Java 基础知识结构

# 环境配置

JDK安装

EditTxt安装

eclipse安装

测试HelloWorld

# Java基础

## 数据类型

Java 是一种强类型语言，这意味着必须为每一个变量声明一种类型。在Java中一共有8种基本类型（primitive type），其中4种整型，2种浮点类型，1种用于表示Unicode编码的字符单元的字符类型char 和 一种表示真值的boolean类型。

### 基础知识

#### 字节 & 位

HZ：所谓字节，原意就是用来表示一个完整的字符的。最初的计算机性能和存储容量都比较差，所以普遍采用4位BCD编码（这个编码出现比计算机还早,最早是用在打孔卡上的）。BCD编码表示数字还可以，但表示字母或符号就很不好用，需要用多个编码来表示。后来又演变出6位的BCD编码（BCDIC），以及至今仍在广泛使用的7位ASCII编码。不过最终决定字节大小的，是大名鼎鼎的System/360。当时IBM为System/360设计了一套8位EBCDIC编码，涵盖了数字、大小写字母和大部分常用符号，同时又兼容广泛用于打孔卡的6位BCDIC编码。System/360很成功，也奠定了字符存储单位采用8位长度的基础，这就是1字节=8位的由来。

——http://zhidao.baidu.com/question/553289126853210892.html

#### 编码

在显示器上看见的文字、图片等信息在电脑里面其实并不是我们看见的样子，即使你知道所有信息都存储在硬盘里，把它拆开也看不见里面有任何东西，只有些盘片。假设，你用显微镜把盘片放大，会看见盘片表面凹凸不平，凸起的地方被磁化，凹的地方是没有被磁化；凸起的地方代表数字1，凹的地方代表数字0。硬盘只能用0和1来表示所有文字、图片等信息。

那么字母”A”在硬盘上是如何存储的呢？可能小张计算机存储字母”A”是1100001，而小王存储字母”A”是11000010，这样双方交换信息时就会误解。比如小张把1100001发送给小王，小王并不认为1100001是字母”A”，可能认为这是字母”X”，于是小王在用记事本访问存储在硬盘上的1100001时，在屏幕上显示的就是字母”X”。也就是说，小张和小王使用了不同的编码表。小张用的编码表是ASCII，ASCII编码表把26个字母都一一的对应到2进制1和0上；小王用的编码表可能是EBCDIC,只不过EBCDIC编码与ASCII编码中的字母和01的对应关系不同。一般地说，开放的操作系统（LINUX 、WINDOWS等）采用ASCII 编码，而大型主机系统（MVS 、OS/390等）采用EBCDIC 编码。在发送数据给对方前，需要事先告知对方自己所使用的编码，或者通过转码，使不同编码方案的两个系统可沟通自如。

ASCII码使用7位2进制数表示一个字符，7位2进制数可以表示出2的7次方个字符，共128个字符。EBCDIC码使用8位，可以表示出2的8次方个字符，256个字符。

无论是ASCII码还是EBCDIC码，都无法对拥有几万个的汉字进行编码。因为上面已经提过，7位2进制数最多对应上128个字符，8位最多对应上256个字符。

##### ASCII

美国（国家）信息交换标准（代）码，一种使用7个或8个二进制位进行编码的方案，最多可以给256个字符（包括字母、数字、标点符号、控制字符及其他符号）分配（或指定）数值。ASCII码于1961年提出，用于在不同计算机硬件和软件系统中实现数据传输标准化，在大多数的小型机和全部的个人计算机都使用此码。

目前使用最广泛的西文字符集及其编码是 ASCII 字符集和 ASCII 码（ ASCII 是 American Standard Code for Information Interchange 的缩写），它同时也被国际标准化组织（ International Organization for Standardization, ISO ）批准为国际标准。

ASCII码划分为两个集合：128个字符的标准ASCII码和附加的128个字符的扩充和ASCII码。比较EBCDIC。其中95个字符可以显示。另外33个不可以显示。 标准ASCII码为7位，扩充为8位。

虽然标准 ASCII 码是 7 位编码，但由于计算机基本处理单位为字节（ 1byte = 8bit ），所以一般仍以一个字节来存放一个 ASCII 字符。每一个字节中多余出来的一位（最高位）在计算机内部通常保持为 0 （在数据传输时可用作奇偶校验位）。

由于标准 ASCII 字符集字符数目有限，在实际应用中往往无法满足要求。为此，国际标准化组织又制定了 ISO2022 标准，它规定了在保持与 ISO646 兼容的前提下将 ASCII 字符集扩充为 8 位代码的统一方法。 ISO 陆续制定了一批适用于不同地区的扩充 ASCII 字符集，每种扩充 ASCII 字符集分别可以扩充 128 个字符，这些扩充字符的编码均为高位为 1 的 8 位代码（即十进制数 128~255 ），称为扩展 ASCII 码。

##### ANSI / MBCS

为了扩充ASCII编码，以用于显示本国的语言，不同的国家和地区制定了不同的标准，由此产生了 GB2312, BIG5, JIS 等各自的编码标准。这些使用 2 个字节来代表一个字符的各种汉字延伸编码方式，称为 ANSI 编码，又称为"MBCS（Muilti-Bytes Charecter Set，多字节字符集）"。在简体中文系统下，ANSI 编码代表 GB2312 编码，在日文操作系统下，ANSI 编码代表 JIS 编码，所以在中文 windows下要转码成gb2312,gbk只需要把文本保存为ANSI 编码即可。 不同 ANSI 编码之间互不兼容，当信息在国际间交流时，无法将属于两种语言的文字，存储在同一段 ANSI 编码的文本中。一个很大的缺点是，同一个编码值，在不同的编码体系里代表着不同的字。这样就容易造成混乱。导致了unicode码的诞生。

其中每个语言下的ANSI编码，都有一套一对一的编码转换器，Unicode变成所有编码转换的中间介质。所有的编码都有一个转换器可以转换到Unicode，而Unicode也可以转换到其他所有的编码。

###### GB2312

GB2312 也是ANSI编码里的一种，对ANSI编码最初始的ASCII编码进行扩充，为了满足国内在计算机中使用汉字的需要，中国国家标准总局发布了一系列的汉字字符集国家标准编码，统称为GB码，或国标码。其中最有影响的是于1980年发布的《信息交换用汉字编码字符集 基本集》，标准号为GB 2312-1980,因其使用非常普遍，也常被通称为国标码。GB2312编码通行于我国内地；新加坡等地也采用此编码。几乎所有的中文系统和国际化的软件都支持GB 2312。

GB 2312是一个简体中文字符集，由6763个常用汉字和682个全角的非汉字字符组成。其中汉字根据使用的频率分为两级。一级汉字3755个，二级汉字3008个。由于字符数量比较大，GB2312采用了二维矩阵编码法对所有字符进行编码。首先构造一个94行94列的方阵，对每一行称为一个“区”，每一列称为一个“位”，然后将所有字符依照下表的规律填写到方阵中。这样所有的字符在方阵中都有一个唯一的位置，这个位置可以用区号、位号合成表示，称为字符的区位码。如第一个汉字“啊”出现在第16区的第1位上，其区位码为1601。因为区位码同字符的位置是完全对应的，因此区位码同字符之间也是一一对应的。这样所有的字符都可通过其区位码转换为数字编码信息。GB2312字符的排列分布情况见下表。

**GB2312 字符编码分布表**

|  |  |
| --- | --- |
| 分区范围 | 符号类型 |
| 第01区 | 中文标点、数学符号以及一些特殊字符 |
| 第02区 | 各种各样的数学序号 |
| 第03区 | 全角西文字符 |
| 第04区 | 日文平假名 |
| 第05区 | 日文片假名 |
| 第06区 | 希腊字母表 |
| 第07区 | 俄文字母表 |
| 第08区 | 中文拼音字母表 |
| 第09区 | 制表符号 |
| 第10－15区 | 无字符 |
| 第16－55区 | 一级汉字（以拼音字母排序） |
| 第56－87区 | 二级汉字（以部首笔画排序） |
| 第88－94区 | 无字符 |

GB2312字符在计算机中存储是以其区位码为基础的，其中汉字的区码和位码分别占一个存储单元，每个汉字占两个存储单元。由于区码和位码的取值范围都是在1－94之间，这样的范围同西文的存储表示冲突。例如汉字‘珀’在GB2312中的区位码为7174，其两字节表示形式为71，74；而两个西文字符‘GJ’的存储码也是71,74。这种冲突将导致在解释编码时到底表示的是一个汉字还是两个西文字符将无法判断。

为避免同西文的存储发生冲突，GB2312字符在进行存储时，通过将原来的每个字节第8bit设置为1同西文加以区别，如果第8bit为0，则表示西文字符，否则表示GB2312中的字符。实际存储时，采用了将区位码的每个字节分别加上A0H（160）的方法转换为存储码，计算机存储规则是此编码的补码，而且是位码在前，区码在后。例如汉字‘啊’的区位码为1601，其存储码为B0A1H，其转换过程为：

|  |  |  |  |
| --- | --- | --- | --- |
| 区位码 | 区码转换 | 位码转换 | 存储码 |
| 1001H | 10H+A0H=B0H | 01H+A0H=A1H | B0A1H |

GB2312编码用两个字节(8位2进制)表示一个汉字，所以理论上最多可以表示256×256=65536个汉字。但这种编码方式也仅仅在中国行得通，如果您的网页使用的GB2312编码，那么很多外国人在浏览你的网页时就可能无法正常显示，因为其浏览器不支持GB2312编码。当然，中国人在浏览外国网页(比如日文)时，也会出现乱码或无法打开的情况，因为我们的浏览器没有安装日文的编码表。

###### GBK

GBK即汉字内码扩展规范，K为扩展的汉语拼音中“扩”字的声母。英文全称Chinese Internal Code Specification。GBK编码标准兼容GB2312，共收录汉字21003个、符号883个，并提供1894个造字码位，简、繁体字融于一库。GB2312码是中华人民共和国国家汉字信息交换用编码，全称《信息交换用汉字编码字符集——基本集》，1980年由国家标准总局发布。基本集共收入汉字6763个和非汉字图形字符682个，通行于中国大陆。新加坡等地也使用此编码。GBK是对GB2312-80的扩展，也就是CP936字码表 (Code Page 936)的扩展（之前CP936和GB 2312-80一模一样）。

基本简介

GB 2312的出现，基本满足了汉字的计算机处理需要，但对于人名、古汉语等方面出现的罕用字，GB 2312不能处理，这导致了后来GBK及GB 18030汉字字符集的出现。

GBK采用双字节表示，总体编码范围为8140-FEFE，首字节在81-FE 之间，尾字节在40-FE 之间，剔除 xx7F一条线。总计23940 个码位，共收入21886个汉字和图形符号，其中汉字（包括部首和构件）21003 个，图形符号883 个。P-Windows3.2和苹果OS以GB2312为基本汉字编码， Windows 95/98则以GBK为基本汉字编码。

有些汉字用五笔和拼音都打不出来，如：溙（五笔IDWI），须调出GBK字符集才能打出这个字。极品五笔中可右击输入法图标，设置，属性中选GBK字符集。极点五笔中可点击工具条中相关图标进行转换。

编码方式

字符有一字节和双字节编码，00–7F范围内是一位，和ASCII保持一致，此范围内严格上说有96个字符和32个控制符号。

之后的双字节中，前一字节是双字节的第一位。总体上说第一字节的范围是81–FE（也就是不含80和FF），第二字节的一部分领域在40–7E，其他领域在80–FE。

###### Big5

在台湾、香港与澳门地区，使用的是繁体中文字符集。而1980年发布的GB2312面向简体中文字符集，并不支持繁体汉字。在这些使用繁体中文字符集的地区，一度出现过很多不同厂商提出的字符集编码，这些编码彼此互不兼容，造成了信息交流的困难。为统一繁体字符集编码，1984年，台湾五大厂商宏碁、神通、佳佳、零壹以及大众一同制定了一种繁体中文编码方案，因其来源被称为五大码，英文写作Big5，后来按英文翻译回汉字后，普遍被称为大五码。

大五码是一种繁体中文汉字字符集，其中繁体汉字13053个，808个标点符号、希腊字母及特殊符号。大五码的编码码表直接针对存储而设计，每个字符统一使用两个字节存储表示。第1字节范围81H－FEH，避开了同ASCII码的冲突，第2字节范围是40H－7EH和A1H－FEH。因为Big5的字符编码范围同GB2312字符的存储码范围存在冲突，所以在同一正文不能对两种字符集的字符同时支持。

Big5编码的分布如表1－5所示，Big5字符主要部分集中在三个段内：标点符号、希腊字母及特殊符号；常用汉字；非常用汉字。其余部分保留给其他厂商支持。

|  |  |
| --- | --- |
| 编码范围 | 符号类别 |
| 8140H－A0FEH | 保留（用作造字区） |
| A140H－A3BFH | 标点符号、希腊字母及特殊符号 |
| A3C0H－A3FEH | 保留（未开放用于造字区） |
| A440H－C67EH | 常用汉字（先按笔划，再按部首排序） |
| C6A1H－C8FEH | 保留（用作造字区） |
| C940H－F9D5H | 非常用汉字（先按笔划，再按部首排序） |
| F9D6H－FEFEH | 保留（用作造字区） |

Big5编码推出后，得到了繁体中文软件厂商的广泛支持，在使用繁体汉字的地区迅速普及使用。目前，Big5编码在台湾、香港、澳门及其他海外华人中普遍使用，成为了繁体中文编码的事实标准。在互联网中检索繁体中文网站，所打开的网页中，大多都是通过Big5编码产生的文档。

##### Unicode

###### Unicode编码

如上ANSI编码条例中所述，世界上存在着多种编码方式，在ANSi编码下，同一个编码值，在不同的编码体系里代表着不同的字。在简体中文系统下，ANSI 编码代表 GB2312 编码，在日文操作系统下，ANSI 编码代表 JIS 编码，可能最终显示的是中文，也可能显示的是日文。在ANSI编码体系下，要想打开一个文本文件，不但要知道它的编码方式，还要安装有对应编码表，否则就可能无法读取或出现乱码。为什么电子邮件和网页都经常会出现乱码，就是因为信息的提供者可能是日文的ANSI编码体系和信息的读取者可能是中文的编码体系，他们对同一个二进制编码值进行显示，采用了不同的编码，导致乱码。这个问题促使了unicode码的诞生。

如果有一种编码，将世界上所有的符号都纳入其中，无论是英文、日文、还是中文等，大家都使用这个编码表，就不会出现编码不匹配现象。每个符号对应一个唯一的编码，乱码问题就不存在了。这就是Unicode编码。

Unicode当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，“汉”这个字的Unicode编码是U+6C49。

Unicode固然统一了编码方式，但是它的效率不高，比如UCS-4(Unicode的标准之一)规定用4个字节存储一个符号，那么每个英文字母前都必然有三个字节是0，这对存储和传输来说都很耗资源。

###### UTF-8

为了提高Unicode的编码效率，于是就出现了UTF-8编码。UTF-8可以根据不同的符号自动选择编码的长短。比如英文字母可以只用1个字节就够了。

UTF-8的编码是这样得出来的，以”汉”这个字为例：“汉”字的Unicode编码是U+00006C49，然后把U+00006C49通过UTF-8编码器进行编码，最后输出的UTF-8编码是E6B189。

##### Base64

有的电子邮件系统(比如国外信箱)不支持非英文字母(比如汉字)传输，这是历史原因造成的(认为只有美国会使用电子邮件?)。因为一个英文字母使用ASCII编码来存储，占存储器的1个字节(8位)，实际上只用了7位2进制来存储，第一位并没有使用，设置为0，所以，这样的系统认为凡是第一位是1的字节都是错误的。而有的编码方案(比如GB2312)不但使用多个字节编码一个字符，并且第一位经常是1，于是邮件系统就把1换成0，这样收到邮件的人就会发现邮件乱码。

为了能让邮件系统正常的收发信件，就需要把由其他编码存储的符号转换成ASCII码来传输。比如，在一端发送GB2312编码－>根据Base64规则－>转换成ASCII码，接收端收到ASCII码－>根据Base64规则－>还原到GB2312编码。。

### 整型类型

整型用于表示没有小数部分的数值，它允许是负数。通常情况下，int类型最常用。但是如果表示星球上的居住人数，就需要使用long类型了。byte和short类型主要用于特定的应用场合，例如，底层的文件处理或者需要控制占用存储空间量的大数组。

#### 整型数据分类

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **存储需求** | **取值范围** | **应用举例** |
| **byte** | **1**字节 | -128 ~ 127 | 1百 |
| **short** | **2**字节 | -32768 ~ 32767 | 3万 |
| **int** | **4**字节 | -2147 483 648 ~ 2147 483 647 | 20亿 |
| **long** | **8**字节 | -9223 372 036 854 775 808 ~ 9223 372 036 854 775 807 |  |

##### byte

HZ：byte类型主要用于操作文件时的byte数组，因为这是最小的字节单位，用它作为数组可以用最小的粒度单位进行数据的拷贝传输，节省资源。

##### short

##### int

在C和C++中，int表示的整型与目标机器相关。在8086这样的16位处理器上整型数值占2字节；在Sun SPARC这样的32位处理器上，整型数值占4字节。而在Java中，所有的数值类型所占据的字节数量与平台无关。注意，Java没有任何无符号类型（unsigned type）。

##### long

#### 整型数据特性

### 浮点类型

#### 分类

浮点数有两种数据类型：Float、Double

float类型的数值有一个后缀F（例如， 3.402F）。没有后缀F的浮点数值（如3.402）默认为double类型。当然，也可以在浮点数值后面添加后缀D（例如， 3.402D）。

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **存储需求** | **取值范围** | **应用举例** |
| **Float** | **4**字节 | 大约±3.4028235E38（有效位数6~7位） |  |
| **Double** | **8**字节 | ±1.7976931348623157E308（有效位数15位） |  |

double表示这种类型的数值精度是float类型的两倍（有人称之为双精度）。绝大部分应用程序都采用double类型。在很多情况下，float类型的精度很难满足需求。例如，用7位有效数字足以表示普通雇员的年薪，但表示公司总裁的年薪可能就不够用了。实际上，只有很少的情况适合使用float类型，例如，需要快速地处理单精度数据，或者需要存储大量数据。，只有很少的情况适合使用float类型，例如，需要快速地处理单精度数据，或者需要存储大量数据。

##### float

##### double

#### 浮点数特性

##### 特殊的浮点数值

所有的浮点数值计算都遵循IEEE 754规范。下面是用于表示溢出和出错情况的三个特殊的浮点数值：

• 正无穷大

• 负无穷大

• NaN（不是一个数字）

例如，一个正整数除以0的结果为正无穷大。计算0/0或者负数的平方根结果为NaN。

##### NaN（非数字浮点数）的判断方法

检测一个值是否等于Nan，不能使用这种方法：

**if** (x == Double.*NaN*) // is never true

所有的“非数值”的值都认为是不相同的。然而，可以使用Dobule.isNaN() 或者Float.isNaN()

**if** (Double.*isNaN*(x)) // check wether x is a “not a number”

##### 浮点数的缺点

浮点数值不适用于禁止出现舍入误差的金融计算中。例如，命令System.*out*.println(2.0-1.1)将打印出0.8999999999999999，而不是0.9。主要原因是浮点数值采用二进制系统表示，在二进制系统中无法精确地表示分数1/10。这就好像十进制无法精确地表示1/3一样。如果需要在数值计算中不含有任何舍入误差，就应该使用BigDecimal类。

HZ问题：其他语言有没有这种问题？比如js，php？

### char类型

char类型用来表示单个字符。通常用来表示字符常量。例如：‘A’（单引号）是编码为65所对应的字节常量；“A”（双引号）是一个包含字符A的字符串。

要想弄清char类型，就必须了解Unicode编码表。参考[编码](#_编码)。

在Java中，char类型使用UTF-16表示一个代码单元。所以不管是英文还是汉字，或者别的什么语言，都是占用了两个字节。

### boolean类型

boolean（布尔）类型有两个值：false和true，用来判断逻辑条件。整型值和布尔值之间不能进行相互转换。

HZ问题：

整型值和布尔值之间不能进行相互转换？整型1不是可以表示true吗？

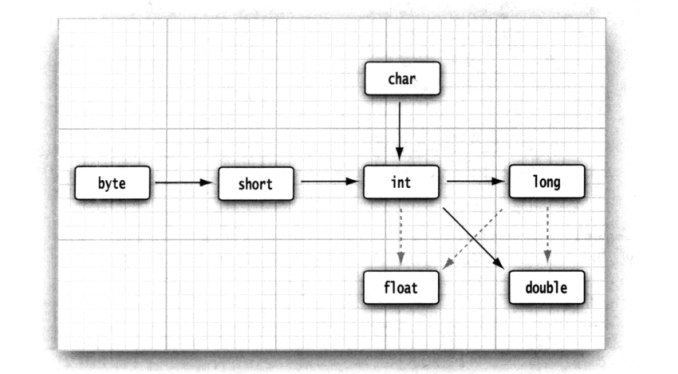
那String属于什么类型？

Java中的类属于什么类型？

各种数据类型在机器中是如何存储的？比如，byte一个字节是8位，整型都有确切的位数，那浮点数、char类型以及boolean类型是如何存储的？

### 数据类型转换

#### 自动类型转换



实心箭头，表示无信息丢失的转换；虚箭头，表示可能有精度损失的转换。

当使用上面两个数值进行二元操作时（例如n + f， n是整数， f是浮点数），先要将两个操作  
数转换为同一种类型，然后再进行计算。  
• 如果两个操作数中有一个是double类型的，另一个操作数就会转换为double类型。  
• 否则，如果其中一个操作数是float类型，另一个操作数将会转换为float类型。  
• 否则，如果其中一个操作数是long类型，另一个操作数将会转换为long类型。  
• 否则，两个操作数都将被转换为int类型。

#### 强制类型转换

在上一小节中看到，在必要的时候， int类型的值将会自动地转换为double类型。但另一方面，有时也需要将double转换成int。在Java中，允许进行这种数值之间的类型转换。当然，有可能会丢失一些信息。在这种情况下，需要通过强制类型转换（ cast）实现这个操作。强制类型转换的语法格式是在圆括号中给出想要转换的目标类型，后面紧跟待转换的变量名。例如：

**double** x = 9.97;

**int** nx = (**int**)x;

### 字符串

Java字符串由char序列组成。从概念上讲， Java字符串就是Unicode字符序列。例如字符串“Java\u2122”由5个Unicode字符J、 a、 v、 a和TM （可以进行如下测试结果来验证System.*out*.println("Java\u2122");）。Java没有内置的字符串类型，而是在标准Java类库中提供了一个预定义类String。每个用双引号括起来的字符串都是String类的一个实例：

String e = "";

String greating = "Hello";

关于字符串更多操作方法参考[String & StringBuffer类](#_String_&_StringBuffter类)。

## 变量

### 变量

#### 声明

在Java中，每一个变量属于一种类型（type）。在声明变量时，变量所属的类型位于变量名之前。

可以在一行中声明多个变量：int i,j; // both are integers

不过不提倡使用这种风格，逐一声明每个变量可以提高程序的可读性。

#### 初始化

使用“=”号赋值进行变量的初始化：

**int** vacationDays;

vacationDays = 12;

也可以将变量的声明和初始化放在同一行：

**int** vacationDays = 12;

### 常量

在Java中，使用final声明常量，例如：

**final** **double** CM\_PER\_INCH = 2.54;// HZ：此常量是在方法中定义的

关键字final表示这个变量只能被赋值一次，一旦被赋值之后，就不能再更改了。习惯上，常量名全部字母都使用大写。

在Java中，经常希望某个变量可以在一个类中的多个方法中使用，通常将这些常量称为类常量。可以使用关键字static final设置一个类常量。下面是使用类常量的示例：

**public** **static** **final** **double** *CM\_PER\_INCH* = 2.54;//HZ：此常量是在类中定义的

需要注意，类常量的定义位于main方法的外部。因此，在同一个类的其他方法中也可以使用这个常量。而且，如果一个常量被声明为public，那么其他类的方法也可以使用这个常量。在这个示例中， Constants2.CM\_PER-INCH就是这样一个常量。

## 运算符

### 加减乘除

### 自增运算符 & 自减运算符

i++

++i

### 关系运算符 & boolean运算符

### 位运算符

与（&）、非（~）、或（|）、异或（^）：

&：当两边操作数的位同时为1时，结果为1，否则为0。如1100 & 1010=1000

| ：当两边操作数的位有一边为1时，结果为1，否则为0。如1100 | 1010=1110

~：0变1,1变0

^：两边的位不同时，结果为1，否则为0.如1100 ^ 1010=0110

### 数值类型之间的转换

#### 自动转换

#### 强制类型转换

### 括号 与 运算符级别

## 控制流程

### 块作用域

块（block，即复合语句）是指由一对花括号括起来的若干条简单的Java语句。块确定了变量的作用域。一个块可以嵌套在另一个块中，但是，不能在嵌套的两个块中声明同名的变量。

### 条件语句

if () {} else if () {} else {}

### 循环

当条件为true时， while循环执行一条语句（也可以是一个语句块）。while循环语句首先检测循环条件。因此，循环体中的代码有可能不被执行。如果希望循环体至少执行一次，则应该将检测条件放在最后。使用do/while循环语句可以实现这种操作方式。

// while ……

**int** i = 1;

**while** (i < 10) {

System.*out*.println("I ->" + i++);

}

// do …… while ……

**int** j = 1;

**do** {

System.*out*.println("J ->" + j++);

} **while** (j < 10);

### 确定循环

for循环语句是支持迭代的一种通用结构，利用每次迭代之后更新的计数器或类似的变量来控制迭代次数。

for循环语句只不过是while循环的一种简化形式。例如，

**for** (**int** i = 10; i > 0; i--) {

System.*out*.println("Counting down..." + i);

}

可以重写为：

**int** i = 10;

**while** (i > 0) {

System.*out*.println("Counting down..." + i);

i--;

}

### 多重选择：switch语句

一般范例：

String charr = "X";

**switch** (charr) {

**case** "A":

System.*out*.println("A");

**break**;

**case** "B":

System.*out*.println("B");

**break**;

**default**:

System.*out*.println("default");

}

### 中断控制流程语句

#### break

#### continue

continue语句将控制转移到最内层循环的首部。

#### lable

lable的使用方法，标签名称 + 冒号 + 循环语句：

lable: **for** (**int** i = 0; i < 10; i++) {

**if** (i == 5)

**break** lable;

System.*out*.println("i -> " + i);

}

**int** i = 0;

lable: **while** (i < 10) {

}

lable: **do** {

} **while** (i < 20);

String[] strs = { "A", "B" };

lable: **for** (String str : strs) {

}

## 大数值

### BigInteger

BigInteger类实现了任意精度的整数运算

### BigDecimal

BigDecimal实现了任意精度的浮点数运算

## 数组

数组是一种数据结构，用来存储同一类型值的集合。一旦创建了数组，就不能再改变它的大小（尽管可以改变每一个数组元素）。如果经常需要在运行过程中扩展数组的大小，就应该使用另一种数据结构—数组列表（ array list）。

### 数组初始化以及匿名数组

// 声明一个10容量的数组

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

// 遍历赋值

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

ints[i] = i;

}

// 声明时赋值：

String[] strA = { "A", "B", "C" };

// 匿名数组：

strA = **new** String[] { "E", "F" };

// 等效于下面语句：

String[] strB = { "E", "F" };

strA = strB;

### For each循环

### 数组拷贝

使用Arrays.*copyOf*进行数组的拷贝，这个方法通常用来增加数组的大小：

Arrays.*copyOf*(strA, 2 \* strA.length);

### 数组排序

### 多维数组

### 不规则数组

## 输入输出

### 读取输入

打印输出到“标准输出流”（即控制台窗口）是一件非常容易的事情，只  
要调用System.out.println即可。然而，读取“标准输入流” System.in就没有那么简单了。要想通过控制台进行输入，首先需要构造一个Scanner对象，并与“标准输入流” System.in关联。

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

System.*out*.println("name:");

String name = in.nextLine();

System.*out*.println("age:");

String age = in.nextLine();

System.*out*.println("name:" + name + ",age:" + age);

in.close();

### 格式化输出

### 文件输入与输出

## 注释

### 注释的三种方法

单行注释：//

多行注释：

/\* \*/ —— 将需要注释的内容放在中间

/\*\* \*/ —— 这种注释和上面的区别是可以自动生成文档

### 类注释

### 方法注释

### 域注释

### 通用注释

### 包与概述注释

### 注释的抽取

# 类

## 基本概念

### 实例域

#### 对象与对象变量

要想使用对象，就必须首先构造对象，并指定其初始状态。然后，对对象施加方法。在Java程序设计语言中，使用构造器（ constructor）构造新实例。构造器是一种特殊的方法，用来构造并初始化对象。一个对象变量并没有实际包含一个对象，而仅仅引用一个对象。

在对象与对象变量之间存在着一个重要的区别。例如，语句：

Date deadline; // deadline doesn’t refer to any object

定义了一个对象变量deadline，它可以引用Date类型的对象。但是一定要认识到：变量deadline不是一个对象，实际上也没有引用对象。此时，不能将任何 Date 方法应用于这个变量上。语句

s = deadline.toString(); // not yet

将产生编译错误。必须先初始化变量deadline。这里有两个选择，当然，可以用新构造的对象初始化这个变量：

deadline = new Date();

也可以让这个变量引用一个已存在的对象：

deadline = brithDay;

如果把一个变量的值赋给另一个变量，两个变量就指向同一个日期，即指向同一个对象。在Java中的null引用对应C++中的NULL指针。

#### Final实例域

可以将实例域定义为final。构建对象时必须初始化这样的域。也就是说，必须确保在每一个构造器执行之后，这个域的值被设置，并且在后面的操作中，不能够再对它进行修改。例如，可以将Employee类中的name域声明为final，因为在对象构建之后，这个值不会再被修改，即没有setName方法。

**public** **class** People {

……

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

}

final修饰符大都应用于基本数据（ primitive）类型域，或不可变（ immutable）类的域（如果类中的每个方法都不会改变其对象，这种类就是不可变的类。例如， String类就是一个不可变的类）。对于可变的类，使用final修饰符可能会对读者造成混乱。例如，

**private** **final** Date hiredate;

仅仅意味着存储在hiredate变量中的对象引用在对象构造之后不能被改变，而并不意味着hiredate 对象是一个常量。任何方法都可以对hiredate引用的对象调用setTime更改器。

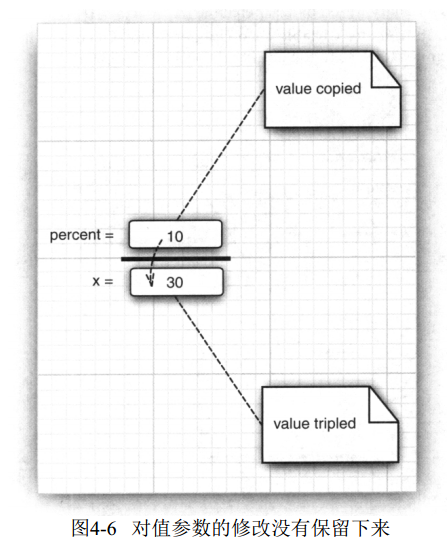
### 方法

#### 方法参数

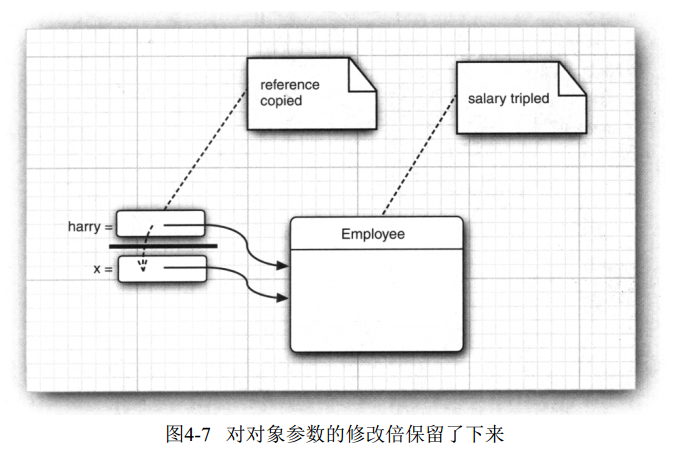
在Java程序设计语言中，方法参数的使用情况：  
• 一个方法不能修改一个基本数据类型的参数（即数值型和布尔型）。  
• 一个方法可以改变一个对象参数的状态。  
• 一个方法不能实现让对象参数引用一个新的对象。

参考范例：4-4 ParamTest.java

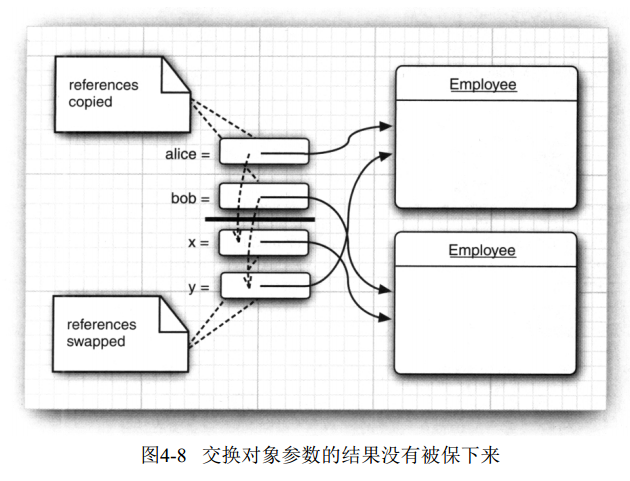
HZ：也就是说，方法接收的参数，如果是基本数据类型，属于值引用，不管方法内部怎么改变这个参数的值，方法外部的值并没有改变；如果是对象参数，因为是对象引用，可以改变这个对象的信息；最难理解的是第三条，可以看下图：



HZ：将拷贝的值传递给方法中，方法中改变成了30，使用之后被丢弃，所以原来的值还是10



HZ：将拷贝的引用传递给方法中，方法中改变了对象的信息，虽然拷贝的引用使用之后被丢弃，但是对象的信息已经被修改，所以原来的引用所引用的对象信息也就更改了。



HZ：将拷贝的引用传递给方法中，虽然方法中互换了引用，但是使用之后拷贝的引用被丢弃，最终外部的引用结果并没有改变。

#### 隐式参数与显示参数

隐式参数（implicit）是方法名前的实例对象

显示参数（explicit）是方法中声明接收的参数

#### 参数数量可变的方法

#### Main方法

main方法不对任何对象进行操作。事实上，在启动程序时还没有任何一个对象。静态的main方法将执行并创建程序所需要的对象。

#### 重载

如果多个方法（比如， GregorianCalendar构造器方法）有相同的名字、不同的参数，便产生了重载。编译器必须挑选出具体执行哪个方法，它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数，或者找出多个可能的匹配，就会产生编译时错误（这个过程被称为重载解析（ overloading resolution）。）

一般方法可以重载，构造方法也可以重载。

### 静态域与静态方法

#### 静态域

如果将域定义为static，每个类中只有一个这样的域。而每一个对象对于所有的实例域却都有自己的一份拷贝。例如，假定需要给每一个雇员赋予惟一的标识码。这里给Employee类添加一个实例域id和一个静态域nextId：

**class** Employee {

**private** **int** id;

**private** **static** **int** *nextId* = 1;

}

现在，每一个雇员对象都有一个自己的id域，但这个类的所有实例将共享一个nextId域。换句话说，如果有1000个Employee类的对象，则有1000个实例域id。但是，只有一个静态域nextId。即使没有一个雇员对象，静态域nextId也存在。它属于类，而不属于任何独立的对象。

下面实现一个简单的方法：

**public** **void** setId(String name) {

**this**.id = *nextId*;

*nextId*++;

}

假定为harry设定雇员标识码：

harry.setId();

harry的id域被设置为静态域nextId当前的值，并且静态域nextId的值加1：

harry.id = Employee.nextId;

Employee.nextId++;

#### 静态常量

静态变量使用得比较少，但静态常量却使用得比较多。例如，在Math类中定义了一个静态常量：

**public** **class** Math {

……

**public** **static** **final** **double** *PI* = 3.14159265358979323846;

……

}

在程序中，可以采用Math.PI的形式获得这个常量。

如果关键字static被省略， PI就变成了Math类的一个实例域。需要通过Math类的对象访问PI，并且每一个Math对象都有它自己的一份PI拷贝。

另一个多次使用的静态常量是System.out。它在System类中声明：

**public** **class** System {

……

**public** **static** **final** PrintStream *out* = . . . ;

……

}

前面曾经提到过，由于每个类对象都可以对公有域进行修改，所以，最好不要将域设计为public。然而，公有常量（即final域）却没问题。因为out被声明为final，所以，不允许再将其他打印流赋给它：

System.out = new PrintStream(. . .); // ERROR -- out is final

注释：如果查看一下System类，就会发现有一个setOut方法，它可以将System.out设置为不同的流。读者可能会感到奇怪，为什么这个方法可以修改final变量的值。原因在于，setOut方法是一个本地方法，而不是用Java语言实现的。本地方法可以绕过Java语言的存取控制机制。这是一种特殊的方法，在自己编写程序时，不应该这样处理。

#### 静态方法

静态方法是一种不能向对象实施操作的方法。例如， Math类的pow方法就是一个静态方法。表达式：

Math.pow(x, a);

计算幂。在运算时，不使用任何Math对象。换句话说，没有隐式的参数。可以认为静态方法是没有this参数的方法（在一个非静态的方法中， this参数表示这个方法的隐式参数）。因为静态方法不能操作对象，所以不能在静态方法中访问实例域。但是，静态方法可以访问自身类中的静态域。下面是使用这种静态方法的一个示例：

**public** **static** **int** getNextId(){

**return** NextId; // returns static field

}

可以通过类名调用这个方法：

**int** n = Employee.getNextId();

这个方法可以省略关键字static吗？答案是肯定的。但是，需要通过Employee类对象的引用调用这个方法。

注释：可以使用对象调用静态方法。例如，如果harry是一个Employee对象，可以用harry.getNextId( )代替Employee.getnextId( )。不过，这种方式很容易造成混淆，其原因是getNextId方法计算的结果与harry毫无关系。我们建议使用类名，而不是对象来调用静态方法。

在下面两种情况下使用静态方法：

• 一个方法不需要访问对象状态，其所需参数都是通过显式参数提供（例如： Math.pow）。• 一个方法只需要访问类的静态域（例如： Employee.getNextId）。

### 构造器

#### 默认构造器

所谓默认构造器是指没有参数的构造器。如果在编写一个类时没有编写构造器，那么系统就会提供一个默认构造器。这个默认构造器将所有的实例域设置为默认值。于是，实例域中的数值型数据设置为0、布尔型数据设置为false、所有对象变量将设置为null。

如果类中提供了至少一个构造器，但是没有提供默认的构造器，则在构造对象时如果没有提供构造参数就会被视为不合法。

#### 默认域初始化

如果在构造器中没有显式地给域赋予初值，那么就会被自动地赋为默认值：数值为0、布尔值为flase、对象引用为null。然而，只有缺少程序设计经验的人才会这样做。确实，如果不明确地对域进行初始化，就会影响程序代码的可读性。

注释：这是域与局部变量的主要不同点。必须明确地初始化方法中的局部变量。但是，如果没有初始化类中的域，将会被初始化为默认值（ 0、 false或null）。例如，仔细看一下Employee类。假定没有在构造器中对某些域进行初始化，就会默认地将salary域初始化为0，将name、 hireDay域初始化为null。但是，这并不是一种良好的编程习惯。如果此时调用getName方法或getHireDay方法，则会得到一个null引用，这应该不是我们所希望的结果。

#### 显示域初始化

由于类的构造器方法可以重载，所以可以采用多种形式设置类的实例域的初始状态。确保不管怎样调用构造器，每个实例域都可以被设置为一个有意义的初值。这是一种很好的设计习惯。可以在类定义中，直接将一个值赋给任何域。

**private** String name = "";

在执行构造器之前，先执行赋值操作。当一个类的所有构造器都希望把相同的值赋予某个特定的实例域时，这种方式特别有用。初始值不一定是常量。在下面的例子中，可以调用方法对域进行初始化。仔细看一下Employee类，其中每个雇员有一个id域。可以使用下列方式进行初始化：

**static** **int** assignId(){

**int** r = *nextId*;

*nextId*++;

**return** r;

}

**private** **int** id = *assignId*();

#### 调用另一个构造器

关键字this引用方法的隐式参数。然而，这个关键字还有另外一个含义。如果构造器的第一个语句形如this(...)，这个构造器将调用同一个类的另一个构造器。下面是一个典型的例子：

**public** Employee(**double** s) {

**this**("Employee #" + *nextId*, s);

}

**public** Employee(String name, **double** salary) {

**this**.name = name;

**this**.salary = salary;

}

当调用new Employee(60000)时， Employee(double)构造器将调用Employee(String, double)构造器。

除了this关键字，还有super关键字可以调用父类的构造方法。

#### 初始化块

前面已经讲过两种初始化数据域的方法：  
• 在构造器中设置值  
• 在声明中赋值  
实际上， Java还有第三种机制，称为初始化块（ initialization block）。在一个类的声明中，  
可以包含多个代码块。只要构造类的对象，这些块就会被执行。但是建议将初始化块放在域定义之后。

#### 对象析构与finalize方法

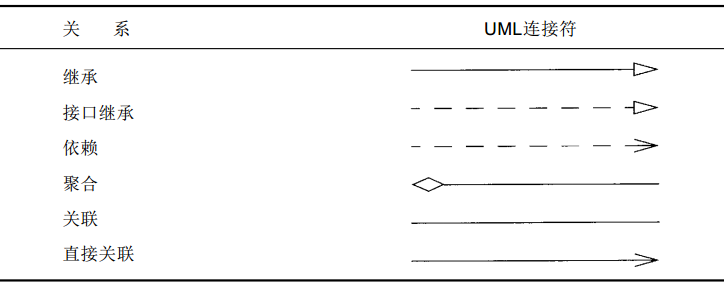
### 访问权限

#### 基于类的访问权限

### 类之间的关系

在类之间，最常见的关系有  
• 依赖（ “ uses-a”）  
• 聚合（ “ has-a”）  
• 继承（ “ is-a”）

UML表示方法：



## 包

### 类的导入

### 静态导入

### 将类放在包中

### 包作用域

### 类路径

## 继承

### 类、超类和子类

### 继承层次

### 多态

### 动态绑定

### 阻止继承

### 强制类型转换

### 抽象类

### 受保护访问

### Object：所有类的超类

Equals方法

相等测试与继承

HashCode方法

ToString方法

### 继承设计的技巧

## 抽象类

## 对象包装器与自动打包

## Java内置实现类

### String & StringBuffter类

#### String 类

##### 不可变字符串

String类没有提供用于修改字符串的方法。如果希望将greeting的内容修改为“ Help!”，不能直接地将greeting的最后两个位置的字符修改为‘ p’和‘ !’。这对于C 程序员来说，将会感到无从下手。如何修改这个字符串呢？在Java中实现这项操作非常容易。首先提取需要的字符，然后再拼接上替换的字符串：

String greeting = "Hello";

greeting = greeting.substring(0, 3) + "p!";

上面这条语句将greeting当前值修改为“ Help！”。

由于不能修改Java字符串中的字符，所以在Java文档中将String类对象称为不可变字符串，如同数字3永远是数字3一样，字符串“ Hello”永远包含字符H、 e、 l、 l和o的代码单元序列，而不能修改其中的任何一个字符。当然，可以修改字符串变量greeting，让它引用另外一个字符串，这就如同可以将存放3的数值变量改成存放4一样。

这样做是否会降低运行效率呢？看起来好像修改一个代码单元要比创建一个新字符串更加简洁。答案是：也对，也不对。的确，通过拼接“ Hel”和“ p!”来创建一个新字符串的效率确实不高。但是，不可变字符串却有一个优点：编译器可以让字符串共享。

为了弄清具体的工作方式，可以想像将各种字符串存放在公共的存储池中。字符串变量指向存储池中相应的位置。如果复制一个字符串变量，原始字符串与复制的字符串共享相同的字符。

总而言之， Java的设计者认为共享带来的高效率远远胜过于提取、拼接字符串所带来的低效率。查看一下程序会发现：很少需要修改字符串，而是往往需要对字符串进行比较（有一种例外情况，将源自于文件或键盘的单个字符或较短的字符串汇集成字符串。为此， Java提供了一个独立的类，在“构建字符串”中将详细地给予介绍）。

##### 代码点与代码单元

Java字符串由char序列组成。从前面已经看到，字符数据类型是一个采用UTF-16编码表示Unicode代码点的代码单元。大多数的常用Unicode字符使用一个代码单元就可以表示，而辅助字符需要一对代码单元表示。

length方法将返回采用UTF-16编码表示的给定字符串所需要的代码单元数量。例如：

String greeting = "Hello海竹！";

**int** len = greeting.length();

要想得到实际的长度，即代码点数量，可以调用：

**int** cpc = greeting.codePointCount(0, len);

System.*out*.println(len);

System.*out*.println(cpc);

调用s.charAt(n) 将返回位置n的代码单元， n介于0～s.length()-1之间。例如：

**char** first = greeting.charAt(0);

要想得到第i个代码点，应该使用下列语句：

**int** index = greeting.offsetByCodePoints(0, i);

**int** cp = greeting.codePointAt(index);

##### 子串

"Hello".substring(0, 3);

String[] split = "H,e,l,l,o".split(",");

##### 拼接

直接使用加号进行拼接：

System.*out*.println("Hello" + "world!");

##### 检测字符串是否相等

可以使用equals方法检测两个字符串是否相等。对于表达式：

**boolean** equals = "Hello".equals("hello");

HZ问题：有面试题说一个常量和一个变量相比较是否相等的时候，把肯定不为空的常量放在前面，可以防止空指针异常。

#### StringBuffter 类

String类时不可变字符串，每次变更数据都是重新创建一个新的对象，那么如果需要频繁修改，就需要使用StringBuffter类来提高效率，这就是构建字符串。

StringBuffer sb = **new** StringBuffer("Hello");

sb.append(" world!");

### Math & StrictMath类

Math类提供了很多常用的数学常量以及方法。在Math类中，为了达到最快的性能，所有的方法都使用计算机浮点单元中的历程。如果得到一个完全可预测的结果比运行速度更重要的话，那么就应该使用StrictMath类。

从JDK 5.0开始，不必在数学方法名和常量名前添加前缀“Math”，而只要在源文件的顶部加上下拉内容就可以了：

**import** **static** java.lang.Math.\*;

### Enum 枚举类

# 接口

接口

接口的特性

对象克隆

接口与回调

抽象类

## 内部类

### 内部类的特性：

使用内部类访问对象状态

内部类的特殊语法规则

由外部方法访问final变量

### 内部类的分类：

成员内部类

局部（方法）内部类

匿名内部类

静态内部类

内部类初始化的时候已经接触到了接口的概念，所以必须放在接口之后

局部内部类

静态内部类

类的访问权限修饰符

# Java 关键字

## final

## static

使用static声明之后，表示“属于类且不属于类对象的变量和函数”。

## this

当前对象

隐式参数，表示当前对象

构造方法调用

参考[调用另一个构造器](#_调用另一个构造器)

## super

# 集合

## 集合接口

### 将集合的接口与实现分裂

### Java类库中的集合接口和迭代器接口

## 具体的集合

### 链表

### 数组列表

### 散列集

### 树集

### 对象的比较

### 队列与双端队列

### 优先级队列

### 映射表

### 专用集与映射表类

## 集合框架

### 视图与包装器

### 批操作

### 集合与数组之间的转换

## 遗留的集合

### HashTable类

### 枚举

### 映射属性表

### 栈

### 位集

# 泛型

## 泛型数组列表

### 访问数组列表元素

### 类型化与原始数组类表的兼容性

## 泛型程序设计

### 为什么使用泛型

### 简单泛型类的定义

### 泛型方法

### 泛型变量的限定

### 泛型代码和虚拟机

### 约束与局限性

### 泛型类型的继承规则

### 通配符类型

### 反射和泛型

# 反射

## Class类

## 捕获异常

## 利用反射分析类的能力

## 在运行时使用反射分析对象

## 使用反射编写泛型数组代码

## 方法指针

# 异常和调试

## 处理异常

## 捕获异常

## 使用异常机制的建议

## 断言

## 日志

## 调试技术