

# 第一章 Java 入门

## 教学目标：

- 理解 Java 是什么
- 了解 Java 能干什么
- 理解 Java 有什么
- 了解 Java 的历史、现状和特点
- 理解 Java 从代码到运行的过程
- 理解 Java 虚拟机的功能
- 理解垃圾收集是如何进行的
- 理解 Java 代码安全性
- 掌握 Java 开发环境配置
- 编写、编译并运行简单的 Java 程序

## 一：Java 是什么

---

通常我们认为，Java 是：

- Ø 一种计算机编程语言
- Ø 一种软件开发平台
- Ø 一种软件运行平台
- Ø 一种软件部署环境

什么意思呢，分别解释一下。

### 1: Java 是一种计算机编程语言

---

#### 1.1: 语言

要准确地给语言下一个定义很困难，对我们来说也没有必要，但是大家都知道汉语、英语等是语言。语言是用来干什么的呢？很明显语言是用来交流的，比如大家现在看这些文字，其实就是我们通过这些文字在交流。

#### 1.2: 计算机编程

所谓计算机编程就是：把我们的要求和想法，按照能够让计算机看懂的规则和约定，编写出来的过程，就是编程。编程的结果就是一些计算机能够看懂并能够执行和处理的东西，我们把它叫做软件或者程序。事实上，程序就是我们对计算机发出的命令集（指令集）。

#### 1.3: Java 是一种计算机编程语言

我们说 Java 是一种计算机编程语言，首先是说：Java 是一种语言，也就是 Java 是用来交流的，那么用来谁和谁交流呢？很明显就是人和计算机交流了，换句话说把我们的要求和想法用 Java 表达出来，那么计算机能看懂，就能够按照我们要求运行，而这个过程就是我们说的使用 Java 编程，所以我们讲 Java 是一种计算机编程语言。

为了让计算机看懂，Java 会有一系列的规则和约定，这些就是 Java 的语法。

### 2: Java 是一种软件开发平台

---

#### 2.1: 什么是软件开发

可以简单地理解为：编程的结果是软件或者程序，而编程的过程就是软件开发。软件开发的基本步骤包括：需求分析、概要设计、详细设计、编码、测试、维护等阶段。

#### 2.2: 什么是开发平台

在软件开发的过程中，我们需要很多的工具来辅助我们的工作，不可能什么都从头自己做。我们把编程的环境和相应的辅助工具统称为开发环境，开发平台就是用来提供这个开发环境的。

#### 2.3: Java 是一种开发平台

Java 不单纯是一个编程的语言，它自身提供了一系列开发 Java 所需要的环境和工具，来进行编译、解释、文档生成、打包等，比如：javac.exe、java.exe 等等，这些我们后面会讲到，现在不明白也没有关系。所以我们讲 Java 是一个开发平台

### **3: Java 是一种软件运行平台**

---

#### **3.1: 什么是软件的运行平台**

如同我们需要阳光、空气、水和食物才能正常存活一样，软件最终要能够运行，也需要一系列的外部环境，来为软件的运行提供支持，而提供这些支持的就是运行平台。

#### **3.2: Java 是一种运行平台**

Java 本身提供 Java 软件所需要的运行环境，Java 应用可运行在安装了 JRE (Java Runtime Environment) 的机器上，所以我们说 Java 是一个运行平台。

JRE: Java Runtime Environment, Java 运行环境。

### **4: Java 是一种软件部署环境**

---

#### **4.1: 什么是软件的部署**

简单地讲，部署就是安装，就是把软件放置到相应的地方，并且进行相应的配置（一般称作部署描述），让软件能够正常运行起来。

#### **4.2: Java 是一种软件部署环境**

Java 本身是一个开发的平台，开发后的 Java 程序也是运行在 Java 平台上的。也就是说，开发后的 Java 程序也是部署在 Java 平台上的，这个尤其在后面学习 JEE（Java 的企业版）的时候，体现更为明显。

## **二: Java 能干什么**

---

Java 能做的事情很多，涉及到编程领域的各个方面。

#### **1: 桌面级应用：**尤其是需要跨平台的桌面级应用程序。

先解释一下桌面级应用：简单的说就是主要功能都在我们本机上运行的程序，比如 word、excel 等运行在本机上的应用就属于桌面应用。

#### **2: 企业级应用**

先解释一下企业级应用：简单的说就是大规模的应用，一般使用人数较多，数据量较大，对系统的稳定性、安全性、可扩展性和可装配性等都有比较高的要求。

这是目前 Java 应用最广泛的一个领域，几乎一枝独秀。包括各种行业应用，企业信息化，也包括电子政务等，领域涉及：办公自动化 OA，客户关系管理 CRM，人力资源 HR，企业资源计划 ERP、知识管理 KM、供应链管理 SCM、企业设备管理系统 EAM、产品生命周期管理 PLM、面向服务体系架构 SOA、商业智能 BI、项目管理 PM、营销管理、流程管理 WorkFlow、财务管理.....等等几乎所有你能想到的应用。

#### **3: 嵌入式设备及消费类电子产品**

包括无线手持设备、智能卡、通信终端、医疗设备、信息家电（如数字电视、机顶盒、电冰箱）、汽车电子设备等都是近年以来热门的 Java 应用领域，尤其是手机上的 Java 应用程序和 Java 游戏，更是普及。

**4: 除了上面提到的，Java 还有很多功能：**如进行数学运算、显示图形界面、进行网络操作、进行数据库操作、进行文件的操作等等。

## 三: Java 有什么

---

Java 体系比较庞杂, 功能繁多, 这也导致很多人在自学 Java 的时候总是感觉无法建立全面的知识体系, 无法从整体上把握 Java 的原因。在这里我们先简单了解一下 Java 的版本。具体的 Java 体系知识结构, 将在后面详细讲述。

Java 分成三种版本, 分别是 Java 标准版(JSE)、Java 微缩版(JME)和 Java 企业版(JEE), 每一种版本都有自己的功能和应用方向。

### 1: Java 标准版: JSE(Java Standard Edition)

JSE(Java Standard Edition)是 Sun 公司针对桌面开发以及低端商务计算解决方案而开发的版本, 例如: 我们平常熟悉的 Application 桌面应用程序。这个版本是个基础, 它也是我们平常开发和使用最多的技术, Java 的主要的技术将在这个版本中体现。本书主要讲的就是 JSE。

### 2: Java 微缩版: JME(Java Micro Edition)

JME(Java , Micro Edition) 是对标准版 JSE 进行功能缩减后的版本, 于 1999 年 6 月由 Sun Microsystems 第一次推向 Java 团体。它是一项能更好满足 Java 开发人员不同需求的广泛倡议的一部分。Sun Microsystems 将 JME 定义为“一种以广泛的消费性产品为目标的高度优化的 Java 运行时环境, 包括寻呼机、移动电话、可视电话、数字机顶盒和汽车导航系统。”

JME 是致力于消费产品和嵌入式设备的开发人员的最佳选择。尽管早期人们对它看好而且 Java 开发人员团体中的热衷人士也不少, 然而, JME 最近才开始从其影响更大的同属产品 JEE 和 JSE 的阴影中走出其不成熟期。

JME 在开发面向内存有限的移动终端(例如寻呼机、移动电话)的应用时, 显得尤其实用。因为它是建立在操作系统之上的, 使得应用的开发无须考虑太多特殊的硬件配置类型或操作系统。因此, 开发商也无须为不同的终端建立特殊的应用, 制造商也只需要简单地使它们的操作平台可以支持 JME 便可。

### 3: Java 企业版: JEE (Java Enterprise Edition)

JEE(Java Enterprise Edition)是一种利用 Java 平台来简化企业解决方案的开发、部署和管理相关的复杂问题的体系结构。JEE 技术的基础就是核心 Java 平台或 Java 平台的标准版, JEE 不仅巩固了标准版中的许多优点, 例如“编写一次、随处运行”的特性、方便存取数据库的 JDBC API、CORBA 技术以及能够在 Internet 应用中保护数据的安全模式等等, 同时还提供了对 EJB(Enterprise Java Beans)、Java Servlets API、JSP(Java Server Pages)以及 XML 技术的全面支持。其最终目的就是成为一个能够使企业开发者大幅缩短投放市场时间的体系结构。

JEE 体系结构提供中间层集成框架来满足无需太多费用而又需要高可用性、高可靠性以及可扩展性的应用的需求。通过提供统一的开发平台, JEE 降低了开发多层应用的费用和复杂性, 同时提供对现有应用程序集成强有力支持, 完全支持 Enterprise Java Beans, 有良好的向导支持打包和部署应用, 添加了目录支持, 增强了安全机制, 提高了性能。

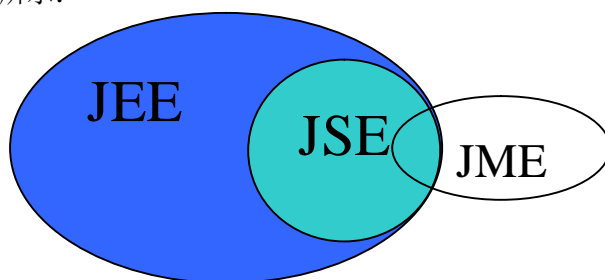
JEE 是对标准版进行功能扩展, 提供一系列功能, 用来解决进行企业应用开发中所面临的复杂的问题。具体的我们会放到后面 JEE 的课程去讲。

### 4: 三个版本之间的关系

JEE 几乎完全包含 JSE 的功能, 然后在 JSE 的基础上添加了很多新的功能。

JME 主要是 JSE 的功能子集, 然后再加上一部分额外添加的功能。

如下图所示:



Java 的 API 类库之中有一组所谓的核心类(CoreClass, 即 `java.*`), 在核心类之外还有所谓的扩充类(Extended Class, 即 `javax.*`)。根据对这两种类的支持程度, 进而区分出几种不同的 Java 版本。

我们必须以 Java Standard Edition(JSE)作为基准, 这个版本做了所有 Java 标准规格之中所定义的核心类, 也支持所有的 Java 基本类。JSE 定位在客户端程序的应用上。

从 JSE 往外延伸, 其外面为 Java Enterprise Edition(JEE), 此版本除了支持所有的标准核心类外, 而且还增加了许多支持企业内部使用的扩充类, 支持 Servlet / JSP 的 `javax.servlet.*` 类、支持 Enterprise Java Bean 的 `javax.ejb.*` 类。当然, JEE 必定支持所有的 Java 基本类。JEE 定位在服务器端(server-side)程序的应用上。

从 JSE 向内看, 是 Java Micro Edition(JME), 它所支持的只有核心类的子集合, 在 JME CLDC 的规格之中, 只支持 `java.lang.*`、`java.io.*`、以及 `java.util.*` 这些类。此版本也增加了一些支持“微小装置”的扩充类, 如 `javax.microedition.io.*` 类。然而, 此版本并不支持所有的 Java 基本类, 就标准的 JMECLDC, 也就是能在 PalmOS 上执行的 KVM(KVirtualMachine)来说, 它就不支持属于浮点数(float、double)的 Java 基本类。JME 定位在嵌入式系统的应用上。

最里层, 还有一个 Java 的 Smart Card 版本, 原本在 Java 的文件之中并没有这样定义, 但是将它画在 JME 内部是很合理的。因为 SmartCard 版本只支持 `java.lang.*` 这个核心类, 比起 JME 所支持的核心类更少, 但它也有属于自己的扩充类, 如 `javacard.*`、`javacardx.*` 这些类。SmartCard 版本只支持 Boolean 与 Byte 这两种 Java 基本类, 此版本定位在 SmartCard 的应用上。

## 四：闲话 Java

### 1: Java 历史

在上世纪 90 年代初, sun 公司有一个叫做 Green 的项目, 目的是为家用消费电子产品开发一个分布式代码系统, 这样就可以对家用电器进行控制, 和它们进行信息交流。詹姆斯·高斯林 (James Gosling) 等人基于 C++ 开发一种新的语言 Oak (Java 的前身)。Oak 是一种用于网络的精巧而安全的语言。Sun 公司曾依此投标一个交互式电视项目, 但结果是被 SGI 打败, Sun 打算抛弃 Oak。随着互联网的发展, Sun 看到了 Oak 在计算机网络上的广阔应用前景, 于是改造 Oak, 在 1995 年 5 月以“Java”的名称正式发布, 从此 Java 走上繁荣之路。

当然提到 Java 历史, 不得不提的一个故事就是 Java 的命名。开始“Oak”的命名是以项目小组办公室外的树而得名, 但是 Oak 商标被其他公司注册了, 必须另外取一个名字, 传说有一天, 几位 Java 成员组的会员正在讨论给这个新的语言取什么名字, 当时他们正在咖啡馆喝着 Java (爪哇) 咖啡, 有一个人灵机一动说就叫 Java 怎样, 得到了其他人的赞同, 于是, Java 这个名字就这样传开了。当然对于传说, 了解一下就好了, 不必过于认真。

### 2: Java 大事记

作为学习 Java 的人士，对 Java 历史上发生的大事件有一个了解是应该的。

**JDK (Java Software Develop Kit):** Java 软件开发工具包。JDK 是 Java 的核心，包括了 Java 运行环境，一系列 Java 开发工具和 Java 基础的类库。目前主流的 JDK 是 Sun 公司发布的 JDK，除了 Sun 之外，还有很多公司和组织都开发了自己的 JDK，例如 IBM 公司开发的 JDK，BEA 公司的 Jrocket，还有 GNU 组织开发的 JDK 等等。

时间	事件
1995 年 5 月 23 日	Java 语言诞生
1996 年 1 月	第一个 JDK—JDK1.0 诞生
1997 年 2 月 18 日	JDK1.1 发布
1998 年 12 月 8 日	Java2 企业平台 J2EE 发布
1999 年 6 月	Sun 发布 Java 三个版本：标准版 J2SE，企业版 J2EE，微型版 J2ME
2004 年 9 月 30 日	JavaSE5.0 发布
2006 年 12 月	JavaSE6.0 发布

### 3: Java 特点

简单地说，Java 具有如下特点：简单的、面向对象、平台无关、多线程、分布式、安全、高性能、可靠的、解释型、自动垃圾回收等特点。

这里只解释一下平台无关和分布式，其余的在后面会逐步接触到。

#### 3.1: 平台无关

所谓平台无关指的是：用 Java 写的程序不用修改就可可在不同的软硬件平台上运行。这样就能实现同样的程序既可以在 Windows 下运行，到了 Unix 或者 Linux 环境不用修改就直接可以运行了。Java 主要靠 Java 虚拟机 (JVM) 实现平台无关性。

平台无关性就是一次编写，到处运行：**Write Once, Run Anywhere**

#### 3.2: 分布式

分布式指的是：软件由很多个可以独立执行的模块组成，这些模块被分布在多台计算机上，可以同时运行，对外看起来还是一个整体。也就是说，分布式能够把多台计算机集合起来就像一台计算机一样，从而提供更好的性能。

### 4: Java 标准组织——JCP

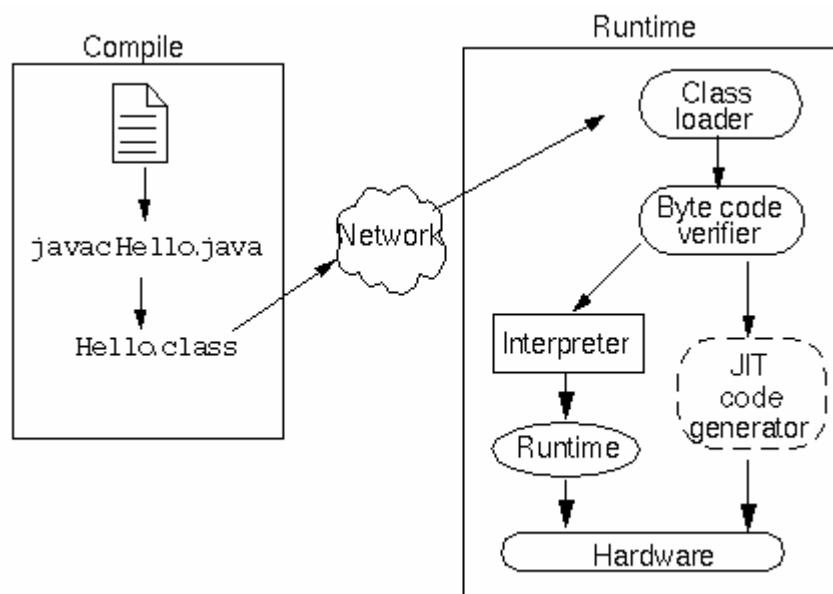
JCP (Java Community Process) 是一个开放的国际组织，成立于 1995 年，主要职能是发展和更新 Java 技术规范、参考实现 (RI)、技术兼容包 (TCK)。Java 技术和 JCP 两者的原创者都是 SUN 计算机公司。组织成员可以提交 JSR (Java Specification Requests)，通过讨论、认可、审核以后，将进入到下一版本的规范里面。

也就是说 JCP 是目前 Java 技术发展事实上的控制者和领导者。

## 五: Java 如何做到让机器理解我们想要做的东西

用一个图来描述这个过程会比较容易理解：





## 1: 编写代码

首先把我们想要计算机做的事情，通过 Java 表达出来，写成 Java 文件，这个过程就是编写代码的过程。如上图所示的 Hello.java 文件。

## 2: 编译

写完 Java 代码后，机器并不认识我们写的 Java 代码，需要进行编译成为字节码，编译后的文件叫做 class 文件。如上图所示的 Hello.class 文件。

## 3: 类装载 ClassLoader

类装载的功能是为执行程序寻找和装载所需要的类。

ClassLoader 能够加强代码的安全性，主要方式是：把本机上的类和网络资源类相分离，在调入类的时候进行检查，因而可以限制任何“特洛伊木马”的应用。

## 4: 字节码 (byte-code) 校验

功能是对 class 文件的代码进行校验，保证代码的安全性。

Java 软件代码在实际运行之前要经过几次测试。JVM 将代码输入一个字节码校验器以测试代码段格式并进行规则检查——检查伪造指针、违反对象访问权限或试图改变对象类型的非法代码。

注意——所有源于网络的类文件都要经过字节码校验器

字节码校验器对程序代码进行四遍校验，这可以保证代码符合 JVM 规范并且不破坏系统的完整性。如果校验器在完成四遍校验后未返回出错信息，则下列各点可被保证：

- 类符合 JVM 规范的类文件格式
- 无访问限制异常
- 代码未引起操作数栈上溢或下溢
- 所有操作代码的参数类型将总是正确的
- 无非法数据转换发生，如将整数转换为对象引用
- 对象域访问是合法的

## 5: 解释 (Interpreter)

可是机器也不能认识 class 文件，还需要被解释器进行解释，机器才能最终理解我们所要表达的东西。

## 6: 运行

最后由运行环境中的 Runtime 对代码进行运行，真正实现我们想要机器完成的工作。

## 7: 说明

由上面的讲述，大家看到，Java 通过一个编译阶段和一个运行阶段，来让机器最终理解我们想要它完成的工作，并按照我们的要求进行运行。

在这两个阶段中，需要我们去完成的就是编译阶段的工作，也就是说：我们需要把我们想要机器完成的工作用 Java 语言表达出来，写成 Java 源文件，然后把源文件进行编译，形成 class 文件，最后就可以在 Java 运行环境中运行了。运行阶段的工作由 Java 平台自身提供，我们不需要做什么工作。

# 六：Java 技术三大特性

---

## 1: 虚拟机

---

Java 虚拟机 JVM (Java Virtual Machine) 在 Java 编程里面具有非常重要的地位，约相当于前面学到的 Java 运行环境，虚拟机的基本功能如下：

- (1): 通过 ClassLoader 寻找和装载 class 文件
- (2): 解释字节码成为指令并执行，提供 class 文件的运行环境
- (3): 进行运行期间垃圾回收
- (4): 提供与硬件交互的平台

Java虚拟机是在真实机器中用软件模拟实现的一种想象机器。Java虚拟机代码被存储在 .class文件中；每个文件都包含最多一个public类。Java 虚拟机规范为不同的硬件平台提供了一种编译Java技术代码的规范，该规范使Java 软件独立于平台，因为编译是针对作为虚拟机的“一般机器”而做。这个“一般机器”可用软件模拟并运行于各种现存的计算机系统，也可用硬件来实现。编译器在获取Java应用程序的源代码后，将其生成字节码，它是为JVM生成的一种机器码指令。每个Java解释器，不管它是Java技术开发工具，还是可运行 applets 的Web浏览器，都可执行JVM。

JVM 为下列各项做出了定义

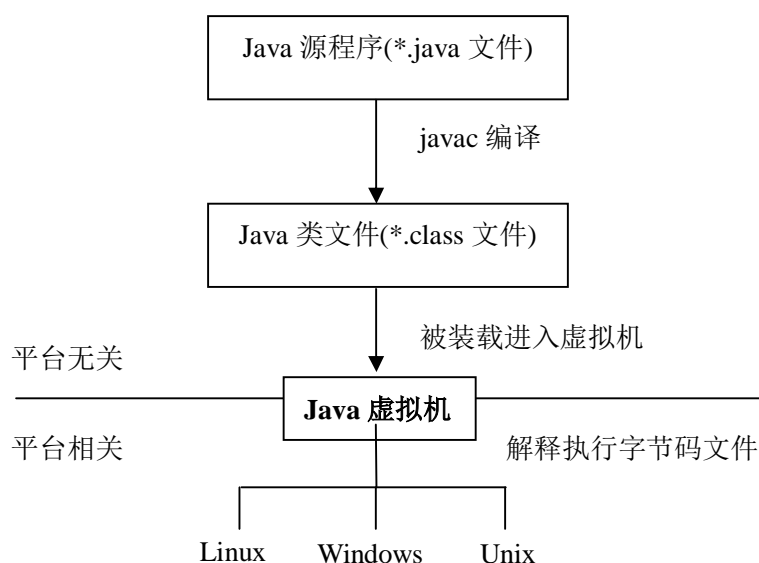
- 指令集（相当于中央处理器 [CPU] ）
- 寄存器
- 类文件格式
- 栈
- 垃圾收集堆
- 存储区

JVM 的代码格式由紧缩有效的字节码构成。由 JVM 字节码编写的程序必须保持适当的类型约束。大部分类型检查是在编译时完成。任何从属的 Java 技术解释器必须能够运行任何含有类文件的程序，这些类文件应符合 Java 虚拟机规范中所指定的类文件格式。

### 1.1: 虚拟机是 Java 平台无关的保障

正是因为有虚拟机这个中间层，Java 才能够实现与平台无关。虚拟机就好比是一个 Java 运行的基本平台，所有的 Java 程序都运行在虚拟机上，如下图所示：





## 2: 垃圾回收

### 2.1: 什么是垃圾

在程序运行的过程中, 存在被分配了的内存块不再被需要的情况, 那么这些内存块对程序来讲就是垃圾。

产生了垃圾, 自然就需要清理这些垃圾, 更为重要的是需要把这些垃圾所占用的内存资源, 回收回来, 加以再利用, 从而节省资源, 提高系统性能。

### 2.2: 垃圾回收

- 不再需要的已分配内存应取消分配(释放内存)
- 在其它语言中, 取消分配是程序员的责任
- Java 编程语言提供了一种系统级线程以跟踪内存分配
- 垃圾收集
  - 可检查和释放不再需要的内存
  - 可自动完成上述工作
  - 可在 JVM 实现周期中, 产生意想不到的变化

许多编程语言都允许在程序运行时动态分配内存, 分配内存的过程由于语言句法不同而有所变化, 但总是要将指针返回到内存的起始位置, 当分配内存不再需要时(内存指针已溢出范围), 程序或运行环境应释放内存。

在 C, C++ 或其它语言中, 程序员负责释放内存。有时, 这是一件很困难的事情。因为你并不总是事先知道内存应在何时被释放。当在系统中没有能够被分配的内存时, 可导致程序瘫痪, 这种程序被称作具有内存漏洞。

Java 编程语言解除了程序员释放内存的责任。它可提供一种系统级线程以跟踪每一次内存的分配情况。在 Java 虚拟机的空闲周期, 垃圾收集线程检查并释放那些可被释放的内存。垃圾收集在 Java 技术程序的生命周期中自动进行, 它解除了释放内存的要求, 这样能够有效避免内存漏洞和内存泄露(内存泄露就是程序运行期间, 所占用的内存一直往上涨, 很容易造成系统资源耗尽而降低性能或崩溃)。

### 2.3: 提示

(1): 在 Java 里面, 垃圾回收是一个自动的系统行为, 程序员不能控制垃圾回收的功能和行为。比如垃圾回收什么时候开始, 什么时候结束, 还有到底哪些资源需要回收等, 都是程序员不能控制的。

(2): 有一些跟垃圾回收相关的方法, 比如: `System.gc()`, 记住一点, 调用这些方法, 仅仅是在通知垃圾回收程序, 至于垃圾回收程序运不运行, 什么时候运行, 都是无法控制的。

(3): 程序员可以通过设置对象为 `null` (后面会讲到) 来标示某个对象不再被需要了, 这只是表示这个对象可以被回收了, 并不是马上被回收。

### 3: 代码安全

Java 如何保证编写的代码是安全可靠的呢?

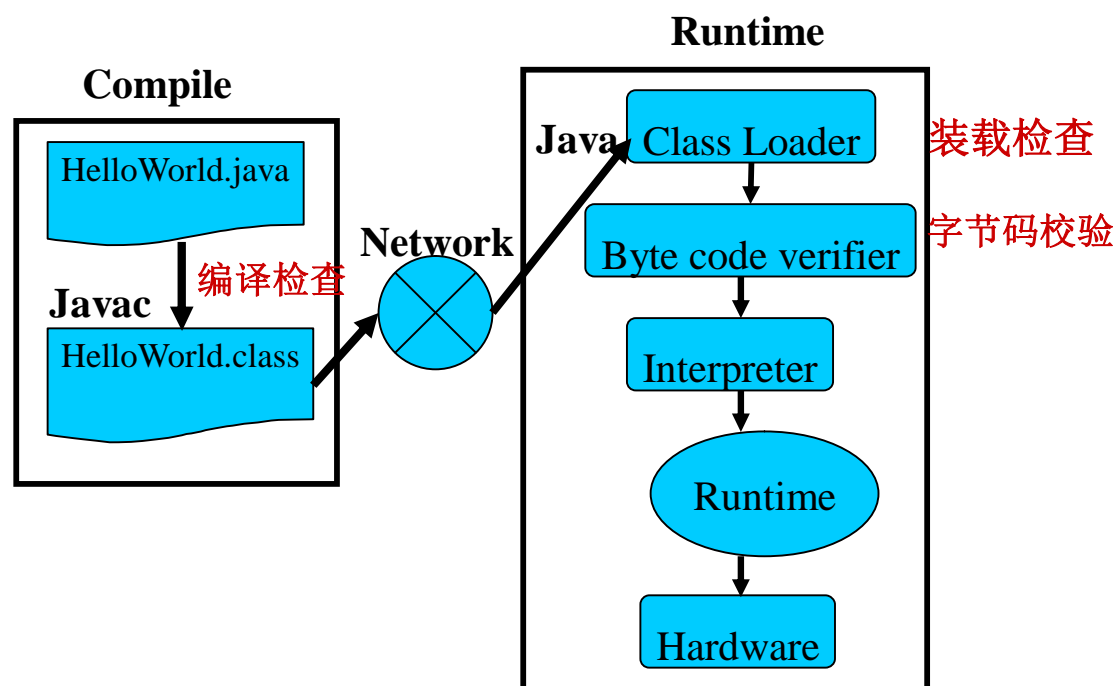
(1): 第一关: 编写的代码首先要被编译成为 `class` 文件, 如果代码写得有问题, 编译期间就会发现, 然后提示有编译错误, 无法编译通过。

(2): 第二关: 通过编译关后, 在类装载的时候, 还会进行类装载检查, 把本机上的类和网络资源类相分离, 在调入类的时候进行检查, 因而可以限制任何“特洛伊木马”的应用。

(3): 第三关: 类装载后, 在运行前, 还会进行字节码校验, 以判断你的程序是安全的。

(4): 第四关: 如果你的程序在网络上运行, 还有沙箱 (Sand Box) 的保护, 什么是沙箱呢? 就是如果你的程序没有获得授权, 只能在沙箱限定的范围内运行, 是不能够访问本地资源的, 从而保证安全性。

如下图所示:



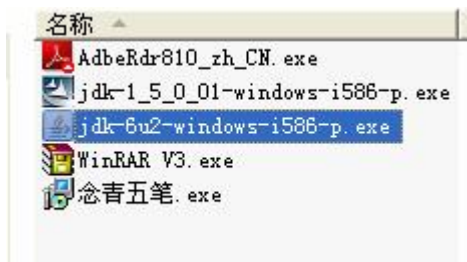
学习到这里, 大家应该对 Java 有了一定的了解了。现在是否想要看看 Java 程序究竟什么样子呢? 是不是想要体会一下如何开发 Java 程序呢? 下面我们先来看看如何构建 JSE 的环境, 这是进行 Java 程序开发的第一步。

## 七: 构建 JSE 开发环境

学习 Java 开发的第一步就是构建开发环境, 下面以 JDK6.0 在 Windows XP 上的安装配置为例来讲述:

### 第一步: 下载 JDK

从 SUN 网站下载 JDK6 或以上版本, 这里以 jdk-6u2-windows-i586-p 版为例, 如下图:



### 第二步: 安装 JDK

(1): 双击 jdk-6u2-windows-i586-p.exe 文件, 出现安装界面如下图:



(2): 然后出现下面的界面



(3): 点击“接受”按钮, 然后出现下列界面



(4)：点击界面上的“更改”按钮，出现如下界面：



(5)：在这个界面设置需要安装到的路径，可以安装到任何你想要安装的路径，如下图：

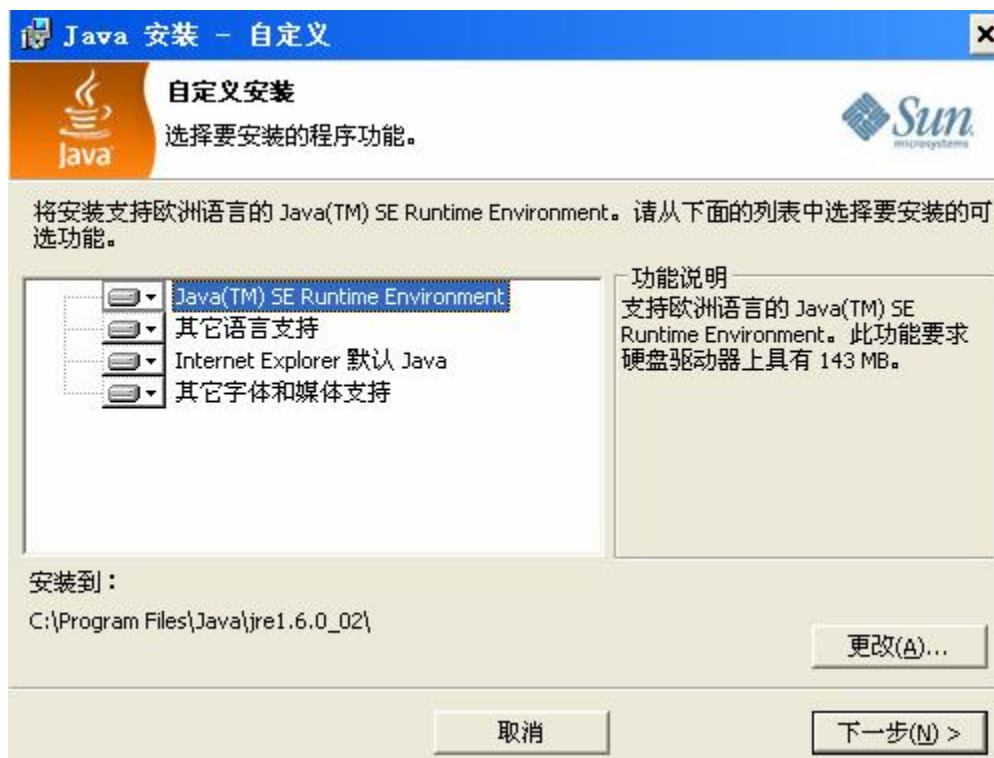


(6): 然后点击确定, 返回到上一个界面, 如下图所示:

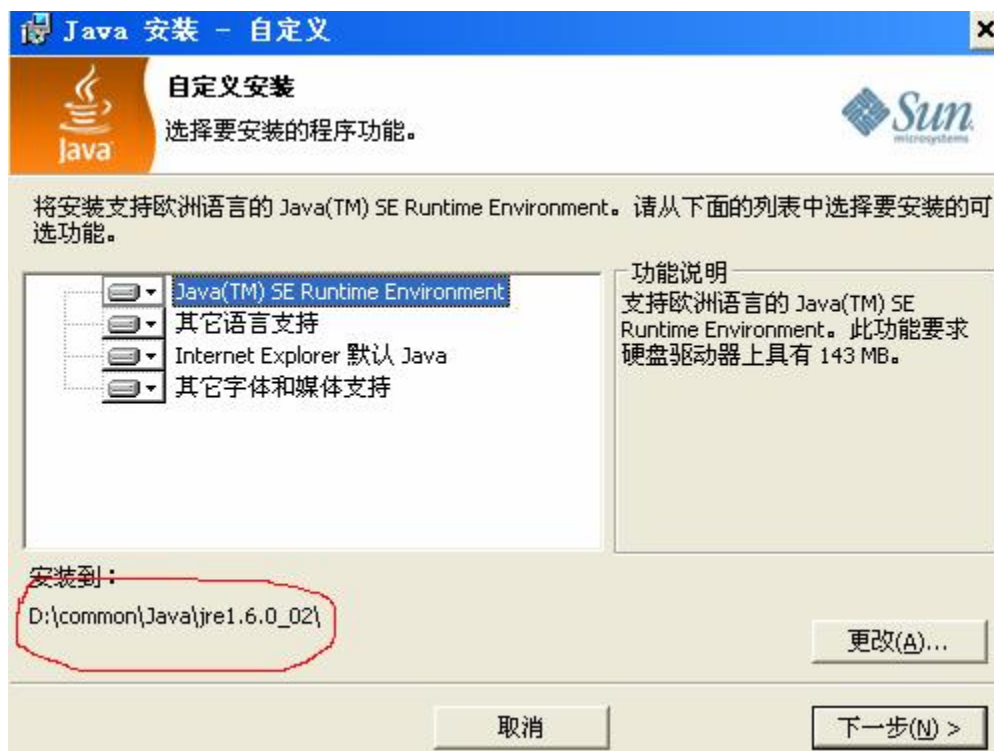


(7): 然后点击下一步, 会进行 JDK 的安装, 请耐心等待, 直到出现 JRE 的安装, 如下图:





(8): 选择更改, 然后在弹出的界面进行安装路径的设置, 跟前面的方式一样, 然后点击确定回来, 如下图所示:



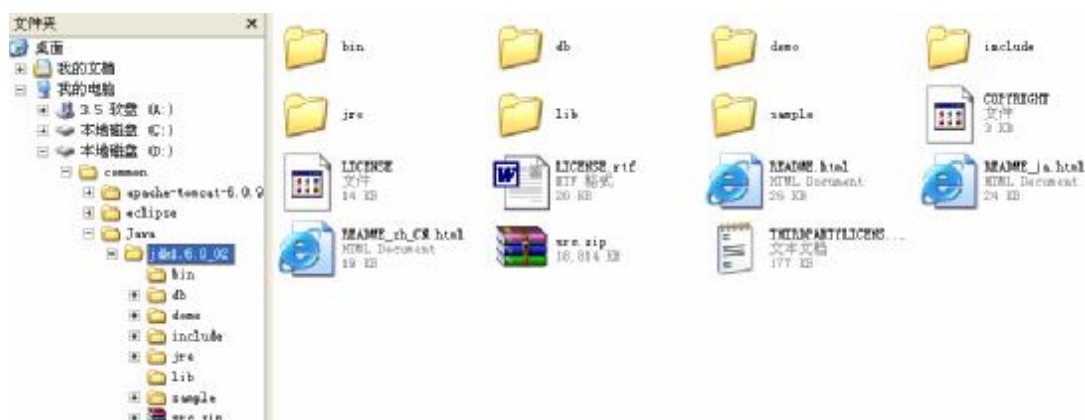
(9): 然后点击下一步, 直到出现下面的界面, 表示安装完成。





(10): 自述文件可看可不看，不想看，把前面的勾点掉即可，然后点击完成按钮。

(11): 安装完成过后，JDK 文件夹如下图：



D:\common\Java\jdk1.6.0\_02: 是 JDK 的安装路径

bin: binary 的简写，下面存放的是 Java 的各种可执行文件

db: JDK6 新加入的 Apache 的 Derby 数据库，支持 JDBC4.0 的规范。

include: 需要引入的一些头文件，主要是 c 和 c++ 的，JDK 本身是通过 C 和 C++ 实现的。

jre: Java 运行环境。

lib: library 的简写，JDK 所需要的一些资源文件和资源包。

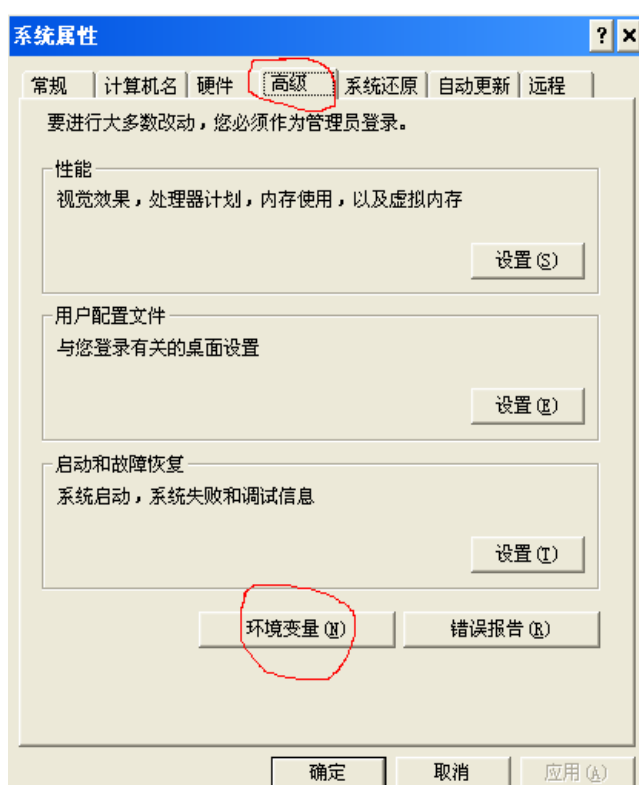
### 第三步：配置环境变量

安装完成后，还要进行 Java 环境的配置，才能正常使用，步骤如下：

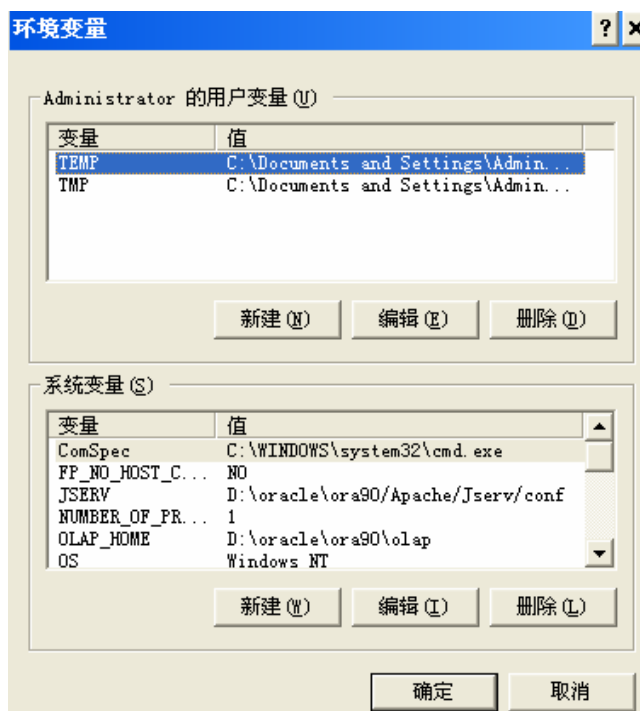
(1)：在我的电脑点击右键——>选择属性，如下图所示：



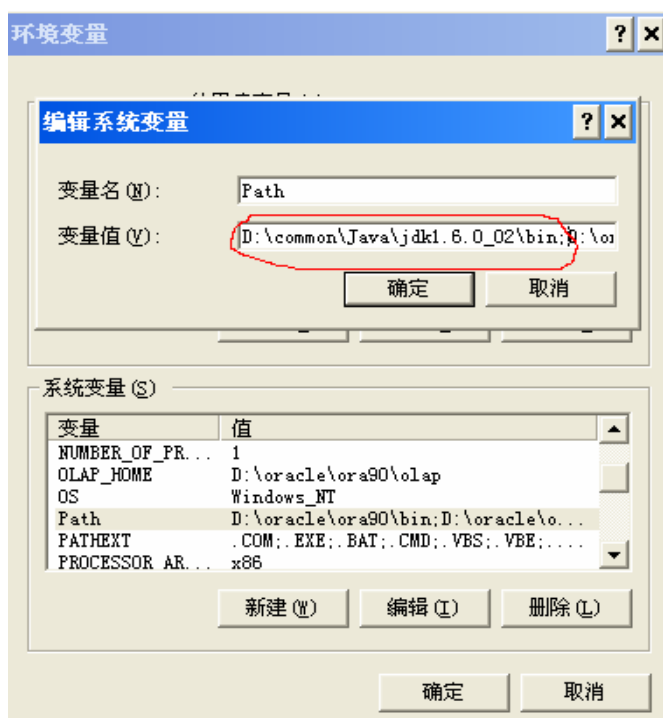
(2)：在弹出界面上：选择高级——>环境变量，如下图所示：



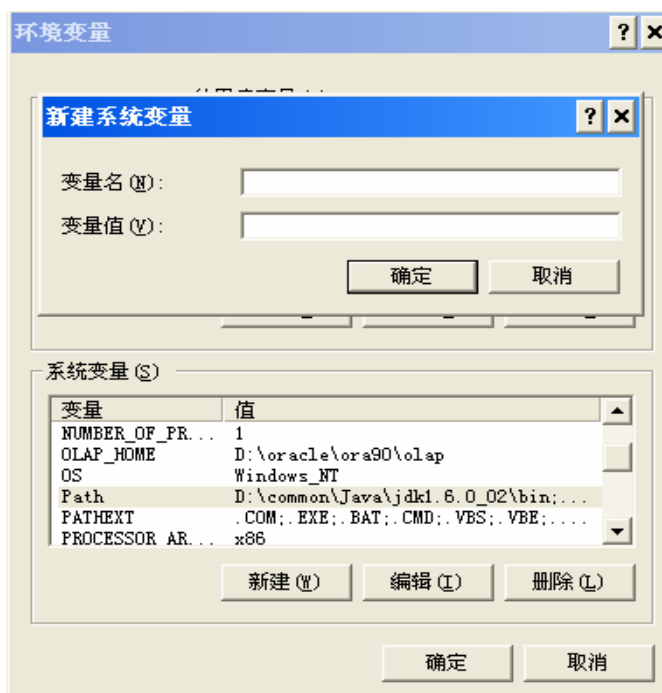
(3)：弹出如下界面，我们的设置就在这个界面上：



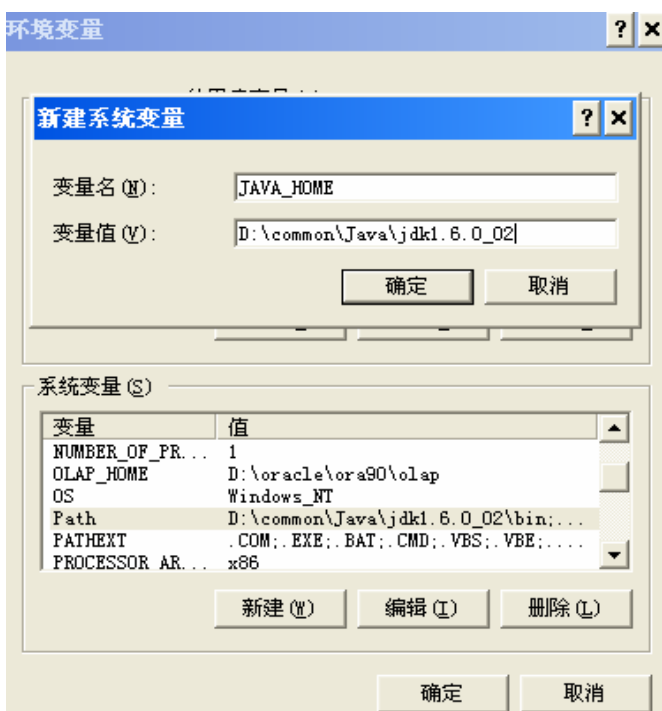
(4): 在系统变量里面找到“Path”这一项, 然后双击它, 在弹出的界面上, 在变量值开头添加如下语句“D:\common\Java\jdk1.6.0\_02\bin;”, 注意不要忘了后面的分号, 如下图所示:



(5): 然后点击编辑系统变量界面的确定按钮, 然后点击环境变量界面的“新建”, 弹出界面如下图:



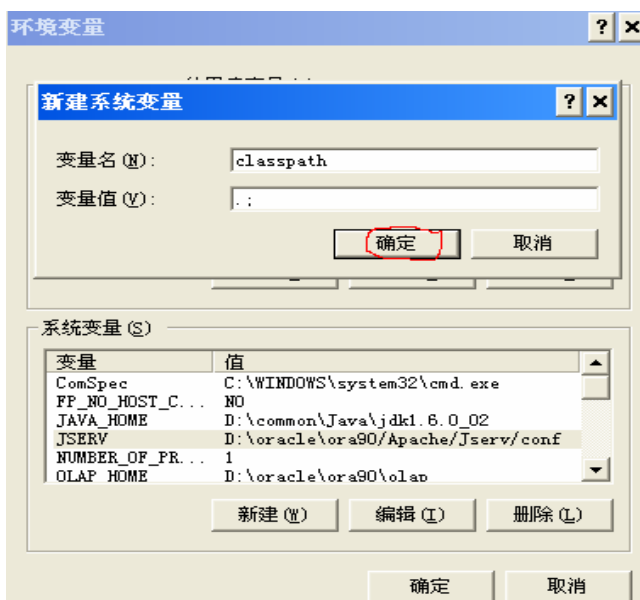
(6): 在上面填写变量名为: JAVA\_HOME , 变量值为: D:\common\Java\jdk1.6.0\_02 , 如下图所示:



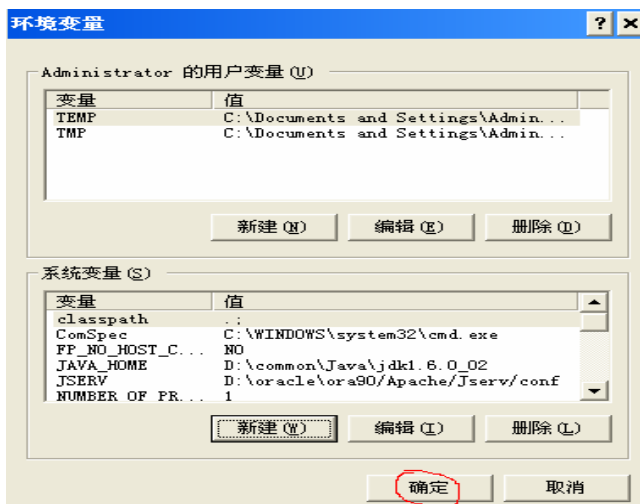
(7): 然后点击新建系统变量界面的确定按钮, 然后点击环境变量界面的“新建”, 弹出新建系统变量界面, 在上面填写变量名为: classpath , 变量值为: . ; , 注意是点和分号, 如下图所示:



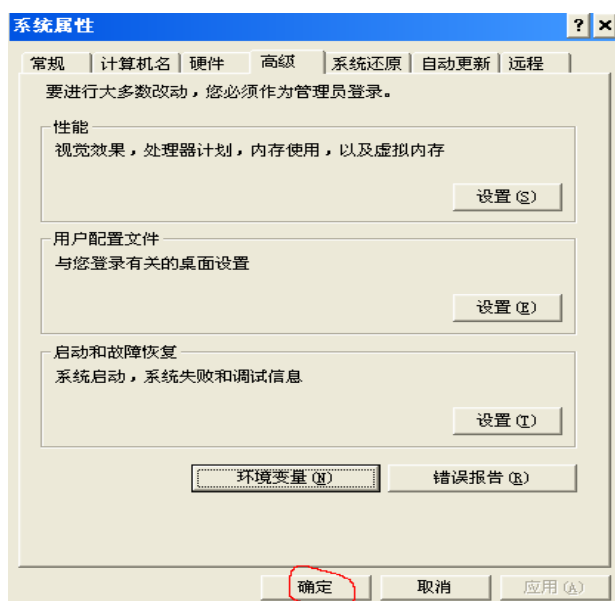
(8): 然后点击新建系统变量界面的确定按钮



(9): 然后点击环境变量界面的确定按钮



(10): 点击系统属性界面的确定按钮



到此设置就完成了。

(11): 那么为何要设置这些环境变量呢，如何设置呢：

#### **PATH:**

提供给操作系统寻找到 Java 命令工具的路径。通常是配置到 JDK 安装路径\bin

#### **JAVA\_HOME:**

提供给其它基于 Java 的程序使用，让它们能够找到 JDK 的位置。通常配置到 JDK 安装路径。注意：这个必须书写正确，全部大写，中间用下划线。

#### **CLASSPATH:**

提供程序在运行期寻找所需资源的路径，比如：类、文件、图片等等。

注意：在 windows 操作系统上，最好在 classpath 的配置里面，始终在前面保持“.”的配置，在 windows 里面“.”表示当前路径。

#### **第四步：检测安装配置是否成功**

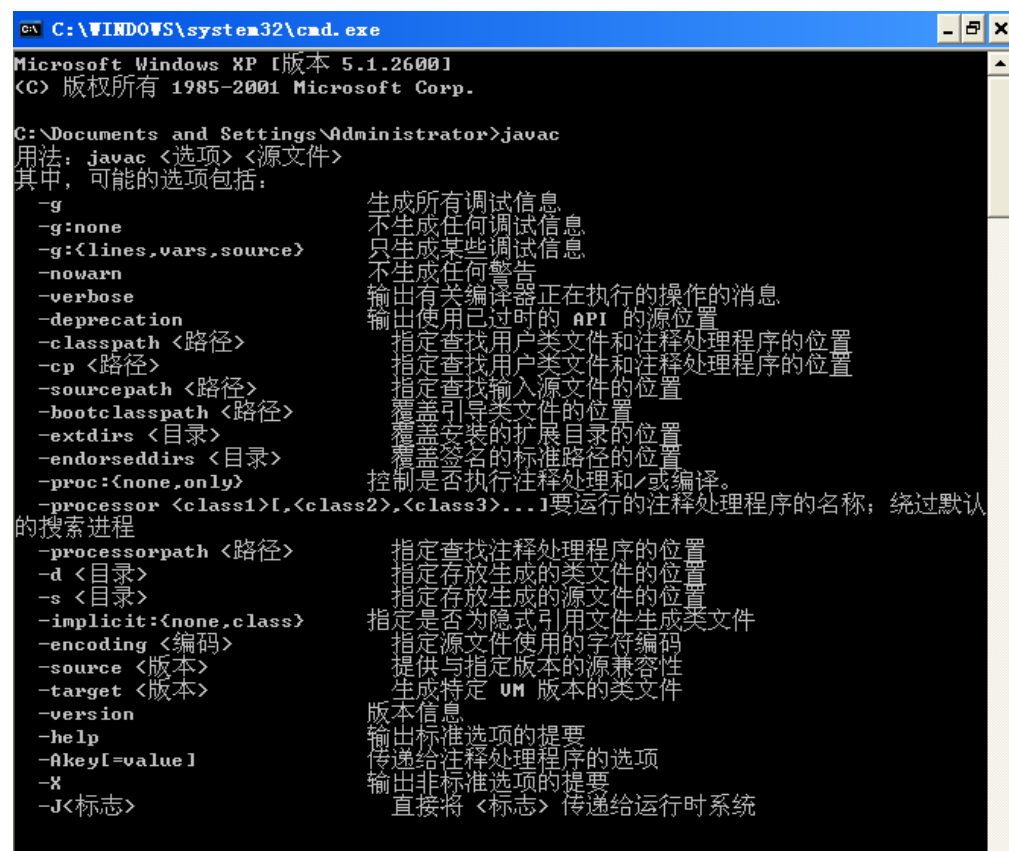
进行完上面的步骤，基本的安装和配置就好了，怎么知道安装成功没有呢？

(1): 点击开始——> 点击运行，在弹出的对话框中输入“cmd”，如下图示：





(2) 然后点击确定, 在弹出的 dos 窗口里面, 输入 “javac”, 然后回车, 出现如下界面则表示安装配置成功。



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>javac
用法: javac <选项> <源文件>
其中, 可能的选项包括:
-g 生成所有调试信息
-g:none 不生成任何调试信息
-g:{lines,vars,source} 只生成某些调试信息
-nowarn 不生成任何警告
-verbose 输出有关编译器正在执行的操作的消息
-deprecation 输出使用已过时的 API 的源位置
-classpath <路径> 指定查找用户类文件和注释处理程序的位置
-cp <路径> 指定查找用户类文件和注释处理程序的位置
-sourcepath <路径> 指定查找输入源文件的位置
-bootclasspath <路径> 覆盖引导类文件的位置
-extdirs <目录> 覆盖安装的扩展目录的位置
-endorseddirs <目录> 覆盖签名的标准路径的位置
-processor <none,only> 控制是否执行注释处理和/或编译。
-processor <class1>[,<class2>,<class3>... ] 要运行的注释处理程序的名称; 绕过默认
的搜索进程
-processorpath <路径> 指定查找注释处理程序的位置
-d <目录> 指定存放生成的类文件的位置
-s <目录> 指定存放生成的源文件的位置
-implicit:{none,class} 指定是否为隐式引用文件生成类文件
-encoding <编码> 指定源文件使用的字符编码
-source <版本> 提供与指定版本的源兼容性
-target <版本> 生成特定 VM 版本的类文件
-version 版本信息
-help 输出标准选项的提要
-Akey=value 传递给注释处理程序的选项
-X 输出非标准选项的提要
-J<标志> 直接将 <标志> 传递给运行时系统
```

好了, 现在 Java 的开发环境就配置好了, 有些人已经跃跃欲试的想要马上开始学习如何编写 Java 程序了, 下面先来体会第一个 Java 程序。

## 八: 初识 Java 程序——HelloWorld

象其它编程语言一样, Java 编程语言也被用来创建应用程序。一个共同的应用程序范例是在屏幕上显示字符串 “Hello World! ”。下列代码给出了这个 Java 应用程序。

虽然很多你可能都不明白, 没有关系, 主要是来体会一下 Java 程序是什么样子, 你可以先看看, 有个印象, 然后可以先模仿着做。

### 1: HelloWorldApp

```
1. //
2. // HelloWorld 应用示例
3. //
4. public class HelloWorldApp{
5.     public static void main (String args[]) {
6.         System.out.println ("Hello World!");
7.     }
8. }
```

以上程序行是在你的屏幕上打印 “Hello World!” 所需的最少代码。

## 2: 描述 HelloWorldApp

---

### 第 1-3 行

程序中的 1-3 行是注释行

```
1 //  
2 // HelloWorld 应用示例  
3 //
```

### 第 4 行

第 4 行声明类名为 HelloWorldApp。类名 (Classname) 是在源文件中指明的，它可在与源代码相同的目录上创建一个 `classname.class` 文件。在本例题中，编译器创建了一个称为 HelloWorldApp.class 的文件，它包含了公共类 HelloWorldApp 的编译代码。

```
4 public class HelloWorldApp {
```

### 第 5 行

第 5 行是程序执行的起始点。Java 解释器必须发现这一严格定义的点，否则将拒绝运行程序。

其它程序语言 (特别是 C 和 C++) 也采用 `main()` 声明作为程序执行的起始点。此声明的不同部分将在本课程的后几部分介绍。

如果在程序的命令行中给出了任何自变量 (命令行参数)，它们将被传递给 `main()` 方法中被称作 `args` 的 `String` 数组。在本例题中，未使用自变量。

```
    public static void main (String args[]) {  
        public—方法 main() 可被任何程序访问，包括 Java 解释器。
```

`static` — 是一个告知编译器 `main()` 是用于类 HelloWorldApp 中的方法的关键字。为使 `main()` 在程序做其它事之前就开始运行，这一关键字是必要的。

`void` — 表明 `main()` 不返回任何信息。这一点是重要的，因为 Java 编程语言要进行谨慎的类型检查，包括检查调用的方法确实返回了这些方法所声明的类型。

`String args[]` — 是一个 `String` 数组的声明，它将包含位于类名之后的命令行中的自变量。

```
        java HelloWorldApp args[0] args[1] ....
```

### 第 6 行

第 6 行声明如何使用类名、对象名和方法调用。它使用由 `System` 类的 `out` 成员引用的 `PrintStream` 对象的 `println()` 方法，将字符串 “Hello World!” 打印到标准输出上。

```
        6    System.out.println(“Hello World!”);
```

在这个例子中，`println()` 方法被输入了一个字符串自变量并将其写在了标准输出流上。

### 第 7-8 行

本程序的 7-8 行分别是方法 `main()` 和类 HelloWorldApp 的下括号。

```
        7        }  
        8    }
```

## 3: 编译并运行 HelloWorldApp

---

### 编译

当你创建了 HelloWorldApp.java 源文件后，用下列程序行进行编译：

```
javac HelloWorldApp.java
```

如果编译器未返回任何提示信息，新文件 HelloWorldApp.class 则被存储在与源文件相

同的目录中, 除非另有指定。

如果在编译中遇到问题, 请参阅本模块的查错提示信息部分。

## 运行

为运行你的 HelloWorldApp 应用程序, 需使用 Java 解释器和位于 bin 目录下的 java 程序:

```
java HelloWorldApp
Hello World!
```

## 4: 编译差错

---

编译时的错误, 以下是编译时的常见错误:

```
javac:Command not found
PATH 变量未正确设置以包括 javac 编译器。javac 编译器位于 JDK 目录下的 bin 目录。
```

```
HelloWorldApp.java:6: Method println(java.lang.String)
not found in class java.io.PrintStream.System.
out.println("Hello World!");
方法名 println 出现打印错误。
```

```
In class HelloWorldApp:main must be public or static
```

该错误的出现是因为词 static 或 public 被放在了包含 main 方法的行之外。

运行时的错误 can't find class HelloWorldApp (这个错误是在打印 java HelloWorldApp 时产生的), 通常, 它表示在命令行中所指定的类名的拼写与 filename.class 文件的拼写不同。Java 编程语言是一种大小写区别对待的语言。

例如: public class HelloWorldapp {

创建了一个 HelloWorldapp.class, 它不是编译器所预期的类名 (HelloWorldApp.class)。

### 4.1: 命名

如果 java 文件包括一个公共类, 那么它必须使用与那个公共类相同的文件名。例如在前例中的类的定义是

```
public class HelloWorldapp
源文件名则必须是 HelloWorldapp.java
```

### 4.2: 类计数

在源文件中每次只能定义一个公共类。

### 4.3: 源文件布局

一个 Java 源文件可包含三个“顶级”要素:

- (1) 一个包(package)声明 (可选)
- (2) 任意数量的导入(import)语句
- (3) 类(class)声明

该三要素必须以上述顺序出现。即, 任何导入语句出现在所有类定义之前; 如果使用包声明, 则包声明必须出现在类和导入语句之前。

## 练习实践

---

本章实践重点：主要实践并掌握如下内容：

### I 用 JDK 编译、运行 JAVA 程序。本章用 JDK 作为开发工具

#### 程序 1

---

第一个 Java 程序：输出信息

需求：输出 “I am XXX ， Now at Java 私塾！ Welcome to Javass， Good Luck!”。

目标：

- 1、了解 Java 程序的基本结构；
- 2、屏幕打印方法 System.out.println()。

程序：

```
//: HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("I am XXX, Now at Java 私塾！ Welcome to Javass, Good Luck!");
    }
}
```

说明：

- 1、// HelloWorld.java 这一行是注释，注明此程序的文件名是 HelloWorld.java；
- 2、第二行是类名定义，在 Java 中，类名必须与文件名需要相同，否则编译会出错；
- 3、第三行是主方法定义，主方法是一个程序的执行入口；
- 4、第四行是在屏幕输出，即打印出 “I am XXX ， Now at Java 私塾！ Welcome to Javass， Good Luck!”；
- 5、第五、六行是与前面括号的对应。

## 作业

---

- 1：简述 Java 从代码到运行的全过程
- 2：简述虚拟机的工作机制
- 3：简述 Java 的垃圾回收机制
- 4：简述 Java 的安全机制
- 5：简述 path、classpath、JAVA\_HOME 各自的含义和配置方式

## 第二章 基础语法

### 教学目标：

- 掌握 Java 关键字和标识符
- 掌握 Java 的基本数据类型
- 掌握变量和常量
- 掌握 Java 代码的基本知识
- 掌握 Java 的运算符
- 掌握 Java 表达式
- 掌握 Java 流程控制结构

## 一: 关键字

---

大家回忆一下我们在学习汉语的时候, 开始学的是什么? 肯定是先学一些单个的字, 只有认识了单个的字, 然后才能组成词, 然后才能慢慢的到句子, 然后到文章。

学习同计算机交流跟这个过程是一样的, 首先我们得学习一些计算机看得懂的单个的字, 那么这些单个字在 Java 里面就是关键字。

### 1: 什么是关键字

Java 语言保留的, Java 的开发和运行平台认识, 并能正确处理的一些单词。

其实就是个约定, 就好比 we 约定好, 我画个勾勾表示去吃饭。那好了, 只要我画个勾勾, 大家就知道是什么意思, 并能够正确执行了。

关键字这个约定在 Java 语言和 Java 的开发和运行平台之间, 我们只要按照这个约定使用了某个关键字, Java 的开发和运行平台就能够认识它, 并正确地处理。

### 2: Java 中有哪些关键字

abstract	do	implements	private	throw
boolean	double	import	protected	throws
break	else	instanceof	public	transient
byte	extends	int	return	true
case	false	interface	short	try
catch	final	long	static	void
char	finally	native	super	volatile
class	float	new	switch	while
continue	for	null	synchronized	enum
default	if	package	this	assert

### 3: Java 中关键字的基本含义

- abstract: 表明类或类中的方法是抽象的;
- boolean: 基本数据类型之一, 布尔类型;
- break: 提前跳出一个块;
- byte: 基本数据类型之一, 字节类型;
- case: 在 switch 语句中, 表明其中的一个分支;
- catch: 用于处理例外情况, 用来捕捉异常;
- char: 基本数据类型之一, 字符类型;
- class: 类;
- continue: 回到一个块的开始处;
- default: 用在 switch 语句中, 表明一个默认的分支;
- do: 用在 "do while" 循环结构中;
- double: 基本数据类型之一, 双精度浮点数类型;
- else: 在条件语句中, 表明当条件不成立时的分支;
- extends: 用来表明一个类是另一个类的子类;
- final: 用来表明一个类不能派生出子类, 或类中的方法不能被覆盖, 或声明一个变量是常量;
- finally: 用于处理异常情况, 用来声明一个肯定会被执行到的块;



- float: 基本数据类型之一，单精度浮点数类型；
- for: 一种循环结构的引导词；
- if: 条件语句的引导词；
- implements: 表明一个类实现了给定的接口；
- import: 表明要访问指定的类或包；
- instanceof: 用来测试一个对象是否是一个指定类的实例；
- int: 基本数据类型之一，整数类型；
- interface: 接口；
- long: 基本数据类型之一，长整数类型；
- native: 用来声明一个方法是由与机器相关的语言(如 C/C++/FORTRAN 语言)实现的；
- new: 用来申请新对象；
- package: 包；
- private: 一种访问方式：私有模式；
- protected: 一种访问方式：保护模式；
- public: 一种访问方式：公共模式；
- return: 从方法中返回值；
- short: 基本数据类型之一，短整数类型；
- static: 表明域或方法是静态的，即该域或方法是属于类的；
- strictfp: 用来声明 FP-strict(双精度或单精度浮点数)表达式，参见 IEEE 754 算术规范；
- super: 当前对象的父类对象的引用；
- switch: 分支结构的引导词；
- synchronized: 表明一段代码的执行需要同步；
- this: 当前对象的引用；
- throw: 抛出一个异常；
- throws: 声明方法中抛出的所有异常；
- transient: 声明不用序列化的域；
- try: 尝试一个可能抛出异常的程序块
- void: 表明方法不返回值；
- volatile: 表明两个或多个变量必须同步地发生变化；
- while: 用在循环结构中；
- assert: 声明断言；
- enum: 声明枚举类型；

#### 4: 解释几点

- (1): 这些关键字的具体含义和使用方法，会在后面用到的地方讲述
- (2): Java 的关键字也是随新的版本发布在不断变动中的，不是一成不变的
- (3): 所有关键字都是小写的
- (4): goto 和 const 不是 Java 编程语言中使用的关键字，但是是 Java 的保留字，也就是说 Java 保留了它们，但是没有使用它们。true 和 false 不是关键字，而是 boolean 类型直接量
- (5): 表示类的关键字是 class

## 二: 标识符

现在我们已经知道如何表述一个类了, 那就是“class”这个关键字, 那么属性和方法怎么表达呢? 我们是不是需要对每个属性和方法定义一个名字呢, 比如: 身高、体重等, 这就需要标识符了。

### 1: 什么是标识符

在 Java 编程语言中, 标识符是赋予变量、类或方法的名称。

### 2: 标识符命名规则

命名规则如下:

- (1): 首字母只能以字母、下划线、\$开头,其后可以跟字母‘下划线、\$和数字  
示例: \$abc 、 \_ab 、 ab123 等都是有效的
- (2): 标识符区分大小写 (事实上整个 Java 编程里面都是区分大小写的)  
abc 和 Abc 是两个不同的标识符
- (3): 标识符不能是关键字
- (4): 标识符长度没有限制

### 3: 标识符命名建议

- (1): 如果标识符由多个单词构成, 那么从第二个单词开始, 首字母大写  
示例: isText 、 canRunTheCar 等
- (2): 标识符尽量命名的有意义, 让人能够望文知意
- (3): 尽量少用带\$符号的标识符, 主要是习惯问题, 大家都不是很习惯使用带\$符号的标识符; 还有在某些特定的场合, \$具有特殊的含义
- (4): 由于 Java 语言使用 Unicode 字符集, 所以字母包括:
  - ✓ ‘A’ - ‘Z’ 和 ‘a’ - ‘z’ ;
  - ✓ Unicode 字符集中序号大于 0xC0 的所有符号;
  - ✓ Unicode 字符集支持多种看起来相同的字母;
  - ✓ 建议标识符中最好使用 ASCII 字母
- (5): 标识符不应该使用中文, 虽然中文标识符也能够正常编译和运行, 其原因如上一点讲到的: 是把中文当作 Unicode 字符集中的符号来对待了。

例如如下程序是可以正常编译和运行的, 但是不建议这么做:

```
public class Test {  
    public static void main(String[] args) {  
        String Java私塾 = "中文标识符测试";  
        System.out.println("Java私塾==" + Java私塾);  
    }  
}
```

运行结果: Java 私塾==中文标识符测试

### 4: 示例一

下列哪些是正确的标识符:

```
myVariable
9pins
i
a+c
testing1-2-3
java&uml
My Variable
It's
```

错误的标识符及其原因分析如下：

```
My Variable    //含有空格
9pins          //首字符为数字
a+c            //加号不是字母
testing1-2-3   //减号不是字母
It's           //单引号不是字母
java&uml       //与号不是字母
```

## 5: 示例二

好了，现在来用 Java 代码表示前面抽象出来的人这个类，如下：

```
class Person{
    //姓名
    name;
    //体重
    weight;
    //身高
    height;
}
```

发现新的问题来了，我们定义的这个 `weight` 和 `height` 应该是有单位的，那么在 Java 中怎么表达给 `weight` 和 `height` 设置单位呢？这就需要下面讲的数据类型了。

## 三：数据类型

---

### 1: 什么叫数据类型

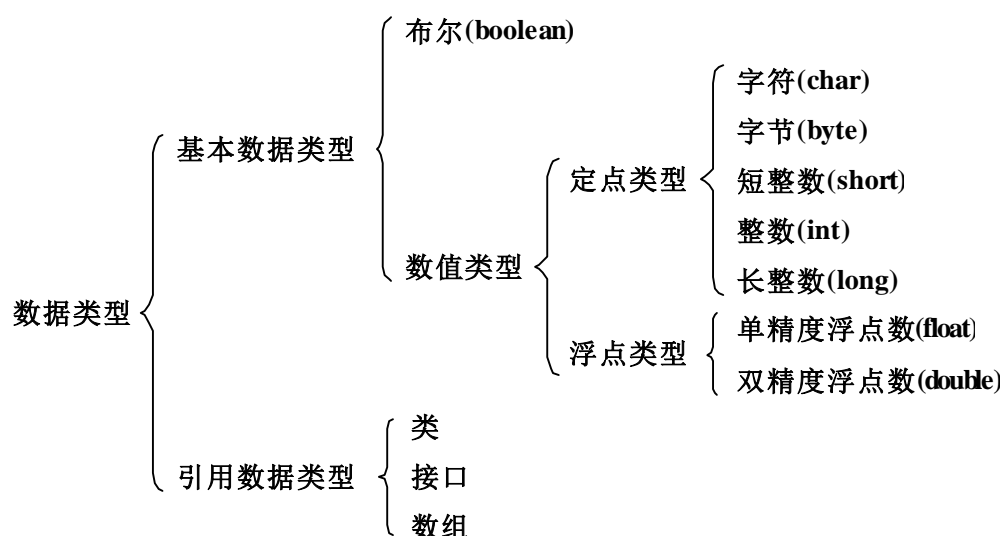
数据类型简单的说就是对数据的分类，对数据各自的特点进行类别的划分，划分的每种数据类型都具有区别于其它类型的特征，每一类数据都有相应的特点和操作功能。例如数字类型的就能够进行加减乘除的操作。

在现实生活中，我们通常会对信息进行分类，从而使得我们能很容易的判断某个数据是表示一个百分数还是一个日期，我们通常是通过判断数字是否带“%”，或者是否是一个我们熟悉的“日期格式”。

类似的在程序中，计算机也需要某种方式来判断某个数字是什么类型的。这通常是需要程序员显示来声明某个数据是什么类型的，Java 就是这样的。Java 是一种强类型的语言，凡是使用到的变量，在编译之前一定要被显示的声明。

## 2: Java 数据类型的分类

Java 里面的数据类型从大的方面分为两类，一是基本数据类型，一是引用类型，基本的 Java 数据类型层次图如下：



Java 数据类型层次图。

## 3: Java 中的基本数据类型

---

Java 中的基本数据类型分为八个原始数据类型，原始类型可分为四种：

- (1): 整数型: byte、short、int、long
- (2): 浮点型: float、double
- (3): 字符型: char
- (4): 逻辑型 : boolean

### 3.1: 整数型: byte、short、int、long

---

byte: 字节型  
short: 短整型  
int: 整型  
long: 长整型

在 Java 中，整数型的值都是带符号的数字，可以用十进制、八进制和十六进制来表示。所谓多少进制，就是满多少就进位的意思，如十进制表示逢十进位，八进制就表示逢八进位。示例：

15 : 十进制的 15  
015 : 八进制的 15，前面是数字 0，相当于十进制的 13，计算公式：1\*8+5=13  
0x15: 十六进制的 15，相当于十进制的 21，计算公式：1\*16+5=21

#### 3.1.1: 在 Java 中的定义示例

示例 1: byte abc = 5;

表示在 Java 中定义一个变量 `abc`, 类型是 `byte` 类型, 值是 5

同理可以定义其它的类型:

比如:

```
short abc1 = 5;
```

```
int abc2 = 5;
```

```
long abc3 = 5;
```

这些都是可以的, 如果要明确表示是 `long` 型的值, 可以在后面直接跟一个字母“L”。L 表示一个 `long` 值。

也就是写成: `long abc4 = 5L;`

请注意, 在 Java 编程语言中使用大写或小写 L 同样都是有效的, 但由于小写 l 与数字 1 容易混淆, 因而, 尽量不要使用小写。

**注意: 整数型的值, 如果没有特别指明, 默认是 `int` 型**

### 3.1.2 取值范围和默认值

取值范围的表示是按 Java 编程语言规范定义的且不依赖于平台

名称	长度	范围	默认值
byte	8 位	$-2^7 \dots 2^7 - 1$	0
short	16 位	$-2^{15} \dots 2^{15} - 1$	0
int	32 位	$-2^{31} \dots 2^{31} - 1$	0
long	64 位	$-2^{63} \dots 2^{63} - 1$	0

### 3.2: 浮点型: `float`、`double`

Java 用浮点型来表示实数, 简单地说就是带小数的数据。

用 `float` 或 `double` 来定义浮点类型, 如果一个数字包括小数点或指数部分, 或者在数字后带有字母 F 或 f (`float`)、D 或 d (`double`), 则该数字文字为浮点型的。

示例:

```
12.3 //简单的浮点型数据
```

```
12.3E10 //数据很大的一个浮点数据
```

#### 3.2.1: 在 Java 中的定义示例

如下定义都是可以的:

```
float abc = 5.6F;
```

```
float abc = 5.6f;
```

```
double abc = 5.6;
```

```
double abc = 5.6D;
```

```
double abc = 5.6d;
```

#### 3.2.2: 提示

(1): 浮点型的值, 如果没有特别指明, 默认是 `double` 型的

(2): 定义 `float` 型的时候, 一定要指明是 `float` 型的, 可以通过在数字后面添加“F”或者“f”来表示。

(3): 定义 double 型的时候，可以不用指明，默认就是 double 型的，也可以通过在数字后面添加”D”或者”d”来表示。

### 3.2.3 取值范围和默认值

名称	长度	默认值
float	32 位	0.0f
double	64 位	0.0

Java 技术规范的浮点数的格式是由电力电子工程师学会（IEEE）754 定义的，是独立于平台的。可以通过 Float.MAX\_VALUE 和 Float.MIN\_VALUE 取得 Float 的最大最小值；可以通过 Double.MAX\_VALUE 和 Double.MIN\_VALUE 来取得 Double 的最大最小值。

### 3.3: 字符型: char

char 类型用来表示单个字符。一个 char 代表一个 16-bit 无符号的（不分正负的）Unicode 字符，一个 char 字符必须包含在单引号内。

示例：

```
‘a’ //表示简单的字符
‘1’ //用数字也可以表示字符
```

下面就错了，只能使用单个字符

```
‘ab’ //错误
‘12’ //错误
```

#### 3.3.1: 什么是 Unicode 编码

Unicode 编码又叫统一码、万国码或单一码，是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990 年开始研发，1994 年正式公布。随着计算机工作能力的增强，Unicode 也在面世以来的十多年里得到普及。unicode 的表达如下：

‘\u????’ 一个 Unicode 字符。????应严格按照四个 16 进制数进行替换。

#### 3.3.2: 在 Java 中的定义示例

```
char c = ‘a’;
char c = ‘1’;
```

#### 3.3.3: 取值范围和默认值

名称	长度	范围	默认值
char	16 位	Unicode $2^{16}-1$	Unicode 0



### 3.3.4: Java 里面的转义字符

转义字符是指，用一些普通字符的组合来代替一些特殊字符，由于其组合改变了原来字符表示的含义，因此称为“转义”。常见的转义字符：

- \n 换行(\u000a)
- \t 水平制表符(\u0009)
- \b 空格(\u0008)
- \r 回车(\u000d)
- \f 换页(\u000c)
- \' 单引号(\u0027)
- \" 双引号(\u0022)
- \\ 反斜杠(\u005c)

### 3.4: 逻辑型: boolean

---

逻辑值有两种状态，即人们经常使用的“on”和“off”或“true”和“false”或“yes”和“no”，这样的值是用 boolean 类型来表示的。boolean 有两个文字值，即 true 和 false。以下是一个有关 boolean 类型变量的声明和初始化：

```
boolean truth = true; //声明变量 truth
```

注意——在整数类型和 boolean 类型之间无转换计算。有些语言（特别值得强调的是 C 和 C++）允许将数字值转换成逻辑值（所谓“非零即真”），这在 Java 编程语言中是不允许的；boolean 类型只允许使用 boolean 值（true 或 false）。

**注意：Java 中不可以直接将整数类型和逻辑类型转换**

### 3.5: 声明和赋值

---

#### 3.5.1: 什么是声明

声明为 Java 程序实体引入标识符，能够使用这些名字访问到这些实体，声明实体包括：类名、属性名、方法名、变量名、参数名、接口名等等。其实简单点说就是定义某个东西并对外宣称它。

#### 3.5.2: 什么是赋值

赋值就是为一个声明的变量或者常量赋予具体的值，也就是赋予值的意思。使用一个等号“=”来表示。

示例：

```
int a = 5;
```

这句话的意思就是，声明一个类型为 int 的变量 a，并将它赋值为 5。

### 3.6: 特别介绍: 字符串型 String

---

字符型只能表示一个字符，那么多个字符怎么表示呢？

Java 中使用 String 这个类来表示多个字符，表示方式是用双引号把要表示的字符串引起来，字符串里面的字符数量是任意多个。字符本身符合 Unicode 标准，且上述 char 类型的反斜线符号(转义字符)适用于 String。与 C 和 C++ 不同，String 不能用 \0 作为结束。String 的文字应用双引号封闭，如下所示：

“The quick brown fox jumped over the lazy dog.”

char 和 String 类型变量的声明和初始化如下所示:

```
char ch = 'A'; // 声明并初始化一个字符变量
char ch1, ch2; // 声明两个字符变量
// 声明两个字符串变量并初始化它们
String greeting = "Good Morning !! \n";
String err_msg = "Record Not Found !";
String str1, str2; // 声明两个字符串变量
String s = "12abc"; // 基本的字符串型
String s = ""; // 表示空串
```

注意:

- (1): **String** 不是原始的数据类型, 而是一个类(class)
- (2): **String** 包含的字符数量是任意多个, 而字符类型只能是一个。  
要特别注意: "a" 表示的是字符串, 而'a'表示的是字符类型, 它们具有不同的功能。
- (3): **String** 的默认值是 null

### 3.7: 示例

---

下列程序显示了如何为整数、浮点数、boolean、字符和 String 类型变量声明和赋值

```
1. public class Assign {
2. public static void main(String args []) {
3. int x, y; // 声明 int 变量
4. float z = 3.414f; // 声明并赋值 float
5. double w = 3.1415; // 声明并赋值 double
6. boolean truth = true; // 声明并赋值 boolean
7. char c; // 声明字符变量
8. String str; // 声明 String 字符串
9. String str1 = "bye"; // 声明并赋值 String 变量
10. c = 'A'; // 给字符变量赋值
11. str = "Hi out there!"; // 给 String 变量赋值
12. x = 6;
13. y = 1000; // 给 int 变量赋值
14. ...
15. }
16. }
```

非法赋值举例

```
y = 3.1415926; // 3.1415926 不是一个 int.
```

```
    // 需要类型转换并且小数位要截掉
```

```
w = 175,000; // 逗号(,) 不能够出现
```

truth = 1; // 一个优秀的 C/C++ 程序员常犯的错误, 在 Java 语言中 boolean 型变量只能为 true 或 false

```
z = 3.14156; // double 型的值不能赋给 float 变量, 需要类型转换
```

对于引用数据类型放到后面再学, 先看看常量和变量。

## 四: 常量和变量

---

### 1: 什么是常量

常量是值不可以改变的标识符。

对常量的定义规则: 建议大家尽量全部大写, 并用下划线将词分隔。

如: JAVASS\_CLASS\_NUMBER, FILE\_PATH

### 2: 什么是变量

变量是值可以改变的标识符, 用来引用一个存储单元, 用标识符来表示, 可以通过操作变量来操作变量所对应的内存区域或值块的值。

下面来理解一下:

#### 2.1: 变量是标识符

也就是说变量本质上是标识符, 但是所有的标识符都是变量吗? 很显然不是的, 那么哪些标识符才是变量呢?

#### 2.2: 值可以改变

一定是值可以改变的这些标识符才被称为变量, 注意是可以改变, 不是一定要改变。

比如:

我们定义人的体重叫做 `weight`, 现在测量某人的体重是 70kg, 也就是 `weight` 的值是 70, 然后让他吃饭, 吃完饭过后马上再次测量, 此时体重可能是 71kg, 也就是说 `weight` 的值发生了变化, 变成了 71 了。对象并没有发生变化, 属性也没有发生变化, 只是这个属性的值发生了变化。

#### 2.3 变量的定义规则

- (1): 遵从所有标识符的规则
- (2): 所有变量都可大小写混用, 但首字符应小写
- (3): 尽量不要使用下划线和\$符号
- (4): 可以先声明再赋值, 如:

```
int i;  
i=9;
```

也可以声明的同时进行赋值:

```
int i=9;
```

#### 2.4 几点说明

- (1): 变量在计算机内部对应着一个存储单元, 而且总是具有某种数据类型: 基本数据类型或引用数据类型
- (2): 变量总是具有与其数据类型相对应的值
- (3): 每个变量均具有: 名字、类型、一定大小的存储单元以及值

## 五: Java 代码的基本知识

---

### 1: 语句

用分号”;  
”结尾的一行代码就是语句, Java 中语句必须以”;  
”结尾。

如: `int a = 10;`

可以有块语句, 例如:

```
int i=0;  
{  
    int j = 0;
```

```
        j = j + 1;
    }
    i++;
```

## 2: 块 (block)

一个块是以 {} 作为边界的语句的集合, 块可以嵌套。如:

```
{
    int a = 10;
    String s = ""; //一条语句或多条语句均可
    {
        System.out.println("块可以嵌套");
    }
}
```

## 3: 注释

什么是注释呢? 就是标注解释的意思, 主要用来对 Java 代码进行说明。Java 中有三种注释方式

(1): // : 注释单行语句

示例:

//定义一个值为 10 的 int 变量

```
int a = 10;
```

(2): /\* \*/ : 多行注释

示例:

```
/*
```

这是一个注释, 不会被 Java 用来运行

这是第二行注释, 可以有任意多行

```
*/
```

(3): /\*\* \*/ : 文档注释

紧放在变量、方法或类的声明之前的文档注释, 表示该注释应该被放在自动生成的文档中(由 javadoc 命令生成的 HTML 文件)以当作对声明项的描述。

示例:

```
/**
```

\* 这是一个文档注释的测试

\* 它会通过 javadoc 生成标准的 java 接口文档

```
*/
```

常常在 javadoc 注释中加入一个以 “@” 开头的标记, 结合 javadoc 指令的参数, 可以在生成的 API 文档中产生特定的标记

### 常用的 javadoc 标记

@author: 作者

@version: 版本

@deprecated: 不推荐使用的方法

@param: 方法的参数类型

@return: 方法的返回类型

@see: “参见”, 用于指定参考的内容

@exception: 抛出的异常

@throws: 抛出的异常, 和 exception 同义

## javadoc 标记的应用范围

在类和接口文档注释中的标记有 @see @deprecated @author @version

在方法或者构造方法中的标记有: @see @deprecated @param @return @exception @throws

在属性文档注释中的标记: @see @deprecated

## 4: 空格

在一个 Java 程序中任何数量的空格都是允许的

## 5: Java 编程基本的编码约定

可能有些还没有学到, 没有关系, 先了解一下。

**类**——类名应该是名词, 大小写可混用, 但首字母应大写。例如:

```
class AccountBook
class ComplexVariable
```

**接口**——接口名大小写规则与类名相同。

```
interface Account
```

**方法**——方法名应该是动词, 大小写可混用, 但首字母应小写。在每个方法名内, 大写字母将词分隔并限制使用下划线。例如:

```
balanceAccount( )
addComplex( )
```

**变量**——所有变量都可大小写混用, 但首字符应小写。词由大写字母分隔, 限制用下划线, 限制使用美元符号 (\$), 因为这个字符对内部类有特殊的含义。

```
currentCustomer
```

变量应该代表一定的含义, 通过它可传达给读者使用它的意图。尽量避免使用单个字符, 除非是临时“即用即扔”的变量 (例如, 用 i, j, k 作为循环控制变量)

**常量**——全部大写并用下划线将词分隔。

```
HEAD_COUNT
MAXIMUM_SIZE
```

**控制结构**——当语句是控制结构的一部分时, 即使是单个语句也应使用括号 ({}) 将语句封闭。例如:

```
if (condition) {
    do something
} else {
    do something else
}
```

**语句行**——每行只写一个语句并使用四个缩进的空格使你的代码更易读。

**注释**——用注释来说明那些不明显的代码段落; 对一般注释使用 // 分隔符, 而大段的代码可使用 /\*...\*/ 分隔符。使用 /\*\*...\*/ 将注释形成文档, 并输入给 javadoc 以生成 HTML 代码文档。

## 六: 运算符

Java 运算符很多, 下面按优先顺序列出了各种运算符 (“L to R” 表示左到右结合, “R to L” 表示右到左结合)

分隔符	. [] () ; ,
右结合	++ -- - !
左结合	* / %
左结合	+ -
左结合	<< >> >>>
左结合	< > <= >= instanceof (Java 特有)
左结合	== !=
左结合	&
左结合	^
左结合	
左结合	&&
左结合	
右结合	?:
右结合	= *= /= %= += -= <<= >>= >>>= &= *=  =

注意: 运算符的结合性决定了同优先级运算符的求值顺序

### 1: 算术运算

算术运算是指: +、-、\*、/ 等基本运算

需要注意的是:

%是求 mod 运算;

整数的除法要小心:

$5/2 = 2$  // 不是 2.5

### 2: 比较运算

比较运算是指: >、<、>=、<=、==、!= 等类似运算

需要注意的是:

字符可以比较大小; (用它们的 ascii 码, 化为整数)

小心浮点数的相等比较

instanceof 也是一个比较运算, 用来判断一个对象是否属于某个类。(以后介绍)

==运算中, 对于基本类型是比较的“内容”, 而对于引用类型, 比较的是地址。(小心)

### 3: 逻辑运算

运算符&& (定义为“与”)和|| (定义为“或”)执行布尔逻辑表达式。请看下面的例子:

```
MyDate d = null;
```

```
if ((d != null) && (d.day() > 31)) {
```

```
    // 利用 d 执行些什么
```

}

形成 if () 语句自变量的布尔表达式是合法且安全的。这是因为当第一个子表达式是假时，第二个子表达式被跳过，而且当第一个子表达式是假时，整个表达式将总是假，所以不必考虑第二个子表达式的值。类似的，如果使用 || 运算符，而且第一个表达式返回真，则第二个表达式不必求值，因为整个表达式已经被认为是真。

#### 4: ++、--运算

++运算相当于：运算的变量加 1，如：x++ 等同于 x=x+1;

--运算恰好相反，相当于运算的变量减 1

**注意：**x++ 和 ++x 并不是一回事情。x++是先使用，然后再加；++x 是先加然后再使用。

#### 5: =赋值运算

x=5 相当于把 5 这个值赋给变量 x

#### 6: 位运算

位逻辑运算符 (Bitwise Logical Operations)。

算术逻辑运算符& (与), | (或), ~ (非), ^ (异或);

位运算示例

$\sim$	<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	0	0	1	1	1	1		<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	0	1	1	0	1		$\wedge$	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	0	1	1	0	1		$\sim$	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	0	1	1	0	1		$\vee$	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	0	1	1	0	1	
0	1	0	0	1	1	1	1																																														
0	0	1	0	1	1	0	1																																														
0	0	1	0	1	1	0	1																																														
0	0	1	0	1	1	0	1																																														
0	0	1	0	1	1	0	1																																														
	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	1	1	0	0	0	0		$\&$	<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	0	0	1	1	1	1			<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	0	0	1	1	1	1			<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	0	0	1	1	1	1			<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	0	0	1	1	1	1
1	0	1	1	0	0	0	0																																														
0	1	0	0	1	1	1	1																																														
0	1	0	0	1	1	1	1																																														
0	1	0	0	1	1	1	1																																														
0	1	0	0	1	1	1	1																																														
				<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	0	0	1	1	0	1				<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	1	0	0	0	1	0				<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	1	0	1	1	1	1																	
0	0	0	0	1	1	0	1																																														
0	1	1	0	0	0	1	0																																														
0	1	1	0	1	1	1	1																																														

#### 7: 移位运算

Java 编程语言提供了两种右移位运算符和一种左移运算符，右移一位 ( >> ) 相对于除以 2；左移 ( << ) 相对于乘以 2。

(1): 运算符>>进行算术或符号右移位。移位的结果是第一个操作数被 2 的幂来除，而指数的值是由第二个数给出的。例如：

128 >> 1 gives  $128/2^1 = 64$

256 >> 4 gives  $256/2^4 = 16$

-256 >> 4 gives  $-256/2^4 = -16$

(2): 逻辑或非符号右移位运算符>>>主要作用于位图，而不是一个值的算术意义；它总是将零置于符号位上。例如：

1010 ... >> 2 gives 111010 ...

1010 ... >>> 2 gives 001010 ...

在移位的过程中， >>运算符使符号位被拷贝。





```
public class Test {  
    public static void main(String[] args) {  
        int i = (5>3) ? 6 : 7;  
        System.out.println("the i="+i);  
    }  
}
```

运行结果为: the i=6

其实三目运算符的基本功能相当于 if-else (马上就要学到了), 使用三目运算符是因为它的表达比相同功能的 if-else 更简洁。上面的例子改成用 if-else 表达如下:

```
public class Test {  
    public static void main(String[] args) {  
        int i = 0;  
        if (5 > 3) {  
            i = 6;  
        } else {  
            i = 7;  
        }  
        System.out.println("the i=" + i);  
    }  
}
```

运行结果为: the i=6

## 七: 控制语句

---

### 1: 分支控制语句

---

分支语句又称条件语句, 条件语句使部分程序可根据某些表达式的值被有选择地执行。Java 编程语言支持双路 if 和多路 switch 分支语句。

#### 1.1 if, else 语句

---

if, else 语句的基本句法是:

```
if (布尔表达式) {  
    语句或块;  
} else {  
    语句或块;  
}
```

例如:

```
int count;  
1. count = getCount(); // a method defined in the program  
2. if (count < 0) {  
3.     System.out.println("Error: count value is negative.");  
4. } else {  
5.     System.out.println("There will be " + count +  
6.         " people for lunch today.");  
7. }
```

在 Java 编程语言中, if ()用的是一个布尔表达式, 而不是数字值, 这一点与 C/C++不同。前面已经讲过, 布尔类型和数字类型不能相互转换。因而, 如果出现下列情况:

```
if (x) // x 是 int 型
```

你应该使用下列语句替代:

```
if (x != 0)
```

### 注意:

(1): if 块和 else 块可以包含任意的 java 代码, 自然也就可以包含新的 if-else, 也就是说: if-else 是可以嵌套的, 嵌套的规则是不可以交叉, 必须完全包含。比如:

```
if(a1 > a2){  
    if(a1 > a3){  
        //包含在里面的块必须先结束  
    }else{  
        //同样可以包含 if-else 块  
    }  
}
```

(2): else 部分是选择性的, 并且当测试条件为假时如不需做任何事, else 部分可被省略。也就是说 if 块可以独立存在, 但是 else 块不能独立存在, 必须要有 if 块才能有 else 块。

(3): 如果 if 块和 else 块的语句只有一句时, 可以省略 {}, 例如:

```
if(a1 > a2)  
    System.out.println("这是 if 块");  
else  
    System.out.println("这是 else 块");
```

上面的代码从语法角度是完全正确的, 但是从代码的可阅读性上, 容易让人产生迷惑, 所以我们**不建议**大家这么写。

(4): 还可以把 else 和 if 组和使用, 就是使用 else if, 表达“否则如果”的意思, 示例如下:

```
if(a1 > a2){  
    System.out.println("a1 > a2");  
}else if(a1 > a3){
```

```
        System.out.println( "a1 > a3" );
    }else if(a1 > a4){
        System.out.println( "a1 > a4" );
    }else{
        System.out.println( "now is else" );
    }
}
```

再来个例子:

```
if (score>=90){
    grade= "very good";
}else if(score>=70){
    grade= "good";
}else if (score>=60){
    grade= "pass";
}else{
    grade= "No pass";
}
```

**(5):** 如果不用 “{ }” , 则 “else” 总是与最接近它的前一个 “if” 相匹配  
例如:

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
else
    System.out.println( "x is <= 5" );
```

执行与上面的对应匹配形式不同, 而是与下面形式匹配

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
else
    System.out.println( "x is <= 5" );
```

## 1.2 switch 语句

---

switch 表示选择分支的情况, switch 语句的句法是:

```
switch (expr1) {  
    case expr2:  
        statements;  
        break;  
    case expr3:  
        statements;  
        break;  
    default:  
        statements;  
        break;  
}
```

说明:

(1): switch 的表达式 expr1 只能是整数型的值或者 enum 枚举类型的常量值, 包含 byte、short、int 和 char, 不能是字符串或对象, 也不能是 long 型的值。

(2): 当变量或表达式的值不能与任何 case 值相匹配时, 可选缺省符 (default) 指出了应该执行的程序代码。

例 1:

```
public class Test {  
    public static void main(String[] args) {  
        int colorNum = 5;  
        switch (colorNum) {  
            case 0:  
                System.out.println(Color.red);  
                break;  
            case 1:  
                System.out.println(Color.green);  
                break;  
            default:  
                System.out.println(Color.black);  
                break;  
        }  
    }  
}
```

运行结果: java.awt.Color[r=0,g=0,b=0]

(3): 如果没有 break 语句作为某一个 case 代码段的结束句, 则程序的执行将继续到下一个 case, 而不检查 case 表达式的值。

例 2:

```
public class Test {  
    public static void main(String[] args) {  
        int colorNum = 0;  
        switch (colorNum) {  
            case 0:  
                System.out.println(Color.red);  
            case 1:  
                System.out.println(Color.green);  
            default:  
                System.out.println(Color.black);  
                break;  
        }  
    }  
}
```

运行结果：

```
java.awt.Color[r=255,g=0,b=0]  
java.awt.Color[r=0,g=255,b=0]  
java.awt.Color[r=0,g=0,b=0]
```

例 2 设定背景颜色为黑色，而不考虑 case 变量 colorNum 的值。如果 colorNum 的值为 0，背景颜色将首先被设定为红色，然后为绿色，再为黑色。

## 2: 循环控制语句

---

循环语句使语句或块的执行得以重复进行。Java 编程语言支持三种循环构造类型：for，while 和 do 循环。for 和 while 循环是在执行循环体之前测试循环条件，而 do 循环是在执行完循环体之后测试循环条件。这就意味着 for 和 while 循环可能连一次循环体都未执行，而 do 循环将至少执行一次循环体。

### 2.1 for 循环

for 循环的句法是：

```
for (初值表达式; boolean 测试表达式; 改变量表达式) {  
    语句或语句块  
}
```

例如：

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Are you finished yet?");  
}  
  
System.out.println("Finally!");
```

注意：for 语句里面的 3 个部分都可以省略，但是我们不建议这么做。

示例如下:

示例 1: 就是一个无限循环

```
public class Test {  
    public static void main(String[] args) {  
        for(;;){  
            System.out.println("Java私塾");  
        }  
    }  
}
```

示例 2: 可以省略部分

```
public class Test {  
    public static void main(String[] args) {  
        for(int i=0;;){  
            System.out.println("Java私塾"+i);  
        }  
    }  
}
```

示例 3: 可以省略部分

```
public class Test {  
    public static void main(String[] args) {  
        for(int i=0;i<3;){  
            System.out.println("Java私塾"+i);  
        }  
    }  
}
```

示例 4: 可以省略部分

```
public class Test {  
    public static void main(String[] args) {  
        for(int i=0;;i++){  
            System.out.println("Java私塾"+i);  
        }  
    }  
}
```

当然还有其他的组合方式, 都是可以的。

## 2.2 while 循环

while 循环的句法是:

```
while (布尔表达式) {  
    语句或块  
}
```

例如:



```
int i = 0;

while (i < 10) {

    System.out.println("Are you finished yet?");

    i++;

};

System.out.println("Finally!");
```

请确认循环控制变量在循环体被开始执行之前已被正确初始化，并确认循环控制变量是真时，循环体才开始执行。控制变量必须被正确更新以防止死循环。

## 2.3 do 循环

do 循环的句法是：

```
do {

    语句或块;

} while (布尔测试);
```

例如：

```
int i = 0;

do {

    System.out.println("Are you finished yet?");

    i++;

} while (i < 10);

System.out.println("Finally!");
```

象 while 循环一样，请确认循环控制变量在循环体中被正确初始化和测试并被适时更新。作为一种编程惯例，for 循环一般用在那种循环次数事先可确定的情况，而 while 和 do 用在那种循环次数事先不可确定的情况。

do 循环与 while 循环的不同这处在于，前者至少执行一次，而后者可能一次都不执行。

## 2.4 特殊循环流程控制

下列语句可被用在更深层次的控制循环语句中：

- ! break [标注];
- ! continue [标注];
- ! label: 语句; //这里必须是任意的合法的语句

break 语句被用来从 switch 语句、循环语句和预先给定了 label 的块中退出，跳出离

break 最近的循环。

continue 语句被用来略过并跳到循环体的结尾, 终止本次循环, 继续下一次循环。

label 可标识控制需要转换到的任何有效语句, 它被用来标识循环构造的复合语句。它类似以前被人诟病的” goto ” 语句, 我们应该尽量避免使用。

例 1: break 的使用

```
public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            if (i == 5) {
                break;
            }
            System.out.println("i==" + i);
        }
    }
}
```

运行的结果: i==0 一直到 i==4

因为当 i==5 的时候, 执行 break, 跳出 for 循环。

例 2: continue 的使用

```
public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            if (i == 3) {
                continue;
            }
            System.out.println("i==" + i);
        }
    }
}
```

运行的结果: i==0 一直到 i==4, 就是不包括 i==3。因为当 i==3 的时候, 执行 continue, 终止本次循环, 继续下一次循环。

例 3: label 的使用

```
public class Test {
    public static void main(String[] args) {
        L: for (int i = 0; i < 5; i++) {
            if (i == 3) {
                break L;
            }
            System.out.println("i==" + i);
        }
    }
}
```

```
    }  
  }  
}
```

运行的结果是: i==0 一直到 i==2

例 4: label 的使用

```
public class Test {  
    public static void main(String[] args) {  
        for (int i = 0; i < 5; i++) {  
            L:if (i == 3) {  
                break L;  
            }  
            System.out.println("i==" + i);  
        }  
    }  
}
```

运行的结果是: i==0 一直到 i==4

## 作业

---

- 1: 叙述标识符的定义规则, 指出下面的标识符中那些是不正确的, 并说明理由  
here , \_there, this, it, 2tol, \_it
- 2: Java 中共有那些基本数据类型? 分别用什么符号来表示, 各自的取值范围是多少?
- 3: 复习所有的 Java 关键字
- 4: 指出正确的表达式  
A byte b=128;  
B char c=65536;  
C long len=0xffffL;  
D double dd=0.9239d;
- 5: 下面哪几个语句将引起编译错?  
A. float f=4096.0;  
B. double d=4096.0;  
C. byte b=4096;  
D. char c=4096;
- 6: 简述 Java 中的运算符, 以及他们的运算次序。
- 7: 创建一个 switch 语句, 为每一种 case 都显示一条消息。并将 switch 置入一个 for 循环里, 令其尝试每一种 case。在每个 case 后面都放置一个 break, 并对其进行测试。然后, 删除 break, 看看会有什么情况出现。
- 8: 执行下列代码后的 x 和 y 的结果分别是什么?  

```
int x, y, a=2;
x=a++;
y=++a;
```
- 9: 下面的程序输出结果是: a=6 b=5  
请将程序补充完整。  

```
public class A
{
    public static void main(String args[])
    {
        int a=5, b=6;
        a= _____;
        b=a-b;
        a= _____;
        System.out.println("a="+a+"    b="+b);
    }
}
```
- 10: 下面哪个语句序列没有错误, 能够通过编译?  
A.  

```
int i=0;
if (i) {
    System.out.println( "Hi" );
}
```

  
B.

```
boolean b=true;
boolean b2=true;
if(b==b2) {
    System.out.println( "So true" );
}
```

C.

```
int i=1;
int j=2;
if(i==1 || j==2)
    System.out.println( "OK" );
```

D.

```
int i=1;
int j=2;
if (i==1 &| j==2)
    System.out.println( "OK" );
```

11: 阅读以下代码行:

```
boolean a=false;
boolean b=true;
boolean c=(a&&b)&&(!b);
int result=c==false?1:2;
```

这段程序执行完后, c 与 result 的值是:

- A c=false; result=1;
- B c=true; result=2;
- C c=true; result=1;
- D c=false; result=2;

12: 下列代码哪行会出错?

```
1) public void modify() {
2)     int i, j, k;
3)     i = 100;
4)     while ( i > 0 ) {
5)         j = i * 2;
6)         System.out.println ( " The value of j is " + j );
7)         k = k + 1;
8)         i--;
9)     }
10) }
```

- A 第 4 行
- B 第 6 行
- C 第 7 行
- D 第 8 行

13: 指出下列程序的运行结果。

```
int i = 9;
switch (i) {
```

```
default:
    System.out.print("default");
case 0:
    System.out.print("zero"); break;
case 1:
    System.out.print("one");
case 2:
    System.out.print("two");
}
```

A default  
B defaultzero  
C 编译错  
D 没有任何输出

**往下是编程题:**

- 14: 将 1 到 1000 之间的奇数打印出来。  
15: 判断一个数能否同时被 3 和 5 整除。  
16: 输入 10 个数, 找出最大一个数, 并打印出来。  
17: 给出一百分制成绩, 要求输出成绩等级 'A', 'B', 'C', 'D', 'E'。90 分以上为 'A', 80~89 分为 'B', 70~79 分为 'C', 60~69 分为 'D', 60 分以下为 'E'。  
18: 输出图案:

```
*
**
***
****
*
**
***
****
```

- 19: 使用 for 语句打印显示下列数字形式: n=4

```
1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
```

## 第三章 Java 类和对象

### 教学目标：

- 面向对象基础
- 掌握对象的三大特性
- 掌握 Java 类的构建
- 掌握如何使用 Java 类
- 理解引用类型
- 理解按值传递和按引用传递
- 深入理解变量
- 掌握包装类
- 理解类型转换
- 理解 Java 类的基本运行顺序



## 一：面向对象初步

---

### 1: 什么是对象

---

对象是真实世界中的物体在人脑中的映象，包括实体对象和逻辑对象。实体对象指的是我们能在现实生活中能看得见、摸得着，实际存在的东西，比如：人，桌子，椅子等。逻辑对象是针对非具体物体，但是在逻辑上存在的东西的反映，比如：人与人的关系。为了简单，这里讨论的对象都是实体对象。

### 2: 对象的基本构成

---

初次接触对象，我们从实体对象入手，因为看得见、摸得着会比较容易理解。

分析实体对象的构成，发现有这样一些共同点，这些实体对象都有自己的属性，这些属性用来决定了对象的具体表现，比如：人有身高、体重等。

除了这些静态的，用于描述实体对象的基本情况外，实体对象还有自己的动作，通过这些动作能够完成一定的功能，我们称之为方法，比如：人的手能动，能够写字，能够刷牙等。

对象同时具备这些静态属性和动态的功能。

### 3: 如何进行对象抽象

---

抽象是在思想上把各种对象或现象之间的共同的本质属性抽取出来而舍去个别的非本质的属性的思维方法。也就是说把一系列相同或类似的实体对象的特点抽取出来，采用一个统一的表达方式，这就是抽象。

比如：张三这个人身高 180cm，体重 75kg，会打篮球，会跑步

李四这个人身高 170cm，体重 70kg，会踢足球

现在想要采用一个统一的对象来描述张三和李四，那么我们就可以采用如下的表述方法来表述：

人{

静态属性：

姓名；

身高；

体重；

动态动作：

打篮球();

跑步();

踢足球();

}

这个“人”这个对象就是对张三和李四的抽象，那么如何表述张三这个具体的个体呢：

人{

静态属性：

姓名=张三；

身高 = 180cm;

体重 = 75kg;

动态动作：

打篮球(); //相应的打篮球的功能实现

跑步();//相应的跑步的功能实现

踢足球();

}

如何表述李四这个具体的个体呢：

```
人{
    静态属性：
        姓名=李四;
        身高 = 170cm;
        体重 = 70kg;
    动态动作：
        打篮球();
        跑步();
        踢足球();//相应的踢足球的功能实现
}
```

对实体对象的抽象一定要很好的练习，可以把你所看到的任何物体都拿来抽象，“一切皆对象”。要练习到，你看到的没有物体，全是对象就好了。

#### 4: 抽象对象和实体对象的关系

---

仔细观察上面的抽象对象——“人”，和具体的实体对象：“张三”、“李四”。你会发现，抽象对象只有一个，实体对象却是无数个，通过对抽象对象设置不同的属性，赋予不同的功能，那么就能够表示不同的实体对象。

这样就大大简化了对象的描述工作，使用一个对象就可以统一地描述某一类实体了，在需要具体的实体的时候，分别设置不同的值就可以表示具体对象了。

#### 5: Java 中的类和对象

---

##### 5.1: Java 中的类

把抽象出来的对象使用 Java 表达出来，那就是类 class。类在 Java 编程语言中作为定义新类型的一种途径，类声明可定义新类型并描述这些类型是如何实现的。接下来将会学习许多关于类的特性。

比如前面讨论过的“人”使用 Java 表达出来就是一个类。

##### 5.2: Java 中的对象

Java 中的对象是在 Java 中一个类的实例，也称实例对象。实例就是实际例子。

类可被认为是一个模板——你正在描述的一个对象模型。一个对象就是你每次使用的时候创建的一个类的实例的结果。

比如前面讨论的张三和李四，他们就是“人”这个类的实例。

## 二：面向对象三大特征

---

### 1: 封装

---

封装这个词听起来好象是将什么东西包裹起来不要别人看见一样，就好像是把东西装进箱子里面，这样别人就不知道箱子里面装的是什么东西了。其实 JAVA 中的封装这个概念也就和这个差不多的意思。

封装是 JAVA 面向对象的特点的表现，封装是一种信息隐蔽技术。它有两个含义：即把对象的全部属性和全部服务结合在一起，形成一个不可分割的独立单位；以及尽可能隐藏对象的内部结构。也就是说，如果我们使用了封装技术的话，别人就只能用我们做出来的东西而看不见我们做的这个东西的内部结构了。

## 封装的功能

- 隐藏对象的实现细节
- 迫使用户去使用一个界面访问数据
- 使代码更好维护

封装迫使用户通过方法访问数据能保护对象的数据不被误修改，还能使对象的重用变得更简单。数据隐藏通常指的就是封装。它将对象的外部界面与对象的实现区分开来，隐藏实现细节。迫使用户去使用外部界面，即使实现细节改变，还可通过界面承担其功能而保留原样，确保调用它的代码还继续工作。封装使代码维护更简单。

## 2: 继承

---

### is a 关系——子对象

在面向对象世界里面，常常要创建某对象（如：一个职员对象），然后需要一个该基本对象的更专业化的版本，比如，可能需要一个经理的对象。显然经理实际上是一个职员，经理和职员具有 is a 的关系，经理只是一个带有附加特征的职员。因此，需要有一种办法从现有对象来创建一个新对象。这个方式就是继承。

“继承”是面向对象软件技术当中的一个概念。如果一个对象 A 继承自另一个对象 B，就把这个 A 称为“B 的子对象”，而把 B 称为“A 的父对象”。继承可以使得子对象具有父对象的各种属性和方法，而不需要再次编写相同的代码。在令子对象继承父对象的同时，可以重新定义某些属性，并重写某些方法，即覆盖父对象的原有属性和方法，使其获得与父对象不同的功能。

## 3: 多态

---

同一行为的多种不同表达，或者同一行为的多种不同实现就叫做多态。

还是用刚才经理和职员这个例子来举例：人事部门需要对公司所有职员统一制作胸卡（一般也就是门禁卡，进出公司证明身份使用），制作的师傅说，只要告诉我一个人员的信息，就可以制作出一份胸卡，简化一下就是：一位职员的信息对应一份胸卡。

这个时候，对胸卡制作的师傅而言，所有的人都是职员，无所谓是经理还是普通职员。也就是说，对于传递职员信息这样一个行为，存在多种不同的实现，既可以传递经理的信息，也可以传递普通职员的信息。这就是多态的表现。

再举一个例子：比如我们说“笔”这个对象，它就有很多不同的表达或实现，比如有钢笔、铅笔、圆珠笔等等。那么我说“请给我一支笔”，你给我钢笔、铅笔或者圆珠笔都可以，这里的“笔”这个对象就具备多态。

## 三: Java 类的基本构成

---

### 1: Java 类的定义形式

---

一个完整的 Java 类通常由下面六个部分组成：

```
包定义语句
import 语句
类定义{
    成员变量
    构造方法
    成员方法
}
```

其中：只有类定义和“{}”是不可或缺的，其余部分都可以根据需要来定义。

下面分别来学习各个部分的基本规则，看看如何写 Java 的类。

## 2: 包

### 2.1: 包是什么

在 Java 中，包是类、接口或其它包的集合，包主要用来将类组织起来成为组，从而对类进行管理。

### 2.2: 包能干什么

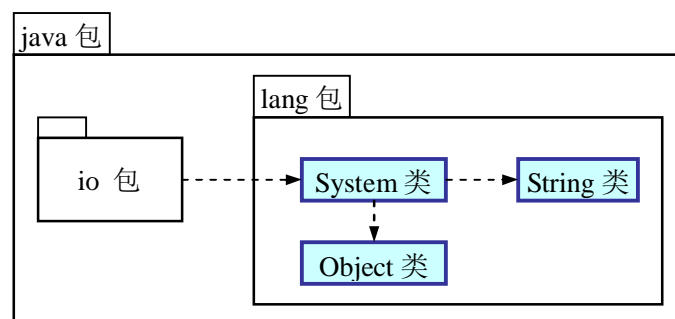
包对于下列工作非常有用：

- (1): 包允许您将包含类代码的文件组织起来，易于查找和使用适当的类。
- (2): 包不止是包含类和接口，还能够包含其它包。形成层次的包空间。
- (3): 它有助于避免命名冲突。当您使用很多类时，确保类和方法名称的唯一性是非常困难的。包能够形成层次命名空间，缩小了名称冲突的范围，易于管理名称。

为便于管理数目众多的类，Java 语言中引入了“包”的概念，可以说是对定义的 Java 类进行“分组”，将多个功能相关的类定义到一个“包”中，以解决命名冲突、引用不方便、安全性等问题。

就好似当今的户籍制度，每个公民除有自己的名字“张三”、“李四”外还被规定了他的户籍地。假定有两个人都叫张三，只称呼名字就无法区分他们，但如果事先登记他们的户籍分别在北京和上海，就可以很容易的用“北京的张三”、“上海的张三”将他们区分开来。如果北京市仍有多个张三，还可以细分为“北京市.海淀区的张三”、“北京市.西城区.平安大街的张三”等等，直到能惟一标识每个“张三”为止。

JDK 中定义的类就采用了“包”机制进行层次式管理，下图显示了其组织结构的一部分：



从图中可以看出，一个名为 java 的包中又包含了两个子包：io 包和 lang 包。lang 包中包含了 System, String, Object 三个类的定义。事实上，Java 包中既可以包含类的定义，也可以包含子包，或同时包含两者。

简而言之：从逻辑上讲，包是一组相关类的集合；从物理上讲，同包即同目录。

### 2.1: JDK 中常用的包

java.lang---包含一些 Java 语言的核心类，包含构成 Java 语言设计基础的类。在此包中定义的最重要的一个类是“Object”，代表类层次的根，Java 是一个单根系统，最终的根就是“Object”，这个类会在后面讲到。

Java 并不具有“自由”的方法，例如，不属于任何类的方法，Java 中的所有方法必须始终属于某个类。经常需要使用数据类型转换方法。Java 在 java.lang 包中定义了“包装对

象”类，使我们能够实现数据类型转换。如 Boolean、Character、Integer、Long、Float 和 Double，这些在后面会讲到。

此包中的其它类包括：

l Math——封装最常用的数学方法，如正弦、余弦和平方根。

l String, StringBuffer——封装最常用的字符串操作。

你不必显示导入该包，该 Java 包通常已经导入。

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

javax.swing----完全 Java 版的图形用户界面(GUI)解决方案，提供了很多完备的组件，可以应对复杂的桌面系统构建。

java.net----包含执行与网络相关的操作的类，如 URL, Socket, ServerSocket 等。

java.io----包含能提供多种输入/输出功能的类。

java.util----包含一些实用工具类，如定义系统特性、使用与日期日历相关的方法。还有重要的集合框架。

## 2.2: Java 中如何表达包——package 语句

Java 语言使用 package 语句来实现包的定义。package 语句必须作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包。若缺省该语句，则指定为无名包，其语法格式为：

```
package pkg1[.pkg2[.pkg3...]];           // “[]”表示可选
```

**Java 编译器把包对应于文件系统的目录管理**，因此包也可以嵌套使用，即一个包中可以含有类的定义也可以含有子包，其嵌套层数没有限制。package 语句中，用 ‘.’ 来指明包的层次；

程序 package 的使用：Test.java

```
package p1;
public class Test{
    public void display(){
        System.out.println("in method display()");
    }
}
```

Java 语言要求包声明的层次和实际保存类的字节码文件的目录结构存在对应关系，以便将来使用该类时能通过包名（也就是目录名）查找到所需要的类文件。简单地说就是包的层次结构需要和文件夹的层次对应。

**注意：每个源文件只有一个包的声明，而且包名应该全部小写。**

具体来说，程序员要做以下工作：

## 2.3: 编译和生成包

如果在程序 Test.java 中已定义了包 p1，就必须将编译生成的字节码文件 Test.class 保存

在与包名同名的子目录中, 可以选用下述两种方式之一:

采用下述命令编译:

```
javac Test.java
```

则编译器会在当前目录下生成 Test.class 文件, 再在适合位置手动创建一个名为 p1 的子目录, 将 Test.class 复制到该 p1 目录下。

采用简化的编译命令, 就是可以带包编译

```
javac -d destpath Test.java
```

归入该包的类的字节代码文件应放在 java 的类库所在路径的 destpath 子目录下。现在包的相对位置已经决定了, 但 java 类库的路径还是不定的。事实上, java 可以有多个存放类库的目录, 其中的缺省路径为 java 目录下的 lib 子目录, 你可以通过使用 -classpath 选项来确定你当前想选择的类库路径。除此之外, 你还可以在 CLASSPATH 环境变量中设置类库路径。destpath 为目标路径, 可以是本地的任何绝对或相对路径。则编译器会自动在 destpath 目录下建立一个子目录 p1, 并将生成的.class 文件自动保存到 destpath/p1 下。例如:

```
javac -d .\ Test.java
```

```
javac -d C:\test\ Test.java
```

## 2.4: 带包运行

运行带包的程序, 需要使用类的全路径, 也就是带包的路径, 比如上面的那个程序, 就使用如下的代码进行运行:

```
java p1.Test
```

## 3: import

---

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 语句之后, 所有类的定义之前, 可以有 0~多条, 其语法格式为:

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

java 运行时环境将到 CLASSPATH + package1.[package2...]路径下寻找并载入相应的字节码文件 classname.class。“\*”号为通配符, 代表所有的类。也就是说 import 语句为编译器指明了寻找类的途径。

例, 使用 import 语句引入类程序: TestPackage.java

```
import p1.Test; //或者 import p1.*;
public class TestPackage{
    public static void main(String args[]){
        Test t = new Test(); //Test 类在 p1 包中定义
        t.display();
    }
}
```



java 编译器默认为所有的 java 程序引入了 JDK 的 java.lang 包中所有的类 (import java.lang.\*;), 其中定义了一些常用类: System、String、Object、Math 等。因此我们可以直接使用这些类而不必显式引入。但使用其它非无名包中的类则必须先引入、后使用。

### 3.1: Java 类搜寻方式

程序中的 import 语句标明要引入 p1 包中的 Test 类, 假定环境变量 CLASSPATH 的值为 “. ; C:\jdk6\lib; D:\ex”, java 运行环境将依次到下述可能的位置寻找并载入该字节码文件 Test.class:

. \p1\Test.class

C:\jdk6\lib\p1\Test.class

D:\ex\p1\Test.class

其中, “.” 代表当前路径, 如果在第一个路径下就找到了所需的类文件, 则停止搜索。否则依次搜索后续路径, 如果在所有的路径中均未找到所需的类文件, 则编译或运行出错。

## 4: 访问修饰符

Java 语言允许对类中定义的各种属性和方法进行访问控制, 即规定不同的保护等级来限制对它们的使用。为什么要这样做? Java 语言引入类似访问控制机制的目的在于实现信息的封装和隐藏。Java 语言为对类中的属性和方法进行有效地访问控制, 将它们分为四个等级: private, 无修饰符, protected, public, 具体规则如下:

表 Java 类成员的访问控制

可否直接访问 控制等级	同一个类中	同一个包中	不同包中的 子类的对象	任何场合
private	Yes			
无修饰符	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

变量和方法可以处于四个访问级别中的一个: 公共, 受保护, 无修饰符或私有。类可以在公共或无修饰级别。

变量、方法或类有缺省 (无修饰符) 访问性, 如果它没有显式受保护修饰符作为它的声明的一部分的话。这种访问性意味着, 访问可以来自任何方法, 当然这些方法只能在作为对象的同一个包中的成员类当中。

以修饰符 protected 标记的变量或方法实际上比以缺省访问控制标记的更易访问。一个 protected 方法或变量可以从同一个包中的类当中的任何方法进行访问, 也可以是从任何子类中的任何方法进行访问。当它适合于一个类的子类但不是不相关的类时, 就可以使用这种受保护访问来访问成员。



## 5: 类定义

---

Java 程序的基本单位是类, 你建立类之后, 就可用它来建立许多你需要的对象。Java 把每一个可执行的成分都变成类。

类的定义形式如下:

```
<权限修饰符> [一般修饰符] class <类名> {  
    [<属性定义>]  
    [<构造方法定义>]  
    [<方法定义>]  
}
```

这里, 类名要是合法的标识符。在类定义的开始与结束处必须使用花括号。你也许想建立一个矩形类, 那么可以用如下代码:

```
public class Rectangle{  
    .....//矩形具体的属性和方法  
}
```

## 6: 构造方法

---

### 6.1: 什么是构造方法

类有一个特殊的成员方法叫作构造方法, 它的作用是创建对象并初始化成员变量。在创建对象时, 会自动调用类的构造方法。

### 6.2: 构造方法定义规则

Java 中的构造方法必须与该类具有相同的名字, 并且没有方法的返回类型 (包括没有 void)。另外, 构造方法一般都应用 public 类型来说明, 这样才能在程序任意的位置创建类的实例——对象。

### 6.3: 示例

下面是一个 Rectangle 类的构造方法, 它带有两个参数, 分别表示矩形的长和宽:

```
public class Rectangle{  
    public Rectangle(int w,int h) {  
        width=w;  
        height=h;  
    }  
}
```

### 6.4: 说明

每个类至少有一个构造方法。如果不写一个构造方法, Java 编程语言将提供一个默认的, 该构造方法没有参数, 而且方法体为空。

**注意:** 如果一个类中已经定义了构造方法则系统不再提供默认的构造方法。

## 7: 析构方法

---

析构方法 `finalize` 的功能是: 当对象被从内存中删除时, 该成员方法将会被自动调用。通常, 在析构方法内, 你可以填写用来回收对象内部的动态空间的代码。

特别注意: 当我们去调用析构方法的时候, 并不会引起该对象实例从内存中删除, 而是不会起到任何作用。

在 Java 编程里面, 一般不需要我们去写析构方法, 这里只是了解一下就可以了。

## 8: 属性

---

### 8.1: 属性是什么

简单点说, 属性就是对象所具有的静态属性。

### 8.2: 定义规则

Java 类中属性的声明采用如下格式:

访问修饰符 修饰符 类型 属性名称=初始值;

**访问修饰符:** 可以使用四种不同的访问修饰符中的一种, 包括 `public` (公共的)、`protected` (受保护的), 无修饰符和 `private` (私有的)。`public` 访问修饰符表示属性可以从任何其它代码调用。`private` 表示属性只可以由该类中的其它方法来调用。`protected` 将在以后的课程中讨论。

**修饰符:** 是对属性特性的描述, 例如后面会学习到的: `static`、`final` 等等。

**类型:** 属性的数据类型, 可以是任意的类型。

**属性名称:** 任何合法标识符

**初始值:** 赋值给属性的初始值。如果不设置, 那么会自动进行初始化, 基本类型使用缺省值, 对象类型自动初始化为 `null`。

### 8.3: 说明

属性有时候也被称为成员变量、实例变量、域, 它们经常被互换使用。

## 9: 方法

---

### 9.1: 方法是什么

方法就是对象所具有的动态功能。

### 9.2: 定义规则

Java 类中方法的声明采用以下格式:

访问修饰符 修饰符 返回值类型 方法名称 (参数列表) throws 异常列表 {方法体}

**访问修饰符:** 可以使用四种不同的访问修饰符中的一种, 包括 `public` (公共的)、`protected` (受保护的), 无修饰符和 `private` (私有的)。`public` 访问修饰符表示方法可以从任何其它代码调用。`private` 表示方法只可以由该类中的其它方法来调用。`protected` 将在以后的课程中讨论。

**修饰符:** 是对方法特性的描述, 例如后面会学习到的: `static`、`final`、`abstract`、`synchronized` 等等。

**返回值类型:** 表示方法返回值的类型。如果方法不返回任何值, 它必须声明为 `void` (空)。Java 技术对返回值是很严格的, 例如, 如果声明某方法返回一个 `int` 值, 那么方法必须从所有可能的返回路径中返回一个 `int` 值 (只能在等待返回该 `int` 值的上下文中被调用。)

**方法名称：**可以是任何合法标识符，并带有用已经使用的名称为基础的某些限制条件。

**参数列表：**允许将参数值传递到方法中。列举的元素由逗号分开，而每一个元素包含一个类型和一个标识符。在下面的方法中只有一个形式参数，用 int 类型和标识符 days 来声明：`public void test(int days) {}`

**throws 异常列表：**子句导致一个运行时错误（异常）被报告到调用的方法中，以便以合适的方式处理它。异常在后面的课程中介绍。

花括号内是方法体，即方法的具体语句序列。

### 9.3: 示例

比如现在有一个“车”的类——Car，“车”具有一些基本的属性，比如四个轮子，一个方向盘，车的品牌等等。当然，车也具有自己的功能，也就是方法，比如车能够“开动”——run。要想车子能够开动，需要给车子添加汽油，也就是说，需要为 run 方法传递一些参数“油”进去。车子跑起来过后，我们需要知道当前车辆运行的速度，就需要 run 方法具有返回值“当前的速度”。

```
package cn.javass.javatest;

public class Car { // 车这个类
    private String make; // 一个车的品牌
    private int tyre; // 一个车具有轮胎的个数
    private int wheel; // 一个车具有方向盘的个数

    public Car() {
        // 初始化属性
        make = "BMW"; // 车的品牌是宝马
        tyre = 4; // 一个车具有4个轮胎
        wheel = 1; // 一个车具有一个方向盘
    }
    /**
     * 车这个对象所具有的功能，能够开动
     * @param oil 为车辆加汽油的数量
     * @return 车辆当前运行的速度
     */
    public double run(int oil) {
        // 进行具体的功能处理
        return 200.0;
    }
}
```

### 9.4: 形参和实参

**形参：**就是形式参数的意思。是在定义方法名的时候使用的参数，用来标识方法接收的参数类型，在调用该方法时传入。

**实参：**就是实际参数的意思。是在调用方法时传递给该方法的实际参数。

比如：上面的例子中“int oil”就是个形式参数，这里只是表示需要加入汽油，这个方法才能正常运行，但具体加入多少，要到真正使用的时候，也就是调用这个方法的时候才具体确定，加入调用的时候传入“80”，这就是个实际参数。

形参和实参有如下基本规则:

- (1): 形参和实参的类型必须要一致, 或者要符合隐含转换规则
- (2): 形参类型不是引用类型时, 在调用该方法时, 是按值传递的。在该方法运行时, 形参和实参是不同的变量, 它们在内存中位于不同的位置, 形参将实参的值复制一份, 在该方法运行结束的时候形参被释放, 而实参内容不会改变。
- (3): 形参类型是引用类型时, 在调用该方法时, 是按引用传递的。运行时, 传给方法的是实参的地址, 在方法体内部使用的也是实参的地址, 即使用的就是实参本身对应的内存空间。所以在函数体内部可以改变实参的值。

### 9.5: 参数可变的方法

从 JDK5.0 开始, 提供了参数可变的方法。

当不能确定一个方法的入口参数的个数时, 5.0 以前版本的 Java 中, 通常的做法是将多个参数放在一个数组或者对象集合中作为参数来传递, 5.0 版本以前的写法是:

```
int sum(Integer[] numbers) {...}
//在别处调用该方法
sum(new Integer[] {12, 13, 20});
```

而在 5.0 版本中可以写为:

```
int sum(Integer... numbers) { //方法内的操作}
注意: 方法定义中是三个点
//在别处调用该方法
sum(12, 13, 20); //正确
sum(10, 11); //正确
```

也就是说, 传入参数的个数并不确定。但请注意: 传入参数的类型必须是一致的, 究其本质, 就是一个数组。

显然, JDK5.0 版本的写法更为简易, 也更为直观, 尤其是方法的调用语句, 不仅简化很多, 而且更符合通常的思维方式, 更易于理解。

## 四: 如何使用一个 Java 类

---

前面学习了如何定义一个类, 下面来学习如何使用一个类

### 1: new 关键字

---

假如定义了一个表示日期的类, 有三个整数变量: 日、月和年的意义即由这些整数变量给出。如下所示:

```
class MyDate {
    int day;
    int month;
    int year;
}
```

名称 MyDate 按照大小写的有关约定处理, 而不是由语意要求来定。

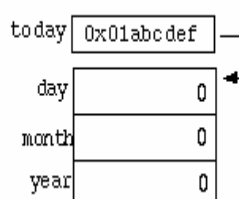
那么怎么来使用这个类呢: 在你可以使用变量之前, 实际内存必须被分配。这个工作是通过使用关键字 **new** 来实现的。如下所示:

在一个方法体中, 声明

```
MyDate today;  
today = new MyDate();
```

第一个语句(声明)仅为引用分配了足够的空间,而第二个语句则通过调用对象的构造方法为构成 MyDate 的三个整数分配了空间。对象的赋值使变量 today 重新正确地引用新的对象。这两个操作被完成后, MyDate 对象的内容则可通过 today 进行访问。

关键字 new 意味着内存的分配和初始化, new 调用的方法就是类的构造方法。



使用一个语句同时为引用 today 和由引用 today 所指的对象分配空间也是可能的。

```
MyDate today = new MyDate();
```

## 2: 如何使用对象中的属性和方法

---

要调用对象中的属性和方法, 使用 “.” 操作符。

例如:

```
today.day = 26;  
today.month = 7;  
today.year = 2008;
```

## 3: this 关键字

---

关键字 this 是用来指向当前对象或类实例的, 功能说明如下:

### 3.1: 点取成员

this.day 指的是调用当前对象的 day 字段, 示例如下:

```
public class MyDate {  
    private int day, month, year;  
    public void tomorrow() {  
        this.day = this.day + 1;  
        //其他代码  
    }  
}
```

Java 编程语言自动将所有实例变量和方法引用与 this 关键字联系在一起, 因此, 使用关键字在某些情况下是多余的。下面的代码与前面的代码是等同的。

```
public class MyDate {  
    private int day, month, year;  
    public void tomorrow() {  
        day = day + 1; // 在 day 前面没有使用 this  
        //其他代码  
    }  
}
```

### 3.2: 区分同名变量

也有关键字 `this` 使用不多余的情况。如, 需要在某些完全分离的类中调用一个方法, 并将当前对象的一个引用作为参数传递时。例如:

```
Birthday bDay = new Birthday (this);
```

还有一种情况, 就是在类属性上定义的变量和方法内部定义的变量相同的时候, 到底是调用谁呢? 例如:

```
public class Test{
    int i = 2;
    public void t(){
        int i = 3; //跟属性的变量名称是相同的
        System.out.println("实例变量 i=" + this.i);
        System.out.println("方法内部的变量 i=" + i);
    }
}
```

也就是说: “`this. 变量`” 调用的是当前属性的变量值, 直接使用变量名称调用的是相对距离最近的变量的值。

### 3.3: 作为方法名来初始化对象

也就是相当于调用本类的其它构造方法, 它必须作为构造方法的第一句。示例如下:

```
public class Test {
    public Test(){
        this(3); //在这里调用本类的另外的构造方法
    }
    public Test(int a){

    }
    public static void main(String[] args) {
        Test t = new Test();
    }
}
```

---

## 五: 引用类型

### 1: 引用类型是什么

一般引用类型 (reference type) 指向一个对象, 不是原始值, 指向对象的变量是引用变量。

在 Java 里面除去基本数据类型的其它类型都是引用数据类型。Java 程序运行时, 会为引用类型分配一定量的存储空间并解释该存储空间的内容。

示例如下:

```
public class MyDate{
    private int day=8;
    private int month=8;
    private int year=2008;
```

```
public MyDate(int day, int month, int year) {...}
public void print() {...}
}
public class TestMyDate{
    public static void main(String args[]) {
        MyDate today = new MyDate(23, 7, 2008); //这个 today 变量
                                                //就是一个引用类型的变量
    }
}
```

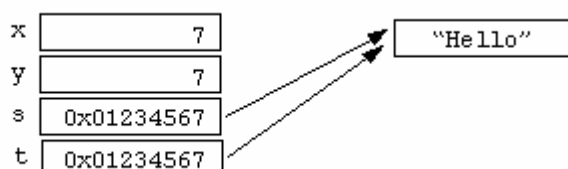
## 2: 引用类型的赋值

在 Java 编程语言中, 用类的一个类型声明的变量被指定为引用类型, 这是因为它正在引用一个非原始类型, 这对赋值具有重要的意义。请看下列代码片段:

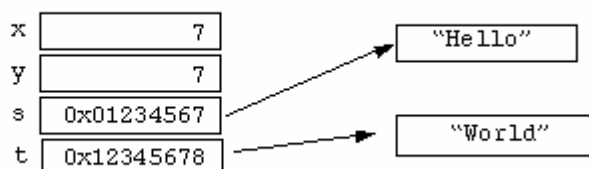
```
int x = 7;
int y = x;
String s = "Hello" ;
String t = s;
```

四个变量被创建: 两个原始类型 `int` 和两个引用类型 `String`。x 的值是 7, 而这个值被复制到 y; x 和 y 是两个独立的变量且其中任何一个的进一步的变化都不对另外一个构成影响。

至于变量 s 和 t, 只有一个 `String` 对象存在, 它包含了文本 "Hello", s 和 t 均引用这个单一的对象。



将变量 t 重新定义为: `t=" World"`; 则新的对象 World 被创建, 而 t 引用这个对象。上述过程被描述如下



## 3: 按值传递还是按引用传递

这个在 Java 里面是经常被提起的问题, 也有一些争论, 似乎最后还有一个所谓的结论: “在 Java 里面参数传递都是按值传递”。事实上, 这很容易让人迷惑, 下面先分别看看什么是按值传递, 什么是按引用传递, 只要能正确理解, 至于称作按什么传递就不是个大问题了。

### 3.1: 按值传递是什么

指的是在方法调用时，传递的参数是按值的拷贝传递。示例如下：

```
public class TempTest {  
    private void test1(int a){  
        //做点事情  
    }  
  
    public static void main(String[] args) {  
        TempTest t = new TempTest();  
        int a = 3;  
        t.test1(a); //这里传递的参数a就是按值传递  
    }  
}
```

按值传递重要特点：传递的是值的拷贝，也就是说传递后就互不相关了。

示例如下：

```
public class TempTest {  
    private void test1(int a){  
        a = 5;  
        System.out.println("test1方法中的a=="+a);  
    }  
  
    public static void main(String[] args) {  
        TempTest t = new TempTest();  
        int a = 3;  
        t.test1(a); //传递后，test1方法对变量值的改变不影响这里的a  
        System.out.println("main方法中的a=="+a);  
    }  
}
```

运行结果是：

```
test1方法中的a==5  
main 方法中的 a==3
```

### 3.2: 按引用传递是什么

指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引用的地址，也就是变量所对应的内存空间的地址。

示例如下：

```
public class TempTest {  
    private void test1(A a){  
  
    }  
  
    public static void main(String[] args) {  
        TempTest t = new TempTest();  
        A a = new A();  
        t.test1(a); //这里传递的参数a就是按引用传递  
    }  
}
```



```
class A{  
    public int age = 0;  
}
```

### 3.3: 按引用传递的重要特点

传递的是值的引用，也就是说传递前和传递后都指向同一个引用（也就是同一个内存空间）。

示例如下：

```
第1行 public class TempTest {  
第2行     private void test1(A a){  
第3行         a.age = 20;  
第4行         System.out.println("test1方法中的age="+a.age);  
第5行     }  
第6行     public static void main(String[] args) {  
第7行         TempTest t = new TempTest();  
第8行         A a = new A();  
第9行         a.age = 10;  
第10行        t.test1(a);  
第11行        System.out.println("main方法中的age="+a.age);  
第12行    }  
第13行 }  
第14行 class A{  
第15行     public int age = 0;  
第16行 }
```

运行结果如下：

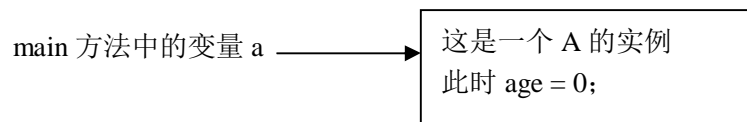
```
test1方法中的age=20  
main 方法中的 age=20
```

### 3.4: 理解按引用传递的过程——内存分配示意图

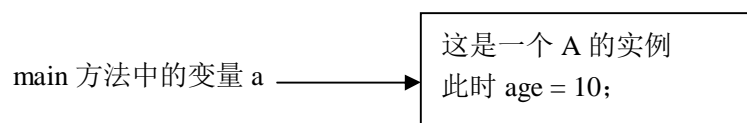
要想正确理解按引用传递的过程，就必须学会理解内存分配的过程，内存分配示意图可以辅助我们去理解这个过程。

用上面的例子来进行分析：

(1): 运行开始，运行第 8 行，创建了一个 A 的实例，内存分配示意如下：

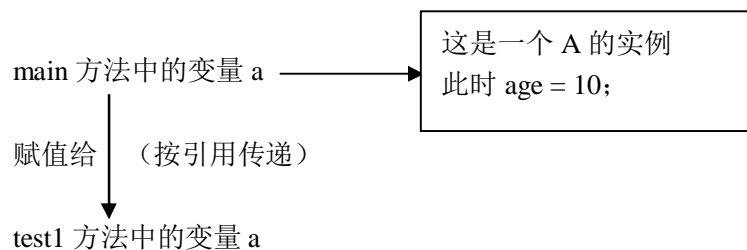


(2): 运行第 9 行，是修改 A 实例里面的 age 的值，运行后内存分配示意如下：

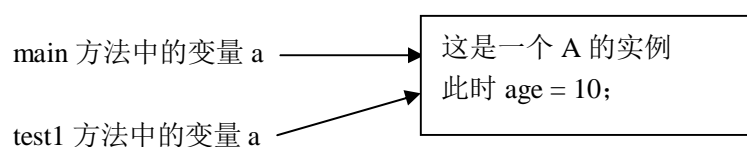


(3): 运行第 10 行, 是把 main 方法中的变量 a 所引用的内存空间地址, 按引用传递给 test1 方法中的 a 变量。请注意: 这两个 a 变量是完全不同的, 不要被名称相同所蒙蔽。

内存分配示意图如下:

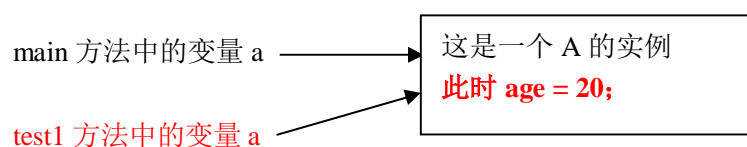


由于是按引用传递, 也就是传递的是内存空间的地址, 所以传递完成后形成的新的内存示意图如下:



也就是说: 是两个变量都指向同一个空间。

(4): 运行第 3 行, 为 test1 方法中的变量 a 指向的 A 实例的 age 进行赋值, 完成后形成的新的内存示意图如下:



此时 A 实例的 age 值的变化是由 test1 方法引起的

(5): 运行第 4 行, 根据此时的内存示意图, 输出 test1 方法中的 age=20

(6): 运行第 11 行, 根据此时的内存示意图, 输出 main 方法中的 age=20

### 3.5: 对上述例子的改变

理解了上面的例子, 可能有人会问, 那么能不能让按照引用传递的值, 相互不影响呢? 就是 test1 方法里面的修改不影响到 main 方法里面呢?

方法是在 test1 方法里面新 new 一个实例就可以了。改变成下面的例子, 其中第 3 行为新加的:

```
第1行 public class TempTest {
第2行     private void test1(A a){
第3行         a = new A(); //新加的一行
第4行         a.age = 20;
第5行         System.out.println("test1方法中的age="+a.age);
第6行     }
第7行     public static void main(String[] args) {
```

```
第8行      TempTest t = new TempTest();
第9行      A a = new A();
第10行     a.age = 10;
第11行     t.test1(a);
第12行     System.out.println("main方法中的age="+a.age);
第13行 }
第14行}
第15行class A{
第16行    public int age = 0;
第17行}
```

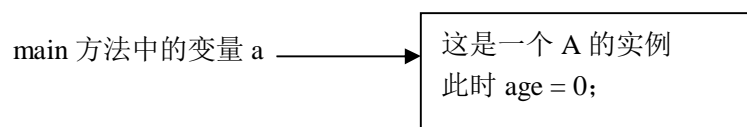
运行结果为：

```
test1方法中的age=20
main 方法中的 age=10
```

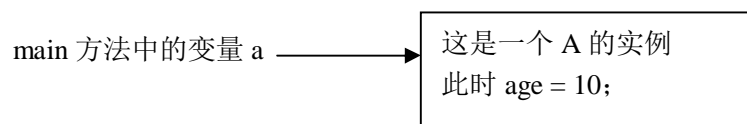
为什么这次的运行结果和前面的例子不一样呢，还是使用内存示意图来理解一下

### 3.6：再次理解按引用传递

(1)：运行开始，运行第 9 行，创建了一个 A 的实例，内存分配示意如下：

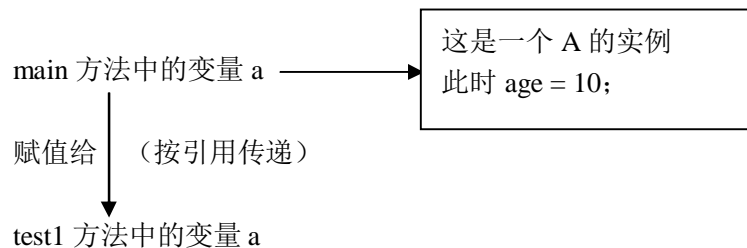


(2)：运行第 10 行，是修改 A 实例里面的 age 的值，运行后内存分配示意如下：



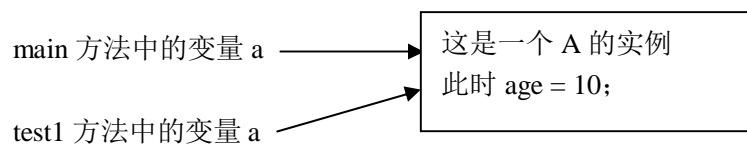
(3)：运行第 11 行，是把 main 方法中的变量 a 所引用的内存空间地址，按引用传递给 test1 方法中的 a 变量。请注意：这两个 a 变量是完全不同的，不要被名称相同所蒙蔽。

内存分配示意如下：



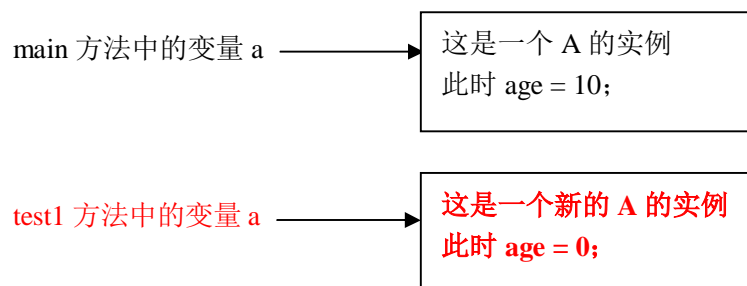
由于是按引用传递，也就是传递的是内存空间的地址，所以传递完成后形成的新的内存

示意图如下:

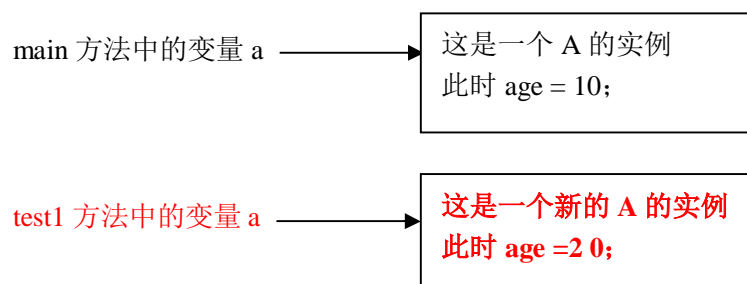


也就是说: 是两个变量都指向同一个空间。

(4): 运行第 3 行, 为 test1 方法中的变量 a 重新生成了新的 A 实例的, 完成后形成的新的内存示意图如下:



(5): 运行第 4 行, 为 test1 方法中的变量 a 指向的新的 A 实例的 age 进行赋值, 完成后形成的新的内存示意图如下:



注意: 这个时候 test1 方法中的变量 a 的 age 被改变, 而 main 方法中的是没有改变的。

(6): 运行第5行, 根据此时的内存示意图, 输出test1方法中的age=20

(7): 运行第 12 行, 根据此时的内存示意图, 输出 main 方法中的 age=10

### 3.7: 说明

(1): “在 Java 里面参数传递都是按值传递”这句话的意思是: 按值传递是传递的值的拷贝, 按引用传递其实传递的是引用的地址值, 所以统称按值传递。

(2): 在 Java 里面只有基本类型和按照下面这种定义方式的 String 是按值传递, 其它的都是按引用传递。就是直接使用双引号定义字符串方式: String str = “Java 私塾”;

## 六: 再谈变量

---

### 1: 实例变量和局部变量

---

在方法外定义的变量主要是实例变量, 它们是在使用 `new Xxx()` 创建一个对象时被分配内存空间的。每当创建一个对象时, 系统就为该类的所有实例变量分配存储空间; 创建多个对象就有多份实例变量。通过对象的引用就可以访问实例变量。

在方法内定义的变量或方法的参数被称为局部(local)变量, 有时也被用为自动(automatic)、临时(temporary)或栈(stack)变量。

方法参数变量定义在一个方法调用中传送的自变量, 每次当方法被调用时, 一个新的变量就被创建并且一直存在到程序的运行跳离了该方法。

当执行进入一个方法遇到局部变量的声明语句时, 局部变量被创建, 当执行离开该方法时, 局部变量被取消, 也就是该方法结束时局部变量的生命周期也就结束了。

因而, 局部变量有时也被引用为“临时或自动”变量。在成员方法内定义的变量对该成员变量是“局部的”, 因而, 你可以在几个成员方法中使用相同的变量名而代表不同的变量。该方法的应用如下所示:

```
public class Test {  
    private int i; // Test类的实例变量  
  
    public int firstMethod() {  
        int j = 1; // 局部变量  
        // 这里能够访问i和j  
        System.out.println("firstMethod 中 i="+i+",j="+j);  
        return 1;  
    } // firstMethod()方法结束  
  
    public int secondMethod(float f) { //method parameter  
        int j = 2; //局部变量, 跟firstMethod()方法中的j是不同的  
        // 这个j的范围是限制在secondMethod()中的  
        // 在这个地方, 可以同时访问i,j,f  
        System.out.println("secondMethod 中 i="+i+",j="+j+",f="+f);  
        return 2;  
    }  
  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.firstMethod();  
        t.secondMethod(3);  
    }  
}
```

### 2: 变量初始化

---

在 Java 程序中, 任何变量都必须经初始化后才能被使用。当一个对象被创建时, 实例变量在分配内存空间时按程序员指定的初始化值赋值, 否则系统将按下列默认值进行初始化:

byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
所有引用类型	null

注意—— 一个具有空值“null”的引用不引用任何对象。试图使用它引用的对象将会引起一个异常。异常是出现在运行时的错误, 这将在模块“异常”中讨论。

在方法外定义的变量被自动初始化。局部变量必须在使用之前做“手工”(由程序员进行)初始化。如果编译器能够确认一个变量在初始化之前可能被使用的情形, 编译器将报错。

```
public class Test {
    private int i; //Test类的实例变量
    public void test1() {
        int x = (int) (Math.random() * 100);
        int y;
        int z;
        if (x > 50) {
            y = 9;
        }
        z = y + x; // 将会引起错误, 因为y可能还没有被初始化就使用了
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.test1();
    }
}
```

### 3: 变量的范围(scope)

Java 变量的范围有四个级别: 类级、对象实例级、方法级、块级

(1): 类级变量又称全局级变量, 在对象产生之前就已经存在, 就是后面会学到的 static 变量。

(2): 对象实例级, 就是前面学到的属性变量

(3): 方法级: 就是在方法内部定义的变量, 就是前面学到的局部变量。

(4): 块级: 就是定义在一个块内部的变量, 变量的生存周期就是这个块, 出了这个块就消失了。

示例如下:

```
public class Test {
    private static String name="Java私塾"; //类级
    private int i; // 对象实例级, Test类的实例变量
    { //属性块, 在类初始化属性时候运行
        int j = 2; //块级
    }
}
```

```
}  
public void test1() {  
    int j = 3; //方法级  
    if(j==3){  
        int k = 5; //块级  
    }  
    //这里不能访问块级的变量, 块级变量只能在块内部访问  
    System.out.println("name="+name+", i="+i+", j="+j);  
}  
public static void main(String[] args) {  
    Test t = new Test();  
    t.test1();  
}  
}
```

运行结果:

name=Java 私塾, i=0, j=3

### 3.1: 访问说明

- (1): 方法内部除了能访问方法级的变量, 还可以访问类级和实例级的变量
- (2): 块内部能够访问类级、实例级变量, 如果块被包含在方法内部, 它还可以访问方法级的变量。
- (3) 变量当然是要在被访问前被定义和初始化, 不能访问后面才定义的变量。

## 七: 包装类

虽然 Java 语言是典型的面向对象编程语言, 但其中的 8 种基本数据类型并不支持面向对象的编程机制, 基本类型的数据不具备“对象”的特性---不携带属性、没有方法可调用。沿用它们只是为了迎合人类根深蒂固的习惯, 并的确能简单、有效地进行常规数据处理。

这种借助于非面向对象技术的做法有时也会带来不便, 比如引用类型数据均继承了 Object 类的特性, 要转换为 String 类型(经常有这种需要)时只要简单调用 Object 类中定义的 toString()即可, 而基本数据类型转换为 String 类型则要麻烦得多。为解决此类问题, Java 语言引入了封装类的概念, 在 JDK 中针对各种基本数据类型分别定义相应的引用类型, 并称之为包装类 (Wrapper Classes)。

下表描述了基本数据类型及对应的包装类

基本数据类型	对应的包装类
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double

每个包装类的对象可以封装一个相应的基本类型的数据, 并提供了其它一些有用的功能。包装类对象一经创建, 其内容(所封装的基本类型数据值)不可改变。

例, 包装类用法程序: Wrapper.java

```
public class Wrapper{
    public static void main(String args[]){
        int i = 500;
        Integer t = new Integer(i);
        int j = t.intValue();    // j = 500
        String s = t.toString(); // s = "500"
        System.out.println(t);
        Integer t1 = new Integer(500);
        System.out.println(t.equals(t1));
    }
}
```

程序运行结果为:

```
500
true
```

包装类一个常用的功能就是把字符串类型的数据造型成为对应的基本数据类型, 如下示例:

```
String str = "123";
int a = Integer.parseInt(str);
```

更过的功能还请查看 JDK 文档。

## 八: 类型转换

---

在赋值的信息可能丢失的地方, 编译器需要程序员用类型转换 (type cast) 的方法确认赋值。Java 中的类型转换分成: 强制类型转换、自动升级类型转换和后面将会学习到的向上造型。

### 1: 强制类型转换

---

把某种类型强制转换成另外一种类型就叫做强制类型转换。

例如, 可以将一个 long 值“挤压”到一个 int 变量中。显式转型做法如下:

```
long bigValue = 99L;
int squashed = (int) (bigValue);
```

在上述程序中, 期待的目标类型被放置在圆括号中, 并被当作表达式的前缀, 该表达式必须被更改。一般来讲, 建议用圆括号将需要转型的全部表达式封闭。否则, 转型操作的优先级可能引起问题。

**注意:** 强制类型转换只能用在原本就是某个类型, 但是被表示成了另外一种类型的时候, 可以把它强制转换回来。强制转换并不能在任意的类型间进行转换。

比如上面的例子: 99 这个数本来就是一个 int 的数, 但是它通过在后面添加 L 来表示成了一个 long 型的值, 所以它能够通过强制转换来转换回 int 类型。



## 2: 升级和表达式的类型转换

---

当没有信息丢失时, 变量可被自动升级为一个较长的形式 (如: int 至 long 的升级)

```
long bigval = 6; // 6 是 int 类型, OK
```

```
int smallval = 99L; // 99L 是 long 型, 非法
```

```
double z = 12.414F; // 12.414F 是 float 型, OK
```

```
float z1 = 12.414; // 12.414 是 double 型, 非法
```

一般来讲, 如果变量类型至少和表达式类型一样大 (位数相同), 则你可认为表达式是赋值兼容的。

## 3: 表达式的升级类型转换

---

对 + 运算符来说, 当两个操作数是原始数据类型时, 其结果至少有一个 int, 并且有一个通过提升操作数到结果类型, 或通过提升结果至一个较宽类型操作数而计算的值, 这可能会导致溢出或精度丢失。例如:

```
short a, b, c
a=1;
b=2;
c= a+b;
```

上述程序会出错是因为在执行 “+” 操作前, a 和 b 会从 short 提升至 int, 两个 int 相加的结果也是 int, 然后把一个 int 的值赋值给 c, 但是 c 是 short 型的, 所以出错。如果 c 被声明为一个 int, 或按如下操作进行类型转换:

```
c = (short) (a+b);
```

则上述代码将会成功通过。

尤其在四则运算表达式里面, 如果不强制进行类型转换, 那么运算最后的结果就是精度最高的那个操作数决定的。比如:

3\*5.0 的结果就是 double 型的, 应该定义成为: double a = 3 \* 5.0;

## 4: 自动包装和解包

---

自动包装: 就是把基础数据类型自动封装并转换成对应的包装类的对象。

自动解包: 就是把包装类的对象自动解包并转换成对应的基础数据类型。

示例如下:

```
public class Test {
    public static void main(String args[]) {
        Integer a1 = 5; // 自动包装
        int a2 = new Integer(5); // 自动解包
        System.out.println("a1="+a1+", a2="+a2);
    }
}
```

运行结果: a1=5, a2=5

## 九: Java 类的基本运行顺序

---

作为程序员, 应该对自己写的程序具备充分的掌控能力, 应该清楚程序的基本运行过程, 否则糊里糊涂的, 不利于对程序的理解和控制, 也不利于技术上的发展。

我们以下面的类来说明一个基本的 Java 类的运行顺序:

```
第1行 public class Test {  
第2行     private String name = "Java私塾";  
第3行     private int age = 2;  
第4行     public Test(){  
第5行         age = 1000; //期望能到1000年, 呵呵  
第6行     }  
第7行     public static void main(String[] args) {  
第8行         Test t = new Test();  
第9行         System.out.println(t.name+"的年龄是"+t.age+"年");  
第10行     }  
第11行 }
```

运行的基本顺序是:

- (1): 先运行到第 7 行, 这是程序的入口
- (2): 然后运行到第 8 行, 这里要 new 一个 Test, 就要调用 Test 的构造方法
- (3): 就运行到第 4 行, **注意:** 可能很多人觉得接下来就应该运行第 5 行了, 错! 初始化一个类, 必须先初始化它的属性
- (4): 因此运行到第 2 行, 然后是第 3 行
- (5): 属性初始化完过后, 才回到构造方法, 执行里面的代码, 也就是第 5 行
- (6): 然后是第 6 行, 表示 new 一个 Test 实例完成
- (7): 然后回到 main 方法中执行第 9 行
- (8): 然后是第 10 行

运行的结果是: Java 私塾的年龄是 1000 年

说明: 这里只是说明一个基本的运行过程, 没有考虑更多复杂的情况。

## 作业

---

- 1: 写一个 MyPoint 完全封装类，其中含有私有的 int 类型的 x 和 y 属性，分别用公有的 getX 和 setX、getY 和 setY 方法访问，定义一个 toString 方法用来显示这个对象的 x、y 的值，如显示 (1, 2)，最后用 main 方法测试。
- 2: 在 MyPoint 类中增加 equals()、toString() 方法，根据命令行参数个数测试：若不传参数，则显示 (0, 0)；若传一个参数，则打印 (此参数值, 0)；若传两个参数，则打印 (第一个参数值, 第二个参数值)。
- 3: 有一个序列，首两项为 0, 1，以后各项值为前两项值之和。写一个方法来实现求这个序列的和
- 4: 请编写一个方法实现如下功能：将 1 至 7 的数字转换为星期日到星期六的字符串。
- 5: 请编写一个方法实现如下功能：有任意三个整数 a, b, c, 请输出其中最大的
- 6: 请编写一个方法实现如下功能：将任意三个整数 a, b, c 按从小到大的顺序输出。
- 7: 请编写一个方法实现如下功能：用程序找出每位数的立方和等于该数本身值的所有的 3 位数。(水仙花数)
- 8: 请编写一个方法实现如下功能：计算 1 加到 n ( $n \geq 2$  的整数) 的总和。
- 9: 请编写一个方法实现如下功能：得到一个整数的绝对值。

## 第四章 Java 高级类特性

### 教学目标：

- 掌握 Java 中的继承
- 掌握方法的覆盖
- 掌握方法的重载
- 掌握 Java 中的多态
- 掌握 static 和 final 的特性和使用
- 了解内部类的构建和使用
- 理解 Java 的内存分配

## 一：Java 中的继承

---

### 1: extends 关键字

---

前面我们已经学习过了什么是继承，那么在 Java 里面如何来表达继承的关系呢，就是使用 extends 关键字，比如：经理这个类继承雇员这个类，示例如下：

```
public class Employee {
    String name;
    Date hireDate;
    Date dateOfBirth;
    String jobTitle;
    int grade;
    ...
}

public class Manager extends Employee {
    String department;
    Employee[] subordinates;
    ...
}
```

在这样的定义中，Manager 类被定义，具有 Employee 所拥有的所有变量及方法。所有这些变量和方法都是从父类的定义中继承来的。所有的程序员需要做的是定义额外特征或规定将适用的变化。

注意：这种方法是在维护和可靠性方面的一个伟大进步。如果在 Employee 类中进行修改，那么，Manager 类就会自动修改，而不需要程序员做任何工作，除了对它进行编译。

### 2: 初始化子类必先初始化父类

---

在 Java 编程语言中，对象的初始化是非常结构化的，这样做是为了保证安全。在前面的模块中，看到了当一个特定对象被创建时发生了什么。由于继承性，对象被完成，而且下述行为按顺序发生：

- (1) 存储空间被分配并初始化到 0 值
- (2) 进行显式初始化
- (3) 调用构造方法

(4) 层次中的每个类都会发生最后两个步骤，是从最上层开始。Java 技术安全模式要求在子类执行任何东西之前，描述父类的一个对象的各个方面都必须初始化。因此，Java 编程语言总是在执行子构造方法前调用父类构造方法的版本。

有继承的类在运行的时候，一定要记得：初始化子类必先初始化父类，这是 Java 程序的一个基本运行过程。比如：

```
第1行 public class Test extends Parent {
第2行     private String name = "Java私塾";
第3行     private int age = 2;
第4行     public Test() {
第5行         age = 1000; //期望能到1000年，呵呵
第6行     }
第7行     public static void main(String[] args) {
```

```
第8行      Test t = new Test();
第9行      System.out.println(t.name+"的年龄是"+t.age+"年");
第10行    }
第11行    }
第12行    class Parent{
第13行    private int num = 1;
第14行    public Parent(){
第15行      System.out.println("现在初始化父类");
第16行    }
第17行    public void test(){
第18行      System.out.println("这是父类的test方法");
第19行    }
第20行    }
```

上述类的基本运行顺序是:

- (1): 先运行到第 7 行, 这是程序的入口
- (2): 然后运行到第 8 行, 这里要 new 一个 Test, 就要调用 Test 的构造方法
- (3): 就运行到第 4 行, **注意: 初始化子类必先初始化父类**
- (4): 要先初始化父类, 所以运行到第 14 行
- (5): 然后是第 13 行, 初始化一个类, 必须先初始化它的属性
- (6): 然后是第 15 行
- (7): 然后是第 16 行, 表示父类初始化完成
- (8): 然后是回到子类, 开始初始化属性, 因此运行到第 2 行, 然后是第 3 行
- (9): 子类属性初始化完过后, 才回到子类的构造方法, 执行里面的代码, 也就是第 5 行
- (10): 然后是第 6 行, 表示 new 一个 Test 实例完成
- (11): 然后回到 main 方法中执行第 9 行
- (12): 然后是第 10 行

运行结果是:

现在初始化父类

Java 私塾的年龄是 1000 年

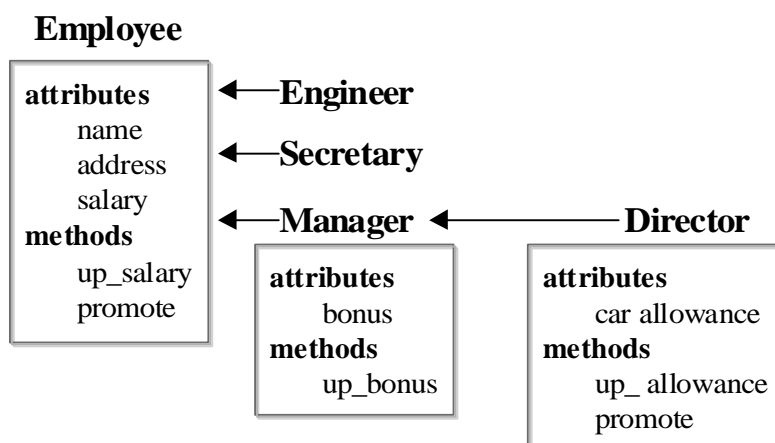
### 3: 单继承性

---

单继承性: 当一个类从一个唯一的类继承时, 被称做单继承性。单继承性使代码更可靠。接口提供多继承性的好处, 而且没有(多继承的)缺点。

Java 编程语言允许一个类仅能继承一个其它类, 即一个类只能有一个父类。这个限制被称做单继承性。单继承性与多继承性的优点是面向对象程序员之间广泛讨论的话题。Java 编程语言加强了单继承性限制而使代码更为可靠, 尽管这样有时会增加程序员的工作。后面会学到一个被叫做接口(interface)的语言特征, 它允许多继承性的大部分好处, 而不受其缺点的影响。

使用继承性的子类的一个例子如图所示:



#### 4: 构造方法不能被继承

尽管一个子类从父类继承所有的方法和变量, 但它不继承构造方法, 掌握这一点很重要。

一个类能得到构造方法, 只有两个办法。或者写构造方法, 或者根本没有写构造方法, 类有一个默认的构造方法。

#### 5: 关键字 super

关键字 super 可被用来引用该类的父类, 它被用来引用父类的成员变量或方法。父类行为被调用, 就好像该行为是本类的行为一样, 而且调用行为不必发生在父类中, 它能自动向上层类追溯。

**super 关键字的功能:**

- (1): 点取父类中被子类隐藏了的数据成员
- (2): 点取已经覆盖了的方法
- (3): 作为方法名表示父类构造方法

例如:

```
public class Employee {
    private String name;
    private int salary;
    public String getDetails() {
        return "Name: " + name + "\nSalary: " + salary;
    }
}

public class Manager extends Employee {
    private String department;

    public String getDetails() {
        return super.getDetails() + // 调用父类的方法
            "\nDepartment: " + department;
    }
}
```

请注意, `super.method()` 格式的调用, 如果对象已经具有父类类型, 那么它的方法的整个行为都将被调用, 也包括其所有负面效果。该方法不必在父类中定义, 它也可以从某些祖先类中继承。也就是说可以从父类的父类去获取, 具有追溯性, 一直向上去找, 直到找到为止, 这是一个很重要的特点。

## 6: 调用父类构造方法

---

在许多情况下, 使用默认构造方法来对父类对象进行初始化。

当然也可以使用 `super` 来显示调用父类的构造方法。

```
public class Employee {
    String name;
    public Employee(String n) {
        name = n;
    }
}

public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
        super(s);
        department = d;
    }
}
```

**注意:** 无论是 `super` 还是 `this`, 都必须放在构造方法的第一行。

通常要定义一个带参数的构造方法, 并要使用这些参数来控制一个对象的父类部分的构造。可能通过从子类构造方法的第一行调用关键字 `super` 的手段调用一个特殊的父类构造方法作为子类初始化的一部分。要控制具体的构造方法的调用, 必须给 `super()` 提供合适的参数。当不调用带参数的 `super` 时, 缺省的父类构造方法 (即, 带 0 个参数的构造方法) 被隐含地调用。在这种情况下, 如果没有缺省的父类构造方法, 将导致编译错误。

```
public class Employee {
    String name;
    public Employee(String n) {
        name = n;
    }
}

public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
        super(s); // 调用父类参数为 String 类型的构造方法
        department = d;
    }
}
```



当被使用时，super 或 this 必须被放在构造方法的第一行。显然，两者不能被放在一个单独行中，但这种情况事实上不是一个问题。如果写一个构造方法，它既没有调用 super(...) 也没有调用 this(...)，编译器自动插入一个调用到父类构造方法中，而不带参数。其它构造方法也能调用 super(...) 或 this(...)，调用一个 static 方法和构造方法的数据链。最终发生的是父类构造方法（可能几个）将在链中的任何子类构造方法前执行。

## 二：方法的覆盖和重载

---

### 1：方法的覆盖

---

#### 1.1：什么是方法的覆盖（Overridden Methods）

在类继承中，子类可以修改从父类继承来的行为，也就是说子类能创建一个与父类方法有不同功能的方法，但具有相同的：名称、返回类型、参数列表

如果在新类中定义一个方法，其名称、返回类型及参数表正好与父类中方法的名称、返回类型及参数相匹配，那么，新方法被称做覆盖旧方法。

#### 1.2：示例

如下在 Employee 和 Manager 类中的这些方法：

```
public class Employee {
    String name;
    int salary;

    public String getDetails() {
        return " Name: " + name + " \n " + "Salary: " + salary;
    }
}

public class Manager extends Employee {
    String department;

    public String getDetails() {
        return " Name: " + name + " \n " + " Manager of " + department;
    }
}
```

Manager 类有一个定义的 getDetails() 方法，因为它是从 Employee 类中继承的。基本的方法被子类的版本所代替或覆盖了。

#### 1.3：到底运行哪一个方法？

这里会给我们带来一个麻烦，父子类中有相同的方法，那么在运行时到底调用哪一个方法呢？假设下述方案：

```
Employee e = new Employee();
Manager m = new Manager();
```

如果请求 e.getDetails() 和 m.getDetails()，就会调用不同的行为。Employee 对象将执行与 Employee 有关的 getDetails 版本，Manager 对象将执行与 Manager 有关的 getDetails() 版本。

不明显的是如下所示:

```
Employee e = new Manager();  
e.getDetails();
```

或某些相似效果, 比如一个通用方法参数或一个来自异类集合的项。

事实上, 你得到与变量的运行时类型 (即, 变量所引用的对象的类型) 相关的行为, 而不是与变量的编译时类型相关的行为。这是面向对象语言的一个重要特征。它也是多态性的一个特征, 并通常被称作虚拟方法调用。

在前例中, 被执行的 `e.getDetails()` 方法来自对象的真实类型, `Manager`。

**因此规则是: 编译时看数据类型, 运行时看实际的对象类型 (new 操作符后跟的构造方法是哪个类的)。**一句话: **new 谁就调用谁的方法。**

#### 1.4: 覆盖方法的规则

记住, 子类的方法的名称以及子类方法参数的顺序必须与父类中的方法的名称以及参数的顺序相同, 以便该方法覆盖父类版本。下述规则适用于覆盖方法:

- (1): 覆盖方法的返回类型、方法名称、参数列表必须与它所覆盖的方法的相同。
- (2): 覆盖方法不能比它所覆盖的方法访问性差 (即访问权限不允许缩小)。
- (3): 覆盖方法不能比它所覆盖的方法抛出更多的异常。

这些规则源自多态性的属性和 Java 编程语言必须保证 “类型安全” 的需要。考虑一下这个无效方案:

```
public class Parent {  
    public void method() {  
    }  
}  
  
public class Child extends Parent {  
    private void method() { // 编译就会出错  
    }  
}  
  
public class Test {  
    public void otherMethod() {  
        Parent p1 = new Parent();  
        Parent p2 = new Child();  
        p1.method();  
        p2.method();  
    }  
}
```

Java 编程语言语义规定, `p2.method()` 导致方法的 `Child` 版本被执行, 但因为方法被声明为 `private`, `p2` (声明为 `Parent`) 不能访问它。于是, 语言语义冲突。

## 2: 方法的重载

---

假如你必须在不同情况下发送不同的信息给同一个成员方法的话, 该怎么办呢? 你可以通过对此成员方法说明多个版本的方法来实现重载。重载的本质是创建了一个新的成员方法: 你只需给它一个不同的参数列表。

## 2.1: 什么是重载

在同一个 Java 类中 (包含父类), 如果出现了方法名称相同, 而参数列表不同的情况就叫做重载。

参数列表不同的情况包括: 个数不同、类型不同、顺序不同等等。特别提示, 仅仅参数变量名称不同是不可以的。

## 2.2: 重载示例

如下例所示:

```
void getArea(int w, int h);  
void getArea(float w, float h);
```

在第二种情况下, 成员方法 `getArea()` 接受两个浮点变量作为它的参数, 编译器根据调用时的不同参数来决定该调用哪一种成员方法, 假如你把两个整数提供给成员方法, 就调用第一个成员方法; 假如你把两个浮点数提供给成员方法, 第二个成员方法就被调用。

当写代码来调用这些方法中的一个方法时, 便以其会根据提供的参数的类型来选择合适的方

**注意:** 跟成员方法一样, 构造方法也可以重载。

## 2.2: 方法的重载的规则

- (1): 方法名称必须相同
- (2): 参数列表必须不同 (个数不同, 或类型不同, 或参数排列顺序不同)。
- (3): 方法的返回类型可以相同也可以不相同。仅仅返回类型不同不足以成为方法的重载。

**注意:** 调用语句的参数表必须有足够的不同, 以至于允许区分出正确的方法被调用。正常的拓展晋升 (如, 单精度类型 `float` 到双精度类型 `double`) 可能被应用, 但是这样会导致在某些条件下的混淆。

## 2.3: 比较方法的覆盖和重载

重载方法:

在一个类 (或父子类) 中用相同的名字创建多个方法 (每个方法的参数表不同)

方法覆盖:

在一个类中创建的方法与父类中方法的名字、返回类型和参数表相同, 覆盖是针对两个类说的, 而且必须是子类 (或孙类, 孙孙类等) 覆盖掉父类的方法

# 三: Java 中的多态

---

## 1: 多态是什么

---

多态是同一个行为具有多个不同表现形式或形态的能力。

将经理描述成职员不只是描述这两个类之间的关系的一个简便方法。回想一下, 经理类具有父类职员类的所有属性、成员和方法。这就是说, 任何在 `Employee` 上的合法操作在 `Manager` 上也合法。如果 `Employee` 有 `raiseSalary()` 和 `fire()` 两个方法, 那么 `Manager` 类也有。

在这种 Manager 继承 Employee 的情况下, 一个 Employee 既可以是一个普通的 Employee 类, 也可以是一个 Manager 类。也就是说下述表示都是对的:

```
Employee e = new Employee();  
Employee e = new Manager();
```

从上面可以看到: 同一个行为 Employee 具有多个不同的表现形式 (既可以是一个普通的 Employee 类, 也可以是一个 Manager 类), 这就被称为多态。

**注意:** 方法没有多态的说法, 严格说多态是类的特性。但是也有对方法说多态的, 了解一下, 比如前面学到的方法覆盖称为动态多态, 是一个运行时问题; 方法重载称为静态多态, 是一个编译时问题。

## 2: 多态与类型

---

一个对象只有一个格式 (是在构造时给它的)。但是, 既然变量能指向不同格式的对象, 那么变量就是多态性的。也就是说一个对象只有一种形式, 但一个变量却有多种不同形式。

象大多数面向对象语言一样, Java 实际上允许父类类型的引用变量指向一个子类的对象。因此, 可以说:

```
Employee e = new Manager();
```

使用变量 e 是因为, 你能访问的对象部分只是 Employee 的一个部分; Manager 的特殊部分是隐藏的。这是因为编译器应意识到, e 是一个 Employee, 而不是一个 Manager。因而, 下述情况是不允许的:

```
e.department = " Finance "; //非法的, 编译时会出错
```

可能有的人会不理解, 为什么明明是 new 的一个 Manager, 却不能访问 Manager 的属性数据。原因在于编译的时候, 变量 e 是一个 Employee 的类型, 编译器并不去管运行时 e 指向的具体对象是一个 Employee 的对象, 还是一个 Manager 的对象, 所以它只能访问到 Employee 里面定义的属性和方法。所以说编译时看数据类型。

那么要想访问到 Manager 里面的 department 该怎么办呢? 这就需要先对 e 进行强制类型转换, 把它还原成为 Manager 类型, 就可以访问到 Manager 里面的属性和方法了, 如下:

```
Employee e = new Manager();  
Manager m = (Manager)e;  
m.department = "开发部"; //这就是合法的了
```

## 3: instanceof 运算符

---

多态性带来了一个问题: 如何判断一个变量所实际引用的对象的类型。C++ 使用 runtime-type information (RTTI), Java 使用 instanceof 操作符。

**instanceof 运算符功能:** 用来判断某个实例变量是否属于某种类的类型。一旦确定了变量所引用的对象的类型后, 可以将对象恢复给对应的子类变量, 以获取对象的完整功能。示例如下:

```
public class Employee extends Object  
public class Manager extends Employee  
public class Contractor extends Employee
```

如果通过 Employee 类型的引用接受一个对象, 它变不变成 Manager 或 Contractor 都可

以。可以象这样用 instanceof 来测试:

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee

public void method(Employee e) {
    if (e instanceof Manager) {
        //如果雇员是经理, 可以做的事情写在这里
    }else if (e instanceof Contractor) {
        //如果雇员是普通的职员, 可以做的事情写在这里
    }else {
        //说明是临时雇员, 可以做的事情写在这里
    }
}
```

#### 4: 多态对象的类型转换

---

在你接收父类的一个引用时, 你可以通过使用 instanceof 运算符判定该对象实际上是你所要的子类, 并可以用类型转换该引用的办法来恢复对象的全部功能。

```
public void method(Employee e) {
    if (e instanceof Manager) {
        Manager m = (Manager)e;
        System.out.println( " This is the manager of " + m.department);
    }
    // rest of operation
}
```

如果不用强制类型转换, 那么引用 e.department 的尝试就会失败, 因为编译器不能将被称做 department 的成员定位在 Employee 类中。

如果不用 instanceof 做测试, 就会有类型转换失败的危险。通常情况下, 类型转换一个对象引用的尝试是要经过几种检查的:

向上强制类型转换类层次总是允许的, 而且事实上不需要强制类型转换运算符。可由简单的赋值实现。

严格讲不存在向下类型转换, 其实就是强制类型转换, 编译器必须满足类型转换至少是可能的这样的条件。比如, 任何将 Manager 引用类型转换成 Contractor 引用的尝试是肯定不允许的, 因为 Contractor 不是一个 Manager。类型转换发生的类必须是当前引用类型的子类。

如果编译器允许类型转换, 那么, 该引用类型就会在运行时被检查。比如, 如果 instanceof 检查从源程序中被省略, 而被类型转换的对象实际上不是它应被类型转换进去的类型, 那么, 就会发生一个运行时错误(exception)。异常是运行时错误的一种形式, 而且是后面章节的主题。

#### 5: 动态绑定

---

**绑定:** 将一个方法调用同一个方法主体连接到一起就称为“绑定”(Binding)。

**动态绑定:** 当给对象发送请求时, 所引起的具体操作既与请求本身有关又与接受对象有关。支持相同请求的不同对象可能对请求激发的操作有不同的实现。发送给对象的请求和它

的相应操作在运行时刻的连接就称之为动态绑定(dynamic binding)。

动态绑定是指发送的请求直到运行时刻才受你的具体的实现的约束。因而, 在知道任何有正确接口的对象都将接受此请求时, 你可以写一个一般的程序, 它期待着那些具有该特定接口的对象。进一步讲, 动态绑定允许你在运行时刻彼此替换有相同接口的对象。这种可替换性就称为多态(polymorphism), 它是面向对象系统中的核心概念之一。多态允许客户对象仅要求其它对象支持特定接口, 除此之外对其假设几近于无。多态简化了客户的定义, 使得对象间彼此独立, 并可以在运行时刻动态改变它们相互的关系。

若在程序运行以前执行绑定(由编译器和链接程序, 如果有的话), 就叫作“早期绑定”。但是在只有一个 `Instrument` 句柄的前提下, 编译器不知道具体该调用哪个方法。

解决的方法就是“后期绑定”, 它意味着绑定在运行期间进行, 以对象的类型为基础。后期绑定也叫作“动态绑定”或“运行期绑定”。若一种语言实现了后期绑定, 同时必须提供一些机制, 可在运行期间判断对象的类型, 并分别调用适当的方法。也就是说, 编译器此时依然不知道对象的类型, 但方法调用机制能自己去调查, 找到正确的方法主体。不同的语言对后期绑定的实现方法是有所区别的。但我们至少可以这样认为: 它们都要在对象中安插某些特殊类型的信息。

Java 中绑定的所有方法都采用后期绑定技术, 除非一个方法已被声明成 `final`。这意味着我们通常不必决定是否应进行后期绑定——它是自动发生的。

## 四: static

---

### 1: static 修饰符

---

`static` 修饰符能够与属性、方法和内部类一起使用, 表示是“静态”的。

类中的静态变量和静态方法能够与“类名”一起使用, 不需要创建一个类的对象来访问该类的静态成员。所以 `static` 修饰的变量又称作“类变量”。这与实例变量不同。实例变量总是用对象来访问, 因为它们的值在对象和对象之间有所不同。

下列示例展示了如何访问一个类的静态变量:

```
class StaticModifier {
    static int i = 10;
    int j;

    StaticModifier() {
        j = 20;
    }
}

public class Test {
    public static void main(String args[]) {
        System.out.println("类变量 i=" + StaticModifier.i);
        StaticModifier s = new StaticModifier();
        System.out.println("实例变量 j=" + s.j);
    }
}
```

上述程序的输出是:



类变量 i=10

实例变量 j=20

## 2: static 属性的内存分配

---

在上面的例子中，无需创建类的对象即可访问静态变量 i。之所以会产生这样的结果，是因为编译器只为整个类创建了一个静态变量的副本，因此它能够用类名进行访问。也就是说：一个类中，一个 **static** 变量只有一个内存空间，虽然有多个类实例，但这些类实例中的这个 **static** 变量会共享同一个内存空间。示例如下：

```
public class Test{
    static UserModel um = new UserModel();
    public static void main(String[] args) {
        Test t1 = new Test();
        t1.um.userName = "张三";
        Test t2 = new Test();
        t2.um.userName = "李四";

        System.out.println("t1.um.userName==" + t1.um.userName);
        System.out.println("t2.um.userName==" + t2.um.userName);
    }
}

class UserModel{
    public String userName="";
}
```

运行结果：

```
t1.um.userName==李四
t2.um.userName==李四
```

为什么会是一样的值呢？就是因为多个实例中的静态变量 **um** 是共享同一内存空间，**t1.um** 和 **t2.um** 其实指向的都是同一个内存空间，所以就得到上面的结果了。

要想看看是不是 **static** 导致这样的结果，你可以尝试去掉 **UserModel** 前面的 **static**，然后再试一试，看看结果，应该如下：

```
t1.um.userName==张三
t2.um.userName==李四
```

还有一点也很重要：**static** 的变量是在类装载的时候就会被初始化。也就是说，只要类被装载，不管你是否使用了这个 **static** 变量，它都会被初始化。

小结一下：类变量（**class variables**）用关键字 **static** 修饰，在类加载的时候，分配类变量的内存，以后在生成类的实例对象时，将共享这块内存（类变量），任何一个对象对类变量的修改，都会影响其它对象。外部有两种访问方式：通过对象来访问或通过类名来访问。

### 3: static 的基本规则

---

有关静态变量或方法的一些要点如下:

- | 一个类的静态方法只能访问静态属性
- | 一个类的静态方法不能够直接调用非静态方法
- | 如访问控制权限允许, static 属性和方法可以使用对象名加 ‘.’ 方式调用; 当然也可以使用实例加 ‘.’ 方式调用
- | 静态方法中不存在当前对象, 因而不能使用 “this”, 当然也不能使用 “super”;
- | 静态方法不能被非静态方法覆盖;
- | 构造方法不允许声明为 static 的

static 方法可以用类名而不是引用来访问, 如:

```
public class GeneralFunction {
    public static int addUp(int x, int y) {
        return x + y;
    }
}

public class UseGeneral {
    public void method() {
        int a = 9;
        int b = 10;
        int c = GeneralFunction.addUp(a, b);
        System.out.println("addUp() gives " + c);
    }
}
```

因为 static 方法不需它所属的类的任何实例就会被调用, 因此没有 this 值。结果是, static 方法不能访问与它本身的参数以及 static 变量之外的任何变量, 访问非静态变量的尝试会引起编译错误。

注: 非静态变量只限于实例, 并只能通过实例引用被访问。

### 4: 静态初始器——静态块

---

#### 4.1: 什么是静态初始器

静态初始器 (Static Initializer) 是一个存在与类中方法外面的静态块。静态初始器仅仅在类装载的时候 (第一次使用类的时候) 执行一次。

静态初始器的功能是: 往往用来初始化静态的类属性。

#### 4.2: 示例

```
class Count {
    public static int counter;
    static { // 只运行一次
        counter = 123;
        System.out.println("Now in static block.");
    }
}
```



```
    }  
    public void test(){  
        System.out.println("test method==" + counter);  
    }  
}  
public class Test {  
    public static void main(String args[]) {  
        System.out.println("counter=" + Count.counter);  
        new Count().test();  
    }  
}
```

运行结果是:

```
Now in static block.  
counter=123  
test method==123
```

## 5: 静态 import

---

当我们要获取一个随机数时，写法是:

```
public class Test {  
    public static void main(String[] args) {  
        double randomNum = Math.random();  
        System.out.println("the randomNum==" + randomNum);  
    }  
}
```

从 JDK5.0 开始可以写为:

```
import static java.lang.Math.random;  
public class Test {  
    public static void main(String[] args) {  
        double randomNum = random();  
        System.out.println("the randomNum==" + randomNum);  
    }  
}
```

静态引用使我们可以象调用本地方法一样调用一个引入的方法, 当我们需要引入同一个类的多个方法时, 只需写为 “import static java.lang.Math.\*” 即可。这样的引用方式对于枚举也同样有效。

## 五: final

---

### 1: final 修饰符

在 Java 中声明类、属性和方法时, 可使用关键字 final 来修饰。final 所标记的成分具有“终态”的特征, 表示“最终的”意思。

### 2: final 的规则

其具体规定如下:

- | final 标记的类不能被继承。
- | final 标记的方法不能被子类重写。
- | final 标记的变量（成员变量或局部变量）即成为常量，只能赋值一次。
- | final 标记的成员变量必须在声明的同时赋值，如果在声明的时候没有赋值，那么只有一次赋值的机会，而且只能在构造方法中显式赋值，然后才能使用。
- | final 标记的局部变量可以只声明不赋值，然后再进行一次性的赋值。
- | final 一般用于标记那些通用性的功能、实现方式或取值不能随意被改变的成分，以避免被误用，

例如实现数学三角方法、幂运算等功能的方法，以及数学常量  $\pi=3.141593$ 、 $e=2.71828$  等。事实上，为确保这终态性，提供了上述方法和常量的 `java.lang.Math` 类也已被定义为 `final` 的。

需要注意的是，如果将引用类型（即，任何类的类型）的变量标记为 `final`，那么该变量不能指向任何其它对象。但可以改变对象的内容，因为只有引用本身是 `final` 的。

如果变量被标记为 `final`，其结果是使它成为常数。想改变 `final` 变量的值会导致一个编译错误。下面是一个正确定义 `final` 变量的例子：

```
public final int MAX_ARRAY_SIZE = 25;
```

例 final 关键字程序：Test.java

```
public final class Test{
    public static final int TOTAL_NUMBER= 5 ;
    public int id;
    public Test(){
        id = ++TOTAL_NUMBER; //非法，对final变量TOTAL_NUMBER进行二次赋值了。
        //因为++TOTAL_NUMBER相当于：TOTAL_NUMBER=TOTAL_NUMBER+1
    }
    public static void main(String[] args) {
        final Test t = new Test();
        final int i= 10;
        final int j;
        j = 20;
        j = 30; //非法，对final变量进行二次赋值
    }
}
```

Java 编程语言允许关键字 `final` 被应用到类上（放在 `class` 关键字前面）。如果这样做了，类便不能被再派生出子类。比如，类 `Java.lang.String` 就是一个 `final` 类。这样做是出于安全原因，因为它保证，如果方法有字符串的引用，它肯定就是类 `String` 的字符串，而不是某个其它类的字符串，这个类是 `String` 的被修改过的子类，因为 `String` 可能被恶意篡改过。

方法也可以被标记为 `final`。被标记为 `final` 的方法不能被覆盖。这是由于安全原因。如果方法具有不能被改变的实现，而且对于对象的一致状态是关键的，那么就要使方法成为

final。被声明为 final 的方法有时被用于优化。编译器能产生直接对方法调用的代码, 而不是通常的涉及运行时查找的虚拟方法调用。

## 六: 内部类

---

### 1: 什么是内部类

---

内部类 (Inner Classes) 的概念是在 JDK1.1 版本中开始引入的。在 Java 中, 允许在一个类 (或方法、语句块) 的内部定义另一个类, 后者称为内部类, 有时也称为嵌套类 (Nested Classes)。内部类和外层封装它的类之间存在逻辑上的所属关系, 一般只用在定义它的类或语句块之内, 实现一些没有通用意义的功能逻辑, 在外部引用它时必须给出完整的名称。

引入内部类的好处在于可使源代码更加清晰并减少类的命名冲突, 就好比工厂制定内部通用的产品或工艺标准, 可以取任何名称而不必担心和外界的标准同名, 因为其使用范围不同。内部类是一个有用的特征, 因为它们允许将逻辑上同属性的类组合到一起, 并在另一个类中控制一个类的可视性。

下述例子表示使用内部类的共同方法:

```
public class Test {
    public static void main(String[] args) {
        MyFrame mf = new MyFrame("Java私塾");
    }
}

class MyFrame extends Frame {
    Button myButton;
    TextArea myTextArea;
    int count;

    public MyFrame(String title) {
        super(title);
        myButton = new Button("click me");
        myTextArea = new TextArea();
        add(myButton, BorderLayout.CENTER);
        add(myTextArea, BorderLayout.NORTH);
        ButtonListener bList = new ButtonListener();
        myButton.addActionListener(bList);
    }

    class ButtonListener implements ActionListener // 这里定义了一个内部类
    {
        public void actionPerformed(ActionEvent e) {
            count++;
            myTextArea.setText("button clicked" + count + "times");
        }
    } // end of innerclass ButtonListener
}
```

```
public static void main(String args[]) {  
    MyFrame f = new MyFrame("Inner Class Frame");  
    f.setSize(300, 300);  
    f.setVisible(true);  
}  
}
```

前面的例子包含一个类 `MyFrame`, 它包括一个内部类 `ButtonListener`。编译器生成一个类文件, `MyFrame$ButtonListener.class` 以及 `MyFrame.class`。它包含在 `MyFrame.class` 中, 是在类的外部创建的。

## 2: 内部类特点

---

(1): 嵌套类 (内部类) 可以体现逻辑上的从属关系。同时对于其他类可以控制内部类对外不可见等

(2): 外部类的成员变量作用域是整个外部类, 包括嵌套类。但外部类不能访问嵌套类的 `private` 成员

(3): 逻辑上相关的类可以在一起, 可以有效的实现信息隐藏。

(4): 内部类可以直接访问外部类的成员。可以用此实现多继承!

(5): 编译后, 内部类也被编译为单独的类, 不过名称为 `outclass$inclass` 的形式。

再来个例子:

```
public class Outer {  
    private int size;  
    public class Inner {  
        private int counter = 10;  
        public void doStuff() { size++; }  
    }  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        Inner inner = outer.new Inner();  
        inner.doStuff();  
        System.out.println(outer.size);  
        System.out.println(inner.counter);  
        // System.out.println(counter); 编译错误, 外部类不能访问内部类的 private 变量  
    }  
}
```

## 3: 内部类的分类

---

内部类按照使用上可以分为四种情形:

- (1): 类级: 成员式, 有 `static` 修饰
- (2): 对象级: 成员式, 普通, 无 `static` 修饰
- (3): 本地内部类: 局部式
- (4): 匿名级: 局部式

## 4: 成员式内部类

---

内部类可以作为外部类的成员，示例如下：

```
public class Outer1{
    private int size;
    public class Inner{
        public void dostuff(){
            size++;
        }
    }
    public void testTheInner(){
        Inner in=new Inner();
        in.dostuff();
    }
}
```

成员式内部类如同外部类的一个普通成员。

#### 4.1：成员式内部类的基本规则

(1)：可以有各种修饰符，可以用 4 种权限、static、final、abstract 定义（这点和普通的类是不同的）；

(2)：若有 static 限定，就为类级，否则为对象级。类级可以通过外部类直接访问；对象级需要先生成外部的对象后才能访问。

(3)：内外部类不能同名

(4)：非静态内部类中不能声明任何 static 成员

(5)：内部类可以互相调用，如下：

```
class A {
    class B { } //B、C 间可以互相调用
    class C { }
}
```

#### 4.2：成员式内部类的访问

内部类的对象以属性的方式记录其所依赖的外层类对象的引用，因而可以找到该外层类对象并访问其成员。该属性是系统自动为非 static 的内部类添加的，名称约定为“外层类名.this”。

在其它场合则必须先获得外部类的对象，再由外部类对象加“.new”操作符调用内部类的构造方法创建内部类的对象，此时依赖关系的双方也可以明确。这样要求是因为：外部类的 static 方法中不存在当前对象，或者其它无关类中方法的当前对象类型不符合要求。

(1)：在另一个外部类中使用非静态内部类中定义的方法时，要先创建外部类的对象，再创建与外部类相关的内部类的对象，再调用内部类的方法，如下所示：

```
class Outer2{
    private int size;
    class Inner{
        public void dostuff(){ size++; }
    }
}
```

```
class TestInner{
    public static void main(String[] args){
        Outer2 outer = new Outer2();
        Outer2.Inner inner=outer.new Inner();
        inner.dostuff();
    }
}
```

(2): static 内部类相当于其外部类的 static 成分, 它的对象与外部类对象间不存在依赖关系, 因此可直接创建。示例如下:

```
class Outer2 {
    private static int size;
    static class Inner {
        public void dostuff() {
            size++;
            System.out.println("size="+size);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer2.Inner inner = new Outer2.Inner();
        inner.dostuff();
    }
}
```

程序运行结果为:

```
size=1
```

(3): 由于内部类可以直接访问其外部类的成分, 因此当内部类与其外部类中存在同名属性或方法时, 也将导致命名冲突。所以在多层调用时要指明, 如下所示:

```
public class Outer3{
    private int size;
    public class Inner{
        private int size;
        public void dostuff (int size) {
            size++;           //本地的 size;
            this.size;        //内部类的 size
            Outer3.this.size++; //外部类的 size
        }
    }
}
```

## 5: 本地内部类

---

本地类(Local class)是定义在代码块中的类,它们只在定义它们的代码块中是可见的。

本地类有几个重要特性:

- (1): 仅在定义了它们的代码块中是可见的;
- (2): 可以使用定义它们的代码块中的任何本地 final 变量;
- (3): 本地类不可以是 static 的, 里边也不能定义 static 成员。
- (4): 本地类不可以用 public、private、protected 修饰, 只能使用缺省的。
- (5): 本地类可以是 abstract 的。

示例如下:

```
public final class Outter{
    public static final int TOTAL_NUMBER= 5 ;
    public int id=123;

    public void t1(){
        final int a = 15;
        String s = "t1";

        class Inner{
            public void innerTest(){
                System.out.println(TOTAL_NUMBER);
                System.out.println(id);
                System.out.println(a);
                //System.out.println(s);不合法, 只能访问本地方法的final变量
            }
        }
        new Inner().innerTest();
    }

    public static void main(String[] args) {
        Outter t = new Outter ();
        t.t1();
    }
}
```

## 6: 匿名内部类

---

### 6.1: 匿名内部类是什么

匿名内部类是本地内部类的一种特殊形式, 也就是没有类名的内部类, 而且具体的类实现会写在这个内部类里面。

把上面的例子改造一下, 如下所示:

```
public final class Test{
    public static final int TOTAL_NUMBER= 5 ;
    public int id=123;

    public void t1(){
```

```
final int a = 15;
String s = "t1";

new Aclass(){
    public void testA(){
        System.out.println(TOTAL_NUMBER);
        System.out.println(id);
        System.out.println(a);
        //System.out.println(s);不合法, 只能访问本地方法的final变量
    }
}.testA();
}
public static void main(String[] args) {
    Test t = new Test();
    t.t1();
}
}
class Aclass{
    public void testA(){}
}
```

注意: 匿名内部类是在一个语句里面, 所以后面需要加 “;”。

## 6.2: 匿名类的规则

- (1): 匿名类没有构造方法;
- (2): 匿名类不能定义静态的成员;
- (3): 匿名类不能用 4 种权限、static、final、abstract 修饰;
- (4): 只可以创建一个匿名类实例

再次示例:

```
public class Outter{
    public Contents getCont() {
        return new Contents() {
            private int i = 11;
            public int value() { return i; }
        };
    }
    public static void main(String[] args) {
        Outter p = new Outter ();
        Contents c = p. getCont ();
    }
}
```

## 7: 内部类规则小结

---

总结一下, 内部类有如下特点:



(1): 类名称只能用在定义过的范围中，除非用在限定的名称中。内部类的名称必须与所嵌套的类不同。

(2): 内部类可以被定义在方法中。这条规则较简单，它支配到所嵌套类方法的变量的访问。任何变量，不论是本地变量还是正式参数，如果变量被标记为 `final`，那么，就可以被内部类中的方法访问。

(3): 内部类可以使用所嵌套类的类变量和实例变量以及所嵌套的块中的本地变量。

(4): 内部类可以被定义为 `abstract`。

(5): 只有内部类可以被声明为 `private` 或 `protected`，以便防护它们不受来自外部类的访问。访问保护不阻止内部类使用其它类的任何成员，只要一个类嵌套另一个。

(6): 一个内部类可以作为一个接口，由另一个内部类实现。

(7): 被自动地声明为 `static` 的内部类成为顶层类。这些内部类失去了在本地范围和其它内部类中使用数据或变量的能力。

(8): 内部类不能声明任何 `static` 成员；只有顶层类可以声明 `static` 成员。因此，一个需求 `static` 成员的内部类必须使用来自顶层类的成员。

## 七：再谈 Java 内存分配

---

Java 程序运行时的内存结构分成：方法区、栈内存、堆内存、本地方法栈几种。

栈和堆都是数据结构的知识，如果不清楚，没有关系，就当成一个不同的名字就好了，下面的讲解不需要用到它们具体的知识。

### 1: 方法区

---

方法区存放装载的类数据信息包括：

(1)：基本信息：

- 1) 每个类的全限定名
- 2) 每个类的直接超类的全限定名(可约束类型转换)
- 3) 该类是类还是接口
- 4) 该类型的访问修饰符
- 5) 直接超接口的全限定名的有序列表

(2)：每个已装载类的详细信息：

1) 运行时常量池：

存放该类型所用的一切常量(直接常量和对其它类型、字段、方法的符号引用)，它们以数组形式通过索引被访问，是外部调用与类联系及类型对象化的桥梁。它是类文件(字节码)常量池的运行时表示。(还有一种静态常量池，在字节码文件中)。

2) 字段信息：

类中声明的每一个字段的信息(名，类型，修饰符)。

3) 方法信息：

类中声明的每一个方法的信息(名，返回类型，参数类型，修饰符，方法的字节码和异常表)。

4) 静态变量

5) 到类 classloader 的引用: 即到该类的类装载器的引用。

6) 到类 class 的引用:

虚拟机为每一个被装载的类型创建一个 class 实例, 用来代表这个被装载的类。

## 2: 栈内存

---

Java 栈内存以帧的形式存放本地方法的调用状态(包括方法调用的参数, 局部变量, 中间结果等)。每调用一个方法就将对应该方法的方法帧压入 Java 栈, 成为当前方法帧。当调用结束(返回)时, 就弹出该帧。

编译器将源代码编译成字节码(.class)时, 就已经将各种类型的方法的局部变量, 操作数栈大小确定并放在字节码中, 随着类一并装载入方法区。当调用方法时, 通过访问方法区中的类的信息, 得到局部变量以及操作数栈的大小。

也就是说: 在方法中定义的一些基本类型的变量和对象的引用变量都在方法的栈内存中分配。当在一段代码块定义一个变量时, Java 就在栈中为这个变量分配内存空间, 当超过变量的作用域后, Java 会自动释放掉为该变量所分配的内存空间, 该内存空间可以立即被另作它用。

### 栈内存的构成:

Java 栈内存由局部变量区、操作数栈、帧数据区组成。

(1): 局部变量区为一个以字为单位的数组, 每个数组元素对应一个局部变量的值。调用方法时, 将方法的局部变量组成一个数组, 通过索引来访问。若为非静态方法, 则加入一个隐含的引用参数 this, 该参数指向调用这个方法的对象。而静态方法则没有 this 参数。因此, 对象无法调用静态方法。

(2): 操作数栈也是一个数组, 但是通过栈操作来访问。所谓操作数是那些被指令操作的数据。当需要对参数操作时如  $a=b+c$ , 就将即将被操作的参数压栈, 如将 b 和 c 压栈, 然后由操作指令将它们弹出, 并执行操作。虚拟机将操作数栈作为工作区。

(3): 帧数据区处理常量池解析, 异常处理等

## 3: 堆内存

---

堆内存用来存放由 new 创建的对象和数组。在堆中分配的内存, 由 Java 虚拟机的自动垃圾回收器来管理。

在堆中产生了一个数组或对象后, 还可以在栈中定义一个特殊的变量, 让栈中这个变量的取值等于数组或对象在堆内存中的首地址, 栈中的这个变量就成了数组或对象的引用变量。引用变量就相当于为数组或对象起的一个名称, 以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。

### 栈内存和堆内存比较

栈与堆都是 Java 用来在内存中存放数据的地方。与 C++不同, Java 自动管理栈和堆, 程序员不能直接地设置栈或堆。

Java 的堆是一个运行时数据区, 对象从中分配空间。堆的优势是可以动态地分配内存大小, 生存期也不必事先告诉编译器, 因为它是在运行时动态分配内存的, Java 的垃圾收集器会自动收走这些不再使用的数据。但缺点是, 由于要在运行时动态分配内存, 存取速度较慢。

栈的优势是, 存取速度比堆要快, 仅次于寄存器, 栈数据可以共享。但缺点是, 存在栈

中的数据大小与生存期必须是确定的, 缺乏灵活性。栈中主要存放一些基本类型的变量(int, short, long, byte, float, double, boolean, char) 和对象句柄。

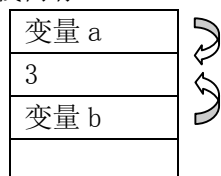
栈有一个很重要的特殊性, 就是存在栈中的数据可以共享。假设我们同时定义:

```
int a = 3;
```

```
int b = 3;
```

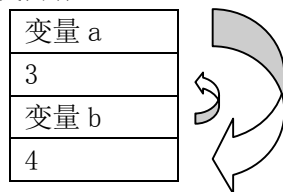
编译器先处理 `int a = 3;` 首先它会在栈中创建一个变量为 `a` 的引用, 然后查找栈中是否有 3 这个值, 如果没找到, 就将 3 存放进来, 然后将 `a` 指向 3。接着处理 `int b = 3;` 在创建完 `b` 的引用变量后, 因为在栈中已经有 3 这个值, 便将 `b` 直接指向 3。这样, 就出现了 `a` 与 `b` 同时均指向 3 的情况。内存示意图如下:

栈内存:



这时, 如果再令 `a=4;` 那么编译器 会重新搜索栈中是否有 4 值, 如果没有, 则将 4 存放进来, 并令 `a` 指向 4; 如果已经有了, 则直接将 `a` 指向这个地址。因此 `a` 值的改变不会影响到 `b` 的值。要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享是不同的, 因为这种情况 `a` 的修改并不会影响到 `b`, 它是由编译器完成的, 它有利于节省空间。此时的内存分配示意图如下:

栈内存:



而一个对象引用变量修改了这个对象的内部状态, 会影响到另一个对象引用变量。

#### 4: 本地方法栈内存

与调用的本地方法的语言相关, 如调用的是一个 `c` 语言方法则为一个 `c` 栈。本地方法可以回调 `java` 方法。若有 `java` 方法调用本地方法, 虚拟机就运行这个本地方法。

在虚拟机看来运行这个本地方法就是执行这个 `java` 方法, 如果本地方法抛出异常, 虚拟机就认为是这个 `java` 方法抛出异常。

Java 通过 Java 本地接口 JNI (Java Native Interface) 来调用其它语言编写的程序, 在 Java 里面用 `native` 修饰符来描述一个方法是本地方法。这个了解一下就好了, 在我们的课程中不会涉及到。

#### 5: String 的内存分配

`String` 是一个特殊的包装类数据。可以用:

```
String str = new String("abc");
```

```
String str = "abc";
```

两种的形式来创建, 第一种是用 `new()` 来新建对象的, 它会在存放于堆中。每调用一次就会创建一个新的对象。

而第二种是先在栈中创建一个对 String 类的对象引用变量 str，然后查找栈中有没有存放“abc”，如果没有，则将“abc”存放进栈，并令 str 指向“abc”，如果已经有“abc” 则直接令 str 指向“abc”。

比较类里面的数值是否相等时，用 equals() 方法；当测试两个包装类的引用是否指向同一个对象时，用==，下面用例子说明上面的理论。

```
String str1 = "abc";
String str2 = "abc";
System.out.println(str1==str2); //true
```

可以看出 str1 和 str2 是指向同一个对象的。

```
String str1 = new String ("abc");
String str2 = new String ("abc");
System.out.println(str1==str2); // false
```

用 new 的方式是生成不同的对象。每一次生成一个。

因此用第一种方式创建多个“abc”字符串，在内存中其实只存在一个对象而已。这种写法有利于节省内存空间。同时它可以在一定程度上提高程序的运行速度，因为 JVM 会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于 String str = new String("abc"); 的代码，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。

另一方面，要注意：我们在使用诸如 String str = "abc"; 的格式时，总是想当然地认为，创建了 String 类的对象 str。担心陷阱！对象可能并没有被创建！而可能只是指向一个先前已经创建的对象。只有通过 new() 方法才能保证每次都创建一个新的对象。

由于 String 类的值不可变性 (immutable)，当 String 变量需要经常变换其值时，应考虑使用 StringBuffer 或 StringBuilder 类，以提高程序效率。

## 作业

---

1: 创建一个构造方法重载的类, 并用另一个类调用

2: 创建 Rodent (啮齿动物): Mouse (老鼠), Gerbil (鼯鼠), Hamster (大颊鼠) 等的一个继承分级结构。在基础类中, 提供适用于所有 Rodent 的方法, 并在衍生类中覆盖它们, 从而根据不同类型的 Rodent 采取不同的行动。创建一个 Rodent 数组, 在其中填充不同类型的 Rodent, 然后调用自己的基础类方法, 看看会有什么情况发生。

3: 编写 MyPoint 的一个子类 MyXYZ, 表示三维坐标点, 重写 toString 方法用来显示这个对象的 x、y、z 的值, 如显示 (1, 2, 3), 最后用 main 方法测试

4: 当你试图编译和执行下面的程序时会发生什么?

```
class Mystery{
    String s;
    public static void main(String[] args){
        Mystery m=new Mystery();
        m.go();
    }
    void Mystery(){
        s="constructor";
    }
    void go(){
        System.out.println(s);
    }
}
```

选择下面的正确答案:

- A 编译不通过
- B 编译通过但运行时产生异常
- C 代码运行但屏幕上看不到任何东西
- D 代码运行, 屏幕上看到 constructor
- E 代码运行, 屏幕上看到 null

5: 当编译和运行下列程序段时, 会发生什么?

```
class Person {}
class Woman extends Person {}
class Man extends Person {}
public class Test
{
    public static void main(String argv[]){
        Man m=new Man();
        Woman w=(Woman) new Man();
    }
}
```

- A 通过编译和并正常运行。
- B 编译时出现例外。
- C 编译通过，运行时出现例外。
- D 编译不通过

6: 对于下列代码:

```
1  class Person {  
2      public void printValue(int i, int j) {//... }  
3      public void printValue(int i){//... }  
4  }  
5  public class Teacher extends Person {  
6      public void printValue() {//... }  
7      public void printValue(int i) {//...}  
8      public static void main(String args[]){  
9          Person t = new Teacher();  
10         t.printValue(10);  
11     }  
12 }
```

第 10 行语句将调用哪行语句?

- A line 2
- B line 3
- C line 6
- D line 7

7: 下列代码运行结果是什么?

```
public class Bool{  
    static boolean b;  
    public static void main(String []args){  
        int x=0;  
        if(b){  
            x=1;  
        }  
        else if(b=false){  
            x=2;  
        }  
        else if(b){  
            x=3;  
        }  
        else{  
            x=4;  
        }  
        System.out.println("x="+x);  
    }  
}
```

```
}
```

8: 在命令行输入 java X Y 的结果是:

```
public class X{  
    public void main(String []args){  
        System.out.println("Hello "+args[0]);  
    }  
}
```

- A. Hello X Y
- B. Hello X
- C. Hello Y
- D. 不能编译
- E. 运行时有异常

9: 下列代码编译并运行的结果是:

```
public class Test{  
    public static void main(String []args){  
        class T1 extends java.lang.Thread{ }  
        class T2 extends T1{ }  
        class T3 implements java.lang.Runnable{ }  
        new T1().start();  
        new T2().start();  
        new Thread(new T3()).start();  
        System.out.println("Executing");  
    }  
}
```

- A. 编译失败
- B. 程序能运行但永远永不终止
- C. 能运行并输出 "Executing"
- D. 运行时有异常

10: 代码运行结果是什么?

```
public class Test{  
    public static void main(String []args){  
        double num=7.4;  
        int a=(int)Math.abs(num+0.5);  
        int b=(int)Math.ceil(num+0.5);  
        int c=(int)Math.floor(num+0.5);  
        int d=(int)Math.round(num+0.5);  
        int e=(int)Math.round(num-0.5);  
        int f=(int)Math.floor(num-0.5);  
        int g=(int)Math.ceil(num-0.5);  
        int h=(int)Math.abs(num-0.5);  
    }  
}
```

```
        System.out.println("a="+a);
        System.out.println("b="+b);
        System.out.println("c="+c);
        System.out.println("d="+d);
        System.out.println("e="+e);
        System.out.println("f="+f);
        System.out.println("g="+g);
        System.out.println("h="+h);
    }
}
```

11: 完成此段代码可以分别添加哪两个选项?

1. public class Test{
  - 2.
  3. public static void main(String []args){
  - 4.
  5. System.out.println("c="+c);
  6. }
- }
- a) 在第 2 行加上语句 static char c;
  - b) 在第 2 行加上语句 char c;
  - c) 在第 4 行加上语句 static char c;
  - d) 在第 4 行加上语句 char c='f';

12: 下列代码运行结果是什么?

```
public class A{
    public static void main(String []args){
        int m=2;
        int p=1;
        int t=0;
        for(;p<5;p++){
            if(t++>m){
                m=p+t;
            }
        }
        System.out.println("t equals "+t);
    }
}
```

- A. 2
- B. 4
- C. 6
- D. 7

13: 已知如下的命令执行 java MyTest a b c



请问哪个是正确的？

- A、args[0] = "MyTest a b c"
- B、args[0] = "MyTest"
- C、args[0] = "a"
- D、args[1]= 'b'

14: 将下面类中的变量和方法改为静态的，使程序能正确编译执行。如果保持用实例变量和方法就必须创建对象，请创建 A 的对象并通过该对象来引用实例变量和方法。

```
public class A
{
    int a=9;
    public void show(int a)
    {
        System.out.println(a*10);
    }
    public static void main(String args[]){
        a+=a;
        show(a);
    }
}
```

15: 设计个 Circle 类，其属性为圆心点（类型为前面设计的类 MyPoint）和半径，并为此类编写以下三个方法：

- 一是计算圆面积的 calArea()方法；
- 二是计算周长的 calLength();
- 三是 boolean inCircle(MyPoint mp)方法，功能是测试作为参数的某个点是否在当前对象圆内（圆内，包括圆上返回 true；在圆外，返回 false）。

## 第五章数组和枚举

### 教学目标：

- 掌握数组的声明和创建
- 掌握数组初始化
- 掌握数组元素的访问
- 掌握多维数组
- 掌握数组的复制□
- 掌握基本的排序算法和数组的排序□
- 理解枚举类型的基本概念□
- 掌握枚举类型的基本使用□
- 理解枚举类型的基本特点□

## 一： 数组的声明和创建

---

### 1: 数组是什么

---

数组是由相同类型的若干项数据组成的一个数据集合。也就是说数组是用来集合相同类型的对象并通过一个名称来引用这个集合，数组是引用类型。

### 2: 数组的声明

---

你可以声明任何类型的数组——原始类型或类类型：

```
char s[];
```

```
Point p[]; // 这里 Point 是一个类
```

在 Java 编程语言中，即使数组是由原始类型构成，甚或带有其它类类型，数组也是一个对象。声明不能创建对象本身，而创建的是一个引用，该引用可被用来引用数组。数组元素使用的实际内存可由 new 语句或数组初始化软件动态分配。在后面，你将看到如何创建和初始化实际数组。

上述这种将方括号置于变量名之后的声明数组的格式，是用于 C、C++ 和 Java 编程语言的标准格式。这种格式会使声明的格式复杂难懂，因而，Java 编程语言允许一种替代的格式，该格式中的方括号位于变量名的左边：

```
char[] s;
```

```
Point[] p;
```

这样的结果是，你可以认为类型部分在左，而变量名在右。上述两种格式并存，你可选择一种你习惯的方式。声明不指出数组的实际大小。

**注意**——当数组声明的方括号在左边时，该方括号可应用于所有位于其右的变量

### 3: 数组的创建

---

可以象创建对象一样，使用关键字 new 创建一个数组。创建的时候要指明数组的长度。

```
s = new char [20];
```

```
p = new Point [100];
```

第一行创建了一个 20 个 char 值的数组，第二行创建了一个 100 个类型 Point 的变量。然而，它并不创建 100 个 Point 对象；创建 100 个对象的工作必须分别完成如下：

```
p[0] = new Point();
```

```
p[1] = new Point();
```

```
•  
•  
•
```

用来指示单个数组元素的下标必须总是从 0 开始，并保持在合法范围之内——大于 0 或等于 0 并小于数组长度。任何访问在上述界限之外的数组元素的企图都会引起运行时出错。

数组的下标也称为数组的索引，必须是整数或者整数表达式，如下：

```
int i[] = new int[(9-2)*3]; //这是合法的
```

其实，声明和创建可以定义到一行，而不用分开写。

## 二： 数组的初始化

---

当创建一个数组时，每个元素都被自动使用默认值进行初始化。在上述 char 数组 s 的例子中，每个值都被初始化为 0 (\u0000-null) 字符；在数组 p 的例子中，每个值都被初始化为 null，表明它还未引用一个 Point 对象。在经过赋值 `p[0] = new Point()` 之后，数组的第一个元素引用为实际 Point 对象。

**注意**——所有变量的初始化(包括数组元素)是保证系统安全的基础，变量绝不能在未初始化状态使用。

Java 编程语言允许使用下列形式快速创建数组，直接定义并初始化：

```
String names [] = {  
    "Georgianna",  
    "Jen",  
    "Simon",  
};
```

其结果与下列代码等同：

```
String names [] ;  
names = new String [3];  
names [0] = "Georgianna" ;  
names [1] = "Jen" ;  
names [2] = "Simon" ;
```

这种”速记”法可用在任何元素类型。例如：

```
Myclass array [] = {  
    new Myclass (),  
    new Myclass (),  
    new Myclass ()  
};
```

适当的类类型的常数值也可被使用：

```
import java.awt.Color;  
  
Color palette [] = {  
    Color.blue,  
    Color.red,  
    Color.white  
};
```

### 1: 数组的内存分配

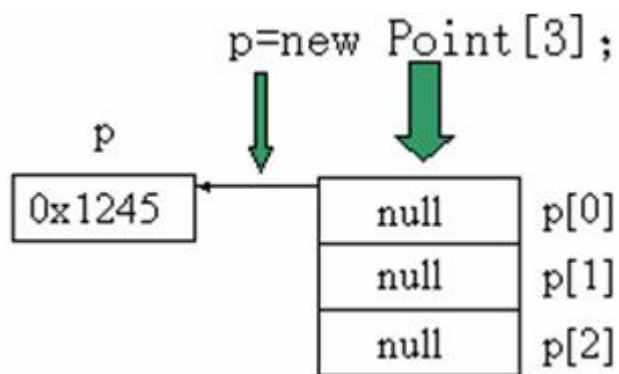
---

数组一旦被创建，在内存里面占用连续的内存地址。

数组还具有一个非常重要的特点——**数组的静态性**：数组一旦被创建，就不能更改数组的长度。

比如，定义数组如下：`Point[] p = new Point [3];`

其中 p 是数组名，数组长度是 3，数组在被创建的时候，内存示意图如下：



### 三： 数组元素的访问

在 Java 编程语言中，所有数组的下标都从 0 开始。一个数组中元素的数量被作为具有 `length` 属性的部分数组对象而存储；这个值被用来检查所有运行时访问的界限。如果发生了一个越出界限的访问，那么运行时的报错也就出现了。

使用 `length` 属性的例子如下：

```
int list [] = new int [10];
for (int i= 0; i< list.length; i++) {
    System.out.println(list[i]);
}
```

使用 `length` 属性使得程序的维护变得更简单。

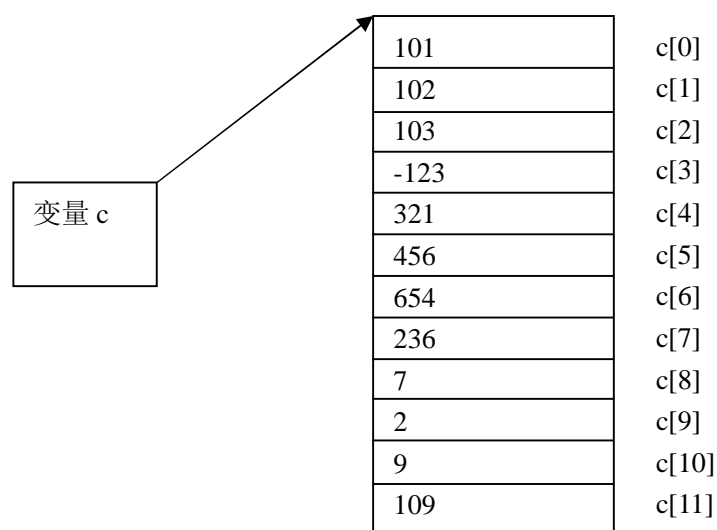
所有元素的访问就通过数组的下标来访问，如上例的 `list[i]`，随着 `i` 的值发生变化，就依次访问 `list[0]`、`list[1]`、`list[2]`...

如果想要给某个数组元素赋值，如下方式：`list[0]=5; list[1]=6;`...

**示例：**假如定义一个数组：`int c [] = new int[12];`

.....//进行赋值的语句

对数组进行赋值后，内存示意图如下：



然后就可以根据数组名[下标]来取值了。

如: `int a = c[3];`

结果就是: 从数组中取出下标为 3 的元素的值 “-123”, 然后赋值给 a。

### 3.1: 更优化的 for 循环语句

在访问数组的时候, 经常使用 for 循环语句。从 JDK5.0 开始, 提供了一个更好的 for 循环语句的写法, 示例如下:

```
public class Test {
    public static void main(String args[]) {
        int a[] = new int[3];
        //旧的写法, 赋值
        for(int i=0;i<a.length;i++){
            a[i] = i;
        }
        //新的写法, 取值
        for(int i : a){
            System.out.println(i);
        }
    }
}
```

显然 JDK5.0 版本的写法比以前是大大简化了。

## 四: 多维数组

---

### 1: 多维数组的基础知识

Java 编程语言没有象其它语言那样提供多维数组。因为一个数组可被声明为具有任何基础类型, 所以你可以创建数组的数组 (和数组的数组的数组, 等等)。一个二维数组如下例所示:

```
int twoDim [][] = new int [4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
```

首次调用 new 而创建的对象是一个数组, 它包含 4 个元素, 每个元素对类型 array of int 的元素都是一个 null 引用并且必须将数组的每个点分别初始化。

因为这种对每个元素的分别初始化, 所以有可能创建非矩形数组的数组。也就是说, twoDim 的元素可按如下方式初始化:

```
twoDim[0] = new int [2]
twoDim[1] = new int [4];
twoDim[2] = new int [6];
twoDim[3] = new int [8];
```

由于此种初始化的方法烦琐乏味, 而且矩形数组的数组是最通用的形式, 因而产生了一种”速记”方法来创建二维数组。例如:

```
int twoDim [][] = new int [3][4];
```

可被用来创建一个每个数组有 4 个整数类型的 3 个数组的数组。

对二维数组 `int [][] a = new int[3][4];`

可以理解成为如下图所示:

a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

行的下标值

列的下标值

**注意**—尽管声明的格式允许方括号在变量名左边或者右边, 但此种灵活性不适用于数组句法的其它方面。例如: `new int [][][4]` 是非法的。

## 2: 示例

```
class FillArray
{
    public static void main (String args[])
    {
        int[ ][ ] matrix = new int[4][5]; //二维数组的声明和创建
        for (int row=0; row < 4; row++)
        {
            for (int col=0; col < 5; col++)
            {
                matrix[row][col] = row + col; //二维数组的访问, 为元素赋值
            }
        }
    }
}
```

当然也可以直接定义并赋值, 如下:

```
double[ ][ ] c =
{
    {1.0, 2.0, 3.0, 4.0},
    {0.0, 1.0, 0.0, 0.0},
    {0.0, 0.0, 1.0, 0.0}
};
```

从上面可以看得很清楚, 二维数组其实就是一维的一维数组。

### 3: 多维数组的本质

**N 维数组就是一维的 N-1 维数组**, 比如: 三维数组就是一维的二维数组。

三维以至多维数组都是一个思路, 一维数组——> 二维数组——> 三维数组的实例:

```
class Fill3DArray
{
    public static void main (String args[])
    {
        int[ ][ ][ ] M = new int[4][5][3];
        for (int row=0; row < 4; row++)
        {
            for (int col=0; col < 5; col++)
            {
                for (int ver=0; ver < 3; ver++)
                {
                    M[row][col][ver] = row + col + ver;
                }
            }
        }
    }
}
```

## 五: 数组的复制

---

数组一旦创建后, 其大小不可调整。然而, 你可使用相同的引用变量来引用一个全新的数组:

```
int myArray [] = new int [6];
myArray = new int [10];
```

在这种情况下, 第一个数组被有效地丢失, 除非对它的其它引用保留在其它地方。

Java 编程语言在 System 类中提供了一种特殊方法拷贝数组, 该方法被称作 arraycopy()。例如, arraycopy 可作如下使用:

```
//原始数组
int myArray[] = { 1, 2, 3, 4, 5, 6 };

//新的数组, 比原始数组大
int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
//把原始数组的值拷贝到新的数组
System.arraycopy(myArray, 0, hold, 0, myArray.length);
```

拷贝完成后, 数组 hold 有如下内容: 1, 2, 3, 4, 5, 6, 4, 3, 2, 1。

**注意**—在处理对象数组时, System.arraycopy() 拷贝的是引用, 而不是对象。对象本身不改变。



## 六: 数组的排序

---

在讨论数组排序之前, 我们先来看看一些基本的排序方法:

### 1: 冒泡排序

对几个无序的数字进行排序, 比较常用的方法是冒泡排序法。冒泡法排序是一个比较简单的排序方法, 在待排序的数列基本有序的情况下排序速度较快。

基本思路: 对未排序的各元素从头到尾依次比较相邻的两个元素是否逆序(与欲排顺序相反), 若逆序就交换这两元素, 经过第一轮比较排序后便可把最大(或最小)的元素排好, 然后再用同样的方法把剩下的元素逐个进行比较, 就得到了你所要的顺序。

可以看出如果有 N 个元素, 那么一共要进行 N-1 轮比较, 第 I 轮要进行 N-I 次比较。(如有 5 个元素, 则要进行 5-1 轮比较。第 3 轮则要进行 5-3 次比较)

示例如下:

```
public class Test {
    public static void main(String[] args) {
        //需要排序的数组, 目前是按照升序排列的
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;

        //冒泡排序
        for(int i=0;i<a.length;i++){
            for(int j=i+1;j<a.length;j++){//注意j的开始值是i+1, 因为按照排序
                //规则, 比a[i]大的值都应该在它后面
                if(a[i] > a[j]){
                    int temp = a[j];
                    a[j] = a[i];
                    a[i] = temp;
                }
            }
        }
        //检测一下排序的结果
        for(int i : a){
            System.out.println("i="+i);
        }
    }
}
```

运行结果:

```
i=1
i=2
i=3
i=4
i=5
```

如果你想要按照降序排列, 很简单, 只需把: `if(a[i] > a[j])` 改成: `if(a[i] < a[j])` 就可以了。

## 2: 选择排序

基本思路: 从所有元素中选择一个最小元素 `a[i]` 放在 `a[0]` (即让最小元素 `a[i]` 与 `a[0]` 交换), 作为第一轮; 第二轮是从 `a[1]` 开始到最后的各个元素中选择一个最小元素, 放在 `a[1]` 中; ……依次类推。n 个数要进行 (n-1) 轮。比较的次数与冒泡法一样多, 但是在每一轮中只进行一次交换, 比冒泡法的交换次数少, 相对于冒泡法效率高。

示例如下:

```
public class Test {
    public static void main(String[] args) {
        //需要排序的数组, 目前是按照升序排列的
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;

        //选择法排序
        int temp;
        for (int i = 0; i < a.length; i++) {
            int lowIndex = i;
            //找出最小的一个的索引
            for (int j = i + 1; j < a.length; j++) {
                if (a[j] < a[lowIndex]) {
                    lowIndex = j;
                }
            }
            //交换
            temp = a[i];
            a[i] = a[lowIndex];
            a[lowIndex] = temp;
        }

        //检测一下排序的结果
        for (int i : a) {
            System.out.println("i=" + i);
        }
    }
}
```

运行结果:

i=1

```
i=2  
i=3  
i=4  
i=5
```

如果你想要按照降序排列, 很简单, 只需要把: `if (a[j] < a[lowIndex])` 这句话修改成: `if (a[j] > a[lowIndex])` 就可以了。

### 3: 插入法排序

基本思路: 每拿到一个元素, 都要将这个元素与所有它之前的元素遍历比较一遍, 让符合排序顺序的元素挨个移动到当前范围内它最应该出现的位置。

举个例子来说, 就用前面的数组, 我们要对一个有 5 个元素的数组进行升序排列, 假设第一个元素的值被假定为已排好序了, 那么我们就将第 2 个元素与数组中的部分进行比较, 如果第 2 个元素的值较小, 则将它插入到第 1 个元素的前面, 现在就有两个元素排好序了, 我们再将没有排序的元素与排好序的元素列表进行比较, 同样, 如果小于第一个元素, 就将它插入到第一个元素前面, 但是, 如果大于第一个元素的话, 我们就将它再与第 2 个元素的值进行比较, 小于的话就排在第 2 个元素前面, 大于的话, 就排在第 2 个元素的后面。以此类推, 直到最后一个元素排好序。

示例如下:

```
public class Test {  
    public static void main(String[] args) {  
        // 需要排序的数组, 目前是按照升序排列的  
        int a[] = new int[5];  
        a[0] = 3;  
        a[1] = 4;  
        a[2] = 1;  
        a[3] = 5;  
        a[4] = 2;  
  
        // 插入法排序  
        int temp;  
        for (int i = 1; i < a.length; i++) { // i=1开始, 因为第一个元素认为是已经排好序了的  
            for (int j = i; (j > 0) && (a[j] < a[j - 1]); j--) {  
                // 交换  
                temp = a[j];  
                a[j] = a[j - 1];  
                a[j - 1] = temp;  
            }  
        }  
        // 检测一下排序的结果  
        for (int i : a) {  
            System.out.println("i=" + i);  
        }  
    }  
}
```

```
    }  
}
```

运行结果:

```
i=1  
i=2  
i=3  
i=4  
i=5
```

如果你想要按照降序排列, 很简单, 只需要把: `a[j] < a[j - 1]`这句话修改成: `a[j] > a[j - 1]`就可以了。

#### 4: 希尔(Shell)法排序

从前面介绍的冒泡排序法, 选择排序法, 插入排序法可以发现, 如果数据已经大致排好序的时候, 其交换数据位置的动作将会减少。例如在插入排序法过程中, 如果某一整数 `d[i]` 不是较小时, 则其往前比较和交换的次数会更少。如何用简单的方式让某些数据有一定的大小次序呢? Donald Shell (Shell 排序的创始人) 提出了希尔法排序。

基本思路: 先将数据按照固定的间隔分组, 例如每隔 4 个分成一组, 然后排序各分组的数据, 形成以分组来看数据已经排序, 从全部数据来看, 较小值已经在前面, 较大值已经在后面。将初步处理了的分组再用插入排序来排序, 那么数据交换和移动的次数会减少。可以得到比插入排序法更高的效率。

示例如下:

```
public class Test {  
    public static void main(String[] args) {  
        // 需要排序的数组, 目前是按照升序排列的  
        int a[] = new int[5];  
        a[0] = 3;  
        a[1] = 4;  
        a[2] = 1;  
        a[3] = 5;  
        a[4] = 2;  
  
        // shell法排序  
        int j = 0;  
        int temp = 0;  
        //分组  
        for (int increment = a.length / 2; increment > 0; increment /= 2)  
        {  
            //每个组内排序  
            for (int i = increment; i < a.length; i++) {  
                temp = a[i];  
                for (j = i; j >= increment; j -= increment) {  
                    if (temp < a[j - increment]){  
                        a[j] = a[j - increment];  

```

```
        }else{
            break;
        }
    }
    a[j] = temp;
}

// 检测一下排序的结果
for (int i2 : a) {
    System.out.println("i=" + i2);
}
}
```

运行结果:

```
i=1
i=2
i=3
i=4
i=5
```

如果你想要按照降序排列, 很简单, 只需要把: `if (temp < a[j - increment])` 这句话修改成: `if (temp > a[j - increment])`就可以了。

## 5: 数组排序

---

事实上, 数组的排序不用那么麻烦, 上面只是想让大家对一些基本的排序算法有所了解而已。在 `java.util.Arrays` 类中有一个静态方法 `sort`, 可以用这个类的 `sort` 方法来对数组进行排序。

示例如下:

```
public class Test {
    public static void main(String[] args) {
        // 需要排序的数组, 目前是按照升序排列的
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;

        //数组排序
        java.util.Arrays.sort(a);
        // 检测一下排序的结果
        for (int i2 : a) {
            System.out.println("i=" + i2);
        }
    }
}
```

```
    }  
  }  
}
```

注意：现在的 sort 方法都是升序的，要想实现降序的，还需要 Comparator 的知识，这个在后面会学到。

## 七： 枚举类型

### 1： 枚举类型是什么

枚举类型 enum 是一种新的类型，在 JDK5.0 加入，允许用常量来表示特定的数据片断，这些数据是分配时预先定义的值的集合，而且全部都以类型安全的形式来表示。

在枚举类型没有加入到 Java 前，我们要想表达常量的集合，通常采用如下的方式：

```
public class Test {  
    public static final int A = 1;  
    public static final int B = 2;  
    public static final int C = 3;  
    public static final int D = 4;  
    public static final int E = 5;  
}
```

那么我们在使用的时候就采用如下代码：

Test.A 或者 Test.B 之类的代码。

但是在这样做的时候，我们需要记住这类常量是 Java 中 int 类型的常量，这意味着该方法可以接受任何 int 类型的值，即使它和 Test 中定义的所有级别都不对应。因此需要检测上界和下界，在出现无效值的时候，可能还要包含一个 IllegalArgumentException。而且，如果后来又添加另外一个级别（例如 TEST.F，那么必须改变所有代码中的上界，才能接受这个新值。

换句话说，在使用这类带有整型常量的类时，该解决方案也许可行，但并不是非常有效。枚举就为处理上述问题提供了更好的方法。

把上面的例子改成用枚举的方式：

```
public class Test {  
    public enum StudentGrade{  
        A,B,C,D,E,F  
    };  
}
```

可以采用如下的方式进行使用

```
public class Test {  
    public enum StudentGrade{  
        A,B,C,D,E,F  
    };  
  
    public static void main(String[] args) {  
        System.out.println("学生的平均成绩为==" + StudentGrade.B);  
    }  
}
```

迄今为止, 您所看到的示例都相当简单, 但是枚举类型提供的东西远不止这些。您可以逐个遍历枚举值, 也可以在 `switch` 语句中使用枚举值, 枚举是非常有价值的。

## 2: 遍历枚举类型

---

示例如下:

```
public class Test {
    public enum StudentGrade{
        A,B,C,D,E,F
    };
    public static void main(String[] args) {
        for(StudentGrade score : StudentGrade.values()){
            System.out.println("学生成绩取值可以为==" + score);
        }
    }
}
```

运行结果:

```
学生成绩取值可以为==A
学生成绩取值可以为==B
学生成绩取值可以为==C
学生成绩取值可以为==D
学生成绩取值可以为==E
学生成绩取值可以为==F
```

`values()` 方法返回了一个由独立的 `StudentGrade` 实例构成的数组。

还有一个常用的方法: `valueOf(String)` : 功能是以字符串的形式返回某一个具体枚举元素的值, 示例如下:

```
public class Test {
    public enum StudentGrade{
        A,B,C,D,E,F
    };
    public static void main(String[] args) {
        Test t = new Test();
        StudentGrade score = StudentGrade.valueOf("A");
        System.out.println("你的成绩是:" + score);
    }
}
```

运行结果: 你的成绩是:A

### 3: 在 switch 中使用枚举类型

---

示例如下:

```
public class Test {
    public enum StudentGrade{
        A,B,C,D,E,F
    };
    public static void main(String[] args) {
        Test t = new Test();
        StudentGrade score = StudentGrade.C;
        switch(score){
            case A:
                System.out.println("你的成绩是优秀");
                break;
            case B:
                System.out.println("你的成绩是好");
                break;
            case C:
                System.out.println("你的成绩是良");
                break;
            case D:
                System.out.println("你的成绩是及格");
                break;
            default:
                System.out.println("你的成绩是不及格");
                break;
        }
    }
}
```

运行结果: 你的成绩是良

在这里, 枚举值被传递到 switch 语句中, 而每个 case 子句将处理一个特定的值。该值在提供时没有枚举前缀, 这意味着不用将代码写成 case StudentGrade.A, 只需将其写成 case A 即可, 编译器不会接受有前缀的值。

### 4: 枚举类型的特点

---

从上面的示例中可以看出, 枚举类型大概有如下特点:

- (1): 类型安全
- (2): 紧凑有效的枚举数值定义
- (3): 运行的高效率

#### 4.1: 类型安全

枚举的申明创建了一个新的类型。它不同于其它的已有类型, 包括原始类型(整数, 浮点数等等)和当前作用域(Scope)内的其它的枚举类型。当你方法的参数进行赋值操作的时候, 整数类型和枚举类型是不能互换的(除非是你进行显式的类型转换), 编译器将强



制这一点。比如说，用上面申明的枚举定义这样一个方法：

```
public void foo(Day);
```

如果你用整数来调用这个函数，编译器会给出错误的。必须用这个类型的值进行调用。

```
foo(4); // compilation error
```

#### 4.2：紧凑有效的枚举数值定义

比较前面写的例子，你看看是枚举定义写得紧凑，还是直接使用 `static final` 紧凑呢。答案是不言而喻的。

#### 4.3：运行的高效率

枚举的运行效率和原始类型的整数类型基本上一样高。在运行时不会由于使用了枚举而导致性能有所下降。

## 作业

---

1: 写一个方法，在方法内部定义一个一维的 int 数组，然后为这个数组赋上初始值，最后再循环取值并打印出来

2: 下面的数组定义那些是正确的

- A: `int a[][] = new int[3,3];`
- B: `int a[3][3] = new int[][];`
- C: `int a[][] = new int[3][3];`
- D: `int []a[] = new int[3][3];`
- E: `int [][]a = new int[3][3];`

3: 定义一个长度为 10 的一维字符串数组，在每一个元素存放一个单词；然后运行时从命令行输入一个单词，程序判断数组是否包含有这个单词，包含这个单词就打印出 “Yes”，不包含就打印出 “No”

4: 请在下面语句中找出一个正确的。

- A. `int arr1[2][3];`
- B. `int[][] a2 = new int[2][];`
- C. `int[][] arr2=new int [][4];`
- D. `int arr3[][4]= new int [3][4];`

5: 用二重循环求出二维数组 b 所有元素的和：

```
int[][] b={{11},{21,22},{31,32,33}}
```

6: 编写一个方法实现将班级同学的名单存放在数组中，并利用随机数(Math.random())随机输出一位同学的姓名。

7: 生成一百个随机数，放到数组中，然后排序输出。

8: 统计字符串中英文字母、空格、数字和其它字符的个数。

## 第六章 常见类的使用

### 教学目标：

- 理解和掌握 Object 类
- 理解和掌握 String 类
- 理解和掌握正则表达式基础知识
- 理解和掌握 StringBuffer 类
- 理解和掌握 StringBuilder 类
- 掌握 Math 类的使用□
- 掌握日期相关类的使用□
- 掌握 System 类的使用□

## 一: Object 类

---

java.lang 包中定义的 Object 类是所有 Java 类的根父类, 其中定义了一些实现和支持面向对象机制的重要方法。任何 Java 对象, 如果没有父类, 就默认它继承了 Object 类。因此, 实际上, 以前的定义是下面的简略:

```
public class Employee extends Object 和
public class Manager extends Employee
```

Object 类定义许多有用的方法, 包括 toString(), 它就是为什么 Java 软件中每样东西都能转换成字符串表示法的原因。(即使这仅具有有限的用途)。

### 1: equals 方法

---

Object 类定义的 equals 方法用于判别某个指定的对象与当前对象(调用 equals 方法的对象)是否等价。在 Java 语言中数据等价的基本含义是指两个数据的值相等。

”==”进行比较的时候, 引用类型数据比较的是引用, 即内存地址, 基本数据类型比较的是值。

#### 1.1: equals 方法与 “==” 运算符的关系

equals()方法只能比较引用类型, “==”可以比较引用类型及基本类型;

**特例:** 当用 equals()方法进行比较时, 对类 File、String、Date 及包装类来说, 是比较类型及内容而不考虑引用的是否是同一个实例;

用“==”进行比较时, 符号两边的数据类型必须一致(可自动转换的数据类型除外), 否则编译出错, 而用 equals 方法比较的两个数据只要都是引用类型即可。

示例如下:

```
class MyDate {
    private int day, month, year;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }
}

public class Test {
    public static void main(String[] args) {
        MyDate m1 = new MyDate(8, 8, 2008);
        MyDate m2 = new MyDate(8, 8, 2008);

        if (m1 == m2) {
            System.out.println("m1==m2");
        } else {
            System.out.println("m1!=m2");
        }
    }
}
```

```
        if (m1.equals(m2)) {
            System.out.println("m1 is equal to m2");
        } else {
            System.out.println("m1 is not equal to m2");
        }

        m2 = m1;
        if (m1 == m2) {
            System.out.println("m1==m2");
        } else {
            System.out.println("m1!=m2");
        }
    }
}
```

程序运行结果为:

```
m1!=m2
m1 is not equal to m2
m1==m2
```

小结一下:

在引用类型的比较上, Object 里面的 equals 方法默认的比较方式, 基本上等同于“==”, 都是比较内存地址, 只有那几个特殊的是比较的值。

## 1.2: 覆盖 equals 方法

对于程序员来说, 如果一个对象需要调用 equals 方法, 应该在类中覆盖 equals 方法。如果覆盖了 equals 方法, 那么具体的比较就按照你的实现进行比较了。

一般来讲: 为了比较两个分离的对象 (也就是内存地址不同的两个对象), 自行覆盖的 equals 方法里面都是检查类型和值是否相同。上面那几个特殊的情况就是这样, 比如 String 类, 它覆盖了 equals 方法, 然后在里面进行值的比较。

覆盖 equals 方法的一般步骤如下:

- (1): 用==检查是否参数就是这个对象的引用
- (2): 判断要比较的对象是否为 null, 如果是 null, 返回 false
- (3): 用 instanceof 判断参数的类型是否正确
- (4): 把参数转换成合适的类型
- (5): 比较对象属性值是不是匹配

示例如下:

覆盖前 equals 和==比较的都是内存地址:

```
public class Test{
    public static void main(String[] args) {
        A a1 = new A();
        a1.age = 3;
        A a2 = new A();
        a2.age = 3;
```

```
        System.out.println("a1 == a2 test =" + (a1 == a2));
        System.out.println("a1 equals a2 test =" + a1.equals(a2));
    }
}
class A{
    public int age = 0;
}
```

运行结果是:

```
a1 == a2 test =false
a1 equals a2 test =false
```

覆盖后 equals 比较的是值, ==比较的是内存地址:

```
public class Test{
    public static void main(String[] args) {
        Test t = new Test();
        A a1 = new A();
        a1.age = 3;
        A a2 = new A();
        a2.age = 3;
        System.out.println("a1 == a2 test =" + (a1 == a2));
        System.out.println("a1 equals a2 test =" + a1.equals(a2));
    }
}
class A{
    public int age = 0;
    public boolean equals(Object obj){
        //第一步先判断是否同一个实例
        if(this == obj){
            return true;
        }
        //第二步判断要比较的对象是否为null
        if (obj == null){
            return false;
        }
        //第三步判断是否同一个类型
        if(obj instanceof A){
            //第四步类型相同, 先转换为同一个类型
            A a = (A)obj;
            //第五步然后进行对象属性值的比较
            if(this.age == a.age){
                return true;
            }else{
                return false;
            }
        }
    }
}
```

```
    }else{
        //类型不同, 直接返回false
        return false;
    }
}
}
```

**说明:** 如果对象的属性又是一个引用类型的话, 会继续调用该引用类型的 equals 方法, 直到最后得出相同还是不同的结果。示例如下:

```
public class Test{
    public static void main(String[] args) {
        Test t = new Test();
        A a1 = new A();
        a1.age = 3;
        A a2 = new A();
        a2.age = 3;
        System.out.println("a1 == a2 test =" + (a1==a2));
        System.out.println("a1 equals a2 test =" + a1.equals(a2));
    }
}

class A{
    public int age = 0;
    public String name = "Java私塾";
    public boolean equals(Object obj){
        //第一步先判断是否同一个实例
        if(this==obj){
            return true;
        }
        //第二步判断要比较的对象是否为null
        if (obj == null){
            return false;
        }
        //第三步判断是否同一个类型
        if(obj instanceof A){
            //第四步类型相同, 先转换为同一个类型
            A a = (A)obj;
            //第五步然后进行对象属性值的比较
            if(this.age == a.age && this.name.equals(a.name)){
                return true;
            }else{
                return false;
            }
        }else{
            //类型不同, 直接返回false
        }
    }
}
```

```
        return false;
    }
}
}
```

最后重要的一点规则：覆盖 `equals` 方法应该连带覆盖 `hashCode` 方法。

## 2: hashCode 方法

---

`hashCode` 是按照一定的算法得到的一个数值，是对象的散列码值。主要用来在集合（后面会学到）中实现快速查找等操作，也可以用于对象的比较。

在 Java 中，对 `hashCode` 的规定如下：

- (1)：在同一个应用程序执行期间，对同一个对象调用 `hashCode()`，必须返回相同的整数结果——前提是 `equals()` 所比较的信息都不曾被改动过。至于同一个应用程序在不同执行期所得的调用结果，无需一致。
- (2)：如果两个对象被 `equals(Object)` 方法视为相等，那么对这两个对象调用 `hashCode()` 必须获得相同的整数结果。
- (3)：如果两个对象被 `equals(Object)` 方法视为不相等，那么对这两个对象调用 `hashCode()` 不必产生不同的整数结果。然而程序员应该意识到，对不同对象产生不同的整数结果，有可能提升 `hashCode`（后面会学到，集合框架中的一个类）的效率。

简单地说：如果两个对象相同，那么它们的 `hashCode` 值一定要相同；如果两个对象的 `hashCode` 相同，它们并不一定相同。

在 Java 规范里面规定，覆盖 `equals` 方法应该连带覆盖 `hashCode` 方法，这就涉及到一个如何实现 `hashCode` 方法的问题了。

**实现一：偷懒的做法：对同一对象始终返回相同的 `hashCode`，如下：**

```
public int hashCode(){
    return 1;
}
```

它是合法的，但是不好，因为每个对象具有相同的 `hashCode`，会使得很多使用 `hashCode` 的类的运行效率大大降低，甚至发生错误。

**实现二：采用一定的算法来保证**

在高效 Java 编程这本书里面，给大家介绍了一个算法，现在 eclipse 自动生成 `equals` 方法和 `hashCode` 方法就是用的这个算法，下面介绍一下这个算法：

- (1)：将一个非 0 常数，例如 31，储存于 `int result` 变量
- (2)：对对象中的每个有意义的属性 `f`（更确切的说是被 `equals()` 所考虑的每一个属性）进行如下处理：
  - A. 对这个属性计算出类型为 `int` 的 hash 码 `c`：
    - i. 如果属性是个 `boolean`，计算 `(f ? 0 : 1)`。
    - ii. 如果属性是个 `byte`, `char`, `short` 或 `int`，计算 `(int)f`。
    - iii. 如果属性是个 `long`，计算 `(int)(f^(f >>> 32))`。
    - iv. 如果属性是个 `float`，计算 `Float.floatToIntBits(f)`。



v. 如果属性是个 double, 计算 Double.doubleToLongBits(f), 然后将计算结果按步骤 2.A.iii 处理。

vi. 如果属性是个对象引用, 而且 class 的 equals() 通过递归调用 equals() 的方式来比较这一属性, 那么就同样也对该属性递归调用 hashCode()。如果需要更复杂的比较, 请对该属性运算一个范式 (canonical representation), 并对该范式调用 hashCode()。如果属性值是 null, 就返回 0 (或其它常数; 返回 0 是传统做法)。

vii. 如果属性是个数组, 请将每个元素视为独立属性。也就是说对每一个有意义的元素施行上述规则, 用以计算出 hash 码, 然后再依步骤 2.B 将这些数值组合起来。

B. 将步骤 A 计算出来的 hash 码 c 按下列公式组合到变量 result 中:

result = 31\*result + c;

(3): 返回 result。

示例如下: 这个就是用 eclipse 自动生成的

```
public class Test{
    private byte byteValue;
    private char charValue;
    private short shortValue;
    private int intValue;
    private long longValue;
    private boolean booleanValue;
    private float floatValue;
    private double doubleValue;
    private String uuid;
    private int[] intArray = new int[3];

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + (booleanValue ? 1231 : 1237);
        result = prime * result + charValue;
        long temp;
        temp = Double.doubleToLongBits(doubleValue);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        result = prime * result + Float.floatToIntBits(floatValue);
        result = prime * result + Arrays.hashCode(intArray);
        result = prime * result + intValue;
        result = prime * result + (int) (longValue ^ (longValue >>> 32));
        result = prime * result + shortValue;
        result = prime * result + ((uuid == null) ? 0 : uuid.hashCode());
        return result;
    }
}
```

### 3: toString 方法

---

toString()方法是 Object 类中定义的另一个重要方法, 其格式为:  
`public String toString(){……}`

方法的返回值是 String 类型, 用于描述当前对象的有关信息。Object 类中实现的 toString() 方法是返回当前对象的类型和内存地址信息, 但在一些子类 (如 String, Date 等) 中进行了重写, 也可以根据需要在用户自定义类型中重写 toString() 方法, 以返回更适用的信息。

除显式调用对象的 toString() 方法外, 在进行 String 与其它类型数据的连接操作时, 会自动调用 toString() 方法, 其中又分为两种情况:

- (1): 引用类型数据直接调用其 toString() 方法转换为 String 类型;
- (2): 基本类型数据先转换为对应的包装类型, 再调用该包装类的 toString() 方法转换为 String 类型。

另外, 在 System.out.println() 方法输出引用类型的数据时, 也先自动调用了该对象的 toString() 方法, 然后再将返回的字符串输出。

示例如下:

```
class MyDate{
    private int day, month, year;
    public MyDate(int d, int m, int y){
        day = d;    month = m;    year = y;
    }
}

class YourDate{
    private int day, month, year;
    public YourDate(int d, int m, int y){
        day = d;    month = m;    year = y;
    }
    public String toString(){
        return day + "-" + month + "-" + year;
    }
}

public class Test{
    public static void main(String args[]){
        MyDate m = new MyDate(8,8,2008);
        System.out.println(m);
        System.out.println(m.toString());
        YourDate y = new YourDate(8,8,2008);
        System.out.println(y);
    }
}
```

运行结果:

```
cn.javass.java6.test.MyDate@1fb8ee3
cn.javass.java6.test.MyDate@1fb8ee3
8-8-2008
```

toString 方法被用来将一个对象转换成 String 表达式。当自动字符串转换发生时, 它被用作编译程序的参照。System.out.println()调用下述代码:

```
Date now = new Date()
System.out.println(now)
```

将被翻译成:

```
System.out.println(now.toString());
```

对象类定义缺省的 toString()方法, 它返回类名称和它的引用的地址(通常情况下不是很有用)。许多类覆盖 toString()以提供更有用的信息。例如, 所有的包装类覆盖 toString()以提供它们所代表的值的字符串格式。甚至没有字符串格式的类为了调试目的常常实现 toString()来返回对象状态信息。

## 二: String 类

---

### 1: String 的直接量

---

双引号括起来的字符序列就是 String 的直接量。实例: “John” 或 “111222333” 字符串赋值, 可以在声明时赋值

```
String color = "blue";
color 是 String 类型的引用。
“blue” 是 String 直接量。
```

String 直接量是存放在栈内存里, 所以一旦定义就不能改变值了, 只能是让变量指向新的内存空间。比如:

```
color = "red" ;
```

如果采用 new 的方法定义 String, 那么是需要分配堆内存空间的, 如下:

```
String str = new String("Hello");
```

一共有两个对象, 在栈和堆内存中各有一个对象, 内容都是 “Hello”。

### 2: String 的常用方法

---

#### 2.1: String 方法:length(), charAt(), getBytes()和 getChars()

方法 length()

- 返回 String 的长度, 是按照 char 返回的长度
- 与数组不同之处: String 类不含有 length 成员域(属性)

方法 charAt(int index)

- 获得字符串指定位置的字符

方法 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

方法 getChars(int srcBegin, int srcEnd,

```
char[ ] dst, int dstBegin)
```

- 拷贝字符串的部分字符序列到指定的字符数组的指定位置

**注意:** 对于字符串中的汉字, 是按照 char 来计算的, 一个中文汉字占两个字节, 也就是说, 通过 length() 得到的是字符串 char 的长度, 而不是字节数, 利用这个特点, 就可以进行中文判断了。

例如: 如何判断一个字符串里面有没有中文呢? 如果字符串对应的 byte[] 和 char[] 的长度是不一样的, 那就说明包含中文, 其实还可以顺带计算出有几个汉字。

```
public class Test{
    public static void main(String[] args) {
        String str = "这里是Java私塾";
        int charLen = str.length();
        int byteLen = str.getBytes().length;
        if(byteLen > charLen){
            int chineseNum = byteLen - charLen;
            System.out.println("str包含汉字, 汉字共"+chineseNum+"个");
        }else{
            System.out.println("str没有包含汉字");
        }
    }
}
```

运行结果是:

str 包含汉字, 汉字共 5 个

## 2.2: 字符串比较

- 字符类型的数据也是数值类型数据
- 比较字符串大小, 实际上就是依次比较其所包含的字符的数值大小
- 小写字母与大小字母是不相同的, Java 是区分大小写

### 方法 compareTo(String s):

比较两个字符串的大小。返回 0 表示相等, 返回大于 0 的数表示前面的字符串大于后面的字符串, 返回小于 0 表示前面的字符串小于后面的字符串, 区分大小写的。如下:

```
public class Test{
    public static void main(String[] args) {
        String str = "这里是 JAVA 私塾";
        String str2 = "这里是 java 私塾";
        if(str.compareTo(str2)==0){
            System.out.println("the str 等于 str2");
        }else if(str.compareTo(str2) > 0){
            System.out.println("the str 大于 str2");
        }else if(str.compareTo(str2) < 0){
            System.out.println("the str 小于 str2");
        }
    }
}
```

```
    }  
}  
运行结果: the str 小于 str2
```

**方法 compareToIgnoreCase(String s):** 忽略大小写, 比较两个字符串的大小。返回 0 表示相等, 返回大于 0 的数表示前面的字符串大于后面的字符串, 返回小于 0 表示前面的字符串小于后面的字符串。如下:

```
public class Test{  
    public static void main(String[] args) {  
        String str = "这里是 JAVA 私塾";  
        String str2 = "这里是 java 私塾";  
        if(str.compareToIgnoreCase(str2)==0){  
            System.out.println("the str 等于 str2");  
        }else if(str.compareToIgnoreCase(str2) > 0){  
            System.out.println("the str 大于 str2");  
        }else if(str.compareToIgnoreCase(str2) < 0){  
            System.out.println("the str 小于 str2");  
        }  
    }  
}
```

运行结果: the str 等于 str2

**方法 equals(Object s):** 比较两个 String 对象的值是否相等, 这个是区分大小写的

**方法 equalsIgnoreCase(String s):** 比较两个 String 对象的值是否相等, 忽略大小写

String 的比较要使用 equals() 方法, 不能使用 "=="

### 2.3: 查找字符串中的字符或子串

查找字符串 (String) 中的字符或子串

#### 方法 indexOf

四种重载方法 indexOf() 返回第一次找到时的下标

如果没有找到, 则返回-1。实例:

```
String name = "CoolTools";  
System.out.println (name.indexOf("oo"));
```

#### 方法 lastIndexOf

- public int lastIndexOf(int ch, int fromIndex)
- 从指定位置往回查找, 返回找到的最大的字符下标位置
- 即返回满足下面条件的最大值:  
(this.charAt(k) == ch) && (k <= fromIndex)
- 返回-1: 如果当前字符串不含该字符

**方法 startsWith(String prefix):** 测试此字符串是否以指定的前缀开始, 如下:

```
public class Test{
```

```
public static void main(String[] args) {  
    String str = "这里是 Java 私塾";  
    String str2 = "Java";  
  
    System.out.println(str.startsWith(str2));  
}  
}
```

运行结果: false

**方法 `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。

```
public class Test{  
    public static void main(String[] args) {  
        String str = "这里是 Java 私塾";  
        String str2 = "Java";  
  
        System.out.println(str.startsWith(str2, 3));  
    }  
}
```

运行结果: true

**方法 `endsWith(String suffix)`:** 测试此字符串是否以指定的后缀结束

## 2.4: 从当前字符串中抽取子字符串

**方法 `substring`**

- `substring(int beginIndex)`  
返回新的字符串: 当前字符串的子串  
该子串从指定的位置开始, 并一直到当前字符串结束为止
- `substring(int beginIndex, int endIndex)`  
返回新的字符串: 当前字符串的子串  
该子串从指定的位置(`beginIndex`)开始, 到指定的位置(`endIndex - 1`)结束

例如:

```
"unhappy".substring(2) 返回    "happy"  
"Harbison".substring(3) 返回    "bison"  
"emptiness".substring(9) 返回    "" (空串)  
"emptiness".substring(10) 返回    StringIndexOutOfBoundsException  
"hamburger".substring(4, 8) 返回    "urge"  
"smiles".substring(1, 5) 返回    "mile"
```

## 2.5: 字符串拼接

**方法 `concat`**

- 拼接两个字符串, 并返回一个新字符串

- 源字符串不会被修改
- `s1.concat( s2 )`  
返回字符串 `s1` 和 `s2` 拼接的结果

实例:

```
String s1 = "ABC";  
String s2 = "XYZ";  
s1 = s1.concat(s2); // 等同于 s1 = s1 + s2;
```

## 2.6: 类 String 的成员方法 `valueOf`

### 静态(`static`)成员方法 `valueOf`

- 将参数的值转化成相应的字符串
- `valueOf(char[ ] data)`  
返回 `new String(data)`;
- `valueOf(char[ ] data, int offset, int count)`  
返回 `new String(data, offset, count)`;
- 其它 `valueOf` 方法的参数的类型: `boolean`、`char`、`int`、`long`、`float`、`double` 和 `Object`  
对象还可以通过方法 `toString` 转化成字符串

## 2.7: 字符串分解

### 方法 `split(String regex)`

根据给定正则表达式的匹配拆分此字符串, 得到拆分好的字符串数组, 示例如下:

```
public class Test{  
    public static void main(String[] args) {  
        String str = "这里, 是 Java, 私塾";  
        String tempS[] = str.split(",");//按照 “,” 对字符串进行拆分  
        for(int i=0;i<tempS.length;i++){  
            System.out.println("tempS["+i+"]=== "+tempS[i]);  
        }  
    }  
}
```

运行结果:

```
tempS[0]===这里  
tempS[1]===是Java  
tempS[2]===私塾
```

注意:

(1) 如果用 “.” 作为分隔的话, 必须是如下写法: `String.split("\\. ")`, 这样才能正确的分隔开, 不能用 `String.split(". ")`;

```
public class Test{  
    public static void main(String[] args) {  
        String str = "这里.是Java.私塾";  
        String tempS[] = str.split("\\.");//按照 "." 对字符串进行拆分  
        for(int i=0;i<tempS.length;i++){
```

```
        System.out.println("tempS["+i+"]==" + tempS[i]);
    }
}
}
```

(2) 如果用“|”作为分隔的话, 必须是如下写法: `String.split("\\|")`, 这样才能正确的分隔开, 不能用 `String.split("|")`;

因为“.”和“|”都是转义字符, 必须得加“\\”;

```
public class Test{
    public static void main(String[] args) {
        String str = "这里|是Java|私塾";
        String tempS[] = str.split("\\|");//按照"|"对字符串进行拆分
        for(int i=0;i<tempS.length;i++){
            System.out.println("tempS["+i+"]==" + tempS[i]);
        }
    }
}
```

### StringTokenizer 类

与 `String` 的 `split` 方法功能类似, 也是用来分解字符串, `StringTokenizer` 是出于兼容性的原因而被保留的遗留类 (在新代码中并不鼓励使用它)。建议所有寻求此功能的人使用 `String` 的 `split` 方法或 `java.util.regex` 包。所以这里就不多说了。

## 2.8: 其它 String 方法

### 方法 `replace( char1, char2 )`

- 返回一个新的字符串, 它是将 `s1` 中的所有 `char1` 替换成的结果 `char2`
- 源字符串没有发生变化
- 如果 `s1` 不含 `char1`, 则返回源字符串的引用, 即 `s1`

实例:

- `"mesquite in your cellar".replace( 'e', 'o' )` 结果返回 `"mosquito in your collar"`
- `"JonL".replace( 'q', 'x' )` 结果返回 `"JonL"` (没有发生变化)

### 方法 `toUpperCase`

- 返回对应的新字符串, 所有小写字母都变为大写的, 其它的不变
- 如果没有字符被修改, 则返回源字符串的引用
- 类似方法 `s1.toLowerCase` (所有大写字母都变为小写字母)

### 方法 `trim()`

- 返回新字符串, 截去了源字符串最前面和最后面的空白符
- 如果字符串没有被改变, 则返回源字符串的引用

### 方法 `toString()`

- 由于 `s1` 本身就是字符串了, 所以返回 `s1` 本身
- 其它引用类型也可以通过方法 `toString`, 生成相应的字符串

### 方法 `toCharArray()`



- 将字符串转换成字符数组

#### 方法 intern

- 返回具有相同内容的字符串的引用
- 如果字符串池含有该内容的字符串, 则返回字符串池中具有该内容的字符串的引用
- 如果字符串池没有字符串的内容与其相同, 则在字符串池中创建具有该内容的字符串, 再返回新创建的字符串的引用

#### 字符串池

- 组成: 字符串直接量 以及 由方法 intern 产生的字符串
- 字符串池中的字符串 s 与 t : s 与 t 具有相同内容 (s.equals(t)) 当且仅当指向 s 与 t 的同一个字符串 (s.intern() == t.intern())
- 可以采用这个机制加速字符串是否相等的判定

## 三: 正则表达式和相关的 String 方法

---

完整的正则表达式语法是比较复杂的, 这里只是简单地入个门, 更多的正则表达式请参考相应的书籍或者文章。不过没有什么大的必要, 常见的正则表达式基本都是写好了的, 拿来用就可以了。

### 1: 正则表达式是什么

---

在编写处理字符串的程序时, 经常会有查找符合某些复杂规则的字符串的需要, 正则表达式就是用于描述这些规则的工具。换句话说, 正则表达式就是记录文本规则的代码。

回忆一下在 Windows 下进行文件查找, 查找的通配符也就是\*和?。如果你想查找某个目录下的所有的 Word 文档的话, 你会搜索\*.doc。在这里, \*会被解释成任意的字符串。和通配符类似, 正则表达式也是用来进行文本匹配的工具, 只不过比起通配符, 它能更精确地描述你的需求——当然, 代价就是更复杂——比如你可以编写一个正则表达式, 用来查找所有以 0 开头, 后面跟着 2-3 个数字, 然后是一个连字号“-”, 最后是 7 或 8 位数字的字符串 (像 010-12345678 或 0123-1234567)。

简言之正则表达式是用于进行文本匹配的工具, 也是一个匹配的表达式。

### 2: 正则表达式基础入门

---

学习正则表达式的最好方法是从例子开始, 理解例子之后再自己对例子进行修改, 实验。下面给出了不少简单的例子, 并对它们作了详细的说明。

假设你在一篇英文小说里查找 hi, 你可以使用正则表达式 hi。

这是最简单的正则表达式了, 它可以精确匹配这样的字符串: 由两个字符组成, 前一个字符是 h, 后一个是 i。通常, 处理正则表达式的工具会提供一个忽略大小写的选项, 如果选中了这个选项, 它可以匹配 hi, HI, Hi, hI 这四种情况中的任意一种。

不幸的是, 很多单词里包含 hi 这两个连续的字符, 比如 him, history, high 等等。用 hi 来查找的话, 这里边的 hi 也会被找出来。如果要精确地查找 hi 这个单词的话, 我们应该使用\bhi\b。

\b 是正则表达式规定的一个特殊代码 (某些人叫它元字符, meta character), 代表着单词的开头或结尾, 也就是单词的分界处。虽然通常英文的单词是由空格或标点符号或换行

来分隔的, 但是**\b**并不匹配这些单词分隔符中的任何一个, 它只匹配一个位置。

假如你要找的是 hi 后面不远处跟着一个 Lucy, 你应该用**\bhi\b.\*\bLucy\b**。

**.**是另一个元字符, 匹配除了换行符以外的任意字符。**\***同样是元字符, 不过它代表的不是字符, 也不是位置, 而是数量——它指定**\***前边的内容可以连续重复出现任意次以使整个表达式得到匹配。因此, **.\***连在一起就意味着任意数量的不包含换行符。现在**\bhi\b.\*\bLucy\b**的意思就很明显了: 先是一个单词 hi, 然后是任意个任意字符(但不能是换行符), 最后是 Lucy 这个单词。

如果同时使用其它的一些元字符, 我们就能构造出功能更强大的正则表达式。比如下面这个例子:

**0\d\d-\d\d\d\d\d\d\d\d** 匹配这样的字符串: 以 0 开头, 然后是两个数字, 然后是一个连字号 “-”, 最后是 8 个数字(也就是中国的电话号码。当然, 这个例子只能匹配区号为 3 位的情形)。

这里的**\d**是一个新的元字符, 匹配任意的数字(0, 或 1, 或 2, 或……)。**-**不是元字符, 只匹配它本身——连字号。

为了避免那么多烦人的重复, 我们也可以这样写这个表达式: **0\d{2}-\d{8}**。这里**\d**后面的**{2}**和**{8}**的意思是前面**\d**必须连续重复匹配 2 次和 8 次。

### 3: 在 Java 中运行正则表达式

---

在 Java 中的 String 类中有好几个方法都跟正则表达式有关, 最典型的就是方法 **matches(String regex)**

告知此字符串是否匹配给定的正则表达式

使用这个方法来测试和运行上面学到的正则表达式, 示例如下:

```
public class Test{
    public static void main(String[] args) {
        String str = "010-86835215";
        System.out.println("str 是一个正确的电话号码? 答案是: "
            +str.matches("0\d{2}-\d{8}"));
    }
}
```

运行结果: str 是一个正确的电话号码? 答案是: true

**注意:** 由于在 Java 里面 “\” 需要转义, 应该变成 “\\”。

在 Java 中, 有专门进行正则表达式的类, 在 **java.util.regex** 包里面。

#### 3.1: java.util.regex.Pattern 类

Pattern 类是正则表达式的编译表示形式。

指定为字符串的正则表达式必须首先被编译为此类的实例。然后, 可将得到的模式用于创建 Matcher 对象, 依照正则表达式, 该对象可以与任意字符序列匹配。执行匹配所涉及

的所有状态都驻留在匹配器中, 所以多个匹配器可以共享同一模式

常见方法如下:

**static Pattern compile(String expression)**

**static Pattern compile(String expression, int flags):** 编译正则表达式字符串到 pattern 对象用以匹配的快速处理

参数:

expression 正则表达式

flags 下列标志中的一个或多个: CASE\_INSENSITIVE, UNICODE\_CASE, MULTILINE, UNIX\_LINES, DOTALL, and CANON\_EQ

**Matcher matcher(CharSequence input):** 返回一个 matcher 对象, 它可以用来在一个输入中定位模式匹配

**String[] split(CharSequence input)**

**String[] split(CharSequence input, int limit):** 将输入字符串分离成记号, 并由 pattern 来指定分隔符的形式。返回记号数组。分隔符并不是记号的一部分。

参数:

input 分离成记号的字符串

limit 生成的最大字符串数。

### 3.2: java.util.regex.Matcher 类

Mathcer 类通过解释 Pattern 对字符序列执行匹配操作的引擎。

常见方法如下:

**boolean matches():** 返回输入是否与模式匹配

**boolean lookingAt():** 如果输入的起始匹配模式则返回 True

**boolean find()**

**boolean find(int start):** 尝试查找下一个匹配, 并在找到匹配时返回 True

参数:

start 开始搜索的索引

**int start():** 返回当前匹配的起始位置位置

**int end():** 返回当前匹配的结尾后位置

**String group():** 返回当前匹配

**int groupCount():** 返回输入模式中的分组数

**int start(int groupIndex)**

**int end(int groupIndex)**

返回一个给定分组当前匹配中的起始位置和结尾后位置

参数:

groupIndex 分组索引 (从 1 开始), 0 表示整个匹配

**String group(int groupIndex):** 返回匹配一个给定分组的字符串

参数:

groupIndex 分组索引 (从 1 开始), 0 表示整个匹配

**String replaceAll(String replacement)**

### String replaceFirst(String replacement)

返回从 matcher 输入得到的字符串，但已经用替换表达式替换所有或第一个匹配参数：

replacement 替换字符串

### Matcher reset()

### Matcher reset(CharSequence input)

复位 matcher 状态。

## 3.3: Java 中操作正则表达式示例

```
public class Test{
    public static void main(String[] args) {
        String str = "010-86835215";
        Pattern p = Pattern.compile("0\\d{2}-\\d{8}");
        Matcher m = p.matcher(str);
        boolean flag = m.matches();
        System.out.println("str是一个正确的电话号码？答案是: "+flag);
    }
}
```

运行结果：str 是一个正确的电话号码？答案是：true

## 4: 元字符

现在你已经知道几个很有用的元字符了，如**\b**、**.**、**\***，还有**\d**。当然还有更多的元字符可用，比如**\s** 匹配任意的空白符，包括空格，制表符(Tab)，换行符，中文全角空格等。**\w** 匹配字母或数字或下划线或汉字等。

常用的元字符	
代码	说明
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	匹配任意的空白符
\d	匹配数字
\b	匹配单词的开始或结束
^	匹配字符串的开始
\$	匹配字符串的结束

下面来试试更多的例子：

### 例 1: \ba\w\*\b

匹配以字母 a 开头的单词——先是某个单词开始处(**\b**)，然后是字母 a，然后是任意数量的字母或数字(**\w\***)，最后是单词结束处(**\b**)（好吧，现在我们说说正则表达式里的单词是什么意思吧：就是几个连续的**\w**。不错，这与学习英文时要背的成千上万个同名的东西的

确关系不大)。

#### 例 2: \d+

匹配 1 个或更多连续的数字。这里的+是和\*类似的元字符, 不同的是\*匹配重复任意次(可能是 0 次), 而+则匹配重复 1 次或更多次。

#### 例 3: \b\w{6}\b

匹配刚好 6 个字母/数字的单词。

#### 例 4: ^\d{5,12}\$

匹配必须为 5 位到 12 位数字的字符串

元字符^ (和数字 6 在同一个键位上的符号) 以及\$和\b有点类似, 都匹配一个位置。^匹配你要用来查找的字符串的开头,\$匹配结尾。这两个代码在验证输入的内容时非常有用, 比如一个网站如果要求你填写的 QQ 号时, 可以使用。

这里的{5,12}和前面介绍过的{2}是类似的, 只不过{2}匹配只能不多不少重复 2 次,{5,12}则是重复的次数不能少于 5 次, 不能多于 12 次, 否则都不匹配。

因为使用了^和\$, 所以输入的整个字符串都要用来和\d{5,12}来匹配, 也就是说整个输入必须是 5 到 12 个数字, 因此如果输入的 QQ 号能匹配这个正则表达式的话, 那就符合要求了。

### 5: 重复

已经看过了前面的\*, +, {2}, {5,12}这几个匹配重复的方式了。下面是正则表达式中所有的限定符(指定数量的代码, 例如\*, {5,12}等):

常用的限定符	
代码/语法	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复 n 次
{n, }	重复 n 次或更多次
{n, m}	重复 n 到 m 次

下面是一些使用重复的例子:

Windows\d+ : 匹配 Windows 后面跟 1 个或更多数字

13\d{9} : 匹配 13 后面跟 9 个数字(中国的手机号)

### 6: 字符类

要想查找数字, 字母或数字, 空白是很简单的, 因为已经有了对应这些字符集合的元字符, 但是如果你想匹配没有预定义元字符的字符集合(比如元音字母 a, e, i, o, u), 应该怎么办?

很简单, 你只需要在中括号里列出它们就行了, 像[aeiou]就匹配任何一个英文元音字母, [.?!]匹配标点符号(. 或?或!) (英文语句通常只以这三个标点结束)。

我们也可以轻松地指定一个字符范围, 像[0-9]代表的含意与\d 就是完全一致的: 一位数字, 同理[a-zA-Z\_]也完全等同于\w (如果只考虑英文的话)。

## 7: 常见正则表达式

---

(1): 检测是否 Email 地址

```
^([\\w-\\.]+)@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.)|((\\w-)+\\.))([a-zA-Z]{2,4}|[0-9]{1,3})(\\w)?$
```

(2): 判断输入的字符串只包含汉字

```
^[\u4e00-\u9fa5]+$
```

(3): 匹配 3 位或 4 位区号的电话号码, 其中区号可以用小括号括起来, 也可以不用, 区号与本地号间可以用连字号或空格间隔, 也可以没有间隔

```
^\\(0\\d{2}\\)\\[-]?\\d{8}$|^0\\d{2}\\[-]?\\d{8}$|^\\(0\\d{3}\\)\\[-]?\\d{7}$|^0\\d{3}\\[-]?\\d{7}$
```

(4): 判断输入的字符串是否是一个合法的手机号, 这个不完全, 只是 13 开头的

```
^13\\d{9}$
```

(5): 判断输入的字符串只包含数字, 可以匹配整数和浮点数

```
^-?\\d+$|^(-?\\d+)(\\.\\d+)?$
```

(6): 匹配非负整数

```
^\\d+$
```

(7): 判断输入的字符串只包含英文字母

```
^[A-Za-z]+$
```

(8): 判断输入的字符串是否只包含数字和英文字母

```
^[A-Za-z0-9]+$
```

好了, 学到这里对基本的正则表达式就有了基本的认识, 下面来看看 String 里面跟正则表达式有关的几个方法的使用。

## 8: 相关 String 类的方法

---

方法 matches(String regex)

告知此字符串是否匹配给定的正则表达式

方法 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

示例如下:

```
public class Test{
    public static void main(String[] args) {
        String str = "这里是Java私塾,到Java私塾学习Java";
        String regStr = "Java";
        str = str.replaceAll(regStr, "ABC");
    }
}
```

```
        System.out.println("str="+str);
    }
}
```

运行结果:

str=这里是 ABC 私塾, 到 ABC 私塾学习 ABC

**方法 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第一个子字符串

**方法 split(String regex)**

根据给定正则表达式的匹配拆分此字符串, 它是全部拆分

**方法 split(String regex, int limit)**

根据匹配给定的正则表达式来拆分此字符串, limit 是限制拆分的次数, 实际分解的次数是 limit-1 次, limit 就是分解后数组的 length。

示例如下:

```
public class Test{
    public static void main(String[] args) {
        String str = "这里,是Java,私塾";
        String tempS[] = str.split(",",2);
        for(int i=0;i<tempS.length;i++){
            System.out.println("tempS["+i+"]==" +tempS[i]);
        }
    }
}
```

运行结果:

tempS[0]==这里  
tempS[1]==是 Java,私塾

## 四: StringBuffer 类和 StringBuilder 类

---

前面学到过 String 类有一个重要的特点, 那就是 String 的值是不可变的, 这就导致每次对 String 的操作都会生成新的 String 对象, 不仅效率低下, 而且大量浪费有限的内存空间。那么对于经常要改变值的字符串应该怎样操作呢?

答案就是使用 StringBuffer 和 StringBuilder 类, 这两个类功能基本相似, 区别主要在于 StringBuffer 类的方法是多线程安全的 (多线程的课程在后面会学习到), 而 StringBuilder 不是线程安全的, 相比而言 StringBuilder 类会略微快一点。

### 1: StringBuffer 类

---

类 String

- 字符串(String)对象一旦创建, 其内容不能再被修改 (read-only)

类 StringBuffer

- StringBuffer 对象的内容是可以被修改的
- 除了字符的长度之外, 还有容量的概念
- 通过动态改变容量的大小, 加速字符管理



### 1.1: StringBuffer 的构造方法

```
buf1 = new StringBuffer();  
    - 创建空的 StringBuffer 对象, 初始容量为 16 字符  
buf2 = new StringBuffer( 容量 );  
    - 创建空的 StringBuffer 对象, 指定容量大小。  
buf3 = new StringBuffer( myString );  
    - 创建含有相应字符序列的 StringBuffer 对象, 容量为 myString.length() +  
    16  
    - 实例:  
      StringBuffer b = new StringBuffer("hello");
```

### 1.2: StringBuffer 的常用方法

- 方法 length()
  - 返回 StringBuffer 的长度
- 方法 capacity()
  - 返回 StringBuffer 的容量
- 方法 setLength(int newLength)
  - 增加或减小 StringBuffer 的长度
- 方法 charAt(int index)
  - 返回 StringBuffer 对象中指定位置的字符
- 方法 setCharAt(int index, char ch)
  - 设置 StringBuffer 对象中指定位置的字符
- 方法 getChars(int srcBegin, int srcEnd,  
Char[] dst, int dstBegin)
  - 将 StringBuffer 对象中指定的字符子序列, 拷贝到指定的字符数组(dst)
- 方法 reverse()
  - 将 StringBuffer 对象中的字符序列按逆序方式排列, 可用作字符串倒序
- 11 种 append(...) 方法
  - 允许数值类型的值添加到 StringBuffer 对象中
- 10 种 insert 方法
  - 允许将各种数据插到 StringBuffer 对象的指定位置
- 方法 delete(int start, int end) 和 deleteCharAt(int index)
  - 允许删除 StringBuffer 对象中的指定字符

其中最常用的恐怕就要算 append 方法和 toString 方法了, 如下示例:

```
public class Test{  
    public static void main(String[] args) {  
        StringBuffer buffer = new StringBuffer();  
        buffer.append("这里");  
        buffer.append("是");  
        buffer.append("Java");  
        buffer.append("私塾");  
        System.out.println("buffer==" + buffer.toString());  
    }  
}
```



```
}
```

运行结果: buffer==这里是 Java 私塾

## 2: StringBuilder 类

---

StringBuilder 类是一个可变的字符序列。此类提供一个与 StringBuffer 兼容的 API, 但不保证同步。该类被设计用作 StringBuffer 的一个简易替换, 用在字符串缓冲区被单个线程使用的时候 (这种情况很普遍)。如果可能, 建议优先采用该类, 因为在大多数实现中, 它比 StringBuffer 要快。

它的功能基本等同于 StringBuffer 类, 就不再赘述了。

```
public class Test{  
    public static void main(String[] args) {  
        StringBuilder builder = new StringBuilder();  
        builder.append("这里");  
        builder.append("是");  
        builder.append("Java");  
        builder.append("私塾");  
        System.out.println("buffer==" + builder.toString());  
    }  
}
```

运行结果: builder==这里是 Java 私塾

## 五: Math 类

---

Java 中的数学 (Math) 类是 final 类, 不可继承。

其中包含一组静态方法和两个常数

### 1: 常数

PI : double, 圆周率

E : double, 自然对数

### 2: 截取 (注意方法的返回类型)

double ceil(double d)

- 返回不小于 d 的最小整数

double floor(double d)

- 返回不大于 d 的最大整数

int round(float f)

- 返回四舍五入后的整数

long round(double d)

- 返回四舍五入后的整数

### 3: 变换 (int long float 各种类型相似)

double abs(double d)

- 返回绝对值

double min(double d1, double d2)

- 返回两个值中较小的值

double max(double d1, double d2)

-返回两个值中较大的值

#### 4: 对数

`double log(double d)`

-自然对数

`double exp(double d)`

-E 的指数

#### 5: 其它

`double sqrt(double d)`

-返回平方根

`double random()`

-返回随机数

还有三角函数的运算等, 请参考 JDK 文档

示例如下: 问题: 请问有 101 条记录, 按照每组 10 条记录进行分组, 应该分多少组?

```
public class Test{
    public static void main(String[] args) {
        int records = 101;//共有 101 条记录
        final int GROUP_NUM = 10;//每组 10 条

        int groups = (int)Math.ceil(1.0*records/GROUP_NUM);//注意这里的 1.0, 目的是要把类型变成 double 型的, 而不是 int/int, 结果还是 int, 就错了。
        System.out.println("应该分的组数为="+groups);
    }
}
```

运行结果: 应该分的组数为=11

## 六: Java 日期操作的类

---

### 1: Date 类

---

`java.util` 包里面的 `Date` 类, 是 Java 里面进行日期操作常用类。`Date` 类用来表示特定的瞬间, 精确到毫秒。

#### 1.1: 如何初始化 Date

**构造方法: `Date()`**

分配 `Date` 对象并初始化此对象, 以表示分配它的时候的当前时间 (精确到毫秒)。使用 `Date` 类得到当前的时间。

**构造方法: `Date(long date)`**

分配 `Date` 对象并初始化此对象, 以表示自从标准基准时间 (称为“历元 (epoch)”, 即 1970 年 1 月 1 日 00:00:00 格林威治时间) 以来的指定毫秒数。

#### 2.2: 常用方法

**方法: `after(Date when)`**

测试此日期是否在指定日期之后

**方法: `before(Date when)`**

测试此日期是否在指定日期之前

方法: `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。

### 2.3: 示例: 简单的性能测试——监控一段代码运行所需要的时间

```
public class Test {  
    public static void main(String args[]) {  
        long d1 = new Date().getTime();//得到此时的时间  
        int sum = 0;  
        for(int i=1;i<=1000000;i++){//看清楚了,可是一百万次  
            sum +=i;  
        }  
        System.out.println("从1加到1000000的和="+sum);  
        long d2 = new Date().getTime();//得到此时的时间  
        System.out.println("从1加到1000000所耗费的时间是="+ (d2-d1) + "毫秒  
");  
    }  
}
```

运行结果:

从1加到1000000的和=1784293664

从 1 加到 1000000 所耗费的时间是=20 毫秒

## 2: DateFormat 类和 SimpleDateFormat 类

---

在 `java.text` 包中的 `DateFormat` 类, 是日期/时间格式化子类的抽象类, 它以与语言无关的方式格式化并解析日期或时间。日期/时间格式化子类 (如 `SimpleDateFormat`) 允许进行格式化 (也就是日期——>文本)、解析 (文本——> 日期) 和标准化。

由于 `DateFormat` 是个抽象类, `SimpleDateFormat` 类是它的子类, 所以下面就主要按照 `SimpleDateFormat` 类来讲解。

### 2.1: 如何初始化

这里只讲述最常用到的构造方法, 更多的请参看 JDK 文档。

**构造方法: `SimpleDateFormat(String pattern)`**

用给定的模式和默认语言环境的日期格式符号构造 `SimpleDateFormat`

### 2.2 日期和时间模式

日期和时间格式由 *日期和时间模式* 字符串指定。在日期和时间模式字符串中, 未加引号的字母 'A' 到 'Z' 和 'a' 到 'z' 被解释为模式字母, 用来表示日期或时间字符串元素。文本可以使用单引号 (') 引起来, 以免进行解释。""" 表示单引号。所有其他字符均不解释; 只是在格式化时将它们简单复制到输出字符串, 或者在解析时与输入字符串进行匹配。

定义了以下模式字母 (所有其他字符 'A' 到 'Z' 和 'a' 到 'z' 都被保留):

字母	日期或时间元素	表示	示例
G	Era	标志符	Text
y	年	Year	1996;
M	年中的月份	Month	July;
w	年中的周数	Number	27
W	月份中的周数	Number	2
D	年中的天数	Number	189
d	月份中的天数	Number	10
F	月份中的星期	Number	2
E	星期中的天数	Text	Tuesday;
a	Am/pm	标记	Text
H	一天中的小时数 (0-23)	Number	0
k	一天中的小时数 (1-24)	Number	24
K	am/pm	中的小时数 (0-11)	Number
h	am/pm	中的小时数 (1-12)	Number
m	小时中的分钟数	Number	30
s	分钟中的秒数	Number	55
S	毫秒数	Number	978
z	时区	General	time
Z	时区	RFC	822

### 2.3: 常用方法

方法: `parse(String source)`

从给定字符串的开始解析文本, 以生成一个日期

方法: `format(Date date)`

将一个 `Date` 格式化为日期/时间字符串

这里只讲述最常用的方法, 更多的方法请参看 JDK 文档。

### 2.4: 示例

```
import java.util.*;
import java.text.*;

public class Test {
    public static void main(String args[]) {
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");

        Date d = new Date();
        //把当前时间转换成为我们熟悉的时间表达格式
        String str = df.format(d);

        System.out.println("当前时间是: "+str);

        //然后再把字符串格式的日期转换成为一个Date类
        try {
            Date d2 = df.parse("2008-08-08 08:08:08 888");
        }
    }
}
```

```
        System.out.println("北京奥运会开幕时间是: "+d2.getTime());
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
}
```

运行结果:

当前时间是: 2008-07-22 00:57:45 612

北京奥运会开幕时间是: 1218154088888

## 2.5: 说明

虽然 JDK 文档上说 Date 的毫秒值, 是相对于格林威治时间 1970 年 1 月 1 号的 0 点, 但实际测试, 这个 Date 是跟时区相关的, 也就是说在中国测试这个基准值应该是 1970 年 1 月 1 日的 8 点, 不过这个不影响我们的处理, 因为只要是同一个基准时间就可以了, 而不用担心具体是多少, 见下面的示例:

```
public class Test {
    public static void main(String args[]) {
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");

        Date d = new Date(0L); //把时间设为0, 表示到基准时间
        //然后转换成字符串看看是什么时候
        String str = df.format(d);

        System.out.println("基准时间是: "+str);
    }
}
```

运行结果:

基准时间是: 1970-01-01 08:00:00 000

## 3: Calendar 类

java.util 包中的 Calendar 类是 Java 里面另外一个常用的日期处理的类。Calendar 类是一个抽象类, 它为特定瞬间与一组诸如 YEAR、MONTH、DAY\_OF\_MONTH、HOUR 等 日历字段之间的转换提供了一些方法, 并为操作日历字段 (例如获得下星期的日期) 提供了一些方法。

### 3.1: 如何初始化

Calendar 类是通过一个静态方法 getInstance() 来获取 Calendar 实例。返回的 Calendar 基于当前时间, 使用了默认时区和默认语言环境

如下: Calendar c = Calendar.getInstance();

### 3.2: 使用 Calendar 对日期进行部分析取

Calendar 类一个重要的功能就是能够从日期里面按照要求析取出数据, 如: 年、月、日、星期等等。

方法: get(int field)

返回给定日历字段的值

示例如下:

```
public class Test {
    public static void main(String args[]) {
```

```
Calendar c = Calendar.getInstance();
int year = c.get(Calendar.YEAR);
int month = c.get(Calendar.MONTH); //注意: month特殊, 是从0开始的,
也就是0表示1月
int day = c.get(Calendar.DAY_OF_MONTH);

System.out.println("现在是"+year+"年"+(month+1)+"月"+day+"日");
}
```

运行结果:

现在是 2008 年 7 月 22 日

### 3.3: 使用 Calendar 进行日期运算

这是 Calendar 另外一个常用的功能, 也就是对日期进行加加减减的运算。

方法: add(int field, int amount)

根据日历的规则, 为给定的日历字段添加或减去指定的时间量

示例如下:

```
public class Test {
    public static void main(String args[]) {
        Calendar c = Calendar.getInstance();
        c.add(Calendar.DATE, 12); //当前日期加12天, 如果是-12就表示当前日期减
        去12天

        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH); //注意: month特殊, 是从0开始的,
        也就是0表示1月
        int day = c.get(Calendar.DAY_OF_MONTH);

        System.out.println("在当前日期加12天是"+year+"年"+(month+1)+"月
        "+day+"日");
    }
}
```

运行结果: 在当前日期加 12 天是 2008 年 8 月 3 日

### 3.4: 为 Calendar 设置初始值

方法 setTime(Date date)

使用给定的 Date 设置此 Calendar 的当前时间

方法 setTimeInMillis(long millis)

用给定的 long 值设置此 Calendar 的当前时间值

```
public class Test {
    public static void main(String args[]) {
        Calendar c = Calendar.getInstance();
```

```
c.setTimeInMillis(1234567890123L);

int year = c.get(Calendar.YEAR);
int month = c.get(Calendar.MONTH); //注意: month特殊, 是从0开始的,
也就是0表示1月
int day = c.get(Calendar.DAY_OF_MONTH);

System.out.println("设置的时间是"+year+"年"+(month+1)+"月"+day+"
日");
}
```

运行结果: 设置的时间是 2009 年 2 月 14 日

## 七: System 类

---

### 1: 命令行参数

当 Java 程序启动时, 可以添加 0 或多个命令行参数 (Command-line Arguments)。不管使用双引号与否都作为字符串自动保存到 main 函数的参数中。参数之间用空格分隔。

```
public class Test{
    public static void main(String args[]){
        System.out.println(args.length);
        for(int i=0; i<args.length; i++){
            System.out.println(args[i]);
        }
    }
}
```

本段代码可以使用下面语句测试: java Test 这里 是 Java 私塾

运行结果:

```
4
这里
是
Java
私塾
```

### 2: 系统属性

(1): 系统属性 (System Properties) 是 Java 提供的另外一种运行时给程序提供参数的方法。属性 (Property) 是名字和价值之间的映射, 这里名字和价值都只能是 String。

(2): Java 有一个类 Properties 描述了名字和价值之间的映射。

System.getProperties 方法返回系统的 Properties 对象。

System.getProperty(String propertyName) 方法返回对应名字属性的值。

System.getProperty(String name, String value) 重载方法当没有 name 指定的属性时, 返回 value 指定的缺省值。

(3): 每一个 JVM 提供了一组缺省属性, 可以通过 System.getProperties 方法的帮助文档了解到。基本类型的包装类中包含静态方法, 用于将属性值转换成对应类型的值。如 Boolean.getBoolean(String); Long.getLong(String)。String 参数这里是属性名字, 如

果属性不存在, 那么返回 false 或 null。

(4): Properties 类的对象包含两个主要方法:

getProperty(String Name)和 getProperty(String Name, String value)方法返回对应名字属性的值。

properNames() 列举系统的一组对象名字, 进而可以列举系统的属性值。

例如:

```
import java.util.Properties;
import java.util.Enumeration;
public class TestProperties{
    public static void main(String args[]){
        Properties props = System.getProperties();
        Enumeration prop_names = props.propertyNames(); //这是个早期的集合类
        while(prop_names.hasMoreElements()){
            String prop_name = (String) prop_names.nextElement();
            String property = props.getProperty(prop_name);
            System.out.println("Property '" + prop_name + "' is '" + property + "'");
        }
    }
}
```

prop\_names 是 Enumeration 是枚举类型的对象, 包含一组属性名, 允许程序循环列出。

hasMoreElements 方法检测是否还有其它元素, nextElement 返回程序的下一个元素。

运行: java -DMyProp=theValue TestProperties

注意: 严格的格式 -D 后没有空格

运行结果: 很多的 Java 系统级属性, 下面列出部分, 太多了, 你可以把程序运行一下:

```
Property 'java.runtime.name' is 'Java(TM) SE Runtime Environment'
Property 'sun.boot.library.path' is 'c:\jdk1.6.0_01\jre\bin'
Property 'java.vm.version' is '1.6.0_01-b06'
Property 'java.vm.vendor' is 'Sun Microsystems Inc.'
Property 'java.vendor.url' is 'http://java.sun.com/'
.....
```

### 3: 控制台输入输出

许多应用程序要与用户进行文本 I/O (输入/输出) 交互, 标准输入是键盘; 标准输出是终端窗口。Java SDK 支持控制台 I/O 使用三个 java.lang.System 类中定义的变量:

System.out 是一个 PrintStream 对象, 初始引用启动 Java 的终端窗口。

System.in 是一个 InputStream 对象, 初始指向用户键盘。

System.err 是一个 PrintStream 对象, 初始引用启动 Java 的终端窗口。

这三个对象都可以重新定向(如文件): System.setOut\setIn\setErr。

往标准输出写东西使用 PrintStream 对象的 println 或 print 方法。print 方法输出参数; 但 println 方法输出参数并追加一个换行符。

println 或 print 方法都对原始类型进行重载, 同时还重载了 char[] 和 Object, String。



参数是 Object 时, 调用参数的 toString 方法。

输出例子:

```
public class Test {
    public static void main(String args[]) {
        char c[] = { 'a', 'b', 'c' };
        System.out.println(c);
    }
}
```

运行结果: abc

输入例子:

```
public class Test {
    public static void main(String args[]) {
        String s = "";
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader in = new BufferedReader(ir);
        System.out.println("Ctrl+z to exit");
        try {
            s = in.readLine();
            while (s != null) {
                System.out.println("Read:" + s);
                s = in.readLine();
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行的时候从控制台输入数据, 然后回车, 就看到具体读入的值的输出了。

这里先看看, 涉及到后面要学习的 I/O 的知识。

#### 4: 格式化输出 printf

从 JDK5.0 开始, Java 里面提供了 C 风格的格式化输出方法 —— printf,

比如输出一个加法算式, JDK5.0 版本以前的写法是:

```
public class Test {
    public static void main(String[] args) {
        int x = 5;
        int y = 7;
        int nSum = x + y;
        System.out.println(x + " + " + y + " = " + nSum);
    }
}
```

运行结果: 5 + 7 = 12

而在JDK5.0以后版本中可以写为:

```
public class Test {
    public static void main(String[] args) {
        int x = 5;
        int y = 7;
        int nSum = x + y;
        System.out.printf("%d + %d = %d\n", x, y, nSum);
    }
}
```

以上两种写法的输出结构是一样的, 即“5 + 7 = 12”。

这种改变不仅仅是形式上的, printf 还可以提供更为灵活、强大的输出功能, 比如限定按照两位整数的形式输出, 可以写为:

```
public class Test {
    public static void main(String[] args) {
        int x = 5;
        int y = 7;
        int nSum = x + y;
        System.out.printf("%02d + %02d = %02d\n", x, y, nSum);
    }
}
```

运行输出结果将是“05 + 07 = 12”。

其实这个功能在 Java 里面并没有什么大用, 具体的 printf 格式化字符串格式请参见 JDK 文档中的具体说明。

## 5: 属性文件

后缀为“.properties”的文件在 Java 中被称为属性文件, 是 Java 中历史悠久, 使用频繁的配置文件(或者叫资源文件)格式。

属性文件的基本格式为: key=value

Java 提供现成的 API 来读取 properties 文件的内容, 并进行解析, 所以使用非常方便。运行时候只要把“.properties”文件放到 classpath 下就可以了。

示例如下: 假设有一个 test.properties 如下:

#表示注释

test1=This is Java properties

#默认的 properties 是不认中文的, 需要编码(后面会学到), 所以这里用英文测试

test2=Welcome to Java World

### Java 读取

Java 有好几种方法来读取 properties 文件, 这里演示其中一种:

```
public class Test {
    public static void main(String args[]) {
        try {
            String name = "test.properties";
```

```
        InputStream in = new BufferedInputStream(new
FileInputStream(name));
        Properties p = new Properties();
        p.load(in);

        System.out.println("test1的值==" + p.getProperty("test1"));
        System.out.println("test2的值==" + p.getProperty("test2"));
    } catch (Exception err) {
        err.printStackTrace();
    }
}
```

运行结果：

```
test1的值==This is Java properties
test2的值==Welcome to Java World
```

## 作业

---

- 1: 设计一个银行帐户类, 具有户名, 帐号, 余额等 属性, 以及存款、取款等方法, 并对此类进行测试
- 2: 写一个方法, 功能: 定义一个一维的 `int` 数组, 长度为 3, 把任意三个整数 `a,b,c` 赋值给数组,然后将它们按从小到大的顺序输出 (使用冒泡排序)
- 3: 有一分数序列:  $2/1, 3/2, 5/3, 8/5, 13/8, 21/13...$  求出这个数列的前 20 项之和。 (不使用数学公式, 要求用递归)
- 4: 输出一个字符数组中的所有字符的所有组合。比如有字符集 `str={A,B,C}`。应输出:  
`A B C AB AC BA BC CA CB ABC ACB BAC BCA CAB CBA`
- 5: 已知两个对像 `String s1,String s2`,已用 `ASC` 码排序好了, 编写程序将两个 `String` 合并, 得到的结果。例如: `s1="abc" s2="abc"` 得 `s="aabbcc"`;结果也是排序的
- 6: 两个乒乓球队进行比赛, 各出三人。甲队为 `a,b,c` 三人, 乙队为 `x,y,z` 三人。已抽签决定比赛名单。有人向队员打听比赛的名单。`a` 说他不和 `x` 比, `c` 说他不和 `x,z` 比, 请编程找出三队赛手的名单。
- 7: 编程: 编写一个截取字符串的方法, 输入为一个字符串和字节数, 输出为按字节截取的字符串。 但是要保证汉字不被截半个, 如"我 ABC"4, 应该截为"我 AB", 输入"我 ABC 汉 DEF", 6, 应该输出为"我 ABC"而不是"我 ABC+汉的半个"。
- 8: 某个公司采用公用电话传递数据, 数据是四位的整数, 在传递过程中是加密的, 加密规则如下: 每位数字都加上 5,然后用和除以 10 的余数代替该数字, 再将第一位和第四位交换, 第二位和第三位交换。请编写一个方法来实现上述加密算法。
- 9: 企业发放的奖金根据利润提成。利润低于或等于 10 万元时, 奖金可提 10%; 利润高于 10 万元, 低于 20 万元时, 低于 10 万元的部分按 10%提成, 高于 10 万元的部分, 可提成 7.5%; 20 万到 40 万之间时, 高于 20 万元的部分, 可提成 5%; 40 万到 60 万之间时高于 40 万元的部分, 可提成 3%; 60 万到 100 万之间时, 高于 60 万元的部分, 可提成 1.5%, 高于 100 万元时, 超过 100 万元的部分按 1%提成, 请编写程序, 输入当月利润, 求应发放奖金总数?
- 10: 老伯伯要带鱼、狗、猫过河到对岸., 有一条船, 只能坐一个人, 老伯每次只能带一样动物过河, 当老伯不在的时候狗会咬猫, 猫会吃鱼., 请问怎么顺序过河呢? 要求: 编写程序, 由程序来推出过河的顺序

## 第七章抽象类和接口

### 教学目标：

- 理解和掌握抽象类
- 掌握接口的基本概念
- 掌握多重接口
- 理解接口的基本思想
- 掌握接口作为类型使用
- 理解接口和抽象类的选择

## 一：抽象类

---

### 1: 抽象类是什么

有时在开发中, 要创建一个体现某些基本行为的类, 并为该类声明方法, 但不能在该类中实现该行为, 取而代之, 在子类中实现该方法。知道其行为的其它类可以在类中实现这些方法。

这种只给出方法定义而不具体实现的方法被称为抽象方法, 抽象方法是没有方法体的, 在代码的表达上就是没有 “{}”。

怎么表示一个方法是抽象的呢? 使用 `abstract` 修饰符来表达抽象。

`abstract` 修饰符可以与类和方法一起使用。被修饰的类不能被实例化, 被修饰的方法必须在包含此方法的类的子类中被实现。

**抽象类**简单地说: 使用 `abstract` 修饰的类就是抽象类。

示例如下:

```
public abstract class Test{//抽象类定义

    public abstract void doItByHand();//抽象方法定义

}
```

### 2: 什么情况下会使用抽象类

例如, 考虑一个 `Drawing` 类。该类包含用于各种绘图设备的方法, 但这些必须以独立平台的方法实现。它不可能去访问机器的录像硬件而且还必须是独立于平台的。其意图是绘图类定义哪种方法应该存在, 但实际上, 由特殊的从属于平台子类去实现这个行为。

正如 `Drawing` 类这样的类, 它声明方法的存在而不是实现, 以及带有对已知行为的方法的实现, 这样的类通常被称做抽象类。通过用关键字 `abstract` 进行标记声明一个抽象类。被声明但没有实现的方法 (即, 这些没有程序体或{}), 也必须标记为抽象。

```
public abstract class Drawing {
    public abstract void drawDot(int x, int y);
    public void drawLine(int x1, int y1, int x2, int y2) {
        // draw using the drawDot() method repeatedly.
    }
}
```

### 3: 抽象类的使用

抽象类不能直接使用, 必须用子类去实现抽象类, 然后使用其子类的实例。然而可以创建一个变量, 其类型是一个抽象类, 并让它指向具体子类的一个实例, 也就是可以使用抽象类来充当形参, 实际实现类作为实参, 也就是多态的应用。

不能有抽象构造方法或抽象静态方法。

`abstract` 类的子类为它们父类中的所有抽象方法提供实现, 否则它们也是抽象类。

```
public class MachineDrawing extends Drawing {
    public void drawDot (int machX, int machY) {
        // 画点
    }
}
```

```
    }  
}  
Drawing d = new MachineDrawing();
```

#### 4: 抽象类和抽象方法

在下列情况下, 一个类将成为抽象类:

- (1): 当一个类的一个或多个方法是抽象方法时;
- (2): 当类是一个抽象类的子类, 并且不能为任何抽象方法提供任何实现细节或方法主体时;
- (3): 当一个类实现一个接口, 并且不能为任何抽象方法提供实现细节或方法主体时;

**注意:**

- (1): 这里说的是这些情况下一个类将成为抽象类, 没有说抽象类一定会有这些情况。
- (2): 一个典型的错误: 抽象类一定包含抽象方法。但是反过来说“包含抽象方法的类一定是抽象类”就是正确的。
- (3): 事实上, 抽象类可以是一个完全正常实现的类

## 二: 接口的基本概念

---

接口可以说是 Java 程序设计中最重要概念之一了, “面向接口编程”是面向对象世界的共识, 所以深刻理解并熟练应用接口是每一个学习 Java 编程人员的重要任务

### 1: 接口概念

---

Java 可以创建一种称作接口(interface)的类, 在这个类中, 所有的成员方法都是抽象的, 也就是说它们都只有定义而没有具体实现, 接口是抽象方法和常量值的定义的集合。从本质上讲, 接口是一种特殊的抽象类, 这种抽象类中只包含常量和方法的定义, 而没有变量和方法的实现。定义接口的语法格式如下:

```
访问修饰符 修饰符 interface 接口名称 {  
    抽象属性集  
    抽象方法集  
}
```

示例程序: Runner.java

```
public interface Runner {  
    int ID = 1;  
    public void start();  
    public void run();  
    public void stop();  
}
```

目前看来接口和类差不多。确实如此, 接口本就是抽象类中演化而来的, 因而除特别规定, 接口享有和类同样的“待遇”。比如, 源程序中可以定义 0~多个类或接口, 但最多只能有一个 public 的类或接口, 如果有则源文件必须取和 public 的类和接口相同的名字。和类的继承格式一样, 接口之间也可以继承, 子接口可以继承父接口中的常量和抽象方法并添加新的抽象方法等。

但接口有其自身的一些特性:

- I 接口中声明的属性默认为，也只能是 `public static final` 的，因而在常量声明时可以省略这些修饰符；
- I 接口中只能定义抽象方法，这些方法默认为 `public abstract` 的、也只能是 `public abstract` 的，因而在声明方法时可以省略这些修饰符；
- I 和继承抽象父类类似，Java 类还可以“实现”接口。

Java 类在继承父类的同时可以同时实现多个接口，也继承了所有接口中的全部成分，仍然必须重写（实现）全部抽象方法，否则只能声明为抽象类。

类实现多个接口时，您需要做的仅是在关键字“implements”后用逗号分隔接口。总之，接口就是其它类需要实现的行为模板（以方法的形式表现）。

例：接口定义及实现

程序：Person.java

```
public class Person implements Runner {
    public void start() {
        //弯腰、蹬腿、咬牙、瞪眼、 开跑
    }
    public void run() {
        // 摆动手臂、 维持直线方向
    }
    public void stop() {
        // 减速直至停止、喝水。
    }
}
```

---

## 2: 为什么使用接口

两个类中的两个类似的功能，调用它们的类动态地决定一种实现，那它们提供一个抽象父类，子类分别实现父类所定义的方法。

问题的出现：Java 是一种单继承的语言，一般情况下，哪个具体类可能已经有了一个父类，解决是给它的父类加父类，或者给它父类的父类加父类，只到移动到类等级结构的最顶端。这样一来，对一个具体类的可插入性的设计，就变成了对整个等级结构中所有类的修改。

**接口是可插入性的保证。**

在一个等级结构中的任何一个类都可以实现一个接口，这个接口会影响到此类的所有子类，但不会影响到此类的任何父类。此类将不得不实现这个接口所规定的方法，而其子类可以从此类自动继承这些方法，当然也可以选择置换掉所有的这些方法，或者其中的某一些方法，这时候，这些子类具有了可插入性（并且可以用这个接口类型装载，传递实现了他的所有子类）。

我们关心的不是哪一个具体的类，而是这个类是否实现了我们需要的接口。

接口提供了关联以及方法调用上的可插入性，软件系统的规模越大，生命周期越长，接口使得软件系统的灵活性和可扩展性，可插入性方面得到保证。

---

## 3: 接口的基本作用



接口把方法的特征和方法的实现分割开来。这种分割体现在接口常常代表一个角色, 它包装与该角色相关的操作和属性, 而实现这个接口的类便是扮演这个角色的演员。一个角色由不同的演员来演, 而不同的演员之间除了扮演一个共同的角色之外, 并不要求其它的共同之处。

对于下述情况, 接口是有用的:

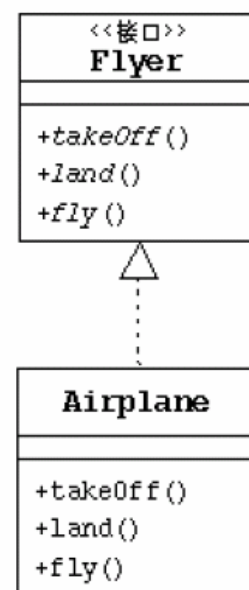
- (1) 声明方法, 期望一个或更多的类来实现该方法。
- (2) 揭示一个对象的编程接口, 而不揭示类的实际程序体。(当将类的一个包输送到其它开发程序中时它是非常有用的。)
- (3) 捕获无关类之间的相似性, 而不强迫类关系。
- (4) 可以作为参数被传递到在其它对象上调用的方法中

#### 4: 接口示例

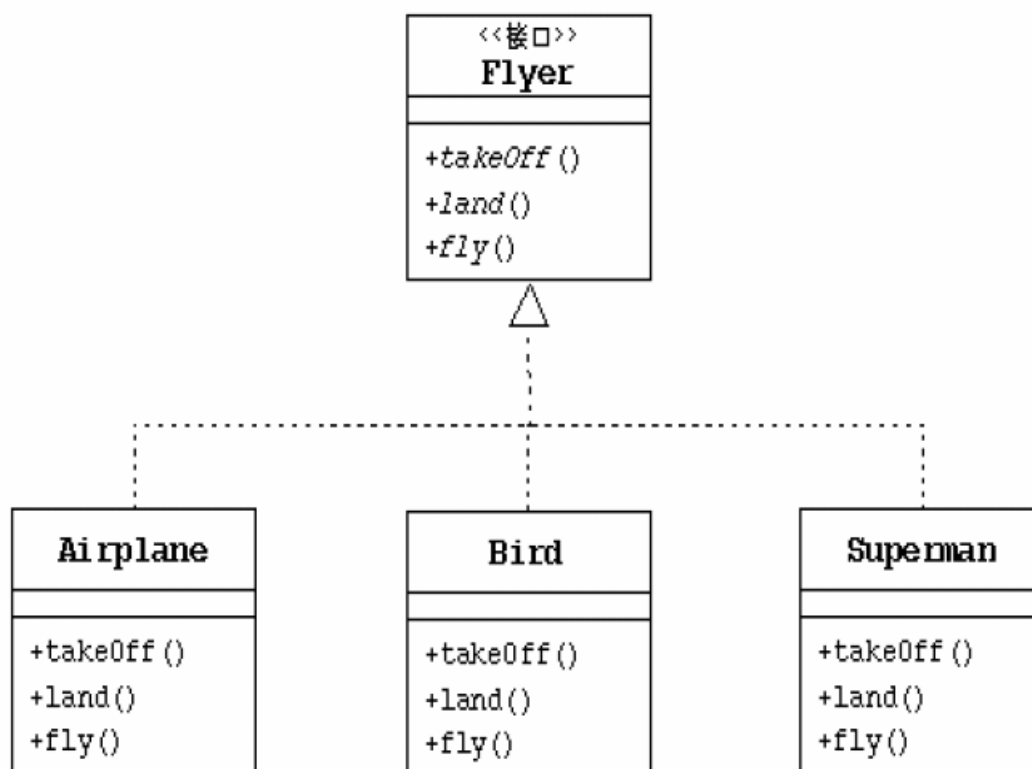
##### 1: 接口示例 1: 接口定义, 接口实现类

```
public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}

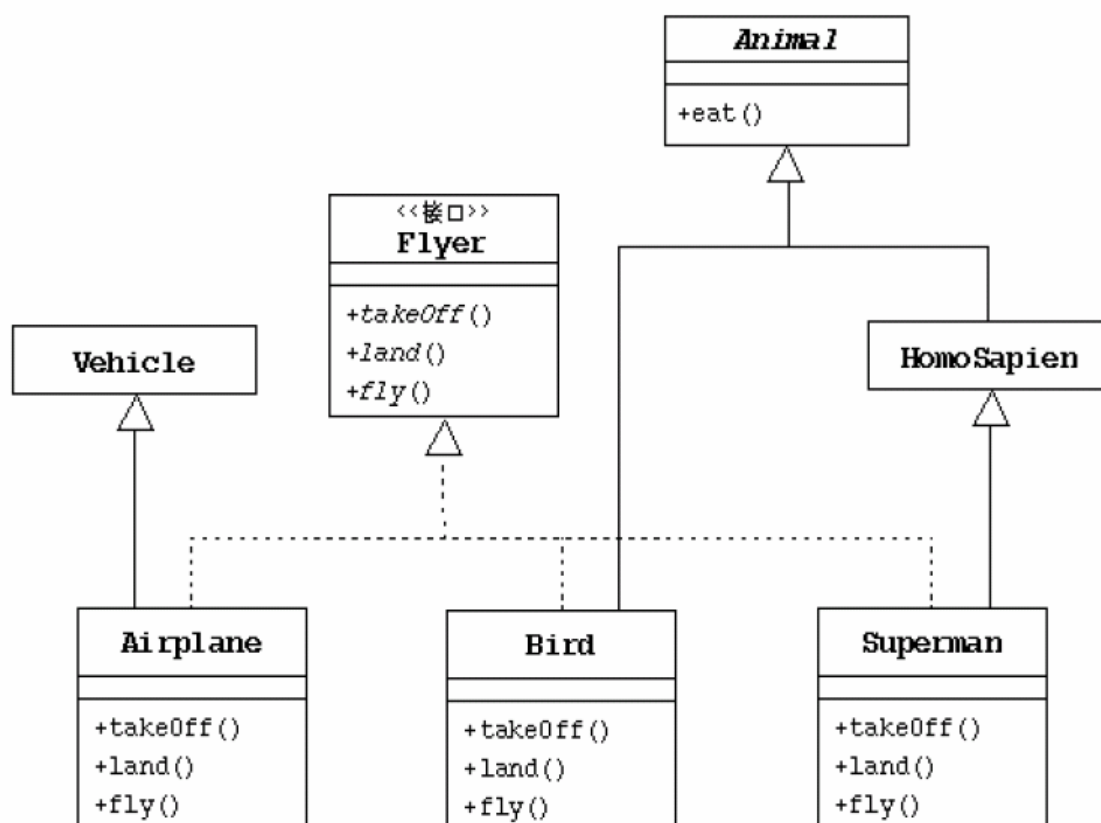
public class Airplane implements Flyer {
    public void takeOff() {
        // accelerate until lift-off
        // raise landing gear
    }
    public void land() {
        // lower landing gear
        // decelerate and lower flaps
    }
    public void takeOff() {
        // keep those engines running
    }
}
```



##### 2: 接口示例 2: 一个接口可以有多个不同的实现类



3: 接口示例 3: 一个类可以继承，也可以实现



```
public class Bird extends Animal implements Flyer {
    public void takeOff() { /* take-off implementation */ }
    public void land() { /* landing implementation */ }
    public void fly() { /* fly implementation */ }
    public void buildNest() { /* nest building behavior */ }
    public void layEggs() { /* egg laying behavior */ }
    public void eat() { /* override eating behavior */ }
}
```

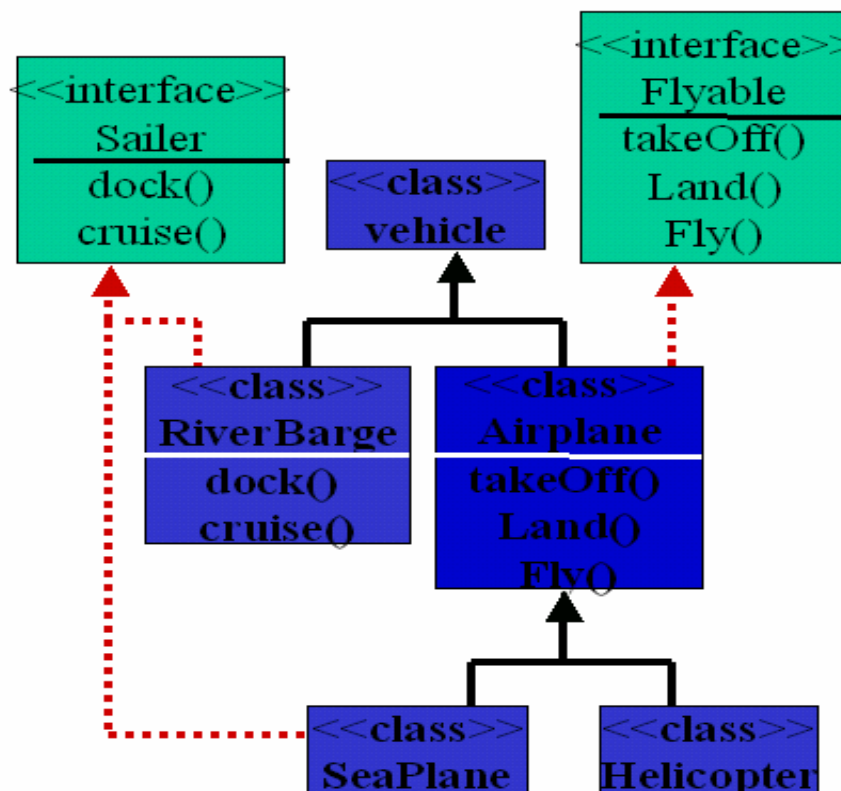
### 三：多重接口

#### 1: 多重接口的基本概念

一个类只可以继承一个父类（保持有兄弟关系，同属一族），但可以实现多个接口，而不需要在继承树中同属一个族。一个类实现多个接口就被称为多重接口。

类不可多重继承，因为两个父类的变量或方法可能重复。而接口不会继承变量，即便两个接口方法定义相同，接口中没有方法实现，类中也只有一个方法实现。

如下图所示：SeaPlane 实现了 Flyable 和 Sailer 两个接口，继承了 Airplane 类



接口的多重实现机制从一定程度上弥补了 Java 类单继承的局限，不但多个无关的类可以实现同一个接口，一个类可以同时实现多个无关的接口。与继承关系类似，接口与其实现类之间存在多态性。如果说继承父类代表着获得一种“家族身份”的话，实现接口则意味着取得了某种“资格”。在 java GUI（Graphical User Interface，图形用户界面）事件处理中，大量使用了接口，届时读者会对此有更深入的了解。

## 2: extends 和 implements

---

注意 extends 从句放在 implements 从句之前，这样一个类既具有自己定义的方法，继承父类方法，同时具有覆盖的接口方法。

```
class Bird extends Animal implements Flyable{
//自己定义
    public void buildNest() {}
    public void layEggs() {}
//继承父类
    public void eat() {}
//覆盖方法
    public void takeOff() { //加速起飞}
    public void land() { //减速着陆}
    public void fly() { //保持引擎运转}
}
```

## 3: 接口多继承

---

事实上，Java的接口是可以实现多继承的，类不允许多继承。如下示例：

```
public interface Test extends A,B {
    //定义
}
interface A{
    //定义
}
interface B{
    //定义
}
```

## 四：接口的基本思想

---

接口及相关机制的最基本作用在于：通过接口可以实现不相关类的相同行为，而不需考虑这些类之间的层次关系。根据接口可以了解对象的交互界面，而不需了解对象所属的类。

面向对象程序设计讲究“**提高内聚，降低耦合**”，那么不同的程序模块怎么相互访问呢，就是通过接口，也就是接口是各部分对外的统一外观。接口在 Java 程序设计中体现的思想就是隔离，因为接口只是描述一个统一的行为，所以开发人员在面向接口编程时并不关心具体的实现。

由以上讲到的接口的作用和基本思想可以看到，接口在面向对象的 Java 程序设计中占有举足轻重的地位。事实上在设计阶段最重要的任务之一就是设计出各部分的接口，然后通过接口的组合，形成程序的基本框架结构。

具体的接口的在设计和实现中的使用，本书将在第二部分 Java 程序设计和第三部分案例分析里面，结合具体的例子详细讲述，希望同学们好好体会。

## 五：接口作为类型使用

---

### 1: 接口的使用

---

接口的使用与类的使用有些不同。

在需要使用类的地方，会直接使用 `new` 关键字来构建一个类的实例进行应用：

```
ClassA a = new ClassA();
```

 这是正确的

但接口不可以这样用，因为接口不能直接使用 `new` 关键字来构建实例。

```
InterfaceA a = new InterfaceA();
```

 这是错误的

接口在使用的时候要实例化相应的实现类。

```
InterfaceA a = new ImplementsA();
```

 这是正确的

下面就可以使用 `a.method()` 的方式来调用接口的方法了。

### 2: 接口作为类型使用

---

接口作为引用类型来使用，任何实现该接口的类的实例都可以存储在该接口类型的变量中，通过这些变量可以访问类中所实现的接口中的方法，Java 运行时系统会动态地确定应该使用哪个类中的方法，实际上是调用相应的实现类的方法。

示例如下：

```
public class Test {
    public void test1(A a){
        a.doSth();
    }
    public static void main(String[] args) {
        Test t = new Test();

        A a = new B();
        t.test1(a);
    }
}

interface A{
    public int doSth();
}

class B implements A{
```

```
@Override
public int doSth() {
    System.out.println("now in B");
    return 123;
}
}
```

运行结果: now in B

大家看到接口可以作为一个类型来使用, 把接口作为方法的参数和返回类型。

## 五: 接口和抽象类的选择

---

由于从某种角度讲, 接口是一种特殊的抽象类, 它们的渊源颇深, 有很大的相似之处, 所以在选择使用谁的问题上很容易迷糊。

很多人有过这样的疑问: 为什么有的地方必须使用接口而不是抽象类, 而在另一些地方, 又必须使用抽象类而不是接口呢? 或者说, 在考虑 Java 类的一般化问题时, 很多人会在接口和抽象类之间犹豫不决, 甚至随便选择一种。

实际上接口和抽象类的选择不是随心所欲的。要理解接口和抽象类的选择原则, 有两个概念很重要: 对象的行为和对象的实现。如果一个实体可以有多种实现方式, 则在设计实体行为的描述方式时, 应当达到这样一个目标: 在使用实体的时候, 无需详细了解实体行为的实现方式。也就是说, 要把对象的行为和对象的实现分离开来。既然 Java 的接口和抽象类都可以定义不提供具体实现的方法, 在分离对象的行为和对象的实现时, 到底应该使用接口还是使用抽象类呢?

在接口和抽象类的选择上, 必须遵守这样一个原则: 行为模型应该总是通过接口而不是抽象类定义。所以通常是:

(1): 优先选用接口, 尽量少用抽象类。

选择抽象类的时候通常是如下情况:

(2): 需要定义子类的行为, 又要为子类提供共性的功能。

## 练习实践

---

### 程序 1:

---

接口与 Java 的“多继承”

需求：定义多个接口，并实现多接口继承

目标：

1. 接口的含义；
2. Java 中多继承的接口实现；

程序：

```
//: Adventure.java
package com.useful.java.part3;
import java.util.*;
interface CanFight {
    void fight();
}
interface CanSwim {
    void swim();
}
interface CanFly {
    void fly();
}
class ActionCharacter {
    public void fight() {}
}
class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}
public class Adventure {
    static void t(CanFight x) { x.fight(); }
    static void u(CanSwim x) { x.swim(); }
    static void v(CanFly x) { x.fly(); }
    static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero i = new Hero();
        t(i); // Treat it as a CanFight
        u(i); // Treat it as a CanSwim
        v(i); // Treat it as a CanFly
        w(i); // Treat it as an ActionCharacter
    }
}
```

说明：

- 1、Java 和 C++不同，不存在多继承，如果非在实现多继承，那只有通过接口，变向实现；
- 2、从中可以看到，Hero 将具体类 ActionCharacter 同接口 CanFight, CanSwim 以及 CanFly 合并起来。按这种形式合并一个具体类与接口的时候，具体类必须首先出现，然后才是接口（否则编译器会报错）。请注意 fight()的签名在 CanFight 接口与 ActionCharacter 类中是相同的，而且没有在 Hero 中为 fight()提供一个具体的定义。

## 作业

---

- 1: 定义一个对象“交通工具”，并定义接口，说明交通工具可以移动。继承交通工具而产生汽车、飞机、轮船，并定义类来实现其移动的方法。
- 2: 定义一个类来使用上面的接口



## 第八章 异常和断言

### 教学目标：

- 掌握异常的定义
- 掌握 try、catch 和 finally 语句
- 掌握 throw 和 throws 语句
- 理解异常的分类
- 掌握自定义异常
- 掌握断言的使用□
- 运行时屏蔽断言的使用

## 一: 异常的定义

---

### 1: 异常基础知识

---

什么是异常? 在 Java 编程语言中, 异常类定义程序中可能遇到的轻微的错误条件。可以写代码来处理异常并继续程序执行, 而不是让程序中断。

在程序执行中, 任何中断正常程序流程的异常条件就是错误或异常。例如, 发生下列情况时, 会出现异常:

- 想打开的文件不存在
- 网络连接中断
- 受控操作数超出预定范围
- 非常感兴趣的正在装载的类文件丢失

在 Java 编程语言中, 错误类定义被认为是不能恢复的严重错误条件。在大多数情况下, 当遇到这样的错误时, 建议让程序中断。

Java 编程语言实现异常处理来帮助建立弹性代码。在程序中发生错误时, 发现错误的方法能抛出一个异常到其调用程序, 发出已经发生问题的信号。然后, 调用方法捕获抛出的异常, 在可能时, 再恢复回来。这个方案给程序员一个写处理程序的选择, 来处理异常。

通过浏览 API, 可以决定方法抛出的是什么样的异常。

### 2: 异常实例

---

考虑一下 HelloWorld.java 程序版本的简单扩展, 它通过信息来循环:

```
1. public class HelloWorld {
2.     public static void main (String args[]) {
3.         int i = 0;
4.
5.         String greetings [] = {
6.             "Hello world!",
7.             "No, I mean it!",
8.             "HELLO WORLD!!"
9.         };
10.
11.        while (i < 4) {
12.            System.out.println (greetings[i]);
13.            i++;
14.        }
15.    }
16. }
```

正常情况下, 当异常被抛出时, 在其循环被执行四次之后, 程序终止, 并带有错误信息, 就象前面所示的程序那样。

- ```
1. c:> java HelloWorld
2. Hello world!
```

3. No, I mean it!
4. HELLO WORLD!!
5. java.lang.ArrayIndexOutOfBoundsException: 3
6. at HelloWorld.main(HelloWorld.java:12)

异常处理允许程序捕获异常, 处理它们, 然后继续程序执行。它是分层把关, 因此, 错误情况不会介入到程序的正常流程中。特殊情况发生时, 在与正常执行的代码分离的代码块中被处理。这就产生了更易识别和管理的代码。

## 二: 异常的处理

---

Java 提供了一种异常处理模型, 它使您能检查异常并进行相应的处理。它实现的是异常处理的抓抛模型。使用此模型, 您只需要注意有必要加以处理的异常情况。Java 提供的这种异常处理模型, 代替了用返回值或参数机制从方法返回异常码的手段。

异常处理的抓抛方法有两大优点:

(1): 异常情况能仅在有必要之处加以处理, 而不在其发生处和需要进行处理处之间的每一级上均进行处理;

(2): 能够编写统一的、可重用的异常处理代码。

应该区别对待程序中的正常控制流和异常处理流。当然, 异常处理流也是程序中的控制流。当异常发生时, 抛出一个异常。异常伴随调用链, 直到它们被捕获或程序退出为止。

下面是 Java 语言中的异常处理块的模型:

```
try{
    //放置可能出现异常的代码
}catch(Exception1 e1){
    //如果 try 块抛出异常对象的类型为 Exception1, 那么就在这里进行处理
}catch(Exception2 e2){
    //如果 try 块抛出异常对象的类型为 Exception2, 那么就在这里进行处理
}catch(ExceptionN eN){
    //如果 try 块抛出异常对象的类型为 ExceptionN, 那么就在这里进行处理
}finally{
    //不管是否有异常发生, 始终执行这个代码块
}
```

在未提供适当异常处理机制的程序中, 无论何时发生异常, 程序均会异常中断, 而之前分配的所有资源则保持其状态不变。这会导致资源遗漏。要避免这一情况, 在适当的异常处理机制中, 我们可以将以前由系统分配的所有资源返还给系统。所以, 当异常可能发生时, 要牢记必须对每一异常分别进行处理。

例如我们处理文件 I/O, 在打开文件时发生 IOException, 程序异常中断而没有机会关闭该文件, 这可能会毁坏文件而且分配给该文件的操作系统资源可能未返还给系统。

### 1: try 块

---

try 块由一组可执行语句组成, 在执行它们时可能会抛出异常。try 块后可随一个或多个 catch 块, 用来捕获在 try 中抛出的异常。另一方面, try 不可以跟随在 catch 块之后。

也就是说: 在 try 语句块中包含可能会产生异常的语句

```
int demo=0;

try{

    System.out.println(20/demo);

}
```

语句 System.out.println(20/demo);

会抛出一个异常, 原因是试图用 0 去除一个数。程序会被成功编译, 但当运行该程序时, 程序将会发生异常而中断, 异常可在 catch 块中被捕获, try 块可以嵌套:

```
try{

    statement 1;

    statement 2;

    try{

        statement 1;

        statement 2;

    }catch(Exception e){

        //异常处理

    }

}catch(Exception e){

    //异常处理

}
```

try 块嵌套时, 首先执行内部的 try 块, 该块中引发的任何异常在随后的 catch 中被捕获; 如果未发现与该内部块匹配的 catch 块, 则检查外部 try 块的 catch 块; 如果发现匹配的 catch 块, 那么在该块中处理这一异常, 否则 Java 运行时环境(JRE)处理这一异常。

## 2: catch 块

---

catch 块, 从其名称就可以看出, 是用来捕获并处理 try 中抛出的异常的代码块。没有 try 块, catch 块不能单独存在, 我们可有多个 catch 块, 以捕获不同类型的异常。下面是 catch 块的语法

```
try{

} catch(异常类型 e){

}
```

这里, e 是异常类型类的对象, 利用这一对象, 我们可以输出这一异常的详细信息

下面是 catch/try 块的一个简单示例:

```
class TryClass{

    public static void main(String args[]) {

        int dem0=0;

        try{

            System.out.println(20/dem0);

        }catch(ArithmeticException a){

            System.out.println(“Can not divided by zero”);

        }

    }

}
```

上述程序的输出结果是: Can not divided by zero

**注意:** 当多个 catch 块存在的时候, 从上往下 catch 异常的范围应该从小到大, 因为 catch 块的运行机制是找到一个匹配的就进行了处理, 如果把范围大的放在前面, 那么后面的代码就没有机会运行了, 这会是一个编译异常, 示例如下:

比如下面这个是正确的:

```
public class Test {

    public static void main(String[] args) {

        try{

            int i = 5/0;

        }catch(ArithmeticException e){

            e.printStackTrace();

        }catch(Exception err){

            err.printStackTrace();

        }

    }

}
```

而下面这个就是错误的了, 编译都发生了错误:

```
public class Test {

    public static void main(String[] args) {

        try{

            int i = 5/0;

        }catch(Exception err){

            err.printStackTrace();

        }catch(ArithmeticException e){

            e.printStackTrace\(\);

        }

    }

}
```

```
}  
  
}
```

### 3: finally 块

---

finally 块表示: 无论是否出现异常, 都会运行的块

通常在 finally 块中可以编写资源返还给系统的语句, 通常, 这些语句包括但不限于:

- (1): 释放动态分配的内存块;
- (2): 关闭文件;
- (3): 关闭数据库结果集;
- (4): 关闭与数据库建立的连接;

它紧跟着最后一个块, 是可选的, 不论是否抛出异常, finally 块总会被执行。

finally 块的语法如下:

```
try{  
  
}catch(异常类型 1 e){  
  
}catch(异常类型 2 e){  
  
}finally{  
  
}
```

下面的程序显示的是 finally 块的使用

```
public class Test {  
    static String name;  
    static int n01, n02;  
    public static void main(String args[]) {  
        try {  
            name = "Aptech Limited";  
            n01 = Integer.parseInt(args[0]);  
            n02 = Integer.parseInt(args[1]);  
            System.out.println(name);  
            System.out.println("Division is" + n01 / n02);  
        } catch(ArithmeticException i) {  
            System.out.println("Can not be divided by zero!");  
        } finally {  
            name = null;  
            System.out.println("finally executed");  
        }  
    }  
}
```

您从下面的命令行执行此程序：

```
Java Test 20 0
```

将会得到下面的输出：

```
Aptech Limited
Can not be divided by zero!
finally executed
```

现在从下面的命令行执行此程序：

```
Java Test 20 4
```

则会得到下面这样的输出：

```
Aptech Limited
Division is 5
finally executed
```

**说明：**当您用不同的命令行参数执行此程序时，均会看见“finally executed”的输出这意味着，无论try块是否抛出异常，都会执行finally块。

#### 4: try、catch、finally 块的关系

---

- (1): try 块不能单独存在，后面必须跟 catch 块或者 finally 块
- (2): 三者之间的组合为：try—catch 、try—catch—finally 、 try—finally 这几种是合法的
- (3): 一个 try 块可以有多个 catch 块，从上到下多个 catch 块的范围为从小到大

#### 5: throw 语句

---

throw 语句用来从代码中主动抛出异常。throw 的操作数是任一种异常类对象。下面是 throw 关键字的一个示例：

```
try {
    int i = 5/0;
} catch (ArithmeticException i) {
    throw new Exception("Can not be divided by zero!");
}
```

#### 6: throws 语句

---

**throws** 用来在方法定义时声明异常。

Java 中对异常的处理有两种方法，一个就是 try-catch，然后自己处理；一个就是不做处理，向外 throws，由别人去处理。

Java 语言要求在方法定义中列出该方法抛出的异常：

```
public Class Example
```

```
{  
  
    public static void exceptionExample() throws ExampleException, LookupException  
  
    {  
  
    }  
  
}
```

在上面的示例中, `exceptionExample()` 声明包括 `throws` 关键字, 其后列出了此方法可能抛出的异常列表。在此案例中列出的是 `ExampleException` 和 `LookupException`。

比如前面那个例子写完整如下:

```
public class Test {  
    public static void main(String args[]) throws Exception {  
        try {  
            int i = 5/0;  
        } catch (ArithmeticException i) {  
            throw new Exception("Can not be divided by zero!");  
        }  
    }  
}
```

---

## 7: 调用栈机制

如果方法中的一个语句抛出一个没有在相应的 `try/catch` 块中处理的异常, 那么这个异常就被抛出到调用方法中。如果异常也没有在调用方法中被处理, 它就被抛出到该方法的调用的程序。这个过程要一直延续到异常被处理。如果异常到这时还没被处理, 它便回到 `main()`, 而且, 即使 `main()` 不处理它, 那么, 该异常就异常地中断程序。

考虑这样一种情况, 在该情况中 `main()` 方法调用另一个方法 (比如, `first()`), 然后它调用另一个 (比如, `second()`)。如果在 `second()` 中发生异常, 那么必须做一个检查来看看该异常是否有一个 `catch`; 如果没有, 那么对调用栈 (`first()`) 中的下一个方法进行检查, 然后检查下一个 (`main()`)。如果这个异常在该调用栈上没有被最后一个方法处理, 那么就会发生一个运行时错误, 程序终止执行。

---

## 8: 处理或声明规则

为了写出健壮的代码, Java 编程语言要求, 当一个方法在栈 (即, 它已经被调用) 上发生 `Exception` (它与 `Error` 或 `RuntimeException` 不同) 时, 那么, 该方法必须决定如果出现问题该采取什么措施。

程序员可以做满足该要求的两件事:

- 第一, 通过将 `try {} catch () {}` 块纳入其代码中, 在这里捕获给被命名为属于某个父类的异常, 并调用方法处理它。即使 `catch` 块是空的, 这也算是处理情况。
- 第二, 让被调用的方法表示它将不处理异常, 而且该异常将被抛回到它所遇到的调用



方法中。按如下所示通过用 throws 子句标记的该调用方法的声明来实现的:

```
public void troublesome() throws IOException
```

关键字 throws 之后是所有异常的列表, 方法可以抛回到它的调用程序中。尽管这里只显示了一个异常, 如果有成倍的可能的异常可以通过该方法被抛出, 那么, 可以使用逗号分开的列表。

是选择处理还是选择声明一个异常取决于是否给你自己或你的调用程序一个更合适的候选的办法来处理异常。

## 三: 异常的分类

### 1: 异常的分类

在 Java 编程语言中, 异常有两种分类。java.lang.Throwable 类充当所有对象的父类, 可以使用异常处理机制将这些对象抛出并捕获。在 Throwable 类中定义方法来检索与异常相关的错误信息, 并打印显示异常发生的栈跟踪信息。它有 Error 和 Exception 两个基本子类。

错误 (Error): JVM 系统内部错误、资源耗尽等严重情况;

异常 (Exception 违例): 其它因编程错误或偶然的外在因素导致的一般性问题, 例如: 对负数开平方根、空指针访问、试图读取不存在的文件、网络连接中断等。

当发生 Error 时, 程序员根本无能为力, 只能让程序终止。比如说内存溢出, 不可能指望程序能处理这样的情况。而对于 Exception, 而有补救或控制的可能, 程序员也可以预先防范, 本章主要讨论 Exception 的处理。

为有效地描述异常状况、传递有关的异常信息, JDK 中针对各种普遍性的异常情况定义了多种异常类型。其层次关系如下图所示:

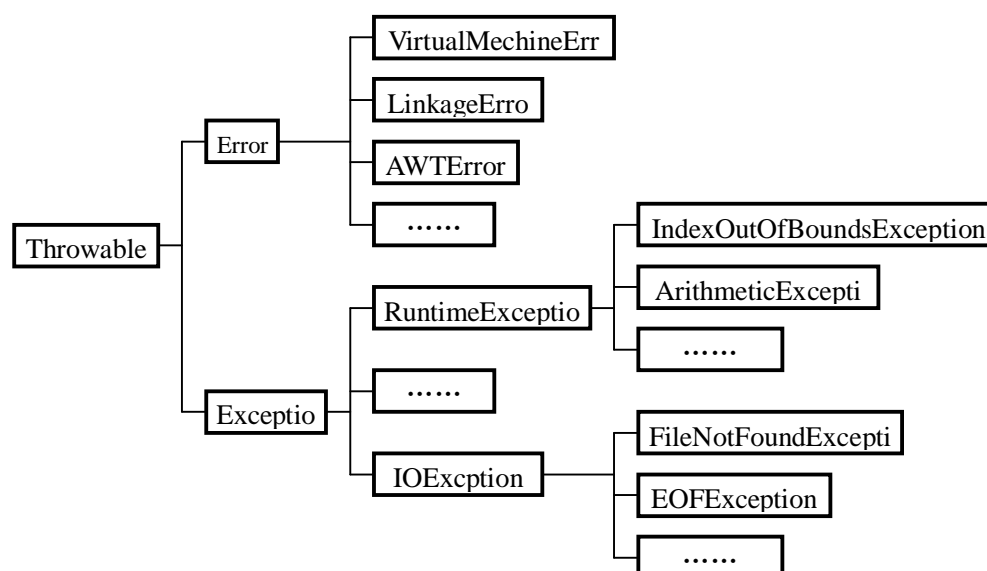


图 Java 异常类继承层次

其中，`RuntimeException`（运行时异常）是指因设计或实现方式不当导致的问题。也可以说，是程序员的原因导致的，本来可以避免发生的情况。比如，如果事先检查数组元素下标保证其不超出数组长度，那么，`ArrayIndexOutOfBoundsException` 异常从不会抛出；再如，先检查并确保一个引用类型变量值不为 `null`，然后在令其访问所需的属性和方法，那么，`NullPointerException` 也就从不会产生。

例题中的异常即属于 `RuntimeException`，出错的原因是数组 `friends` 中只含有三个元素，当 `for` 循环执行到第四次时，试图访问根本不存在的第四个数组元素 `friends[3]`，因此出错。

包括 `IOException` 在内的其它违例，则可以认为是描述运行时遇到的困难，它通常由环境而并非程序员的原因引起，可以进行处理。例如：文件未找到或无效 URL 异常都经常容易出现。

读者可能的疑问是：既然运行错误经常发生，是不是所有的 Java 程序也都应采取这种异常处理措施？答案是否定的，Java 程序异常处理的原则为：

- (1) 对于 `Error` 和 `RuntimeException`，可以在程序中进行捕获和处理，但不是必须的；
- (2) 对于 `IOException` 及其它异常，必须在程序中进行捕获和处理。

`ArrayIndexOutOfBoundsException` 属于 `RuntimeException`，一个正确设计和实现的程序不会出现这种异常，因此可根据实际情况选择是否进行捕获和处理。现实生活中也采取了类似的原则：比如火灾（一般属人为因素导致的），由于会导致严重后果，因此可以在生产车间配备灭火器、准备好灭火措施，处理可能发生的火灾。但是，一般家庭不会准备发电机去应付可能出现的停电事故。

程序运行时，`try` 语句块中的语句产生异常对象后，系统立即将之与后跟的 `catch` 语句块依次进行匹配性检查，只要异常对象是 `catch` 块中指定的类型或其子类类型，均认为匹配成功，一旦匹配成功则停止匹配检查、开始执行该 `catch` 块。

如果程序运行时，某个方法中的一个语句产生了一个没有在相应的 `try/catch` 块中处理的异常（肯定是 `Error` 或 `RuntimeException`），那么这个异常就被抛出到调用方法中。如果异常也没有在调用方法中被处理，它就被抛出到该方法的调用方法，直到异常被处理。如果异常对象一直被传递到 `main()` 方法仍未得到捕获处理，则程序将异常终止。

## 2: 预定义异常

---

Java 编程语言中预先定义好的异常叫预定义异常，下面是可能遇到的更具共同性的异常中的几种：

- `ArithmeticException`—整数被 0 除，运算得出的结果。

如：`int i =12 / 0;`

- `NullPointerException`—当对象没被实例化时，访问对象的属性或方法的尝试。

如：`Date d= null;`

`System.out.println(d.toString());`

- `NegativeArraySizeException`—创建带负维数大小的数组的尝试。
- `ArrayIndexOutOfBoundsException`—访问超过数组大小范围的一个元素的尝试。
- `SecurityException`—典型地被抛出到浏览器中，`SecurityManager` 类将抛出 applets 的一个异常，该异常企图做下述工作（除非明显地得到允许）：
  - 访问一个本地文件。
  - 打开主机的一个 socket，这个主机与服务于 applet 的主机不是同一个。
  - 在运行时环境中执行另一个程序。

## 四：自定义异常

---

Java 语言允许用户需要时创建自己的异常类型，用于表述 JDK 中未涉及到的其它异常状况，这些类型也必须继承 `Throwable` 类或其子类。用户自定义异常类通常属 `Exception` 范畴，依据命名惯例，应以 `Exception` 结尾。用户自定义异常未被加入 JRE 的控制逻辑中，因此永远不会自动抛出，只能由人工创建并抛出。

看一个用户自定义异常的例子：

程序：Test.java

```
class MyException extends Exception {
    private int idnumber;
    public MyException(String message, int id) {
        super(message);
        this.idnumber = id;
    }
    public int getId() {
        return idnumber;
    }
}

public class Test{
    public void regist(int num) throws MyException {
        if (num < 0) {
            throw new MyException("人数为负值，不合理", 3);
        }
        System.out.println("登记人数"+ num);
    }
    public void manager() {
        try {
            regist(-100);
        } catch (MyException e) {
            System.out.print("登记出错，类别："+e.getId());
        }
        System.out.print("本次登记操作结束");
    }
}
```

```
}  
public static void main(String args[]) {  
    Test t = new Test();  
    t.manager();  
}  
}
```

## 五：断言和断言的使用

---

### 1: 断言

---

断言用于证明和测试程序的假设, 比如 “这里的值大于 5”。

断言可以在运行时从代码中完全删除, 所以对代码的运行速度没有影响。

### 2: 断言的使用

---

断言有两种方法:

一种是 `assert<<布尔表达式>>`; 另一种是 `assert<<布尔表达式>>: <<细节描述>>`。

如果布尔表达式的值为 `false`, 将抛出 `AssertionError` 异常; 细节描述是 `AssertionError` 异常的描述文本

使用 `javac -source 1.4 MyClass.java` 的方式进行编译

示例如下:

```
public class AssertExample {  
    public static void main(String[] args) {  
        int x = 10;  
        if (args.length > 0) {  
            try {  
                x = Integer.parseInt(args[0]);  
            } catch (NumberFormatException nfe) {  
                /* Ignore */  
            }  
        }  
        System.out.println("Testing assertion that x == 10");  
        assert x == 10: "Our assertion failed";  
        System.out.println("Test passed");  
    }  
}
```

由于引入了一个新的关键字, 所以在编译的时候就需要增加额外的参数, 要编译成功, 必须使用 JDK1.4 的 `javac` 并加上参数 `'-source 1.4'`, 例如可以使用以下的命令编译上面的代码:

```
javac -source 1.4 AssertExample.java
```

以上程序运行使用断言功能也需要使用额外的参数（并且需要一个数字的命令行参数），例如：

```
java -ea AssertExample 1
```

程序的输出为：

```
Testing assertion that x == 10
Exception in thread "main" java.lang.AssertionError:
Our assertion failed
at AssertExample.main(AssertExample.java:20)
```

由于输入的参数不等于 10，因此断言功能使得程序运行时抛出断言错误，注意是错误，这意味着程序发生严重错误并且将强制退出。断言使用 boolean 值，如果其值不为 true 则抛出 AssertionError 并终止程序的运行。

### 3: 断言推荐使用方法

---

用于验证方法中的内部逻辑，包括：

- 内在不变式
- 控制流程不变式
- 后置条件和类不变式

注意：不推荐用于公有方法内的前置条件的检查

### 4: 运行时屏蔽断言

---

运行时要屏蔽断言，可以用如下方法：

```
java -disableassertions 或 java -da
```

运行时要允许断言，可以用如下方法：

```
java -enableassertions 或 java -ea
```

## 练习实践

---

本章的主要内容是异常处理，实践重点：

- I 异常处理的两种常用方式；
- I try...catch...finally 结构

### 程序 1

---

创建异常

需求：主动创建一个异常并抛出；

目标：

深入了解异常的工作机制。

程序：

```
//: ExceptionMethods.java
package com.useful.java.part2;

public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("Here's my Exception");
        } catch (Exception e) {
            System.out.println("Caught Exception");
            System.out.println(
                "e.getMessage(): " + e.getMessage());
            System.out.println(
                "e.toString(): " + e.toString());
            System.out.println("e.printStackTrace():");
            e.printStackTrace();
        }
    }
}
```

说明：

- 1、try 的时候，创建一个异常，没有处理，上抛了出去；
- 2、catch 时，抓住抛出的异常，并以多种情况显示出来；
- 3、运行情况如下图所示：

## 程序 2:

定义自己的异常类

需求：继承 Exception 类，定义自己的 Exception 类。

目标：

- 1、复习 Exception 类的使用；
- 2、了解继承的基本用法。

程序：

```
//: Inheriting.java
package com.useful.java.part3;

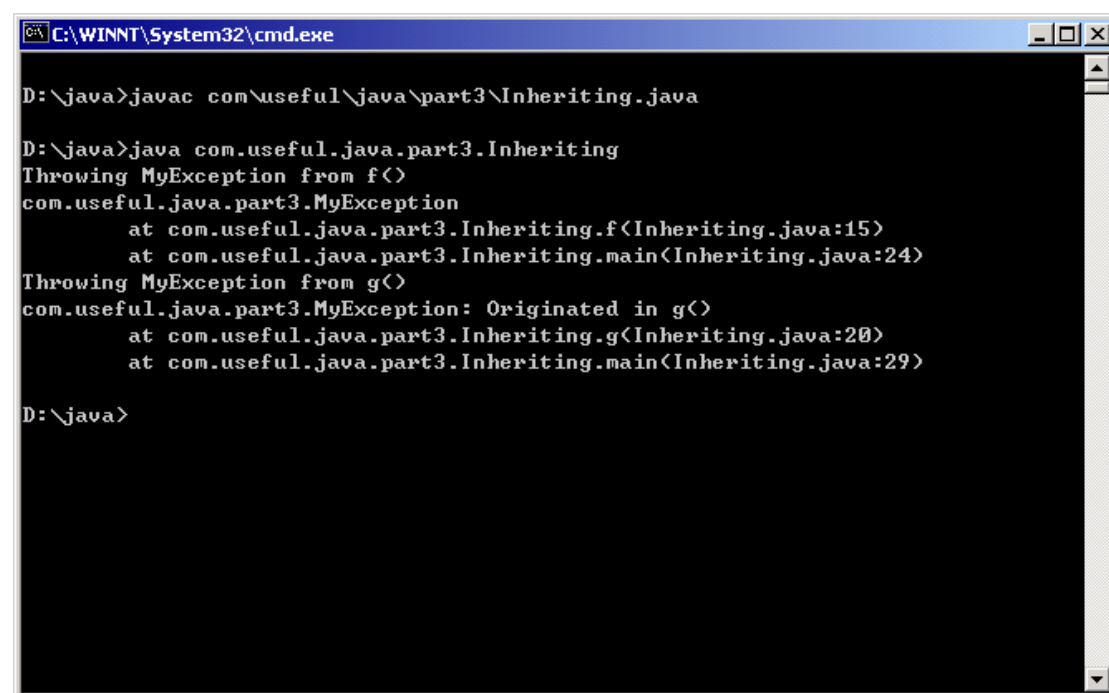
class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}

public class Inheriting {
    public static void f() throws MyException {
        System.out.println(
            "Throwing MyException from f()");
        throw new MyException();
    }
    public static void g() throws MyException {
        System.out.println(
```

```
        "Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException e) {
            e.printStackTrace();
        }
        try {
            g();
        } catch(MyException e) {
            e.printStackTrace();
        }
    }
}
```

说明:

- 1、根据异常一章及本章内容, 我们自定义了一个异常类;
- 2、程序运行情况如下:
- 3、需要注意的是 super 的用法, 另外, 还有一个 this。super 代表父类, this 代表本类。



```
C:\WINNT\System32\cmd.exe

D:\java>javac com\useful\java\part3\Inheriting.java

D:\java>java com.useful.java.part3.Inheriting
Throwing MyException from f()
com.useful.java.part3.MyException
    at com.useful.java.part3.Inheriting.f(Inheriting.java:15)
    at com.useful.java.part3.Inheriting.main(Inheriting.java:24)
Throwing MyException from g()
com.useful.java.part3.MyException: Originated in g()
    at com.useful.java.part3.Inheriting.g(Inheriting.java:20)
    at com.useful.java.part3.Inheriting.main(Inheriting.java:29)

D:\java>
```



## 作业

---

1: 用 `extends` 关键字创建自己的例外类;

2: 用 `main()` 创建一个类, 令其抛出 `try` 块内的 `Exception` 类的一个对象。为 `Exception` 的构造器赋予一个字串参数。在 `catch` 从句内捕获异常, 并打印出字串参数。添加一个 `finally` 从句, 并打印一条消息, 证明自己真正到达那里。

3: 给定下面代码:

```
public void example() {
    try {
        unsafe();
        System.out.println("Test 1");
    } catch (Exception e) {System.out.println("Test 2");}
    finally {System.out.println("Test 3");}
    System.out.println("Test 4");
}
```

如果方法 `unsafe()` 运行正常, 哪个结果不会被显示出来?

A. Test 1    B. Test 2    C. Test 3    D. Test 4

4: 编写应用程序, 从命令行传入两个整型数作为除数和被除数。要求程序中捕获 `NumberFormatException` 异常和 `ArithmeticException` 异常, 在命令行输入不同的参数时能输出如下各种结果:

```
java A
总是被执行
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException at
A.main(A.java:7)
```

```
java A 1 2
0
总是被执行
```

```
java A 1 3a
java.lang.NumberFormatException: 3a
at java.lang.Integer.parseInt(Integer.java:435)
at java.lang.Integer.parseInt(Integer.java:476)
at A.main(A.java:8)
总是被执行
```

```
java A 1 0
java.lang.ArithmeticException: / by zero at A.main(A.java:9)
总是被执行
```