

# Java 教程 | 菜鸟教程

 [runoob.com/java/java-tutorial.html](https://runoob.com/java/java-tutorial.html)

## Java 简介

## Java 教程

Java 是由 Sun Microsystems 公司于 1995 年 5 月推出的高级程序设计语言。

Java 可运行于多个平台，如 Windows, Mac OS 及其他多种 UNIX 版本的系统。



本教程通过简单的实例将让大家更好的了解 Java 编程语言。

移动操作系统 Android 大部分的代码采用 Java 编程语言编程。

## Java 在线工具

### JDK 11 在线中文手册

### JDK 1.6 在线中文手册(旧版)

## 我的第一个 JAVA 程序

以下我们通过一个简单的实例来展示 Java 编程，创建文件 **HelloWorld.java**(文件名需与类名一致)，代码如下：

## 实例

```
public class HelloWorld { public static void main(String[] args) {  
    System.out.println("Hello World"); } }
```

### 运行实例 »

| 注：String args[] 与 String[] args 都可以执行，但推荐使用 String[] args，这样可以避免歧义和误读。

运行以上实例，输出结果如下：

```
$ javac HelloWorld.java  
$ java HelloWorld  
Hello World
```

## 执行命令解析：

以上我们使用了两个命令 **javac** 和 **java**。

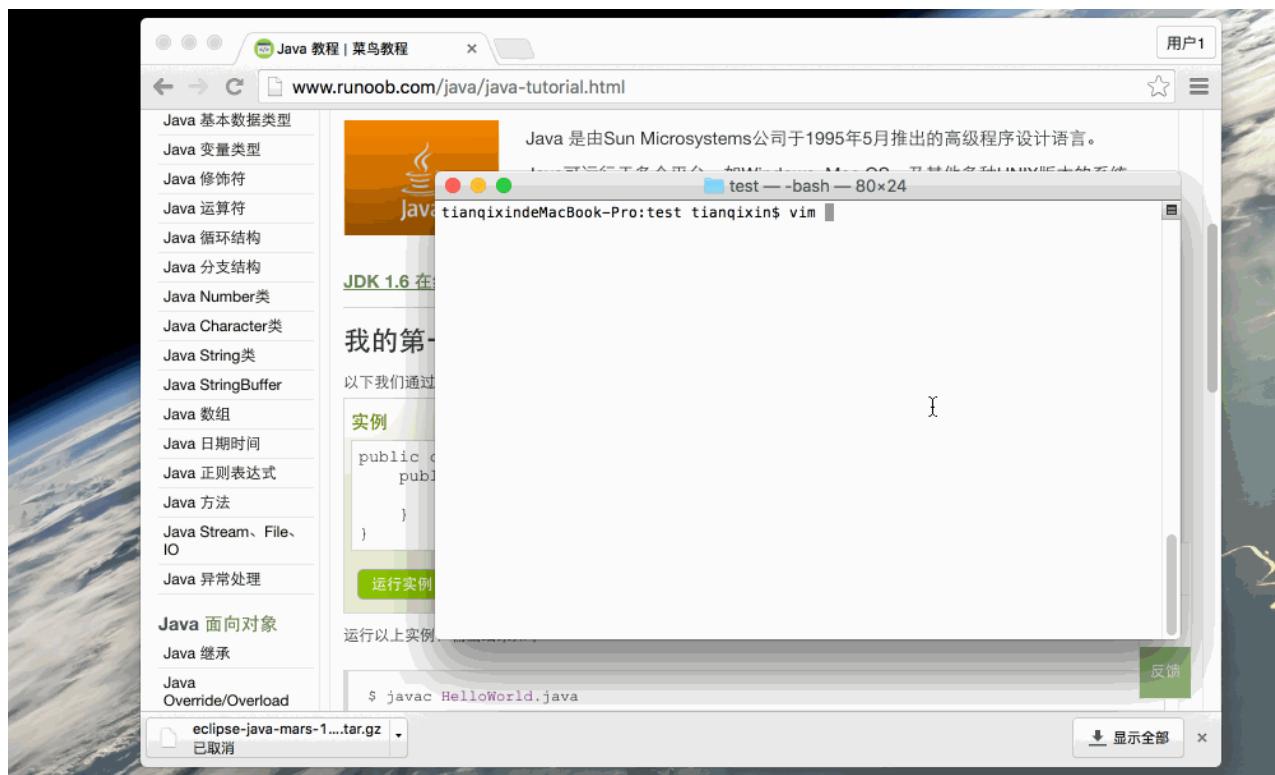
**javac** 后面跟着的是java文件的文件名，例如 HelloWorld.java。该命令用于将 java 源文件编译为 class 字节码文件，如：**javac HelloWorld.java**。

运行javac命令后，如果成功编译没有错误的话，会出现一个 HelloWorld.class 的文件。

**java** 后面跟着的是java文件中的类名，例如 HelloWorld 就是类名，如：java HelloWorld。

注意：java命令后面不要加.class。

Gif 图演示：



## 开始学习JAVA编程

- [开始学习Java课程](#)
- [Java 面向对象课程](#)
- [Java 高级课程](#)

## Java 简介

## 4 篇笔记 写笔记

# Java 简介 | 菜鸟教程



[runoob.com/java/java-intro.html](http://runoob.com/java/java-intro.html)

## Java 教程

### Java 开发环境配置

## 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。

后来 Sun 公司被 Oracle (甲骨文) 公司收购，Java 也随之成为 Oracle 公司的产品。

Java 分为三个体系：

- JavaSE (J2SE) (Java 2 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。

## 主要特性

- **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 去弃了 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。并提供了自动分配和回收内存空间，使得程序员不必为内存管理而担忧。

- **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。

- **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。

- **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。

- **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。

- **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。

- **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。

- **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。

- **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++。

- **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) 的构造子类将一个实现了 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）。

- **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。

## 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。
- 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。
- 2017年9月21日，Oracle公司发表Java SE 9
- 2018年3月21日，Oracle公司发表Java SE 10
- 2018年9月25日，Java SE 11发布
- 2019年3月20日，Java SE 12发布

---

## Java 开发工具

---

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.....
- vscode 编辑器或者其他编辑器。
- IDE：Eclipse、IntelliJ IDEA、NetBeans等。

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"); } }
```

在下一章节我们将介绍如何配置Java开发环境。

[Java 教程](#)

[Java 开发环境配置](#)

**4 篇笔记 写笔记**

---

# Java 开发环境配置 | 菜鸟教程

 [runoob.com/java/java-environment-setup.html](http://runoob.com/java/java-environment-setup.html)

[Java 简介](#)

[Java 基础语法](#)

## Java 开发环境配置

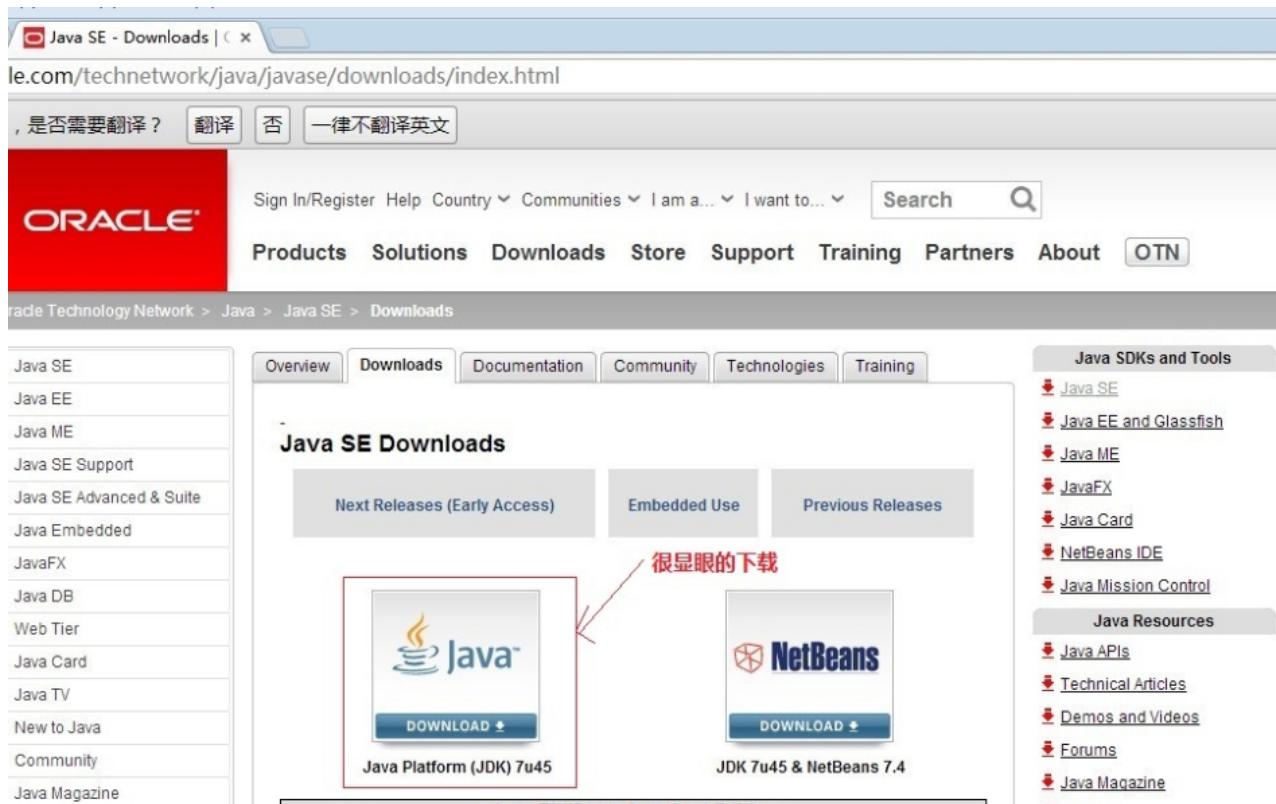
在本章节中我们将为大家介绍如何搭建Java 开发环境。

## window系统安装java

### 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：



The screenshot shows the Oracle Java SE Downloads page. At the top, there's a navigation bar with links for Sign In/Register, Help, Country, Communities, I am a..., I want to..., Search, and a magnifying glass icon. Below the navigation bar is a menu with Products, Solutions, Downloads, Store, Support, Training, Partners, About, and OTN. A red arrow points from the text "很显眼的下载" (Very prominent download) to the "DOWNLOAD" button for the "Java Platform (JDK) 7u45". To the right of this section is a sidebar titled "Java SDKs and Tools" with links for Java SE, Java EE and Glassfish, Java ME, JavaFX, Java Card, NetBeans IDE, Java Mission Control, Java Resources, Java APIs, Technical Articles, Demos and Videos, Forums, and Java Magazine.

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：

**Java SE Development Kit 8u91**

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

**接受许可**

Accept License Agreement     Decline License Agreement

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.72 MB	jdk-8u91-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.69 MB	jdk-8u91-linux-arm64-vfp-hflt.tar.gz
Linux x86	154.74 MB	jdk-8u91-linux-i586.rpm
Linux x86	174.92 MB	jdk-8u91-linux-i586.tar.gz
Linux x64	152.74 MB	jdk-8u91-linux-x64.rpm
Linux x64	172.97 MB	jdk-8u91-linux-x64.tar.gz
Mac OS X	227.29 MB	jdk-8u91-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.59 MB	jdk-8u91-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	98.95 MB	jdk-8u91-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140.29 MB	jdk-8u91-solaris-x64.tar.Z
Solaris x64	96.78 MB	jdk-8u91-solaris-x64.tar.gz
Windows x86	182.11 MB	jdk-8u91-windows-i586.exe
<b>Windows x64</b>	<b>187.41 MB</b>	<b>jdk-8u91-windows-x64.exe</b>

**选择64位**

**Java SE Development Kit 8u92**

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。

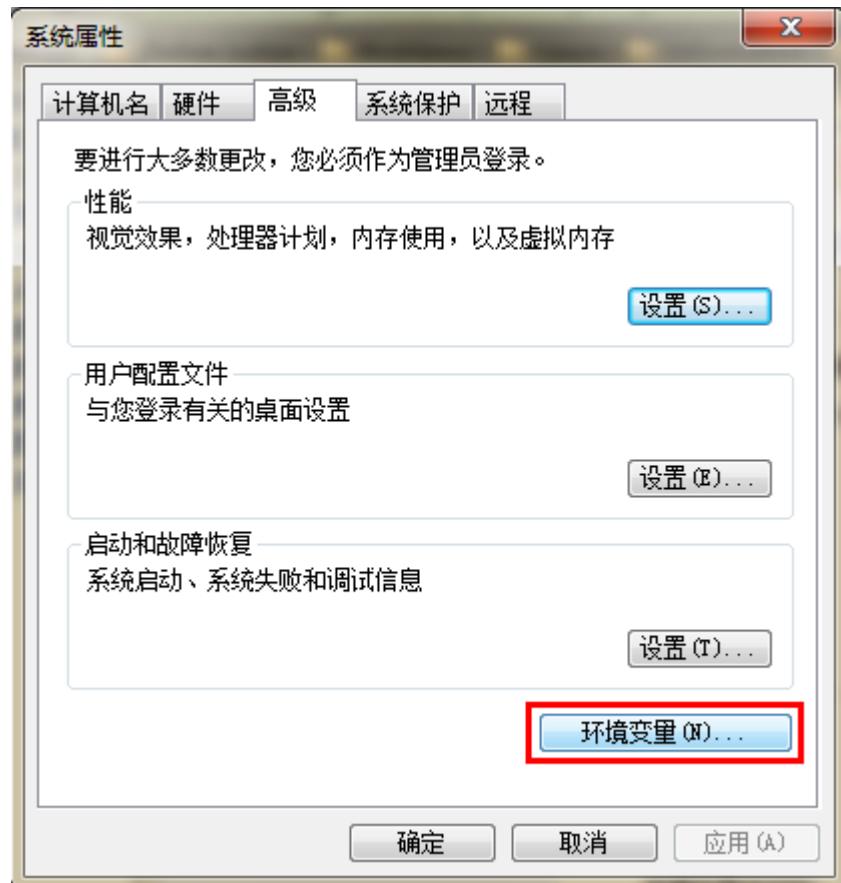
安装JDK，安装过程中可以自定义安装目录等信息，例如我们选择安装目录为 **C:\Program Files (x86)\Java\jdk1.8.0\_91**。

## 配置环境变量

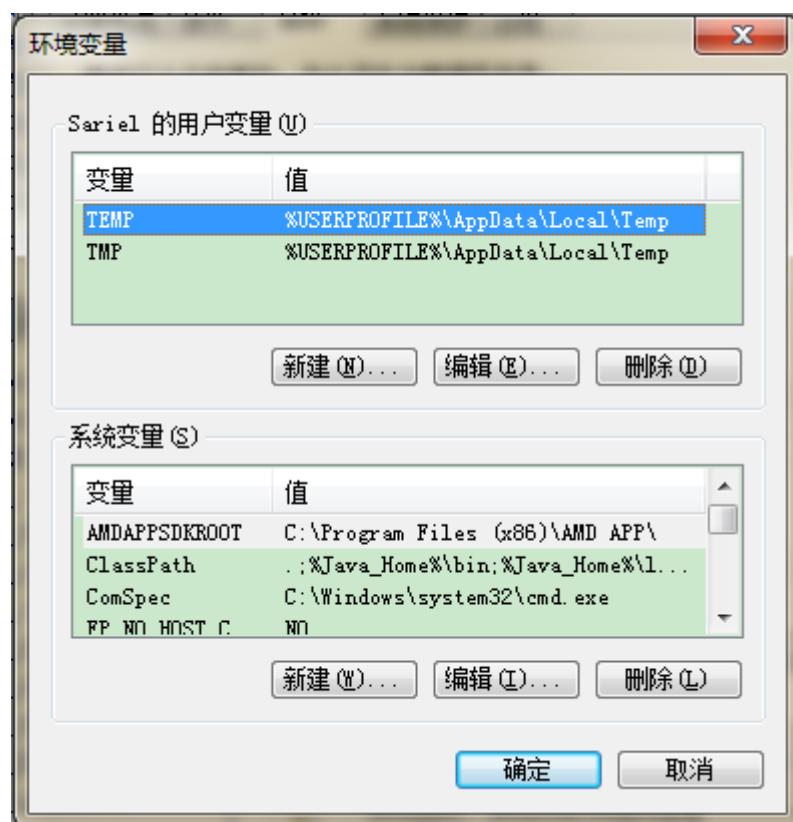
1. 安装完成后，右击“我的电脑”，点击“属性”，选择“高级系统设置”；



2. 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”。

**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。

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;**

## **JAVA\_HOME 设置**

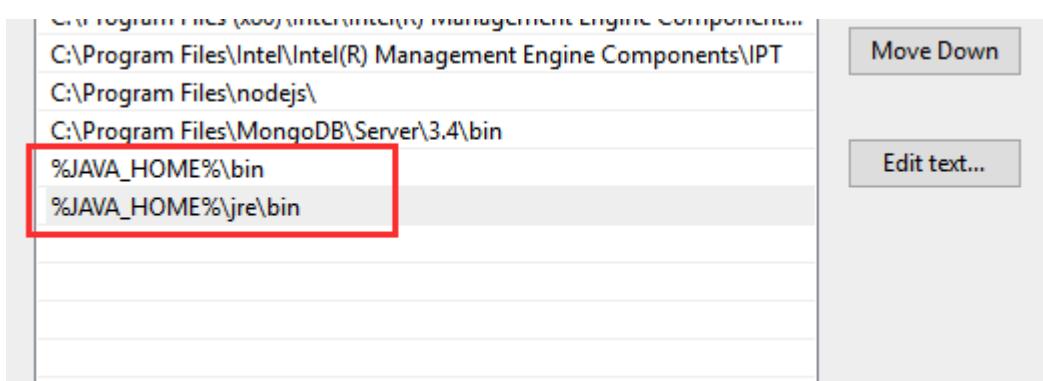


## **PATH 设置**



注意：在 Windows10 中，Path 变量里是分条显示的，我们需要将 %JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin; 分开添加，否则无法识别：

```
%JAVA_HOME%\bin;
%JAVA_HOME%\jre\bin;
```



更多内容可参考：[Windows 10 配置 Java 环境变量](#)

## 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。



## 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：

```
C:\Users\prado>java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
```

## 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

## 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一款不错的开发工具，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/downloads/packages/>

选择 **Eclipse IDE for Java Developers**：

Eclipse IDE for Java Developers  
165 MB 1,253,367 DOWNLOADS  
The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven integration and WindowBuilder...

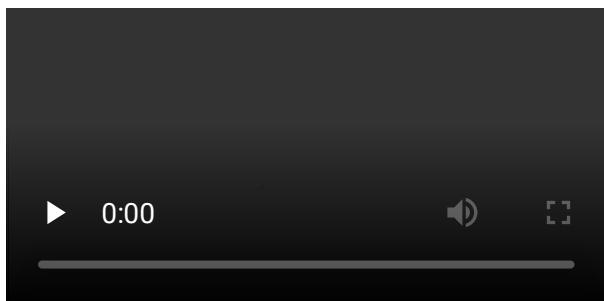
- **JetBrains** 的 IDEA，现在很多人开始使用了，功能很强大，下载地址：  
<https://www.jetbrains.com/idea/download/>

- **NotePad++** : NotePad++ 是在微软视窗环境之下一个免费的代码编辑器，下载地址：<http://notepad-plus-plus.org/>
- **Netbeans**: 开源免费的java IDE，下载地址：<http://www.netbeans.org/index.html>

## 使用 Eclipse 运行第一个 Java 程序

---

视频演示如下所示：



HelloWorld.java 文件代码：

```
public class HelloWorld { public static void main(String[] args) {  
    System.out.println("Hello World"); } }
```

[Java 简介](#)

[Java 基础语法](#)

## 5 篇笔记 写笔记

---

# Java 基础语法 | 菜鸟教程

 [runoob.com/java/java-basic-syntax.html](http://runoob.com/java/java-basic-syntax.html)

[Java 开发环境配置](#)

[Java 对象和类](#)

## Java 基础语法

一个 Java 程序可以认为是一系列对象的集合，而这些对象通过调用彼此的方法来协同工作。下面简要介绍下类、对象、方法和实例变量的概念。

- **对象**：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- **类**：类是一个模板，它描述一类对象的行为和状态。
- **方法**：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。
- **实例变量**：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。

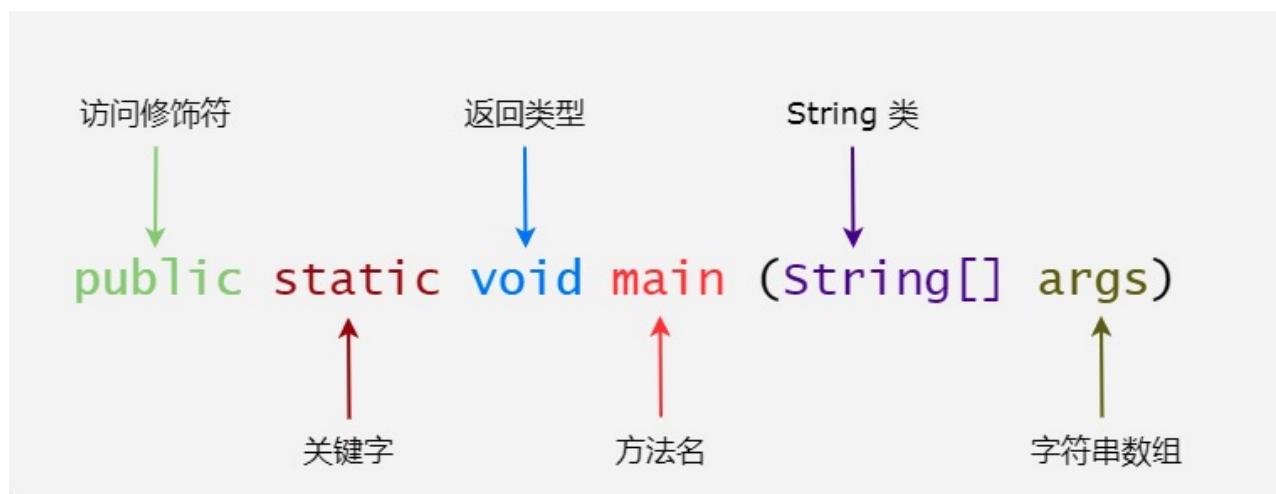
## 第一个Java程序

下面看一个简单的 Java 程序，它将输出字符串 *Hello World*

### 实例

```
public class HelloWorld { /* 第一个Java程序 * 它将输出字符串 Hello World */ public static void main(String[] args) { System.out.println("Hello World"); // 输出 Hello World } }
```

[运行实例 »](#)



下面将逐步介绍如何保存、编译以及运行这个程序：

- 打开代码编辑器，把上面的代码添加进去；

- 把文件名保存为：HelloWorld.java；
- 打开 cmd 命令窗口，进入目标文件所在的位置，假设是 C:\
- 在命令行窗口输入 javac HelloWorld.java 按下回车键编译代码。如果代码没有错误，cmd 命令提示符会进入下一行（假设环境变量都设置好了）。
- 再键输入 java HelloWorld 按下回车键就可以运行程序了

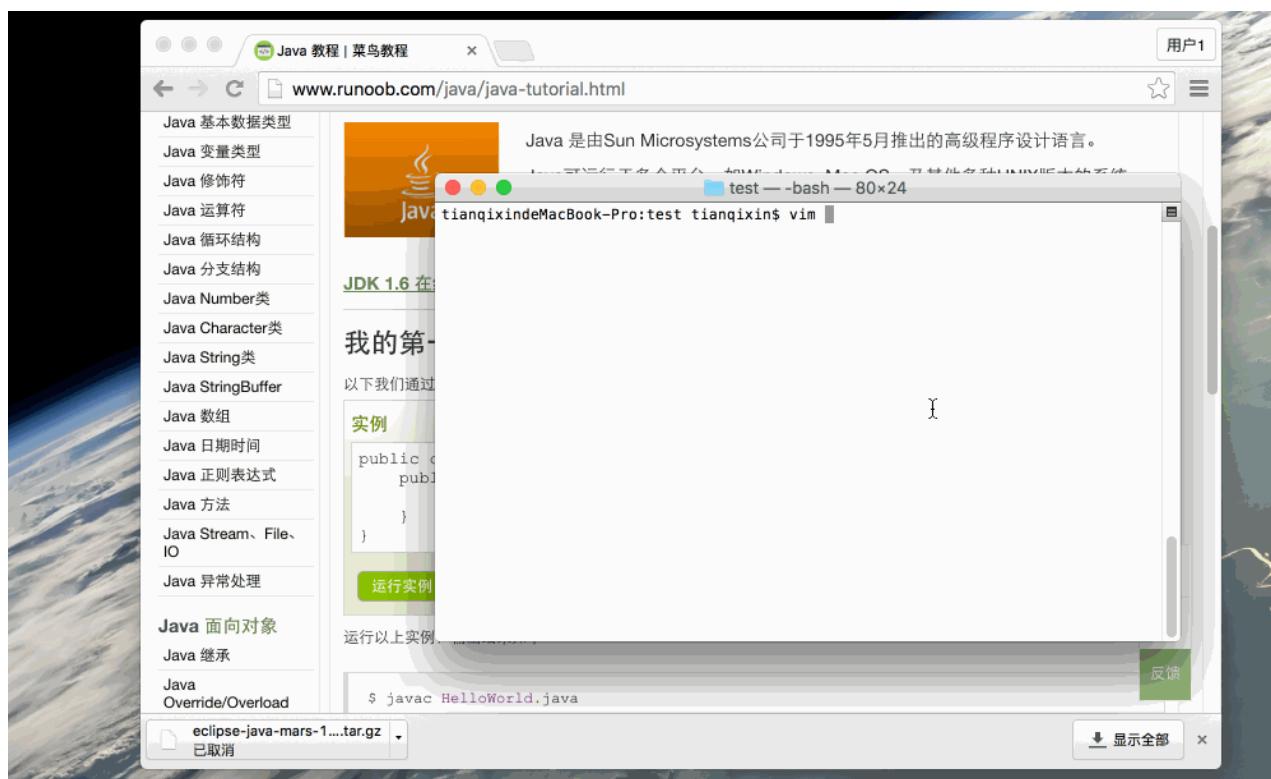
你将会在窗口看到 Hello World

```
$ javac HelloWorld.java
$ java HelloWorld
Hello World
```

如果遇到编码问题，我们可以使用 -encoding 选项设置 utf-8 来编译：

```
javac -encoding UTF-8 HelloWorld.java
java HelloWorld
```

Gif 图演示：



## 基本语法

编写 Java 程序时，应注意以下几点：

- **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。
- **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**。
- **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。

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

## 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。

关于 Java 标识符，有以下几点需要注意：

- 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
  - 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
  - 关键字不能用作标识符
  - 标识符是大小写敏感的
  - 合法标识符举例：age、\$salary、\_value、\_\_1\_value
  - 非法标识符举例：123abc、-salary
- 

## Java 修饰符

---

像其他语言一样，Java 可以使用修饰符来修饰类中方法和属性。主要有两类修饰符：

- 访问控制修饰符：default, public, protected, private
- 非访问控制修饰符：final, abstract, static, synchronized

在后面的章节中我们会深入讨论 Java 修饰符。

---

## Java 变量

---

Java 中主要有如下几种类型的变量

- 局部变量
  - 类变量（静态变量）
  - 成员变量（非静态变量）
- 

## Java 数组

---

数组是储存在堆上的对象，可以保存多个同类型变量。在后面的章节中，我们将会学到如何声明、构造以及初始化一个数组。

---

## Java 枚举

---

Java 5.0引入了枚举，枚举限制变量只能是预先设定好的值。使用枚举可以减少代码中的bug。

例如，我们为果汁店设计一个程序，它将限制果汁为小杯、中杯、大杯。这就意味着它不允许顾客点除了这三种尺寸外的果汁。

## 实例

```
class FreshJuice { enum FreshJuiceSize{ SMALL, MEDIUM , LARGE } FreshJuiceSize size; } public class FreshJuiceTest { public static void main(String[] args){ FreshJuice juice = new FreshJuice(); juice.size = FreshJuice.FreshJuiceSize.MEDIUM ; } }
```

注意：枚举可以单独声明或者声明在类里面。方法、变量、构造函数也可以在枚举中定义。

## Java 关键字

下面列出了 Java 关键字。这些保留字不能用于常量、变量、和任何标识符的名称。

类别	关键字	说明
访问控制	private	私有的
	protected	受保护的
	public	公共的
	default	默认
类、方法和变量修饰符	abstract	声明抽象
	class	类
	extends	扩充,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

	<code>volatile</code>	易失
程序控制语句	<code>break</code>	跳出循环
	<code>case</code>	定义一个值以供 switch 选择
	<code>continue</code>	继续
	<code>default</code>	默认
	<code>do</code>	运行
	<code>else</code>	否则
	<code>for</code>	循环
	<code>if</code>	如果
	<code>instanceof</code>	实例
	<code>return</code>	返回
	<code>switch</code>	根据值选择执行
	<code>while</code>	循环
错误处理	<code>assert</code>	断言表达式是否为真
	<code>catch</code>	捕捉异常
	<code>finally</code>	有没有异常都执行
	<code>throw</code>	抛出一个异常对象
	<code>throws</code>	声明一个异常可能被抛出
	<code>try</code>	捕获异常
包相关	<code>import</code>	引入
	<code>package</code>	包
基本类型	<code>boolean</code>	布尔型
	<code>byte</code>	字节型
	<code>char</code>	字符型
	<code>double</code>	双精度浮点
	<code>float</code>	单精度浮点
	<code>int</code>	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

## 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 * 它将输出 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"); } }
```

## Java 空行

空白行或者有注释的行, Java 编译器都会忽略掉。

## 继承

在 Java 中, 一个类可以由其他类派生。如果你要创建一个类, 而且已经存在一个类具有你所需要的属性或方法, 那么你可以将新创建的类继承该类。

利用继承的方法, 可以重用已存在类的方法和属性, 而不用重写这些代码。被继承的类称为超类 (super class), 派生类称为子类 (subclass)。

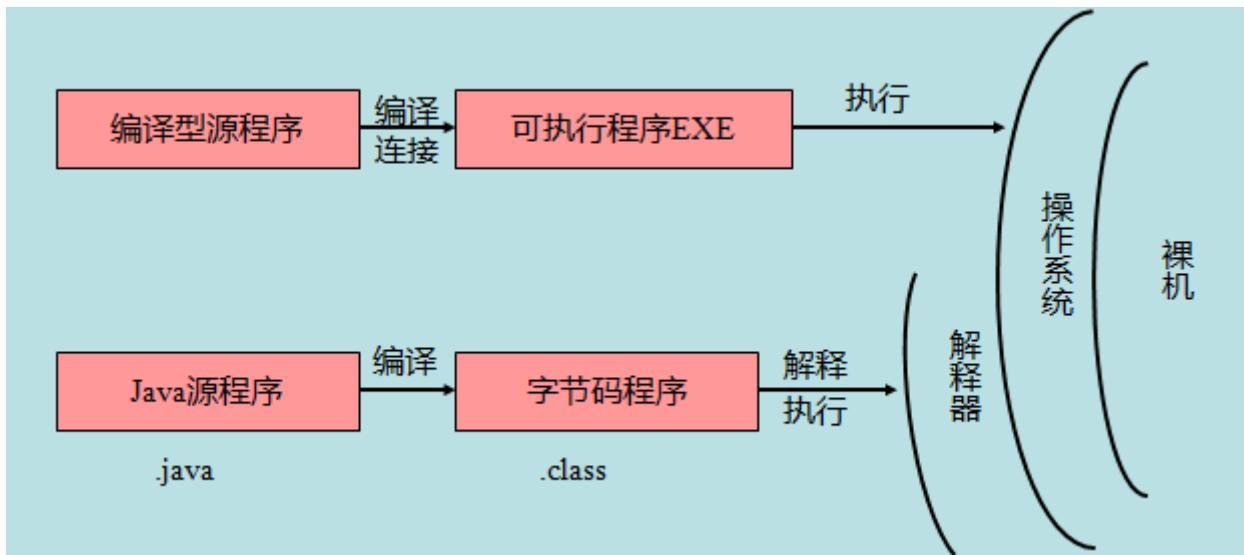
## 接口

在 Java 中, 接口可理解为对象间相互通信的协议。接口在继承中扮演着很重要的角色。

接口只定义派生要用到的方法, 但是方法的具体实现完全取决于派生类。

## Java 源程序与编译型运行区别

如下图所示：



下一节介绍 Java 编程中的类和对象。之后你将会有对 Java 中的类和对象有更清楚的认识。

[Java 开发环境配置](#)

[Java 对象和类](#)

## 8 篇笔记 写笔记

---

# Java 对象和类 | 菜鸟教程

 [runoob.com/java/java-object-classes.html](https://www.runoob.com/java/java-object-classes.html)

[Java 基础语法](#)

[Java 基本数据类型](#)

## Java 对象和类

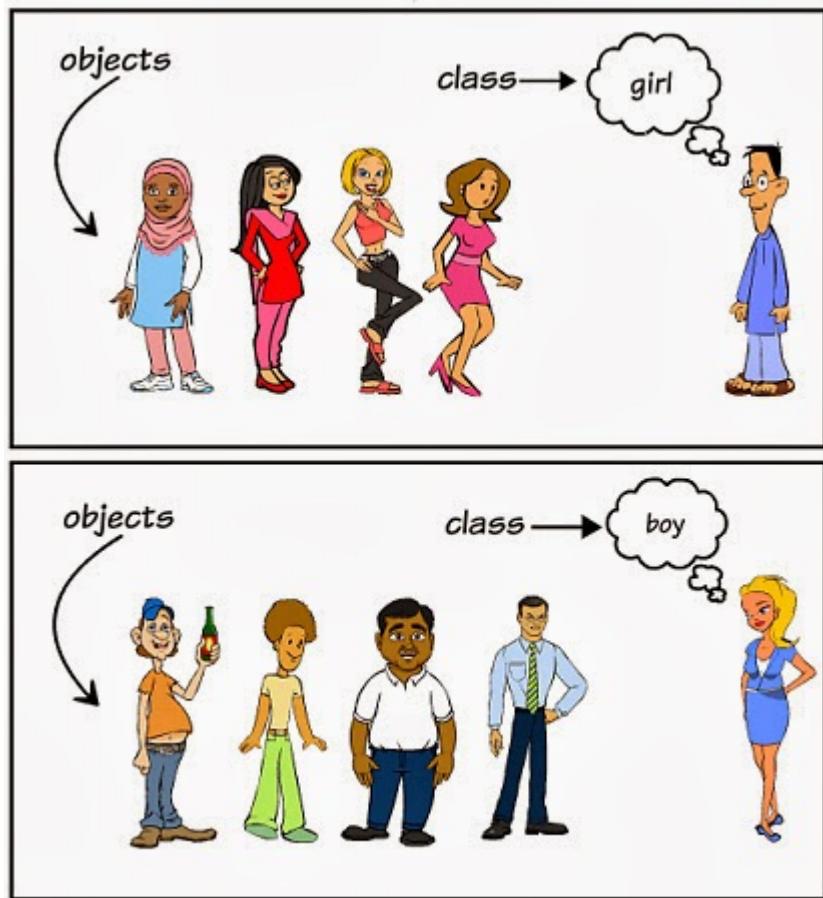
Java作为一种面向对象语言。支持以下基本概念：

- 多态
- 继承
- 封装
- 抽象
- 类
- 对象
- 实例
- 方法
- 重载

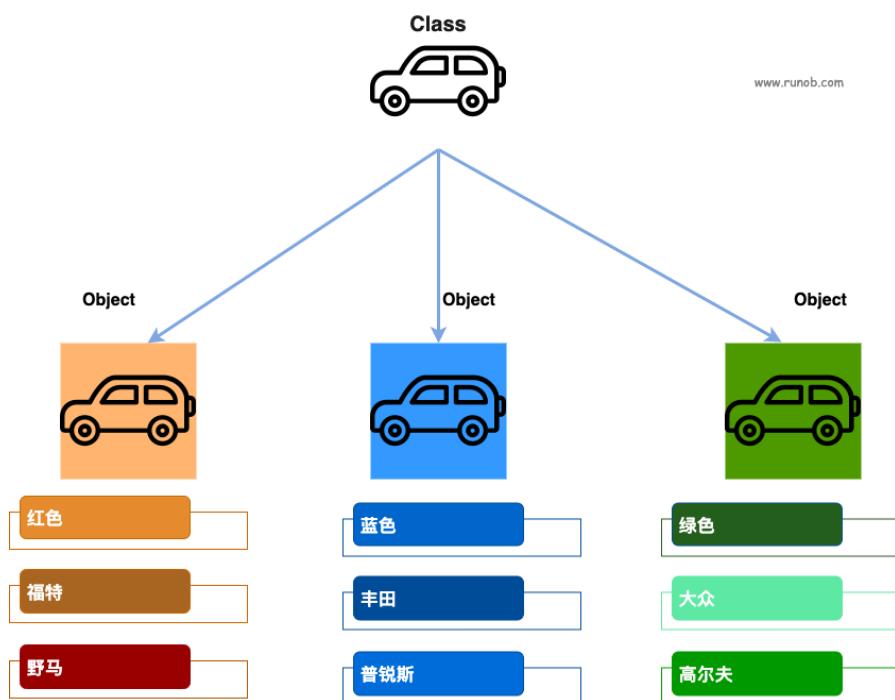
本节我们重点研究对象和类的概念。

- **对象**：对象是类的一个实例（**对象不是找个女朋友**），有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- **类**：类是一个模板，它描述一类对象的行为和状态。

下图中男孩 (boy) 、女孩 (girl) 为类 (class)，而具体的每个人为该类的对象 (object)：



下图中汽车为类（class），而具体的每个人车该类的对象（object），对象包含含来汽车的颜色、品牌、名称等：



## Java中的对象

现在让我们深入了解什么是对象。看看周围真实的世界，会发现身边有很多对象，车，狗，人等等。所有这些对象都有自己的状态和行为。

拿一条狗来举例，它的状态有：名字、品种、颜色，行为有：叫、摇尾巴和跑。

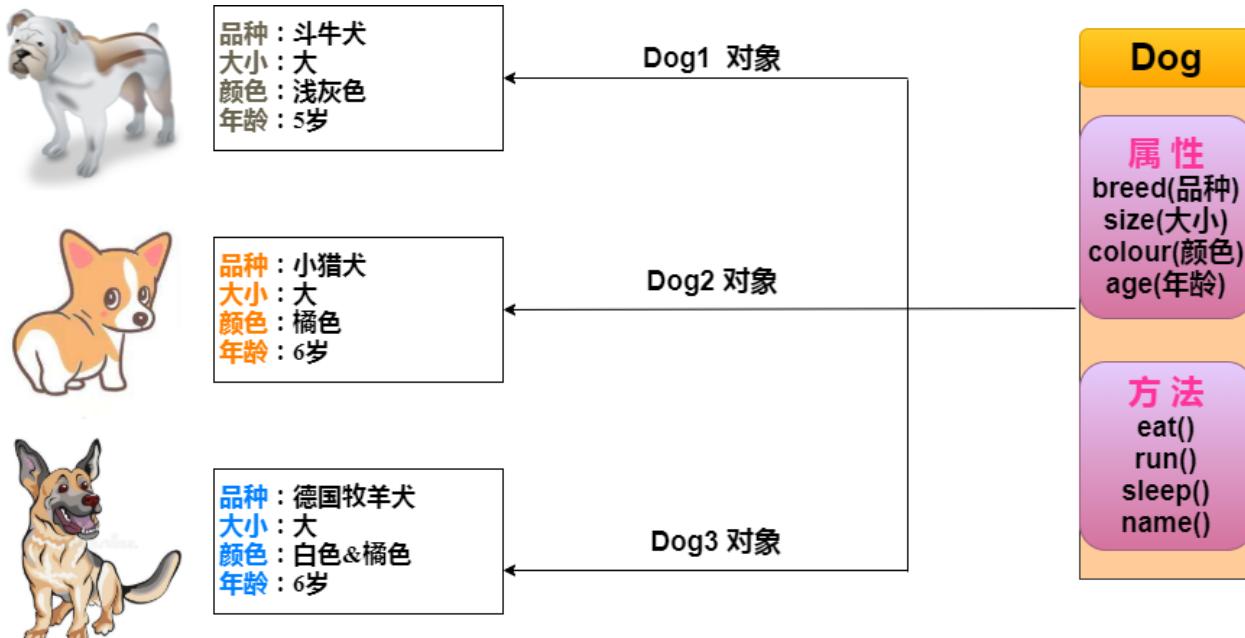
对比现实对象和软件对象，它们之间十分相似。

软件对象也有状态和行为。软件对象的状态就是属性，行为通过方法体现。

在软件开发中，方法操作对象内部状态的改变，对象的相互调用也是通过方法来完成。

## Java 中的类

类可以看成是创建 Java 对象的模板。



通过上图创建一个简单的类来理解下 Java 中类的定义：

```
public class Dog { String breed; int size; String colour; int age; void eat() {} void run() {} void sleep(){} void name(){} }
```

一个类可以包含以下类型变量：

- 局部变量**：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
- 成员变量**：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
- 类变量**：类变量也声明在类中，方法体之外，但必须声明为 static 类型。

一个类可以拥有多个方法，在上面的例子中：eat()、run()、sleep() 和 name() 都是 Dog 类的方法。

## 构造方法

每个类都有构造方法。如果没有显式地为类定义构造方法，Java 编译器将会为该类提供一个默认构造方法。

在创建一个对象的时候，至少要调用一个构造方法。构造方法的名称必须与类同名，一个类可以有多个构造方法。

下面是一个构造方法示例：

```
public class Puppy{ public Puppy(){ } public Puppy(String name){ //这个构造器仅有一个参数：name } }
```

---

## 创建对象

对象是根据类创建的。在Java中，使用关键字 new 来创建一个新的对象。创建对象需要以下三步：

- **声明**：声明一个对象，包括对象名称和对象类型。
- **实例化**：使用关键字 new 来创建一个对象。
- **初始化**：使用 new 创建对象时，会调用构造方法初始化对象。

下面是一个创建对象的例子：

```
public class Puppy{ public Puppy(String name){ //这个构造器仅有一个参数：name System.out.println("小狗的名字是：" + name ); } public static void main(String[] args){ //下面的语句将创建一个Puppy对象 Puppy myPuppy = new Puppy( "tommy" ); } }
```

编译并运行上面的程序，会打印出下面的结果：

小狗的名字是 : tommy

---

## 访问实例变量和方法

通过已创建的对象来访问成员变量和成员方法，如下所示：

```
/* 实例化对象 */ Object referenceVariable = new Constructor(); /* 访问类中的变量 */ referenceVariable.variableName; /* 访问类中的方法 */ referenceVariable.methodName();
```

---

## 实例

下面的例子展示如何访问实例变量和调用成员方法：

```
public class Puppy{ int puppyAge; public Puppy(String name){ //这个构造器仅有一个参数：name System.out.println("小狗的名字是：" + name ); } public void setAge( int age ){ puppyAge = age; } public int getAge( ){ System.out.println("小狗的年龄为：" + puppyAge ); return puppyAge; } public static void main(String[] args){ /* 创建对象 */ Puppy myPuppy = new Puppy( "tommy" ); /* 通过方法来设定age */ myPuppy.setAge( 2 ); /* 调用另一个方法获取age */ myPuppy.getAge( ); /* 你也可以像下面这样访问成员变量 */ System.out.println("变量值：" + myPuppy.puppyAge ); } }
```

编译并运行上面的程序，产生如下结果：

```
小狗的名字是 : tommy  
小狗的年龄为 : 2  
变量值 : 2
```

---

## 源文件声明规则

在本节的最后部分，我们将学习源文件的声明规则。当在一个源文件中定义多个类，并且还有 import 语句和 package 语句时，要特别注意这些规则。

- 一个源文件中只能有一个 public 类
- 一个源文件可以有多个非 public 类
- 源文件的名称应该和 public 类的类名保持一致。例如：源文件中 public 类的类名是 Employee，那么源文件应该命名为 Employee.java。
- 如果一个类定义在某个包中，那么 package 语句应该在源文件的首行。
- 如果源文件包含 import 语句，那么应该放在 package 语句和类定义之间。如果没有 package 语句，那么 import 语句应该在源文件中最前面。
- import 语句和 package 语句对源文件中定义的所有类都有效。在同一源文件中，不能给不同的类不同的包声明。

类有若干种访问级别，并且类也分不同的类型：抽象类和 final 类等。这些将在访问控制章节介绍。

除了上面提到的几种类型，Java 还有一些特殊的类，如：内部类、匿名类。

---

## Java 包

包主要用来对类和接口进行分类。当开发 Java 程序时，可能编写成百上千的类，因此很有必要对类和接口进行分类。

### import 语句

在 Java 中，如果给出一个完整的限定名，包括包名、类名，那么 Java 编译器就可以很容易地定位到源代码或者类。import 语句就是用来提供一个合理的路径，使得编译器可以找到某个类。

例如，下面的命令行将会命令编译器载入 java\_installation/java/io 路径下的所有类

```
import java.io.*;
```

---

## 一个简单的例子

在该例子中，我们创建两个类：Employee 和 EmployeeTest。

首先打开文本编辑器，把下面的代码粘贴进去。注意将文件保存为 Employee.java。

Employee 类有四个成员变量：name、age、designation 和 salary。该类显式声明了一个构造方法，该方法只有一个参数。

## **Employee.java 文件代码 :**

---

```
import java.io.*; public class Employee{ String name; int age; String designation; double salary; // Employee 类的构造器 public Employee(String name){ this.name = name; } // 设置age的值 public void empAge(int empAge){ age = empAge; } /* 设置designation的值 */ public void empDesignation(String empDesig){ designation = empDesig; } /* 设置salary的值 */ public void empSalary(double empSalary){ salary = empSalary; } /* 打印信息 */ public void printEmployee(){ System.out.println("名字:" + name ); System.out.println("年龄:" + age ); System.out.println("职位:" + designation ); System.out.println("薪水:" + salary); } }
```

程序都是从main方法开始执行。为了能运行这个程序，必须包含main方法并且创建一个实例对象。

下面给出EmployeeTest类，该类实例化2个 Employee 类的实例，并调用方法设置变量的值。

将下面的代码保存在 EmployeeTest.java文件中。

## **EmployeeTest.java 文件代码 :**

---

```
import java.io.*; public class EmployeeTest{ public static void main(String[] args){ /* 使用构造器创建两个对象 */ Employee empOne = new Employee("RUNOOB1"); Employee empTwo = new Employee("RUNOOB2"); // 调用这两个对象的成员方法 empOne.empAge(26); empOne.empDesignation("高级程序员"); empOne.empSalary(1000); empOne.printEmployee(); empTwo.empAge(21); empTwo.empDesignation("菜鸟程序员"); empTwo.empSalary(500); empTwo.printEmployee(); } }
```

编译这两个文件并且运行 EmployeeTest 类，可以看到如下结果：

```
$ javac EmployeeTest.java
$ java EmployeeTest
名字:RUNOOB1
年龄:26
职位:高级程序员
薪水:1000.0
名字:RUNOOB2
年龄:21
职位:菜鸟程序员
薪水:500.0
```

[Java 基础语法](#)

[Java 基本数据类型](#)

---

**15 篇笔记 写笔记**

# Java 基本数据类型 | 菜鸟教程

 [runoob.com/java/java-basic-datatypes.html](http://runoob.com/java/java-basic-datatypes.html)

## 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。

Java 的两大数据类型：

- 内置数据类型
- 引用数据类型

代码：

```
int x = 7;  
int y = 10;
```

内存：



## 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，不容易分辨。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';。

## **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

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 $3.14 \times 10^3 = 3140$ ，3.14E-3就是 $3.14 \times 10^{-3} = 0.00314$ 。

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。

## 类型默认值

下表列出了 Java 各个类型的默认值：

数据类型	默认值
byte	0
short	0
int	0
long	0L

---

float	0.0f
double	0.0d
char	'u0000'
String (or any object)	null
boolean	false

---

## 实例

---

```
public class Test { static boolean bool; static byte by; static char ch; static double d; static float f; static int i; static long l; static short sh; static String str; public static void main(String[] args) { System.out.println("Bool :" + bool); System.out.println("Byte :" + by); System.out.println("Character:" + ch); System.out.println("Double :" + d); System.out.println("Float :" + f); System.out.println("Integer :" + i); System.out.println("Long :" + l); System.out.println("Short :" + sh); System.out.println("String :" + str); } }
```

实例输出结果为：

```
Bool      :false
Byte     :0
Character:
Double   :0.0
Float    :0.0
Integer  :0
Long     :0
Short    :0
String   :null
```

---

## 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")。
- 

## Java 常量

---

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。

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。

当使用字面量的时候，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;
```

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\\"
```

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";
```

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 (0x0)
\s	空格 (0x20)
\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)

## 自动类型转换

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

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
```

## 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。

## 实例

```
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。

## 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：

### 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); } }
```

运行结果：

```
int强制类型转换为byte后的值等于123
```

## 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。

这一节讲解了 Java 的基本数据类型。下一节将探讨不同的变量类型以及它们的用法。

# Java 变量类型 | 菜鸟教程

 [runoob.com/java/java-variable-types.html](https://runoob.com/java/java-variable-types.html)

## Java 基本数据类型

### Java 修饰符

## Java 变量类型

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'。
```

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 } }
```

## 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，当它们执行完成后，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。

## 实例 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
```

## 实例 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
```

---

## 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：ObjectReference.VariableName。

## 实例

---

### 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(1000.0); empOne.printEmp(); } }
```

以上实例编译运行结果如下：

```
$ javac Employee.java  
$ java Employee  
名字：RUNOOB  
薪水：1000.0
```

## 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，静态变量是指声明为 public/private, final 和 static 类型的变量。静态变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，大多数静态变量声明为 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。

实例：

### 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); } }
```

以上实例编译运行结果如下：

```
开发人员平均工资：10000.0
```

注意：如果其他类想要访问该变量，可以这样访问：**Employee.DEPARTMENT**。

本章节中我们学习了Java的变量类型，下一章节中我们将介绍Java修饰符的使用。

[Java 基本数据类型](#)

[Java 修饰符](#)

## 12 篇笔记 写笔记

---

# Java 修饰符 | 菜鸟教程

 [runoob.com/java/java-modifier-types.html](https://runoob.com/java/java-modifier-types.html)

[Java 变量类型](#)

[Java 运算符](#)

## Java 修饰符

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

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 }
```

## 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** (即默认，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）。

我们可以通过以下表来说明访问权限：

修饰符	当前类	同一包内	子类(同一包)	子类(不同包)	其他包
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	Y/N (说明)	N
default	Y	Y	Y	N	N
private	Y	N	N	N	N

访问控制

## 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; }
```

## 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）

## 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

---

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` 可以修饰数据成员，构造方法，方法成员，**不能修饰类（内部类除外）**。

接口及接口的成员变量和成员方法不能声明为 `protected`。可以看看下图演示：

```
1 package com.runoob.test;
2
3 public interface Runoob {
4
5     default void fun() {}
6
7 }
```

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](#)。

## 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。

## 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。

## static 修饰符

---

- **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。

- **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。

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
```

## final 修饰符

---

- **final 变量：**

**final** 表示“最后的、最终的”含义，变量一旦赋值后，不能被重新赋值。被 **final** 修饰的实例变量必须显式指定初始值。

**final** 修饰符通常和 **static** 修饰符一起使用来创建类常量。

## 实例

---

```
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 } }
```

## final 方法

父类中的 final 方法可以被子类继承，但是不能被子类重写。

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 } }
```

## 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 }
```

## abstract 修饰符

---

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。

## 实例

---

```
abstract class Caravan{ private double price; private String model; private String year;  
public abstract void goFast(); //抽象方法 public abstract void changeColor(); }
```

## 抽象方法

抽象方法是一种没有任何实现的方法，该方法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();**

## 实例

---

```
public abstract class SuperClass{ abstract void m(); //抽象方法 } class SubClass extends  
SuperClass{ //实现抽象方法 void m(){ ..... } }
```

## **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。

### **实例**

---

```
public synchronized void showDetails(){ ..... }
```

## **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**

---

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。

### **实例**

---

```
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。

[Java 变量类型](#)

[Java 运算符](#)

## **12 篇笔记 写笔记**

---

# Java 运算符 | 菜鸟教程

 [runoob.com/java/java-operators.html](https://www.runoob.com/java/java-operators.html)

## Java 运算符

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

## 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。

表格中的实例假设整数变量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)

## 实例

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：

## 实例

```
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)); } }
```

### 运行实例»

以上实例编译运行结果如下：

```
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
```

## 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。

### 实例

```
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 实例：

### 实例

```
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
```

## 关系运算符

下表为Java支持的关系运算符

表格中的实例整数变量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。
$!=$	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。	$(A > B)$ 为假。
$<$	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。
$\geq$	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。	$(A \geq B)$ 为假。
$\leq$	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。	$(A \leq B)$ 为真。

## 实例

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)); } }
```

以上实例编译运行结果如下：

```
a == b = false  
a != b = true  
a > b = false  
a < b = true  
b >= a = true  
b <= a = false
```

## 位运算符

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
```

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。 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1111 A>>>2得到15即0000 1111

## 实例

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); } }
```

以上实例编译运行结果如下：

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 2 = 15
a >>> 2 = 15
```

## 逻辑运算符

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。

## 实例

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：

## 实例

```
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)); } }
```

以上实例编译运行结果如下：

```
a && b = false  
a || b = true  
!(a && b) = true
```

## 短路逻辑运算符

当使用与逻辑运算符时，在两个操作数都为true时，结果才为true，但是当得到第一个操作为false时，其结果就必定是false，这时候就不会再判断第二个操作了。

## 实例

```
public class LuoJi{ public static void main(String[] args){ int a = 5;//定义一个变量；  
boolean b = (a<4)&&(a++<10); System.out.println("使用短路逻辑运算符的结果为"+b);  
System.out.println("a的结果为"+a); } }
```

运行结果为：

```
使用短路逻辑运算符的结果为false  
a的结果为5
```

**解析：**该程序使用到了短路逻辑运算符(&&)，首先判断 a<4 的结果为 false，则 b 的结果必定是 false，所以不再执行第二个操作 a++<10 的判断，所以 a 的值为 5。

## 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	$C / = A, C$ 与 $A$ 同类型时等价于 $C = C / A$
$(\%) =$	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 $C = C \& 2$
$\wedge =$	按位异或赋值操作符	$C \wedge = 2$ 等价于 $C = C \wedge 2$
$  =$	按位或赋值操作符	$C   = 2$ 等价于 $C = C   2$

---

## 实例

下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 &= a; System.out.println("c &= a = " + c); c ^= a; System.out.println("c ^= a = " + c); c |= a; System.out.println("c |= a = " + c); } }
```

以上实例编译运行结果如下：

```
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
```

---

## 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。该运算符主要是决定哪个值应该赋值给变量。

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

## 实例

---

### 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 ); } }
```

以上实例编译运行结果如下：

```
Value of b is : 30  
Value of b is : 20
```

---

## instanceof 运算符

---

该运算符用于操作对象实例，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。

下面是一个例子：

```
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); } }
```

以上实例编译运行结果如下：

```
true
```

---

## Java运算符优先级

---

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。

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	expr++ expr--	从左到右
一元	++expr --expr + - ~ !	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

## 8 篇笔记 写笔记

# Java 循环结构 - for, while 及 do...while

---

 [runoob.com/java/java-loop.html](http://runoob.com/java/java-loop.html)

顺序结构的程序语句只能被执行一次。如果您想要同样的操作执行多次，就需要使用循环结构。

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。

---

## 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。

## 实例

---

### 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"); } } }
```

以上实例编译运行结果如下：

```
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
```

---

## 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。

## 实例

---

### 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 ); } }
```

以上实例编译运行结果如下：

```
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
```

---

## 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。

## 实例

---

### 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"); } } }
```

以上实例编译运行结果如下：

```
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
```

---

## 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。

## 实例

---

### 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(","); } } }
```

以上实例编译运行结果如下：

```
10, 20, 30, 40, 50,  
James, Larry, Tom, Lacy,
```

---

## 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。

## 语法

---

break 的用法很简单，就是循环结构中的一条语句：

```
break;
```

## 实例

---

### 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"); } } }
```

以上实例编译运行结果如下：

```
10  
20
```

---

## 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;
```

## 实例

---

### 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"); } } }
```

以上实例编译运行结果如下：

```
10  
20  
40  
50
```

## 5 篇笔记 写笔记

---



# Java 条件语句 – if...else | 菜鸟教程

---

 [runoob.com/java/java-if-else-switch.html](http://runoob.com/java/java-if-else-switch.html)

## Java 条件语句 - if...else

---

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
```

---

## 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 }
```

### 实例

---

### 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
```

---

## if...else if...else 语句

---

if 语句后面可以跟 else if...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 else if 语句之后。
- if 语句可以有若干个 else if 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 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 }
```

## 实例

---

### 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
```

---

## 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 else if 语句中使用 if 或者 else if 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。

## 实例

---

### 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

## 5 篇笔记 写笔记

---

# Java switch case 语句 | 菜鸟教程

 [runoob.com/java/java-switch-case.html](https://runoob.com/java/java-switch-case.html)

## Java switch case 语句

switch case 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 case 语句语法格式如下：

```
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 case 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 String 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 语句不一定要包含 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，该分支一般是 switch 语句的最后一个分支（可以在任何位置，但建议在最后一个）。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。

**switch case 执行时，一定会先进行匹配，匹配成功返回当前 case 的值，再根据是否有 break，判断是否继续输出，或是跳出判断。**

### 实例

**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"); break; 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
```

如果 case 语句块中没有 break 语句时，JVM 并不会顺序输出每一个 case 对应的返回值，而是继续匹配，匹配不成功则返回默认 case。

### Test.java 文件代码：

---

```
public class Test { public static void main(String args[]){ int i = 5; switch(i){ case 0:  
System.out.println("0"); case 1: System.out.println("1"); case 2: System.out.println("2");  
default: System.out.println("default"); } } }
```

以上代码编译运行结果如下：

```
default
```

如果 case 语句块中没有 break 语句时，匹配成功后，从当前 case 开始，后续所有 case 的值都会输出。

### Test.java 文件代码：

---

```
public class Test { public static void main(String args[]){ int i = 1; switch(i){ case 0:  
System.out.println("0"); case 1: System.out.println("1"); case 2: System.out.println("2");  
default: System.out.println("default"); } } }
```

以上代码编译运行结果如下：

```
1  
2  
default
```

如果当前匹配成功的 case 语句块没有 break 语句，则从当前 case 开始，后续所有 case 的值都会输出，如果后续的 case 语句块有 break 语句则会跳出判断。

### Test.java 文件代码：

---

```
public class Test { public static void main(String args[]){ int i = 1; switch(i){ case 0:  
System.out.println("0"); case 1: System.out.println("1"); case 2: System.out.println("2");  
case 3: System.out.println("3"); break; default: System.out.println("default"); } } }
```

以上代码编译运行结果如下：

```
1  
2  
3
```



# Java Number & Math 类

 [runoob.com/java/java-number.html](http://runoob.com/java/java-number.html)

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

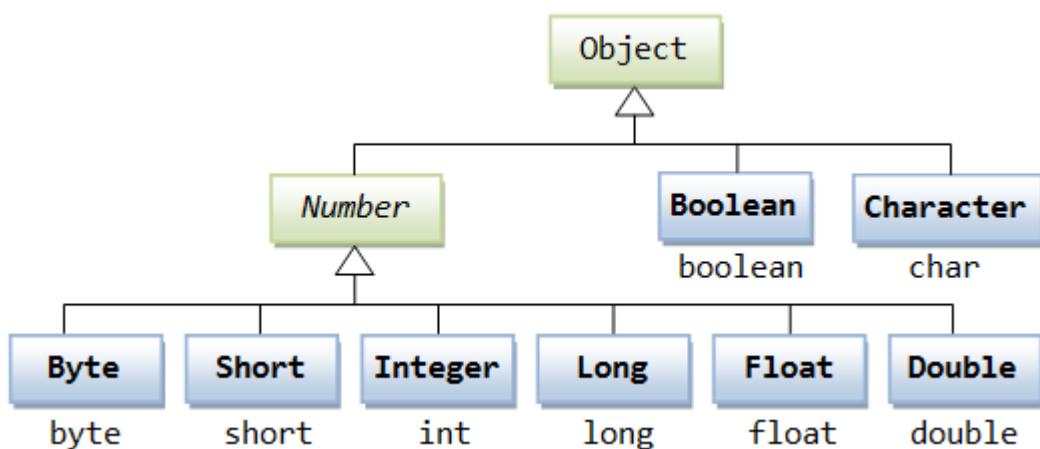
```
int a = 5000; float b = 13.65f; byte c = 0x4a;
```

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。

### 包装类 基本数据类型

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



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。

---

### 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
```

---

### 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\(\)](#)  
将 Number 对象转换为xxx数据类型的值并返回。
- 
- 2 [compareTo\(\)](#)  
将number对象与参数比较。
- 
- 3 [equals\(\)](#)  
判断number对象是否与参数相等。
- 
- 4 [valueOf\(\)](#)  
返回一个 Number 对象指定的内置数据类型
- 
- 5 [toString\(\)](#)  
以字符串形式返回值。
- 
- 6 [parseInt\(\)](#)  
将字符串解析为int类型。
- 
- 7 [abs\(\)](#)  
返回参数的绝对值。
- 
- 8 [ceil\(\)](#)  
返回大于等于(  $\geq$  )给定参数的最小整数，类型为双精度浮点型。
- 
- 9 [floor\(\)](#)  
返回小于等于 (  $\leq$  ) 给定参数的最大整数。
- 
- 10 [rint\(\)](#)  
返回与参数最接近的整数。返回类型为double。
- 
- 11 [round\(\)](#)  
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\(\)](#)  
返回两个参数中的最小值。
- 
- 13 [max\(\)](#)  
返回两个参数中的最大值。
- 
- 14 [exp\(\)](#)  
返回自然数底数e的参数次方。
- 
- 15 [log\(\)](#)  
返回参数的自然数底数的对数值。
- 
- 16 [pow\(\)](#)  
返回第一个参数的第二个参数次方。
- 
- 17 [sqrt\(\)](#)  
求参数的算术平方根。
-

---

18 [sin\(\)](#)

求指定double类型参数的正弦值。

---

19 [cos\(\)](#)

求指定double类型参数的余弦值。

---

20 [tan\(\)](#)

求指定double类型参数的正切值。

---

21 [asin\(\)](#)

求指定double类型参数的反正弦值。

---

22 [acos\(\)](#)

求指定double类型参数的反余弦值。

---

23 [atan\(\)](#)

求指定double类型参数的反正切值。

---

24 [atan2\(\)](#)

将笛卡尔坐标转换为极坐标，并返回极坐标的角度值。

---

25 [toDegrees\(\)](#)

将参数转化为角度。

---

26 [toRadians\(\)](#)

将角度转换为弧度。

---

27 [random\(\)](#)

返回一个随机数。

---

## 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
```

## 6 篇笔记 写笔记

---

# Java Character 类 | 菜鸟教程

 [runoob.com/java/java-character.html](https://runoob.com/java/java-character.html)

## Java Character 类

Character 类用于对单个字符进行操作。

Character 类在对象中包装一个基本类型 **char** 的值

### 实例

```
char ch = 'a'; // Unicode 字符表示形式 char uniChar = '\u039A'; // 字符数组 char[]  
charArray = { 'a', 'b', 'c', 'd', 'e' };
```

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况。为了解决这个问题，Java语言为内置数据类型char提供了包装类Character类。

Character类提供了一系列方法来操纵字符。你可以使用Character的构造方法创建一个Character类对象，例如：

```
Character ch = new Character('a');
```

在某些情况下，Java编译器会自动创建一个Character对象。

例如，将一个char类型的参数传递给需要一个Character类型参数的方法时，那么编译器会自动地将char类型参数转换为Character对象。这种特征称为装箱，反过来称为拆箱。

### 实例

```
// 原始字符 'a' 装箱到 Character 对象 ch 中 Character ch = 'a'; // 原始字符 'x' 用 test 方法装  
// 箱 // 返回拆箱的值到 'c' char c = test('x');
```

## 转义序列

前面有反斜杠 (\) 的字符代表转义字符，它对编译器来说是有特殊含义的。

下面列表展示了Java的转义序列：

### 转义序列 描述

\t 在文中该处插入一个tab键

\b 在文中该处插入一个后退键

\n 在文中该处换行

\r 在文中该处插入回车

---

\f	在文中该处插入换页符
\'	在文中该处插入单引号
\"	在文中该处插入双引号
\\\	在文中该处插入反斜杠

---

## 实例

---

当打印语句遇到一个转义序列时，编译器可以正确地对其进行解释。

以下实例转义双引号并输出：

### Test.java 文件代码：

---

```
public class Test { public static void main(String args[]) { System.out.println("访问\"菜鸟教程!\""); } }
```

以上实例编译运行结果如下：

```
访问"菜鸟教程!"
```

---

## Character 方法

---

下面是Character类的方法：

### 序号 方法与描述

---

1	<u>isLetter()</u> 是否是一个字母
2	<u>isDigit()</u> 是否是一个数字字符
3	<u>isWhitespace()</u> 是否是一个空白字符
4	<u>isUpperCase()</u> 是否是大写字母
5	<u>isLowerCase()</u> 是否是小写字母
6	<u>toUpperCase()</u> 指定字母的大写形式

---

---

7 [toLowerCase\(\)](#)  
指定字母的小写形式

---

8 [toString\(\)](#)  
返回字符的字符串形式，字符串的长度仅为1

对于方法的完整列表，请参考的 [java.lang.Character API](#) 规范。

## 5 篇笔记 写笔记

---

# Java String 类 | 菜鸟教程

 [runoob.com/java/java-string.html](http://runoob.com/java/java-string.html)

## Java String 类

字符串广泛应用于 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。

### 创建字符串

创建字符串最简单的方式如下：

```
String str = "Runoob";
```

在代码中遇到字符串常量时，这里的值是 "Runoob""，编译器会使用该值创建一个 String 对象。

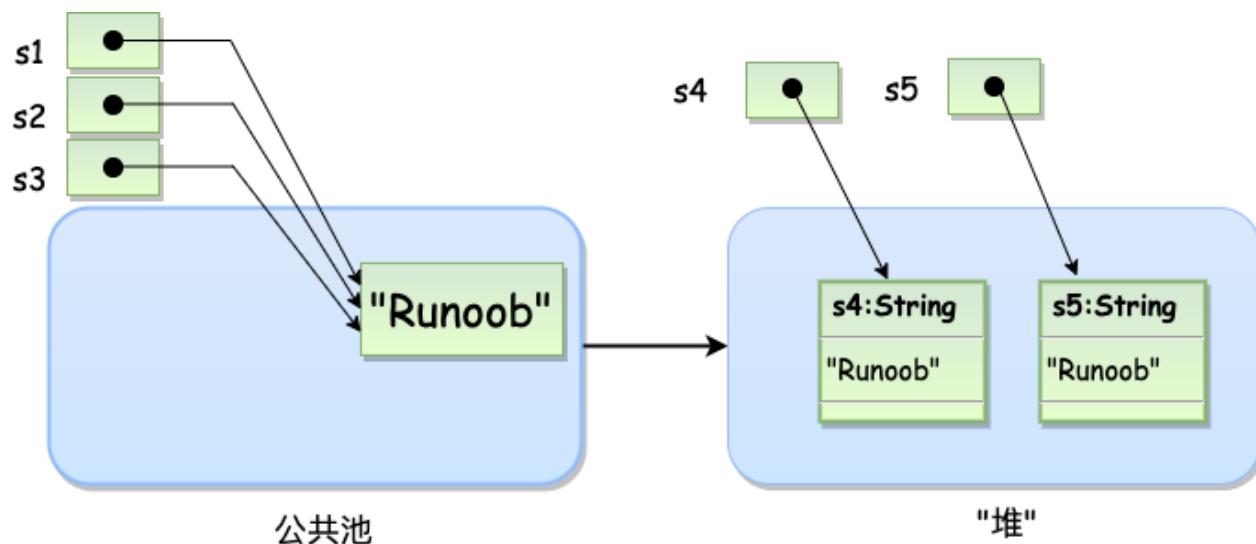
和其他对象一样，可以使用关键字和构造方法来创建 String 对象。

用构造函数创建字符串：

```
String str2=new String("Runoob");
```

String 创建的字符串存储在公共池中，而 new 创建的字符串对象在堆上：

```
String s1 = "Runoob"; // String 直接创建 String s2 = "Runoob"; // String 直接创建 String  
s3 = s1; // 相同引用 String s4 = new String("Runoob"); // String 对象创建 String s5 = new  
String("Runoob"); // String 对象创建
```



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。

---

## 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

---

## 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

## 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 " + "%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);
```

## String 方法

下面是 String 类支持的方法，更多详细，参看 [Java String API 文档](#):

### SN(序号)

### 方法描述

1 [char charAt\(int index\)](#)  
返回指定索引处的 char 值。

2 [int compareTo\(Object o\)](#)  
把这个字符串和另一个对象比较。

3 [int compareTo\(String anotherString\)](#)  
按字典顺序比较两个字符串。

4 [int compareToIgnoreCase\(String str\)](#)  
按字典顺序比较两个字符串，不考虑大小写。

- 
- 5      [String concat\(String str\)](#)  
        将指定字符串连接到此字符串的结尾。
- 
- 6      [boolean contentEquals\(StringBuffer sb\)](#)  
        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\)](#)  
        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\)](#)  
        返回指定数组中表示该字符序列的 String。
- 
- 9      [boolean endsWith\(String suffix\)](#)  
        测试此字符串是否以指定的后缀结束。
- 
- 10     [boolean equals\(Object anObject\)](#)  
        将此字符串与指定的对象比较。
- 
- 11     [boolean equalsIgnoreCase\(String anotherString\)](#)  
        将此 String 与另一个 String 比较，不考虑大小写。
- 
- 12     [byte\[\].getBytes\(\)](#)  
        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\(String charsetName\)](#)  
        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\)](#)  
        将字符从此字符串复制到目标字符数组。
- 
- 15     [int hashCode\(\)](#)  
        返回此字符串的哈希码。
- 
- 16     [int indexOf\(int ch\)](#)  
        返回指定字符在此字符串中第一次出现处的索引。
- 
- 17     [int indexOf\(int ch, int fromIndex\)](#)  
        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\(String str\)](#)  
        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\(String str, int fromIndex\)](#)  
        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\(\)](#)  
        返回字符串对象的规范化表示形式。
- 
- 21     [int lastIndexOf\(int ch\)](#)  
        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\(int ch, int fromIndex\)](#)  
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\(String str\)](#)  
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\(String str, int fromIndex\)](#)  
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\(\)](#)  
返回此字符串的长度。
- 
- 26 [boolean matches\(String regex\)](#)  
告知此字符串是否匹配给定的正则表达式。
- 
- 27 [boolean regionMatches\(boolean ignoreCase, int toffset, String other, int ooffset, int len\)](#)  
测试两个字符串区域是否相等。
- 
- 28 [boolean regionMatches\(int toffset, String other, int ooffset, int len\)](#)  
测试两个字符串区域是否相等。
- 
- 29 [String replace\(char oldChar, char newChar\)](#)  
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\(String regex, String replacement\)](#)  
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\(String regex, String replacement\)](#)  
使用给定的 replacement 替换此字符串匹配给定的正则表达式的一个子字符串。
- 
- 32 [String\[\] split\(String regex\)](#)  
根据给定正则表达式的匹配拆分此字符串。
- 
- 33 [String\[\] split\(String regex, int limit\)](#)  
根据匹配给定的正则表达式来拆分此字符串。
- 
- 34 [boolean startsWith\(String prefix\)](#)  
测试此字符串是否以指定的前缀开始。
- 
- 35 [boolean startsWith\(String prefix, int toffset\)](#)  
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\(int beginIndex, int endIndex\)](#)  
返回一个新的字符序列，它是此序列的一个子序列。
- 
- 37 [String substring\(int beginIndex\)](#)  
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\(int beginIndex, int endIndex\)](#)  
        返回一个新字符串，它是此字符串的一个子字符串。
- 
- 39      [char\[\] toCharArray\(\)](#)  
        将此字符串转换为一个新的字符数组。
- 
- 40      [String toLowerCase\(\)](#)  
        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\(Locale locale\)](#)  
        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\(\)](#)  
        返回此对象本身（它已经是一个字符串！）。
- 
- 43      [String toUpperCase\(\)](#)  
        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\(Locale locale\)](#)  
        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\(\)](#)  
        返回字符串的副本，忽略前导空白和尾部空白。
- 
- 46      [static String valueOf\(primitive data type x\)](#)  
        返回给定 data type 类型 x 参数的字符串表示形式。
- 
- 47      [contains\(CharSequence chars\)](#)  
        判断是否包含指定的字符系列。
- 
- 48      [isEmpty\(\)](#)  
        判断字符串是否为空。

---

## 9 篇笔记 写笔记

---

# Java StringBuffer 和 StringBuilder 类

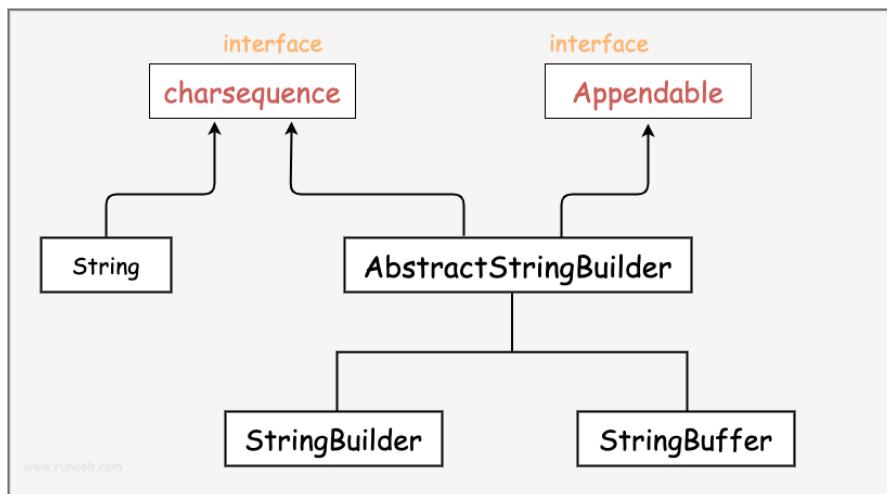
 [runoob.com/java/java-stringbuffer.html](http://runoob.com/java/java-stringbuffer.html)

## Java String 类

### Java 数组

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。



在使用 StringBuffer 类时，每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，所以如果需要对字符串进行修改推荐使用 StringBuffer。

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）。

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。

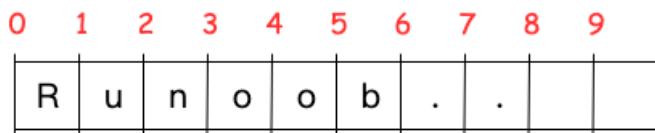
## 实例

```
public class RunoobTest{ public static void main(String args[]){ StringBuilder sb = new
StringBuilder(10); sb.append("Runoob.."); System.out.println(sb); sb.append("!");
System.out.println(sb); sb.insert(8, "Java"); System.out.println(sb); sb.delete(5,8);
System.out.println(sb); } }
```

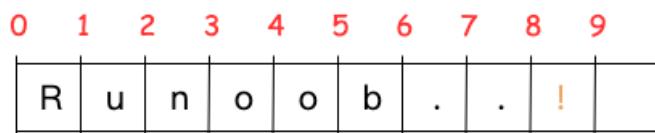
```
StringBuilder sb = new StirngBuilder(10);
```



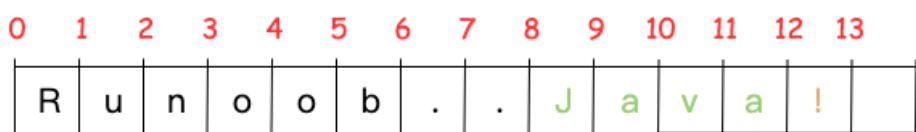
```
sb.append("Runoob..");
```



```
sb.append('!');
```



```
sb.insert(8,"Java") ;
```



```
sb.delete(5,8);
```



以上实例编译运行结果如下：

```
Runoob..  
Runoob..!  
Runoob..Java!  
RunooJava!
```

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

## 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。

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)

返回最右端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。

[Java String 类](#)

[Java 数组](#)

### 3 篇笔记 写笔记

---

# Java 数组 | 菜鸟教程

---

 [runoob.com/java/java-array.html](http://runoob.com/java/java-array.html)

## Java 数组

---

数组对于每一门编程语言来说都是重要的数据结构之一，当然不同语言对数组的实现及处理也不尽相同。

Java 语言中提供的数组是用来存储固定大小的同类型元素。

你可以声明一个数组变量，如 numbers[100] 来代替直接声明 100 个独立变量 numero, number1, ..., number99。

本教程将为大家介绍 Java 数组的声明、创建和初始化，并给出其对应的代码。

---

## 声明数组变量

---

首先必须声明数组变量，才能在程序中使用数组。下面是声明数组变量的语法：

```
dataType[] arrayRefVar; // 首选的方法 或 dataType arrayRefVar[]; // 效果相同，但不是首选方法
```

**注意：**建议使用 **dataType[] arrayRefVar** 的声明风格声明数组变量。 dataType arrayRefVar[] 风格是来自 C/C++ 语言，在 Java 中采用是为了让 C/C++ 程序员能够快速理解 java 语言。

## 实例

---

下面是这两种语法的代码示例：

```
double[] myList; // 首选的方法 或 double myList[]; // 效果相同，但不是首选方法
```

---

## 创建数组

---

Java 语言使用 new 操作符来创建数组，语法如下：

```
arrayRefVar = new dataType[arraySize];
```

上面的语法语句做了两件事：

- 一、使用 dataType[arraySize] 创建了一个数组。
- 二、把新创建的数组的引用赋值给变量 arrayRefVar。

数组变量的声明，和创建数组可以用一条语句完成，如下所示：

```
dataType[] arrayRefVar = new dataType[arraySize];
```

另外，你还可以使用如下的方式创建数组。

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

数组的元素是通过索引访问的。数组索引从 0 开始，所以索引值从 0 到 `arrayRefVar.length - 1`。

## 实例

下面的语句首先声明了一个数组变量 `myList`，接着创建了一个包含 10 个 double 类型元素的数组，并且把它的引用赋值给 `myList` 变量。

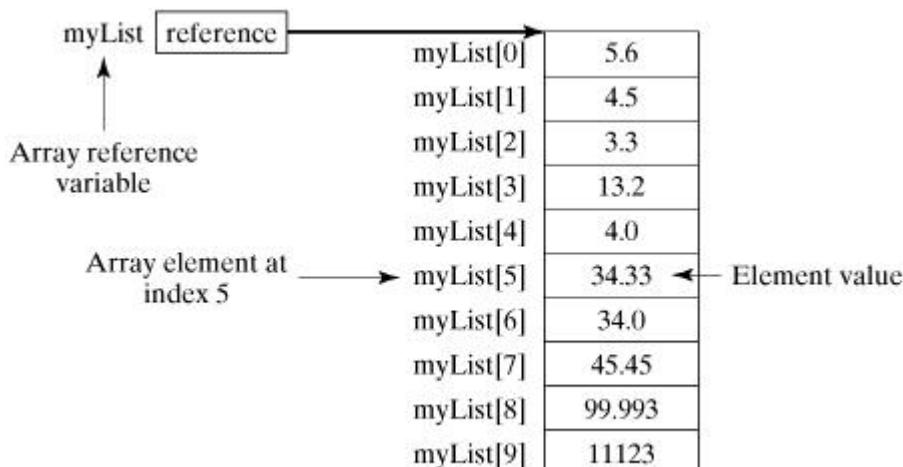
### TestArray.java 文件代码：

```
public class TestArray { public static void main(String[] args) { // 数组大小 int size = 10;  
// 定义数组 double[] myList = new double[size]; myList[0] = 5.6; myList[1] = 4.5;  
myList[2] = 3.3; myList[3] = 13.2; myList[4] = 4.0; myList[5] = 34.33; myList[6] = 34.0;  
myList[7] = 45.45; myList[8] = 99.993; myList[9] = 11123; // 计算所有元素的总和 double  
total = 0; for (int i = 0; i < size; i++) { total += myList[i]; } System.out.println("总和为："  
+ total); } }
```

以上实例输出结果为：

总和为： 11367.373

下面的图片描绘了数组 `myList`。这里 `myList` 数组里有 10 个 double 元素，它的下标从 0 到 9。



## 处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本循环或者 For-Each 循环。

## 示例

该实例完整地展示了如何创建、初始化和操纵数组：

## TestArray.java 文件代码：

---

```
public class TestArray { public static void main(String[] args) { double[] myList = {1.9, 2.9, 3.4, 3.5}; // 打印所有数组元素 for (int i = 0; i < myList.length; i++) { System.out.println(myList[i] + " "); } // 计算所有元素的总和 double total = 0; for (int i = 0; i < myList.length; i++) { total += myList[i]; } System.out.println("Total is " + total); // 查找最大元素 double max = myList[0]; for (int i = 1; i < myList.length; i++) { if (myList[i] > max) max = myList[i]; } System.out.println("Max is " + max); } }
```

以上实例编译运行结果如下：

```
1.9  
2.9  
3.4  
3.5  
Total is 11.7  
Max is 3.5
```

---

## For-Each 循环

---

JDK 1.5 引进了一种新的循环类型，被称为 For-Each 循环或者加强型循环，它能在不使用下标的情况下遍历数组。

语法格式如下：

```
for(type element: array)  
{  
    System.out.println(element);  
}
```

## 实例

---

该实例用来显示数组 myList 中的所有元素：

## TestArray.java 文件代码：

---

```
public class TestArray { public static void main(String[] args) { double[] myList = {1.9, 2.9, 3.4, 3.5}; // 打印所有数组元素 for (double element: myList) { System.out.println(element); } } }
```

以上实例编译运行结果如下：

```
1.9  
2.9  
3.4  
3.5
```

---

## 数组作为函数的参数

---

数组可以作为参数传递给方法。

例如，下面的例子就是一个打印 int 数组中元素的方法：

```
public static void printArray(int[] array) { for (int i = 0; i < array.length; i++) { System.out.print(array[i] + " "); } }
```

下面例子调用 printArray 方法打印出 3, 1, 2, 6, 4 和 2：

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

---

## 数组作为函数的返回值

```
public static int[] reverse(int[] list) { int[] result = new int[list.length]; for (int i = 0, j = result.length - 1; i < list.length; i++, j--) { result[j] = list[i]; } return result; }
```

以上实例中 result 数组作为函数的返回值。

---

## 多维数组

多维数组可以看成是数组的数组，比如二维数组就是一个特殊的一维数组，其每一个元素都是一个一维数组，例如：

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

### 多维数组的动态初始化（以二维数组为例）

1. 直接为每一维分配空间，格式如下：

```
type[][] typeName = new type[typeLength1][typeLength2];
```

type 可以为基本数据类型和复合数据类型，arraylength1 和 arraylength2 必须为正整数，arraylength1 为行数，arraylength2 为列数。

例如：

```
int a[][] = new int[2][3];
```

解析：

二维数组 a 可以看成一个两行三列的数组。

2. 从最高维开始，分别为每一维分配空间，例如：

```
String s[][] = new String[2][]; s[0] = new String[2]; s[1] = new String[3]; s[0][0] = new String("Good"); s[0][1] = new String("Luck"); s[1][0] = new String("to"); s[1][1] = new String("you"); s[1][2] = new String("!");
```

解析：

**s[0]=new String[2]** 和 **s[1]=new String[3]** 是为最高维分配引用空间，也就是为最高维限制其能保存数据的最长的长度，然后再为其每个数组元素单独分配空间 **so=new String("Good")** 等操作。

## 多维数组的引用（以二维数组为例）

对二维数组中的每个元素，引用方式为 **arrayName[index1][index2]**，例如：

num[1][0];

## Arrays 类

java.util.Arrays 类能方便地操作数组，它提供的所有方法都是静态的。

具有以下功能：

- 给数组赋值：通过 **fill** 方法。
- 对数组排序：通过 **sort** 方法，按升序。
- 比较数组：通过 **equals** 方法比较数组中元素值是否相等。
- 查找数组元素：通过 **binarySearch** 方法能对排序好的数组进行二分查找法操作。

具体说明请查看下表：

序号	方法和说明
1	<b>public static int binarySearch(Object[] a, Object key)</b> 用二分查找算法在给定数组中搜索给定值的对象(Byte, Int, double等)。数组在调用前必须排序好的。如果查找值包含在数组中，则返回搜索键的索引；否则返回(-(插入点)-1)。
2	<b>public static boolean equals(long[] a, long[] a2)</b> 如果两个指定的 long 型数组彼此相等，则返回 true。如果两个数组包含相同数量的元素，并且两个数组中的所有相应元素对都是相等的，则认为这两个数组是相等的。换句话说，如果两个数组以相同顺序包含相同的元素，则两个数组是相等的。同样的方法适用于所有的其他基本数据类型 (Byte, short, Int等)。
3	<b>public static void fill(int[] a, int val)</b> 将指定的 int 值分配给指定 int 型数组指定范围中的每个元素。同样的方法适用于所有的其他基本数据类型 (Byte, short, Int等)。
4	<b>public static void sort(Object[] a)</b> 对指定对象数组根据其元素的自然顺序进行升序排列。同样的方法适用于所有的其他基本数据类型 (Byte, short, Int等)。

### 练习

#### Java 数组测验

## 11 篇笔记 写笔记

---

# Java 日期时间 | 菜鸟教程

 [runoob.com/java/java-date-time.html](http://runoob.com/java/java-date-time.html)

[Java 数组](#)

[Java 正则表达式](#)

## Java 日期时间

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实例化 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)

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)。

---

## **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：

### **实例**

```
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()); } }
```

#### 运行实例»

以上实例编译运行结果如下：

```
Mon May 04 09:51:52 CDT 2013
```

---

## **日期比较**

Java使用以下三种方法来比较两个日期：

- 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。
- 

## **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：

### **实例**

```
import java.util.*; import java.text.*; public class DateDemo { public static void main(String args[]) { Date dNow = new Date(); SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd hh:mm:ss"); System.out.println("当前时间为：" +
```

```
ft.format(dNow)); } }
```

### 运行实例 »

```
SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd hh:mm:ss");
```

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。

以上实例编译运行结果如下：

```
当前时间为： 2018-09-06 10:16:34
```

---

## 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	一天中的小時(1~24)	24
K	A.M./P.M. (0~11)格式小時	10
z	時区	Eastern Standard Time
'	文字定界符	Delimiter
"	单引号	'

## 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時制)	02:25:51 下午
T "HH:MM:SS"格式 (24時制)	14:28:16
R "HH:MM"格式 (24時制)	14:28

更多 **printf** 解析可以参见：[Java 格式化输出 printf 例子](#)

## 实例

### 实例

```
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時制) : %tr%n",date); //t的使用  
System.out.printf("HH:MM:SS格式 (24時制) : %tT%n",date); //R的使用  
System.out.printf("HH:MM格式 (24時制) : %tR",date); } }
```

以上实例编译运行结果如下：

```
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時制) : 10:43:36 上午  
HH:MM:SS格式 (24時制) : 10:43:36  
HH:MM格式 (24時制) : 10:43
```

如果你需要重复提供日期，那么利用这种方式来格式化它的每一部分就有点复杂了。因此，可以利用一个格式化字符串指出要被格式化的参数的索引。

索引必须紧跟在%后面，而且必须以\$结束。例如：

## 实例

---

```
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() 显示日期和时间  
System.out.printf("%1$s %2$tB %2$td, %2$tY", "Due date:", date); } }
```

### 运行实例»

以上实例编译运行结果如下：

```
Due date: February 09, 2014
```

或者，你可以使用 < 标志。它表明先前被格式化的参数要被再次使用。例如：

## 实例

---

```
import java.util.Date; public class DateDemo { public static void main(String args[]) { //  
初始化 Date 对象 Date date = new Date(); // 显示格式化时间 System.out.printf("%s %tB %  
<te, %<tY", "Due date:", date); } }
```

### 运行实例»

以上实例编译运行结果如下：

```
Due date: February 09, 2014
```

定义日期格式的转换符可以使日期通过指定的转换符生成新字符串。这些日期转换符如下所示：

## 实例

---

```
import java.util.*; public class DateDemo { public static void main(String args[]) { Date date=new Date(); //b的使用, 月份简称 String str=String.format(Locale.US,"英文月份简称 : %tb",date); System.out.println(str); System.out.printf("本地月份简称 : %tb%n",date); //B的使用, 月份全称 str=String.format(Locale.US,"英文月份全称 : %tB",date); System.out.println(str); System.out.printf("本地月份全称 : %tB%n",date); //a的使用, 星期简称 str=String.format(Locale.US,"英文星期的简称 : %ta",date); System.out.println(str); //A的使用, 星期全称 System.out.printf("本地星期的简称 : %tA%n",date); //C的使用, 年前两位 System.out.printf("年的前两位数字 (不足两位前面补0) : %tC%n",date); //y的使用, 年后两位 System.out.printf("年的后两位数字 (不足两位前面补0) : %ty%n",date); //j的使用, 一年的天数 System.out.printf("一年中的天数 (即年的第几天) : %tj%n",date); //m的使用, 月份 System.out.printf("两位数字的月份 (不足两位前面补0) : %tm%n",date); //d的使用, 日 (二位, 不够补零) System.out.printf("两位数字的日 (不足两位前面补0) : %td%n",date); //e的使用, 日 (一位不补零) System.out.printf("月份的日 (前面不补0) : %te",date); } }
```

输出结果为：

```
英文月份简称：May  
本地月份简称：五月  
英文月份全称：May  
本地月份全称：五月  
英文星期的简称：Thu  
本地星期的简称：星期四  
年的前两位数字 (不足两位前面补0) : 20  
年的后两位数字 (不足两位前面补0) : 17  
一年中的天数 (即年的第几天) : 124  
两位数字的月份 (不足两位前面补0) : 05  
两位数字的日 (不足两位前面补0) : 04  
月份的日 (前面不补0) : 4
```

---

## 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：

---

### 实例

---

```
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); } } }
```

---

#### 运行实例»

以上实例编译运行结果如下：

```
$ 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
```

---

## Java 休眠(sleep)

sleep()使当前线程进入停滞状态（阻塞当前线程），让出CPU的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会。

你可以让程序休眠一毫秒的时间或者到您的计算机的寿命长的任意段时间。例如，下面的程序会休眠3秒：

### 实例

```
import java.util.*; public class SleepDemo { public static void main(String args[]) { try {  
    System.out.println(new Date() + "\n"); Thread.sleep(1000*3); // 休眠3秒  
    System.out.println(new Date() + "\n"); } catch (Exception e) { System.out.println("Got  
    an exception!"); } } }
```

#### 运行实例»

以上实例编译运行结果如下：

```
Thu Sep 17 10:20:30 CST 2015
```

```
Thu Sep 17 10:20:33 CST 2015
```

---

### 测量时间

下面的一个例子表明如何测量时间间隔（以毫秒为单位）：

### 实例

```
import java.util.*; public class DiffDemo { public static void main(String args[]) { try {  
    long start = System.currentTimeMillis(); System.out.println(new Date() + "\n");  
    Thread.sleep(5*60*10); System.out.println(new Date() + "\n"); long end =  
    System.currentTimeMillis(); long diff = end - start; System.out.println("Difference is :" +  
    diff); } catch (Exception e) { System.out.println("Got an exception!"); } } }
```

#### 运行实例»

以上实例编译运行结果如下：

```
Fri Jan 08 09:48:47 CST 2016
```

```
Fri Jan 08 09:48:50 CST 2016
```

```
Difference is : 3019
```

## 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);
```

### 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

### Calendar类对象信息的设置

#### Set设置

如：

```
Calendar c1 = Calendar.getInstance();
```

调用：

```
public final void set(int year,int month,int date)  
c1.set(2009, 6, 12); //把Calendar对象c1的年月日分别设为：2009、6、12
```

利用字段类型设置

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);
```

其他字段属性的add的意义以此类推

## 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代表星期一、3代表星期二，以此类推） int day = c1.get(Calendar.DAY_OF_WEEK);
```

## 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。

下面列出 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。

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)**  
    获取指定字段的時間值
- 
- 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) 单个時間单元, 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。

## 实例

---

### 实例

---

```
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"); } } }
```

#### 运行实例 »

以上实例编译运行结果如下：

```
Date: Apr 22 2009  
Time: 11:25:27  
当前年份不是闰年
```

关于 Calender 壳的完整列表，你可以参考标准的 [Java 文档](#)。

[Java 数组](#)

[Java 正则表达式](#)

## 2 篇笔记 写笔记

---

## 1. 女芳单身骗

lhm\*\*\*@126.com

136

Calender的月份是从0开始的，但日期和年份是从1开始的

示例代码：

```
import java.util.Calendar;

public class Test {
    public static void main(String[] args) {
        Calendar c1 = Calendar.getInstance();
        c1.set(2017, 1, 1);
        System.out.println(c1.get(Calendar.YEAR)
            + "-" + c1.get(Calendar.MONTH)
            + "-" + c1.get(Calendar.DATE));
        c1.set(2017, 1, 0);
        System.out.println(c1.get(Calendar.YEAR)
            + "-" + c1.get(Calendar.MONTH)
            + "-" + c1.get(Calendar.DATE));
    }
}
```

运行结果：

```
2017-1-1
2017-0-31
```

可见，将日期设为0以后，月份变成了上个月，但月份可以为0

把月份改为2试试：

```
import java.util.Calendar;

public class Test {
    public static void main(String[] args) {
        Calendar c1 = Calendar.getInstance();
        c1.set(2017, 2, 1);
        System.out.println(c1.get(Calendar.YEAR)
            + "-" + c1.get(Calendar.MONTH)
            + "-" + c1.get(Calendar.DATE));
        c1.set(2017, 2, 0);
        System.out.println(c1.get(Calendar.YEAR)
            + "-" + c1.get(Calendar.MONTH)
            + "-" + c1.get(Calendar.DATE));
    }
}
```

运行结果：

```
2017-2-1
2017-1-28
```

可以看到上个月的最后一天是28号，所以Calendar.MONTH为1的时候是2月

既然日期设为0表示上个月的最后一天，那是不是可以设为负数呢？

```
import java.util.Calendar;

public class Test {
    public static void main(String[] args) {
        Calendar c1 = Calendar.getInstance();
        c1.set(2017, 2, 1);
        System.out.println(c1.get(Calendar.YEAR)
            + "-" + c1.get(Calendar.MONTH)
            + "-" + c1.get(Calendar.DATE));
        c1.set(2017, 2, -10);
        System.out.println(c1.get(Calendar.YEAR)
            + "-" + c1.get(Calendar.MONTH)
            + "-" + c1.get(Calendar.DATE));
    }
}
```

运行结果：

```
2017-2-1
2017-1-18
```

果然可以，所以日期才可以自由加减。

月份也可以是负数，规则与日期一样，就不上代码了。

实测将年份设为非正数时，会自动变为绝对值+1，不知其意义。

lhm\*\*\*@126.com

lhm\*\*\*@126.com

3年前 (2017-11-22)

## 2. 冲冲冲

173\*\*\*2867@qq.com

4

```
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小时制) : %tr%n", date);
        //t的使用
        System.out.printf("HH:MM:SS格式 (24小时制) : %tT%n", date);
        //R的使用
        System.out.printf("HH:MM格式 (24小时制) : %tR", date);
    }
}
```

输出结果为：

```
全部日期和时间信息：周六 8月 22 11:53:36 CST 2020
年-月-日格式：2020-08-22
月/日/年格式：08/22/20
HH:MM:SS PM格式 (12小时制) : 11:53:36 上午
HH:MM:SS格式 (24小时制) : 11:53:36
HH:MM格式 (24小时制) : 11:53
```

冲冲冲

冲冲冲

173\*\*\*2867@qq.com

5个月前 (08-22)

# Java 正则表达式 | 菜鸟教程

 [runoob.com/java/java-regular-expressions.html](http://runoob.com/java/java-regular-expressions.html)

[Java 日期时间](#)

[Java 方法](#)

## Java 正则表达式

正则表达式定义了字符串的模式。

正则表达式可以用来搜索、编辑或处理文本。

正则表达式并不限于某一种语言，但是在每种语言中有细微的差别。

## 正则表达式实例

一个字符串其实就是一个简单的正则表达式，例如 **Hello World** 正则表达式匹配 "Hello World" 字符串。

. (点号) 也是一个正则表达式，它匹配任何一个字符如："a" 或 "1"。

下表列出了一些正则表达式的实例及描述：

正则表达式	描述
this is text	匹配字符串 "this is text"
this\+\s+is\s+text	注意字符串中的 \s+。 匹配单词 "this" 后面的 \s+ 可以匹配多个空格，之后匹配 is 字符串，再之后 \s+ 匹配多个空格然后再跟上 text 字符串。 可以匹配这个实例：this is text
^\d+(\.\d+)?	^ 定义了以什么开始 \d+ 匹配一个或多个数字 ? 设置括号内的选项是可选的 \. 匹配 ":" 可以匹配的实例："5", "1.5" 和 "2.21"。

Java 正则表达式和 Perl 的是最为相似的。

java.util.regex 包主要包括以下三个类：

- **Pattern 类：**

pattern 对象是一个正则表达式的编译表示。Pattern 类没有公共构造方法。要创建一个 Pattern 对象，你必须首先调用其公共静态编译方法，它返回一个 Pattern 对象。该方法接受一个正则表达式作为它的第一个参数。

- **Matcher 类：**

Matcher 对象是对输入字符串进行解释和匹配操作的引擎。与 Pattern 类一样，Matcher 也没有公共构造方法。你需要调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

- **PatternSyntaxException :**

PatternSyntaxException 是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

以下实例中使用了正则表达式 `.*runoob.*` 用于查找字符串中是否包了 **runoob** 子串：

## 实例

---

```
import java.util.regex.*; class RegexExample1{ public static void main(String[] args){ String content = "I am noob " + "from runoob.com."; String pattern = ".*runoob.*"; boolean isMatch = Pattern.matches(pattern, content); System.out.println("字符串中是否包含了 'runoob' 子字符串? " + isMatch); } }
```

实例输出结果为：

字符串中是否包含了 'runoob' 子字符串? true

---

## 捕获组

---

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

例如，正则表达式 (dog) 创建了单一分组，组里包含 "d"，"o"，和 "g"。

捕获组是通过从左至右计算其开括号来编号。例如，在表达式 ((A) (B (C)) )，有四个这样的组：

- ((A)(B(C)))
- (A)
- (B(C))
- (C)

可以通过调用 matcher 对象的 groupCount 方法来查看表达式有多少个分组。groupCount 方法返回一个 int 值，表示 matcher 对象当前有多少个捕获组。

还有一个特殊的组 (group(0))，它总是代表整个表达式。该组不包括在 groupCount 的返回值中。

## 实例

---

下面的例子说明如何从一个给定的字符串中找到数字串：

### RegexMatches.java 文件代码：

---

```
import java.util.regex.Matcher; import java.util.regex.Pattern; public class RegexMatches { public static void main( String[] args ){ // 按指定模式在字符串查找 String line = "This order was placed for QT3000! OK?"; String pattern = "(\\\\D*)(\\\\d+)(.*)"; // 创建 Pattern 对象 Pattern r = Pattern.compile(pattern); // 现在创建 matcher 对象 Matcher m = r.matcher(line); if (m.find( )) { System.out.println("Found value: " + m.group(0)); System.out.println("Found value: " + m.group(1)); System.out.println("Found value: " + m.group(2)); System.out.println("Found value: " + m.group(3)); } else { System.out.println("NO MATCH"); } }
```

以上实例编译运行结果如下：

```
Found value: This order was placed for QT3000! OK?  
Found value: This order was placed for QT  
Found value: 3000  
Found value: ! OK?
```

---

## 正则表达式语法

---

在其他语言中，\\ 表示：我想要在正则表达式中插入一个普通的（字面上的）反斜杠，请不要给它任何特殊的意义。

在 Java 中，\\ 表示：我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。

所以，在其他的语言中（如Perl），一个反斜杠\ 就足以具有转义的作用，而在 Java 中正则表达式中则需要有两个反斜杠才能被解析为其他语言中的转义作用。也可以简单的理解在 Java 的正则表达式中，两个\\ 代表其他语言中的一个\，这也就是为什么表示一位数字的正则表达式是\\d，而表示一个普通的反斜杠是\\\\。

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，“n”匹配字符“n”。\\n匹配换行符。序列“\\\\”匹配“\\”，“\\(”匹配“(”。
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与“\\n”或“\\r”之后的位置匹配。
\$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，\$ 还会与“\\n”或“\\r”之前的位置匹配。

---

*	零次或多次匹配前面的字符或子表达式。例如， <code>zo*</code> 匹配"z"和"zoo"。* 等效于 <code>{0,}</code> 。
+	一次或多次匹配前面的字符或子表达式。例如， <code>"zo+"</code> 与"zo"和"zoo"匹配，但与"z"不匹配。+ 等效于 <code>{1,}</code> 。
?	零次或一次匹配前面的字符或子表达式。例如， <code>"do(es)?"</code> 匹配"do"或"does"中的"do"。? 等效于 <code>{0,1}</code> 。
<code>{n}</code>	<code>n</code> 是非负整数。正好匹配 <code>n</code> 次。例如， <code>"o{2}"</code> 与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
<code>{n,}</code>	<code>n</code> 是非负整数。至少匹配 <code>n</code> 次。例如， <code>"o{2,}"</code> 不匹配"Bob"中的"o"，而匹配"fooooood"中的所有 o。 <code>"o{1,}"</code> 等效于 <code>"o+"</code> 。 <code>"o{0,}"</code> 等效于 <code>"o*"</code> 。
<code>{n,m}</code>	<code>m</code> 和 <code>n</code> 是非负整数，其中 <code>n &lt;= m</code> 。匹配至少 <code>n</code> 次，至多 <code>m</code> 次。例如， <code>"o{1,3}"</code> 匹配"fooooood"中的头三个 o。 <code>'o{0,1}'</code> 等效于 <code>'o?'</code> 。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（*、+、?、 <code>{n}</code> 、 <code>{n,}</code> 、 <code>{n,m}</code> ）之后时，匹配模式是"非贪婪的"。"非贪婪的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪婪的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中， <code>"o+?"</code> 只匹配单个"o"，而 <code>"o+"</code> 匹配所有"o"。
.	匹配除" <code>\r\n</code> "之外的任何单个字符。若要匹配包括" <code>\r\n</code> "在内的任意字符，请使用诸如 <code>"[\s\S]"</code> 之类的模式。
<code>(pattern)</code>	匹配 <code>pattern</code> 并捕获该匹配的子表达式。可以使用 <code>\$0...\$9</code> 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 <code>()</code> ，请使用 <code>"\("或者"\)"</code> 。
<code>(?:pattern)</code>	匹配 <code>pattern</code> 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 <code>( )</code> 组合模式部件的情况很有用。例如， <code>'industr(?:y ies)'</code> 是比 <code>'industry industries'</code> 更经济的表达式。
<code>(?=pattern)</code>	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 <code>pattern</code> 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如， <code>'Windows (?=95 98 NT 2000)'</code> 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。

---

---

(?!pattern)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 <i>pattern</i> 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95 98 NT 2000)' 匹配"Windows 3.1"中的"Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x y	匹配 x 或 y。例如，'z food' 匹配"z"或"food"。'(z f)ood' 匹配"zood"或"food"。
[xyz]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^xyz]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p", "l", "i", "n"。
[a-z]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^a-z]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。
\b	匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\cx	匹配 x 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。x 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。
\D	非数字字符匹配。等效于 [^0-9]。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。

---

---

\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [ \f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与 "[A-Za-z0-9_]" 等效。
\W	与任何非单词字符匹配。与 "[^A-Za-z0-9_]" 等效。
\xn	匹配 <i>n</i> , 此处的 <i>n</i> 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如, "\x41" 匹配 "A"。"\x041" 与 "\x04" & "1" 等效。允许在正则表达式中使用 ASCII 代码。
\num	匹配 <i>num</i> , 此处的 <i>num</i> 是一个正整数。到捕获匹配的反向引用。例如, "(.)\1" 匹配两个连续的相同字符。
\n	标识一个八进制转义码或反向引用。如果 \n 前面至少有 <i>n</i> 个捕获子表达式, 那么 <i>n</i> 是反向引用。否则, 如果 <i>n</i> 是八进制数 (0-7), 那么 <i>n</i> 是八进制转义码。
\nm	标识一个八进制转义码或反向引用。如果 \nm 前面至少有 <i>nm</i> 个捕获子表达式, 那么 <i>nm</i> 是反向引用。如果 \nm 前面至少有 <i>n</i> 个捕获, 则 <i>n</i> 是反向引用, 后面跟有字符 <i>m</i> 。如果两种前面的情况都不存在, 则 \nm 匹配八进制值 <i>nm</i> , 其中 <i>n</i> 和 <i>m</i> 是八进制数字 (0-7)。
\nml	当 <i>n</i> 是八进制数 (0-3), <i>m</i> 和 <i>l</i> 是八进制数 (0-7) 时, 匹配八进制转义码 <i>nml</i> 。
\un	匹配 <i>n</i> , 其中 <i>n</i> 是以四位十六进制数表示的 Unicode 字符。例如, \u00A9 匹配版权符号 (©)。

根据 Java Language Specification 的要求，Java 源代码的字符串中的反斜线被解释为 Unicode 转义或其他字符转义。因此必须在字符串字面值中使用两个反斜线，表示正则表达式受到保护，不被 Java 字节码编译器解释。例如，当解释为正则表达式时，字符串字面值 "\b" 与单个退格字符匹配，而 "\\b" 与单词边界匹配。字符串字面值 "\\\n(hello)\" 是非法的，将导致编译时错误；要与字符串 (hello) 匹配，必须使用字符串字面值 "\\n(hello\\\")"。

## Matcher 类的方法

### 索引方法

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

#### 序号 方法及说明

1 **public int start()**

返回以前匹配的初始索引。

2 **public int start(int group)**

返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引

3 **public int end()**

返回最后匹配字符之后的偏移量。

4 **public int end(int group)**

返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

### 查找方法

查找方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

#### 序

#### 号 方法及说明

1 **public boolean lookingAt()**

尝试将从区域开头开始的输入序列与该模式匹配。

2 **public boolean find()**

尝试查找与该模式匹配的输入序列的下一个子序列。

3 **public boolean find(int start)**

重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。

4 **public boolean matches()**

尝试将整个区域与模式匹配。

## 替换方法

---

替换方法是替换输入字符串里文本的方法：

序

号 方法及说明

---

1 **public Matcher appendReplacement(StringBuffer sb, String replacement)**

实现非终端添加和替换步骤。

---

2 **public StringBuffer appendTail(StringBuffer sb)**

实现终端添加和替换步骤。

---

3 **public String replaceAll(String replacement)**

替换模式与给定替换字符串相匹配的输入序列的每个子序列。

---

4 **public String replaceFirst(String replacement)**

替换模式与给定替换字符串匹配的输入序列的第一个子序列。

---

5 **public static String quoteReplacement(String s)**

返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给 Matcher类的appendReplacement 方法一个字面字符串一样工作。

## start 和 end 方法

---

下面是一个对单词 "cat" 出现在输入字符串中出现次数进行计数的例子：

### RegexMatches.java 文件代码：

---

```
import java.util.regex.Matcher; import java.util.regex.Pattern; public class RegexMatches { private static final String REGEX = "\\\\bcat\\\\b"; private static final String INPUT = "cat cat cat cattie cat"; public static void main( String[] args ){ Pattern p = Pattern.compile(REGEX); Matcher m = p.matcher(INPUT); // 获取 matcher 对象 int count = 0; while(m.find()) { count++; System.out.println("Match number "+count); System.out.println("start(): "+m.start()); System.out.println("end(): "+m.end()); } }
```

以上实例编译运行结果如下：

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

可以看到这个例子是使用单词边界，以确保字母 "c" "a" "t" 并非仅是一个较长的词的子串。它也提供了一些关于输入字符串中匹配发生位置的有用信息。

Start 方法返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引，end 方法最后一个匹配字符的索引加 1。

## matches 和 lookingAt 方法

matches 和 lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是 matches 要求整个序列都匹配，而lookingAt 不要求。

lookingAt 方法虽然不需要整句都匹配，但是需要从第一个字符开始匹配。

这两个方法经常在输入字符串的开始使用。

我们通过下面这个例子，来解释这个功能：

### RegexMatches.java 文件代码：

```
import java.util.regex.Matcher; import java.util.regex.Pattern; public class RegexMatches { private static final String REGEX = "foo"; private static final String INPUT = "foooooooooooooooo"; private static final String INPUT2 = "ooooofoooooooooooo"; private static Pattern pattern; private static Matcher matcher; private static Matcher matcher2; public static void main( String[] args ){ pattern = Pattern.compile(REGEX); matcher = pattern.matcher(INPUT); matcher2 = pattern.matcher(INPUT2); System.out.println("Current REGEX is: "+REGEX); System.out.println("Current INPUT is: "+INPUT); System.out.println("Current INPUT2 is: "+INPUT2); System.out.println("lookingAt(): "+matcher.lookingAt()); System.out.println("matches(): "+matcher.matches()); System.out.println("lookingAt(): "+matcher2.lookingAt()); } }
```

以上实例编译运行结果如下：

```
Current REGEX is: foo
Current INPUT is: foooooooooooooooo
Current INPUT2 is: ooooofooooooooooooo
lookingAt(): true
matches(): false
lookingAt(): false
```

## replaceFirst 和 replaceAll 方法

replaceFirst 和 replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

下面的例子来解释这个功能：

### RegexMatches.java 文件代码：

```
import java.util.regex.Matcher; import java.util.regex.Pattern; public class RegexMatches
{ private static String REGEX = "dog"; private static String INPUT = "The dog says meow.
" + "All dogs say meow."; private static String REPLACE = "cat"; public static void
main(String[] args) { Pattern p = Pattern.compile(REGEX); // get a matcher object
Matcher m = p.matcher(INPUT); INPUT = m.replaceAll(REPLACE);
System.out.println(INPUT); } }
```

以上实例编译运行结果如下：

```
The cat says meow. All cats say meow.
```

## appendReplacement 和 appendTail 方法

---

Matcher 类也提供了appendReplacement 和 appendTail 方法用于文本替换：

看下面的例子来解释这个功能：

### RegexMatches.java 文件代码：

---

```
import java.util.regex.Matcher; import java.util.regex.Pattern; public class RegexMatches
{ private static String REGEX = "a*b"; private static String INPUT =
"aabfooaabfooabfoobkkk"; private static String REPLACE = "-"; public static void
main(String[] args) { Pattern p = Pattern.compile(REGEX); // 获取 matcher 对象 Matcher
m = p.matcher(INPUT); StringBuffer sb = new StringBuffer(); while(m.find()){
m.appendReplacement(sb,REPLACE); } m.appendTail(sb);
System.out.println(sb.toString()); } }
```

以上实例编译运行结果如下：

```
-foo-foo-foo-kkk
```

## PatternSyntaxException 类的方法

---

PatternSyntaxException 是一个非强制异常类，它指示一个正则表达式模式中的语法错误。

PatternSyntaxException 类提供了下面的方法来帮助我们查看发生了什么错误。

序

号 方法及说明

---

1 **public String getDescription()**

获取错误的描述。

---

2 **public int getIndex()**

获取错误的索引。

---

3 **public String getPattern()**

获取错误的正则表达式模式。

---

#### 4 **public String getMessage()**

返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。

[Java 日期时间](#)

[Java 方法](#)

---

## 4 篇笔记 写笔记

# Java 方法 | 菜鸟教程

 [runoob.com/java/java-methods.html](https://runoob.com/java/java-methods.html)

## Java 方法

在前面几个章节中我们经常使用到 **System.out.println()**，那么它是什么呢？

- `println()` 是一个方法。
- `System` 是系统类。
- `out` 是标准输出对象。

这句话的用法是调用系统类 `System` 中的标准输出对象 `out` 中的方法 `println()`。

## 那么什么是方法呢？

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

## 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。

## 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`。

## 方法的定义

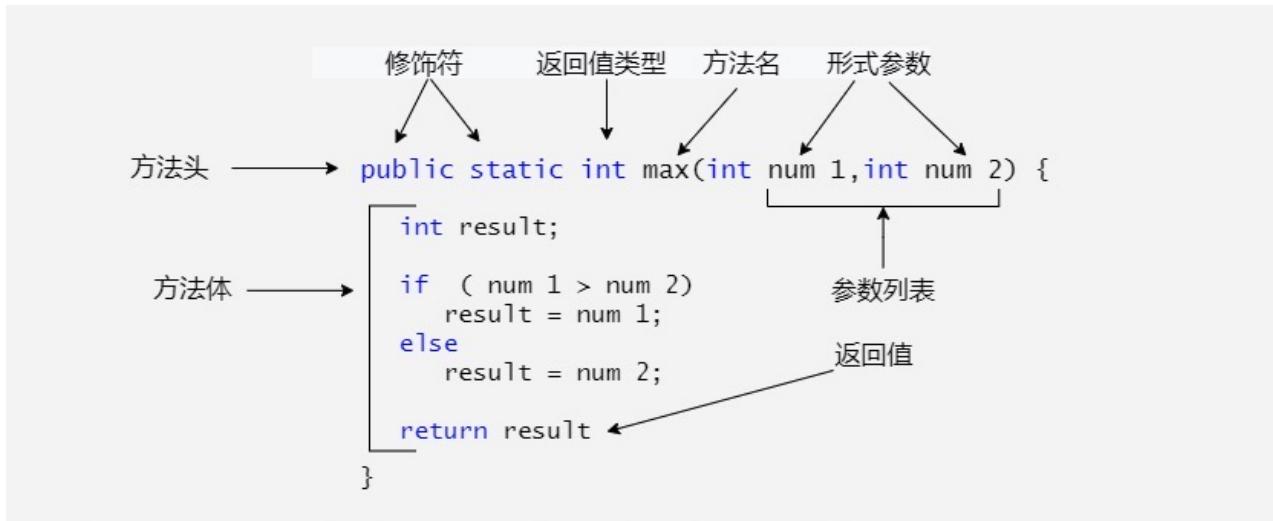
一般情况下，定义一个方法包含以下语法：

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; }

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。

## 实例

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最大值 */ public static int max(int num1, int num2) { int result; if (num1 > num2) result = num1; else result = num2; return result; }
```

更简略的写法（三元运算符）：

```
public static int max(int num1, int num2) { return num1 > num2 ? num1 : num2; }
```

---

## 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時候，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 方法和其他方法没什么区别。

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，此外带一个 String[] 类型参数。String[] 表明参数是字符串数组。

---

## **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。

## 示例

---

### **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：

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); } }
```

以上实例编译运行结果如下：

```
交换前 num1 的值为：1， num2 的值为：2  
    进入 swap 方法  
        交换前 n1 的值为：1， n2 的值：2  
        交换后 n1 的值为：2， n2 的值：1  
交换后 num1 的值为：1， num2 的值为：2
```

传递两个参数调用swap方法。有趣的是，方法被调用后，实参的值并没有改变。

---

## 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。

---

## 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。

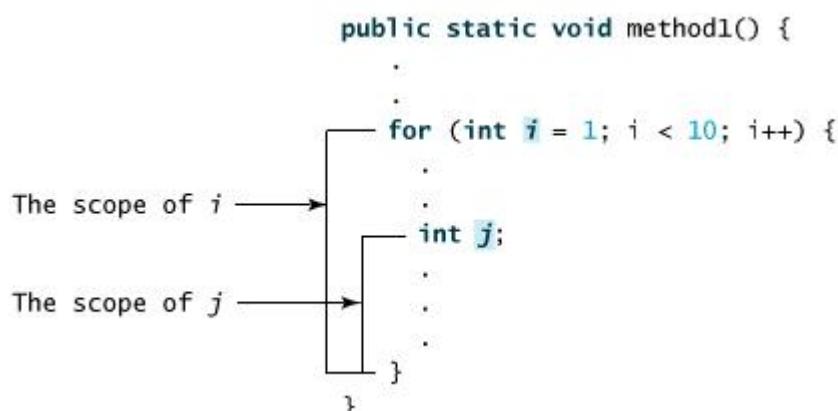
局部变量的作用范围从声明开始，直到包含它的块结束。

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。

## 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。

## 实例

---

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
```

---

## 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。

不管你是否自定义构造方法，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，默认构造方法的访问修饰符和类的访问修饰符相同(类为 public，构造函数也为 public；类改为 protected，构造函数也改为 protected)。

一旦你定义了自己的构造方法，默认构造方法就会失效。

## 实例

---

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); } }
```

大多時候需要一个有参数的构造方法。

## 实例

---

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
```

## 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。

## 实例

---

## 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
```

## 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。

## 实例

---

### 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：

```
$ javac FinalizationDemo.java  
$ java FinalizationDemo  
Cake Object 1is created  
Cake Object 2is created  
Cake Object 3is created  
Cake Object 3is disposed  
Cake Object 2is disposed
```

# Java 流(Stream)、文件(File)和IO

---



[runoob.com/java/java-files-io.html](http://runoob.com/java/java-files-io.html)

## Java 方法

### Java Scanner 类

Java.io 包几乎包含了所有操作输入、输出需要的类。所有这些流类代表了输入源和输出目标。

Java.io 包中的流支持很多种格式，比如：基本类型、对象、本地化字符集等等。

一个流可以理解为一个数据的序列。输入流表示从一个源读取数据，输出流表示向一个目标写数据。

Java 为 I/O 提供了强大的而灵活的支持，使其更广泛地应用到文件传输和网络编程中。

但本节讲述最基本的和流与 I/O 相关的功能。我们将通过一个个例子来学习这些功能。

---

## 读取控制台输入

Java 的控制台输入由 System.in 完成。

为了获得一个绑定到控制台的字符流，你可以把 System.in 包装在一个 BufferedReader 对象中来创建一个字符流。

下面是创建 BufferedReader 的基本语法：

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

BufferedReader 对象创建后，我们便可以使用 read() 方法从控制台读取一个字符，或者用 readLine() 方法读取一个字符串。

---

## 从控制台读取多字符输入

从 BufferedReader 对象读取一个字符要使用 read() 方法，它的语法如下：

```
int read() throws IOException
```

每次调用 read() 方法，它从输入流读取一个字符并把该字符作为整数值返回。当流结束的时候返回 -1。该方法抛出 IOException。

下面的程序示范了用 read() 方法从控制台不断读取字符直到用户输入 "q"。

## BRRead.java 文件代码：

```
//使用 BufferedReader 在控制台读取字符 import java.io.*; public class BRRead { public static void main(String args[]) throws IOException { char c; // 使用 System.in 创建 BufferedReader BufferedReader br = new BufferedReader(new
```

```
InputStreamReader(System.in)); System.out.println("输入字符, 按下 'q' 键退出。"); // 读取字符 do { c = (char) br.read(); System.out.println(c); } while (c != 'q'); }
```

以上实例编译运行结果如下:

输入字符, 按下 'q' 键退出。

runoob

r

u

n

o

o

b

---

q

q

## 从控制台读取字符串

---

从标准输入读取一个字符串需要使用 BufferedReader 的 readLine() 方法。

它的一般格式是：

```
String readLine( ) throws IOException
```

下面的程序读取和显示字符行直到你输入了单词"end"。

### BRReadLines.java 文件代码：

---

```
//使用 BufferedReader 在控制台读取字符 import java.io.*; public class BRReadLines {  
public static void main(String args[]) throws IOException { // 使用 System.in 建  
BufferedReader BufferedReader br = new BufferedReader(new  
InputStreamReader(System.in)); String str; System.out.println("Enter lines of text.");  
System.out.println("Enter 'end' to quit."); do { str = br.readLine();  
System.out.println(str); } while (!str.equals("end")); } }
```

以上实例编译运行结果如下:

```
Enter lines of text.  
Enter 'end' to quit.  
This is line one  
This is line one  
This is line two  
This is line two  
end  
end
```

| JDK 5 后的版本我们也可以使用 [Java Scanner](#) 类来获取控制台的输入。

## 控制台输出

---

在此前已经介绍过，控制台的输出由 print( ) 和 println() 完成。这些方法都由类 PrintStream 定义，System.out 是该类对象的一个引用。

PrintStream 继承了 OutputStream 类，并且实现了方法 write()。这样，write() 也可以用来往控制台写操作。

PrintStream 定义 write() 的最简单格式如下所示：

```
void write(int byteval)
```

该方法将 byteval 的低八位字节写到流中。

## 实例

---

下面的例子用 write() 把字符 "A" 和紧跟着的换行符输出到屏幕：

### WriteDemo.java 文件代码：

---

```
import java.io.*; //演示 System.out.write(). public class WriteDemo { public static void main(String args[]) { int b; b = 'A'; System.out.write(b); System.out.write('\n'); } }
```

运行以上实例在输出窗口输出 "A" 字符

```
A
```

注意：write() 方法不经常使用，因为 print() 和 println() 方法用起来更为方便。

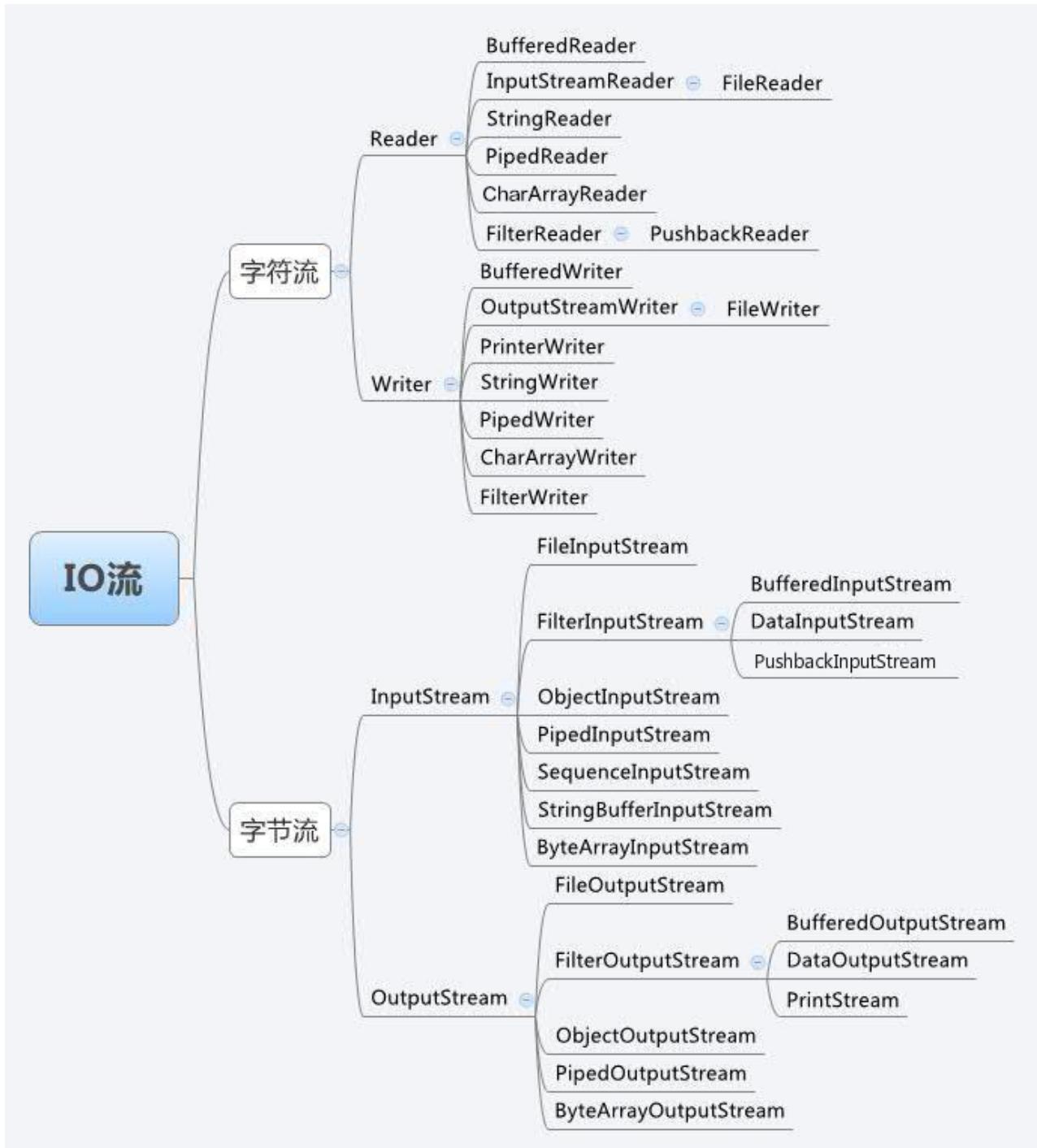
---

## 读写文件

---

如前所述，一个流被定义为一个数据序列。输入流用于从源读取数据，输出流用于向目标写数据。

下图是一个描述输入流和输出流的类层次图。



下面将要讨论的两个重要的流是 FileInputStream 和 FileOutputStream：

## FileInputStream

该流用于从文件读取数据，它的对象可以用关键字 new 来创建。

有多种构造方法可用来创建对象。

可以使用字符串类型的文件名来创建一个输入流对象来读取文件：

```
InputStream f = new FileInputStream("C:/java/hello");
```

也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用 File() 方法来创建一个文件对象：

```
File f = new File("C:/java/hello"); InputStream out = new FileInputStream(f);
```

创建了InputStream对象，就可以使用下面的方法来读取流或者进行其他的流操作。

## 序号 方法及描述

---

### 1 **public void close() throws IOException{}**

关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。

---

### 2 **protected void finalize()throws IOException {}**

这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出 IOException 异常。

---

### 3 **public int read(int r)throws IOException{}**

这个方法从 InputStream 对象读取指定字节的数据。返回为整数值。返回下一字节数据，如果已经到结尾则返回-1。

---

### 4 **public int read(byte[] r) throws IOException{}**

这个方法从输入流读取r.length长度的字节。返回读取的字节数。如果是文件结尾则返回-1。

---

### 5 **public int available() throws IOException{}**

返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取的字节数。返回一个整数值。

除了 InputStream 外，还有一些其他的输入流，更多的细节参考下面链接：

- [ByteArrayInputStream](#)
  - [DataInputStream](#)
- 

## FileOutputStream

该类用来创建一个文件并向文件中写数据。

如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。

有两个构造方法可以用来创建 FileOutputStream 对象。

使用字符串类型的文件名来创建一个输出流对象：

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用File()方法来创建一个文件对象：

```
File f = new File("C:/java/hello"); OutputStream f = new FileOutputStream(f);
```

创建OutputStream 对象完成后，就可以使用下面的方法来写入流或者进行其他的流操作。

## 序号 方法及描述

---

### 1 public void close() throws IOException{}

关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException 异常。

---

### 2 protected void finalize()throws IOException {}

这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException 异常。

---

### 3 public void write(int w)throws IOException{}

这个方法把指定的字节写到输出流中。

---

### 4 public void write(byte[] w)

把指定数组中w.length 长度的字节写到OutputStream 中。

除了OutputStream 外，还有一些其他的输出流，更多的细节参考下面链接：

- [ByteArrayOutputStream](#)
- [DataOutputStream](#)

## 实例

下面是一个演示 InputStream 和 OutputStream 用法的例子：

### fileStreamTest.java 文件代码：

```
import java.io.*; public class fileStreamTest { public static void main(String args[]) { try { byte bWrite[] = { 11, 21, 3, 40, 5 }; OutputStream os = new FileOutputStream("test.txt"); for (int x = 0; x < bWrite.length; x++) { os.write(bWrite[x]); // writes the bytes } os.close(); InputStream is = new FileInputStream("test.txt"); int size = is.available(); for (int i = 0; i < size; i++) { System.out.print((char) is.read() + " "); } is.close(); } catch (IOException e) { System.out.print("Exception"); } } }
```

上面的程序首先创建文件test.txt，并把给定的数字以二进制形式写进该文件，同时输出到控制台上。

以上代码由于是二进制写入，可能存在乱码，你可以使用以下代码实例来解决乱码问题：

### fileStreamTest2.java 文件代码：

```
//文件名 :fileStreamTest2.java import java.io.*; public class fileStreamTest2 { public static void main(String[] args) throws IOException { File f = new File("a.txt"); FileOutputStream fop = new FileOutputStream(f); // 构建FileOutputStream对象,文件不存在会自动新建 OutputStreamWriter writer = new OutputStreamWriter(fop, "UTF-8"); //
```

```
构建OutputStreamWriter对象,参数可以指定编码,默认为操作系统默认编码,Windows上是gbk
writer.append("中文输入"); // 写入到缓冲区 writer.append("\r\n"); // 换行
writer.append("English"); // 刷新缓存区,写入到文件,如果下面已经没有写入的内容了,直接close也会写入
writer.close(); // 关闭写入流,同时会把缓冲区内容写入文件,所以上面的注释掉
fop.close(); // 关闭输出流,释放系统资源 FileInputStream fip = new FileInputStream(f); //
构建InputStream对象 InputStreamReader reader = new InputStreamReader(fip,
"UTF-8"); // 构建InputStreamReader对象,编码与写入相同 StringBuffer sb = new
StringBuffer(); while (reader.ready()) { sb.append((char) reader.read()); // 转成char加到
StringBuffer对象中 } System.out.println(sb.toString()); reader.close(); // 关闭读取流
fip.close(); // 关闭输入流,释放系统资源 } }
```

---

## 文件和I/O

---

还有一些关于文件和I/O的类,我们也需要知道:

---

## Java中的目录

---

### 创建目录:

---

File类中有两个方法可以用来创建文件夹:

- **mkdir()**方法创建一个文件夹,成功则返回true,失败则返回false。失败表明File对象指定的路径已经存在,或者由于整个路径还不存在,该文件夹不能被创建。
- **mkdirs()**方法创建一个文件夹和它的所有父文件夹。

下面的例子创建"/tmp/user/java/bin"文件夹:

### CreateDir.java 文件代码:

---

```
import java.io.File; public class CreateDir { public static void main(String args[]) { String
dirname = "/tmp/user/java/bin"; File d = new File(dirname); // 现在创建目录 d.mkdirs();
} }
```

编译并执行上面代码来创建目录"/tmp/user/java/bin"。

**注意:** Java 在 UNIX 和 Windows 自动按约定分辨文件路径分隔符。如果你在 Windows 版本的 Java 中使用分隔符(/),路径依然能够被正确解析。

---

## 读取目录

---

一个目录其实就是一个File对象,它包含其他文件和文件夹。

如果创建一个File对象并且它是一个目录,那么调用isDirectory()方法会返回true。

可以通过调用该对象上的list()方法,来提取它包含的文件和文件夹的列表。

下面展示的例子说明如何使用 list() 方法来检查一个文件夹中包含的内容：

## DirList.java 文件代码：

---

```
import java.io.File; public class DirList { public static void main(String args[]) { String  
dirname = "/tmp"; File f1 = new File(dirname); if (f1.isDirectory()) {  
System.out.println("目录 " + dirname); String s[] = f1.list(); for (int i = 0; i < s.length;  
i++) { File f = new File(dirname + "/" + s[i]); if (f.isDirectory()) { System.out.println(s[i]  
+ " 是一个目录"); } else { System.out.println(s[i] + " 是一个文件"); } } } else {  
System.out.println(dirname + " 不是一个目录"); } } }
```

以上实例编译运行结果如下：

```
目录 /tmp  
bin 是一个目录  
lib 是一个目录  
demo 是一个目录  
test.txt 是一个文件  
README 是一个文件  
index.html 是一个文件  
include 是一个目录
```

---

## 删除目录或文件

---

删除文件可以使用 **java.io.File.delete()** 方法。

以下代码会删除目录 **/tmp/java/**，需要注意的是当删除某一目录时，必须保证该目录下没有其他文件才能正确删除，否则将删除失败。

测试目录结构：

```
/tmp/java/  
|--- 1.log  
|--- test
```

## DeleteFileDemo.java 文件代码：

---

```
import java.io.File; public class DeleteFileDemo { public static void main(String args[]) {  
// 这里修改为自己的测试目录 File folder = new File("/tmp/java/"); deleteFolder(folder); }  
// 删除文件及目录 public static void deleteFolder(File folder) { File[] files =  
folder.listFiles(); if (files != null) { for (File f : files) { if (f.isDirectory()) { deleteFolder(f); }  
else { f.delete(); } } } folder.delete(); } }
```

[Java 方法](#)

[Java Scanner 基本用法](#)

## 5 篇笔记 写笔记

---



# Java Scanner 类 | 菜鸟教程

 [runoob.com/java/java-scanner-class.html](http://runoob.com/java/java-scanner-class.html)

## Java Scanner 类

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：

**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。

**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。

## next() 与 nextLine() 区别

---

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。

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()) { // 判断输入的是不是整数 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()) { // 判断输入的是不是小数 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
```

更多内容可以参考 API 文档：<http://www.runoob.com/manual/jdk1.6/>。

## 5 篇笔记 写笔记

---

# Java 异常处理 | 菜鸟教程

---

 [runoob.com/java/java-exceptions.html](http://runoob.com/java/java-exceptions.html)

[Java Scanner 类](#)

[Java 继承](#)

## Java 异常处理

---

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。

比如说，你的代码少了一个分号，那么运行出来结果是提示是错误 `java.lang.Error`；如果你用 `System.out.println(11/0)`，那么你是因为你用0做了除数，会抛出 `java.lang.ArithmaticException` 的异常。

异常发生的原因有很多，通常包含以下几大类：

- 用户输入了非法数据。
- 要打开的文件不存在。
- 网络通信时连接中断，或者JVM内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。-

要理解Java异常处理是如何工作的，你需要掌握以下三种类型的异常：

- **检查性异常**：最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
  - **运行时异常**：运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
  - **错误**：错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译时也检查不到的。
- 

## Exception 类的层次

---

所有的异常类是从 `java.lang.Exception` 类继承的子类。

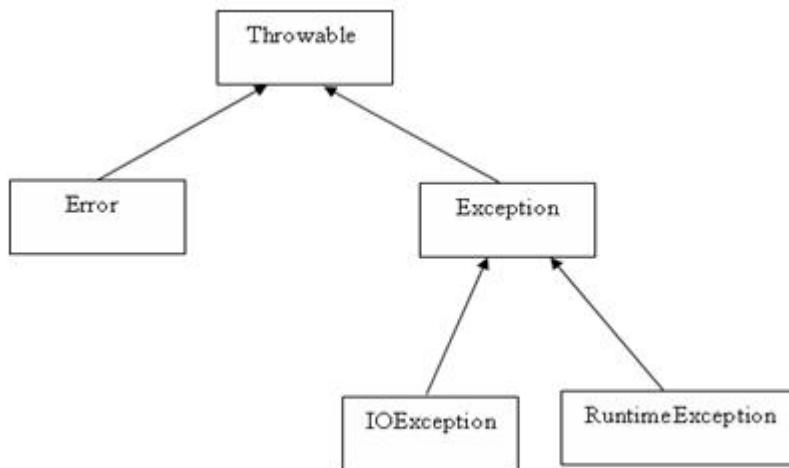
`Exception` 类是 `Throwable` 类的子类。除了`Exception`类外，`Throwable`还有一个子类`Error`。

Java 程序通常不捕获错误。错误一般发生在严重故障时，它们在Java程序处理的范畴之外。

`Error` 用来指示运行时环境发生的错误。

例如，JVM 内存溢出。一般地，程序不会从错误中恢复。

异常类有两个主要的子类：IOException 类和 RuntimeException 类。



在 Java 内置类中(接下来会说明)，有大部分常用检查性和非检查性异常。

## Java 内置异常类

Java 语言定义了一些异常类在 `java.lang` 标准包中。

标准运行时异常类的子类是最常见的异常类。由于 `java.lang` 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。

Java 根据各个库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

异常	描述
<code>ArithmaticException</code>	当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。
<code>ArrayIndexOutOfBoundsException</code>	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
<code>ArrayStoreException</code>	试图将错误类型的对象存储到一个对象数组时抛出的异常。
<code>ClassCastException</code>	当试图将对象强制转换为不是实例的子类时，抛出该异常。
<code>IllegalArgumentException</code>	抛出的异常表明向方法传递了一个不合法或不正确的参数。
<code>IllegalMonitorStateException</code>	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。

IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。
IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 <code>null</code> 时，抛出该异常。
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 <code>String</code> 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

下面的表中列出了 Java 定义在 `java.lang` 包中的检查性异常类。

异常	描述
ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedException	当调用 <code>Object</code> 类中的 <code>clone</code> 方法克隆对象，但该对象的类无法实现 <code>Cloneable</code> 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 <code>Class</code> 类中的 <code>newInstance</code> 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

## 异常方法

下面的列表是 Throwable 类的主要方法：

序号	方法及说明
1	<b>public String getMessage()</b> 返回关于发生的异常的详细信息。这个消息在 Throwable 类的构造函数中初始化了。
2	<b>public Throwable getCause()</b> 返回一个 Throwable 对象代表异常原因。
3	<b>public String toString()</b> 使用 getMessage() 的结果返回类的串级名字。
4	<b>public void printStackTrace()</b> 打印 toString() 结果和栈层次到 System.err，即错误输出流。
5	<b>public StackTraceElement [] getStackTrace()</b> 返回一个包含堆栈层次的数组。下标为 0 的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。
6	<b>public Throwable fillInStackTrace()</b> 用当前的调用栈层次填充 Throwable 对象栈层次，添加到 栈层次 任何先前信息中。

## 捕获异常

使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。

try/catch 代码块中的代码称为保护代码，使用 try/catch 的语法如下：

```
try
{
    // 程序代码
} catch (ExceptionName e1)
{
    // Catch 块
}
```

Catch 语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，try 后面的 catch 块就会被检查。

如果发生的异常包含在 catch 块中，异常会被传递到该 catch 块，这和传递一个参数到方法是一样的。

## 实例

下面的例子中声明有两个元素的一个数组，当代码试图访问数组的第三个元素的时候就会抛出一个异常。

### ExcepTest.java 文件代码：

```
// 文件名 : ExcepTest.java import java.io.*; public class ExcepTest{ public static void main(String args[]){ try{ int a[] = new int[2]; System.out.println("Access element three :" + a[3]); }catch(ArrayIndexOutOfBoundsException e){ System.out.println("Exception thrown :" + e); } System.out.println("Out of the block"); } }
```

以上代码编译运行输出结果如下：

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3  
out of the block
```

---

## 多重捕获块

一个 try 代码块后面跟随多个 catch 代码块的情况就叫多重捕获。

多重捕获块的语法如下所示：

```
try{ // 程序代码 }catch(异常类型1 异常的变量名1){ // 程序代码 }catch(异常类型2 异常的变量名2){ // 程序代码 }catch(异常类型3 异常的变量名3){ // 程序代码 }
```

上面的代码段包含了 3 个 catch 块。

可以在 try 语句后面添加任意数量的 catch 块。

如果保护代码中发生异常，异常被抛给第一个 catch 块。

如果抛出异常的数据类型与 ExceptionType1 匹配，它在这里就会被捕获。

如果不匹配，它会被传递给第二个 catch 块。

如此，直到异常被捕获或者通过所有的 catch 块。

## 实例

该实例展示了怎么使用多重 try/catch。

```
try { file = new FileInputStream(fileName); x = (byte) file.read(); }  
catch(FileNotFoundException f) { // Not valid! f.printStackTrace(); return -1; }  
catch(IOException i) { i.printStackTrace(); return -1; }
```

---

## throws/throw 关键字：

如果一个方法没有捕获到一个检查性异常，那么该方法必须使用 throws 关键字来声明。 throws 关键字放在方法签名的尾部。

也可以使用 throw 关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

下面方法的声明抛出一个 RemoteException 异常：

```
import java.io.*; public class className { public void deposit(double amount) throws  
RemoteException { // Method implementation throw new RemoteException(); }  
//Remainder of class definition }
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

例如，下面的方法声明抛出 RemoteException 和 InsufficientFundsException：

```
import java.io.*; public class className { public void withdraw(double amount) throws  
RemoteException, InsufficientFundsException { // Method implementation }  
//Remainder of class definition }
```

---

## finally 关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句。

finally 代码块出现在 catch 代码块最后，语法如下：

```
try{ // 程序代码 }catch(异常类型1 异常的变量名1){ // 程序代码 }catch(异常类型2 异常的变  
量名2){ // 程序代码 }finally{ // 程序代码 }
```

## 实例

### ExcepTest.java 文件代码：

```
public class ExcepTest{ public static void main(String args[]){ int a[] = new int[2]; try{  
System.out.println("Access element three :" + a[3]);  
}catch(ArrayIndexOutOfBoundsException e){ System.out.println("Exception thrown :" +  
e); } finally{ a[0] = 6; System.out.println("First element value: " +a[0]);  
System.out.println("The finally statement is executed"); } } }
```

以上实例编译运行结果如下：

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3  
First element value: 6  
The finally statement is executed
```

注意下面事项：

- catch 不能独立于 try 存在。
  - 在 try/catch 后面添加 finally 块并非强制性要求的。
  - try 代码后不能既没 catch 块也没 finally 块。
  - try, catch, finally 块之间不能添加任何代码。
-

## 声明自定义异常

---

在 Java 中你可以自定义异常。编写自己的异常类时需要记住下面的几点。

- 所有异常都必须是 Throwable 的子类。
- 如果希望写一个检查性异常类，则需要继承 Exception 类。
- 如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

可以像下面这样定义自己的异常类：

```
class MyException extends Exception{ }
```

只继承Exception类来创建的异常类是检查性异常类。

下面的 InsufficientFundsException 类是用户定义的异常类，它继承自 Exception。

一个异常类和其它任何类一样，包含有变量和方法。

### 实例

---

以下实例是一个银行账户的模拟，通过银行卡的号码完成识别，可以进行存钱和取钱的操作。

#### InsufficientFundsException.java 文件代码：

---

```
// 文件名InsufficientFundsException.java import java.io.*; //自定义异常类，继承  
Exception类 public class InsufficientFundsException extends Exception { //此处的  
amount用来储存当出现异常（取出钱多于余额时）所缺乏的钱 private double amount;  
public InsufficientFundsException(double amount) { this.amount = amount; } public  
double getAmount() { return amount; } }
```

为了展示如何使用我们自定义的异常类，

在下面的 CheckingAccount 类中包含一个 withdraw() 方法抛出一个 InsufficientFundsException 异常。

#### CheckingAccount.java 文件代码：

---

```
// 文件名称 CheckingAccount.java import java.io.*; //此类模拟银行账户 public class  
CheckingAccount { //balance为余额，number为卡号 private double balance; private int  
number; public CheckingAccount(int number) { this.number = number; } //方法：存钱  
public void deposit(double amount) { balance += amount; } //方法：取钱 public void  
withdraw(double amount) throws InsufficientFundsException { if(amount <= balance) {  
balance -= amount; } else { double needs = amount - balance; throw new  
InsufficientFundsException(needs); } } //方法：返回余额 public double getBalance() {  
return balance; } //方法：返回卡号 public int getNumber() { return number; } }
```

下面的 BankDemo 程序示范了如何调用 CheckingAccount 类的 deposit() 和 withdraw() 方法。

## **BankDemo.java 文件代码：**

---

```
//文件名称 BankDemo.java public class BankDemo { public static void main(String [] args) { CheckingAccount c = new CheckingAccount(101); System.out.println("Depositing $500..."); c.deposit(500.00); try { System.out.println("\nWithdrawning $100..."); c.withdraw(100.00); System.out.println("\nWithdrawning $600..."); c.withdraw(600.00); }catch(InsufficientFundsException e) { System.out.println("Sorry, but you are short $" + e.getAmount()); e.printStackTrace(); } } }
```

编译上面三个文件，并运行程序 BankDemo，得到结果如下所示：

```
Depositing $500...
Withdrawning $100...
Withdrawning $600...
Sorry, but you are short $200.0
InsufficientFundsException
    at CheckingAccount.withdraw(CheckingAccount.java:25)
    at BankDemo.main(BankDemo.java:13)
```

---

## **通用异常**

---

在Java中定义了两种类型的异常和错误。

- **JVM(Java虚拟机) 异常**：由 JVM 抛出的异常或错误。例如：NullPointerException 类，ArrayIndexOutOfBoundsException 类，ClassCastException 类。
- **程序级异常**：由程序或者API程序抛出的异常。例如 IllegalArgumentException 类，IllegalStateException 类。

[Java Scanner 类](#)

[Java 继承](#)

## **7 篇笔记 写笔记**

---

# Java 继承 | 菜鸟教程

 [runoob.com/java/java-inheritance.html](http://runoob.com/java/java-inheritance.html)

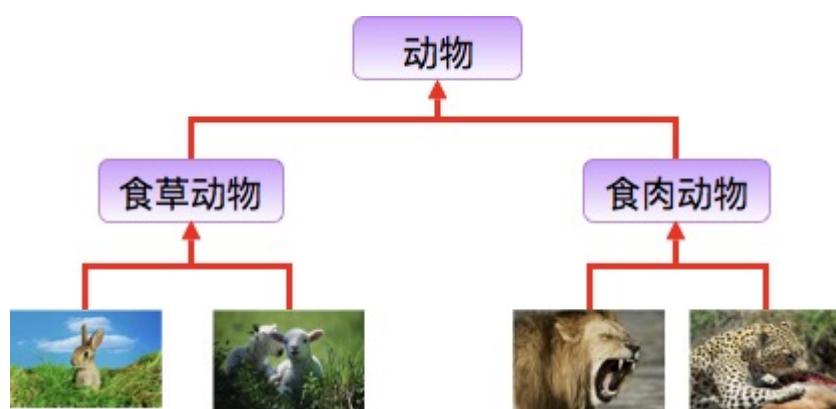
## Java 继承

### 继承的概念

继承是java面向对象编程技术的一块基石，因为它允许创建分等级层次的类。

继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

### 生活中的继承：



兔子和羊属于食草动物类，狮子和豹属于食肉动物类。

食草动物和食肉动物又是属于动物类。

所以继承需要符合的关系是：is-a，父类更通用，子类更具体。

虽然食草动物和食肉动物都是属于动物，但是两者的属性和行为上有差别，所以子类会具有父类的一般特性也会具有自身的特性。

### 类的继承格式

在 Java 中通过 extends 关键字可以声明一个类是从另外一个类继承而来的，一般形式如下：

### 类的继承格式

```
class 父类 {} class 子类 extends 父类 {}
```

### 为什么需要继承

接下来我们通过实例来说明这个需求。

开发动物类，其中动物分别为企鹅以及老鼠，要求如下：

- 企鹅：属性（姓名， id），方法（吃， 睡， 自我介绍）
- 老鼠：属性（姓名， id），方法（吃， 睡， 自我介绍）

### 企鹅类：

---

```
public class Penguin { private String name; private int id; public Penguin(String myName, int myid) { name = myName; id = myid; } public void eat(){ System.out.println(name+"正在吃"); } public void sleep(){ System.out.println(name+"正在睡"); } public void introduction() { System.out.println("大家好！我是" + id + "号" + name + "."); } }
```

### 老鼠类：

---

```
public class Mouse { private String name; private int id; public Mouse(String myName, int myid) { name = myName; id = myid; } public void eat(){ System.out.println(name+"正在吃"); } public void sleep(){ System.out.println(name+"正在睡"); } public void introduction() { System.out.println("大家好！我是" + id + "号" + name + "."); } }
```

从这两段代码可以看出来，代码存在重复了，导致后果就是代码量大且臃肿，而且维护性不高（维护性主要是后期需要修改的时候，就需要修改很多的代码，容易出错），所以要从根本上解决这两段代码的问题，就需要继承，将两段代码中相同的部分提取出来组成一个父类：

### 公共父类：

---

```
public class Animal { private String name; private int id; public Animal(String myName, int myid) { name = myName; id = myid; } public void eat(){ System.out.println(name+"正在吃"); } public void sleep(){ System.out.println(name+"正在睡"); } public void introduction() { System.out.println("大家好！我是" + id + "号" + name + "."); } }
```

这个Animal类就可以作为一个父类，然后企鹅类和老鼠类继承这个类之后，就具有父类当中的属性和方法，子类就不会存在重复的代码，维护性也提高，代码也更加简洁，提高代码的复用性（复用性主要是可以多次使用，不用再多次写同样的代码）继承之后的代码：

### 企鹅类：

---

```
public class Penguin extends Animal { public Penguin(String myName, int myid) { super(myName, myid); } }
```

### 老鼠类：

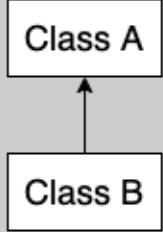
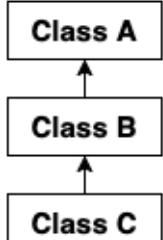
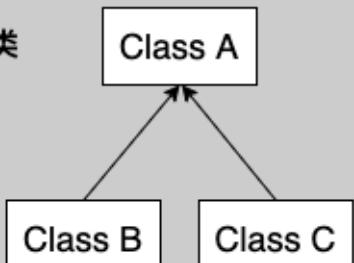
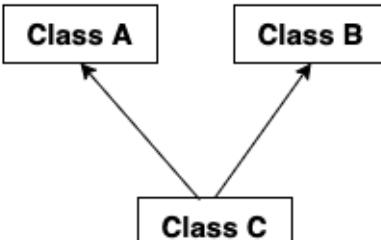
---

```
public class Mouse extends Animal { public Mouse(String myName, int myid) { super(myName, myid); } }
```

### 继承类型

---

需要注意的是 Java 不支持多继承，但支持多重继承。

<b>单继承</b> 	<pre>public class A {     .... } public class B extends A {     .... }</pre>
<b>多重继承</b> 	<pre>public class A {.....} public class B extends A {.....} public class C extends B {.....}</pre>
<b>不同类继承同一个类</b> 	<pre>public class A {.....} public class B extends A {.....} public class C extends A {.....}</pre>
<b>多继承（不支持）</b> 	<pre>public class A {.....} public class B {.....} public class C extends A, B {     .... } // Java 不支持多继承</pre>

## 继承的特性

- 子类拥有父类非 private 的属性、方法。
- 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。
- 子类可以用自己的方式实现父类的方法。
- Java 的继承是单继承，但是可以多重继承，单继承就是一个子类只能继承一个父类，多重继承就是，例如 B 类继承 A 类，C 类继承 B 类，所以按照关系就是 B 类是 C 类的父类，A 类是 B 类的父类，这是 Java 继承区别于 C++ 继承的一个特性。
- 提高了类之间的耦合性（继承的缺点，耦合度高就会造成代码之间的联系越紧密，代码独立性越差）。

## 继承关键字

---

继承可以使用 extends 和 implements 这两个关键字来实现继承，而且所有的类都是继承于 java.lang.Object，当一个类没有继承的两个关键字，则默认继承Object（这个类在 **java.lang** 包中，所以不需要 import）祖先类。

### extends 关键字

---

在 Java 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类，所以 extends 只能继承一个类。

### extends 关键字

---

```
public class Animal { private String name; private int id; public Animal(String myName, String myid) { //初始化属性值 } public void eat() { //吃东西方法的具体实现 } public void sleep() { //睡觉方法的具体实现 } } public class Penguin extends Animal{ }
```

### implements 关键字

---

使用 implements 关键字可以使 Java 具有多继承的特性，使用范围为类继承接口的情况，可以同时继承多个接口（接口跟接口之间采用逗号分隔）。

### implements 关键字

---

```
public interface A { public void eat(); public void sleep(); } public interface B { public void show(); } public class C implements A,B { }
```

### super 与 this 关键字

---

super 关键字：我们可以通过 super 关键字来实现对父类成员的访问，用来引用当前对象的父类。

this 关键字：指向自己的引用。

## 实例

---

```
class Animal { void eat() { System.out.println("animal : eat"); } } class Dog extends Animal { void eat() { System.out.println("dog : eat"); } void eatTest() { this.eat(); // this 调用自己的方法 super.eat(); // super 调用父类方法 } } public class Test { public static void main(String[] args) { Animal a = new Animal(); a.eat(); Dog d = new Dog(); d.eatTest(); } }
```

输出结果为：

```
animal : eat  
dog : eat  
animal : eat
```

### final 关键字

---

`final` 关键字声明类可以把类定义为不能继承的，即最终类；或者用于修饰方法，该方法不能被子类重写：

- 声明类：

```
final class 类名 { //类体}
```

- 声明方法：

```
修饰符(public/private/default/protected) final 返回值类型 方法名() { //方法体}
```

注：实例变量也可以被定义为 `final`，被定义为 `final` 的变量不能被修改。被声明为 `final` 类的方法自动地声明为 `final`，但是实例变量并不是 `final`

## 构造器

子类是不继承父类的构造器（构造方法或者构造函数）的，它只是调用（隐式或显式）。如果父类的构造器带有参数，则必须在子类的构造器中显式地通过 `super` 关键字调用父类的构造器并配以适当的参数列表。

如果父类构造器没有参数，则在子类的构造器中不需要使用 `super` 关键字调用父类构造器，系统会自动调用父类的无参构造器。

## 实例

```
class SuperClass { private int n; SuperClass(){ System.out.println("SuperClass()"); }
SuperClass(int n) { System.out.println("SuperClass(int n)"); this.n = n; } } // SubClass 类继承
class SubClass extends SuperClass{ private int n; SubClass(){ // 自动调用父类的无参数构造器
System.out.println("SubClass"); } public SubClass(int n){ super(300); // 调用父类中带有参数的构造器
System.out.println("SubClass(int n):"+n); this.n = n; } } // SubClass2 类继承
class SubClass2 extends SuperClass{ private int n; SubClass2(){ super(300); // 调用父类中带有参数的构造器
System.out.println("SubClass2"); } public SubClass2(int n){ // 自动调用父类的无参数构造器
System.out.println("SubClass2(int n):"+n); this.n = n; } } public class TestSuperSub{
public static void main (String args[]){
System.out.println("-----SubClass 类继承-----");
SubClass sc1 = new SubClass();
SubClass sc2 = new SubClass(100);
System.out.println("-----SubClass2 类继承-----");
SubClass2 sc3 = new SubClass2();
SubClass2 sc4 = new SubClass2(200); } }
```

输出结果为：

```
-----SubClass 类继承-----
SuperClass()
SubClass
SuperClass(int n)
SubClass(int n):100
-----SubClass2 类继承-----
SuperClass(int n)
SubClass2
SuperClass()
SubClass2(int n):200
```

# 10 篇笔记 写笔记

---

# Java 重写(Override)与重载(Overload)

 [runoob.com/java/java-override-overload.html](http://runoob.com/java/java-override-overload.html)

[Java 继承](#)

[Java 多态](#)

## 重写(Override)

重写是子类对父类的允许访问的方法的实现过程进行重新编写，返回值和形参都不能改变。即  
外壳不变，核心重写！

重写的好处在于子类可以根据需要，定义特定于自己的行为。也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如：父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，只能抛出 IOException 的子类异常。

在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

### TestDog.java 文件代码：

```
class Animal{ public void move(){ System.out.println("动物可以移动"); } } class Dog  
extends Animal{ public void move(){ System.out.println("狗可以跑和走"); } } public class  
TestDog{ public static void main(String args[]){ Animal a = new Animal(); // Animal 对象  
Animal b = new Dog(); // Dog 对象 a.move(); // 执行 Animal 类的方法 b.move(); // 执行  
Dog 类的方法 } }
```

以上实例编译运行结果如下：

```
动物可以移动  
狗可以跑和走
```

在上面的例子中可以看到，尽管 b 属于 Animal 类型，但是它运行的是 Dog 类的 move 方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java 虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为 Animal 类中存在 move 方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

### TestDog.java 文件代码：

```
class Animal{ public void move(){ System.out.println("动物可以移动"); } } class Dog  
extends Animal{ public void move(){ System.out.println("狗可以跑和走"); } public void  
bark(){ System.out.println("狗可以吠叫"); } } public class TestDog{ public static void  
main(String args[]){ Animal a = new Animal(); // Animal 对象 Animal b = new Dog(); //  
Dog 对象 a.move(); // 执行 Animal 类的方法 b.move(); // 执行 Dog 类的方法 b.bark(); } }
```

以上实例编译运行结果如下：

```
TestDog.java:30: cannot find symbol  
symbol : method bark()  
location: class Animal  
        b.bark();  
               ^
```

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

---

## 方法的重写规则

---

- 参数列表与被重写方法的参数列表必须完全相同。
- 返回类型与被重写方法的返回类型可以不相同，但是必须是父类返回值的派生类（java5及更早版本返回类型要一样，java7及更高版本可以不同）。
- 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为 public，那么在子类中重写该方法就不能声明为 protected。
- 父类的成员方法只能被它的子类重写。
- 声明为 final 的方法不能被重写。
- 声明为 static 的方法不能被重写，但是能够被再次声明。
- 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为 private 和 final 的方法。
- 子类和父类不在同一个包中，那么子类只能重写父类的声明为 public 和 protected 的非 final 方法。
- 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
- 构造方法不能被重写。
- 如果不能继承一个类，则不能重写该类的方法。

---

## Super 关键字的使用

---

当需要在子类中调用父类的被重写方法时，要使用 super 关键字。

## TestDog.java 文件代码 :

---

```
class Animal{ public void move(){ System.out.println("动物可以移动"); } } class Dog  
extends Animal{ public void move(){ super.move(); // 应用super类的方法  
System.out.println("狗可以跑和走"); } } public class TestDog{ public static void  
main(String args[]){ Animal b = new Dog(); // Dog 对象 b.move(); // 执行 Dog类的方法 } }
```

以上实例编译运行结果如下：

```
动物可以移动  
狗可以跑和走
```

## 重载(Overload)

---

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

### 重载规则:

- 被重载的方法必须改变参数列表(参数个数或类型不一样)；
- 被重载的方法可以改变返回类型；
- 被重载的方法可以改变访问修饰符；
- 被重载的方法可以声明新的或更广的检查异常；
- 方法能够在同一个类中或者在一个子类中被重载。
- 无法以返回值类型作为重载函数的区分标准。

## 实例

---

### Overloading.java 文件代码 :

---

```
public class Overloading { public int test(){ System.out.println("test1"); return 1; } public  
void test(int a){ System.out.println("test2"); } // 以下两个参数类型顺序不同 public String  
test(int a,String s){ System.out.println("test3"); return "returntest3"; } public String  
test(String s,int a){ System.out.println("test4"); return "returntest4"; } public static void  
main(String[] args){ Overloading o = new Overloading(); System.out.println(o.test());  
o.test(1); System.out.println(o.test(1,"test3")); System.out.println(o.test("test4",1)); } }
```

## 重写与重载之间的区别

---

区别点    重载方法    重写方法

---

参数列表    必须修改    一定不能修改

---

---

返回类型 可以修改 一定不能修改

---

异常 可以修改 可以减少或删除，一定不能抛出新的或者更广的异常

---

访问 可以修改 一定不能做更严格的限制（可以降低限制）

---

## 总结

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。

- (1)方法重载是一个类中定义了多个方法名相同,而他们的参数的数量不同或数量相同而类型和次序不同,则称为方法的重载(Overloading)。
- (2)方法重写是在子类存在方法与父类的方法的名字相同,而且参数的个数与类型一样,返回值也一样的方法,就称为重写(Overriding)。
- (3)方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。

### Overriding 重写

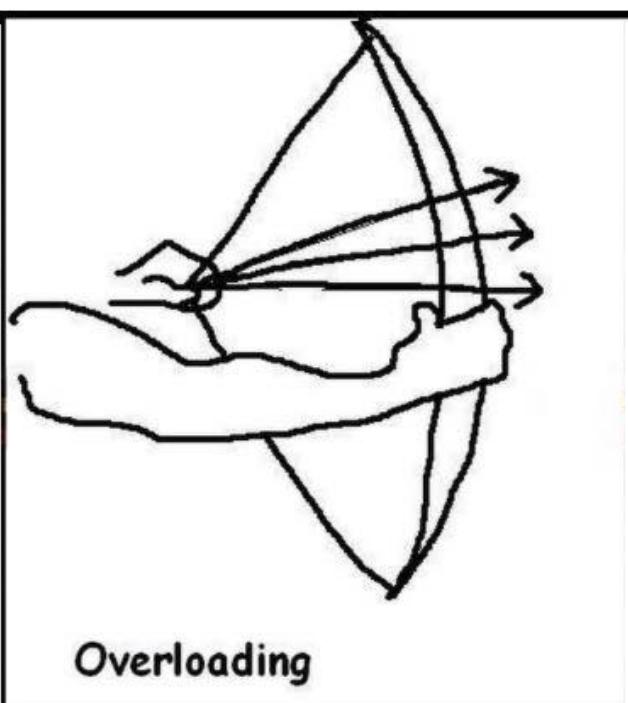
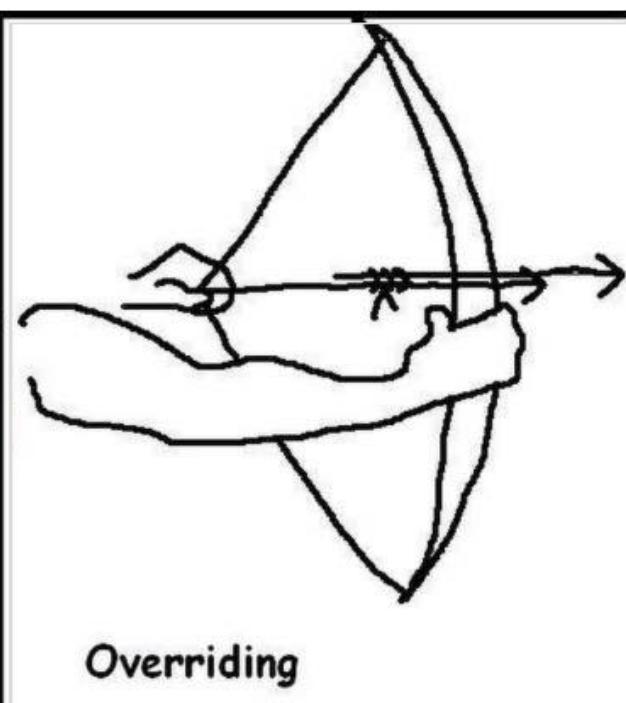
```
class Dog{  
    public void bark(){  
        System.out.println("woof ");  
    }  
}  
  
class Hound extends Dog{  
    public void sniff(){  
        System.out.println("sniff ");  
    }  
  
    public void bark(){  
        System.out.println("bowl");  
    }  
}
```

方法名与参数都一样

### Overloading 重载

```
class Dog{  
    public void bark(){  
        System.out.println("woof ");  
    }  
}  
  
//overloading method  
public void bark(int num){  
    for(int i=0; i<num; i++)  
        System.out.println("woof ");  
}
```

方法名相同，参数不同



Java 继承

Java 多态

## 11 篇笔记 写笔记

---

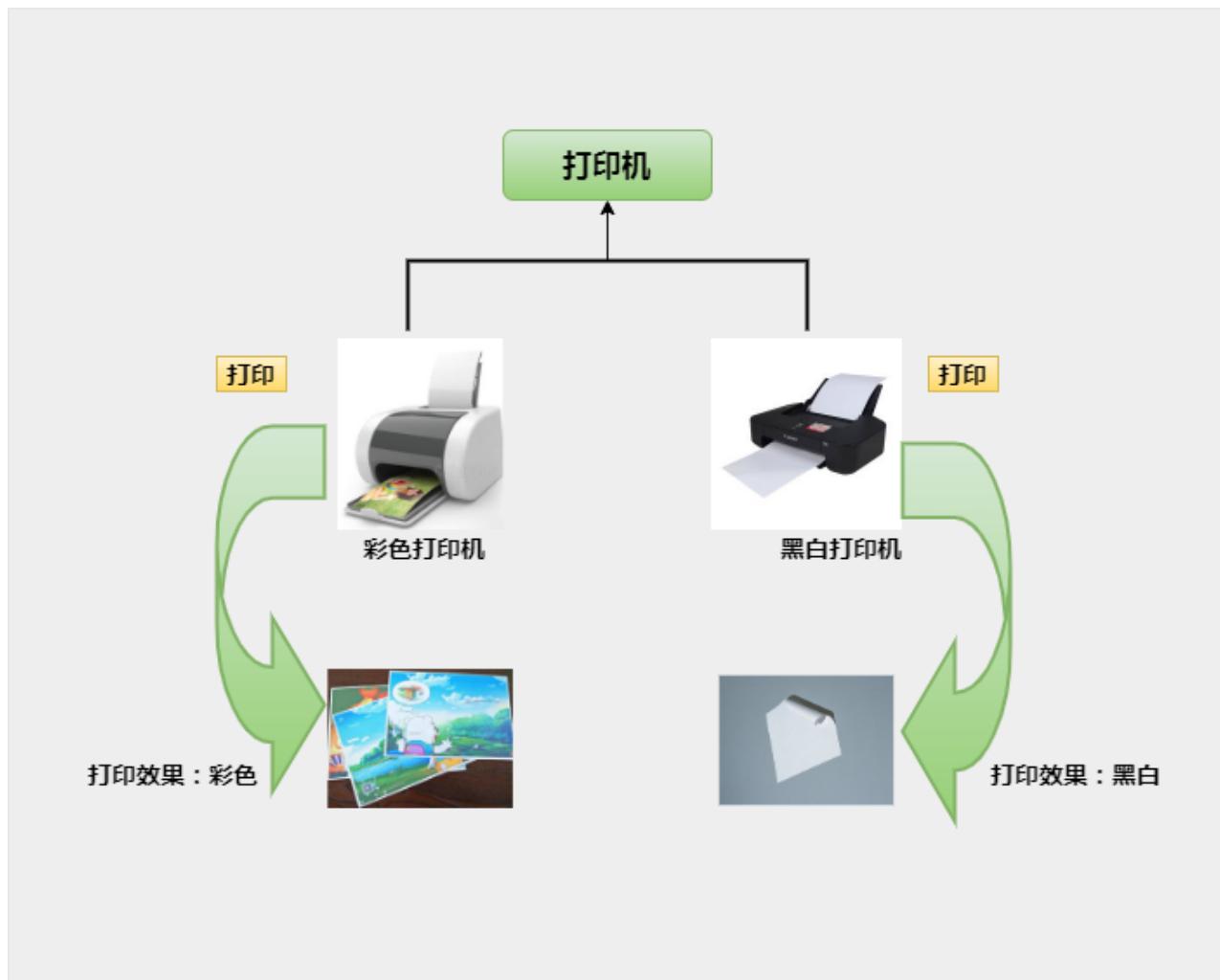
# Java 多态 | 菜鸟教程

 [runoob.com/java/java-polymorphism.html](http://runoob.com/java/java-polymorphism.html)

## Java 多态

多态是同一个行为具有多个不同表现形式或形态的能力。

多态就是同一个接口，使用不同的实例而执行不同操作，如图所示：



多态性是对象多种表现形式的体现。

现实中，比如我们按下 F1 键这个动作：

- 如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；
- 如果当前在 Word 下弹出的就是 Word 帮助；
- 在 Windows 下弹出的就是 Windows 帮助和支持。

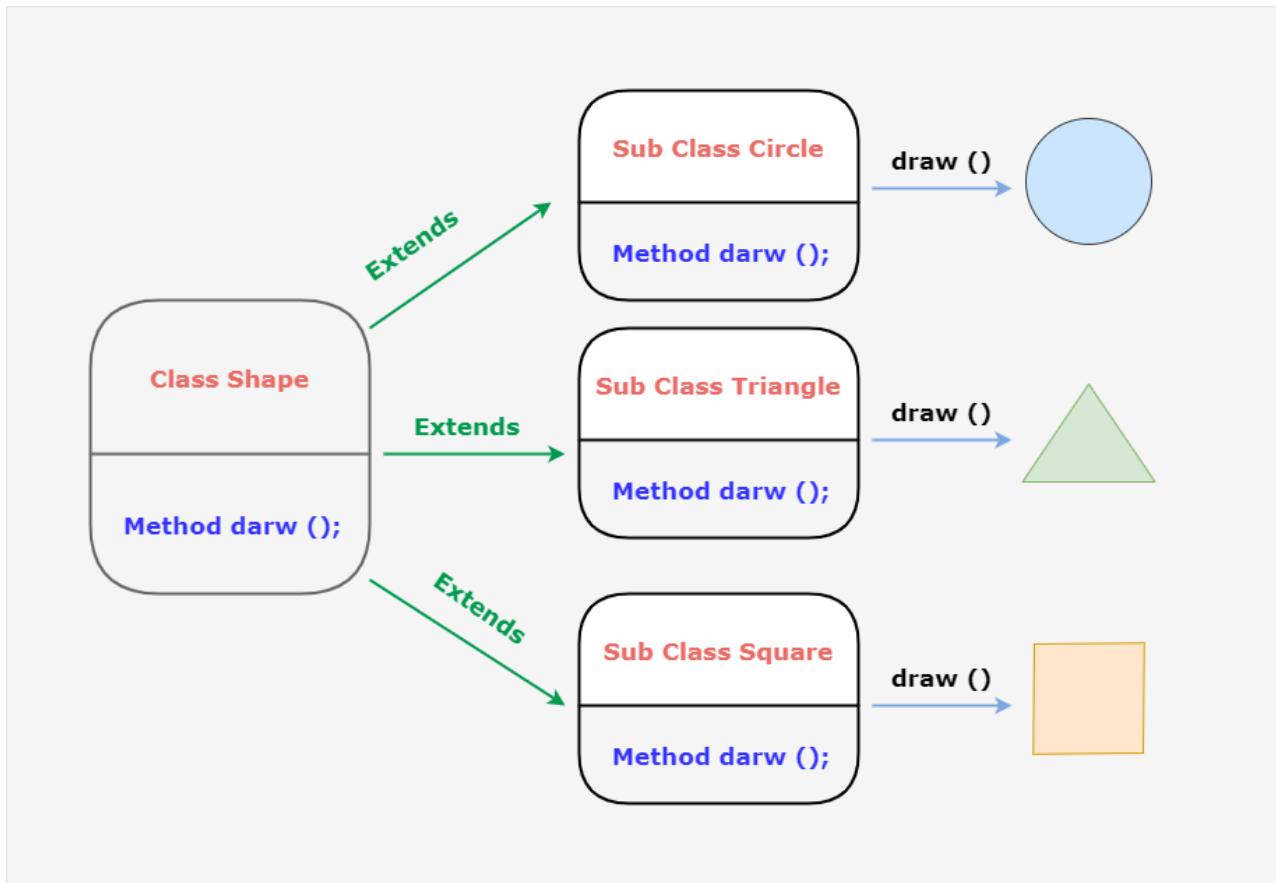
同一个事件发生在不同的对象上会产生不同的结果。

## 多态的优点

- 1. 消除类型之间的耦合关系
- 2. 可替换性
- 3. 可扩充性
- 4. 接口性
- 5. 灵活性
- 6. 简化性

## 多态存在的三个必要条件

- 继承
- 重写
- 父类引用指向子类对象：Parent p = new Child();



```
classShape{
void draw(){}
}
```

```
class Circle extendsShape{
void draw(){
System.out.println("Circle.draw()");
}
}
```

```
class Square extends Shape{  
void draw(){  
System.out.println("Square.draw()");  
}  
}  
  
class Triangle extends Shape {  
void draw() {  
System.out.println("Triangle.draw()");  
}  
}  
}
```

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

以下是一个多态实例的演示，详细说明请看注释：

## Test.java 文件代码：

```
public class Test { public static void main(String[] args) { show(new Cat()); // 以 Cat 对象  
调用 show 方法 show(new Dog()); // 以 Dog 对象调用 show 方法 Animal a = new Cat(); //  
向上转型 a.eat(); // 调用的是 Cat 的 eat Cat c = (Cat)a; // 向下转型 c.work(); // 调用的是  
Cat 的 work } public static void show(Animal a) { a.eat(); // 类型判断 if (a instanceof Cat)  
{ // 猫做的事情 Cat c = (Cat)a; c.work(); } else if (a instanceof Dog) { // 狗做的事情 Dog c =  
(Dog)a; c.work(); } } abstract class Animal { abstract void eat(); } class Cat extends  
Animal { public void eat() { System.out.println("吃鱼"); } public void work() {  
System.out.println("抓老鼠"); } } class Dog extends Animal { public void eat() {  
System.out.println("吃骨头"); } public void work() { System.out.println("看家"); } }
```

执行以上程序，输出结果为：

```
吃鱼  
抓老鼠  
吃骨头  
看家  
吃鱼  
抓老鼠
```

## 虚函数

虚函数的存在是为了多态。

Java 中其实没有虚函数的概念，它的普通函数就相当于 C++ 的虚函数，动态绑定是 Java 的默认行为。如果 Java 中不希望某个函数具有虚函数特性，可以加上 final 关键字变成非虚函数。

## 重写

我们将介绍在 Java 中，当设计类时，被重写的方法的行为怎样影响多态性。

我们已经讨论了方法的重写，也就是子类能够重写父类的方法。

当子类对象调用重写的方法时，调用的是子类的方法，而不是父类中被重写的方法。

要想调用父类中被重写的方法，则必须使用关键字 **super**。

## Employee.java 文件代码：

---

```
/* 文件名 : Employee.java */ public class Employee { private String name; private String address; private int number; public Employee(String name, String address, int number) { System.out.println("Employee 构造函数"); this.name = name; this.address = address; this.number = number; } public void mailCheck() { System.out.println("邮寄支票给：" + this.name + " " + this.address); } public String toString() { return name + " " + address + " " + number; } public String getName() { return name; } public String getAddress() { return address; } public void setAddress(String newAddress) { address = newAddress; } public int getNumber() { return number; } }
```

假设下面的类继承Employee类：

## Salary.java 文件代码：

---

```
/* 文件名 : Salary.java */ public class Salary extends Employee { private double salary; // 全年工资 public Salary(String name, String address, int number, double salary) { super(name, address, number); setSalary(salary); } public void mailCheck() { System.out.println("Salary 类的 mailCheck 方法"); System.out.println("邮寄支票给：" + getName() + "， 工资为：" + salary); } public double getSalary() { return salary; } public void setSalary(double newSalary) { if(newSalary >= 0.0) { salary = newSalary; } } public double computePay() { System.out.println("计算工资，付给：" + getName()); return salary/52; } }
```

现在我们仔细阅读下面的代码，尝试给出它的输出结果：

## VirtualDemo.java 文件代码：

---

```
/* 文件名 : VirtualDemo.java */ public class VirtualDemo { public static void main(String [] args) { Salary s = new Salary("员工 A", "北京", 3, 3600.00); Employee e = new Salary("员工 B", "上海", 2, 2400.00); System.out.println("使用 Salary 的引用调用 mailCheck -- "); s.mailCheck(); System.out.println("\n使用 Employee 的引用调用 mailCheck--"); e.mailCheck(); } }
```

以上实例编译运行结果如下：

```
Employee 构造函数  
Employee 构造函数  
使用 Salary 的引用调用 mailCheck --  
Salary 类的 mailCheck 方法  
邮寄支票给：员工 A , 工资为：3600.0
```

```
使用 Employee 的引用调用 mailCheck--  
Salary 类的 mailCheck 方法  
邮寄支票给：员工 B , 工资为：2400.0
```

## 例子解析

---

- 实例中，实例化了两个 Salary 对象：一个使用 Salary 引用 s，另一个使用 Employee 引用 e。
- 当调用 s.mailCheck() 时，编译器在编译时会在 Salary 类中找到 mailCheck()，执行过程 JVM 就调用 Salary 类的 mailCheck()。
- e 是 Employee 的引用，但引用 e 最终运行的是 Salary 类的 mailCheck() 方法。
- 在编译的时候，编译器使用 Employee 类中的 mailCheck() 方法验证该语句，但是在运行的时候，Java 虚拟机 (JVM) 调用的是 Salary 类中的 mailCheck() 方法。

以上整个过程被称为虚拟方法调用，该方法被称为虚拟方法。

Java 中所有的方法都能以这种方式表现，因此，重写的方法能在运行时调用，不管编译的时候源代码中引用变量是什么数据类型。

---

## 多态的实现方式

---

### 方式一：重写：

这个内容已经在上一章节详细讲过，就不再阐述，详细可访问：[Java 重写\(Override\)与重载\(Overload\)](#)。

### 方式二：接口

---

- 1. 生活中的接口最具代表性的就是插座，例如一个三接头的插头都能接在三孔插座中，因为这个是每个国家都有各自规定的接口规则，有可能到国外就不行，那是因为国外自己定义的接口类型。
- 2. Java 中的接口类似于生活中的接口，就是一些方法特征的集合，但没有方法的实现。具体可以看 [java 接口](#) 这一章节的内容。

### 方式三：抽象类和抽象方法

---

详情请看 [Java 抽象类](#) 章节。

## 11 篇笔记 写笔记

---



# Java 抽象类 | 菜鸟教程

---

 [runoob.com/java/java-abstraction.html](http://runoob.com/java/java-abstraction.html)

[Java 多态](#)

[Java 封装](#)

## Java 抽象类

---

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。也是因为这个原因，通常在设计阶段决定要不要设计抽象类。

父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。

在Java中抽象类表示的是一种继承关系，一个类只能继承一个抽象类，而一个类却可以实现多个接口。

---

## 抽象类

---

在Java语言中使用abstract class来定义抽象类。如下实例：

### Employee.java 文件代码：

---

```
/* 文件名 : Employee.java */ public abstract class Employee { private String name; private String address; private int number; public Employee(String name, String address, int number) { System.out.println("Constructing an Employee"); this.name = name; this.address = address; this.number = number; } public double computePay() { System.out.println("Inside Employee computePay"); return 0.0; } public void mailCheck() { System.out.println("Mailing a check to " + this.name + " " + this.address); } public String toString() { return name + " " + address + " " + number; } public String getName() { return name; } public String getAddress() { return address; } public void setAddress(String newAddress) { address = newAddress; } public int getNumber() { return number; } }
```

注意到该 Employee 类没有什么不同，尽管该类是抽象类，但是它仍然有 3 个成员变量，7 个成员方法和 1 个构造方法。现在如果你尝试如下的例子：

### AbstractDemo.java 文件代码：

---

---

```
/* 文件名 : AbstractDemo.java */ public class AbstractDemo { public static void main(String [] args) { /* 以下是不允许的, 会引发错误 */ Employee e = new Employee("George W.", "Houston, TX", 43); System.out.println("\n Call mailCheck using Employee reference--"); e.mailCheck(); } }
```

当你尝试编译AbstractDemo类时，会产生如下错误：

```
Employee.java:46: Employee is abstract; cannot be instantiated  
    Employee e = new Employee("George W.", "Houston, TX", 43);  
                           ^  
1 error
```

---

## 继承抽象类

我们能通过一般的方法继承Employee类：

### Salary.java 文件代码：

```
/* 文件名 : Salary.java */ public class Salary extends Employee { private double salary;  
//Annual salary public Salary(String name, String address, int number, double salary) {  
super(name, address, number); setSalary(salary); } public void mailCheck() {  
System.out.println("Within mailCheck of Salary class "); System.out.println("Mailing  
check to " + getName() + " with salary " + salary); } public double getSalary() { return  
salary; } public void setSalary(double newSalary) { if(newSalary >= 0.0) { salary =  
newSalary; } } public double computePay() { System.out.println("Computing salary pay  
for " + getName()); return salary/52; } }
```

尽管我们不能实例化一个Employee类的对象，但是如果我们实例化一个Salary类对象，该对象将从Employee类继承7个成员方法，且通过该方法可以设置或获取三个成员变量。

### AbstractDemo.java 文件代码：

```
/* 文件名 : AbstractDemo.java */ public class AbstractDemo { public static void main(String [] args) { Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3,  
3600.00); Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);  
System.out.println("Call mailCheck using Salary reference --"); s.mailCheck();  
System.out.println("\n Call mailCheck using Employee reference--"); e.mailCheck(); } }
```

以上程序编译运行结果如下：

```
Constructing an Employee  
Constructing an Employee  
Call mailCheck using Salary reference --  
Within mailCheck of Salary class  
Mailing check to Mohd Mohtashim with salary 3600.0  
  
Call mailCheck using Employee reference--  
Within mailCheck of Salary class  
Mailing check to John Adams with salary 2400.
```

## 抽象方法

---

如果你想设计这样一个类，该类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么你可以在父类中声明该方法为抽象方法。

Abstract 关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

抽象方法没有定义，方法名后面直接跟一个分号，而不是花括号。

```
public abstract class Employee { private String name; private String address; private int number; public abstract double computePay(); //其余代码 }
```

声明抽象方法会造成以下两个结果：

- 如果一个类包含抽象方法，那么该类必须是抽象类。
- 任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

继承抽象方法的子类必须重写该方法。否则，该子类也必须声明为抽象类。最终，必须有子类实现该抽象方法，否则，从最初的父类到最终的子类都不能用来实例化对象。

如果Salary类继承了Employee类，那么它必须实现computePay()方法：

**Salary.java 文件代码：**

```
/* 文件名 : Salary.java */ public class Salary extends Employee { private double salary; // Annual salary public double computePay() { System.out.println("Computing salary pay for " + getName()); return salary/52; } //其余代码 }
```

---

## 抽象类总结规定

- 1. 抽象类不能被实例化(初学者很容易犯的错)，如果被实例化，就会报错，编译无法通过。只有抽象类的非抽象子类可以创建对象。
- 2. 抽象类中不一定包含抽象方法，但是有抽象方法的类必定是抽象类。
- 3. 抽象类中的抽象方法只是声明，不包含方法体，就是不给出方法的具体实现也就是方法的具体功能。
- 4. 构造方法，类方法（用 static 修饰的方法）不能声明为抽象方法。
- 5. 抽象类的子类必须给出抽象类中的抽象方法的具体实现，除非该子类也是抽象类。

[Java 多态](#)

[Java 封装](#)

---

## 4 篇笔记 写笔记



# Java 封装 | 菜鸟教程

---

 [runoob.com/java/java-encapsulation.html](https://runoob.com/java/java-encapsulation.html)

[Java 抽象类](#)

[Java 接口](#)

## Java 封装

---

在面向对象程式设计方法中，封装（英语：Encapsulation）是指一种将抽象性函式接口的实现细节部分包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

## 封装的优点

---

- 1. 良好的封装能够减少耦合。
  - 2. 类内部的结构可以自由修改。
  - 3. 可以对成员变量进行更精确的控制。
  - 4. 隐藏信息，实现细节。
- 

## 实现Java封装的步骤

---

1. 修改属性的可见性来限制对属性的访问（一般限制为private），例如：

```
public class Person { private String name; private int age; }
```

这段代码中，将 **name** 和 **age** 属性设置为私有的，只能本类才能访问，其他类都访问不了，如此就对信息进行了隐藏。

2. 对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问，例如：

```
public class Person{ private String name; private int age; public int getAge(){ return age; } public String getName(){ return name; } public void setAge(int age){ this.age = age; } public void setName(String name){ this.name = name; } }
```

采用 **this** 关键字是为了解决实例变量 (private String name) 和局部变量 (setName(String name)中的name变量) 之间发生的同名的冲突。

---

## 实例

让我们来看一个java封装类的例子：

### EncapTest.java 文件代码：

```
/* 文件名: EncapTest.java */ public class EncapTest{ private String name; private String idNum; private int age; public int getAge(){ return age; } public String getName(){ return name; } public String getIdNum(){ return idNum; } public void setAge( int newAge){ age = newAge; } public void setName(String newName){ name = newName; } public void setIdNum( String newId){ idNum = newId; } }
```

以上实例中public方法是外部类访问该类成员变量的入口。

通常情况下，这些方法被称为getter和setter方法。

因此，任何要访问类中私有成员变量的类都要通过这些getter和setter方法。

通过如下的例子说明EncapTest类的变量怎样被访问：

### RunEncap.java 文件代码：

```
/* F文件名 : RunEncap.java */ public class RunEncap{ public static void main(String args[]){ EncapTest encap = new EncapTest(); encap.setName("James"); encap.setAge(20); encap.setIdNum("12343ms"); System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge()); } }
```

以上代码编译运行结果如下：

Name : James Age : 20

[Java 抽象类](#)

[Java 接口](#)

[点我分享笔记](#)

---

# Java 接口 | 菜鸟教程

 [runoob.com/java/java-interfaces.html](https://runoob.com/java/java-interfaces.html)

[Java 封装](#)

[Java 枚举](#)

## Java 接口

接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

### 接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在 .java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在 .class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

### 接口与类的区别：

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多继承。

### 接口特性

- 接口中每一个方法也是隐式抽象的，接口中的方法会被隐式的指定为 **public abstract**（只能是 public abstract，其他修饰符都会报错）。
- 接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量（并且只能是 public，用 private 修饰会报编译错误）。
- 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

### 抽象类和接口的区别

- 1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。

- 2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的。
- 3. 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。
- 4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

| 注：JDK 1.8 以后，接口里可以有静态方法和方法体了。

---

## 接口的声明

接口的声明语法格式如下：

[可见度] interface 接口名称 [extends 其他的接口名] { // 声明变量 // 抽象方法 }

Interface 关键字用来声明一个接口。下面是接口声明的一个简单例子。

### NameOfInterface.java 文件代码：

```
/* 文件名 : NameOfInterface.java */ import java.lang.*; // 引入包 public interface NameOfInterface { // 任何类型 final, static 字段 // 抽象方法 }
```

接口有以下特性：

- 接口是隐式抽象的，当声明一个接口的时候，不必使用 **abstract** 关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要 **abstract** 关键字。
- 接口中的方法都是公有的。

## 实例

### Animal.java 文件代码：

```
/* 文件名 : Animal.java */ interface Animal { public void eat(); public void travel(); }
```

## 接口的实现

当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。

类使用 **implements** 关键字实现接口。在类声明中， **Implements** 关键字放在 **class** 声明后面。

实现一个接口的语法，可以使用这个公式：

### Animal.java 文件代码：

```
...implements 接口名称[, 其他接口名称, 其他接口名称..., ...] ...
```

## 实例

## MammalInt.java 文件代码：

---

```
/* 文件名 : MammalInt.java */ public class MammalInt implements Animal{ public void eat(){ System.out.println("Mammal eats"); } public void travel(){ System.out.println("Mammal travels"); } public int noOfLegs(){ return 0; } public static void main(String args[]){ MammalInt m = new MammalInt(); m.eat(); m.travel(); } }
```

以上实例编译运行结果如下：

```
Mammal eats  
Mammal travels
```

重写接口中声明的方法时，需要注意以下规则：

- 类在实现接口的方法时，不能抛出强制性异常，只能在接口中，或者继承接口的抽象类中抛出该强制性异常。
- 类在重写方法时要保持一致的方法名，并且应该保持相同或者相兼容的返回值类型。
- 如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口的时候，也要注意一些规则：

- 一个类可以同时实现多个接口。
  - 一个类只能继承一个类，但是能实现多个接口。
  - 一个接口能继承另一个接口，这和类之间的继承比较相似。
- 

## 接口的继承

一个接口能继承另一个接口，和类之间的继承方式比较相似。接口的继承使用extends关键字，子接口继承父接口的方法。

下面的Sports接口被Hockey和Football接口继承：

```
// 文件名: Sports.java public interface Sports { public void setHomeTeam(String name);  
public void setVisitingTeam(String name); } // 文件名: Football.java public interface  
Football extends Sports { public void homeTeamScored(int points); public void  
visitingTeamScored(int points); public void endOfQuarter(int quarter); } // 文件名:  
Hockey.java public interface Hockey extends Sports { public void homeGoalScored();  
public void visitingGoalScored(); public void endOfPeriod(int period); public void  
overtimePeriod(int ot); }
```

Hockey接口自己声明了四个方法，从Sports接口继承了两个方法，这样，实现Hockey接口的类需要实现六个方法。

相似的，实现Football接口的类需要实现五个方法，其中两个来自于Sports接口。

---

## 接口的多继承

在Java中，类的多继承是不合法，但接口允许多继承。

在接口的多继承中`extends`关键字只需要使用一次，在其后跟着继承接口。如下所示：

```
public interface Hockey extends Sports, Event
```

以上的程序片段是合法定义的子接口，与类不同的是，接口允许多继承，而`Sports`及`Event`可能定义或是继承相同的方法

---

## 标记接口

最常用的继承接口是没有包含任何方法的接口。

标记接口是没有任何方法和属性的接口。它仅仅表明它的类属于一个特定的类型，供其他代码来测试允许做一些事情。

标记接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：`java.awt.event`包中的`MouseListener`接口继承的`java.util.EventListener`接口定义如下：

```
package java.util; public interface EventListener {}
```

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

- 建立一个公共的父接口：

正如`EventListener`接口，这是由几十个其他接口扩展的Java API，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了`EventListener`接口，Java虚拟机(JVM)就知道该接口将要被用于一个事件的代理方案。

- 向一个类添加数据类型：

这种情况是标记接口最初的目的，实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

[Java 封装](#)

[Java 枚举](#)

## 8 篇笔记 写笔记

---

# Java 枚举(enum) | 菜鸟教程

---

 [runoob.com/java/java-enum.html](http://runoob.com/java/java-enum.html)

[Java 接口](#)

[Java 包\(package\)](#)

## Java 枚举(enum)

---

Java 枚举是一个特殊的类，一般表示一组常量，比如一年的 4 个季节，一个年的 12 个月份，一个星期的 7 天，方向有东南西北等。

Java 枚举类使用 enum 关键字来定义，各个常量使用逗号，来分割。

例如定义一个颜色的枚举类。

```
enum Color
{
    RED, GREEN, BLUE;
}
```

以上枚举类 Color 颜色常量有 RED, GREEN, BLUE，分别表示红色，绿色，蓝色。

使用实例：

## 实例

---

```
enumColor
{
    RED, GREEN, BLUE;
}

public class Test
{
    // 执行输出结果
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

执行以上代码输出结果为：

RED

## 内部类中使用枚举

---

枚举类也可以声明在内部类中：

## 实例

---

```
public class Test
{
enumColor
{
    RED, GREEN, BLUE;
}

// 执行输出结果
public static void main(String[] args)
{
    Color c1 = Color.RED;
    System.out.println(c1);
}
}
```

执行以上代码输出结果为：

RED

每个枚举都是通过 Class 在内部实现的，且所有的枚举值都是 public static final 的。

以上的枚举类 Color 转化在内部类实现：

```
class Color
{
    public static final Color RED = new Color();
    public static final Color BLUE = new Color();
    public static final Color GREEN = new Color();
}
```

## 迭代枚举元素

---

可以使用 for 语句来迭代枚举元素：

## 实例

---

```
enum Color
{
    RED, GREEN, BLUE;
}

public class MyClass {
    public static void main(String[] args) {
        for (Color myVar : Color.values()) {
            System.out.println(myVar);
        }
    }
}
```

执行以上代码输出结果为：

```
RED  
GREEN  
BLUE
```

## 在 switch 中使用枚举类

枚举类常应用于 switch 语句中：

### 实例

```
enumColor  
{  
    RED, GREEN, BLUE;  
}  
public class MyClass {  
    public static void main(String[] args){  
        Color myVar = Color.BLUE;  
  
        switch(myVar) {  
            case RED:  
                System.out.println("红色");  
                break;  
            case GREEN:  
                System.out.println("绿色");  
                break;  
            case BLUE:  
                System.out.println("蓝色");  
                break;  
        }  
    }  
}
```

执行以上代码输出结果为：

```
蓝色
```

## values(), ordinal() 和 valueOf() 方法

enum 定义的枚举类默认继承了 java.lang.Enum 类，并实现了 java.lang.Serializable 和 java.lang.Comparable 两个接口。

values(), ordinal() 和 valueOf() 方法位于 java.lang.Enum 类中：

- values() 返回枚举类中所有的值。
- ordinal()方法可以找到每个枚举常量的索引，就像数组索引一样。
- valueOf()方法返回指定字符串值的枚举常量。

## 实例

---

```
enumColor
{
    RED, GREEN, BLUE;
}

public class Test
{
    public static void main(String[] args)
    {
        // 调用 values()
        Color[] arr = Color.values();

        // 迭代枚举
        for(Color col : arr)
        {
            // 查看索引
            System.out.println(col + " at index " + col.ordinal());
        }
    }
}
```

// 使用 valueOf() 返回枚举常量，不存在的会报错  
IllegalArgumentException  
System.out.println(Color.valueOf("RED"));  
// System.out.println(Color.valueOf("WHITE"));  
}

执行以上代码输出结果为：

```
RED at index 0
GREEN at index 1
BLUE at index 2
RED
```

## 枚举类成员

---

枚举跟普通类一样可以用自己的变量、方法和构造函数，构造函数只能使用 private 访问修饰符，所以外部无法调用。

枚举既可以包含具体方法，也可以包含抽象方法。如果枚举类具有抽象方法，则枚举类的每个实例都必须实现它。

## 实例

---

```
enumColor
{
    RED, GREEN, BLUE;

// 构造函数
privateColor()
{
    System.out.println("Constructor called for : "+this.toString());
}
```

```
publicvoid colorInfo()
{
    System.out.println("Universal Color");
}
```

```
    public class Test
    {
        // 输出
        public static void main(String[] args)
        {
            Color c1 = Color.RED;
            System.out.println(c1);
            c1.colorInfo();
        }
    }
```

执行以上代码输出结果为：

```
Constructor called for : RED
Constructor called for : GREEN
Constructor called for : BLUE
RED
Universal Color
```

Java 接口

Java 包(package)

**1 篇笔记 写笔记**

---

# 1. doggy

158\*\*\*85269@163.com

参考地址

42

枚举类中的抽象方法实现，需要枚举类中的每个对象都对其进行实现。

```
enum Color{
    RED{
        public String getColor(){//枚举对象实现抽象方法
            return "红色";
        }
    },
    GREEN{
        public String getColor(){//枚举对象实现抽象方法
            return "绿色";
        }
    },
    BLUE{
        public String getColor(){//枚举对象实现抽象方法
            return "蓝色";
        }
    };
    public abstract String getColor();//定义抽象方法
}

public class Test{
    public static void main(String[] args) {
        for (Color c:Color.values()){
            System.out.print(c.getColor() + "、");
        }
    }
}
```

doggy

doggy

158\*\*\*85269@163.com

参考地址

1个月前 (11-30)

# Java 包(package) | 菜鸟教程

 [runoob.com/java/java-package.html](http://runoob.com/java/java-package.html)

[Java 枚举](#)

[Java 数据结构](#)

## Java 包(package)

为了更好地组织类，Java 提供了包机制，用于区别类名的命名空间。

### 包的作用

- 1、把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。
- 2、如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。
- 3、包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。

Java 使用包（package）这种机制是为了防止命名冲突，访问控制，提供搜索和定位类（class）、接口、枚举（enumerations）和注释（annotation）等。

包语句的语法格式为：

```
package pkg1[. pkg2[. pkg3...]];
```

例如，一个Something.java 文件它的内容

```
package net.java.util; public class Something{ ... }
```

那么它的路径应该是 **net/java/util/Something.java** 这样保存的。 package(包) 的作用是把不同的 java 程序分类保存，更方便的被其他 java 程序调用。

一个包（package）可以定义为一组相互联系的类型（类、接口、枚举和注释），为这些类型提供访问保护和命名空间管理的功能。

以下是一些 Java 中的包：

- **java.lang**-打包基础的类
- **java.io**-包含输入输出功能的函数

开发者可以自己把一组类和接口等打包，并定义自己的包。而且在实际开发中这样做是值得提倡的，当你自己完成类的实现之后，将相关的类分组，可以让其他的编程者更容易地确定哪些类、接口、枚举和注释等是相关的。

由于包创建了新的命名空间（namespace），所以不会跟其他包中的任何名字产生命名冲突。使用包这种机制，更容易实现访问控制，并且让定位相关类更加简单。

## 创建包

---

创建包的时候，你需要为这个包取一个合适的名字。之后，如果其他的一个源文件包含了这个包提供的类、接口、枚举或者注释类型的时候，都必须将这个包的声明放在这个源文件的开头。

包声明应该在源文件的第一行，每个源文件只能有一个包声明，这个文件中的每个类型都应用于它。

如果一个源文件中没有使用包声明，那么其中的类，函数，枚举，注释等将被放在一个无名的包（unnamed package）中。

## 例子

---

让我们来看一个例子，这个例子创建了一个叫做animals的包。通常使用小写的字母来命名避免与类、接口名字的冲突。

在 animals 包中加入一个接口（interface）：

### Animal.java 文件代码：

---

```
/* 文件名: Animal.java */ package animals; interface Animal { public void eat(); public void travel(); }
```

接下来，在同一个包中加入该接口的实现：

### MammalInt.java 文件代码：

---

```
package animals; /* 文件名 : MammalInt.java */ public class MammalInt implements Animal{ public void eat(){ System.out.println("Mammal eats"); } public void travel(){ System.out.println("Mammal travels"); } public int noOfLegs(){ return 0; } public static void main(String args[]){ MammalInt m = new MammalInt(); m.eat(); m.travel(); } }
```

然后，编译这两个文件，并把他们放在一个叫做animals的子目录中。用下面的命令来运行：

```
$ mkdir animals  
$ cp Animal.class MammalInt.class animals  
$ java animals/MammalInt  
Mammal eats  
Mammal travel
```

---

## import 关键字

---

为了能够使用某一个包的成员，我们需要在 Java 程序中明确导入该包。使用 "import" 语句可完成此功能。

在 java 源文件中 import 语句应位于 package 语句之前，所有类的定义之前，可以没有，也可以有多条，其语法格式为：

```
import package1[.package2...].(classname|*);
```

如果在一个包中，一个类想要使用本包中的另一个类，那么该包名可以省略。

## 例子

---

下面的 payroll 包已经包含了 Employee 类，接下来向 payroll 包中添加一个 Boss 类。Boss 类引用 Employee 类的时候可以不用使用 payroll 前缀，Boss 类的实例如下。

### Boss.java 文件代码：

---

```
package payroll; public class Boss { public void payEmployee(Employee e) {  
e.mailCheck(); } }
```

如果 Boss 类不在 payroll 包中又会怎样？Boss 类必须使用下面几种方法之一来引用其他包中的类。

使用类全名描述，例如：

```
payroll.Employee
```

用 **import** 关键字引入，使用通配符 “\*”

```
import payroll.*;
```

使用 **import** 关键字引入 Employee 类：

```
import payroll.Employee;
```

注意：

类文件中可以包含任意数量的 import 声明。import 声明必须在包声明之后，类声明之前。

---

## package 的目录结构

---

类放在包中会有两种主要的结果：

- 包名成为类名的一部分，正如我们前面讨论的一样。
- 包名必须与相应的字节码所在的目录结构相吻合。

下面是管理你自己 java 中文件的一种简单方式：

将类、接口等类型的源码放在一个文本中，这个文件的名字就是这个类型的名字，并以.java作为扩展名。例如：

```
// 文件名：Car.java package vehicle; public class Car { // 类实现 }
```

接下来，把源文件放在一个目录中，这个目录要对应类所在包的名字。

```
....\vehicle\Car.java
```

现在，正确的类名和路径将会是如下样子：

- 类名 -> vehicle.Car
- 路径名 -> vehicle\Car.java (在 windows 系统中)

通常，一个公司使用它互联网域名的颠倒形式来作为它的包名.例如：互联网域名是 runoob.com, 所有的包名都以 com.runoob 开头。包名中的每一个部分对应一个子目录。

例如：有一个 **com.runoob.test** 的包，这个包包含一个叫做 Runoob.java 的源文件，那么相应的，应该有如下面的一连串子目录：

....\com\runoob\test\Runoob.java

编译的时候，编译器为包中定义的每个类、接口等类型各创建一个不同的输出文件，输出文件的名字就是这个类型的名字，并加上 .class 作为扩展后缀。例如：

```
// 文件名: Runoob.java package com.runoob.test; public class Runoob { } class Google { }
```

现在，我们用-d选项来编译这个文件，如下：

```
$javac -d . Runoob.java
```

这样会像下面这样放置编译了的文件：

```
.\com\runoob\test\Runoob.class  
.com\runoob\test\Google.class
```

你可以像下面这样来导入所有 \com\runoob\test\ 中定义的类、接口等：

```
import com.runoob.test.*;
```

编译之后的 .class 文件应该和 .java 源文件一样，它们放置的目录应该跟包的名字对应起来。但是，并不要求 .class 文件的路径跟相应的 .java 的路径一样。你可以分开来安排源码和类的目录。

```
<path-one>\sources\com\runoob\test\Runoob.java  
<path-two>\classes\com\runoob\test\Google.class
```

这样，你可以将你的类目录分享给其他的编程人员，而不用透露自己的源码。用这种方法管理源码和类文件可以让编译器和 java 虚拟机 (JVM) 可以找到你程序中使用的所有类型。

类目录的绝对路径叫做 **class path**。设置在系统变量 **CLASSPATH** 中。编译器和 java 虚拟机通过将 package 名字加到 class path 后来构造 .class 文件的路径。

<path-two>\classes 是 class path, package 名字是 com.runoob.test, 而编译器和 JVM 会在 <path-two>\classes\com\runoob\test 中找 .class 文件。

一个 class path 可能会包含好几个路径，多路径应该用分隔符分开。默认情况下，编译器和 JVM 查找当前目录。JAR 文件按包含 Java 平台相关的类，所以他们的目录默认放在了 class path 中。

## 设置 CLASSPATH 系统变量

---

用下面的命令显示当前的CLASSPATH变量：

- Windows 平台 (DOS 命令行下) : C:\> set CLASSPATH
- UNIX 平台 (Bourne shell 下) : # echo \$CLASSPATH

删除当前CLASSPATH变量内容：

- Windows 平台 (DOS 命令行下) : C:\> set CLASSPATH=
- UNIX 平台 (Bourne shell 下) : # unset CLASSPATH; export CLASSPATH

设置CLASSPATH变量：

- Windows 平台 (DOS 命令行下) : C:\> set  
CLASSPATH=C:\users\jack\java\classes
- UNIX 平台 (Bourne shell 下) : # CLASSPATH=/home/jack/java/classes; export  
CLASSPATH

[Java 枚举](#)

[Java 数据结构](#)

## 1 篇笔记 写笔记

---

1. Anne

287\*\*\*935@qq.com

参考地址

96

### Java 中带包（创建及引用）的类的编译

只有一个文件时编译：

javac A.java

一个包的文件都在时编译：

javac -d . \*.java

运行：编译之后会自己生成文件夹，不要进入这个文件夹，直接运行 java -cp /home/test test.Run，其中源文件在 test 文件夹中，包名为 test，启动文件为 Run.java。

更多内容参考：Java 中带包（创建及引用）的类的编译与调试

Anne

Anne

287\*\*\*935@qq.com

参考地址

2年前 (2018-09-05)