**Java基础**

## Java 简介

**1 Java 简介**

Java是由Sun Microsystems公司于1995年5月推出的Java面向对象程序设计语言和Java平台的总称。由James Gosling和同事们共同研发，并在1995年正式推出。

**2 Java分为三个体系：**

**JavaSE（J2SE）（Java2 Platform Standard Edition，java平台标准版）**

**JavaEE(J2EE)(Java 2 Platform,Enterprise Edition，java平台企业版)**

**JavaME(J2ME)(Java 2 Platform Micro Edition，java平台微型版)。**

2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名以取消其中的数字"2"：J2EE更名为Java EE, J2SE更名为Java SE，J2ME更名为Java ME。

**3 主要特性**

**（1）Java语言是简单的：**

Java语言的语法与C语言和C++语言很接近，使得大多数程序员很容易学习和使用。另一方面，Java丢弃了C++中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。特别地，Java语言不使用指针，而是引用。并提供了自动的废料收集，使得程序员不必为内存管理而担忧。

**（2）Java语言是面向对象的：**

Java语言提供类、接口和继承等面向对象的特性，为了简单起见，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机制（关键字为implements）。Java语言全面支持动态绑定，而C++语言只对虚函数使用动态绑定。总之，Java语言是一个纯的面向对象程序设计语言。

**（3）Java语言是分布式的：**

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。

**（4）Java语言是健壮的：**

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的安全检查机制使得Java更具健壮性。

**（5）Java语言是安全的：**

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。除了Java语言具有的许多安全特性以外，Java对通过网络下载的类具有一个安全防范机制（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，并提供安全管理机制（类SecurityManager）让Java应用设置安全哨兵。

**（6）Java语言是体系结构中立的：**

Java程序（后缀为java的文件）在Java平台上被编译为体系结构中立的字节码格式（后缀为class的文件），然后可以在实现这个Java平台的任何系统中运行。这种途径适合于异构的网络环境和软件的分发。

**（7）Java语言是可移植的：**

这种可移植性来源于体系结构中立性，另外，Java还严格规定了各个基本数据类型的长度。Java系统本身也具有很强的可移植性，Java编译器是用Java实现的，Java的运行环境是用ANSI C实现的。

**（8）Java语言是解释型的：**

如前所述，Java程序在Java平台上被编译为字节码格式，然后可以在实现这个Java平台的任何系统中运行。在运行时，Java平台中的Java解释器对这些字节码进行解释执行，执行过程中需要的类在联接阶段被载入到运行环境中。

**（9）Java是高性能的：**

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++。

**（10）Java语言是多线程的：**

在Java语言中，线程是一种特殊的对象，它必须由Thread类或其子（孙）类来创建。通常有两种方法来创建线程：其一，使用型构为Thread(Runnable)的构造子将一个实现了Runnable接口的对象包装成一个线程，其二，从Thread类派生出子类并重写run方法，使用该子类创建的对象即为线程。值得注意的是Thread类已经实现了Runnable接口，因此，任何一个线程均有它的run方法，而run方法中包含了线程所要运行的代码。线程的活动由一组方法来控制。Java语言支持多个线程的同时执行，并提供多线程之间的同步机制（关键字为synchronized）。

**（11）Java语言是动态的：**

Java语言的设计目标之一是适应于动态变化的环境。Java程序需要的类能够动态地被载入到运行环境，也可以通过网络来载入所需要的类。这也有利于软件的升级。另外，Java中的类有一个运行时刻的表示，能进行运行时刻的类型检查。

**4 发展历史**

1995年5月23日，Java语言诞生

1996年1月，第一个JDK-JDK1.0诞生

1996年4月，10个最主要的操作系统供应商申明将在其产品中嵌入JAVA技术

1996年9月，约8.3万个网页应用了JAVA技术来制作

1997年2月18日，JDK1.1发布

1997年4月2日，JavaOne会议召开，参与者逾一万人，创当时全球同类会议规模之纪录

1997年9月，JavaDeveloperConnection社区成员超过十万

1998年2月，JDK1.1被下载超过2,000,000次

1998年12月8日，JAVA2企业平台J2EE发布

1999年6月，SUN公司发布Java的三个版本：标准版（JavaSE,以前是J2SE）、企业版（JavaEE以前是J2EE）和微型版（JavaME，以前是J2ME）

2000年5月8日，JDK1.3发布

2000年5月29日，JDK1.4发布

2001年6月5日，NOKIA宣布，到2003年将出售1亿部支持Java的手机

2001年9月24日，J2EE1.3发布

2002年2月26日，J2SE1.4发布，自此Java的计算能力有了大幅提升

2004年9月30日18:00PM，J2SE1.5发布，成为Java语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5更名为Java SE 5.0

2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名，以取消其中的数字"2"：J2EE更名为Java EE，J2SE更名为Java SE，J2ME更名为Java ME

2006年12月，SUN公司发布JRE6.0

2009年04月20日，甲骨文74亿美元收购Sun。取得java的版权。

2010年11月，由于甲骨文对于Java社区的不友善，因此Apache扬言将退出JCP[4]。

2011年7月28日，甲骨文发布java7.0的正式版。

2014年3月18日，Oracle公司发表Java SE 8。

**5 Java语言尽量保证系统内存在1G以上，其他工具如下所示：**

Linux 系统、Mac OS 系统、Windows 95/98/2000/XP，WIN 7/8系统。

Java JDK 7、8……

Notepad 编辑器或者其他编辑器。

IDE：Eclipse

安装好以上的工具后，我们就可以输出Java的第一个程序"Hello World！"

public class HelloWorld {

public static void main(String []args) {

System.out.println("Hello World");

}

}

**6 总结**

（1）Java语言其实是有个曾用名的~叫Oak，而且起这个名字的时候也是很随心的，只是因为看到了窗口外的一颗橡树（只是因为面向窗外多看了你一眼~），所以就叫为Oak，但Oak这个名字已经被注册。最后他们以Java这个名字命名了这个语言，据说是Sun公司的程序猿们都很喜欢喝咖啡，而且对爪哇岛的一种咖啡印象很深，所以就有了Java这个经典的名字和咖啡的图标。

（2）在正式学习java并安装配置开发环境之前，有必要了解一些有关Java的专业术语：

JDK（Java Development Kit ）：编写Java程序的程序员使用的软件

JRE（Java Runtime Environment）：运行Java程序的用户使用的软件

Server JRE （Java SE Runtime Environment）：服务端使用的 Java 运行环境

SDK（Software Development Kit）：软件开发工具包，在Java中用于描述1998年~2006年之间的JDK

DAO（Data Access Object）：数据访问接口，数据访问，顾名思义就是与数据库打交道

MVC（Model View Controller）：模型(model)－视图(view)－控制器(controller)的缩写，一种软件设计典范，用于组织代码用一种业务逻辑和数据显示分离的方法

（3）面向对象程序设计的 3 个主要特征：封装性、继承性、多态性。

**封装性（encapsulation）：**封装是一种信息隐蔽技术，它体现于类的说明，是对象的重要特性。封装使数据和加工该数据的方法（函数）封装为一个整体，以实现独立性很强的模块，使得用户只能见到对象的外特性（对象能接受哪些消息，具有哪些处理能力），而对象的内特性（保存内部状态的私有数据和实现加工能力的算法）对用户是隐蔽的。封装的目的在于把对象的设计者和对象的使用者分开，使用者不必知晓其行为实现的细节，只须用设计者提供的消息来访问该对象。

**继承性：**继承性是子类共享其父类数据和方法的机制。它由类的派生功能体现。一个类直接继承其他类的全部描述，同时可修改和扩充。继承具有传递性。继承分为单继承（一个子类有一父类）和多重继承（一个类有多个父类）。类的对象是各自封闭的，如果没继承性机制，则类的对象中的数据、方法就会出现大量重复。继承不仅支持系统的可重用性，而且还促进系统的可扩充性。

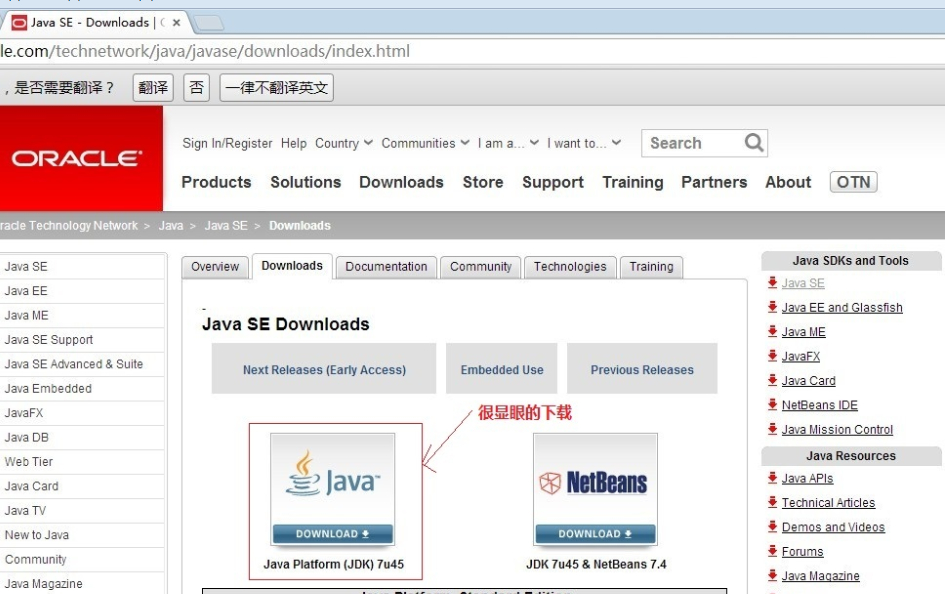
**多态性：**对象根据所接收的消息而做出动作。同一消息被不同的对象接受时可产生完全不同的行动，这种现象称为多态性。利用多态性用户可发送一个通用的信息，而将所有的实现细节都留给接受消息的对象自行决定，如是，同一消息即可调用不同的方法。例如：同样是 run 方法，飞鸟调用时是飞，野兽调用时是奔跑。多态性的实现受到继承性的支持，利用类继承的层次关系，把具有通用功能的协议存放在类层次中尽可能高的地方，而将实现这一功能的不同方法置于较低层次，这样，在这些低层次上生成的对象就能给通用消息以不同的响应。在 OOPL 中可通过在派生类中重定义基类函数（定义为重载函数或虚函数）来实现多态性。

## 环境配置

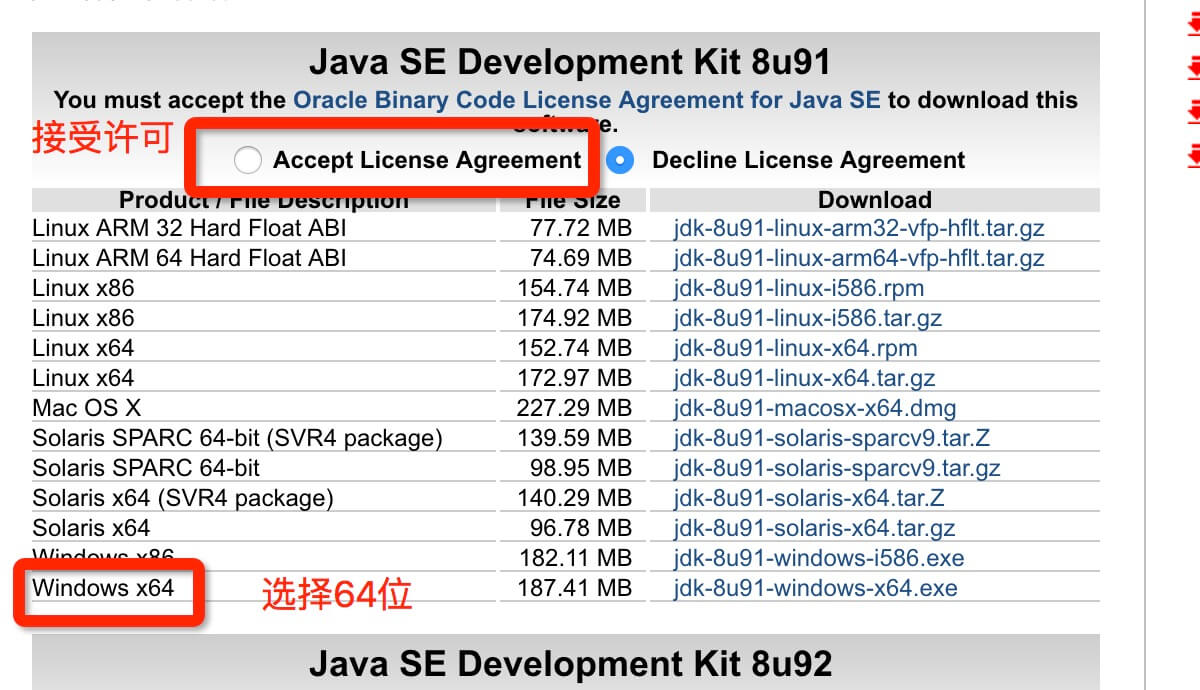
**1 window系统安装java**

（1）下载JDK

首先我们需要下载java开发工具包JDK，下载地址：http://www.oracle.com/technetwork/java/javase/downloads/index.html，点击如下下载按钮：



在下载页面中你需要选择接受许可，并根据自己的系统选择对应的版本，本文以 Window 64位系统为例：



下载后JDK的安装根据提示进行，还有安装JDK的时候也会安装JRE，一并安装就可以了。

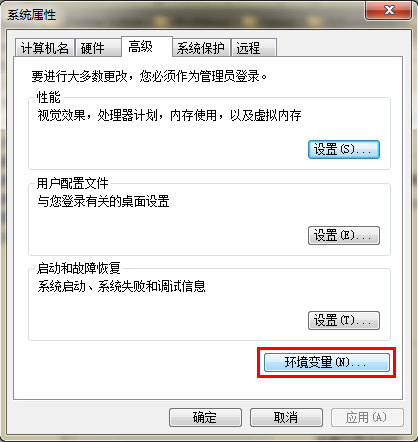
安装JDK，安装过程中可以自定义安装目录等信息，例如我们选择安装目录为 C:\Program Files (x86)\Java\jdk1.8.0\_91。

**（2）配置环境变量**

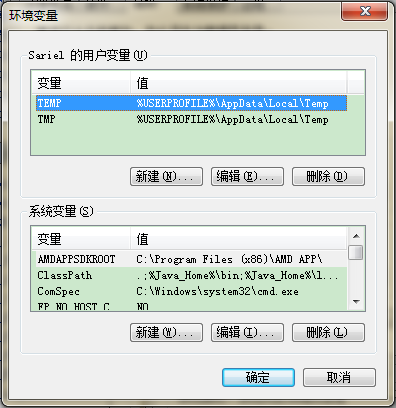
@安装完成后，右击"我的电脑"，点击"属性"，选择"高级系统设置"；



@选择"高级"选项卡，点击"环境变量"；



@然后就会出现如下图所示的画面



在"系统变量"中设置3项属性，JAVA\_HOME,PATH,CLASSPATH(大小写无所谓),若已存在则点击"编辑"，不存在则点击"新建"。

**（3）变量设置参数如下：**

**变量名：JAVA\_HOME**

变量值：C:\Program Files (x86)\Java\jdk1.8.0\_91 // 要根据自己的实际路径配置

**变量名：CLASSPATH**

变量值：.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar; //记得前面有个"."

**变量名：Path**

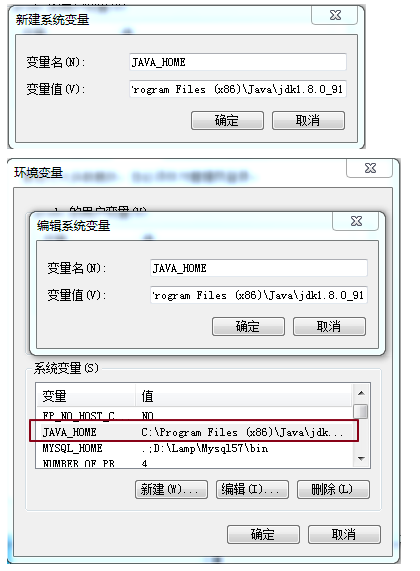
变量值：%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;

注意：在 Windows10 中，因为系统的限制，path 变量只可以使用 JDK 的绝对路径。%JAVA\_HOME% 会无法识别，导致配置失败。如下所示：

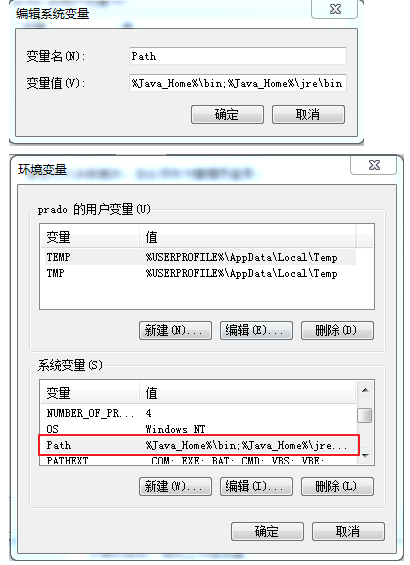
C:\Program Files (x86)\Java\jdk1.8.0\_91\bin;C:\Program Files (x86)\Java\jdk1.8.0\_91\jre\bin;

**（4）设置环境**

JAVA\_HOME 设置



PATH设置



CLASSPATH 设置

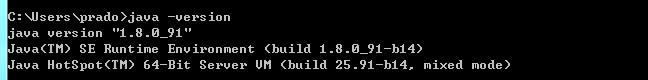


这是 Java 的环境配置，配置完成后，你可以启动 Eclipse 来编写代码，它会自动完成java环境的配置。注意：如果使用1.5以上版本的JDK，不用设置CLASSPATH环境变量，也可以正常编译和运行Java程序。

**（5）测试JDK是否安装成功**

**1 "开始"->"运行"，键入"cmd"；**

**2 键入命令: java -version、java、javac 几个命令**，出现以下信息，说明环境变量配置成功；



**3 Linux，UNIX，Solaris，FreeBSD环境变量设置**

环境变量PATH应该设定为指向Java二进制文件安装的位置。如果设置遇到困难，请参考shell文档。

例如，假设你使用bash作为shell，你可以把下面的内容添加到你的 .bashrc文件结尾: export PATH=/path/to/java:$PATH

**4 流行JAVA开发工具**

正所谓工欲善其事必先利其器，我们在开发java语言过程中同样需要依款不错的开发工具，目前市场上的IDE很多，本文为大家推荐以下下几款java开发工具：

Eclipse（推荐）:另一个免费开源的java IDE，下载地址： http://www.eclipse.org/ 选择 Eclipse IDE for Java Developers：

Idea : Idea IDE

Netbeans:开源免费的java IDE，下载地址： http://www.netbeans.org/index.html

**5 使用 Eclipse 运行第一个 Java 程序**

视频演示如下所示：

您的浏览器不支持 HTML5 video 标签。

HelloWorld.java 文件代码：

public class HelloWorld {

public static void main(String []args) {

System.out.println("Hello World");

}

}

**6 总结**

**（1）在配置环境变量中**

设置JAVA\_HOME:

一是为了方便引用，比如，JDK安装在C:\jdk1.6.0目录里，则设置JAVA\_HOME为该目录路径, 那么以后要使用这个路径的时候, 只需输入%JAVA\_HOME%即可, 避免每次引用都输入很长的路径串;

二则是归一原则, 当JDK路径改变的时候, 仅需更改JAVA\_HOME的变量值即可, 否则,就要更改任何用绝对路径引用JDK目录的文档, 要是万一没有改全, 某个程序找不到JDK, 后果是可想而知的----系统崩溃!

三则是第三方软件会引用约定好的 JAVA\_HOME 变量, 不然, 你不能正常使用该软件。

在系统环境变量那一栏中点 -> 新建 JAVA\_HOME （JAVA\_HOME指向的是JDK的安装路径）

**（2）path 变量**

path 变量使得我们能够在系统中的任何地方运行java应用程序，比如 javac、java、javah 等等,这就要找到我们安装 JDK 的目录，假设我们的JDK安装在 C:\jdk1.6.0 目录下,那么在 C:\jdk1.6.0\bin 目录下就是我们常用的 java 应用程序,我们就需要把 C:\jdk1.6.0\bin 这个目录加到 path 环境变量里面。

**（3）classpath 变量**

classpath 环境变量，是当我们在开发java程序时需要引用别人写好的类时，要让 java 解释器知道到哪里去找这个类。通常，sun 为我们提供了一些额外的丰富的类包，一个是 dt.jar，一个是 tools.jar，这两个 jar 包都位于 C:\jdk1.6.0\lib 目录下，所以通常我们都会把这两个 jar 包加到我们的 classpath 环境变量中 set classpath=.;C:\jdk1.6.0\lib\tools.jar;C:\jdk1.6.0\lib\dt.jar。注意在完成配置环境变量后测试JDK是否安装成功时键入命令：java -version

**（4）JDK 和 JRE 的区别**

JRE(Java Runtime Enviroment) 是 Java 的运行环境。面向 Java 程序的使用者，而不是开发者。如果你仅下载并安装了JRE，那么你的系统只能运行 Java 程序。JRE 是运行 Java 程序所必须环境的集合，包含JVM标准实现及 Java 核心类库。它包括 Java 虚拟机、Java 平台核心类和支持文件。它不包含开发工具(编译器、调试器等)。

JDK(Java Development Kit) 又称 J2SDK(Java2 Software Development Kit)，是 Java 开发工具包，它提供了 Java 的开发环境(提供了编译器 javac 等工具，用于将 java 文件编译为 class 文件)和运行环境(提 供了 JVM 和 Runtime 辅助包，用于解析 class 文件使其得到运行)。如果你下载并安装了 JDK，那么你不仅可以开发 Java 程序，也同时拥有了运行 Java 程序的平台。JDK 是整个 Java 的核心，包括了Java 运行环境(JRE)，一堆 Java 工具 tools.jar 和 Java 标准类库 (rt.jar)。

**（5）为什么要配置 Java 的环境变量呢？如果不搭建环境怎么运行 Java 程序呢?**

下面将会演示在没有搭建环境变量的时候怎么运行 Java 程序，本次演示使用的是 1.6 版本的 JDK，还有就是 Windows 的命令提示符。在安装好 JDK 的前提下，我们在 E:\Test 路径下建一个 Hello.java 的文件，里面就是简单的 Hello World 代码：

public class Hello {

public static void main(String[] args) {

System.out.println("Hello World!");

}

}

然后打开命令提示符：切换盘符至 E 盘的 Test 路径下，如图所示，接着运行下面的命令：

其中双引号引起来的是本机的 jdk 安装路径，javac 是编译 Hello.java 文件的指令，运行上面的指令之后会在 E:\Test 的目录下生成 Hello.class 文件，接着使用下面的命令运行 .class 文件，并输出结果：当我们按照上文说的方法配置好环境变量之后就可以，我们只需要在 E:\Test 路径下输入 javac Hello.java 和 java Hello 这两条命令就可以轻松完成 Hello World! 的输出。这也说明了我们设置的 Java 环境变量是很有用的，**就是能让我们在任何路径之下让我们都能使用 javac 和 java 这两条命令，而不用在前面附带冗长的 JDK 安装路径，能更方便我们工作。**

## 基础语法

**1 基本语法**

（1）大小写敏感：Java是大小写敏感的，这就意味着标识符Hello与hello是不同的。

（2）类名：对于所有的类来说，类名的首字母应该大写。如果类名由若干单词组成，那么每个单词的首字母应该大写，例如 MyFirstJavaClass 。

（3）方法名：所有的方法名都应该以小写字母开头。如果方法名含有若干单词，则后面的每个单词首字母大写。

（4）源文件名：源文件名必须和类名相同。当保存文件的时候，你应该使用类名作为文件名保存（切记Java是大小写敏感的），文件名的后缀为.java。（如果文件名和类名不相同则会导致编译错误）。

（5）主方法入口：所有的Java 程序由public static void main(String []args)方法开始执行。

**2 Java标识符**

Java所有的组成部分都需要名字。类名、变量名以及方法名都被称为标识符。

（1）所有的标识符都应该以字母（A-Z或者a-z）,美元符（$）、或者下划线（\_）开始

（2）首字符之后可以是字母（A-Z或者a-z）,美元符（$）、下划线（\_）或数字的任何字符组合

（3）关键字不能用作标识符

（4）标识符是大小写敏感的

（5）合法标识符举例：age、$salary、\_value、\_\_1\_value

（6）非法标识符举例：123abc、-salary

**3 Java修饰符**

（1）访问控制修饰符 : default, public , protected, private

（2）非访问控制修饰符 : final, abstract, strictfp

**4 在后面的章节中我们会深入讨论Java修饰符**。

局部变量

类变量（静态变量）

成员变量（非静态变量）

**5 Java 关键字**

|  |  |  |
| --- | --- | --- |
| **类别** | **关键字** | **说明** |
| **访问控制** | private | 私有的 |
| protected | 受保护的 |
| public | 公共的 |
| **类、方法和变量修饰符** | abstract | 声明抽象 |
| class | 类 |
| extends | 扩允,继承 |
| final | 最终值,不可改变的 |
| implements | 实现（接口） |
| interface | 接口 |
| native | 本地，原生方法（非Java实现） |
| new | 新,创建 |
| static | 静态 |
| strictfp | 严格,精准 |
| synchronized | 线程,同步 |
| transient | 短暂 |
| volatile | 易失 |
| **程序控制语句** | break | 跳出循环 |
| case | 定义一个值以供switch选择 |
| continue | 继续 |
| default | 默认 |
| do | 运行 |
| else | 否则 |
| for | 循环 |
| if | 如果 |
| instanceof | 实例 |
| return | 返回 |
| switch | 根据值选择执行 |
| while | 循环 |
| **错误处理** | assert | 断言表达式是否为真 |
| catch | 捕捉异常 |
| finally | 有没有异常都执行 |
| throw | 抛出一个异常对象 |
| throws | 声明一个异常可能被抛出 |
| try | 捕获异常 |
| **包相关** | import | 引入 |
| package | 包 |
| **基本类型** | boolean | 布尔型 |
| byte | 字节型 |
| char | 字符型 |
| double | 双精度浮点 |
| float | 单精度浮点 |
| int | 整型 |
| long | 长整型 |
| short | 短整型 |
| **变量引用** | super | 父类,超类 |
| this | 本类 |
| void | 无返回值 |
| **保留关键字** | goto | 是关键字，但不能使用 |
| const | 是关键字，但不能使用 |
| null | 空 |

**6 Java注释**

类似于C/C++，Java也支持单行以及多行注释。注释中的字符将被Java编译器忽略。

public class HelloWorld {

/\* 这是第一个Java程序

\*它将打印Hello World

\* 这是一个多行注释的示例

\*/

public static void main(String []args){

// 这是单行注释的示例

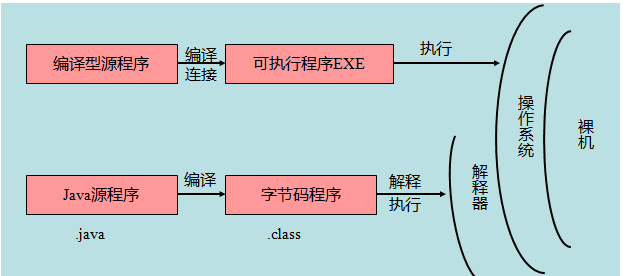
/\* 这个也是单行注释的示例 \*/

System.out.println("Hello World");

}

}

**7 Java 源程序与编译型运行区别**



**8 总结**

**（1）标识符可以用来标识变量名、类名、类中的方法名和文件名等。**

**（2）命名规则：**

1) 由字母、数字、下划线、$组成，不能以数字开头。

2) 大小写敏感。

3) 不得使用java中的关键字和保留字。

**（3）关键字：**都是小写的，jdk1.2多了strictfp(经准浮点型)，关键字 jdk1.4多了assert(断言)关键字，jdk1.5多了enum(枚举) 关键字。

true、false、null 严格说不应该算关键字，应称其为保留字更合适。

**（4）习惯：**

1) 标识符要符合语义信息。

2) 包名所有字母小写。

3) 类名每个单词首字母大写，其它小写，如：TarenaStudent。

4) 变量和方法：第一个单词小写，从第二个单词开始首字母大写，如：tarenaStudent。

5) 常量：所有字母大写，每个单词之间用 \_ 连接。

**（5）常用的转义字符：**

"\b" (退格)

"\f" (换页)

"\n" (换行)

"\r" (回车)

"\t" (水平制表符(到下一个tab位置))

"\' " (单引号)

"\" " (双引号)

"\\" (反斜杠)

**（6）Java的八种基本类型：（按字节来分）**

boolean 布尔型 1个字节 8bit（8位）

byte 字节类型 1个字节

char 字符类型 2个字节

short 短整型 2个字节

int 整型 4个字节

float 浮点型（单精度）4个字节

long 长整型 8个字节

double 双精度类型 8个字节

Java中默认的整数类型是int，如果要定义为long ，则要在数值后加上L或者l

默认的浮点型是双精度浮点，如果要定义float，则要在数值后面加上f或者F

一个字节等于8位，1个字节等于256个数。2^8

一个英文字母或者阿拉伯数字占一个字节

一个汉字占2个字节

**（7）命名规范**

1、 项目名全部小写

2、 包名全部小写

3、 类名首字母大写，如果类名由多个单词组成，每个单词的首字母都要大写。如：public class MyFirstClass{}

4、 变量名、方法名首字母小写，如果名称由多个单词组成，每个单词的首字母都要大写。如：

int index=0;

public void toString(){}

5、 常量名全部大写

public static final String GAME\_COLOR="RED";

**（8）所有命名规则必须遵循以下规则：**

1)、名称只能由字母、数字、下划线、$符号组成

2)、不能以数字开头

3)、名称不能使用JAVA中的关键字。

4)、坚决不允许出现中文及拼音命名。

**（9）注释规范**

1、类注释

在每个类前面必须加上类注释，注释模板如下：

/\*\*

\* Copyright (C), 2006-2010, ChengDu Lovo info. Co., Ltd.

\* FileName: Test.java

\* 类的详细说明

\*

\* @author 类创建者姓名

\* @Date 创建日期

\* @version 1.00

\*/

2、属性注释

在每个属性前面必须加上属性注释，注释模板如下：

/\*\* 提示信息 \*/

private String strMsg = null;

3、方法注释

在每个方法前面必须加上方法注释，注释模板如下：

/\*\*

\* 类方法的详细使用说明

\*

\* @param 参数1 参数1的使用说明

\* @return 返回结果的说明

\* @throws 异常类型.错误代码 注明从此类方法中抛出异常的说明

\*/

4、构造方法注释

在每个构造方法前面必须加上注释，注释模板如下：

/\*\*

\* 构造方法的详细使用说明

\*

\* @param 参数1 参数1的使用说明

\* @throws 异常类型.错误代码 注明从此类方法中抛出异常的说明

\*/

5、方法内部注释

在方法内部使用单行或者多行注释，该注释根据实际情况添加。

//背景颜色

Color bgColor = Color.RED

**（10）Java编程规范**

package的命名: package 的名字由全部小写的字母组成，例如：com.runoob。

class和interface的命名: class和interface的名字由大写字母开头而其他字母都小写的单词组成，例如：Person，RuntimeException。

class变量的命名: 变量的名字用一个小写字母开头，后面的单词用大写字母开头,例如：index，currentImage。

class 方法的命名: 方法的名字用一个小写字母开头，后面的单词用大写字母开头,例如：run()，getBalance()。

staticfinal变量的命名: static final变量的名字所有字母都大写，并且能表示完整含义。例如：PI，PASSWORD。

参数的命名: 参数的名字和变量的命名规范一致。

数组的命名: 数组应该总是用这样的方式来命名：byte[] buffer。

**（11）一个完整的Java。源程序应该包括下列部分：**

package”语句，该部分至多只有一句，必须放在源程序的第一句。

import语句，该部分可以有若干import语句或者没有，必须放在所有的类定义之前。

public classDefinition，公共类定义部分，至多只有一个公共类的定义，Java语言规定该Java源程序的文件名必须与该公共类名完全一致。

classDefinition，类定义部分，可以有0个或者多个类定义。

interfaceDefinition，接口定义部分，可以有0个或者多个接口定义。

package javawork.helloworld;

/\*把编译生成的所有．class文件放到包javawork.helloworld中\*/

import java awt.\*;

//告诉编译器本程序中用到系统的AWT包

import javawork.newcentury;

/\*告诉编译器本程序中用到用户自定义的包javawork.newcentury\*/

public class HelloWorldApp{...｝

/\*公共类HelloWorldApp的定义，名字与文件名相同\*/

class TheFirstClass｛...｝;

//第一个普通类TheFirstClass的定义

interface TheFirstInterface{......}

/\*定义一个接口TheFirstInterface\*/

（12）package语句：由于Java编译器为每个类生成一个字节码文件，且文件名与类名相同因此同名的类有可能发生冲突。为了解决这一问题，Java提供包来管理类名空间，包实 提供了一种命名机制和可见性限制机制。

## 数据类型

**1 Java 基本数据类型**

变量就是申请内存来存储值。也就是说，当创建变量的时候，需要在内存中申请空间。内存管理系统根据变量的类型为变量分配存储空间，分配的空间只能用来储存该类型数据。因此，通过定义不同类型的变量，可以在内存中储存整数、小数或者字符。

**2 Java 的两大数据类型:**

内置数据类型

引用数据类型

**3 内置数据类型**

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

**byte：**

byte 数据类型是8位、有符号的，以二进制补码表示的整数；

最小值是 -128（-2^7）；

最大值是 127（2^7-1）；

默认值是 0；

byte 类型用在大型数组中节约空间，主要代替整数，因为 byte 变量占用的空间只有 int 类型的四分之一；

例子：byte a = 100，byte b = -50。

**short：**

short 数据类型是 16 位、有符号的以二进制补码表示的整数

最小值是 -32768（-2^15）；

最大值是 32767（2^15 - 1）；

Short 数据类型也可以像 byte 那样节省空间。一个short变量是int型变量所占空间的二分之一；

默认值是 0；

例子：short s = 1000，short r = -20000。

**int：**

int 数据类型是32位、有符号的以二进制补码表示的整数；

最小值是 -2,147,483,648（-2^31）；

最大值是 2,147,483,647（2^31 - 1）；

一般地整型变量默认为 int 类型；

默认值是 0 ；

例子：int a = 100000, int b = -200000。

**long：**

long 数据类型是 64 位、有符号的以二进制补码表示的整数；

最小值是 -9,223,372,036,854,775,808（-2^63）；

最大值是 9,223,372,036,854,775,807（2^63 -1）；

这种类型主要使用在需要比较大整数的系统上；

默认值是 0L；

例子： long a = 100000L，Long b = -200000L。

"L"理论上不分大小写，但是若写成"l"容易与数字"1"混淆，不容易分辩。所以最好大写。

**float：**

float 数据类型是单精度、32位、符合IEEE 754标准的浮点数；

float 在储存大型浮点数组的时候可节省内存空间；

默认值是 0.0f；

浮点数不能用来表示精确的值，如货币；

例子：float f1 = 234.5f。

**double：**

double 数据类型是双精度、64 位、符合IEEE 754标准的浮点数；

浮点数的默认类型为double类型；

double类型同样不能表示精确的值，如货币；

默认值是 0.0d；

例子：double d1 = 123.4。

**boolean：**

boolean数据类型表示一位的信息；

只有两个取值：true 和 false；

这种类型只作为一种标志来记录 true/false 情况；

默认值是 false；

例子：boolean one = true。

**char：**

char类型是一个单一的 16 位 Unicode 字符；

最小值是 \u0000（即为0）；

最大值是 \uffff（即为65,535）；

char 数据类型可以储存任何字符；

例子：char letter = 'A';。

**4 实例**

对于数值类型的基本类型的取值范围，我们无需强制去记忆，因为它们的值都已经以常量的形式定义在对应的包装类中了。请看下面的例子：

实例

public class PrimitiveTypeTest {

public static void main(String[] args) {

// byte

System.out.println("基本类型：byte 二进制位数：" + Byte.SIZE);

System.out.println("包装类：java.lang.Byte");

System.out.println("最小值：Byte.MIN\_VALUE=" + Byte.MIN\_VALUE);

System.out.println("最大值：Byte.MAX\_VALUE=" + Byte.MAX\_VALUE);

System.out.println();

// short

System.out.println("基本类型：short 二进制位数：" + Short.SIZE);

System.out.println("包装类：java.lang.Short");

System.out.println("最小值：Short.MIN\_VALUE=" + Short.MIN\_VALUE);

System.out.println("最大值：Short.MAX\_VALUE=" + Short.MAX\_VALUE);

System.out.println();

// int

System.out.println("基本类型：int 二进制位数：" + Integer.SIZE);

System.out.println("包装类：java.lang.Integer");

System.out.println("最小值：Integer.MIN\_VALUE=" + Integer.MIN\_VALUE);

System.out.println("最大值：Integer.MAX\_VALUE=" + Integer.MAX\_VALUE);

System.out.println();

// long

System.out.println("基本类型：long 二进制位数：" + Long.SIZE);

System.out.println("包装类：java.lang.Long");

System.out.println("最小值：Long.MIN\_VALUE=" + Long.MIN\_VALUE);

System.out.println("最大值：Long.MAX\_VALUE=" + Long.MAX\_VALUE);

System.out.println();

// float

System.out.println("基本类型：float 二进制位数：" + Float.SIZE);

System.out.println("包装类：java.lang.Float");

System.out.println("最小值：Float.MIN\_VALUE=" + Float.MIN\_VALUE);

System.out.println("最大值：Float.MAX\_VALUE=" + Float.MAX\_VALUE);

System.out.println();

// double

System.out.println("基本类型：double 二进制位数：" + Double.SIZE);

System.out.println("包装类：java.lang.Double");

System.out.println("最小值：Double.MIN\_VALUE=" + Double.MIN\_VALUE);

System.out.println("最大值：Double.MAX\_VALUE=" + Double.MAX\_VALUE);

System.out.println();

// char

System.out.println("基本类型：char 二进制位数：" + Character.SIZE);

System.out.println("包装类：java.lang.Character");

// 以数值形式而不是字符形式将Character.MIN\_VALUE输出到控制台

System.out.println("最小值：Character.MIN\_VALUE="

+ (int) Character.MIN\_VALUE);

// 以数值形式而不是字符形式将Character.MAX\_VALUE输出到控制台

System.out.println("最大值：Character.MAX\_VALUE="

+ (int) Character.MAX\_VALUE);

}

}

运行实例 »

编译以上代码输出结果如下所示：

基本类型：byte 二进制位数：8

包装类：java.lang.Byte

最小值：Byte.MIN\_VALUE=-128

最大值：Byte.MAX\_VALUE=127

基本类型：short 二进制位数：16

包装类：java.lang.Short

最小值：Short.MIN\_VALUE=-32768

最大值：Short.MAX\_VALUE=32767

基本类型：int 二进制位数：32

包装类：java.lang.Integer

最小值：Integer.MIN\_VALUE=-2147483648

最大值：Integer.MAX\_VALUE=2147483647

基本类型：long 二进制位数：64

包装类：java.lang.Long

最小值：Long.MIN\_VALUE=-9223372036854775808

最大值：Long.MAX\_VALUE=9223372036854775807

基本类型：float 二进制位数：32

包装类：java.lang.Float

最小值：Float.MIN\_VALUE=1.4E-45

最大值：Float.MAX\_VALUE=3.4028235E38

基本类型：double 二进制位数：64

包装类：java.lang.Double

最小值：Double.MIN\_VALUE=4.9E-324

最大值：Double.MAX\_VALUE=1.7976931348623157E308

基本类型：char 二进制位数：16

包装类：java.lang.Character

最小值：Character.MIN\_VALUE=0

最大值：Character.MAX\_VALUE=65535

**5 Float和Double的最小值和最大值都是以科学记数法的形式输出的**，结尾的"E+数字"表示E之前的数字要乘以10的多少次方。比如3.14E3就是3.14 × 103 =3140，3.14E-3 就是 3.14 x 10-3 =0.00314。实际上，JAVA中还存在另外一种基本类型void，它也有对应的包装类 java.lang.Void，不过我们无法直接对它们进行操作。

**6 引用类型**

在Java中，引用类型的变量非常类似于C/C++的指针。引用类型指向一个对象，指向对象的变量是引用变量。这些变量在声明时被指定为一个特定的类型，比如 Employee、Puppy 等。变量一旦声明后，类型就不能被改变了。对象、数组都是引用数据类型。所有引用类型的默认值都是null。一个引用变量可以用来引用任何与之兼容的类型。

例子：Site site = new Site("Runoob")。

**7 Java 常量**

（1）常量在程序运行时是不能被修改的。

在 Java 中使用 final 关键字来修饰常量，声明方式和变量类似：

final double PI = 3.1415927;

虽然常量名也可以用小写，但为了便于识别，通常使用大写字母表示常量。

（2）字面量可以赋给任何内置类型的变量。例如：

byte a = 68;

char a = 'A'

byte、int、long、和short都可以用十进制、16进制以及8进制的方式来表示。

（3）当使用常量的时候，前缀 0 表示 8 进制，而前缀 0x 代表 16 进制, 例如：

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

（4）和其他语言一样，Java的字符串常量也是包含在两个引号之间的字符序列。下面是字符串型字面量的例子：

"Hello World"

"two\nlines"

"\"This is in quotes\""

（5）字符串常量和字符常量都可以包含任何Unicode字符。例如：

char a = '\u0001';

String a = "\u0001";

8 Java语言支持一些特殊的转义字符序列。

**符号 字符含义**

\n 换行 (0x0a)

\r 回车 (0x0d)

\f 换页符(0x0c)

\b 退格 (0x08)

\0 空字符 (0x20)

\s 字符串

\t 制表符

\" 双引号

\' 单引号

\\ 反斜杠

\ddd 八进制字符 (ddd)

\uxxxx 16进制Unicode字符 (xxxx)

**9 自动类型转换**

（1）整型、实型（常量）、字符型数据可以混合运算。运算中，不同类型的数据先转化为同一类型，然后进行运算。

转换从低级到高级。

**低 ------------------------------------> 高**

**byte,short,char—> int —> long—> float —> double**

（2）数据类型转换必须满足如下规则：

1. 不能对boolean类型进行类型转换。

2. 不能把对象类型转换成不相关类的对象。

3. 在把容量大的类型转换为容量小的类型时必须使用强制类型转换。

4. 转换过程中可能导致溢出或损失精度

例如：

int i =128;

byte b = (byte)i;

因为 byte 类型是 8 位，最大值为127，所以当 int 强制转换为 byte 类型时，值 128 时候就会导致溢出。

5. 浮点数到整数的转换是通过舍弃小数得到，而不是四舍五入，例如：

(int)23.7 == 23;

(int)-45.89f == -45

（3）自动类型转换必须满足转换前的数据类型的位数要低于转换后的数据类型，例如: short数据类型的位数为16位，就可以自动转换位数为32的int类型，同样float数据类型的位数为32，可以自动转换为64位的double类型。

public class ZiDongLeiZhuan{

public static void main(String[] args){

char c1='a';//定义一个char类型

int i1 = c1;//char自动类型转换为int

System.out.println("char自动类型转换为int后的值等于"+i1);

char c2 = 'A';//定义一个char类型

int i2 = c2+1;//char 类型和 int 类型计算

System.out.println("char类型和int计算后的值等于"+i2);

}

}

运行结果为:

char自动类型转换为int后的值等于97

char类型和int计算后的值等于66

解析：c1 的值为字符 a ,查 ASCII 码表可知对应的 int 类型值为 97， A 对应值为 65，所以 i2=65+1=66。

**10 强制类型转换**

（1）条件是转换的数据类型必须是兼容的。

（2）格式：(type)value type是要强制类型转换后的数据类型 实例：

public class QiangZhiZhuanHuan{

public static void main(String[] args){

int i1 = 123;

byte b = (byte)i1;//强制类型转换为byte

System.out.println("int强制类型转换为byte后的值等于"+b);

}

}

（3）运行结果：

int强制类型转换为byte后的值等于123

**11 隐含强制类型转换**

（1）整数的默认类型是 int。

（2）浮点型不存在这种情况，因为在定义 float 类型时必须在数字后面跟上 F 或者 f。

**12 总结**

**（1）Java 里使用 long 类型的数据一定要在数值后面加上 L，否则将作为整**型解析：

long g = (long)9223372036854775807;

long h = (long)-9223372036854775808;

或者

long g = 9223372036854775807;

long h = -9223372036854775808;

会出现以下报错信息：

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

The literal 9223372036854775807 of type int is out of range

The literal 9223372036854775808 of type int is out of range

溢出了~

解决方法在数值后面加上 L：

long value = 9223372036854775807L;

**（2）引用类型是一个对象类型，它的值是指向内存空间的引用，就是地址，所指向的内存中保存着变量所表示的一个值或一组值。**

int a;

a = 250; // 声明变量a的同时，系统给a分配了空间。

引用类型就不是了，只给变量分配了引用空间，数据空间没有分配，因为不知道数据是什么。

错误的例子：

MyDate today;

today.day = 4; // 发生错误，因为today对象的数据空间未分配。

引用类型变量在声明后必须通过实例化开辟数据空间，才能对变量所指向的对象进行访问。

MyDate today; //将变量分配一个保存引用的空间

today = new MyDate(); // 这句话是2步，首先执行new MyDate（），给today变量开辟数据空间，然后再执行赋值操作

引用变量赋值：

MyDate a，b; // 在内存开辟两个引用空间

a = new MyDate(); // 开辟MyDate对象的数据空间，并把该空间的首地址赋给a

b = a; // 将a存储空间中的地址写到b的存储空间中

**（3）强转成int类型**

short a = 1;

short b = 2;

那么 a+b 是什么类型？

答：在java的世界里，如果比int类型小的类型做运算，java在编译的时候就会将它们统一强转成int类型。当是比int类型大的类型做运算，就会自动转换成它们中最大类型那个。

**（4）数据类型转换的补充**

1）包装类过渡类型转换

一般情况下，我们首先声明一个变量，然后生成一个对应的包装类，就可以利用包装类的各种方法进行类型转换了。例如：

当希望把float型转换为double型时：

float f1=100.00f;

Float F1=new Float(f1);

double d1=F1.doubleValue();//F1.doubleValue()为Float类的返回double值型的方法

简单类型的变量转换为相应的包装类，可以利用包装类的构造函数。即：Boolean(boolean value)、Character(char value)、Integer(int value)、Long(long value)、Float(float value)、Double(double value)

而在各个包装类中，总有形为××Value()的方法，来得到其对应的简单类型数据。利用这种方法，也可以实现不同数值型变量间的转换，例如，对于一个双精度实型类，intValue()可以得到其对应的整型变量，而doubleValue()可以得到其对应的双精度实型变量。

2）字符串与其它类型间的转换

其它类型向字符串的转换

调用类的串转换方法:X.toString();

自动转换:X+"";

使用String的方法:String.valueOf(X);

3）字符串作为值,向其它类型的转换

@先转换成相应的封装器实例,再调用对应的方法转换成其它类型

例如，字符中"32.1"转换double型的值的格式为:new Float("32.1").doubleValue()。也可以用:Double.valueOf("32.1").doubleValue()

@静态parseXXX方法

String s = "1";

byte b = Byte.parseByte( s );

short t = Short.parseShort( s );

int i = Integer.parseInt( s );

long l = Long.parseLong( s );

Float f = Float.parseFloat( s );

Double d = Double.parseDouble( s );

@Character的getNumericValue(char ch)方法

@Date类与其它数据类型的相互转换

整型和Date类之间并不存在直接的对应关系，只是你可以使用int型为分别表示年、月、日、时、分、秒，这样就在两者之间建立了一个对应关系，在作这种转换时，你可以使用Date类构造函数的三种形式：

Date(int year, int month, int date)：以int型表示年、月、日

Date(int year, int month, int date, int hrs, int min)：以int型表示年、月、日、时、分

Date(int year, int month, int date, int hrs, int min, int sec)：以int型表示年、月、日、时、分、秒

在长整型和Date类之间有一个很有趣的对应关系，就是将一个时间表示为距离格林尼治标准时间1970年1月1日0时0分0秒的毫秒数。对于这种对应关系，Date类也有其相应的构造函数：Date(long date)。

获取Date类中的年、月、日、时、分、秒以及星期你可以使用Date类的getYear()、getMonth()、getDate()、getHours()、getMinutes()、getSeconds()、getDay()方法，你也可以将其理解为将Date类转换成int。

而Date类的getTime()方法可以得到我们前面所说的一个时间对应的长整型数，与包装类一样，Date类也有一个toString()方法可以将其转换为String类。

有时我们希望得到Date的特定格式，例如20020324，我们可以使用以下方法，首先在文件开始引入：

import java.text.SimpleDateFormat;

java.util.Date date = new java.util.Date();

//如果希望得到YYYYMMDD的格式

SimpleDateFormat sy1=new SimpleDateFormat("yyyyMMdd");

String dateFormat=sy1.format(date);

//如果希望分开得到年，月，日

SimpleDateFormat sy=new SimpleDateFormat("yyyy");

SimpleDateFormat sm=new SimpleDateFormat("MM");

SimpleDateFormat sd=new SimpleDateFormat("dd");

String syear=sy.format(date);

String smon=sm.format(date);

String sday=sd.format(date);

**（5）总结：**

1）只有 boolean 不参与数据类型的转换

2）自动类型的转换：

a.常数在表数范围内是能够自动类型转换的

b.数据范围小的能够自动数据类型大的转换（注意特例）

float 到 int，float 到 long，double 到 int，double 到 long 等由浮点类型转换成整数类型时，是不会自动转换的，不然将会丢失精度。

c.引用类型能够自动转换为父类的

d.基本类型和它们包装类型是能够互相转换的

3）强制类型转换：用圆括号括起来目标类型，置于变量前

**（6）包装类 Integer 的自动封装**

Integer a = 1;

int 类型在赋值到 Integer 类时，会自动封装，调用 Integer 的 valueOf(int i) 方法。

Integer a = Integer.valueOf(1);

public static Integer valueOf(int i) {

assert IntegerCache.high >= 127;

if (i >= IntegerCache.low && i <= IntegerCache.high)

return IntegerCache.cache[i + (-IntegerCache.low)];

return new Integer(i);

}

当 i >= -128 && i <= 127 时，Integer.valueOf(i) 会将 i 存储在内部类 IntegerCache的static final Integer cache[]里，这一字节的缓存内存地址是静态的，返回值即:

IntegerCache.cache[i + (-IntegerCache.low)]

因此:

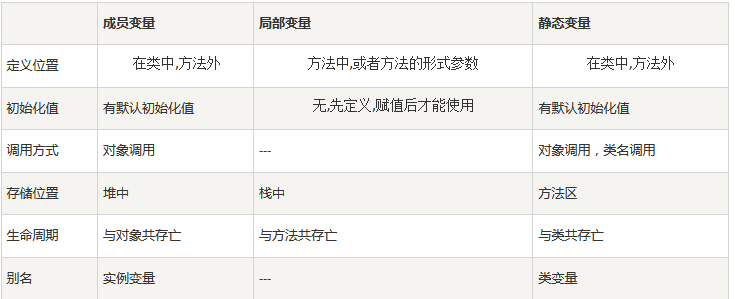
Integer a = 1;

Integer b = 1;

a 和 b 的引用都指向同一个对象，即 a == b。

## 变量类型

**1 Java 变量类型**



（1）在Java语言中，所有的变量在使用前必须声明。声明变量的基本格式如下：

type identifier [ = value][, identifier [= value] ...] ;

格式说明：type为Java数据类型。identifier是变量名。可以使用逗号隔开来声明多个同类型变量。

以下列出了一些变量的声明实例。注意有些包含了初始化过程。

int a, b, c; // 声明三个int型整数：a、 b、c

int d = 3, e = 4, f = 5; // 声明三个整数并赋予初值

byte z = 22; // 声明并初始化 z

String s = "runoob"; // 声明并初始化字符串 s

double pi = 3.14159; // 声明了双精度浮点型变量 pi

char x = 'x'; // 声明变量 x 的值是字符 'x'。

（2）Java语言支持的变量类型有：

类变量：独立于方法之外的变量，用 static 修饰。

实例变量：独立于方法之外的变量，不过没有 static 修饰。

局部变量：类的方法中的变量。

实例

public class Variable{

static int allClicks=0; // 类变量

String str="hello world"; // 实例变量

public void method(){

int i =0; // 局部变量

}

}

**2 Java 局部变量**

局部变量声明在方法、构造方法或者语句块中；

局部变量在方法、构造方法、或者语句块被执行的时候创建，当执行完成，变量将会被销毁；

访问修饰符不能用于局部变量；

局部变量只在声明它的方法、构造方法或者语句块中可见；

局部变量是在栈上分配的。

局部变量没有默认值，所以局部变量被声明后，必须经过初始化，才可以使用。

**（1）在以下实例中age是一个局部变量。定义在pupAge()方法中，它的作用域就限制在这个方法中。**

package com.runoob.test;

public class Test{

public void pupAge(){

int age = 0;

age = age + 7;

System.out.println("小狗的年龄是: " + age);

}

public static void main(String[] args){

Test test = new Test();

test.pupAge();

}

}

以上实例编译运行结果如下:

小狗的年龄是: 7

**（2）在下面的例子中 age 变量没有初始化，所以在编译时会出错：**

package com.runoob.test;

public class Test{

public void pupAge(){

int age;

age = age + 7;

System.out.println("小狗的年龄是 : " + age);

}

public static void main(String[] args){

Test test = new Test();

test.pupAge();

}

}

以上实例编译运行结果如下:

Test.java:4:variable number might not have been initialized

age = age + 7;

1 error

**3 实例变量**

实例变量声明在一个类中，但在方法、构造方法和语句块之外；

当一个对象被实例化之后，每个实例变量的值就跟着确定；

实例变量在对象创建的时候创建，在对象被销毁的时候销毁；

实例变量的值应该至少被一个方法、构造方法或者语句块引用，使得外部能够通过这些方式获取实例变量信息；

实例变量可以声明在使用前或者使用后；

访问修饰符可以修饰实例变量；

实例变量对于类中的方法、构造方法或者语句块是可见的。一般情况下应该把实例变量设为私有。通过使用访问修饰符可以使实例变量对子类可见；

实例变量具有默认值。数值型变量的默认值是0，布尔型变量的默认值是false，引用类型变量的默认值是null。变量的值可以在声明时指定，也可以在构造方法中指定；

实例变量可以直接通过变量名访问。但在静态方法以及其他类中，就应该使用完全限定名：ObejectReference.VariableName。

**（1）Employee.java 文件代码：**

import java.io.\*;

public class Employee{

// 这个实例变量对子类可见

public String name;

// 私有变量，仅在该类可见

private double salary;

//在构造器中对name赋值

public Employee (String empName){

name = empName;

}

//设定salary的值

public void setSalary(double empSal){

salary = empSal;

}

// 打印信息

public void printEmp(){

System.out.println("名字 : " + name );

System.out.println("薪水 : " + salary);

}

public static void main(String[] args){

Employee empOne = new Employee("RUNOOB");

empOne.setSalary(1000);

empOne.printEmp();

}

}

（2）以上实例编译运行结果如下:

$ javac Employee.java

$ java Employee

名字 : RUNOOB

薪水 : 1000.0

**4 类变量（静态变量）**

类变量也称为静态变量，在类中以static关键字声明，但必须在方法构造方法和语句块之外。

无论一个类创建了多少个对象，类只拥有类变量的一份拷贝。

静态变量除了被声明为常量外很少使用。常量是指声明为public/private，final和static类型的变量。常量初始化后不可改变。

静态变量储存在静态存储区。经常被声明为常量，很少单独使用static声明变量。

静态变量在第一次被访问时创建，在程序结束时销毁。

与实例变量具有相似的可见性。但为了对类的使用者可见，大多静态变量声明public类型。

默认值和实例变量相似。数值型变量默认值是0，布尔型默认值是false，引用类型默认值是null。变量的值可以在声明的时候指定，也可以在构造方法中指定。此外，静态变量还可以在静态语句块中初始化。

静态变量可以通过：ClassName.VariableName的方式访问。

类变量被声明为public static final类型时，类变量名称一般建议使用大写字母。如果静态变量不是public和final类型，其命名方式与实例变量以及局部变量的命名方式一致。

Employee.java 文件代码：

import java.io.\*;

public class Employee {

//salary是静态的私有变量

private static double salary;

// DEPARTMENT是一个常量

public static final String DEPARTMENT = "开发人员";

public static void main(String[] args){

salary = 10000;

System.out.println(DEPARTMENT+"平均工资:"+salary);

}

}

## 修饰符

**1 Java语言提供了很多修饰符，主要分为以下两类：**

访问修饰符

非访问修饰符

（1）修饰符用来定义类、方法或者变量，通常放在语句的最前端。我们通过下面的例子来说明：

public class className {

// ...

}

private boolean myFlag;

static final double weeks = 9.5;

protected static final int BOXWIDTH = 42;

public static void main(String[] arguments) {

// 方法体

}

**2 访问控制修饰符**

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

default (即缺省，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。

private : 在同一类内可见。使用对象：变量、方法。 注意：不能修饰类（外部类）

public : 对所有类可见。使用对象：类、接口、变量、方法

protected : 对同一包内的类和所有子类可见。使用对象：变量、方法。 注意：不能修饰类（外部类）。

**3 我们可以可以通过以下表来说明访问权限：**

**访问控制 修饰符 当前类 同一包内 子孙类 其他包 其他包子孙类**

public Y Y Y Y Y

protected Y Y Y N Y/N（说明）

default Y Y N N N

private Y N N N N

默认访问修饰符-不使用任何关键字

使用默认访问修饰符声明的变量和方法，对同一个包内的类是可见的。接口里的变量都隐式声明为 public static final,而接口里的方法默认情况下访问权限为 public。

如下例所示，变量和方法的声明可以不使用任何修饰符。

实例

String version = "1.5.1";

boolean processOrder() {

return true;

}

**3 私有访问修饰符-private**

私有访问修饰符是最严格的访问级别，所以被声明为 private 的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为 private。声明为私有访问类型的变量只能通过类中公共的 getter 方法被外部类访问。Private 访问修饰符的使用主要用来隐藏类的实现细节和保护类的数据。下面的类使用了私有访问修饰符：

public class Logger {

private String format;

public String getFormat() {

return this.format;

}

public void setFormat(String format) {

this.format = format;

}

}

实例中，Logger 类中的 format 变量为私有变量，所以其他类不能直接得到和设置该变量的值。为了使其他类能够操作该变量，定义了两个 public 方法：getFormat() （返回 format的值）和 setFormat(String)（设置 format 的值）

**4 公有访问修饰符-public**

被声明为 public 的类、方法、构造方法和接口能够被任何其他类访问。

如果几个相互访问的 public 类分布在不同的包中，则需要导入相应 public 类所在的包。由于类的继承性，类所有的公有方法和变量都能被其子类继承。

以下函数使用了公有访问控制：

public static void main(String[] arguments) {

// ...

}

Java 程序的 main() 方法必须设置成公有的，否则，Java 解释器将不能运行该类。

受保护的访问修饰符-protected

**5 protected 需要从以下两个点来分析说明：**

子类与基类在同一包中：被声明为 protected 的变量、方法和构造器能被同一个包中的任何其他类访问；子类与基类不在同一包中：那么在子类中，子类实例可以访问其从基类继承而来的 protected 方法，而不能访问基类实例的protected方法。protected 访问修饰符可以修饰类和接口，方法和成员变量能够声明为 protected，但是接口的成员变量和成员方法不能声明为 protected。子类能访问 protected 修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。下面的父类使用了 protected 访问修饰符，子类重写了父类的 openSpeaker() 方法。

class AudioPlayer {

protected boolean openSpeaker(Speaker sp) {

// 实现细节

}

}

class StreamingAudioPlayer extends AudioPlayer {

protected boolean openSpeaker(Speaker sp) {

// 实现细节

}

}

如果把 openSpeaker() 方法声明为 private，那么除了 AudioPlayer 之外的类将不能访问该方法。

如果把 openSpeaker() 声明为 public，那么所有的类都能够访问该方法。

如果我们只想让该方法对其所在类的子类可见，则将该方法声明为 protected。

protected 是最难理解的一种 Java 类成员访问权限修饰词，更多详细内容请查看 Java protected 关键字详解。

**6 访问控制和继承**

请注意以下方法继承的规则：

父类中声明为 public 的方法在子类中也必须为 public。

父类中声明为 protected 的方法在子类中要么声明为 protected，要么声明为 public，不能声明为 private。

父类中声明为 private 的方法，不能够被继承。

**7 非访问修饰符**

为了实现一些其他的功能，Java 也提供了许多非访问修饰符。

static 修饰符，用来修饰类方法和类变量。

final 修饰符，用来修饰类、方法和变量，final 修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。

abstract 修饰符，用来创建抽象类和抽象方法。

synchronized 和 volatile 修饰符，主要用于线程的编程。

**8 static 修饰符**

（1）静态变量：static 关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被称为类变量。局部变量不能被声明为 static 变量。

（2）静态方法：static 关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

（3）对类变量和方法的访问可以直接使用 classname.variablename 和 classname.methodname 的方式访问。

如下例所示，static修饰符用来创建类方法和类变量。

public class InstanceCounter {

private static int numInstances = 0;

protected static int getCount() {

return numInstances;

}

private static void addInstance() {

numInstances++;

}

InstanceCounter() {

InstanceCounter.addInstance();

}

public static void main(String[] arguments) {

System.out.println("Starting with " +

InstanceCounter.getCount() + " instances");

for (int i = 0; i < 500; ++i){

new InstanceCounter();

}

System.out.println("Created " +

InstanceCounter.getCount() + " instances");

}

}

以上实例运行编辑结果如下:

Starting with 0 instances

Created 500 instances

**9 final 修饰符**

**（1）final 变量能被显式地初始化并且只能初始化一**次。被声明为 final 的对象的引用不能指向不同的对象。但是 final 对象里的数据可以被改变。也就是说 final 对象的引用不能改变，但是里面的值可以改变。

final 修饰符通常和 static 修饰符一起使用来创建类常量。

public class Test{

final int value = 10;

// 下面是声明常量的实例

public static final int BOXWIDTH = 6;

static final String TITLE = "Manager";

public void changeValue(){

value = 12; //将输出一个错误

}

}

**（2）final 方法**

类中的 final 方法可以被子类继承，但是不能被子类修改。

声明 final 方法的主要目的是防止该方法的内容被修改。

如下所示，使用 final 修饰符声明方法。

public class Test{

public final void changeName(){

// 方法体

}

}

**（3）final 类**

final 类不能被继承，没有类能够继承 final 类的任何特性。

实例

public final class Test {

// 类体

}

**10 abstract 修饰符**

**（1）抽象类：**

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。

一个类不能同时被 abstract 和 final 修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。

抽象类可以包含抽象方法和非抽象方法。

abstract class Caravan{

private double price;

private String model;

private String year;

public abstract void goFast(); //抽象方法

public abstract void changeColor();

}

**（2）抽象方法**

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。

抽象方法不能被声明成 final 和 static。

任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

抽象方法的声明以分号结尾，例如：public abstract sample();。

public abstract class SuperClass{

abstract void m(); //抽象方法

}

class SubClass extends SuperClass{

//实现抽象方法

void m(){

.........

}

}

**11 synchronized 修饰符**

synchronized 关键字声明的方法同一时间只能被一个线程访问。synchronized 修饰符可以应用于四个访问修饰符。

public synchronized void showDetails(){

.......

}

**12 transient 修饰符**

序列化的对象包含被 transient 修饰的实例变量时，java 虚拟机(JVM)跳过该特定的变量。

该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

实例

public transient int limit = 55; // 不会持久化

public int b; // 持久化

volatile 修饰符

**13 volatile 修饰的成员变**量在每次被线程访问时，都强制从共享内存中重新读取该成员变量的值。而且，当成员变量发生变化时，会强制线程将变化值回写到共享内存。

这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

一个 volatile 对象引用可能是 null。

public class MyRunnable implements Runnable

{

private volatile boolean active;

public void run()

{

active = true;

while (active) // 第一行

{

// 代码

}

}

public void stop()

{

active = false; // 第二行

}

}

通常情况下，在一个线程调用 run() 方法（在 Runnable 开启的线程），在另一个线程调用 stop() 方法。 如果 第一行 中缓冲区的 active 值被使用，那么在 第二行 的 active 值为 false 时循环不会停止。但是以上代码中我们使用了 volatile 修饰 active，所以该循环会停止。

## 运算符

**1 计算机的最基本用途之一就是执行数学运算，作为一门计算机语言，Java也提供了一套丰富的运算符来操纵变量。我们可以把运算符分成以下几组：**

算术运算符

关系运算符

位运算符

逻辑运算符

赋值运算符

其他运算符

**2 算术运算符**

算术运算符用在数学表达式中，它们的作用和在数学中的作用一样。下表列出了所有的算术运算符。表格中的假设整数变量A的值为10，变量B的值为20：

|  |  |  |
| --- | --- | --- |
| **操作符** | **描述** | **例子** |
| **+** | 加法 - 相加运算符两侧的值 | A + B 等于 30 |
| **-** | 减法 - 左操作数减去右操作数 | A – B 等于 -10 |
| **\*** | 乘法 - 相乘操作符两侧的值 | A \* B等于200 |
| **/** | 除法 - 左操作数除以右操作数 | B / A等于2 |
| **％** | 取余 - 左操作数除以右操作数的余数 | B%A等于0 |
| **++** | 自增: 操作数的值增加1 | B++ 或 ++B 等于 21（区别详见下文） |
| **--** | 自减: 操作数的值减少1 | B-- 或 --B 等于 19（区别详见下文） |

下面的简单示例程序演示了算术运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

public class Test {

public static void main(String[] args) {

int a = 10;

int b = 20;

int c = 25;

int d = 25;

System.out.println("a + b = " + (a + b) );

System.out.println("a - b = " + (a - b) );

System.out.println("a \* b = " + (a \* b) );

System.out.println("b / a = " + (b / a) );

System.out.println("b % a = " + (b % a) );

System.out.println("c % a = " + (c % a) );

System.out.println("a++ = " + (a++) );

System.out.println("a-- = " + (a--) );

// 查看 d++ 与 ++d 的不同

System.out.println("d++ = " + (d++) );

System.out.println("++d = " + (++d) );

}

}

运行 »

以上编译运行结果如下：

a + b = 30

a - b = -10

a \* b = 200

b / a = 2

b % a = 0

c % a = 5

a++ = 10

a-- = 11

d++ = 25

++d = 27

**3 自增自减运算符**

（1）自增（++）自减（--）运算符是一种特殊的算术运算符，在算术运算符中需要两个操作数来进行运算，而自增自减运算符是一个操作数。

public class selfAddMinus{

public static void main(String[] args){

int a = 3;//定义一个变量；

int b = ++a;//自增运算

int c = 3;

int d = --c;//自减运算

System.out.println("进行自增运算后的值等于"+b);

System.out.println("进行自减运算后的值等于"+d);

}

}

运行结果为：

进行自增运算后的值等于4

进行自减运算后的值等于2

解析：

int b = ++a; 拆分运算过程为: a=a+1=4; b=a=4, 最后结果为b=4,a=4

int d = --c; 拆分运算过程为: c=c-1=2; d=c=2, 最后结果为d=2,c=2

**（2）前缀自增自减法(++a,--a):** 先进行自增或者自减运算，再进行表达式运算。

**（3）后缀自增自减法(a++,a--):** 先进行表达式运算，再进行自增或者自减运算 ：

public class selfAddMinus{

public static void main(String[] args){

int a = 5;//定义一个变量；

int b = 5;

int x = 2\*++a;

int y = 2\*b++;

System.out.println("自增运算符前缀运算后a="+a+",x="+x);

System.out.println("自增运算符后缀运算后b="+b+",y="+y);

}

}

运行结果为：

自增运算符前缀运算后a=6，x=12

自增运算符后缀运算后b=6，y=10

**4 关系运算符**

（1）下表为Java支持的关系运算符

表格中的整数变量A的值为10，变量B的值为20：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **例子** |
| **==** | 检查如果两个操作数的值是否相等，如果相等则条件为真。 | （A == B）为假(非真)。 |
| **!=** | 检查如果两个操作数的值是否相等，如果值不相等则条件为真。 | (A != B) 为真。 |
| **>** | 检查左操作数的值是否大于右操作数的值，如果是那么条件为真。 | （A> B）非真。 |
| **<** | 检查左操作数的值是否小于右操作数的值，如果是那么条件为真。 | （A <B）为真。 |
| **>=** | 检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真。 | （A> = B）为假。 |
| **<=** | 检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真。 | （A <= B）为真。 |

（2）下面的简单示例程序演示了关系运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

Test.java 文件代码：

public class Test {

public static void main(String[] args) {

int a = 10;

int b = 20;

System.out.println("a == b = " + (a == b) );

System.out.println("a != b = " + (a != b) );

System.out.println("a > b = " + (a > b) );

System.out.println("a < b = " + (a < b) );

System.out.println("b >= a = " + (b >= a) );

System.out.println("b <= a = " + (b <= a) );

}

}

以上编译运行结果如下：

a == b = false

a != b = true

a > b = false

a < b = true

b >= a = true

b <= a = false

**5 位运算符**

（1）Java定义了位运算符，应用于整数类型(int)，长整型(long)，短整型(short)，字符型(char)，和字节型(byte)等类型。

位运算符作用在所有的位上，并且按位运算。假设a = 60，b = 13;它们的二进制格式表示将如下：

A = 0011 1100

B = 0000 1101

-----------------

A&b = 0000 1100

A | B = 0011 1101

A ^ B = 0011 0001

~A= 1100 0011

（2）下表列出了位运算符的基本运算,假设整数变量A的值为60和变量B的值为13：

|  |  |  |
| --- | --- | --- |
| **操作符** | **描述** | **例子** |
| **＆** | 如果相对应位都是1，则结果为1，否则为0 | （A＆B），得到12，即0000 1100 |
| **|** | 如果相对应位都是0，则结果为0，否则为1 | （A | B）得到61，即 0011 1101 |
| **^** | 如果相对应位值相同，则结果为0，否则为1 | （A ^ B）得到49，即 0011 0001 |
| **〜** | 按位取反运算符翻转操作数的每一位，即0变成1，1变成0。 | （〜A）得到-61，即1100 0011 |
| **<<** | 按位左移运算符。左操作数按位左移右操作数指定的位数。 | A << 2得到240，即 1111 0000 |
| **>>** | 按位右移运算符。左操作数按位右移右操作数指定的位数。 | A >> 2得到15即 1111 |
| **>>>** | 按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充。 | A>>>2得到15即0000 11 |

（3）下面的简单示例程序演示了位运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

Test.java 文件代码：

public class Test {

public static void main(String[] args) {

int a = 60; /\* 60 = 0011 1100 \*/

int b = 13; /\* 13 = 0000 1101 \*/

int c = 0;

c = a & b; /\* 12 = 0000 1100 \*/

System.out.println("a & b = " + c );

c = a | b; /\* 61 = 0011 1101 \*/

System.out.println("a | b = " + c );

c = a ^ b; /\* 49 = 0011 0001 \*/

System.out.println("a ^ b = " + c );

c = ~a; /\*-61 = 1100 0011 \*/

System.out.println("~a = " + c );

c = a << 2; /\* 240 = 1111 0000 \*/

System.out.println("a << 2 = " + c );

c = a >> 2; /\* 15 = 1111 \*/

System.out.println("a >> 2 = " + c );

c = a >>> 2; /\* 15 = 0000 1111 \*/

System.out.println("a >>> 2 = " + c );

}

}

以上编译运行结果如下：

a & b = 12

a | b = 61

a ^ b = 49

~a = -61

a << 2 = 240

a >> 2 = 15

a >>> 2 = 15

**6 逻辑运算符**

（1）下表列出了逻辑运算符的基本运算，假设布尔变量A为真，变量B为假

|  |  |  |
| --- | --- | --- |
| **操作符** | **描述** | **例子** |
| **&&** | 称为逻辑与运算符。当且仅当两个操作数都为真，条件才为真。 | （A && B）为假。 |
| **| |** | 称为逻辑或操作符。如果任何两个操作数任何一个为真，条件为真。 | （A | | B）为真。 |
| **！** | 称为逻辑非运算符。用来反转操作数的逻辑状态。如果条件为true，则逻辑非运算符将得到false。 | ！（A && B）为真。 |

（2）下面的简单示例程序演示了逻辑运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

public class Test {

public static void main(String[] args) {

boolean a = true;

boolean b = false;

System.out.println("a && b = " + (a&&b));

System.out.println("a || b = " + (a||b) );

System.out.println("!(a && b) = " + !(a && b));

}

}

以上编译运行结果如下：

a && b = false

a || b = true

!(a && b) = true

**7 赋值运算符**

下面是Java语言支持的赋值运算符：

|  |  |  |
| --- | --- | --- |
| **操作符** | **描述** | **例子** |
| **=** | 简单的赋值运算符，将右操作数的值赋给左侧操作数 | C = A + B将把A + B得到的值赋给C |
| **+ =** | 加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数 | C + = A等价于C = C + A |
| **- =** | 减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数 | C - = A等价于C = C -  A |
| **\* =** | 乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数 | C \* = A等价于C = C \* A |
| **/ =** | 除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数 | C / = A等价于C = C / A |
| **（％）=** | 取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数 | C％= A等价于C = C％A |
| **<< =** | 左移位赋值运算符 | C << = 2等价于C = C << 2 |
| **>> =** | 右移位赋值运算符 | C >> = 2等价于C = C >> 2 |
| **＆=** | 按位与赋值运算符 | C＆= 2等价于C = C＆2 |
| **^ =** | 按位异或赋值操作符 | C ^ = 2等价于C = C ^ 2 |
| **| =** | 按位或赋值操作符 | C | = 2等价于C = C | 2 |

面的简单示例程序演示了赋值运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

Test.java 文件代码：

public class Test {

public static void main(String[] args) {

int a = 10;

int b = 20;

int c = 0;

c = a + b;

System.out.println("c = a + b = " + c );

c += a ;

System.out.println("c += a = " + c );

c -= a ;

System.out.println("c -= a = " + c );

c \*= a ;

System.out.println("c \*= a = " + c );

a = 10;

c = 15;

c /= a ;

System.out.println("c /= a = " + c );

a = 10;

c = 15;

c %= a ;

System.out.println("c %= a = " + c );

c <<= 2 ;

System.out.println("c <<= 2 = " + c );

c >>= 2 ;

System.out.println("c >>= 2 = " + c );

c >>= 2 ;

System.out.println("c >>= a = " + c );

c &= a ;

System.out.println("c &= 2 = " + c );

c ^= a ;

System.out.println("c ^= a = " + c );

c |= a ;

System.out.println("c |= a = " + c );

}

}

以上编译运行结果如下：

c = a + b = 30

c += a = 40

c -= a = 30

c \*= a = 300

c /= a = 1

c %= a = 5

c <<= 2 = 20

c >>= 2 = 5

c >>= 2 = 1

c &= a = 0

c ^= a = 10

c |= a = 10

**8 条件运算符（?:）**

条件运算符也被称为三元运算符。该运算符有3个操作数，并且需要判断布尔表达式的值。该运算符的主要是决定哪个值应该赋值给变量。

variable x = (expression) ? value if true : value if false

Test.java 文件代码：

public class Test {

public static void main(String[] args){

int a , b;

a = 10;

// 如果 a 等于 1 成立，则设置 b 为 20，否则为 30

b = (a == 1) ? 20 : 30;

System.out.println( "Value of b is : " + b );

// 如果 a 等于 10 成立，则设置 b 为 20，否则为 30

b = (a == 10) ? 20 : 30;

System.out.println( "Value of b is : " + b );

}

}

以上编译运行结果如下：

Value of b is : 30

Value of b is : 20

**9 instanceof 运算符**

该运算符用于操作对象，检查该对象是否是一个特定类型（类类型或接口类型）。

instanceof运算符使用格式如下：

( Object reference variable ) instanceof (class/interface type)

如果运算符左侧变量所指的对象，是操作符右侧类或接口(class/interface)的一个对象，那么结果为真。

String name = "James";

boolean result = name instanceof String; // 由于 name 是 String 类型，所以返回真

如果被比较的对象兼容于右侧类型,该运算符仍然返回true。

看下面的例子：

class Vehicle {}

public class Car extends Vehicle {

public static void main(String[] args){

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result);

}

}

以上编译运行结果如下：

true

**10 Java运算符优先级**

（1）当多个运算符出现在一个表达式中，谁先谁后呢？这就涉及到运算符的优先级别的问题。在一个多运算符的表达式中，运算符优先级不同会导致最后得出的结果差别甚大。

**例如，（1+3）＋（3+2）\*2，**这个表达式如果按加号最优先计算，答案就是 18，如果按照乘号最优先，答案则是 14。

**再如，x = 7 + 3 \* 2;**这里x得到13，而不是20，因为乘法运算符比加法运算符有较高的优先级，所以先计算3 \* 2得到6，然后再加7。

（2）下表中具有最高优先级的运算符在的表的最上面，最低优先级的在表的底部。

|  |  |  |
| --- | --- | --- |
| **类别** | **操作符** | **关联性** |
| **后缀** | () [] . (点操作符) | 左到右 |
| **一元** | + + - ！〜 | 从右到左 |
| **乘性** | \* /％ | 左到右 |
| **加性** | + - | 左到右 |
| **移位** | >> >>>  << | 左到右 |
| **关系** | >> = << = | 左到右 |
| **相等** | ==  != | 左到右 |
| **按位与** | ＆ | 左到右 |
| **按位异或** | ^ | 左到右 |
| **按位或** | | | 左到右 |
| **逻辑与** | && | 左到右 |
| **逻辑或** | | | | 左到右 |
| **条件** | ？： | 从右到左 |
| **赋值** | = + = - = \* = / =％= >> = << =＆= ^ = | = | 从右到左 |
| **逗号** | ， | 左到右 |

## 八、循环结构

**1 Java中有三种主要的循环结构：**

while 循环

do…while 循环

for 循环

在Java5中引入了一种主要用于数组的增强型for循环。

**2 while 循环**

while是最基本的循环，它的结构为：

while( 布尔表达式 ) {

//循环内容

}

只要布尔表达式为 true，循环就会一直执行下去。

Test.java 文件代码：

public class Test {

public static void main(String args[]) {

int x = 10;

while( x < 20 ) {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}

}

}

以上编译运行结果如下：

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**3 do…while 循环**

对于 while 语句而言，如果不满足条件，则不能进入循环。但有时候我们需要即使不满足条件，也至少执行一次。

do…while 循环和 while 循环相似，不同的是，do…while 循环至少会执行一次。

do {

//代码语句

}while(布尔表达式);

注意：布尔表达式在循环体的后面，所以语句块在检测布尔表达式之前已经执行了。 如果布尔表达式的值为 true，则语句块一直执行，直到布尔表达式的值为 false。

Test.java 文件代码：

public class Test {

public static void main(String args[]){

int x = 10;

do{

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}while( x < 20 );

}

}

以上编译运行结果如下：

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**4 for循环**

虽然所有循环结构都可以用 while 或者 do...while表示，但 Java 提供了另一种语句 —— for 循环，使一些循环结构变得更加简单。

for循环执行的次数是在执行前就确定的。语法格式如下：

for(初始化; 布尔表达式; 更新) {

//代码语句

}

关于 for 循环有以下几点说明：

最先执行初始化步骤。可以声明一种类型，但可初始化一个或多个循环控制变量，也可以是空语句。

然后，检测布尔表达式的值。如果为 true，循环体被执行。如果为false，循环终止，开始执行循环体后面的语句。

执行一次循环后，更新循环控制变量。

再次检测布尔表达式。循环执行上面的过程。

Test.java 文件代码：

public class Test {

public static void main(String args[]) {

for(int x = 10; x < 20; x = x+1) {

System.out.print("value of x : " + x );

System.out.print("\n");

}

}

}

以上编译运行结果如下：

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**5 Java 增强 for 循环**

Java5 引入了一种主要用于数组的增强型 for 循环。

Java 增强 for 循环语法格式如下:

for(声明语句 : 表达式)

{

//代码句子

}

声明语句：声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。

表达式：表达式是要访问的数组名，或者是返回值为数组的方法。

Test.java 文件代码：

public class Test {

public static void main(String args[]){

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ){

System.out.print( x );

System.out.print(",");

}

System.out.print("\n");

String [] names ={"James", "Larry", "Tom", "Lacy"};

for( String name : names ) {

System.out.print( name );

System.out.print(",");

}

}

}

以上编译运行结果如下：

10,20,30,40,50,

James,Larry,Tom,Lacy,

**6 break 关键字**

break 主要用在循环语句或者 switch 语句中，用来跳出整个语句块。

break 跳出最里层的循环，并且继续执行该循环下面的语句。

break 的用法很简单，就是循环结构中的一条语句：

break;

Test.java 文件代码：

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) {

// x 等于 30 时跳出循环

if( x == 30 ) {

break;

}

System.out.print( x );

System.out.print("\n");

}

}

}

以上编译运行结果如下：

10

20

**7 continue 关键字**

continue 适用于任何循环控制结构中。作用是让程序立刻跳转到下一次循环的迭代。

在 for 循环中，continue 语句使程序立即跳转到更新语句。

在 while 或者 do…while 循环中，程序立即跳转到布尔表达式的判断语句。

语法

continue 就是循环体中一条简单的语句：

continue;

Test.java 文件代码：

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) {

if( x == 30 ) {

continue;

}

System.out.print( x );

System.out.print("\n");

}

}

}

## 分支结构

**1 Java 有两种分支结构：**

if 语句

switch 语句

**2 if 语句**

一个 if 语句包含一个布尔表达式和一条或多条语句。

语法

if 语句的用语法如下：

if(布尔表达式)

{

//如果布尔表达式为true将执行的语句

}

如果布尔表达式的值为 true，则执行 if 语句中的代码块，否则执行 if 语句块后面的代码。

Test.java 文件代码：

public class Test {

public static void main(String args[]){

int x = 10;

if( x < 20 ){

System.out.print("这是 if 语句");

}

}

}

以上代码编译运行结果如下：

这是 if 语句

**3 if...else语句**

if 语句后面可以跟 else 语句，当 if 语句的布尔表达式值为 false 时，else 语句块会被执行。

语法

if…else 的用法如下：

if(布尔表达式){

//如果布尔表达式的值为true

}else{

//如果布尔表达式的值为false

}

Test.java 文件代码：

public class Test {

public static void main(String args[]){

int x = 30;

if( x < 20 ){

System.out.print("这是 if 语句");

}else{

System.out.print("这是 else 语句");

}

}

}

以上代码编译运行结果如下：

这是 else 语句

**4 if...else if...else 语句**

if 语句后面可以跟 elseif…else 语句，这种语句可以检测到多种可能的情况。

使用 if，else if，else 语句的时候，需要注意下面几点：

if 语句至多有 1 个 else 语句，else 语句在所有的 elseif 语句之后。

if 语句可以有若干个 elseif 语句，它们必须在 else 语句之前。

一旦其中一个 else if 语句检测为 true，其他的 else if 以及 else 语句都将跳过执行。

语法

if...else 语法格式如下:

if(布尔表达式 1){

//如果布尔表达式 1的值为true执行代码

}else if(布尔表达式 2){

//如果布尔表达式 2的值为true执行代码

}else if(布尔表达式 3){

//如果布尔表达式 3的值为true执行代码

}else {

//如果以上布尔表达式都不为true执行代码

}

Test.java 文件代码：

public class Test {

public static void main(String args[]){

int x = 30;

if( x == 10 ){

System.out.print("Value of X is 10");

}else if( x == 20 ){

System.out.print("Value of X is 20");

}else if( x == 30 ){

System.out.print("Value of X is 30");

}else{

System.out.print("这是 else 语句");

}

}

}

以上代码编译运行结果如下：

Value of X is 30

**5 嵌套的 if…else 语句**

使用嵌套的 if…else 语句是合法的。也就是说你可以在另一个 if 或者 elseif 语句中使用 if 或者 elseif 语句。

语法

嵌套的 if…else 语法格式如下：

if(布尔表达式 1){

////如果布尔表达式 1的值为true执行代码

if(布尔表达式 2){

////如果布尔表达式 2的值为true执行代码

}

}

你可以像 if 语句一样嵌套 else if...else。

Test.java 文件代码：

public class Test {

public static void main(String args[]){

int x = 30;

int y = 10;

if( x == 30 ){

if( y == 10 ){

System.out.print("X = 30 and Y = 10");

}

}

}

}

以上代码编译运行结果如下：

X = 30 and Y = 10

**6 switch 语句**

switch 语句判断一个变量与一系列值中某个值是否相等，每个值称为一个分支。

语法

switch 语法格式如下：

switch(expression){

case value :

//语句

break; //可选

case value :

//语句

break; //可选

//你可以有任意数量的case语句

default : //可选

//语句

}

switch 语句有如下规则：

switch 语句中的变量类型可以是： byte、short、int 或者 char。从 Java SE 7 开始，switch 支持字符串类型了，同时 case 标签必须为字符串常量或字面量。

switch 语句可以拥有多个 case 语句。每个 case 后面跟一个要比较的值和冒号。

case 语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量。

当变量的值与 case 语句的值相等时，那么 case 语句之后的语句开始执行，直到 break 语句出现才会跳出 switch 语句。

当遇到 break 语句时，switch 语句终止。程序跳转到 switch 语句后面的语句执行。case 语句不必须要包含 break 语句。如果没有 break 语句出现，程序会继续执行下一条 case 语句，直到出现 break 语句。

switch 语句可以包含一个 default 分支，该分支必须是 switch 语句的最后一个分支。default 在没有 case 语句的值和变量值相等的时候执行。default 分支不需要 break 语句。

Test.java 文件代码：

public class Test {

public static void main(String args[]){

//char grade = args[0].charAt(0);

char grade = 'C';

switch(grade)

{

case 'A' :

System.out.println("优秀");

break;

case 'B' :

case 'C' :

System.out.println("良好");

break;

case 'D' :

System.out.println("及格");

case 'F' :

System.out.println("你需要再努力努力");

break;

default :

System.out.println("未知等级");

}

System.out.println("你的等级是 " + grade);

}

}

以上代码编译运行结果如下：

良好

你的等级是 C

## 十、Java Number & Math 类

**1 一般地，当需要使用数字的时候，我们通常使用内置数据类型，如：byte、int、long、double 等。**

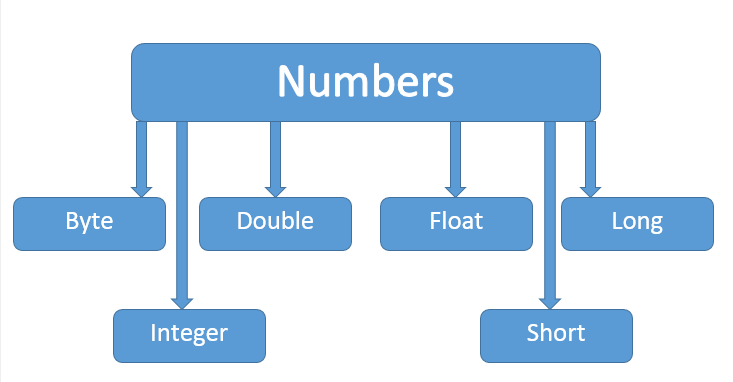
实例

int a = 5000;

float b = 13.65f;

byte c = 0x4a;

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情形。为了解决这个问题，Java 语言为每一个内置数据类型提供了对应的包装类。所有的包装类（Integer、Long、Byte、Double、Float、Short）都是抽象类 Number 的子类。



**2 Java Number类**

这种由编译器特别支持的包装称为装箱，所以当内置数据类型被当作对象使用的时候，编译器会把内置类型装箱为包装类。相似的，编译器也可以把一个对象拆箱为内置类型。Number 类属于 java.lang 包。

下面是一个使用 Integer 对象的实例：

Test.java 文件代码：

public class Test{

public static void main(String args[]){

Integer x = 5;

x = x + 10;

System.out.println(x);

}

}

以上实例编译运行结果如下：

15

当 x 被赋为整型值时，由于x是一个对象，所以编译器要对x进行装箱。然后，为了使x能进行加运算，所以要对x进行拆箱。

**3 Java Math 类**

Java 的 Math 包含了用于执行基本数学运算的属性和方法，如初等指数、对数、平方根和三角函数。Math 的方法都被定义为 static 形式，通过 Math 类可以在主函数中直接调用。

Test.java 文件代码：

public class Test {

public static void main (String []args)

{

System.out.println("90 度的正弦值：" + Math.sin(Math.PI/2));

System.out.println("0度的余弦值：" + Math.cos(0));

System.out.println("60度的正切值：" + Math.tan(Math.PI/3));

System.out.println("1的反正切值： " + Math.atan(1));

System.out.println("π/2的角度值：" + Math.toDegrees(Math.PI/2));

System.out.println(Math.PI);

}

}

以上实例编译运行结果如下：

90 度的正弦值：1.0

0度的余弦值：1.0

60度的正切值：1.7320508075688767

1的反正切值： 0.7853981633974483

π/2的角度值：90.0

3.141592653589793

**4 Number & Math 类方法**

下面的表中列出的是 Number & Math 类常用的一些方法：

|  |  |
| --- | --- |
| **序号** | **方法与描述** |
| 1 | [xxxValue()](http://www.runoob.com/java/number-xxxvalue.html" \t "http://www.runoob.com/java/_blank) 将 Number 对象转换为xxx数据类型的值并返回。 |
| 2 | [compareTo()](http://www.runoob.com/java/number-compareto.html" \t "http://www.runoob.com/java/_blank) 将number对象与参数比较。 |
| 3 | [equals()](http://www.runoob.com/java/number-equals.html" \t "http://www.runoob.com/java/_blank) 判断number对象是否与参数相等。 |
| 4 | [valueOf()](http://www.runoob.com/java/number-valueof.html" \t "http://www.runoob.com/java/_blank) 返回一个 Number 对象指定的内置数据类型 |
| 5 | [toString()](http://www.runoob.com/java/number-tostring.html" \t "http://www.runoob.com/java/_blank) 以字符串形式返回值。 |
| 6 | [parseInt()](http://www.runoob.com/java/number-parseInt.html" \t "http://www.runoob.com/java/_blank) 将字符串解析为int类型。 |
| 7 | [abs()](http://www.runoob.com/java/number-abs.html" \t "http://www.runoob.com/java/_blank) 返回参数的绝对值。 |
| 8 | [ceil()](http://www.runoob.com/java/number-ceil.html" \t "http://www.runoob.com/java/_blank) 返回大于等于( >= )给定参数的的最小整数。 |
| 9 | [floor()](http://www.runoob.com/java/number-floor.html" \t "http://www.runoob.com/java/_blank) 返回小于等于（<=）给定参数的最大整数 。 |
| 10 | [rint()](http://www.runoob.com/java/number-rint.html" \t "http://www.runoob.com/java/_blank) 返回与参数最接近的整数。返回类型为double。 |
| 11 | [round()](http://www.runoob.com/java/number-round.html" \t "http://www.runoob.com/java/_blank) 它表示四舍五入，算法为 Math.floor(x+0.5)，即将原来的数字加上 0.5 后再向下取整，所以，Math.round(11.5) 的结果为12，Math.round(-11.5) 的结果为-11。 |
| 12 | [min()](http://www.runoob.com/java/number-min.html" \t "http://www.runoob.com/java/_blank) 返回两个参数中的最小值。 |
| 13 | [max()](http://www.runoob.com/java/number-max.html" \t "http://www.runoob.com/java/_blank) 返回两个参数中的最大值。 |
| 14 | [exp()](http://www.runoob.com/java/number-exp.html" \t "http://www.runoob.com/java/_blank) 返回自然数底数e的参数次方。 |
| 15 | [log()](http://www.runoob.com/java/number-log.html" \t "http://www.runoob.com/java/_blank) 返回参数的自然数底数的对数值。 |
| 16 | [pow()](http://www.runoob.com/java/number-pow.html" \t "http://www.runoob.com/java/_blank) 返回第一个参数的第二个参数次方。 |
| 17 | [sqrt()](http://www.runoob.com/java/number-sqrt.html" \t "http://www.runoob.com/java/_blank) 求参数的算术平方根。 |
| 18 | [sin()](http://www.runoob.com/java/number-sin.html" \t "http://www.runoob.com/java/_blank) 求指定double类型参数的正弦值。 |
| 19 | [cos()](http://www.runoob.com/java/number-cos.html" \t "http://www.runoob.com/java/_blank) 求指定double类型参数的余弦值。 |
| 20 | [tan()](http://www.runoob.com/java/number-tan.html" \t "http://www.runoob.com/java/_blank) 求指定double类型参数的正切值。 |
| 21 | [asin()](http://www.runoob.com/java/number-asin.html" \t "http://www.runoob.com/java/_blank) 求指定double类型参数的反正弦值。 |
| 22 | [acos()](http://www.runoob.com/java/number-acos.html" \t "http://www.runoob.com/java/_blank) 求指定double类型参数的反余弦值。 |
| 23 | [atan()](http://www.runoob.com/java/number-atan.html" \t "http://www.runoob.com/java/_blank) 求指定double类型参数的反正切值。 |
| 24 | [atan2()](http://www.runoob.com/java/number-atan2.html" \t "http://www.runoob.com/java/_blank) 将笛卡尔坐标转换为极坐标，并返回极坐标的角度值。 |
| 25 | [toDegrees()](http://www.runoob.com/java/number-todegrees.html" \t "http://www.runoob.com/java/_blank) 将参数转化为角度。 |
| 26 | [toRadians()](http://www.runoob.com/java/number-toradians.html" \t "http://www.runoob.com/java/_blank) 将角度转换为弧度。 |
| 27 | [random()](http://www.runoob.com/java/number-random.html" \t "http://www.runoob.com/java/_blank) 返回一个随机数。 |

**5 Math 的 floor,round 和 ceil 方法实例比较**

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **Math.floor** | **Math.round** | **Math.ceil** |
| 1.4 | 1 | 1 | 2 |
| 1.5 | 1 | 2 | 2 |
| 1.6 | 1 | 2 | 2 |
| -1.4 | -2 | -1 | -1 |
| -1.5 | -2 | -1 | -1 |
| -1.6 | -2 | -2 | -1 |

floor,round 和 ceil 实例：

public class Main {

public static void main(String[] args) {

double[] nums = { 1.4, 1.5, 1.6, -1.4, -1.5, -1.6 };

for (double num : nums) {

test(num);

}

}

private static void test(double num) {

System.out.println("Math.floor(" + num + ")=" + Math.floor(num));

System.out.println("Math.round(" + num + ")=" + Math.round(num));

System.out.println("Math.ceil(" + num + ")=" + Math.ceil(num));

}

}

以上实例执行输出结果为：

Math.floor(1.4)=1.0

Math.round(1.4)=1

Math.ceil(1.4)=2.0

Math.floor(1.5)=1.0

Math.round(1.5)=2

Math.ceil(1.5)=2.0

Math.floor(1.6)=1.0

Math.round(1.6)=2

Math.ceil(1.6)=2.0

Math.floor(-1.4)=-2.0

Math.round(-1.4)=-1

Math.ceil(-1.4)=-1.0

Math.floor(-1.5)=-2.0

Math.round(-1.5)=-1

Math.ceil(-1.5)=-1.0

Math.floor(-1.6)=-2.0

Math.round(-1.6)=-2

Math.ceil(-1.6)=-1.0

## 十一、Java String 类

**1 字符串广泛应用 在Java 编程中，在 Java 中字符串属于对象**，Java 提供了 String 类来创建和操作字符串。

**2 创建字符串**

创建字符串最简单的方式如下:

String greeting = "菜鸟教程";

在代码中遇到字符串常量时，这里的值是 "菜鸟教程""，编译器会使用该值创建一个 String 对象。和其它对象一样，可以使用关键字和构造方法来创建 String 对象。

String 类有 11 种构造方法，这些方法提供不同的参数来初始化字符串，比如提供一个字符数组参数:

StringDemo.java 文件代码：

public class StringDemo{

public static void main(String args[]){

char[] helloArray = { 'r', 'u', 'n', 'o', 'o', 'b'};

String helloString = new String(helloArray);

System.out.println( helloString );

}

}

以上实例编译运行结果如下：

runoob

注意:String 类是不可改变的，所以你一旦创建了 String 对象，那它的值就无法改变了（详看笔记部分解析）。

如果需要对字符串做很多修改，那么应该选择使用 StringBuffer & StringBuilder 类。

**3 字符串长度**

用于获取有关对象的信息的方法称为访问器方法。

String 类的一个访问器方法是 length() 方法，它返回字符串对象包含的字符数。

下面的代码执行后，len变量等于14:

StringDemo.java 文件代码：

public class StringDemo {

public static void main(String args[]) {

String site = "www.runoob.com";

int len = site.length();

System.out.println( "菜鸟教程网址长度 : " + len );

}

}

以上实例编译运行结果如下：

菜鸟教程网址长度 : 14

**4 连接字符串**

String 类提供了连接两个字符串的方法：

string1.concat(string2);

返回 string2 连接 string1 的新字符串。也可以对字符串常量使用 concat() 方法，如：

"我的名字是 ".concat("Runoob");

更常用的是使用'+'操作符来连接字符串，如：

"Hello," + " runoob" + "!"

结果如下:

"Hello, runoob!"

下面是一个例子:

StringDemo.java 文件代码：

public class StringDemo {

public static void main(String args[]) {

String string1 = "菜鸟教程网址：";

System.out.println("1、" + string1 + "www.runoob.com");

}

}

以上实例编译运行结果如下：

1、菜鸟教程网址：www.runoob.com

**5 创建格式化字符串**

我们知道输出格式化数字可以使用 printf() 和 format() 方法。

String 类使用静态方法 format() 返回一个String 对象而不是 PrintStream 对象。

String 类的静态方法 format() 能用来创建可复用的格式化字符串，而不仅仅是用于一次打印输出。

如下所示：

System.out.printf("浮点型变量的值为 " +

"%f, 整型变量的值为 " +

" %d, 字符串变量的值为 " +

"is %s", floatVar, intVar, stringVar);

你也可以这样写

String fs;

fs = String.format("浮点型变量的值为 " +

"%f, 整型变量的值为 " +

" %d, 字符串变量的值为 " +

" %s", floatVar, intVar, stringVar);

**6 String 方法**

|  |  |
| --- | --- |
| **SN(序号)** | **方法描述** |
| 1 | [char charAt(int index)](http://www.runoob.com/java/java-string-charat.html" \t "http://www.runoob.com/java/_blank) 返回指定索引处的 char 值。 |
| 2 | [int compareTo(Object o)](http://www.runoob.com/java/java-string-compareto.html" \t "http://www.runoob.com/java/_blank) 把这个字符串和另一个对象比较。 |
| 3 | [int compareTo(String anotherString)](http://www.runoob.com/java/java-string-compareto.html" \t "http://www.runoob.com/java/_blank) 按字典顺序比较两个字符串。 |
| 4 | [int compareToIgnoreCase(String str)](http://www.runoob.com/java/java-string-comparetoignorecase.html" \t "http://www.runoob.com/java/_blank) 按字典顺序比较两个字符串，不考虑大小写。 |
| 5 | [String concat(String str)](http://www.runoob.com/java/java-string-concat.html" \t "http://www.runoob.com/java/_blank) 将指定字符串连接到此字符串的结尾。 |
| 6 | [boolean contentEquals(StringBuffer sb)](http://www.runoob.com/java/java-string-contentequals.html" \t "http://www.runoob.com/java/_blank) 当且仅当字符串与指定的StringBuffer有相同顺序的字符时候返回真。 |
| 7 | [static String copyValueOf(char[] data)](http://www.runoob.com/java/java-string-copyvalueof.html" \t "http://www.runoob.com/java/_blank) 返回指定数组中表示该字符序列的 String。 |
| 8 | [static String copyValueOf(char[] data, int offset, int count)](http://www.runoob.com/java/java-string-copyvalueof.html" \t "http://www.runoob.com/java/_blank) 返回指定数组中表示该字符序列的 String。 |
| 9 | [boolean endsWith(String suffix)](http://www.runoob.com/java/java-string-endswith.html" \t "http://www.runoob.com/java/_blank) 测试此字符串是否以指定的后缀结束。 |
| 10 | [boolean equals(Object anObject)](http://www.runoob.com/java/java-string-equals.html" \t "http://www.runoob.com/java/_blank) 将此字符串与指定的对象比较。 |
| 11 | [boolean equalsIgnoreCase(String anotherString)](http://www.runoob.com/java/java-string-equalsignorecase.html" \t "http://www.runoob.com/java/_blank) 将此 String 与另一个 String 比较，不考虑大小写。 |
| 12 | [byte[] getBytes()](http://www.runoob.com/java/java-string-getbytes.html" \t "http://www.runoob.com/java/_blank)  使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。 |
| 13 | [byte[] getBytes(String charsetName)](http://www.runoob.com/java/java-string-getbytes.html" \t "http://www.runoob.com/java/_blank) 使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。 |
| 14 | [void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)](http://www.runoob.com/java/java-string-getchars.html" \t "http://www.runoob.com/java/_blank) 将字符从此字符串复制到目标字符数组。 |
| 15 | [int hashCode()](http://www.runoob.com/java/java-string-hashcode.html" \t "http://www.runoob.com/java/_blank) 返回此字符串的哈希码。 |
| 16 | [int indexOf(int ch)](http://www.runoob.com/java/java-string-indexof.html" \t "http://www.runoob.com/java/_blank) 返回指定字符在此字符串中第一次出现处的索引。 |
| 17 | [int indexOf(int ch, int fromIndex)](http://www.runoob.com/java/java-string-indexof.html" \t "http://www.runoob.com/java/_blank) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。 |
| 18 | [int indexOf(String str)](http://www.runoob.com/java/java-string-indexof.html" \t "http://www.runoob.com/java/_blank)  返回指定子字符串在此字符串中第一次出现处的索引。 |
| 19 | [int indexOf(String str, int fromIndex)](http://www.runoob.com/java/java-string-indexof.html" \t "http://www.runoob.com/java/_blank) 返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。 |
| 20 | [String intern()](http://www.runoob.com/java/java-string-intern.html" \t "http://www.runoob.com/java/_blank)  返回字符串对象的规范化表示形式。 |
| 21 | [int lastIndexOf(int ch)](http://www.runoob.com/java/java-string-lastindexof.html" \t "http://www.runoob.com/java/_blank)  返回指定字符在此字符串中最后一次出现处的索引。 |
| 22 | [int lastIndexOf(int ch, int fromIndex)](http://www.runoob.com/java/java-string-lastindexof.html" \t "http://www.runoob.com/java/_blank) 返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。 |
| 23 | [int lastIndexOf(String str)](http://www.runoob.com/java/java-string-lastindexof.html" \t "http://www.runoob.com/java/_blank) 返回指定子字符串在此字符串中最右边出现处的索引。 |
| 24 | [int lastIndexOf(String str, int fromIndex)](http://www.runoob.com/java/java-string-lastindexof.html" \t "http://www.runoob.com/java/_blank)  返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。 |
| 25 | [int length()](http://www.runoob.com/java/java-string-length.html" \t "http://www.runoob.com/java/_blank) 返回此字符串的长度。 |
| 26 | [boolean matches(String regex)](http://www.runoob.com/java/java-string-matches.html" \t "http://www.runoob.com/java/_blank) 告知此字符串是否匹配给定的正则表达式。 |
| 27 | [boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)](http://www.runoob.com/java/java-string-regionmatches.html" \t "http://www.runoob.com/java/_blank) 测试两个字符串区域是否相等。 |
| 28 | [boolean regionMatches(int toffset, String other, int ooffset, int len)](http://www.runoob.com/java/java-string-regionmatches.html" \t "http://www.runoob.com/java/_blank) 测试两个字符串区域是否相等。 |
| 29 | [String replace(char oldChar, char newChar)](http://www.runoob.com/java/java-string-replace.html" \t "http://www.runoob.com/java/_blank) 返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。 |
| 30 | [String replaceAll(String regex, String replacement)](http://www.runoob.com/java/java-string-replaceall.html" \t "http://www.runoob.com/java/_blank) 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。 |
| 31 | [String replaceFirst(String regex, String replacement)](http://www.runoob.com/java/java-string-replacefirst.html" \t "http://www.runoob.com/java/_blank)  使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。 |
| 32 | [String[] split(String regex)](http://www.runoob.com/java/java-string-split.html" \t "http://www.runoob.com/java/_blank) 根据给定正则表达式的匹配拆分此字符串。 |
| 33 | [String[] split(String regex, int limit)](http://www.runoob.com/java/java-string-split.html" \t "http://www.runoob.com/java/_blank) 根据匹配给定的正则表达式来拆分此字符串。 |
| 34 | [boolean startsWith(String prefix)](http://www.runoob.com/java/java-string-startswith.html" \t "http://www.runoob.com/java/_blank) 测试此字符串是否以指定的前缀开始。 |
| 35 | [boolean startsWith(String prefix, int toffset)](http://www.runoob.com/java/java-string-startswith.html" \t "http://www.runoob.com/java/_blank) 测试此字符串从指定索引开始的子字符串是否以指定前缀开始。 |
| 36 | [CharSequence subSequence(int beginIndex, int endIndex)](http://www.runoob.com/java/java-string-subsequence.html" \t "http://www.runoob.com/java/_blank)  返回一个新的字符序列，它是此序列的一个子序列。 |
| 37 | [String substring(int beginIndex)](http://www.runoob.com/java/java-string-substring.html" \t "http://www.runoob.com/java/_blank) 返回一个新的字符串，它是此字符串的一个子字符串。 |
| 38 | [String substring(int beginIndex, int endIndex)](http://www.runoob.com/java/java-string-substring.html" \t "http://www.runoob.com/java/_blank) 返回一个新字符串，它是此字符串的一个子字符串。 |
| 39 | [char[] toCharArray()](http://www.runoob.com/java/java-string-tochararray.html" \t "http://www.runoob.com/java/_blank) 将此字符串转换为一个新的字符数组。 |
| 40 | [String toLowerCase()](http://www.runoob.com/java/java-string-tolowercase.html" \t "http://www.runoob.com/java/_blank) 使用默认语言环境的规则将此 String 中的所有字符都转换为小写。 |
| 41 | [String toLowerCase(Locale locale)](http://www.runoob.com/java/java-string-tolowercase.html" \t "http://www.runoob.com/java/_blank)  使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。 |
| 42 | [String toString()](http://www.runoob.com/java/java-string-tostring.html" \t "http://www.runoob.com/java/_blank)  返回此对象本身（它已经是一个字符串！）。 |
| 43 | [String toUpperCase()](http://www.runoob.com/java/java-string-touppercase.html" \t "http://www.runoob.com/java/_blank) 使用默认语言环境的规则将此 String 中的所有字符都转换为大写。 |
| 44 | [String toUpperCase(Locale locale)](http://www.runoob.com/java/java-string-touppercase.html" \t "http://www.runoob.com/java/_blank) 使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。 |
| 45 | [String trim()](http://www.runoob.com/java/java-string-trim.html" \t "http://www.runoob.com/java/_blank) 返回字符串的副本，忽略前导空白和尾部空白。 |
| 46 | [static String valueOf(primitive data type x)](http://www.runoob.com/java/java-string-valueof.html" \t "http://www.runoob.com/java/_blank) 返回给定data type类型x参数的字符串表示形式。 |

**十二、Java StringBuffer 和 StringBuilder 类**

**1 当对字符串进行修改的时候，需要使用 StringBuffer 和 StringBuilder 类。**

和 String 类不同的是，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。

StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。

**2 由于 StringBuilder 相较于 StringBuffer 有速度优势**，所以多数情况下建议使用 StringBuilder 类。然而在应用程序要求线程安全的情况下，则必须使用 StringBuffer 类。

Test.java 文件代码：

public class Test{

public static void main(String args[]){

StringBuffer sBuffer = new StringBuffer("菜鸟教程官网：");

sBuffer.append("www");

sBuffer.append(".runoob");

sBuffer.append(".com");

System.out.println(sBuffer);

}

}

以上实例编译运行结果如下：

菜鸟教程官网：www.runoob.com

**3 StringBuffer 方法**

以下是 StringBuffer 类支持的主要方法：

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | public StringBuffer append(String s) 将指定的字符串追加到此字符序列。 |
| 2 | public StringBuffer reverse()  将此字符序列用其反转形式取代。 |
| 3 | public delete(int start, int end) 移除此序列的子字符串中的字符。 |
| 4 | public insert(int offset, int i) 将 int 参数的字符串表示形式插入此序列中。 |
| 5 | replace(int start, int end, String str) 使用给定 String 中的字符替换此序列的子字符串中的字符。 |

**4 下面的列表里的方法和 String 类的方法类似：**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | int capacity() 返回当前容量。 |
| 2 | char charAt(int index) 返回此序列中指定索引处的 char 值。 |
| 3 | void ensureCapacity(int minimumCapacity) 确保容量至少等于指定的最小值。 |
| 4 | void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此序列复制到目标字符数组 dst。 |
| 5 | int indexOf(String str) 返回第一次出现的指定子字符串在该字符串中的索引。 |
| 6 | int indexOf(String str, int fromIndex) 从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。 |
| 7 | int lastIndexOf(String str) 返回最右边出现的指定子字符串在此字符串中的索引。 |
| 8 | int lastIndexOf(String str, int fromIndex) 返回 String 对象中子字符串最后出现的位置。 |
| 9 | int length()  返回长度（字符数）。 |
| 10 | void setCharAt(int index, char ch) 将给定索引处的字符设置为 ch。 |
| 11 | void setLength(int newLength) 设置字符序列的长度。 |
| 12 | CharSequence subSequence(int start, int end) 返回一个新的字符序列，该字符序列是此序列的子序列。 |
| 13 | String substring(int start) 返回一个新的 String，它包含此字符序列当前所包含的字符子序列。 |
| 14 | String substring(int start, int end) 返回一个新的 String，它包含此序列当前所包含的字符子序列。 |
| 15 | String toString() 返回此序列中数据的字符串表示形式。 |

## 十三、Java 日期时间

**1 java.util 包提供了 Date 类来封装当前的日期和时间。 Date 类提供两个构造函数来化 Date 对象。**

第一个构造函数使用当前日期和时间来初始化对象。

Date( )

第二个构造函数接收一个参数，该参数是从1970年1月1日起的毫秒数。

Date(long millisec)

**2 Date对象创建以后，可以调用下面的方法。**

|  |  |
| --- | --- |
| **序号** | **方法和描述** |
| 1 | boolean after(Date date) 若当调用此方法的Date对象在指定日期之后返回true,否则返回false。 |
| 2 | boolean before(Date date) 若当调用此方法的Date对象在指定日期之前返回true,否则返回false。 |
| 3 | Object clone( ) 返回此对象的副本。 |
| 4 | int compareTo(Date date) 比较当调用此方法的Date对象和指定日期。两者相等时候返回0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数。 |
| 5 | int compareTo(Object obj) 若obj是Date类型则操作等同于compareTo(Date) 。否则它抛出ClassCastException。 |
| 6 | boolean equals(Object date) 当调用此方法的Date对象和指定日期相等时候返回true,否则返回false。 |
| 7 | long getTime( ) 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。 |
| 8 | int hashCode( )  返回此对象的哈希码值。 |
| 9 | void setTime(long time)   用自1970年1月1日00:00:00 GMT以后time毫秒数设置时间和日期。 |
| 10 | String toString( ) 把此 Date 对象转换为以下形式的 String： dow mon dd hh:mm:ss zzz yyyy 其中： dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)。 |

**3 获取当前日期时间**

Java中获取当前日期和时间很简单，使用 Date 对象的 toString() 方法来打印当前日期和时间，如下所示：

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// 初始化 Date 对象

Date date = new Date();

// 使用 toString() 函数显示日期时间

System.out.println(date.toString());

}

}

运行 »

以上编译运行结果如下:

Mon May 04 09:51:52 CDT 2013

**4 日期比较**

**使用 getTime() 方法获取两个日期**（自1970年1月1日经历的毫秒数值），然后比较这两个值。

**使用方法 before()，after() 和 equals()。**例如，一个月的12号比18号早，则 new Date(99, 2, 12).before(new Date (99, 2, 18)) 返回true。

**使用 compareTo() 方法，**它是由 Comparable 接口定义的，Date 类实现了这个接口。

**5 使用 SimpleDateFormat 格式化日期**

SimpleDateFormat 是一个以语言环境敏感的方式来格式化和分析日期的类。SimpleDateFormat 允许你选择任何用户自定义日期时间格式来运行。例如：

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

Date dNow = new Date( );

SimpleDateFormat ft = new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

System.out.println("Current Date: " + ft.format(dNow));

}

}

运行 »

SimpleDateFormat ft = new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

这一行代码确立了转换的格式，其中 yyyy 是完整的公元年，MM 是月份，dd 是日期，HH:mm:ss 是时、分、秒。注意:有的格式大写，有的格式小写，例如 MM 是月份，mm 是分；HH 是 24 小时制，而 hh 是 12 小时制。

以上编译运行结果如下:

Current Date: Wed 2016.11.09 at 08:23:19 AM UTC

**6 日期和时间的格式化编码**

时间模式字符串用来指定时间格式。在此模式中，所有的 ASCII 字母被保留为模式字母，定义如下：

|  |  |  |
| --- | --- | --- |
| **字母** | **描述** | **示例** |
| G | 纪元标记 | AD |
| y | 四位年份 | 2001 |
| M | 月份 | July or 07 |
| d | 一个月的日期 | 10 |
| h | A.M./P.M. (1~12)格式小时 | 12 |
| H | 一天中的小时 (0~23) | 22 |
| m | 分钟数 | 30 |
| s | 秒数 | 55 |
| S | 毫秒数 | 234 |
| E | 星期几 | Tuesday |
| D | 一年中的日子 | 360 |
| F | 一个月中第几周的周几 | 2 (second Wed. in July) |
| w | 一年中第几周 | 40 |
| W | 一个月中第几周 | 1 |
| a | A.M./P.M. 标记 | PM |
| k | 一天中的小时(1~24) | 24 |
| K | A.M./P.M. (0~11)格式小时 | 10 |
| z | 时区 | Eastern Standard Time |
| ' | 文字定界符 | Delimiter |
| " | 单引号 | ` |

**7 使用printf格式化日期**

printf 方法可以很轻松地格式化时间和日期。使用两个字母格式，它以 %t 开头并且以下面表格中的一个字母结尾。

|  |  |  |
| --- | --- | --- |
| **转  换  符** | **说    明** | **示    例** |
| c | 包括全部日期和时间信息 | 星期六 十月 27 14:21:20 CST 2007 |
| F | "年-月-日"格式 | 2007-10-27 |
| D | "月/日/年"格式 | 10/27/07 |
| r | "HH:MM:SS PM"格式（12时制） | 02:25:51 下午 |
| T | "HH:MM:SS"格式（24时制） | 14:28:16 |
| R | "HH:MM"格式（24时制） | 14:28 |

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// 初始化 Date 对象

Date date = new Date();

//c的使用

System.out.printf("全部日期和时间信息：%tc%n",date);

//f的使用

System.out.printf("年-月-日格式：%tF%n",date);

//d的使用

System.out.printf("月/日/年格式：%tD%n",date);

//r的使用

System.out.printf("HH:MM:SS PM格式（12时制）：%tr%n",date);

//t的使用

System.out.printf("HH:MM:SS格式（24时制）：%tT%n",date);

//R的使用

System.out.printf("HH:MM格式（24时制）：%tR",date);

}

}

运行 »

以上编译运行结果如下:

全部日期和时间信息：星期一 九月 10 10:43:36 CST 2012

年-月-日格式：2012-09-10

月/日/年格式：09/10/12

HH:MM:SS PM格式（12时制）：10:43:36 上午

HH:MM:SS格式（24时制）：10:43:36

HH:MM格式（24时制）：10:43

**8 解析字符串为时间**

SimpleDateFormat 类有一些附加的方法，特别是parse()，它试图按照给定的SimpleDateFormat 对象的格式化存储来解析字符串。例如：

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

String input = args.length == 0 ? "1818-11-11" : args[0];

System.out.print(input + " Parses as ");

Date t;

try {

t = ft.parse(input);

System.out.println(t);

} catch (ParseException e) {

System.out.println("Unparseable using " + ft);

}

}

}

运行 »

以上编译运行结果如下:

$ java DateDemo

1818-11-11 Parses as Wed Nov 11 00:00:00 GMT 1818

$ java DateDemo 2007-12-01

2007-12-01 Parses as Sat Dec 01 00:00:00 GMT 2007

**9 Calendar类**

我们现在已经能够格式化并创建一个日期对象了，但是我们如何才能设置和获取日期数据的特定部分呢，比如说小时，日，或者分钟? 我们又如何在日期的这些部分加上或者减去值呢? 答案是使用Calendar 类。Calendar类的功能要比Date类强大很多，而且在实现方式上也比Date类要复杂一些。Calendar类是一个抽象类，在实际使用时实现特定的子类的对象，创建对象的过程对程序员来说是透明的，只需要使用getInstance方法创建即可。

创建一个代表系统当前日期的Calendar对象

Calendar c = Calendar.getInstance();//默认是当前日期

创建一个指定日期的Calendar对象

使用Calendar类代表特定的时间，需要首先创建一个Calendar的对象，然后再设定该对象中的年月日参数来完成。

//创建一个代表2009年6月12日的Calendar对象

Calendar c1 = Calendar.getInstance();

c1.set(2009, 6 - 1, 12);

**10 Calendar类对象字段类型**

Calendar类中用以下这些常量表示不同的意义，jdk内的很多类其实都是采用的这种思想

|  |  |
| --- | --- |
| **常量** | **描述** |
| Calendar.YEAR | 年份 |
| Calendar.MONTH | 月份 |
| Calendar.DATE | 日期 |
| Calendar.DAY\_OF\_MONTH | 日期，和上面的字段意义完全相同 |
| Calendar.HOUR | 12小时制的小时 |
| Calendar.HOUR\_OF\_DAY | 24小时制的小时 |
| Calendar.MINUTE | 分钟 |
| Calendar.SECOND | 秒 |
| Calendar.DAY\_OF\_WEEK | 星期几 |

**11 Calendar类对象信息的设置**

Set设置

Calendar c1 = Calendar.getInstance();

调用：

public final void set(int year,int month,int date)

c1.set(2009, 6 - 1, 12);//把Calendar对象c1的年月日分别设这为：2009、6、12

如果只设定某个字段，例如日期的值，则可以使用如下set方法：

public void set(int field,int value)

把 c1对象代表的日期设置为10号，其它所有的数值会被重新计算

c1.set(Calendar.DATE,10);

把c1对象代表的年份设置为2008年，其他的所有数值会被重新计算

c1.set(Calendar.YEAR,2008);

其他字段属性set的意义以此类推

Add设置

Calendar c1 = Calendar.getInstance();

把c1对象的日期加上10，也就是c1也就表示为10天后的日期，其它所有的数值会被重新计算

c1.add(Calendar.DATE, 10);

把c1对象的日期减去10，也就是c1也就表示为10天前的日期，其它所有的数值会被重新计算

c1.add(Calendar.DATE, -10);

**12 Calendar类对象信息的获得**

Calendar c1 = Calendar.getInstance();

// 获得年份

int year = c1.get(Calendar.YEAR);

// 获得月份

int month = c1.get(Calendar.MONTH) + 1;

// 获得日期

int date = c1.get(Calendar.DATE);

// 获得小时

int hour = c1.get(Calendar.HOUR\_OF\_DAY);

// 获得分钟

int minute = c1.get(Calendar.MINUTE);

// 获得秒

int second = c1.get(Calendar.SECOND);

// 获得星期几（注意（这个与Date类是不同的）：1代表星期日、2代表星期1、3代表星期二，以此类推）

int day = c1.get(Calendar.DAY\_OF\_WEEK);

**13 GregorianCalendar类**

Calendar类实现了公历日历，GregorianCalendar是Calendar类的一个具体实现。

Calendar 的getInstance（）方法返回一个默认用当前的语言环境和时区初始化的GregorianCalendar对象。GregorianCalendar定义了两个字段：AD和BC。这是代表公历定义的两个时代。面列出GregorianCalendar对象的几个构造方法：

|  |  |
| --- | --- |
| **序号** | **构造函数和说明** |
| 1 | GregorianCalendar()  在具有默认语言环境的默认时区内使用当前时间构造一个默认的 GregorianCalendar。 |
| 2 | GregorianCalendar(int year, int month, int date)  在具有默认语言环境的默认时区内构造一个带有给定日期设置的 GregorianCalendar |
| 3 | GregorianCalendar(int year, int month, int date, int hour, int minute)  为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。 |
| 4 | GregorianCalendar(int year, int month, int date, int hour, int minute, int second)    为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。 |
| 5 | GregorianCalendar(Locale aLocale)  在具有给定语言环境的默认时区内构造一个基于当前时间的 GregorianCalendar。 |
| 6 | GregorianCalendar(TimeZone zone)  在具有默认语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。 |
| 7 | GregorianCalendar(TimeZone zone, Locale aLocale)   在具有给定语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。 |

**14 这里是GregorianCalendar 类提供的一些有用的方法列表**

|  |  |
| --- | --- |
| **序号** | **方法和说明** |
| 1 | void add(int field, int amount)  根据日历规则，将指定的（有符号的）时间量添加到给定的日历字段中。 |
| 2 | protected void computeFields()  转换UTC毫秒值为时间域值 |
| 3 | protected void computeTime()  覆盖Calendar ，转换时间域值为UTC毫秒值 |
| 4 | boolean equals(Object obj)  比较此 GregorianCalendar 与指定的 Object。 |
| 5 | int get(int field)  获取指定字段的时间值 |
| 6 | int getActualMaximum(int field)  返回当前日期，给定字段的最大值 |
| 7 | int getActualMinimum(int field)  返回当前日期，给定字段的最小值 |
| 8 | int getGreatestMinimum(int field)   返回此 GregorianCalendar 实例给定日历字段的最高的最小值。 |
| 9 | Date getGregorianChange()  获得格里高利历的更改日期。 |
| 10 | int getLeastMaximum(int field)  返回此 GregorianCalendar 实例给定日历字段的最低的最大值 |
| 11 | int getMaximum(int field)  返回此 GregorianCalendar 实例的给定日历字段的最大值。 |
| 12 | Date getTime() 获取日历当前时间。 |
| 13 | long getTimeInMillis()  获取用长整型表示的日历的当前时间 |
| 14 | TimeZone getTimeZone()  获取时区。 |
| 15 | int getMinimum(int field)  返回给定字段的最小值。 |
| 16 | int hashCode()  重写hashCode. |
| 17 | boolean isLeapYear(int year) 确定给定的年份是否为闰年。 |
| 18 | void roll(int field, boolean up)  在给定的时间字段上添加或减去（上/下）单个时间单元，不更改更大的字段。 |
| 19 | void set(int field, int value)  用给定的值设置时间字段。 |
| 20 | void set(int year, int month, int date)  设置年、月、日的值。 |
| 21 | void set(int year, int month, int date, int hour, int minute)  设置年、月、日、小时、分钟的值。 |
| 22 | void set(int year, int month, int date, int hour, int minute, int second)  设置年、月、日、小时、分钟、秒的值。 |
| 23 | void setGregorianChange(Date date)  设置 GregorianCalendar 的更改日期。 |
| 24 | void setTime(Date date)  用给定的日期设置Calendar的当前时间。 |
| 25 | void setTimeInMillis(long millis)  用给定的long型毫秒数设置Calendar的当前时间。 |
| 26 | void setTimeZone(TimeZone value)  用给定时区值设置当前时区。 |
| 27 | String toString()  返回代表日历的字符串。 |

**15 实例分析**

import java.util.\*;

public class GregorianCalendarDemo {

public static void main(String args[]) {

String months[] = {

"Jan", "Feb", "Mar", "Apr",

"May", "Jun", "Jul", "Aug",

"Sep", "Oct", "Nov", "Dec"};

int year;

// 初始化 Gregorian 日历

// 使用当前时间和日期

// 默认为本地时间和时区

GregorianCalendar gcalendar = new GregorianCalendar();

// 显示当前时间和日期的信息

System.out.print("Date: ");

System.out.print(months[gcalendar.get(Calendar.MONTH)]);

System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");

System.out.println(year = gcalendar.get(Calendar.YEAR));

System.out.print("Time: ");

System.out.print(gcalendar.get(Calendar.HOUR) + ":");

System.out.print(gcalendar.get(Calendar.MINUTE) + ":");

System.out.println(gcalendar.get(Calendar.SECOND));

// 测试当前年份是否为闰年

if(gcalendar.isLeapYear(year)) {

System.out.println("当前年份是闰年");

}

else {

System.out.println("当前年份不是闰年");

}

}

}

运行 »

以上编译运行结果如下：

Date: Apr 22 2009

Time: 11:25:27

当前年份不是闰年

## Java 方法

**1 Java方法是语句的集合，它们在一起执行一个功能。**

方法是解决一类问题的步骤的有序组合

方法包含于类或对象中

方法在程序中被创建，在其他地方被引用

**2 方法的优点**

1. 使程序变得更简短而清晰。

2. 有利于程序维护。

3. 可以提高程序开发的效率。

4. 提高了代码的重用性。

**3 方法的命名规则**

1.方法的名字的第一个单词应以小写字母作为开头，后面的单词则用大写字母开头写，不使用连接符。例如：addPerson。

2.下划线可能出现在 JUnit 测试方法名称中用以分隔名称的逻辑组件。一个典型的模式是：test<MethodUnderTest>\_<state>，例如 testPop\_emptyStack。

**4 方法的定义**

一般情况下，定义一个方法包含以下语法：

修饰符 返回值类型 方法名(参数类型 参数名){

...

方法体

...

return 返回值;

}

**5 方法包含一个方法头和一个方法体。下面是一个方法的所有部分：**

修饰符：修饰符，这是可选的，告诉编译器如何调用该方法。定义了该方法的访问类型。

返回值类型 ：方法可能会返回值。returnValueType 是方法返回值的数据类型。有些方法执行所需的操作，但没有返回值。在这种情况下，returnValueType 是关键字void。

方法名：是方法的实际名称。方法名和参数表共同构成方法签名。

参数类型：参数像是一个占位符。当方法被调用时，传递值给参数。这个值被称为实参或变量。参数列表是指方法的参数类型、顺序和参数的个数。参数是可选的，方法可以不包含任何参数。

方法体：方法体包含具体的语句，定义该方法的功能。

如：

public static int age(int birthday){...}

参数可以有多个：

static float interest(float principal, int year){...}

注意： 在一些其它语言中方法指过程和函数。一个返回非void类型返回值的方法称为函数；一个返回void类型返回值的方法叫做过程。

下面的方法包含 2 个参数 num1 和 num2，它返回这两个参数的最大值。

/\*\* 返回两个整型变量数据的较大值 \*/

public static int max(int num1, int num2) {

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

**6 方法调用**

Java 支持两种调用方法的方式，根据方法是否返回值来选择。

当程序调用一个方法时，程序的控制权交给了被调用的方法。当被调用方法的返回语句执行或者到达方法体闭括号时候交还控制权给程序。

当方法返回一个值的时候，方法调用通常被当做一个值。例如：

int larger = max(30, 40);

如果方法返回值是void，方法调用一定是一条语句。例如，方法println返回void。下面的调用是个语句：

System.out.println("欢迎访问菜鸟教程！");

示例

下面的例子演示了如何定义一个方法，以及如何调用它：

TestMax.java 文件代码：

public class TestMax {

/\*\* 主方法 \*/

public static void main(String[] args) {

int i = 5;

int j = 2;

int k = max(i, j);

System.out.println( i + " 和 " + j + " 比较，最大值是：" + k);

}

/\*\* 返回两个整数变量较大的值 \*/

public static int max(int num1, int num2) {

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

}

以上实例编译运行结果如下：

5 和 2 比较，最大值是：5

这个程序包含 main 方法和 max 方法。main 方法是被 JVM 调用的，除此之外，main 方法和其它方法没什么区别。main 方法的头部是不变的，如例子所示，带修饰符 public 和 static,返回 void 类型值，方法名字是 main,此外带个一个 String[] 类型参数。String[] 表明参数是字符串数组。

**7 void 关键字**

本节说明如何声明和调用一个 void 方法。

下面的例子声明了一个名为 printGrade 的方法，并且调用它来打印给定的分数。

TestVoidMethod.java 文件代码：

public class TestVoidMethod {

public static void main(String[] args) {

printGrade(78.5);

}

public static void printGrade(double score) {

if (score >= 90.0) {

System.out.println('A');

}

else if (score >= 80.0) {

System.out.println('B');

}

else if (score >= 70.0) {

System.out.println('C');

}

else if (score >= 60.0) {

System.out.println('D');

}

else {

System.out.println('F');

}

}

}

以上实例编译运行结果如下：

C

这里printGrade方法是一个void类型方法，它不返回值。

一个void方法的调用一定是一个语句。 所以，它被在main方法第三行以语句形式调用。就像任何以分号结束的语句一样。

通过值传递参数

调用一个方法时候需要提供参数，你必须按照参数列表指定的顺序提供。

例如，下面的方法连续n次打印一个消息：

TestVoidMethod.java 文件代码：

public static void nPrintln(String message, int n) {

for (int i = 0; i < n; i++) {

System.out.println(message);

}

}

示例

下面的例子演示按值传递的效果。

该程序创建一个方法，该方法用于交换两个变量。

TestPassByValue.java 文件代码：

public class TestPassByValue {

public static void main(String[] args) {

int num1 = 1;

int num2 = 2;

System.out.println("交换前 num1 的值为：" +

num1 + " ，num2 的值为：" + num2);

// 调用swap方法

swap(num1, num2);

System.out.println("交换后 num1 的值为：" +

num1 + " ，num2 的值为：" + num2);

}

/\*\* 交换两个变量的方法 \*/

public static void swap(int n1, int n2) {

System.out.println("\t进入 swap 方法");

System.out.println("\t\t交换前 n1 的值为：" + n1

+ "，n2 的值：" + n2);

// 交换 n1 与 n2的值

int temp = n1;

n1 = n2;

n2 = temp;

System.out.println("\t\t交换后 n1 的值为 " + n1

+ "，n2 的值：" + n2);

}

}

**8 方法的重载**

上面使用的max方法仅仅适用于int型数据。但如果你想得到两个浮点类型数据的最大值呢？

解决方法是创建另一个有相同名字但参数不同的方法，如下面代码所示：

public static double max(double num1, double num2) {

if (num1 > num2)

return num1;

else

return num2;

}

如果你调用max方法时传递的是int型参数，则 int型参数的max方法就会被调用；如果传递的是double型参数，则double类型的max方法体会被调用，这叫做方法重载；就是说一个类的两个方法拥有相同的名字，但是有不同的参数列表。Java编译器根据方法签名判断哪个方法应该被调用。方法重载可以让程序更清晰易读。执行密切相关任务的方法应该使用相同的名字。重载的方法必须拥有不同的参数列表。你不能仅仅依据修饰符或者返回类型的不同来重载方法。

**9 变量作用域**

变量的范围是程序中该变量可以被引用的部分。

方法内定义的变量被称为局部变量。

局部变量的作用范围从声明开始，直到包含它的块结束。

局部变量必须声明才可以使用。

方法的参数范围涵盖整个方法。参数实际上是一个局部变量。

for循环的初始化部分声明的变量，其作用范围在整个循环。

但循环体内声明的变量其适用范围是从它声明到循环体结束。它包含如下所示的变量声明：

你可以在一个方法里，不同的非嵌套块中多次声明一个具有相同的名称局部变量，但你不能在嵌套块内两次声明局部变量。

**10 命令行参数的使用**

有时候你希望运行一个程序时候再传递给它消息。这要靠传递命令行参数给main()函数实现。

命令行参数是在执行程序时候紧跟在程序名字后面的信息。

下面的程序打印所有的命令行参数：

CommandLine.java 文件代码：

public class CommandLine {

public static void main(String args[]){

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

System.out.println("args[" + i + "]: " + args[i]);

}

}

}

如下所示，运行这个程序：

$ javac CommandLine.java

$ java CommandLine this is a command line 200 -100

args[0]: this

args[1]: is

args[2]: a

args[3]: command

args[4]: line

args[5]: 200

args[6]: -100

**11 构造方法**

当一个对象被创建时候，构造方法用来初始化该对象。构造方法和它所在类的名字相同，但构造方法没有返回值。通常会使用构造方法给一个类的实例变量赋初值，或者执行其它必要的步骤来创建一个完整的对象。不管你与否自定义构造方法，所有的类都有构造方法，因为Java自动提供了一个默认构造方法，它把所有成员初始化为0。一旦你定义了自己的构造方法，默认构造方法就会失效。下面是一个使用构造方法的例子：

// 一个简单的构造函数

class MyClass {

int x;

// 以下是构造函数

MyClass() {

x = 10;

}

}

你可以像下面这样调用构造方法来初始化一个对象：

ConsDemo.java 文件代码：

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

System.out.println(t1.x + " " + t2.x);

}

}

大多时候需要一个有参数的构造方法。

下面是一个使用构造方法的例子：

// 一个简单的构造函数

class MyClass {

int x;

// 以下是构造函数

MyClass(int i ) {

x = i;

}

}

你可以像下面这样调用构造方法来初始化一个对象：

ConsDemo.java 文件代码：

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass( 10 );

MyClass t2 = new MyClass( 20 );

System.out.println(t1.x + " " + t2.x);

}

}

运行结果如下：

10 20

**12 可变参数**

JDK 1.5 开始，Java支持传递同类型的可变参数给一个方法。

方法的可变参数的声明如下所示：

typeName... parameterName

在方法声明中，在指定参数类型后加一个省略号(...) 。

一个方法中只能指定一个可变参数，它必须是方法的最后一个参数。任何普通的参数必须在它之前声明。

VarargsDemo.java 文件代码：

public class VarargsDemo {

public static void main(String args[]) {

// 调用可变参数的方法

printMax(34, 3, 3, 2, 56.5);

printMax(new double[]{1, 2, 3});

}

public static void printMax( double... numbers) {

if (numbers.length == 0) {

System.out.println("No argument passed");

return;

}

double result = numbers[0];

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

if (numbers[i] > result) {

result = numbers[i];

}

}

System.out.println("The max value is " + result);

}

}

以上实例编译运行结果如下：

The max value is 56.5

The max value is 3.0

**13 finalize() 方法**

Java 允许定义这样的方法，它在对象被垃圾收集器析构(回收)之前调用，这个方法叫做 finalize( )，它用来清除回收对象。

例如，你可以使用 finalize() 来确保一个对象打开的文件被关闭了。

在 finalize() 方法里，你必须指定在对象销毁时候要执行的操作。

finalize() 一般格式是：

protected void finalize()

{

// 在这里终结代码

}

关键字 protected 是一个限定符，它确保 finalize() 方法不会被该类以外的代码调用。

当然，Java 的内存回收可以由 JVM 来自动完成。如果你手动使用，则可以使用上面的方法。

FinalizationDemo.java 文件代码：

public class FinalizationDemo {

public static void main(String[] args) {

Cake c1 = new Cake(1);

Cake c2 = new Cake(2);

Cake c3 = new Cake(3);

c2 = c3 = null;

System.gc(); //调用Java垃圾收集器

}

}

class Cake extends Object {

private int id;

public Cake(int id) {

this.id = id;

System.out.println("Cake Object " + id + "is created");

}

protected void finalize() throws java.lang.Throwable {

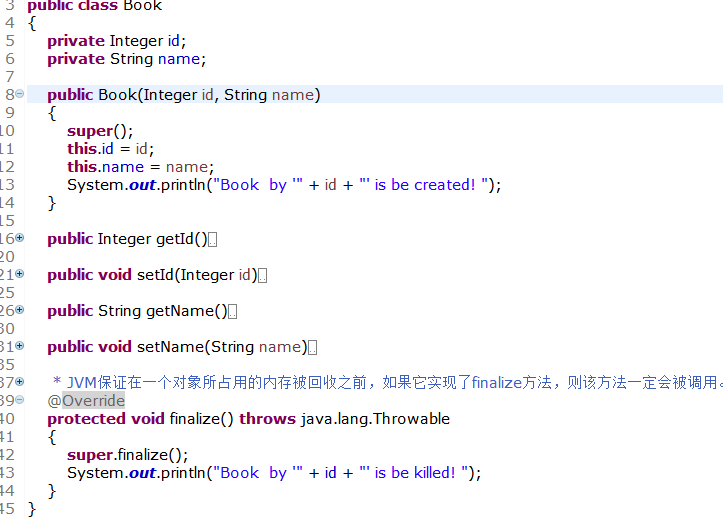
super.finalize();

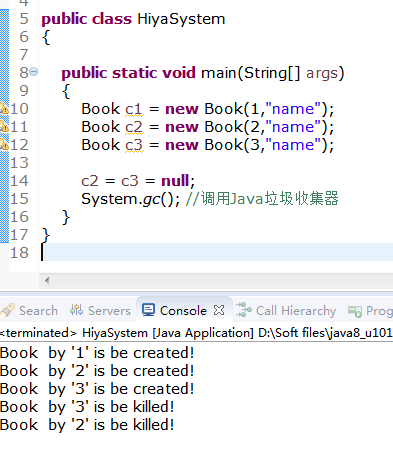
System.out.println("Cake Object " + id + "is disposed");

}

}

运行以上代码，输出结果如下：





## 十五、Java Scanner 类

**1 java.util.Scanner 是 Java5 的新特征，**我们可以通过 Scanner 类来获取用户的输入。

下面是创建 Scanner 对象的基本语法：

Scanner s = new Scanner(System.in);

接下来我们演示一个最简单的数据输入，并通过 Scanner 类的 next() 与 nextLine() 方法获取输入的字符串，在读取前我们一般需要 使用 hasNext 与 hasNextLine 判断是否还有输入的数据：

**2 使用 next 方法：**

ScannerDemo.java 文件代码：

import java.util.Scanner;

public class ScannerDemo {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

// 从键盘接收数据

// next方式接收字符串

System.out.println("next方式接收：");

// 判断是否还有输入

if (scan.hasNext()) {

String str1 = scan.next();

System.out.println("输入的数据为：" + str1);

}

scan.close();

}

}

执行以上程序输出结果为：

$ javac ScannerDemo.java

$ java ScannerDemo

next方式接收：

runoob com

输入的数据为：runoob

可以看到 com 字符串并未输出，接下来我们看 nextLine。

**3 使用 nextLine 方法：**

ScannerDemo.java 文件代码：

import java.util.Scanner;

public class ScannerDemo {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

// 从键盘接收数据

// nextLine方式接收字符串

System.out.println("nextLine方式接收：");

// 判断是否还有输入

if (scan.hasNextLine()) {

String str2 = scan.nextLine();

System.out.println("输入的数据为：" + str2);

}

scan.close();

}

}

执行以上程序输出结果为：

$ javac ScannerDemo.java

$ java ScannerDemo

nextLine方式接收：

runoob com

输入的数据为：runoob com

可以看到 com 字符串输出。

**4 next() 与 nextLine() 区别**

（1）next():

1、一定要读取到有效字符后才可以结束输入。

2、对输入有效字符之前遇到的空白，next() 方法会自动将其去掉。

3、只有输入有效字符后才将其后面输入的空白作为分隔符或者结束符。

next() 不能得到带有空格的字符串。

（2）nextLine()：

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

2、可以获得空白。

如果要输入 int 或 float 类型的数据，在 Scanner 类中也有支持，但是在输入之前最好先使用 hasNextXxx() 方法进行验证，再使用 nextXxx() 来读取：

ScannerDemo.java 文件代码：

import java.util.Scanner;

public class ScannerDemo {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

// 从键盘接收数据

int i = 0;

float f = 0.0f;

System.out.print("输入整数：");

if (scan.hasNextInt()) {

// 判断输入的是否是整数

i = scan.nextInt();

// 接收整数

System.out.println("整数数据：" + i);

} else {

// 输入错误的信息

System.out.println("输入的不是整数！");

}

System.out.print("输入小数：");

if (scan.hasNextFloat()) {

// 判断输入的是否是小数

f = scan.nextFloat();

// 接收小数

System.out.println("小数数据：" + f);

} else {

// 输入错误的信息

System.out.println("输入的不是小数！");

}

scan.close();

}

}

执行以上程序输出结果为：

$ javac ScannerDemo.java

$ java ScannerDemo

输入整数：12

整数数据：12

输入小数：1.2

小数数据：1.2

以下实例我们可以输入多个数字，并求其总和与平均数，每输入一个数字用回车确认，通过输入非数字来结束输入并输出执行结果：

ScannerDemo.java 文件代码：

import java.util.Scanner;

class ScannerDemo {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

double sum = 0;

int m = 0;

while (scan.hasNextDouble()) {

double x = scan.nextDouble();

m = m + 1;

sum = sum + x;

}

System.out.println(m + "个数的和为" + sum);

System.out.println(m + "个数的平均值是" + (sum / m));

scan.close();

}

}

执行以上程序输出结果为：

$ javac ScannerDemo.java

$ java ScannerDemo

12

23

15

21.4

end

4个数的和为71.4

4个数的平均值是17.85

## 十六、Java 数据结构

**1 Java工具包提供了强大的数据结构。**在Java中的数据结构主要包括以下几种接口和类：

枚举（Enumeration）

位集合（BitSet）

向量（Vector）

栈（Stack）

字典（Dictionary）

哈希表（Hashtable）

属性（Properties）

**2 枚举（Enumeration）**

枚举（Enumeration）接口虽然它本身不属于数据结构,但它在其他数据结构的范畴里应用很广。 枚举（The Enumeration）接口定义了一种从数据结构中取回连续元素的方式。例如，枚举定义了一个叫nextElement 的方法，该方法用来得到一个包含多元素的数据结构的下一个元素。关于枚举接口的更多信息，请参见枚举（Enumeration）。

**3 位集合（BitSet）**

位集合类实现了一组可以单独设置和清除的位或标志。该类在处理一组布尔值的时候非常有用，你只需要给每个值赋值一"位"，然后对位进行适当的设置或清除，就可以对布尔值进行操作了。关于该类的更多信息，请参见位集合（BitSet）。

**4 向量（Vector）**

向量（Vector）类和传统数组非常相似，但是Vector的大小能根据需要动态的变化。和数组一样，Vector对象的元素也能通过索引访问。使用Vector类最主要的好处就是在创建对象的时候不必给对象指定大小，它的大小会根据需要动态的变化。关于该类的更多信息，请参见向量(Vector)

**5 栈（Stack）**

栈（Stack）实现了一个后进先出（LIFO）的数据结构。你可以把栈理解为对象的垂直分布的栈，当你添加一个新元素时，就将新元素放在其他元素的顶部。当你从栈中取元素的时候，就从栈顶取一个元素。换句话说，最后进栈的元素最先被取出。关于该类的更多信息，请参见栈（Stack）。

**6 字典（Dictionary）**

字典（Dictionary） 类是一个抽象类，它定义了键映射到值的数据结构。当你想要通过特定的键而不是整数索引来访问数据的时候，这时候应该使用Dictionary。由于Dictionary类是抽象类，所以它只提供了键映射到值的数据结构，而没有提供特定的实现。关于该类的更多信息，请参见字典（ Dictionary）。

**7 哈希表（Hashtable）**

Hashtable类提供了一种在用户定义键结构的基础上来组织数据的手段。例如，在地址列表的哈希表中，你可以根据邮政编码作为键来存储和排序数据，而不是通过人名。哈希表键的具体含义完全取决于哈希表的使用情景和它包含的数据。关于该类的更多信息，请参见哈希表（HashTable）。

**8 属性（Properties）**

Properties 继承于 Hashtable.Properties 类表示了一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。Properties 类被许多Java类使用。例如，在获取环境变量时它就作为System.getProperties()方法的返回值。关于该类的更多信息，请参见属性（Properties）。

## 十七、Java 文档注释

**1 Java 支持三种注释方式。**前两种分别是 // 和 /\* \*/，第三种被称作说明注释，它以 /\*\* 开始，以 \*/结束。

说明注释允许你在程序中嵌入关于程序的信息。你可以使用 javadoc 工具软件来生成信息，并输出到HTML文件中。

说明注释，使你更加方便的记录你的程序信息。

**2 javadoc 标签**

javadoc 工具软件识别以下标签：

|  |  |  |
| --- | --- | --- |
| **标签** | **描述** | **示例** |
| @author | 标识一个类的作者 | @author description |
| @deprecated | 指名一个过期的类或成员 | @deprecated description |
| {@docRoot} | 指明当前文档根目录的路径 | Directory Path |
| @exception | 标志一个类抛出的异常 | @exception exception-name explanation |
| {@inheritDoc} | 从直接父类继承的注释 | Inherits a comment from the immediate surperclass. |
| {@link} | 插入一个到另一个主题的链接 | {@link name text} |
| {@linkplain} | 插入一个到另一个主题的链接，但是该链接显示纯文本字体 | Inserts an in-line link to another topic. |
| @param | 说明一个方法的参数 | @param parameter-name explanation |
| @return | 说明返回值类型 | @return explanation |
| @see | 指定一个到另一个主题的链接 | @see anchor |
| @serial | 说明一个序列化属性 | @serial description |
| @serialData | 说明通过writeObject( ) 和 writeExternal( )方法写的数据 | @serialData description |
| @serialField | 说明一个ObjectStreamField组件 | @serialField name type description |
| @since | 标记当引入一个特定的变化时 | @since release |
| @throws | 和 @exception标签一样. | The @throws tag has the same meaning as the @exception tag. |
| {@value} | 显示常量的值，该常量必须是static属性。 | Displays the value of a constant, which must be a static field. |
| @version | 指定类的版本 | @version info |

**3 文档注释**

在开始的 /\*\* 之后，第一行或几行是关于类、变量和方法的主要描述。

之后，你可以包含一个或多个何种各样的 @ 标签。每一个 @ 标签必须在一个新行的开始或者在一行的开始紧跟星号(\*).

多个相同类型的标签应该放成一组。例如，如果你有三个 @see 标签，可以将它们一个接一个的放在一起。

下面是一个类的说明注释的实例：

/\*\*\* 这个类绘制一个条形图

\* @author runoob

\* @version 1.2

\*/

**4 javadoc 输出什么**

javadoc 工具将你 Java 程序的源代码作为输入，输出一些包含你程序注释的HTML文件。

每一个类的信息将在独自的HTML文件里。javadoc 也可以输出继承的树形结构和索引。

由于 javadoc 的实现不同，工作也可能不同，你需要检查你的 Java 开发系统的版本等细节，选择合适的 Javadoc 版本。

下面是一个使用说明注释的简单实例。注意每一个注释都在它描述的项目的前面。

在经过 javadoc 处理之后，SquareNum 类的注释将在 SquareNum.html 中找到。

SquareNum.java 文件代码：

import java.io.\*;

/\*\*

\* 这个类演示了文档注释

\* @author Ayan Amhed

\* @version 1.2

\*/

public class SquareNum {

/\*\*

\* This method returns the square of num.

\* This is a multiline description. You can use

\* as many lines as you like.

\* @param num The value to be squared.

\* @return num squared.

\*/

public double square(double num) {

return num \* num;

}

/\*\*

\* This method inputs a number from the user.

\* @return The value input as a double.

\* @exception IOException On input error.

\* @see IOException

\*/

public double getNumber() throws IOException {

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader inData = new BufferedReader(isr);

String str;

str = inData.readLine();

return (new Double(str)).doubleValue();

}

/\*\*

\* This method demonstrates square().

\* @param args Unused.

\* @return Nothing.

\* @exception IOException On input error.

\* @see IOException

\*/

public static void main(String args[]) throws IOException

{

SquareNum ob = new SquareNum();

double val;

System.out.println("Enter value to be squared: ");

val = ob.getNumber();

val = ob.square(val);

System.out.println("Squared value is " + val);

}

}

**5 如下，使用 javadoc 工具处理 SquareNum.java 文件：**

$ javadoc SquareNum.java

Loading source file SquareNum.java...

Constructing Javadoc information...

Standard Doclet version 1.5.0\_13

Building tree for all the packages and classes...

Generating SquareNum.html...

SquareNum.java:39: warning - @return tag cannot be used\

in method with void return type.

Generating package-frame.html...

Generating package-summary.html...

Generating package-tree.html...

Generating constant-values.html...

Building index for all the packages and classes...

Generating overview-tree.html...

Generating index-all.html...

Generating deprecated-list.html...

Building index for all classes...

Generating allclasses-frame.html...

Generating allclasses-noframe.html...

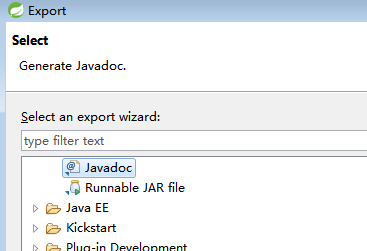
Generating index.html...

Generating help-doc.html...

Generating stylesheet.css...

1 warning

$



## Java 8 新特性

**1. 简介**

毫无疑问，Java 8是Java自Java 5（发布于2004年）之后的最重要的版本。这个版本包含语言、编译器、库、工具和JVM等方面的十多个新特性。

**2. Java语言的新特性**

Java 8是Java的一个重大版本，有人认为，虽然这些新特性领Java开发人员十分期待，但同时也需要花不少精力去学习。在这一小节中，我们将介绍Java 8的大部分新特性。

**（1）Lambda表达式和函数式接口**

Lambda表达式（也称为闭包）是Java 8中最大和最令人期待的语言改变。它允许我们将函数当成参数传递给某个方法，或者把代码本身当作数据处理：函数式开发者非常熟悉这些概念。很多JVM平台上的语言（Groovy、Scala等）从诞生之日就支持Lambda表达式，但是Java开发者没有选择，只能使用匿名内部类代替Lambda表达式。

Lambda的设计耗费了很多时间和很大的社区力量，最终找到一种折中的实现方案，可以实现简洁而紧凑的语言结构。最简单的Lambda表达式可由逗号分隔的参数列表、->符号和语句块组成，例如：

Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );

在上面这个代码中的参数e的类型是由编译器推理得出的，你也可以显式指定该参数的类型，例如：

Arrays.asList( "a", "b", "d" ).forEach( ( String e ) -> System.out.println( e ) );

如果Lambda表达式需要更复杂的语句块，则可以使用花括号将该语句块括起来，类似于Java中的函数体，例如：

Arrays.asList( "a", "b", "d" ).forEach( e -> {

System.out.print( e );

System.out.print( e );

} );

Lambda表达式可以引用类成员和局部变量（会将这些变量隐式得转换成final的），例如下列两个代码块的效果完全相同：

String separator = ",";

Arrays.asList( "a", "b", "d" ).forEach(

( String e ) -> System.out.print( e + separator ) );

和

final String separator = ",";

Arrays.asList( "a", "b", "d" ).forEach(

( String e ) -> System.out.print( e + separator ) );

Lambda表达式有返回值，返回值的类型也由编译器推理得出。如果Lambda表达式中的语句块只有一行，则可以不用使用return语句，下列两个代码片段效果相同：

Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo( e2 ) );

和

Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {

int result = e1.compareTo( e2 );

return result;

} );

Lambda的设计者们为了让现有的功能与Lambda表达式良好兼容，考虑了很多方法，于是产生了函数接口这个概念。函数接口指的是只有一个函数的接口，这样的接口可以隐式转换为Lambda表达式。java.lang.Runnable和java.util.concurrent.Callable是函数式接口的最佳例子。在实践中，函数式接口非常脆弱：只要某个开发者在该接口中添加一个函数，则该接口就不再是函数式接口进而导致编译失败。为了克服这种代码层面的脆弱性，并显式说明某个接口是函数式接口，Java 8 提供了一个特殊的注解@FunctionalInterface（Java 库中的所有相关接口都已经带有这个注解了），举个简单的函数式接口的定义：

@FunctionalInterface

public interface Functional {

void method();

}

不过有一点需要注意，默认方法和静态方法不会破坏函数式接口的定义，因此如下的代码是合法的。

@FunctionalInterface

public interface FunctionalDefaultMethods {

void method();

default void defaultMethod() {

}

}

Lambda表达式作为Java 8的最大卖点，它有潜力吸引更多的开发者加入到JVM平台，并在纯Java编程中使用函数式编程的概念。如果你需要了解更多Lambda表达式的细节，可以参考官方文档。

**（2） 接口的默认方法和静态方法**

Java 8使用两个新概念扩展了接口的含义：默认方法和静态方法。默认方法使得接口有点类似traits，不过要实现的目标不一样。默认方法使得开发者可以在 不破坏二进制兼容性的前提下，往现存接口中添加新的方法，即不强制那些实现了该接口的类也同时实现这个新加的方法。

默认方法和抽象方法之间的区别在于抽象方法需要实现，而默认方法不需要。接口提供的默认方法会被接口的实现类继承或者覆写，例子代码如下：

private interface Defaulable {

// Interfaces now allow default methods, the implementer may or

// may not implement (override) them.

default String notRequired() {

return "Default implementation";

}

}

private static class DefaultableImpl implements Defaulable {

}

private static class OverridableImpl implements Defaulable {

@Override

public String notRequired() {

return "Overridden implementation";

}

}

Defaulable接口使用关键字default定义了一个默认方法notRequired()。DefaultableImpl类实现了这个接口，同时默认继承了这个接口中的默认方法；OverridableImpl类也实现了这个接口，但覆写了该接口的默认方法，并提供了一个不同的实现。

Java 8带来的另一个有趣的特性是在接口中可以定义静态方法，例子代码如下：

private interface DefaulableFactory {

// Interfaces now allow static methods

static Defaulable create( Supplier< Defaulable > supplier ) {

return supplier.get();

}

}

下面的代码片段整合了默认方法和静态方法的使用场景：

public static void main( String[] args ) {

Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );

System.out.println( defaulable.notRequired() );

defaulable = DefaulableFactory.create( OverridableImpl::new );

System.out.println( defaulable.notRequired() );

}

这段代码的输出结果如下：

Default implementation

Overridden implementation

由于JVM上的默认方法的实现在字节码层面提供了支持，因此效率非常高。默认方法允许在不打破现有继承体系的基础上改进接口。该特性在官方库中的应用是：给java.util.Collection接口添加新方法，如stream()、parallelStream()、forEach()和removeIf()等等。

尽管默认方法有这么多好处，但在实际开发中应该谨慎使用：在复杂的继承体系中，默认方法可能引起歧义和编译错误。如果你想了解更多细节，可以参考官方文档。

**（3）方法引用**

方法引用使得开发者可以直接引用现存的方法、Java类的构造方法或者实例对象。方法引用和Lambda表达式配合使用，使得java类的构造方法看起来紧凑而简洁，没有很多复杂的模板代码。

西门的例子中，Car类是不同方法引用的例子，可以帮助读者区分四种类型的方法引用。

public static class Car {

public static Car create( final Supplier< Car > supplier ) {

return supplier.get();

}

public static void collide( final Car car ) {

System.out.println( "Collided " + car.toString() );

}

public void follow( final Car another ) {

System.out.println( "Following the " + another.toString() );

}

public void repair() {

System.out.println( "Repaired " + this.toString() );

}

}

第一种方法引用的类型是构造器引用，语法是Class::new，或者更一般的形式：Class<T>::new。注意：这个构造器没有参数。

final Car car = Car.create( Car::new );

final List< Car > cars = Arrays.asList( car );

第二种方法引用的类型是静态方法引用，语法是Class::static\_method。注意：这个方法接受一个Car类型的参数。

cars.forEach( Car::collide );

第三种方法引用的类型是某个类的成员方法的引用，语法是Class::method，注意，这个方法没有定义入参：

cars.forEach( Car::repair );

第四种方法引用的类型是某个实例对象的成员方法的引用，语法是instance::method。注意：这个方法接受一个Car类型的参数：

final Car police = Car.create( Car::new );

cars.forEach( police::follow );

运行上述例子，可以在控制台看到如下输出（Car实例可能不同）：

Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d

Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d

Following the com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d

如果想了解和学习更详细的内容，可以参考官方文档

**（4）重复注解**

自从Java 5中引入注解以来，这个特性开始变得非常流行，并在各个框架和项目中被广泛使用。不过，注解有一个很大的限制是：在同一个地方不能多次使用同一个注解。Java 8打破了这个限制，引入了重复注解的概念，允许在同一个地方多次使用同一个注解。

在Java 8中使用@Repeatable注解定义重复注解，实际上，这并不是语言层面的改进，而是编译器做的一个trick，底层的技术仍然相同。可以利用下面的代码说明：

package com.javacodegeeks.java8.repeatable.annotations;

import java.lang.annotation.ElementType;

import java.lang.annotation.Repeatable;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

public class RepeatingAnnotations {

@Target( ElementType.TYPE )

@Retention( RetentionPolicy.RUNTIME )

public @interface Filters {

Filter[] value();

}

@Target( ElementType.TYPE )

@Retention( RetentionPolicy.RUNTIME )

@Repeatable( Filters.class )

public @interface Filter {

String value();

};

@Filter( "filter1" )

@Filter( "filter2" )

public interface Filterable {

}

public static void main(String[] args) {

for( Filter filter: Filterable.class.getAnnotationsByType( Filter.class ) ) {

System.out.println( filter.value() );

}

}

}

正如我们所见，这里的Filter类使用@Repeatable(Filters.class)注解修饰，而Filters是存放Filter注解的容器，编译器尽量对开发者屏蔽这些细节。这样，Filterable接口可以用两个Filter注解注释（这里并没有提到任何关于Filters的信息）。

另外，反射API提供了一个新的方法：getAnnotationsByType()，可以返回某个类型的重复注解，例如Filterable.class.getAnnoation(Filters.class)将返回两个Filter实例，输出到控制台的内容如下所示：

filter1

filter2

如果你希望了解更多内容，可以参考官方文档。

**（5）更好的类型推断**

Java 8编译器在类型推断方面有很大的提升，在很多场景下编译器可以推导出某个参数的数据类型，从而使得代码更为简洁。例子代码如下：

package com.javacodegeeks.java8.type.inference;

public class Value< T > {

public static< T > T defaultValue() {

return null;

}

public T getOrDefault( T value, T defaultValue ) {

return ( value != null ) ? value : defaultValue;

}

}

下列代码是Value<String>类型的应用：

package com.javacodegeeks.java8.type.inference;

public class TypeInference {

public static void main(String[] args) {

final Value< String > value = new Value<>();

value.getOrDefault( "22", Value.defaultValue() );

}

}

参数Value.defaultValue()的类型由编译器推导得出，不需要显式指明。在Java 7中这段代码会有编译错误，除非使用Value.<String>defaultValue()。

**（6）拓宽注解的应用场景**

Java 8拓宽了注解的应用场景。现在，注解几乎可以使用在任何元素上：局部变量、接口类型、超类和接口实现类，甚至可以用在函数的异常定义上。下面是一些例子：

package com.javacodegeeks.java8.annotations;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

import java.util.ArrayList;

import java.util.Collection;

public class Annotations {

@Retention( RetentionPolicy.RUNTIME )

@Target( { ElementType.TYPE\_USE, ElementType.TYPE\_PARAMETER } )

public @interface NonEmpty {

}

public static class Holder< @NonEmpty T > extends @NonEmpty Object {

public void method() throws @NonEmpty Exception {

}

}

@SuppressWarnings( "unused" )

public static void main(String[] args) {

final Holder< String > holder = new @NonEmpty Holder< String >();

@NonEmpty Collection< @NonEmpty String > strings = new ArrayList<>();

}

}

ElementType.TYPE\_USER和ElementType.TYPE\_PARAMETER是Java 8新增的两个注解，用于描述注解的使用场景。Java 语言也做了对应的改变，以识别这些新增的注解。

**3. Java编译器的新特性**

（1）参数名称

为了在运行时获得Java程序中方法的参数名称，老一辈的Java程序员必须使用不同方法，例如Paranamer liberary。Java 8终于将这个特性规范化，在语言层面（使用反射API和Parameter.getName()方法）和字节码层面（使用新的javac编译器以及-parameters参数）提供支持。

package com.javacodegeeks.java8.parameter.names;

import java.lang.reflect.Method;

import java.lang.reflect.Parameter;

public class ParameterNames {

public static void main(String[] args) throws Exception {

Method method = ParameterNames.class.getMethod( "main", String[].class );

for( final Parameter parameter: method.getParameters() ) {

System.out.println( "Parameter: " + parameter.getName() );

}

}

}

在Java 8中这个特性是默认关闭的，因此如果不带-parameters参数编译上述代码并运行，则会输出如下结果：

Parmeter: arg0

如果带-parameters参数，则会输出如下结果（正确的结果）：

Parameter: args

如果你使用Maven进行项目管理，则可以在maven-compiler-plugin编译器的配置项中配置-parameters参数：

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<compilerArgument>-parameters</compilerArgument>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

**4. Java官方库的新特性**

Java 8增加了很多新的工具类（date/time类），并扩展了现存的工具类，以支持现代的并发编程、函数式编程等。

**（1） Optional**

Java应用中最常见的bug就是空值异常。在Java 8之前，Google Guava引入了Optionals类来解决NullPointerException，从而避免源码被各种null检查污染，以便开发者写出更加整洁的代码。Java 8也将Optional加入了官方库。

Optional仅仅是一个容易：存放T类型的值或者null。它提供了一些有用的接口来避免显式的null检查，可以参考Java 8官方文档了解更多细节。

接下来看一点使用Optional的例子：可能为空的值或者某个类型的值：

Optional< String > fullName = Optional.ofNullable( null );

System.out.println( "Full Name is set? " + fullName.isPresent() );

System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );

System.out.println( fullName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );

如果Optional实例持有一个非空值，则isPresent()方法返回true，否则返回false；orElseGet()方法，Optional实例持有null，则可以接受一个lambda表达式生成的默认值；map()方法可以将现有的Opetional实例的值转换成新的值；orElse()方法与orElseGet()方法类似，但是在持有null的时候返回传入的默认值。

上述代码的输出结果如下：

Full Name is set? false

Full Name: [none]

Hey Stranger!

再看下另一个简单的例子：

Optional< String > firstName = Optional.of( "Tom" );

System.out.println( "First Name is set? " + firstName.isPresent() );

System.out.println( "First Name: " + firstName.orElseGet( () -> "[none]" ) );

System.out.println( firstName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );

System.out.println();

这个例子的输出是：

First Name is set? true

First Name: Tom

Hey Tom!

**（2）Streams**

新增的Stream API（java.util.stream）将生成环境的函数式编程引入了Java库中。这是目前为止最大的一次对Java库的完善，以便开发者能够写出更加有效、更加简洁和紧凑的代码。Steam API极大得简化了集合操作（后面我们会看到不止是集合），首先看下这个叫Task的类：

public class Streams {

private enum Status {

OPEN, CLOSED

};

private static final class Task {

private final Status status;

private final Integer points;

Task( final Status status, final Integer points ) {

this.status = status;

this.points = points;

}

public Integer getPoints() {

return points;

}

public Status getStatus() {

return status;

}

@Override

public String toString() {

return String.format( "[%s, %d]", status, points );

}

}

}

Task类有一个分数（或伪复杂度）的概念，另外还有两种状态：OPEN或者CLOSED。现在假设有一个task集合：

final Collection< Task > tasks = Arrays.asList(

new Task( Status.OPEN, 5 ),

new Task( Status.OPEN, 13 ),

new Task( Status.CLOSED, 8 )

);

首先看一个问题：在这个task集合中一共有多少个OPEN状态的点？在Java 8之前，要解决这个问题，则需要使用foreach循环遍历task集合；但是在Java 8中可以利用steams解决：包括一系列元素的列表，并且支持顺序和并行处理。

// Calculate total points of all active tasks using sum()

final long totalPointsOfOpenTasks = tasks

.stream()

.filter( task -> task.getStatus() == Status.OPEN )

.mapToInt( Task::getPoints )

.sum();

System.out.println( "Total points: " + totalPointsOfOpenTasks );

运行这个方法的控制台输出是：

Total points: 18

**（3）Date/Time API(JSR 310)**

Java 8引入了新的Date-Time API(JSR 310)来改进时间、日期的处理。时间和日期的管理一直是最令Java开发者痛苦的问题。java.util.Date和后来的java.util.Calendar一直没有解决这个问题（甚至令开发者更加迷茫）。因为上面这些原因，诞生了第三方库Joda-Time，可以替代Java的时间管理API。Java 8中新的时间和日期管理API深受Joda-Time影响，并吸收了很多Joda-Time的精华。新的java.time包包含了所有关于日期、时间、时区、Instant（跟日期类似但是精确到纳秒）、duration（持续时间）和时钟操作的类。新设计的API认真考虑了这些类的不变性（从java.util.Calendar吸取的教训），如果某个实例需要修改，则返回一个新的对象。

我们接下来看看java.time包中的关键类和各自的使用例子。首先，Clock类使用时区来返回当前的纳秒时间和日期。Clock可以替代System.currentTimeMillis()和TimeZone.getDefault()。

// Get the system clock as UTC offset

final Clock clock = Clock.systemUTC();

System.out.println( clock.instant() );

System.out.println( clock.millis() );

这个例子的输出结果是：

2014-04-12T15:19:29.282Z

1397315969360

**第二，关注下LocalDate和LocalTime类。**LocalDate仅仅包含ISO-8601日历系统中的日期部分；LocalTime则仅仅包含该日历系统中的时间部分。这两个类的对象都可以使用Clock对象构建得到。

// Get the local date and local time

final LocalDate date = LocalDate.now();

final LocalDate dateFromClock = LocalDate.now( clock );

System.out.println( date );

System.out.println( dateFromClock );

// Get the local date and local time

final LocalTime time = LocalTime.now();

final LocalTime timeFromClock = LocalTime.now( clock );

System.out.println( time );

System.out.println( timeFromClock );

上述例子的输出结果如下：

2014-04-12

2014-04-12

11:25:54.568

15:25:54.568

LocalDateTime类包含了LocalDate和LocalTime的信息，但是不包含ISO-8601日历系统中的时区信息。这里有一些关于LocalDate和LocalTime的例子：

// Get the local date/time

final LocalDateTime datetime = LocalDateTime.now();

final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );

System.out.println( datetime );

System.out.println( datetimeFromClock );

上述这个例子的输出结果如下：

2014-04-12T11:37:52.309

2014-04-12T15:37:52.309

如果你需要特定时区的data/time信息，则可以使用ZoneDateTime，它保存有ISO-8601日期系统的日期和时间，而且有时区信息。下面是一些使用不同时区的例子：

// Get the zoned date/time

final ZonedDateTime zonedDatetime = ZonedDateTime.now();

final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock );

final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( ZoneId.of( "America/Los\_Angeles" ) );

System.out.println( zonedDatetime );

System.out.println( zonedDatetimeFromClock );

System.out.println( zonedDatetimeFromZone );

这个例子的输出结果是：

2014-04-12T11:47:01.017-04:00[America/New\_York]

2014-04-12T15:47:01.017Z

2014-04-12T08:47:01.017-07:00[America/Los\_Angeles]

最后看下Duration类，它持有的时间精确到秒和纳秒。这使得我们可以很容易得计算两个日期之间的不同，例子代码如下：

// Get duration between two dates

final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );

final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );

final Duration duration = Duration.between( from, to );

System.out.println( "Duration in days: " + duration.toDays() );

System.out.println( "Duration in hours: " + duration.toHours() );

这个例子用于计算2014年4月16日和2015年4月16日之间的天数和小时数，输出结果如下：

Duration in days: 365

Duration in hours: 8783

对于Java 8的新日期时间的总体印象还是比较积极的，一部分是因为Joda-Time的积极影响，另一部分是因为官方终于听取了开发人员的需求。如果希望了解更多细节，可以参考官方文档。

**（4） Nashorn JavaScript引擎**

Java 8提供了新的Nashorn JavaScript引擎，使得我们可以在JVM上开发和运行JS应用。Nashorn JavaScript引擎是javax.script.ScriptEngine的另一个实现版本，这类Script引擎遵循相同的规则，允许Java和JavaScript交互使用，例子代码如下：

ScriptEngineManager manager = new ScriptEngineManager();

ScriptEngine engine = manager.getEngineByName( "JavaScript" );

System.out.println( engine.getClass().getName() );

System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) );

这个代码的输出结果如下：

jdk.nashorn.api.scripting.NashornScriptEngine

Result: 2

**（5）Base64**

对Base64编码的支持已经被加入到Java 8官方库中，这样不需要使用第三方库就可以进行Base64编码，例子代码如下：

package com.javacodegeeks.java8.base64;

import java.nio.charset.StandardCharsets;

import java.util.Base64;

public class Base64s {

public static void main(String[] args) {

final String text = "Base64 finally in Java 8!";

final String encoded = Base64

.getEncoder()

.encodeToString( text.getBytes( StandardCharsets.UTF\_8 ) );

System.out.println( encoded );

final String decoded = new String(

Base64.getDecoder().decode( encoded ),

StandardCharsets.UTF\_8 );

System.out.println( decoded );

}

}

这个例子的输出结果如下：

QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==

Base64 finally in Java 8!

新的Base64API也支持URL和MINE的编码解码。

(Base64.getUrlEncoder() / Base64.getUrlDecoder(), Base64.getMimeEncoder() / Base64.getMimeDecoder())。

**（6）并行数组**

Java8版本新增了很多新的方法，用于支持并行数组处理。最重要的方法是parallelSort()，可以显著加快多核机器上的数组排序。下面的例子论证了parallexXxx系列的方法：

package com.javacodegeeks.java8.parallel.arrays;

import java.util.Arrays;

import java.util.concurrent.ThreadLocalRandom;

public class ParallelArrays {

public static void main( String[] args ) {

long[] arrayOfLong = new long [ 20000 ];

Arrays.parallelSetAll( arrayOfLong,

index -> ThreadLocalRandom.current().nextInt( 1000000 ) );

Arrays.stream( arrayOfLong ).limit( 10 ).forEach(

i -> System.out.print( i + " " ) );

System.out.println();

Arrays.parallelSort( arrayOfLong );

Arrays.stream( arrayOfLong ).limit( 10 ).forEach(

i -> System.out.print( i + " " ) );

System.out.println();

}

}

上述这些代码使用parallelSetAll()方法生成20000个随机数，然后使用parallelSort()方法进行排序。这个程序会输出乱序数组和排序数组的前10个元素。上述例子的代码输出的结果是：

Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839 119108 552378

Sorted: 39 220 263 268 325 607 655 678 723 793

**（7）并发性**

基于新增的lambda表达式和steam特性，为Java 8中为java.util.concurrent.ConcurrentHashMap类添加了新的方法来支持聚焦操作；另外，也为java.util.concurrentForkJoinPool类添加了新的方法来支持通用线程池操作（更多内容可以参考我们的并发编程课程）。

Java 8还添加了新的java.util.concurrent.locks.StampedLock类，用于支持基于容量的锁——该锁有三个模型用于支持读写操作（可以把这个锁当做是java.util.concurrent.locks.ReadWriteLock的替代者）。

在java.util.concurrent.atomic包中也新增了不少工具类，列举如下：

DoubleAccumulator

DoubleAdder

LongAccumulator

LongAdder

**5. 新的Java工具**

Java 8提供了一些新的命令行工具，这部分会讲解一些对开发者最有用的工具。

（1）Nashorn引擎：jjs

jjs是一个基于标准Nashorn引擎的命令行工具，可以接受js源码并执行。例如，我们写一个func.js文件，内容如下：

function f() {

return 1;

};

print( f() + 1 );

可以在命令行中执行这个命令：jjs func.js，控制台输出结果是：

2如果需要了解细节，可以参考官方文档。

**（2）类依赖分析器：jdeps**

jdeps是一个相当棒的命令行工具，它可以展示包层级和类层级的Java类依赖关系，它以.class文件、目录或者Jar文件为输入，然后会把依赖关系输出到控制台。

我们可以利用jedps分析下Spring Framework库，为了让结果少一点，仅仅分析一个JAR文件：org.springframework.core-3.0.5.RELEASE.jar。

jdeps org.springframework.core-3.0.5.RELEASE.jar

这个命令会输出很多结果，我们仅看下其中的一部分：依赖关系按照包分组，如果在classpath上找不到依赖，则显示"not found".

org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\Java\jdk1.8.0\jre\lib\rt.jar

org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)

-> java.io

-> java.lang

-> java.lang.annotation

-> java.lang.ref

-> java.lang.reflect

-> java.util

-> java.util.concurrent

-> org.apache.commons.logging not found

-> org.springframework.asm not found

-> org.springframework.asm.commons not found

org.springframework.core.annotation (org.springframework.core-3.0.5.RELEASE.jar)

-> java.lang

-> java.lang.annotation

-> java.lang.reflect

-> java.util

更多的细节可以参考官方文档。

**6. JVM的新特性**

使用Metaspace（JEP 122）代替持久代（PermGen space）。在JVM参数方面，使用-XX:MetaSpaceSize和-XX:MaxMetaspaceSize代替原来的-XX:PermSize和-XX:MaxPermSize。

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*jdk8实战 start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1 使用Metaspace（JEP 122）代替持久代（PermGen space）。在JVM参数方面，使用-XX:MetaSpaceSize和-XX:MaxMetaspaceSize代替原来的-XX:PermSize和-XX:MaxPermSize。

2 类依赖分析器：jdeps

jdeps org.springframework.core-3.0.5.RELEASE.jar

org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\Java\jdk1.8.0\jre\lib\rt.jar

org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)

-> java.io

-> java.lang

-> java.lang.annotation

-> java.lang.ref

-> java.lang.reflect

-> java.util

-> java.util.concurrent

-> org.apache.commons.logging not found

-> org.springframework.asm not found

-> org.springframework.asm.commons not found

org.springframework.core.annotation (org.springframework.core-3.0.5.RELEASE.jar)

-> java.lang

-> java.lang.annotation

-> java.lang.reflect

-> java.util

3 jjs是一个基于标准Nashorn引擎的命令行工具，可以接受js源码并执行

jjs func.js

4 Java 8还添加了新的java.util.concurrent.locks.StampedLock类，用于支持基于容量的锁——该锁有三个模型用于支持读写操作

（可以把这个锁当做是java.util.concurrent.locks.ReadWriteLock的替代者）

5 并行数组排序 和设值

long[] arrayOfLong = new long [ 20000 ];

Arrays.parallelSetAll( arrayOfLong, index -> ThreadLocalRandom.current().nextInt( 1000000 ) );

Arrays.parallelSort( arrayOfLong );

6 Nashorn JavaScript引擎

Java 8提供了新的Nashorn JavaScript引擎，使得我们可以在JVM上开发和运行JS应用。Nashorn JavaScript引擎是javax.script.ScriptEngine的另一个实现版本，

这类Script引擎遵循相同的规则，允许Java和JavaScript交互使用

ScriptEngineManager manager = new ScriptEngineManager();

ScriptEngine engine = manager.getEngineByName( "JavaScript" );

System.out.println( engine.getClass().getName() );

System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) );

7 Date/Time API

LocalDate，LocalTime，Clock

8 Streams

见代码

9 Optional

见代码

10 Lambda表达式和函数式接口

见代码

11 重复注解

见代码

12 方法引用

见代码

13 接口的默认方法和静态方法

见代码

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*jdk8实战 end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

## Java 9 新特性

**1. Java 平台级模块系统**

Java 9 的定义功能是一套全新的模块系统。当代码库越来越大，创建复杂，盘根错节的“意大利面条式代码”的几率呈指数级的增长。这时候就得面对两个基础的问题: 很难真正地对代码进行封装, 而系统并没有对不同部分（也就是 JAR 文件）之间的依赖关系有个明确的概念。每一个公共类都可以被类路径之下任何其它的公共类所访问到, 这样就会导致无意中使用了并不想被公开访问的 API。此外，类路径本身也存在问题: 你怎么知晓所有需要的 JAR 都已经有了, 或者是不是会有重复的项呢? 模块系统把这俩个问题都给解决了。

模块化的 JAR 文件都包含一个额外的模块描述器。在这个模块描述器中, 对其它模块的依赖是通过 “requires” 来表示的。另外, “exports” 语句控制着哪些包是可以被其它模块访问到的。所有不被导出的包默认都封装在模块的里面。如下是一个模块描述器的示例，存在于 “module-info.java” 文件中:

module blog {

exports com.pluralsight.blog;

requires cms;

}

我们可以如下展示模块：

请注意，两个模块都包含封装的包，因为它们没有被导出（使用橙色盾牌可视化）。 没有人会偶然地使用来自这些包中的类。Java 平台本身也使用自己的模块系统进行了模块化。通过封装 JDK 的内部类，平台更安全，持续改进也更容易。

当启动一个模块化应用时， JVM 会验证是否所有的模块都能使用，这基于 `requires` 语句——比脆弱的类路径迈进了一大步。模块允许你更好地强制结构化封装你的应用并明确依赖。你可以在这个课程中学习更多关于 Java 9 中模块工作的信息 。

**2. Linking**

当你使用具有显式依赖关系的模块和模块化的 JDK 时，新的可能性出现了。你的应用程序模块现在将声明其对其他应用程序模块的依赖以及对其所使用的 JDK 模块的依赖。为什么不使用这些信息创建一个最小的运行时环境，其中只包含运行应用程序所需的那些模块呢？ 这可以通过 Java 9 中的新的 jlink 工具实现。你可以创建针对应用程序进行优化的最小运行时映像而不需要使用完全加载 JDK 安装版本。

**3. JShell : 交互式 Java REPL**

许多语言已经具有交互式编程环境，Java 现在加入了这个俱乐部。您可以从控制台启动 jshell ，并直接启动输入和执行 Java 代码。 jshell 的即时反馈使它成为探索 API 和尝试语言特性的好工具。

测试一个 Java 正则表达式是一个很好的说明 jshell 如何使您的生活更轻松的例子。 交互式 shell 还可以提供良好的教学环境以及提高生产力，您可以在此了解更多信息。在教人们如何编写 Java 的过程中，不再需要解释 “public static void main（String [] args）” 这句废话。

**4. 改进的 Javadoc**

有时一些小事情可以带来很大的不同。你是否就像我一样在一直使用 Google 来查找正确的 Javadoc 页面呢？ 这不再需要了。Javadoc 现在支持在 API 文档中的进行搜索。另外，Javadoc 的输出现在符合兼容 HTML5 标准。此外，你会注意到，每个 Javadoc 页面都包含有关 JDK 模块类或接口来源的信息。

**5. 集合工厂方法**

通常，您希望在代码中创建一个集合（例如，List 或 Set ），并直接用一些元素填充它。 实例化集合，几个 “add” 调用，使得代码重复。 Java 9，添加了几种集合工厂方法：

Set<Integer> ints = Set.of(1, 2, 3);

List<String> strings = List.of("first", "second");

除了更短和更好阅读之外，这些方法也可以避免您选择特定的集合实现。 事实上，从工厂方法返回已放入数个元素的集合实现是高度优化的。这是可能的，因为它们是不可变的：在创建后，继续添加元素到这些集合会导致 “UnsupportedOperationException” 。

**6. 改进的 Stream API**

长期以来，Stream API 都是 Java 标准库最好的改进之一。通过这套 API 可以在集合上建立用于转换的申明管道。在 Java 9 中它会变得更好。Stream 接口中添加了 4 个新的方法：dropWhile, takeWhile, ofNullable。还有个 iterate 方法的新重载方法，可以让你提供一个 Predicate (判断条件)来指定什么时候结束迭代：

IntStream.iterate(1, i -> i < 100, i -> i + 1).forEach(System.out::println);

第二个参数是一个 Lambda，它会在当前 IntStream 中的元素到达 100 的时候返回 true。因此这个简单的示例是向控制台打印 1 到 99。

除了对 Stream 本身的扩展，Optional 和 Stream 之间的结合也得到了改进。现在可以通过 Optional 的新方法 `stram` 将一个 Optional 对象转换为一个(可能是空的) Stream 对象：

Stream<Integer> s = Optional.of(1).stream();

在组合复杂的 Stream 管道时，将 Optional 转换为 Stream 非常有用。

**7. 私有接口方法**

Java 8 为我们带来了接口的默认方法。 接口现在也可以包含行为，而不仅仅是方法签名。 但是，如果在接口上有几个默认方法，代码几乎相同，会发生什么情况？ 通常，您将重构这些方法，调用一个可复用的私有方法。 但默认方法不能是私有的。 将复用代码创建为一个默认方法不是一个解决方案，因为该辅助方法会成为公共API的一部分。 使用 Java 9，您可以向接口添加私有辅助方法来解决此问题：

public interface MyInterface {

void normalInterfaceMethod();

default void interfaceMethodWithDefault() { init(); }

default void anotherDefaultMethod() { init(); }

// This method is not part of the public API exposed by MyInterface

private void init() { System.out.println("Initializing"); }

}

如果您使用默认方法开发 API ，那么私有接口方法可能有助于构建其实现。

**8. HTTP/2**

Java 9 中有新的方式来处理 HTTP 调用。这个迟到的特性用于代替老旧的 `HttpURLConnection` API，并提供对 WebSocket 和 HTTP/2 的支持。注意：新的 HttpClient API 在 Java 9 中以所谓的孵化器模块交付。也就是说，这套 API 不能保证 100% 完成。不过你可以在 Java 9 中开始使用这套 API：

HttpClient client = HttpClient.newHttpClient();

HttpRequest req =

HttpRequest.newBuilder(URI.create("http://www.google.com"))

.header("User-Agent","Java")

.GET()

.build();

HttpResponse<String> resp = client.send(req, HttpResponse.BodyHandler.asString());

HttpResponse<String> resp = client.send(req, HttpResponse.BodyHandler.asString());

除了这个简单的请求/响应模型之外，HttpClient 还提供了新的 API 来处理 HTTP/2 的特性，比如流和服务端推送。

**9. 多版本兼容 JAR**

我们最后要来着重介绍的这个特性对于库的维护者而言是个特别好的消息。当一个新版本的 Java 出现的时候，你的库用户要花费数年时间才会切换到这个新的版本。这就意味着库得去向后兼容你想要支持的最老的 Java 版本 (许多情况下就是 Java 6 或者 7)。这实际上意味着未来的很长一段时间，你都不能在库中运用 Java 9 所提供的新特性。幸运的是，多版本兼容 JAR 功能能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本：

**multirelease.jar**

**├── META-INF**

**│ └── versions**

**│ └── 9**

**│ └── multirelease**

**│ └── Helper.class**

**├── multirelease**

**├── Helper.class**

**└── Main.class**

在上述场景中， multirelease.jar 可以在 Java 9 中使用, 不过 Helper 这个类使用的不是顶层的 multirelease.Helper 这个 class, 而是处在“META-INF/versions/9”下面的这个。这是特别为 Java 9 准备的 class 版本，可以运用 Java 9 所提供的特性和库。同时，在早期的 Java 诸版本中使用这个 JAR 也是能运行的，因为较老版本的 Java 只会看到顶层的这个 Helper 类。