  **int** b = 20;  //条件表达式中对a的运算结果改变内存中a的值  **int** c = ++a<=b?++a:++b;  System.***out***.println(a );  //条件表达式的结果为false，则执行表达式2  System.***out***.println(c);  **double** d = 30.0;  //三元运算符后面的两个表达式类型要相同，或者可以自动转换的，先将其自动转换为相同类型  **double** e = **true**?a:d;  System.***out***.println(e);  A a2 = **new** A();  B b2 = **new** B();  A a3 = **new** A();  //三元运算符后面的两个表达式类型也可以是继承关系，然后先将两者转换为父类类型  a3 = **true**?b2:a2;  System.***out***.println(a3 **instanceof** B);  }  }  **class** A{}  **class** B **extends** A{ } |
| 12  12  12.0  true |

## 运算符的优先级别

运算符有不同的优先级，所谓优先级就是表达式运算中的运算顺序。如右表，上一行运算符总优先于下一行。

只有单目运算符、赋值运算符是从右向左运算的。

|  |  |  |
| --- | --- | --- |
| 优先级 | 描述 | 运算符 |
| 1 | 括号 | ()、[] |
| 2 | 正负号 | +、-（右->左） |
| 3 | 自增自减，非 | ++、--、! （右->左） |
| 4 | 乘除，取余 | \*、/、% |
| 5 | 加减 | +、- |
| 6 | 移位运算 | <<、>>、>>> |
| 7 | 大小关系 | >、>=、<、<= |
| 8 | 相等关系 | ==、!= |
| 9 | 按位与 | & |
| 10 | 按位异或 | ^ |
| 11 | 按位或 | | |
| 12 | 逻辑与 | && |
| 13 | 逻辑或 | || |
| 14 | 条件运算 | ?: （右->左） |
| 15 | 赋值运算 | =、+=、-=、\*=、/=、%=（右->左） |
| 16 | 位赋值运算 | &=、|=、<<=、>>=、>>>=（右->左） |

# Process\_Control

## 顺序控制

程序从上到下逐行地执行，中间没有任何判断和跳转。

Java中定义成员变量时采用合法的前向引用，即先定义，在使用。

## 分支控制语句

### if else语句

单分支 if(){}

双分支 if(){} else{}

多分支 if(){} else if(){} else if(){}... else{}

双分支和多分支的else不是必须的，可以不用。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String []args)  {  **boolean** b = **true**;  **if**(b == **false**)  System.***out***.println("a");  //如果写成if(b=false)能编译通过吗？如果能，结果是？  **else** **if**(b = **false**)  System.***out***.println("b");  //if(b=false)相当于把false赋值给b，然后判断，结果为false，因此!b为true，打印出c  **else** **if**(!b)  System.***out***.println("c");//  **else**  System.***out***.println("d");  }  } |
| c |

Java语法的条件编译，是通过判断条件为常量的if语句实现的。根据if判断条件的真假，编译器直接把分支为false的代码块消除。通过该方式实现的条件编译，必须在方法体内实现，而无法在正整个Java类的结构或者类的属性上进行条件编译。

### switch语句

选择结构switch

switch(表达式){

case 值1：

语句体1;

break;

case 值2：

语句体2;

break;

…

default：

语句体n+1;

break;

}

表达式的取值：**byte、short、int、char**，JDK5以后可以是**枚举**，JDK7以后可以是**String**。

对于byte、short、int类型，直接进行数值的比较；

对于char类型则是比较其ASCII码；

对于String类型则是调用了String的hashCode()和equals()方法，比较的是hashCode()方法返回的int型数值（不是long类型的数值）。

注：利用equals方法比较（安全检查）是必要的，因为不同的对象哈希值可能相同。因此使用String进行switch分支的性能不如枚举和纯整数常量。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **final** **int** a = 1;  **int** b = 1;  **switch** (b) {  //case后面只能是常量，而不能是变量。如果该常量实用final定义的，也可以使用  **case** a:  System.***out***.println("OK");  **break**;  //表达式数据类型应和case后的常量类型保持一致，或者可以自动转成可以相互比较的类型。  **case** (**byte**)2:  System.***out***.println("2");  **break**;  //default子句可选，当没有匹配的case时，执行default下的语句。  **default**:  System.***out***.println("没有匹配的");  **break**;  }  }  } |
| OK |

break用来执行完一个case分支后使程序跳出switch语句块，如果没有break，程序会顺序执行到switch结尾，直到执行到break，这就是switch的switch穿透特性。

|  |
| --- |
| **import** java.util.Scanner;  **public** **class** Example {  **public** **static** **void** main(String[] args) {  Scanner scanner = **new** Scanner(System.***in***);  String string = scanner.next();  **switch** (string) {  //case后面的值不能重复，否则会发生编译错误  **case** "1月":  **case** "2月":  **case** "3月":  System.***out***.println("春天");  //switch穿透特性：如果没有break，程序会顺序执行到switch结尾，直到执行到break  **break**;  **case** "4月":  System.***out***.print("4月");  **case** "5月":  System.***out***.print("5月");  **case** "6月":  System.***out***.print("6月");  System.***out***.println("是夏天");  **break**;  **case** "7月":  **case** "8月":  **case** "9月":  System.***out***.println("秋天");  **break**;  **case** "10月":  **case** "11月":  **case** "12月":  System.***out***.println("冬天");  **break**;  }  }  } |
| 控制台上输入4月，输出结果：  4月5月6月是夏天 |

## 循环控制语句

### for语句

**for循环格式**

for(初始化;判断条件;变量迭代){

循环体;

}

判断条件是返回一个布尔值的表达式，返回true则执行循环体，返回false则结束循环。

**for循环的执行顺序**

初始化（执行且只执行一次）

-> 判断条件

->返回true，执行循环体

->变量迭代表达式，然后返回执行判断条件，达成循环

->返回false，结束循环

for(;循环判断条件;) 中的初始化和变量迭代可以不写（写到其它地方），但是两边的分号不能省略。如果中间的循环判断条件也没有的话会变成死循环。

循环初始值可以有**多条初始化语句**，但要求类型一样，并且中间用逗号隔开，循环变量迭代也可以有**多条变量迭代语句**，中间用逗号隔开。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int** i = 1;  **int** j = 2;  //初始化和变量迭代可以不写（写到其它地方）  **for**(;i<3;j++) {  i++;  System.***out***.println(i);  }  //for循环中j的操作作用到了内存中  System.***out***.println("j="+j);  }  } |
| 2  3  j=4 |

JDK1.5以后有了增强for循环（foreach循环）

for(元素类型 临时变量：集合/数组的对象）{

遍历数组和集合中的元素（循环体）

}

只能用来遍历集合和数组，且集合数组中的元素无法被修改。如果要修改其中的元素的内容，则必须要使用普通的for循环。

for-each实现原理就是使用了普通的for循环和迭代器，因此使用增强for循环来遍历集合时无法对集合进行移除，添加等操作，会抛出ConcurrentModificationException异常。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int**[] num = { 1, 2, 3, 4, 5 };  **for** (**int** i : num) {  System.***out***.print(i);  }  String[] str = { "a", "b", "c" };  **for** (String s : str) {  System.***out***.print(s);  }  }  } |
| 12345abc |

### while语句

while(判断条件语句) {

循环体;

}

### do while语句

do {

循环体;

}while((判断条件语句)**;（注意括号后面有一个分号）**

如何选择那个循环完成编码：

1、有明确退出条件的，循环次数确定的一般选用for循环

2、当不确定何时退出时，循环次数不确定的往往是while或者do while配合break使用

3、如果需要至少执行一次，则选用do while。for和while循环只会在条件成立时循环。

4、如果想继续使用控制条件语句的变量，就用while或do while，否则使用for。原因是for循环结束，该变量就从内存中消失，能够提高内存的使用效率。

### 多重循环控制

将一个循环放在另一个循环体内，就形成了嵌套循环。其中，for ,while ,do…while均可以作为外层循环和内层循环。【建议一般使用两层，最多不要超过3层】

实质上，嵌套循环就是把内层循环当成外层循环的循环体。当只有内层循环的循环条件为false时，才会完全跳出内层循环，才可结束外层的当次循环，开始下一次的循环。

设外层循环次数为m次，内层为n次，则内层循环体实际上需要执行m\*n=mn次。

## 跳转控制语句

### Label

标签的基本使用

label1: {……

label2: {……

label3: { ……

break label2;

……

}

}

}

实际开发中尽量不要使用label，但是如果想要结束外层循环，可以通过使外层循环的条件表达式的结果受内存循环代码的控制，示例如下：

|  |
| --- |
| **boolean** flag = **false**;  **for** (**int** i = 0; i < 10 && !flag; i++) {  **for**(**int** j = 0; j < 5;j++) {  System.***out***.print(" "+ i + j);  **if**(j == 3) {  flag = **true**;  **break**;  }  }  System.***out***.println();  } |

### break语句

break的使用场景：在选择结构switch语句和循环语句中

break表示中断，结束的意思，可以结束switch语句；如果没有遇到break，就会将匹配的值及其后面可以执行的语句全部执行。

break（就近原则）在哪个循环的循环体内，就默认作用于哪个循环。

### continue语句

continue的使用场景：在循环语句中

然后总结两个的区别：break是退出当前循环，且可以在switch和循环中使用；continue是退出本次循环，并继续下一次循环，且只能在循环中使用。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  labelJ:  **for**(**int** i=1;i<=5;i++) {  **if**(i==5) {  //当i=5时，会结束i的循环，导致循环全部结束，最后打印出循环结束  **break**;  }  **for**(**int** j=1;j<=5;j++) {  **if** (j==3) {  //当j=3时，会跳出当次j的循环，继续下一次循环，即j=4的循环  **continue**;  }  **if** (j==5) {  //当j=5时，会跳出当次j的循环，跳到i的下一次循环  **continue** labelJ;  }  **for**(**int** k=1;k<=4;k++) {  **if** (k==3) {  **break**;  //如果此处跳到labelJ，则在k=3时结束i的循环，最后的结果只会打印一句  //i=1;j=1;k=1 i=1;j=1;k=2 循环结束  // break labelJ;  }  System.***out***.print("i="+i+";j="+j+";k="+k+" ");  }  System.***out***.print("\n");  }  //因为在j=5时，直接跳出了这一次循环而进入下一次循环，因此这一语句永远不会执行到  System.***out***.print("hello");  }  System.***out***.println("循环结束");  }  } |
| i=1;j=1;k=1 i=1;j=1;k=2  i=1;j=2;k=1 i=1;j=2;k=2  i=1;j=4;k=1 i=1;j=4;k=2  i=2;j=1;k=1 i=2;j=1;k=2  i=2;j=2;k=1 i=2;j=2;k=2  i=2;j=4;k=1 i=2;j=4;k=2  i=3;j=1;k=1 i=3;j=1;k=2  i=3;j=2;k=1 i=3;j=2;k=2  i=3;j=4;k=1 i=3;j=4;k=2  i=4;j=1;k=1 i=4;j=1;k=2  i=4;j=2;k=1 i=4;j=2;k=2  i=4;j=4;k=1 i=4;j=4;k=2  循环结束 |

### return语句

return语句的原则是哪里调用就返回哪里

在有返回值的方法中，使用return返回数据给方法的调用者同时结束当前方法。

在无返回值的方法中，也可以使用return，结束当前方法。

return后面不能直接再写其他的执行语句，否则会报错。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int** a = 0;  a = *exa*(a);  **if**(a == 1) {  //此处调用的return是main函数，因此跳出了整个函数，后面的语句不再执行  **return**;  }  //因为之前的return，导致此处没有打印  System.***out***.println("Hello1");  }  **static** **int** exa(**int** a) {  a = 1;  System.***out***.println("Hello2");  //此处调用的return是exa方法，因此跳出了这个方法，后面的语句不再执行  **return** a;  }  } |
| Hello2 |

## 递归

递归的本质就是方法自身调用自身，**在栈内存中不断的加载同一个方法**。在使用一个功能过程中，又对该功能有需求。

**递归使用**

当一个功能被重复使用，而每一次使用该功能时的参数不确定，都由上次的功能元素结果来确定。

简单说：**功能内部又用到该功能，但是传递的参数值不确定。**(每次功能参与运算的未知内容不确定)。

**递归的注意事项：**

1：一定要定义递归的条件，否则会内存溢出。

2：递归的次数不要过多。也会出现 **StackOverflowError 栈内存溢出错误**。

**遍历指定目录下的所有文件**

此时就需要用到递归进行遍历

private void listDir(File dir, int level){  
 level++;  
 File[] files = dir.listFiles();  
 for (int i = 0; i < files.length; i++) {  
 if(files[i].isDirectory()){  
 System.*out*.println(getLevel(level)+files[i].getName());  
 listDir(files[i],level);  
 }else {  
 System.*out*.println(getLevel(level)+files[i].getName());  
 }  
 }  
}  
private String getLevel(int level){  
 StringBuilder sb = new StringBuilder();  
 sb.append("|--");  
 for (int i = 0; i < level-1; i++) {  
 sb.insert(0,"| ");  
 }  
 return sb.toString();  
}

# OOP

面向对象编程（**Object Oriented Programming**）的编程思想，不同于面向过程编程（**Procedure Oriented Programming**），是将功能封装进对象（**object**），强调具备了功能的对象。面向对象更加强调运用人类在日常的思维逻辑中采用的思想方法与原则，如抽象、分类、继承、聚合、多态等。

创建对象的说法：创建类的实例（**instance**）；类的实例化；实例化类。

Java面向对象三大特性：

封装（**Encapsulation**）

继承（**Inheritance**）

多态（**Polymorphism**）

## 类成员一、变量

**变量（variable）的区分：**

按类型分：基本数据类型和引用数据类型

按位置分：成员变量（属性**field**）和局部变量

成员变量（属性）按加载时间分：

类变量：static修饰的变量

实例变量：不是static修饰的变量

局部变量包括形参（方法、构造器中声明的变量）、方法局部变量（在方法内声明）、代码块局部变量（在代码块内声明）

**属性和局部变量的相同点**：先声明后使用，都有作用域。

**属性和局部变量的不同点**：

**代码中位置**：

成员变量：在类中，方法、构造器和初始化块等结构外声明的变量

局部变量：在方法和方法的形参、构造器和构造器的形参、初始化块中声明的变量

**权限修饰符**：

成员变量：可以用四种权限修饰符（public protected default private）

局部变量：不可以使用权限修饰符

**默认值**：

成员变量：有默认值

局部变量：无默认值

|  |  |
| --- | --- |
| **数据类型** | **默认值** |
| boolean | false |
| char | \u0000 |
| byte | (byte)0 |
| short | (short)0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| reference | null |

**内存中位置**：

成员变量：堆（heap），随着对象的存在而存在

局部变量：栈（stack），方法调用完毕就消失

**作用域**：

成员变量：作用域为整个类体

局部变量：作用域为定义它的代码块中

属性和局部变量可以同名，访问时遵循就近原则，可以使用this将其区分。

在类中，属性的声明赋值可以写在调用它的方法前，也可以写在调用它的方法后。因为**类在加载时，首先加载成员变量和方法，而方法是不调用不执行的**，因此不会因为成员变量的代码写在方法的后面就显示cannot be resolved to a variable错误。而在方法中的局部变量就不可以这样，必须遵循顺序控制的原理，先声明赋值，再使用，否则就会报错。

|  |
| --- |
| **class** Handsome{  //内中只能声明属性、方法，不能有纯运算表达式  String name= "许芒芒";  **int** age = 25;  //声明赋值时可以用表达式的形式将其结果赋值给声明的对象  //num = age+2;这种没有声明的赋值就会编译出错  **int** num = age+1;  //初始化代码块中可以有多条执行语句  {  name = "bob";  }  Handsome(){  age += 100;  }  } |

**给属性赋值的方式及其执行顺序**

默认值>显式赋值/代码块赋值（两者按顺序执行）>构造器赋值>方法赋值

**静态属性**

初始化两次，一次在“准备”阶段，先进行一次初始化，系统附上默认值；第二次在“初始化”阶段，根据代码中的赋值情况再进行一次初始化。

**非静态属性**

仅“初始化”阶段赋值。根据代码中的赋值情况，代码不赋值直接赋默认值，有赋值则等于代码中的赋值。对象实例化后，该变量随java对象分配到java堆中。

**局部变量**

局部变量没有初始化就不能使用，原因就是类方法中的代码，是在字节码执行的时候，才会被运行到，此时局部变量是存储在虚拟机栈-栈帧中的局部变量表中。

|  |
| --- |
| **public** **class** SetValue {  **public** **static** **void** main(String[] args) {  Dog dog = **new** Dog();  dog.show();  }  }  **class** Dog{  //10和20赋值的顺序相同，从上往下执行，10会覆盖掉20  {  age = 20;  }  **int** age = 10;  //构造器中的赋值方式顺序靠后，因此会将之前赋值的10覆盖掉，赋值为30  Dog(){  age = 30;  }  **public** **void** show(){  //方法中赋值最靠后，因此会将30覆盖掉，赋值我40，最后的结果为40  age = 40;  System.***out***.println(age);  }  } |
| 40 |

## 类成员二、方法

### 方法简介

**方法的格式**：

权限修饰符 返回值类型 方法名（method）（参数类型 形参1，参数类型 形参2，…）｛

程序代码:方法功能的具体实现

(return 返回值;)

｝

**方法的四种表示方式**：无返回值无参；无返回值有参；有返回值无参；有返回值有参。

**方法的权限修饰符**：public protected default private

**返回值类型**：

有：基本数据类型和引用数据类型，使用return将返回值返回给方法的调用者，并结束方法。

无：void，此时方法中也可以有return，表示结束方法，此时没有返回值。

**方法命名规范**：只要遵守标识符的规则和规范即可，首字母小写，其他单词首字母大写。

**方法体**：方法的具体功能的实现，方法不调用方法体不会被执行。

实参：调用方法时实际传给函数形式参数的数据，可以是常量也可以是变量。

形参：方法声明的变量，不能是常量，接收方法调用时传递进来的数据。

**Java中方法的参数传递方式只有一种：值传递**，即将实际参数值的副本传入方法内，而参数本身不受影响。

同一个类中的方法之间可以互相调用，也可以调用自身，并通过分支语句避免死循环，这种方法的调用方式称为递归。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  Test t = **new** Test();  //通过test1可以调用类中的私有方法  //可以通过创建匿名对象直接调用方法，而匿名对象常常作为实参来传递  **new** Test().test1();  t.test2(9);  System.***out***.println(t.test4()+" "+t.num);  System.***out***.println(t.test3()+" "+t.num);  }  }  **class** Test{  //str和num都为成员变量，存放在堆中，不是必须初始化，可以初始化，可以用权限修饰符修饰  **private** String str;  **public** **int** num = 1;  //无返回值无参，方法可以被权限修饰符修饰  **public** **void** test1() {  //str1、str2、num2都为局部变量，存放在栈中，必须要先初始化，不可以用权限修饰符修饰  String str1 = "str1";  String str2 = "str2";  //str没有初始化，因此为null  System.***out***.print(str+"\t");  System.***out***.print(str1+"\t");  //方法可以调用同一个类中的任何方法  System.***out***.println(test4(str2));  }  //无返回值有参  **protected** **void** test2(**int** num2) {  **if**(num2>num) {  System.***out***.println(num2+">"+**this**.num);  //return可以用在没有返回值的方法中，表示结束方法  **return**;  //return后面不可以直接有执行语句，会编译错误  // System.out.println();  }  //该语句在return后面，并且可能会被执行，因此可以存在  System.***out***.println("compare two number");  }  //有返回值无参  **int** test4() {  **return** num\*8;//这种返回方式不同于下面那种，此时的num的值没有变化  }  **int** test3() {  **return** num \*= 8;//这种返回方式虽然结果和上面相同，但是会导致num的值发生变化，因为对num有赋值操作  }  //有返回值有参  **private** String test4(String str3) {  **return** str3;  }  } |
| null str1 str2  9>1  8 1  8 8 |

方法返回不同类型返回值：

利用集合

编写一个类并返回这个类的对象，对象中含需要的不同类型的变量

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  p.say(p);  System.***out***.println(p.name+p.age);  }  }  **class** Person{  String name;  **int** age;  //返回不能类型的数据，可以将其编写到同一个类中的成员变量  **public** Object say(Person p) {  p.name = "Bob";  p.age = 19;  **return** p;  }  } |
| Bob19 |

### 方法重载（overload）

方法的重载（**overload**）：在同一个类中，具有相同方法名、不同形参列表的方法之间构成方法的重载。形参的不同表现在形参的**个数**、**类型**和**顺序**上，和返回值有无及类型是否不同、是否有权限修饰符和形参的名字无关。

如果参数列表中的参数没有合适的方法，可以自动提升数据类型，寻找可以调用的方法。

对象调用哪个方法：先判断方法名，再判断形参列表。

|  |
| --- |
| **public** **class** Overload {  **public** **static** **void** main(String[] args) {  **new** Overload().add(1,1);  **new** Overload().add(0.1,1);  **new** Overload().add(2.0,2.0);  **new** Overload().add(1,2,3);    }  **void** add(**int** a,**int** b) {  System.***out***.println("1:"+a+b);  }  //此时是因为形参类型不同构成重载，而不是返回值的因素  Double add(**int** c,**double** d) {  **return** c+d;  }  //形参类型不同构成重载  **protected** **void** add(**double** a,**double** b) {  System.***out***.println("3:"+a+b);  }  //形参个数不同构成重载  **public** **void** add(**double** a,**double** b,**double** c) {  System.***out***.println("4:"+a+b+c);  }  //形参顺序不同构成重载  **private** **void** add(**double** c,**int** d) {  System.***out***.println("5:"+c+d);  }  } |
| 1:11  5:0.11  3:2.02.0  4:1.02.03.0 |

### 方法重写/覆盖（override）

当子类继承父类以后，继承的父类方法不能满足子类的需求，就可以使用方法的重写。重写父类方法后，再用子类的对象（或this）进行调用被重写方法，实际调用的是子类重写后的方法。

**细节**：

**重写方法和被重写方法方法名和参数列表必须完全一致。**

重写方法权限修饰符**不小于**被重写方法权限修饰符。

重写方法返回值类型**不大于**被重写方法返回值类型，且和被重写方法返回值之间有继承关系或相 同。

示例：父类返回值类型 子类返回值类型

void void

Number Number及其子类

double int(没有继承关系，返回值类型要么一样，要么有继承关系）

子类方法抛出的异常**不大于**父类被重写方法的异常

被private、static、final修饰的方法不能被重写

子类和父类中相同（同名同参）的方法，要么同时加static，要么同时不加static，加static不是方法的重写（一般称之为隐藏），不加static才是方法的重写。重写只适用于非静态方法.不能用于静态方法。静态方法的调用不需要实例化，不实例化即不能用多态了，也就没有所谓的父类引用指向子类实例。所谓静态，就是在运行时，虚拟机已经认定此方法属于哪个类。

子类一旦重写父类的方法, 就无法通过子类对象调用父类的被重写方法了, 如果要调用父类被重写方法, 可以使用super关键字进行调用。

|  |
| --- |
| **public** **class** SubClass **extends** SuperClass{  **public** **static** **void** main(String[] args) {  **new** SubClass().setSon("son");  **new** SubClass().method1();  }  //该方法重写了父类中的setSon(){}方法，重写的方法方法名和形参列表要相同  **void** setSon(String str) {  System.***out***.println("this is son"+str);  }  //子类重写的方法的返回值类型不大于父类被重写方法的返回值类型  Number methodNum() {  **return** **null**;  }  //子类重写方法的权限修饰符不小于父类被重写方法的权限修饰符  **public** **void** method1() {  //可以使用super调用父类中被重写的方法  **super**.setSon(father);  }  //子类和父类中同名同参的方法,要么同时加static且不是方法的重写，要么都不加static则是方法的重写  **static** **void** method() {}  }  **class** SuperClass {  String father = "father";  **void** setSon(String str) {  System.***out***.println("father is son's father"+str);  }  **void** method1() {}  Number methodNum() {**return** **null**;}  **static** **void** method() {}  } |

### **方法重载和方法重写的区别**

重载是在同一个类中；重写是在有继承关系的父子类中，子类中的方法重写父类的方法。

重载是类名相同。参数不同（数量、类型、顺序），与其他无关；重写是类名相同，参数相同，权限修饰符不小于父类，返回值类型不大于父类。

重载是编译时多态（静态绑定），重写是运行时多态（动态绑定）。属性不能重写，只有方法能够重写。方法的重写是动态绑定，属性是静态绑定。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **范围** | **方法名** | **参数列表** | **返回值类型** | **权限修饰符** | **抛出异常** |
| 重载overload | 同一个类中 | 同名 | 个数、类型、顺序不完全相同 | 无关 | 无关 | 无关 |
| 重写override | 具有继承关系的子父类中 | 同名 | 完全一致 | 子类重写的方法返回值类型不大于父类 | 子类重写的方法权限修饰符不小于父类 | 子类重写的方法抛出的异常类型不大于父类 |

### 方法的可变参数

如果需要给方法的形参传递的**个数不确定**且**类型相同**，那么用可变形参。

可变参数在使用时，会先创建一个数组，数组的长度就是调用改方法时传递的实参的个数，然后再将参数值全部放到这个数组中，最后将这个数组作为参数传递到被调用的方法中。可变参数是对数组的优化，和相同类型可变形参不构成方法的重载，因为可变形参底层本身就是一个数组。

可变形参必须放在形参列表的最后，因此一个方法中只能有一个可变形参。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  *add*(1,2,3,4,5);  *add*("1",2,3,4,5);  }  //形参传递的个数是不确定的且是同一类型，那么用可变形参  //如果方法中没有准确的参数，则传递的值会自动提升类型进行匹配  **static** **void** add(**double**...i) {  **int** sum = 0;  //可变形参相当于数组，因此可以用for循环遍历  **for**(**int** a =0;a<i.length;a++) {  sum += i[a];  }  System.***out***.println(sum);  }  //形参可以不是同一类型的，此时可变参数的位置要在参数列表的最后  **static** **void** add(String a,**double** b,**double**...c) {  System.***out***.println(b);  **for**(**double** num:c) {  System.***out***.print(num+"\t");  }  }  } |

### main方法

main方法是程序的入口，一个程序只有一个入口。main方法中不能直接调用非静态方法和实例变量。

public：公共的，用它修改的类或成员在任意位置可见

static：静态的，用它修改的方法，可以不用创建对象就可以调用

void：表示该方法没有返回值

main：Java的主方法名，JavaSE的程序入口

String[]：字符串数组，这是main方法的形参类型，可以通过命令行参数传值

args：这是main方法的形参名，如果要在main中使用命令行参数，可以遍历该args数组

main方法中形参传值方式：

在eclipse中给主程序的形参传值，右键RunAs -> RunConfigurations ->左边选类名（需要先运行一次 然后才能配置），右边选Arguments，直接输入就可以，会自动转换为字符串，多个参数用空格分开。

在Dos窗口中 java 字节码文件名 参数1 参数2 .....直接输入就可以，会自动转换为字符串，多个参数用空格分开。

jvm默认传递的是长度为0的字符串数组，我们在运行该类时，也可以指定具体的参数进行传递。可以在控制台，运行该类时，在后面加入参数。参数之间通过空格隔开。jvm会自动将这些字符串参数作为args数组中的元素，进行存储。

## 类成员三、构造器/构造方法

**构造器简介：**

构造器（构造方法）（**construtor**）：创建对象，给对象进行初始化。

格式：权限修饰符 类名（形参列表）{

方法体；

}

**构造器的特征**：

构造器名称必须和类名保持一致

没有返回值（不是返回值为空，与声明为void不同）

不能被static、final、synchronized、abstract、native修饰，可以有return表示结束构造器执行

构造器不像普通方法可以随意调用, 只能在创建对象时调用一次

**根据参数不同，构造器可以分为如下两类**：

隐式无参构造器（系统默认提供）

显式定义一个或多个构造器（无参、有参）

**注 意**：

每个类至少有一个构造器

创建对象时自动调用了参数一致的构造器

默认空参构造器的修饰符与所属类的修饰符一致

一旦显式定义了构造器，系统就不再提供默认构造器

一个类可以创建多个重载的构造器

构造器不可被子类继承

必须有一个构造器没有this，会造成构造器递归调用

在set方法中限制赋值的方式也可以在构造器中以同样的方式限制赋值，构造器和set方法并不冲突，后期可能要对对象中的属性值进行修改，而此时不能再通过构造器了。

构造器调用构造器时，只能将构造器调用语句放在代码的首行，即一个构造器只能调用一个构造器。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **new** Human();  **new** Human("Tom");  **new** Human("Tom",2.5);  }  }  **class** Human{  String name;  /\*  \* 构造器在对象创建的时候会被调用，且一个对象的调用只会执行一次构造器  \* 构造器名一定与类名相同，且不声明返回值类型，可以被权限修饰符修饰  \* 构造器不能被static\final\synchronized\abstract\native修饰  \*/  //每个类都有构造器，如果没有显示声明构造器就会默认提供一个隐式无参构造器  //一旦显示定义了一个构造器，系统则不再提供默认构造器  Human(){  System.***out***.println("this human have nothing");  }  //构造器可以传入参数并初始化属性值  Human(String name){  //构造器中可以对属性的赋值进行限制，并且可以进行其他功能的实现  **if**(name.length()<5) {  **this**.name = name;  }**else** {  System.***out***.println("this is not a legal");  }  System.***out***.println("this human is "+name);  }  //构造器也称为构造方法，支持方法的重载，一个类可以创建多个重载的构造器  Human(String name,**double** money){  System.***out***.println("this human is "+name+",and have"+money+"dollars");  }  } |
| this human have nothing  this human is Tom  this human is Tom,and have2.5dollars |

只有new的时候才是新建对象，单纯的使用“=”赋值并不是新建对象，可以使用BeanUtils.copyproperties方法快速复制属性到新对象。

## 类成员四、初始化块

**格式**：{

代码块；

}

**作用**：对Java类或对象进行初始化，代码块中声明的变量是局部变量，代码块只管初始化，构造器还起到创建对象的作用。

初始化块只能被static修饰，有static修饰的是静态代码块（**static block**），没有static修饰的是非静态代码块。

**静态代码块**：对类进行初始化；随着的类的加载而加载，类加载只加载一次，所以静态代码块也只执行一次；优先于非静态代码块；可以有多个，从上到下依次执行；不可调用实例变量和非静态方法。

**非静态代码块**：对对象进行初始化；非静态代码块时随着对象的创建而加载的，创建几次对象就加载几次非静态代码块；优先于构造器；可以有多个，从上到下依次执行；可以调用类变量和静态方法。

**执行顺序**：静态代码块->非静态代码块->构造器；

创建多个对象时，静态代码块只会在第一次创建对象时执行一次。非静态和构造器在创建一次对象时就会被调用一次，没有创建对象就不会调用。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  System.***out***.println(p.name);  System.***out***.println(p.eag);  System.***out***.println(p.id);  }  }  **class** Person{  **static** String *name*;  **static** **int** *id*;  **int** age;  **static**{  //静态代码块中不能调用非静态方法属性  *name* = "Tom";  }  **static**{  //执行顺序静态代码块先于非静态代码块，因此此处的赋值会被后面的赋值覆盖掉，结果为10001  *id* = 10000;  }  { //执行顺序非静态代码块先于构造器，因此此处的赋值会被后面的赋值覆盖掉  *name* = "Bob";  age = 18;  }  { //非静态代码块中可以调用静态属性方法  *id* = 10001;  }  //代码块只有初始化属性的作用，构造器还有创建对象的作用  **public** Person() {  *name* = "John";  age = 20;  }  } |
| John  20  10001 |

## 类成员五、内部类

内部类编译成功后会生成两个完全不同的字节码文件，内部类的字节码文件名称为：外部类名$内部类名.class，因此内部类的名字可以和外部类名字相同。

内部类可以有多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立.在单个外部类当中，可以让多个内部类以不同的方式实现同一接口，或者继承同一个类.创建内部类对象的时刻不依赖于外部类对象的创建。内部类并没有令人疑惑的”is-a”约束，它就像是一个独立的实体。

内部类提供了更好的封装，除了该外部类，其他类都不能访问。

内部类就是在一个类的内部定义的类， 非静态内部类中不能定义静态成员。 静态内部类不能访问外部类的静态成员。内部类作为其外部类的一个成员， 因此内部类可以直接访问外部类的成员。 但有一点需要指出：静态成员不能访问非静态成员，因此静态内部类不能访问外部类的非静态成员。

如果内部类使用了 static 修饰，那这个内部类就是静态内部类，也就是所谓的 static Nested Class；如果内部类没有使用修饰，它就是 Inner Class。除此之外，还有一种局部内部类：在方法中定义的内部类就是局部内部类，局部内部类只在方法中有效。

对于Static Nested Class来说， 它使用了static修饰， 因此它属于类成员， Static Nested Class的实例只要寄生在外部类中即可。因此使用 Static Nested Class 十分方便，开发者可以把外部类当成 Static Nested Class 的一个包即可。

对于 Inner Class 而言，它是属于实例成员，因此 Inner Class 的实例必须寄生在外部类的实例中， 因此程序在创建 Inner Class 实例之前， 必须先获得一个它所寄生的外部类的实例。否则程序无法创建 Inner Class 的实例。

### 内部类

在一个类A的内部再创建一个类B，类A叫做外部类（outer class），类B叫做内部类（inner class），内部类最大的特点是可以访问外部类私有属性，并且可以体现类与类之间的包含关系。

内部类可以直接访问外部类的成员，包括私有属性；外部类访问内部类，必须建立内部类对象；之所以可以直接访问外部类的成员，是因为内部类中持有了一个外部类的引用，格式为外部类名.this。

**内部类分为成员内部类和局部内部类**

成员内部类分为静态（成员）内部类和非静态（成员）内部类

内部类作为：

类的成员：可以用四种权限修饰符，可以用static修饰，内部类可以访问外部类的属性方法，

类：可以继承、可以有属性方法等等，类有的内部类都有

**局部内部类的使用**：

局部内部类的生命周期很短，获取局部内部类的对象需要利用多态，让局部内部类继承一个父类，方法的返回值类型为父类的类型。内部类在编译的时候也会生成class文件，不调用就不会加载，内部类在实际开发中几乎很少用。

局部内部类可以访问外部类的所有成员，但是不能添加权限修饰符，可以使用final修饰。局部内部类只能访问final修饰且在同一个作用域中的局部变量。

实际一般使用内部类都是为了封装，会使用private修饰从而防止内部类在外部创建对象，而是使用公有方法提供内部类的对象。

内部类编译后的文件名为：“外部类名$内部类名.java”。

**为什么内部类可以直接访问外部类中的成员呢？**

那是因为内部中都持有一个外部类的引用。这个是引用是 **外部类名.this**

内部类可以定义在外部类中的成员位置上，也可以定义在外部类中的局部位置上。

**当内部类被定义在局部位置上，只能访问局部中被final修饰的局部变量。**

|  |
| --- |
| //创建内部类的对象需要导入内部类  **import** test420.InnerA.AA;  **import** test420.InnerA.AB;  **public** **class** InnerClassTest {  **public** **static** **void** main(String[] args) {  InnerA ia = **new** InnerA();  //创建内部类对象：外部类对象.new 内部类();  AA aa = ia.**new** AA();  aa.method();  //创建内部类对象：new 外部类().new 内部类();  AA aa2 = **new** InnerA().**new** AA();  aa2.method();  //创建内部类对象：外部类对象.内部类();  AB ab = **new** InnerA.AB();  ab.method();    }  }  **class** InnerA{  //内部类可以访问外部类的属性和方法  String name = "InnerA";  String include = "include";  **static** **int** *staticNum* = 20;  //内部类作为类的成员，可以被四种权限修饰符修饰，可以被static修饰，可以访问外部类的属性方法  //非静态内部类  **public** **class** AA{  String name = "AA";  **public** **void** method() {  innerMethod();  //调用外部类的同名属性可以用外部类类名.this.属性/方法  System.***out***.println(InnerA.**this**.name +" "+ include +" "+ name);  }  }  //静态内部类  **protected** **static** **class** AB{  //静态内部类只能访问外部类的类变量和静态方法  **static** **int** *num* = 20;  **public** **void** method() {  *staticMethod*();  System.***out***.println(*staticNum*+*num*);  }  }  **private** **void** innerMethod() {  System.***out***.print("inner:");  }  **private** **static** **void** staticMethod() {  System.***out***.print("staticInner:");  }  } |

### 匿名内部类

匿名内部类没有类名，其声明的对象引用是利用了多态性，创建的对象实际上是其子类、实现类的对象，而不是类、抽象类和接口的对象，抽象类和接口是无法创建对象的。

匿名内部类可以直接访问外部类的所有成员，包含私有的；不能添加权限修饰符，且作用域仅仅在定义它的方法或代码块中；外部其他内不能访问匿名内部类。

|  |
| --- |
| **public** **class** AnonymousInnerClassTest {  //匿名内部类也可以作为属性，然后被调用  A aa = **new** A() {  **public** **void** method() {  System.***out***.println("aa + A");  }  };  **public** **static** **void** main(String[] args) {  AnonymousInnerClassTest aict = **new** AnonymousInnerClassTest();  aict.aa.method();  aict.aa.method2();  //普通类、抽象类、接口都可以设置匿名内部类，且创建的对象是其子类和实现类  //创建A的子类对象时就会调用A的构造器  A a = **new** A() {  //如果内部类中的方法不是重写的方法，此时对象的声明无法调用该方法，因为多态中父类无法调用子类特有的方法  **public** **void** method() {  System.***out***.println("A");  }  };  a.method();  a.method2();  System.***out***.println(a.a);  //创建B的子类对象时就会调用B的构造器，同时传入参数  B b = **new** B(100) {  **public** **void** method() {  System.***out***.println("B");  }  };  b.method();  System.***out***.println(b.b);  C c = **new** C(){  **public** **void** method() {  System.***out***.println("C");  }  };  c.method();  System.***out***.println(c.***c***);  }  }  **class** A{  **int** a = 10;  A(){  System.***out***.println("AAAA");  }  **public** **void** method() {  }  **public** **void** method2() {  System.***out***.println("method2 + AAAA");  }  }  **abstract** **class** B{  **int** b = 10;  B(**int** b){  System.***out***.println("BBBB"+" "+**this**.b+" "+b);  }  **abstract** **void** method();  }  **interface** C{  **int** ***c*** = 10;  **void** method();  } |
| AAAA  aa + A  method2 + AAAA  AAAA  A  method2 + AAAA  10  BBBB 10 100  B  10  C  10 |

## 面向对象三大特性

### 封装

使用者对类内部定义的属性和某些方法的直接操作会导致数据的错误、混乱或安全性问题，因此需要将这些属性和方法进行封装。

封装（encapsulation）是指通过将数据声明为私有的(private)，再提供公共的（public）方法实现对该属性的操作。创建对象以后，就可以通过公开的方法获取封装属性和对封装属性赋值，并在赋值过程中限制该数据的类型和范围等。

封装性的体现（狭义上）：

私有化属性并提供公共方法给属性赋值和获取属性值

封装性的体现（广义上）：

四种权限修饰符public protected default private用来修饰属性、方法、构造器、内部类。

类只能被public和default修饰。

从广义上看，也可以将属性使用public和final修饰来体现封装性，因为虽然public修饰的属性可以被所有人调用，但是由于final的修饰导致其不可被修改，对属性的改变必须重新创建对象，因此避免了对属性的直接操作导致的安全问题。在实际开发中可以根据需求使用这种方法。

### 继承

多个类中存在相同属性和方法时，将这些内容抽取到单独一个类中，那么多个类无需再定义这些属性和行为，只要继承那个类即可。子类继承父类后就拥有了父类中的属性和方法，但是构造器不能被继承。Java中是单继承的，一个子类只能有一个父类，但是一个父类可以有多个子类。

类的继承：A extends B ：A类继承（扩展）B类

A 子类（subclass）

B 父类，超类，基类（superclass）

**优点**： 减少了代码的冗余，提高了代码的复用性；提高了代码的扩展性；为多态提供了前提。

父类中的属性方法可能因为封装私有化，子类不能直接调用，但是被封装属性方法仍是被继承到的。

父类分为直接父类和间接父类，子类不但继承了直接父类中属性方法，还继承了间接父类中属性方法。

一个类如果没有显式的继承其他类，那么默认继承Object类，Object类是所有类的基类。

子类除了继承父类中属性方法，还能定义自己的独有属性方法。

确认继承关系时要考虑实际情况，使子类继承父类存在 **子类 is a 父类** 的关系。

跨包继承必须要将类定义为public类型的，默认类型的类无法被跨包继承，public修饰的类必须要单独存放一个文件。

当创建子类时，不管你使用子类的哪个构造方法，默认情况下总会去调用父类的无参构造函数，如果父类没有提供无参构造函数，则必须在子类的构造函数中用 super 去指定使用父类的哪个构造函数完成对父类的初始化工作，否则，编译不会通过。

**为什么子类一定要访问父类的构造函数**：因为父类中的数据子类可以直接获取，所以子类对象在建立时，需要先查看父类是如何对这些数据进行初始化的。所以子类在对象初始化时，要先访问一下父类中构造函数。如果要访问父类中指定的构造函数，可以通过手动定义super语句的方式来指定。

子类中至少会有一个构造函数会访问父类中的构造函数。

只有子类调用了父类的构造器，才将类信息加载到内存中，因此创建子类对象时一定会调用父类的构造器。

|  |
| --- |
| //直接继承的是直接父类，间接继承的是间接父类  //如SuperClass是直接父类，SuperSuperClass是间接父类  **public** **class** SubClass **extends** SuperClass{  **public** **static** **void** main(String[] args) {  SubClass son = **new** SubClass();  //子类可以调用直接父类和间接父类的属性和方法  System.***out***.println(son.father);  //子类无法直接调用直接父类或间接父类的私有属性和方法  //System.out.println(son.grandfather);  System.***out***.println(son.getGrandfather());  son.setSon();  }  }  **class** SuperClass **extends** SuperSuperClass{  String father = "father";  **void** setSon() {  //super指向父类的对象，利用父类的对象调用父类的方法  **super**.setSon();  System.***out***.println("father is son's father");  }  }  //一个父类可以有多个子类，但是一个子类只能有一个父类  **class** SuperClassCopy **extends** SuperSuperClass{  }  **class** SuperSuperClass{  **private** String grandfather = "grandfather";  **void** setSon() {  System.***out***.println("grandfather is father's father");  }  String getGrandfather() {  **return** grandfather;  }  } |
| father  grandfather  grandfather is father's father  father is son's father |

### 多态

Java引用变量有两个类型：编译时类型和运行时类型。编译时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，若编译时类型和运行时类型不一致，就出现多态（Polymorphism）。

**本态**：superclass sc = new superclass();

**多态**：superclass sc = new subclass();

**广义上多态性**

方法的重载（静态绑定/编译时多态）：编译时就确定调用的是哪一个

方法的重写（动态绑定(**dynamic binding**)/运行时多态）：编译时不确定调用的是哪一个，运行时才能确定——虚拟机实现方式

子类对象的多态性：父类的引用指向子类的对象

**狭义上多态性**

子类对象的多态性：父类的引用指向子类的对象

**多态的前提**

要有继承关系

要有方法的重写

父类的引用指向子类的对象

**多态的实现**

接口实现

继承父类重写方法

同一类中方法重载

多态下的动态绑定技术：编译看左边，运行看右边，编译看左边有没有调用的方法属性，运行的时候看右边有没有调用的方法。

父类的引用虽然指向子类的对象，但是该引用无法直接调用子类独有的方法属性。可以通过将父类的引用向下转型的方式调用子类中独有的方法：（subclass）superclass.子类独有方法。superclass向下转型如果转的不是其指向的子类对象的类型，虽然编译不会报错，但是运行时会报ClassCastException（类型转换异常）错误。多态看的是对象本身是什么类型，才可以强转成那个类型，一般强转之前会使用instanceof判断能否强转。

**注意：普通对象类型转换要有多态作为前提，然后还要判断能否转型成子类。**

类型判断：

a instanceof A

A对象如果是a对象的相同类型或者a的直接父类或间接父类，都会返回true，否则返回false。

A <- B <- C :C继承了B，B继承了A

C的对象c c instanceof C true

c instanceof B true

c instanceof A true

A的对象a a instanceof A true

a instanceof B false

a instanceof C false

如果有多个条件判断语句用来判断对象的类型，那么子类写在上面，父类写在下面。

属性有没有多态性？没有，属性属于静态绑定。

静态方法也没有多态性，因为多态指的是父类的引用指向子类的对象，而static修饰的内容和对象无关，可以直接由类名调用。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //多态：父类的引用指向子类的对象  SuperClass sc = **new** Subclass();  //调用子类的方法  sc.method();  //属性没有多态  System.***out***.println(sc.num);  //可以通过强转调用子类特有的方法  ((Subclass)sc).special();  //此时虽然编译没有错误，但是运行时会报类型转换异常错误，可以先用instanceof判断  //A对象如果是a对象的相同类型或者a的直接父类或间接父类，都会返回true，否则返回false。  **if**(sc **instanceof** Subclass2) {  Subclass2 sub2 = (Subclass2)sc;  }**else** {  System.***out***.println("对象本身是什么类型，才可以强转成那个类型");  }  //这种情况下调用的仍然是对象中的方法，即子类对象中的方法  SuperSuperClass spsp = **new** Subclass();  SuperClass sp = (SuperClass)spsp;  sp.method();  }  }  **class** Subclass **extends** SuperClass{  **int** num = 10;  **public** **void** method() {  System.***out***.println("subMethod");  }  **public** **void** special() {  System.***out***.println("special");  }  }  **class** Subclass2 **extends** SuperClass{  **public** **void** method() {  System.***out***.println("sub2Method");  }  }  **class** SuperClass **extends** SuperSuperClass{  **int** num = 20;  **public** **void** method() {  System.***out***.println("superMethod");  }  }  **class** SuperSuperClass{  **public** **void** method() {  System.***out***.println("supersuperMethod");  }  } |

多态使用的场景：

多态多用于方法的形参，将子类的对象赋值给了父类的引用，只要是superclass的子类的对象都可以作为实参传递过来。

**多态参数**

在设计方法时, 有的方法内部需要用到另外的类的对象, 但是在设计方法时, 又不需要关心究竟是哪个对象, 只关心这些实参对象都是某个类型的, 此时, 方法的参数的类型就可以写成父类类型的参数,这样的方法的参数就是多态参数。**即对象作为参数时可以传入本类对象及其子类对象。**

**多态参数优点**

替换性，多态对已存在代码具有可替换性

扩充性：增加新的子类不影响已存在的类结构

接口性：多态是父类通过方法，向子类提供一个公共接口，由子类来完善或重写

灵活性

简化性

**多态数组**

在引用类型的数组中，使用多态形式存放对象

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //多态数组，用多态的形式将对象存放到数组中  Person[] person = **new** Person[2];  person[0] = **new** Student();  person[1] = **new** Teacher();  **for**(Person p : person) {  p.talk();  *exa*(p);  }  }  //多态用于方法的形参，将子类的对象赋值给了父类的引用  **static** **void** exa(Person p) {  //根据父类的引用指向的子类对象不同而调用不同的属性方法  System.***out***.println(p.name);  }  }  **class** Person{  String name;  **int** age;  **void** talk() {  }  }  **class** Student **extends** Person{  {  **this**.name = "stu";  }  **void** talk() {  System.***out***.println("Student");  }  }  **class** Teacher **extends** Person{  {  **this**.name = "tea";  }  **void** talk() {  System.***out***.println("Teacher");  }  } |

**多态在子父类中的成员上的体现的特点：**

**1，成员变量：**在多态中，子父类成员变量同名。

在编译时期：参考的是**引用型变量所属的类**中是否有调用的成员。（**编译时不产生对象，只检查语法错误**）

运行时期：也是参考**引用型变量所属的类**中是否有调用的成员。

**简单一句话：无论编译和运行，成员变量参考的都是引用变量所属的类中的成员变量。**

再说的更容易记忆一些：**成员变量** --- **编译运行都看 = 左边。**

**2，成员函数。**

编译时期：参考引用型变量所属的类中是否有调用的方法。

运行事情：参考的是对象所属的类中是否有调用的方法。

为什么是这样的呢？**因为在子父类中，对于一模一样的成员函数，有一个特性：覆盖。**

简单一句：成员函数，编译看引用型变量所属的类，运行看对象所属的类。

**更简单：成员函数** --- **编译看 = 左边，运行看 = 右边。**

**3，静态函数。**

编译时期：参考的是引用型变量所属的类中是否有调用的成员。

运行时期：也是参考引用型变量所属的类中是否有调用的成员。

为什么是这样的呢？**因为静态方法，其实不所属于对象，而是所属于该方法所在的类。**

调用静态的方法引用是哪个类的引用调用的就是哪个类中的静态方法。

**简单说：静态函数** --- **编译运行都看 = 左边。**

## 关键字

### package关键字

包（package）的作用：帮助管理大型软件系统；包可以包含类和子包，划分项目层次，方便对类进行统一的管理和划分；解决类命名冲突的问题；控制访问权限。

包命名规范：机构的类型(com,org) + 机构的名称（baidu,sina） + 项目名 + 模块名，每"."代表一层目录，包名应该遵守标识符的规则和规范 : xxxyyyzzz，同一个包中的类名不能相同。

package com.xxx.xxx;必须放在源文件的首行，告诉编译器生成的字节码文件应该存放的目录。

**JDK中主要的包介绍**

1.java.lang----包含一些Java语言的核心类，如String、Math、Integer、System和Thread，提供常用功能。

2.java.net----包含执行与网络相关的操作的类和接口。

3.java.io----包含能提供多种输入/输出功能的类。

4.java.util----包含一些实用工具类，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。

5.java.text----包含了一些java格式化相关的类

6.java.sql----包含了java进行JDBC数据库编程的相关类/接口

7.java.awt----包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。

### import关键字

在源文件中使用import显式的导入指定包下的类或接口，import声明在包（package）的声明和类（class）的声明之间。如果需要导入多个类或接口，那么就并列显式多个import语句。可以使用java.util.\*的方式，一次性导入util包下所有的类或接口。如果导入的类或接口是java.lang包下的，或者是当前包下的，则可以省略此import语句。

如果在代码中使用不同包下的同名的类，那么就需要使用类的全类名（包名.类名）的方式指明调用的是哪个类。

静态导入：import和static组合使用，调用指定类或接口下的静态属性和方法；静态导入可以直接导入到方法的级别；已经导入java.a包下的类，如果需要使用a包的子包下的类，仍需要导入。

### this关键字

this相当于当前对象，可以调用属性，方法，无法调用局部变量。哪个对象调用的方法，this就代表哪个对象。

this对象后面跟上 **.** 调用的是成员属性和成员方法(一般方法)；

this对象后面跟上 () 调用的是本类中的对应参数的构造函数。

方法内，通过this调用当前对象中属性和方法，一般情况下，都会省略掉this。但是当局部变量和属性同名时，不可省略this，需要this区分局部变量和属性。

构造器中使用this调用别的构造器时要位于方法体的第一行，一个构造器中只能有一个this，且类中至少有一个构造器没有用this调用其他构造器。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **new** Human();  **new** Human("Tom");  **new** Human("Tom",2.5);  }  }  **class** Human{  String name;  Human(String name){  //构造器中使用this调用别的构造器时，this必须写在第一行，因此可以知道一个构造器中只能有一个this调用别的构造器  **this**();  **if**(name.length()<5) {  //this相当于当前对象，哪个对象调用的方法、属性，this就指向哪个对象  //可以用this来区分局部变量和成员变量（属性）  **this**.name = name;  }**else** {  System.***out***.println("this is not a legal");  }  //this指向当前对象并调用saySomething方法  **this**.saySomething();  //此处虽然没有显示的写出this，但实际上是省略了this  //一般情况下，都会省略this，但是当形参和属性同名时，为了区分开来，this就不可省略了  saySomething();  System.***out***.println("this human is "+name);  }  Human(){  //一个类中至少有一个构造器不包含this调用别的构造器  System.***out***.println("this human have nothing");  }  Human(String name,**double** money){  **this**(name);  System.***out***.println("this human is "+name+",and have"+money+"dollars");  }  **public** **void** saySomething() {  System.***out***.println("this man is changed");  }  } |
| this human have nothing  this human have nothing  this man is changed  this man is changed  this human is Tom  this human have nothing  this man is changed  this man is changed  this human is Tom  this human is Tom,and have2.5dollars |

### super关键字

super关键字调用父类的属性、方法、构造器，作用和this类似，但是是相当于父类对象。

this区分局部变量和属性，super区分子类和父类。调用属性方法时，this先从本类中查找，如果本类中不存在再从父类中查找；super直接在父类中查找，如果父类中不存在则报错。

子类所有构造器默认都会访问父类中空参构造器，父类中没有空参构造器时，子类构造器必须使用this或super指定调用本类或父类中对应构造器。

**super用来调用属性和方法**

子类继承父类后，子类直接调用父类的属性方法，此时默认省略super。若子类的属性和父类的属性名同名或子类重写父类的方法后，再调用父类的属性方法，就要用super区分子类和父类的属性方法。

super.getClass().getName()的结果仍然是本类的类名而不是父类的类名，因为super只是一个限定词，当用super引用时，也是引用当前对象本身，只是super限定了访问当前对象从父类那里继承得到的属性和方法。如果需要访问到父类的类名，使用如下方式：super.getClass().getSuperclass().getName()。

**super用来调用构造器**

构造器中如果有super则必须位于第一行，因此this和super在同一构造器中只能使用其中一个。若构造器中没有显式使用this或super，则默认调用的是super()。

**子类的实例化过程**

结果上：子类继承父类后就有了父类的属性方法。

过程上：创建子类对象时，会调用直接父类构造器，间接父类构造器......Object类构造器，JVM会将这些父类信息加载到内存中。自始至终，我们只认为创建了子类一个对象。

**注意：只要调用父类的方法，无论子类是否有同名方法，都要在方法前使用super关键字，避免混淆。**

|  |
| --- |
| **public** **class** SubClass **extends** SuperClass{  **public** **static** **void** main(String[] args) {  //当创建子类对象时，一定会调用到直接父类的构造器，间接父类的构造器......Object类的构造器  SubClass son = **new** SubClass();  System.***out***.println(son.father);  System.***out***.println(son.getFather());  son.setGrandfather();    SubClass son2 = **new** SubClass("Bob");  SuperSuperClass superSuperClass = **new** SuperSuperClass("TOM");  }  String father = "my father";  //子类中直接调用父类的属性，省略掉了super  String grandParents = grandfather;  SubClass(){  //如果在一个构造器中既没有显式的使用this（形参列表）和super（形参列表），则默认调用的是super();  //如果父类没有空参构造器，报错：因此父类一定要显式的提供一个空参构造器。  **super**();  System.***out***.println("this is son");  }  SubClass(String name){  //如果构造器中有this调用了其他构造器，则不再会有super调用父类的构造器，因为这两个关键字不能同时用于构造器  **this**();  System.***out***.println(name);  }  String getFather() {  //this可以省略，指向的是子类的对象；super不可省略，用于区分子类和父类，指向父类的对象  **this**.father = **super**.father;  **return** **this**.father;  }  **void** setGrandfather() {  //this先在子类中查找，如果不存在则会去父类中查找，super则直接在父类中查抄。  System.***out***.println(**super**.grandfather);  System.***out***.println(**this**.grandParents);  }  }  **class** SuperClass **extends** SuperSuperClass{  String father = "father";  SuperClass() {  **super**("forever");  System.***out***.println("father is son's father");  }  }  **class** SuperSuperClass{  String grandfather = "grandfather";  SuperSuperClass() {  System.***out***.println("who want me");  }  SuperSuperClass(String str){  System.***out***.println("grandfather is father's father"+str);  }  } |
| grandfather is father's fatherforever  father is son's father  this is son  my father  father  grandfather  grandfather  grandfather is father's fatherforever  father is son's father  this is son  Bob  grandfather is father's fatherTOM |

### this和super的区别

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 访问属性 | 访问本类的属性，如果没有此则在父类查找 | 访问父类的属性 |
| 2 | 调用方法 | 访问本类的方法，如果没有此则在父类查找 | 访问父类的方法 |
| 3 | 调用构造器 | 调用本类构造器，必须放在构造器首行 | 调用父类构造器，必须放在子类构造器首行 |
| 4 | 特殊 | 表示当前对象 | 表示父类对象 |

### static关键字

static可以修饰属性、方法、代码块和内部类，不可以修饰局部变量。static 修饰的数据在内存空间里只有一份，被所有类的对象共享，可以节省内存空间。使用static修饰的内容生命周期很长。

**static修饰属性**

有static修饰的属性称为类变量，所有对象共享，存放在方法区。

同一个类的多个对象，各自拥有一份实例变量，共同拥有一份类变量。某个对象对类变量进行赋值或修改后，其他对象的类变量随之发生改变。实例变量在内存的堆中，类变量在内存的方法区中。类变量随着类的加载而加载，因为类加载只加载一次，所以类变量也只加载一次；实例变量是随着对象的创建而加载。

**调用类变量**

对象名.类变量名;（不推荐这种调用方式，不符合类变量共享和对象无关的特性）

类名.类变量名;

不能用类名.实例变量名调用实例变量，因为实例变量是随着对象的创建而加载，所以只能由对象去调用。而类变量随着类的加载而加载，所以可直接用类名调用。

**属性分类**

按位置：局部变量和成员变量（属性）

成员变量可以分为类变量和实例变量

类变量：static修饰的变量

实例变量：不是static修饰的变量

按类型：基本数据类型和引用数据类型

**成员变量和静态变量的区别：**

1，成员变量所属于对象。所以也称为实例变量。

静态变量所属于类。所以也称为类变量。

2，成员变量存在于堆内存中，每创建一个对象堆中就有一份成员变量。

静态变量存在于方法区中，无论创建多少个对象，都只有一份。

3，成员变量随着对象创建而存在。随着对象被回收而消失。

静态变量随着类的加载而存在。随着类的消失而消失。

4，成员变量只能被对象所调用 。

静态变量可以被对象调用，也可以被类名调用。

**所以，成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。**

**类加载的过程**

1、首次使用类（创建对象、直接调用类变量或静态方法等）时，先去方法区查看是否有该类的信息。

2、如果方法区中没有该类的信息，则进行类加载（将字节码文件加载到JVM中，将类的信息放到方法区中）；如果方法区中有了该类的信息，则直接创建对象。

3、JVM将字节码文件加载到方法区的同时，会在方法区中创建类变量、静态方法等信息。

4、方法区中加载类的信息后，就可以创建对象了。

5、第二次使用该类时，方法区中已存在该类的信息，便不再进行类加载了（类加载只加载一次）。

**static修饰方法**

有static修饰的方法是静态方法/类方法，没有static修饰的方法就是非静态方法。

静态方法是随着类的加载而加载，静态方法中不能调用实例变量和非静态方法，因为加载时机不同，实例变量和非静态方法是随着对象的创建而加载，静态方法是随着类的加载而加载，类加载优先于对象创建，没有对象就不能调用实例变量和非静态方法。

静态方法中不能使用this和super，因为这两个都指的是对象，而静态方法优先于对象创建。

非静态方法在方法区中，不同于堆中的实例变量有多份，非静态方法只有一份，在调用时创建引用。非静态方法随着对象的创建而加载，和静态方法一样都只有一份，只有创建对象时才有引用指向非静态方法。

**调用静态方法**

对象名.静态方法名;（不推荐这种调用方式，不符合类变量共享和对象无关的特性）

类名.静态方法名;

**静态方法和非静态方法的调用**

非静态方法可以直接调用非静态方法和静态方法。静态方法可以直接调用静态方法，不可以直接调用非静态方法，但是可以通过new对象和传参两种方式间接调用。

非静态环境中可以直接使用静态成员, 因为此时对象必然已经创建了, 类也肯定已经加载过了。

静态环境中不可以直接使用非静态成员，因为静态成员不要求创建对象，只要求类加载就可以了，但是类加载并不意味要创建对象，所以静态环境中不可以直接访问非静态成员。因为非静态成员隶属于对象，所以静态环境中只能通过创建对象, 再通过对象来间接访问非静态成员。

通过类名可以直接调用静态方法，不可以直接调用非静态方法。

通过对象名可以直接调用非静态方法和用静态方法。

**静态和非静态的本质**

非静态也可以称为动态, 意思是不确定的, 在程序中对象的创建是不确定的, 因为只要new就会产生对象, 所以非静态的概念必然是和对象相关的。

静态的就是确定的, 唯一的, 在程序中是稳定的, 和有无对象无关, 也和有多少对象无关, 只和类相关, 所以, 所以静态成员是稳定并唯一的。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  //实例变量必须由对象来调用，因为其随着对象的创建而加载  System.***out***.println(p.age);  //类变量可以由类名直接调用，因为其随着类的加载而加载  //此时name的值因为没有任何赋值操作，因此默认为空  System.***out***.println(Person.*name*);  //非静态方法必须由对象来调用  p.method2();  //静态方法可以由类名直接调用  Person.*method1*();  }  }  **class** Person{  //static可以修饰属性、方法、代码块和内部类，不能修饰局部变量  //name称为类变量，age称为实例变量，实例变量在内存的堆中，类变量在内存的方法区中  **static** String *name*;  **int** age;  **static** **void** method1() {  //静态方法可以只能调用类变量和静态方法，不能调用实例变量和非静态方法  *name* = "Tom";  System.***out***.println(*name*);  //静态方法不能使用this和super，因为两者都是指向对象，而静态方法先于对象加载  // this.age = 10;  }  **void** method2() {  //非静态方法可以调用静态方法和类变量，也能调用实例变量和非静态方法  *name* = "Tom";  System.***out***.println(*name*);  **this**.age = 10;  }  } |
| 0  null  Tom  Tom |

**static使用场景**

static修饰属性：

如果同一个类的多个对象共同拥有一份属性，那么使用static修饰

常量需要使用static修饰，便于用类名直接调用

static修饰方法：

作为工具类的方法，需要用static修饰，便于调用

为了调用类变量，将方法使用static修饰

static修饰初始化块

static修饰内部类

static静态导包：

在JDK 1.5之后引入的新特性，可以用来指定导入某个类中的静态资源，并且不需要使用类名。

|  |
| --- |
| **public** **class** FrockTest {  **public** **static** **void** main(String[] args) {  //此时多个对象拥有同一份属性，该属性共有，一个对象的变化会影响到另一个对象  **new** Frock();  **new** Frock();  **new** Frock();  }  }  **class** Frock{  **private** **static** **int** *currentNum* = 100000;  **private** **int** serialNumber;  Frock(){  serialNumber = *getNextNum*();  System.***out***.println(serialNumber);  }  **public** **static** **int** getNextNum() {  *currentNum* += 100;  **return** *currentNum*;  }  **public** **int** getSerialNumber() {  **return** serialNumber;  }  **public** **void** setSerialNumber(**int** serialNumber) {  **this**.serialNumber = serialNumber;  }  } |
| 100100  100200  100300 |

### final关键字

**final简介**

final关键字用于声明类、变量和方法，变量包括属性和局部变量。

final修饰的类不能被继承，提高了安全性和程序的可读性，比如String、StringBuffer都被final修饰。

final修饰的方法不能被子类重写，JVM会尝试将其内联，以提高运行效率。

final修饰的变量的值只能赋值一次，赋值后就不能被修改（用final修饰的变量叫做常量，存入常量池中）。常量前面经常用static和final一起修饰，例如Math类中的PI常量，static方便用类名直接调用，final确保常量不会被修改。

当 final 修饰局部变量时，该局部变量可以被一次赋值，以后该变量的值不能发生改变；当 final 修饰实例变量时，实例变量必须由程序员在构造器、初始化块、定义时这 3 个位置的其中之一指定初始值；当 final修饰静态变量时， 静态变量必须由程序在静态初始化块、 定义时这 2 个位置的其中之一指定初始值。

**编译器对final域要遵守的两个重排序规则**

1.在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量,这两个操作之间不能重排序。  
 2.初次读一个包含final域的对象的引用，与随后初次读这个final域,这两个操作之间不能重排序。

**final修饰的属性的赋值方式**

显式赋值：final修饰的属性没有默认值，声明时进行初始化（一般都是显式赋值）。

构造器赋值：若有多个构造器，则分别给final修饰的属性赋值。

代码块赋值：若已经在代码块中为final修饰的属性赋值，则无法再在构造器中对其赋值。

**无法在方法内给final修饰的属性赋值**

|  |
| --- |
| //final修饰的类不能被继承  **final** **class** Creature{    }  **class** Person{  //final定义的常量必须要进行初始化，可以显式赋值，构造器赋值和代码块赋值  **final** String species1 = "human";  **final** String species2;  **final** String species3;  {  species2 = "human";  }  Person(){  species3 = "human";  //final修饰分变量不能被修改，一般将这样的变量称之为常量，存储在常量池中  // species1 = "no human";  }  **public** **void** method() {  //final修饰的属性不能在方法中对其赋值初始化  // species3 = "human";  }  **final** **void** method2() {  System.***out***.println("no change ");  }  }  **class** Man **extends** Person{  //Cannot override the final method from Personfinal修饰的方法无法进行重写  // void method2() {  //  // }  } |

**final、finalize和finally的不同之处**

final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。

finalize 方法是在对象被回收之前调用的方法，给对象自己最后一个复活的机会，但是什么时候调用 finalize 没有保证。

finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

### 权限修饰符关键字

权限修饰符public、protected、private置于类的成员（属性、构造器、方法、内部类）定义前，用来限定对象对该类成员的访问权限，对于class的权限修饰只可以用public和default。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 权限修饰符 | 同一个类中 | 同一个包中 | （不同包中的）子类 | 任何地方 |
| public | yes | yes | yes | yes |
| protected | yes | yes | yes |  |
| default | yes | yes |  |  |
| private | yes |  |  |  |

### native关键字

**native含义**

native表示本地的意思，可以修饰方法，被native修饰的方法没有方法体。native修饰的方法和当前操作系统密切相关，是指用C/C++等编写的, 封装在操作系统相关的库文件中的方法，执行该方法时，实际执行的是库中的代码。

**native关键字使用场景**

有一些代码需要和底层设备交互, 而这些交互的功能并未由JDK提供现成的类库, 那么必须使用native方式让java程序和库交互, 再由库完成对底层设备的使用。

**native关键字特点**

用native修饰的方法是调用的库中的代码，该代码换一个平台就会出现问题，无法实现跨平台。

## 抽象与接口

### 抽象

抽象类除了不能创建实例，其他的和普通类一样，可以继承普通类，可以有main方法，可以实现接口等等。

**抽象的意义**

为其他子类提供一个公共的类型

封装子类中重复定义的内容

定义抽象方法，子类虽然由不同的实现，但是定义时是一致的

**abstract**关键字

abstract修饰的方法叫做抽象方法，抽象方法没有方法体且其所在的类必须为抽象类。

abstract修饰的类叫做抽象类，抽象类不能被实例化（创建对象）；但是有构造器，子类实例化过程会调用父类的构造器，加载父类的信息；抽象类中可以没有抽象方法而有非抽象方法。

抽象类的子类必须要重写直接和间接父类中所有的抽象方法，否则这个类还是抽象类。

不能用abstract修饰属性、构造器、私有方法、静态方法和final的方法,因为构造器、私有方法、静态方法和final修饰的方法都不能被重写。

private：子类无法显式继承并重写这个方法；

final：修饰类无法被继承，修饰方法无法被重写；

static：静态方法可以通过类名直接调用，但是抽象方法没有方法体，因此这样调用无意义；

constructor：无法被继承。

native：本地方法表示要用另外一种依赖平台的语言实现的，不存在被子类实现的情况

synchronized：该关键字修饰一个方法时，表明将会使用该方法的调用者作为同步监视器，但对于一个 abstract 方法而言，它所在类是一个抽象类，抽象类也无法创建实例，因此也就无法确定 synchronized 修饰方法时的同步监视器了，因此 synchronized 与 abstract 不能同时使用。

|  |
| --- |
| **public** **class** AbstractTest {  **public** **static** **void** main(String[] args) {  //抽象类虽然不能实例化，但是其仍然有构造器，因为其子类创建对象时就会调用其构造器  subA sa = **new** subA();  sa.method();  sa.superMethod();  sa.nonAbstractMethod();  }  }  **abstract** **class** abstractSuperA{  **abstract** **void** superMethod();  **abstract** **void** superOverrideMethod();  }  **abstract** **class** abstractA **extends** abstractSuperA{  //抽象方法所在的类必须为抽象类  abstractA(){  System.***out***.println("I have constructors");  }  **abstract** **void** method();  //抽象类中可以有非抽象方法  **public** **void** nonAbstractMethod() {  System.***out***.println("I have non-abstract method");  }  **public** **void** superOverrideMethod() {  System.***out***.println("I have override this method");  }  }  **class** subA **extends** abstractA{  //子类必须重写其直接父类和间接父类的抽象方法，如果不想进行方法的重写，就必须要将子类也声明为抽象类  **public** **void** method() {  System.***out***.println("override method");  }  **public** **void** superMethod() {  System.***out***.println("override supersuperclass method");  }  } |

### 接口

接口通过interface关键字创建，可以用public和abstract修饰；类通过implements关键字实现接口，接口可以被理解为一种标准，所有实现接口的类都要实现接口中的方法。

接口的格式：interface 接口名{

}

接口和抽象类都不能实例化，即不能创建对象，接口内没有构造器和初始化块。

接口中（JDK1.8前）只能有常量和抽象方法，（JDK1.8后）多了静态方法和默认方法。

接口中的所有成员变量都默认是由public static final修饰的，接口中的所有抽象方法都默认是由public abstract修饰的。

接口和类的关系是实现且是多实现的关系，类 implements 接口1，接口2...{}，如果实现接口的类中没有实现接口中的全部方法，必须将此类定义为抽象类；接口和接口之间是继承且是多继承的关系。

接口和实现接口的类之间也有多态性。

**接口的意义**：**规范、扩展和回调**

|  |
| --- |
| **public** **class** InterfaceTest {  **public** **static** **void** main(String[] args) {  InterfaceTest it = **new** InterfaceTest();  Mouse mouse = **new** Mouse();  //接口和类之间的多态性，和子类与父类之间的多态性相同  USB usb = mouse;  BlueTooth bt = mouse;  it.open(usb);  it.bt(bt);  }  **void** open(USB usb) {  System.***out***.println(usb.***version***);  usb.start();  usb.close();  usb.change();  usb.update();  }  **void** bt(BlueTooth bt) {  System.***out***.println(bt.***version***);  bt.connect();  bt.disconnect();  }  }  **interface** OldUSB{  **public** **abstract** **void** update();  }  **interface** NewUSB{  **public** **abstract** **void** change();  }  //接口之间的关系是继承关系，且是多继承  **interface** USB **extends** OldUSB,NewUSB{  **public** **static** **final** String ***version*** = "1.0";  **public** **abstract** **void** start();  **public** **abstract** **void** close();  }  //接口和抽象类都不能实例化，即不能创建对象，接口内没有构造器和初始化块  **interface** BlueTooth{  //默认省略了public static final  String ***version*** = "3.0";  //默认省略了public abstract  **void** connect();  **void** disconnect();  }  //实现类，可以多实现，必须要重写实现的所有接口即其父类接口中的抽象方法  **class** Mouse **implements** USB,BlueTooth{  **public** **void** update() {  System.***out***.println("OldUSB update");  }  **public** **void** connect() {  System.***out***.println("BlueTooth connect");  }  **public** **void** disconnect() {  System.***out***.println("BlueTooth disconnect");  }  **public** **void** start() {  System.***out***.println("USB start");  }  **public** **void** close() {  System.***out***.println("USB close");  }  **public** **void** change() {  System.***out***.println("NewUSB change");  }  } |

### 接口JDK1.8新特性

JDK1.8之后接口中声明静态方法和默认方法

static void 方法名(){

方法体;

}

default void 方法名(){

方法体；

}

如何调用接口中的静态方法：接口名.方法名();

如何调用接口中的默认方法：类实现接口后，可以直接调用，其实和普通方法一样省略了this，实际为this（实现类对象）.默认方法名();

类优先原则：如果继承的父类和实现的接口中有同名的默认方法，则父类中的方法优先

如果实现的多个接口中有相同的默认方法，则强制重写该方法，否则会报错；此时如果要调用接口中被重写的相同的默认方法，则可以用：接口名.super.方法名();来调用。

|  |
| --- |
| **public** **class** NewInterfaceTest {  **public** **static** **void** main(String[] args) {  //接口中的静态方法可以直接用接口名.方法名直接调用  newInter.*smethod*();  newInter a = **new** interA();  ((interA)a).method();  }  }  **interface** newInter{  **static** **void** smethod() {  System.***out***.println("this is a static method");  }  **default** **void** dmethod() {  System.***out***.println("this is a default method");  }  **default** **void** dmethod2() {  System.***out***.println("this is a default method2");  }  }  **interface** newInter2{  **default** **void** dmethod() {  System.***out***.println("this is a default method,too");  }  }  **class** superA{  **public** **void** dmethod2() {  System.***out***.println("I'm father's method");  }  }  **class** interA **extends** superA **implements** newInter,newInter2{  **public** **void** method() {  //实现接口后可以直接调用接口中的默认方法，和普通方法一样省略了this，实际为this.默认方法名();  dmethod();  //如果要调用接口中被重写的默认方法，可以用接口名.super.方法名；来调用  newInter2.**super**.dmethod();  newInter.**super**.dmethod();  //类优先原则，继承的父类有实现接口中默认方法同名的方法，则父类方法优先  dmethod2();  }  @Override  //实现的多个接口中有同名的默认方法，则强制重写该方法  **public** **void** dmethod() {  System.***out***.println("I'm interA");  newInter.**super**.dmethod();  newInter2.**super**.dmethod();  }  } |

### 接口和抽象的区别

接口和抽象都不能实例化。是上层的抽象。

|  |  |  |
| --- | --- | --- |
| 区别 | 抽象类 | 接口 |
| 定义 | 包含抽象方法的类 | 主要是抽象方法和全局常量的集合 |
| 组成 | 构造器、代码块、抽象方法（可以由public、protected、default、private修饰）、静态方法、普通方法、常量、变量 | 常量、抽象方法（只能由public修饰）、JDK8之后有默认方法和静态方法 |
| 使用 | 子类继承且单继承抽象类(extends) | 类实现且多实现接口(implements) |
| 关系 | 抽象类可以实现多个接口 | 接口不能继承抽象类，但允许继承多个接口 |
| 对象 | 都通过对象的多态性产生实例化对象 | |
| 局限 | 抽象类有单继承的局限 | 接口没有此局限 |
| 实际 | 作为一个模板 | 是作为一个标准或是表示一种能力 |
| 选择 | 如果抽象类和接口都可以使用的话，优先使用接口，因为避免单继承的局限 | |

接口更多的是在系统架构设计方法发挥作用， 接口体现的是一种规范。 而抽象类在代码实现方面发挥作用，可以实现代码的重用，例如，模板模式是抽象类的一个典型应用，假设项目中需要使用大量的 DAO 组件，这些 DAO 组件通常都具有增、删、改、查等基本方法，因此我们就可以定义一个抽象的 DAO 基类，然后让其他 DAO 组件来继承这个 DAO 基类，把这个 DAO 基类当成模板使用。

## 对象

### 创建对象方式

使用new关键字

使用反射

使用clone方法

使用序列化机制

**注意**：前2者都需要显式的调用构造方法，造成耦合性最高的是第一种，因此无论是什么框架，涉及到解耦必然会先减少new的使用。

### 创建对象内部顺序

父类中的静态成员和静态初始化块，按照在代码中出现的顺序依次执行。

子类静态成员和静态初始化代码块，按照在代码中出现的顺序依次执行。

父类实例成员和实例初始化块，按照在代码中出现的顺序依次执行。(创建每一个实例对象的时候都执行)

执行父类的构造方法。(创建每一个实例对象的时候都执行)

子类实例成员和实例初始化块，按照在代码中出现的顺序依次执行。(创建每一个实例对象的时候都执行)

执行子类构造方法。(创建每一个实例对象的时候都执行)

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //a：第一次创建对象，按照上述顺序来执行  //a: 20)至此创建新对象所有的语句全部执行完毕  **new** Man();  System.***out***.println("--------------------------------------");  //b：第二次创建对象，因为子类和父类中的静态成员和静态代码块都已经执行过一次，因此不再执行  //然后开始直接执行父类的实例成员、实例初始化块和构造器，最后执行本类的实例成员、实例初始化块和构造器  //b: 5)至此第二次创建对象的语句全部执行完毕  **new** Man();  System.***out***.println("--------------------------------------");  //c: 此时因为Person类已经加载完成，因此static修饰的内容不再执行，直接执行实例成员、实例初始化块和构造器  //c: 3)至此创建对象的语句全部执行完毕  **new** Person();  }  }  **class** Person{  //a：1)按顺序执行父类的静态成员和静态初始化块，因此执行到这一句  //a: 2)本句静态类变量初始化创建了父类对象，因为父类的父类为Object，所以没有语句可以执行  //a: 3)然后执行本类中的实例成员和实例初始化块，然后执行构造器，而不是执行静态方法，  //a: 6)执行完类变量的初始化，开始执行下一个静态成员，即静态初始化块  **static** Person *p* = **new** Person();  String name = "person";  {  //a: 4)  //a: 11)  //a: 16)  //b: 1)  //c: 1)  System.***out***.println("five");  }  **static** {  //a: 7)执行完这一句就已经执行完了所有的父类static修饰的内容，开始执行本类static修饰的内容  System.***out***.println("one");  }  Person(){  //a: 5)  //a: 12)执行完这一句，开始执行本类的实例成员、实例初始化块和构造器  //a: 17)  //b: 2)  //c: 2)  System.***out***.println("three");  }  }  **class** Man **extends** Person{  String name = "man";  {  //a: 13)  //a: 18)  //b: 3)  System.***out***.println("six");  }  **static** {  //a: 8)因为本句是顺序在前的，因此优先执行，然后执行下一个static修饰的内容  System.***out***.println("two");  }  //a: 9)本句因为类变量创建了新的对象，因此先执行父类中的static修饰的内容，再执行本类中static修饰的内容  //a: 10)因为父类和本类中的static修饰的内容都已经执行过一次，所以下一步执行父类的实例成员、实例初始化块和构造器  //a: 15)执行完类变量的初始化，父类和子类中所有static修饰的内容都已经执行完，开始执行父类的实例成员、实例初始化块和构造器  **static** Man *m* = **new** Man();  Man(){  //a: 14)  //a: 19)  //b: 4)  System.***out***.println("four");  }  } |
| five  three  one  two  five  three  six  four  five  three  six  four  --------------------------------------  five  three  six  four  --------------------------------------  five  three |

### 匿名对象

在创建对象后并不把对象的地址保存在引用变量中, 而是直接使用创建好的对象的引用访问成员。适用于对象的一次性使用和作为实际参数传递的场景中。

**特点**：因为对象没有使用引用变量保存, 所以对象访问完成后, 就无法再次访问了。

### 对象数组

该数组的类型是引用数据类型，该数组中存放的元素是该类型对象的地址值。

动态方式：Object[] obj = new Object[10];

静态方式：Object[] obj = new Object[]{引用1，引用2，引用3...}([]内不允许出现数组的长度)

|  |
| --- |
| **public** **class** ObjArrayPer {  **public** **static** **void** main(String[] args) {  //对象数组的类型是引用数据类型，该数组中存放的元素是该类型对象的地址值  Person[] p = **new** Person[2];  p[0] = **new** Person("Bob",18);  p[1] = **new** Person().setInfo("Tom", 19);  **for**(Person m :p) {  System.***out***.println(m.getName()+" "+m.getAge());  }  }  }  **class** Person{  **private** String name;  **private** **int** age;  //保留空构造方法，以便创建对象时不用赋值  Person(){  }  //可以用两种方法将参数赋值到person类的属性，构造方法和普通方法  Person(String name,**int** age){  **this**.name = name;  **this**.age = age;  }  Person setInfo(String name,**int** age) {  **this**.name = name;  **this**.age = age;  **return** **this**;  }  String getName() {  **return** name;  }  **int** getAge() {  **return** age;  }  } |
| Bob 18  Tom 19 |

### 对象的传递

为什么要传递对象：在有些方法中，功能的完成除了需要一些基本数据外，还需要复杂的引用数据。

对象如何传递：在方法调用时，由调用者负责将对象作为实参传递给方法。

传递的是什么：方法执行时，接受的是对象的引用或者匿名对象，然后通过引用使用了对象。

对象传递的优点：即使对象很大，属性很多，在对象传递时，并不是把对象本身复制了一份，而是只把对象的引用传递了，因此速度很快。

对象传递的危险性：如果通过引用直接修改对象的属性，也会导致对象属性的变化。

### 对象的关联

简单地说，是指一个对象中使用了另一个对象，在当前类中把要关联的对象作为属性即可。

|  |
| --- |
| **public** **class** TeacherTest {  **public** **static** **void** main(String[] args) {  //创建一个老师的对象  Teacher teacher = **new** Teacher();  Teacher teacher2 = **new** Teacher();  teacher.name = "苍老师";  //创建一台电脑  Computer computer = **new** Computer();  computer.color = "red";  computer.type = "联想";  //让老师的对象和电脑的对象产生关联  teacher.computer = computer;  teacher2.computer = computer;  //修改苍老电脑的颜色为yellow  teacher.computer.color = "yellow";  //输出苍老师的所有详细信息  //1.苍老师的个人信息  System.***out***.println(teacher.name);  //2.苍老师的电脑的信息  System.***out***.println(teacher.computer.type);  System.***out***.println(teacher.computer.color);  //因为teacher2和teacher的computer指向的是同一块内存地址，因此teacher的computer属性改变会影响到teacher2的computer属性  System.***out***.println(teacher2.computer.color);  }  }  **class** Teacher{  String name;  Computer computer;  }  **class** Computer{  String color;  String type;  } |

### 对象类型转换

基本数据类型的Casting：

自动类型转换：小的数据类型可以自动转换成大的数据类型

如long g=20; double d=12.0f

强制类型转换：可以把大的数据类型强制转换(casting)成小的数据类型

如 float f=(float)12.0; int a=(int)1200L

对Java对象的强制类型转换称为造型

从子类到父类的类型可以自动进行

从父类到子类的类型转换必须通过造型(强制类型转换)实现

无继承关系的引用类型间的转换是非法的

在造型前可以使用instanceof操作符测试一个对象的类型

### 类的加载流程

**Person p = new Person();**

**创建一个对象都在内存中做了什么事情？**

1：先将硬盘上指定位置的Person.class文件加载进内存。

2：执行main方法时，在栈内存中开辟了main方法的空间(压栈-进栈)，然后在main方法的栈区分配了一个变量p。

3：在堆内存中开辟一个实体空间，分配了一个内存首地址值。new

4：在该实体空间中进行属性的空间分配，并进行了默认初始化。

5：对空间中的属性进行显示初始化。

6：进行实体的构造代码块初始化。

7：调用该实体对应的构造函数，进行构造函数初始化。（）

8：将首地址赋值给p ，p变量就引用了该实体。(指向了该对象)

Java中对象的声明、创建、赋值、调用在内存中的实现。

JVM通过加载字节码文件获取类的结构信息，在堆中分配内存空间，对属性进行初始化，先默认初始化，再显式初始化，最后使用构造器初始化，最后将堆中对象的地址返回给其引用。

属性在堆中，方法信息在方法区中，调用方法时，会从方法区中的方法信息生成栈帧，并存入栈中，其中有方法的具体信息，方法循环就会抛出栈溢出错误。

一个方法执行完，会将栈中的信息弹出销毁，接着进入第二个栈帧，执行第二个方法。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //对象的声明，即先在栈中创建对象的引用  Person p;  //对象的创建（实例化），创建好的对象存放在堆中，new出来的对象都存放在堆中  p = **new** Person();  //对堆中对象的属性赋值  p.name = "xmm";  p.age = 18;  //调用堆中对象的方法  p.say();  //一个类创建的多个对象，每个对象各自拥有一份属性，每个对象的属性修改不会影响其他对象。  Person p2 = **new** Person();  p2.name = "xsq";  p2.age = 30;  p2.say();  //同一个对象可以有多个引用，不同引用对对象的属性改变会影响到所有该对象的引用。  Person p3 = p;  p3.age = 100;  p.say();  p3.say();  //匿名对象：即没有引用的对象，但是可以直接调用方法  **new** Person().say();  }  }  **class** Person{  String name;  **int** age;  **protected** **void** say() {  System.***out***.println("my name is "+name+",and I'm "+age+" years old.");  }  } |
| my name is xmm,and I'm 18 years old.  my name is xsq,and I'm 30 years old.  my name is xmm,and I'm 100 years old.  my name is xmm,and I'm 100 years old.  my name is null,and I'm 0 years old. |

### JavaBean

拥有三个特点的Java类，就是JavaBean，其对象用于存储数据。

三个特点：

1、类是公共的

2、有一个无参的公共构造器

3、有属性，且有对应的get，set方法

用户可以使用JavaBean将功能、处理、值、数据库访问和其他任何可以用java代码创造的对象进行打包，并且其他的开发者可以通过内部的JSP页面、Servlet、其他JavaBean、applet程序或者应用来使用这些对象。用户可以认为JavaBean提供了一种随时随地的复制和粘贴的功能，而不用关心任何改变。