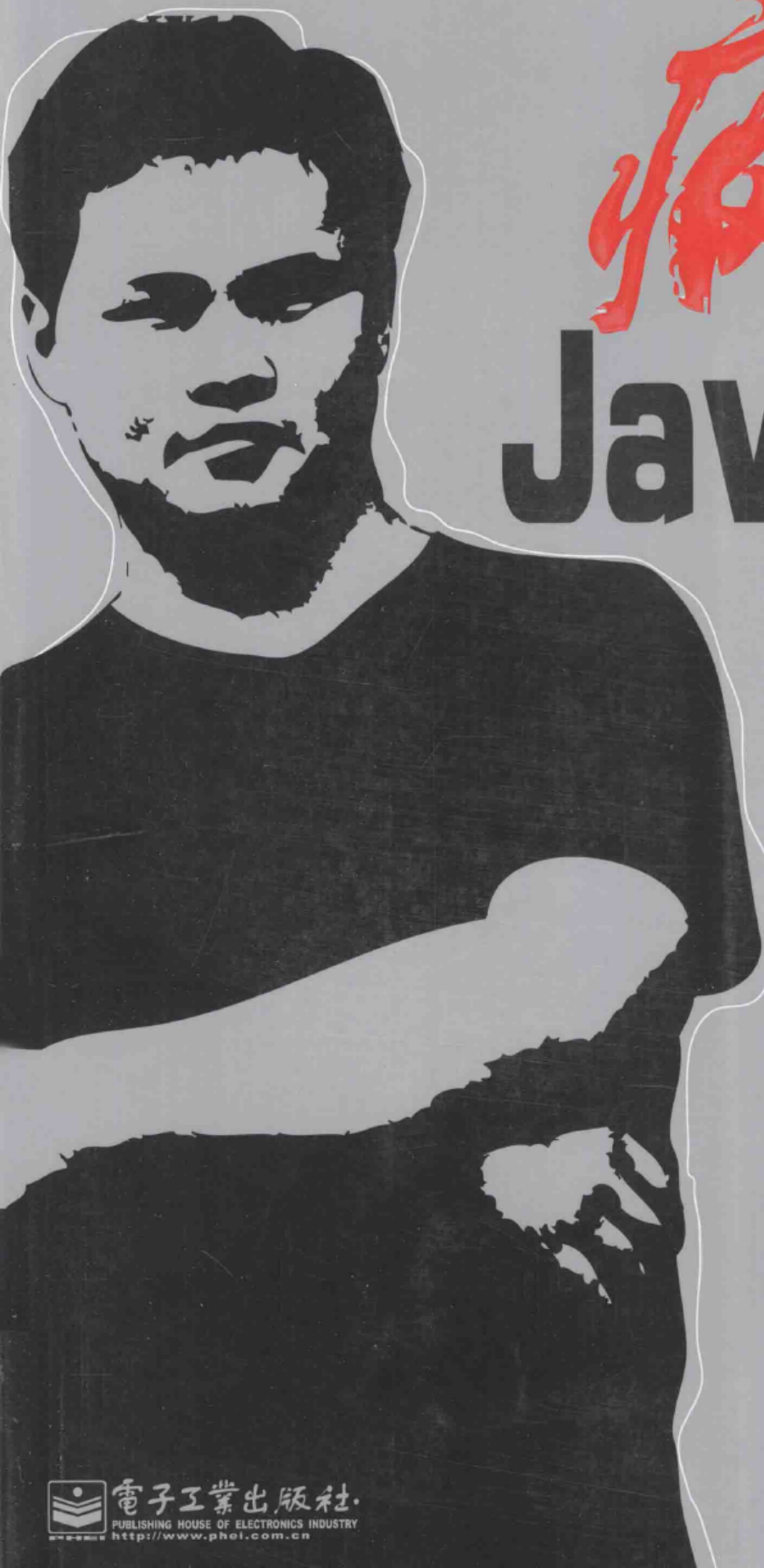


提供各种IT类书籍pdf下载，如有需要，请QQ:2404062482

注：链接至淘宝，不喜者勿入！整理那么多资料也不容易，请多多见谅！非诚勿扰！

[更多此类书籍](#)



疯狂

Java讲义

(第3版)

李刚 编著

疯狂源自梦想

技术成就辉煌

疯狂源自梦想
技术成就辉煌



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



● 疯狂源自梦想 技术成就辉煌

本书来自作者**8年**的Java培训经历，凝结了作者近**8000个小时**的授课经验，总结了**几千个Java学员**学习过程中的典型错误。

1. 案例驱动，引爆编程激情

本书不再是知识点的铺陈，而是致力于将知识点融入实际项目的开发中，所以书中涉及了大量Java案例：仿QQ的游戏大厅、MySQL企业管理器、仿EditPlus的文本编辑器、多线程、断点下载工具……希望读者通过编写这些程序找到编程的乐趣。

2. 再现李刚老师课堂氛围

本书的内容是笔者近8年授课经历的总结，知识体系取自李刚疯狂Java实战课程体系。

本书力求再现笔者的课堂氛围：以浅显比喻代替乏味的讲解，以疯狂实战代替空洞的理论。

3. 注释详细，轻松上手

为了降低读者阅读的难度，书中代码的注释非常详细，几乎每两行代码就有一行注释。不仅如此，本书甚至还把一些简单理论作为注释穿插到代码中，力求让读者能轻松上手。

阅读此书有任何技术问题，都可以登录如下站点获得解决：

疯狂Java联盟：<http://www.crazyit.org>



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview

上架建议：编程语言>Java

ISBN 978-7-121-23669-3



9 787121 236693 >

定价：109.00元（含光盘1张）



策划编辑：张月萍
责任编辑：葛娜
封面设计：李玲



疯狂 Java讲义

(第3版)

李刚 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书是《疯狂 Java 讲义》的第3版,第3版保持了前两版系统、全面、讲解浅显、细致的特性,全面新增介绍了 Java 8 的新特性,本书大部分示例程序都采用 Lambda 表达式、流式 API 进行了改写,因此务必使用 Java 8 的 JDK 来编译、运行。

本书深入介绍了 Java 编程的相关方面,全书内容覆盖了 Java 的基本语法结构、Java 的面向对象特征、Java 集合框架体系、Java 泛型、异常处理、Java GUI 编程、JDBC 数据库编程、Java 注释、Java 的 IO 流体系、Java 多线程编程、Java 网络通信编程和 Java 反射机制。覆盖了 java.lang、java.util、java.text、java.io 和 java.nio、java.sql、java.awt、javax.swing 包下绝大部分类和接口。本书全面介绍了 Java 8 的新的接口语法、Lambda 表达式、方法引用、构造器引用、函数式编程、流式编程、新的日期、时间 API、并行支持、改进的类型推断、重复注解、JDBC 4.2 新特性等新特性。

与前两版类似,本书并不单纯从知识角度来讲解 Java,而是从解决问题的角度来介绍 Java 语言,所以本书中涉及大量实用案例开发:五子棋游戏、梭哈游戏、仿 QQ 的游戏大厅、MySQL 企业管理器、仿 EditPlus 的文本编辑器、多线程、断点下载工具、Spring 框架的 IoC 容器……这些案例既能让读者巩固每章的知识,又可以让读者学以致用,激发编程自豪感,进而引爆内心的编程激情。本书光盘里包含书中所有示例的代码和《疯狂 Java 实战演义》的所有项目代码,这些项目可以作为本书课后练习的“非标准答案”,如果读者需要获取关于课后习题的解决方法、编程思路,可以登录 <http://www.crazyit.org> 站点与笔者及本书庞大的读者群相互交流。

本书为所有打算深入掌握 Java 编程的读者而编写,适合各种层次的 Java 学习者和工作者阅读,也适合作为大学教育、培训机构的 Java 教材。但如果只是想简单涉猎 Java,则本书过于庞大,不适合阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

疯狂 Java 讲义 / 李刚编著. —3 版. —北京:电子工业出版社,2014.7

ISBN 978-7-121-23669-3

I. ①疯… II. ①李… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2014)第 141615 号

策划编辑:张月萍

责任编辑:葛娜

印刷:北京京科印刷有限公司

装订:三河市皇庄路通装订厂

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开本:850×1168 1/16 印张:54.25 字数:1757 千字 彩插:4

版次:2008 年 6 月第 1 版

2014 年 7 月第 3 版

印次:2014 年 7 月第 1 次印刷

印数:5000 册 定价:109.00 元(含光盘 1 张)

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。

疯狂Java习题效果图

Socket网络斗地主
游戏房间



Socket网络斗地主出牌



梭哈跟注



梭哈胜负判定



注：彩插图所展示的项目均为李刚老师疯狂Java训练营的学员所作，程序代码下载见前言中说明。

仿MySQL Front工具的工作界面

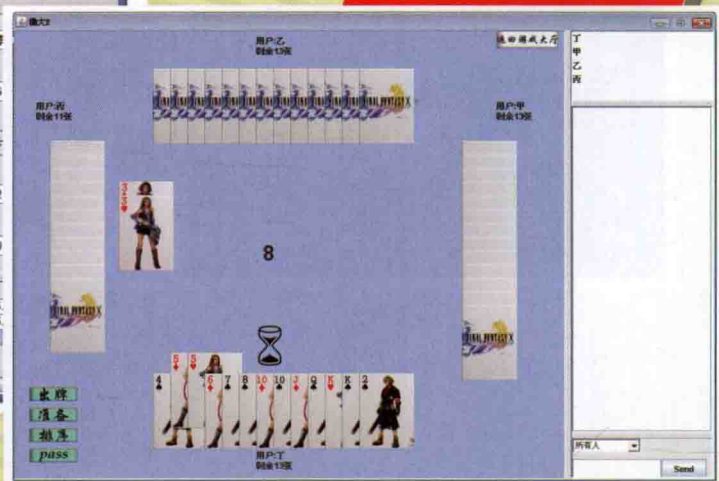
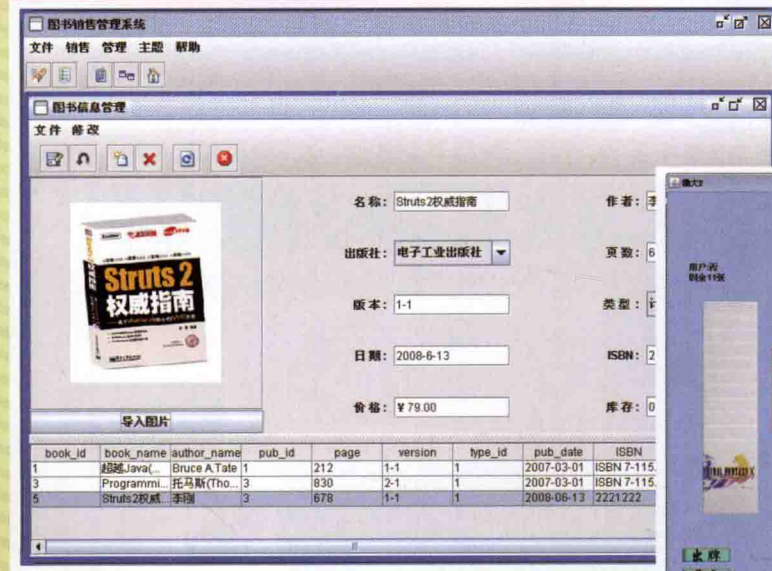


C-S图书销售管理系统
图书管理界面

Socket网络锄大地的大厅界面

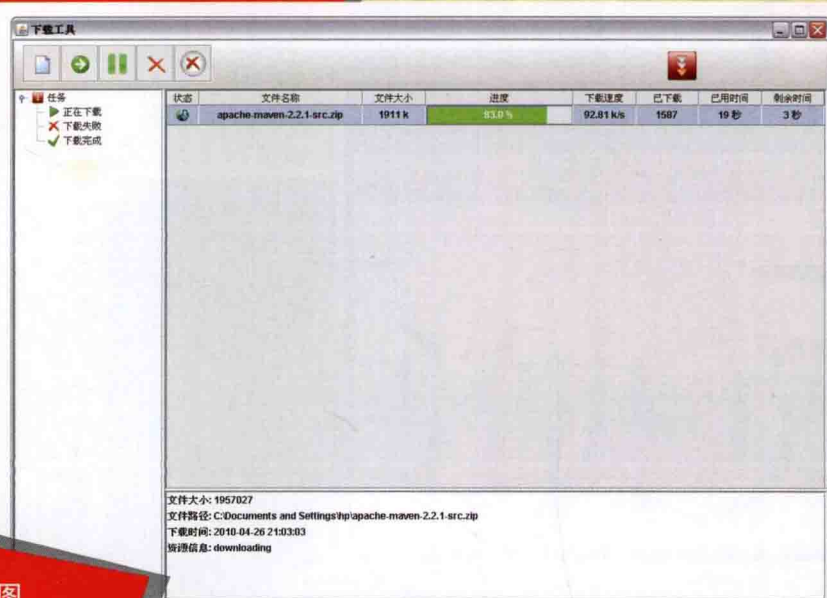


Socket网络锄大地的出牌界面



疯狂Java习题效果图

多线程断点下载工具



截图工具截图



截图工具浏览截图



疯狂Java习题效果图

仿QQ游戏大厅的Socket
网络围棋大厅界面



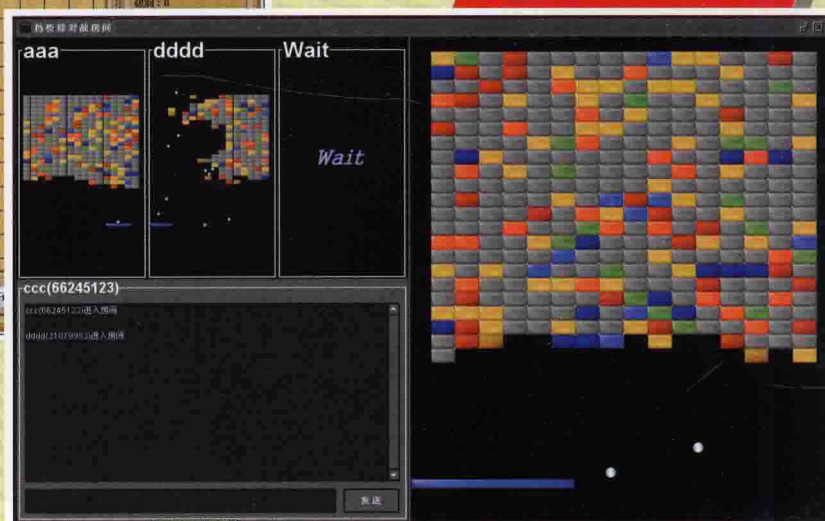
Socket网络弹球游戏的大厅界面



仿QQ游戏大厅的Socket
网络围棋对弈界面

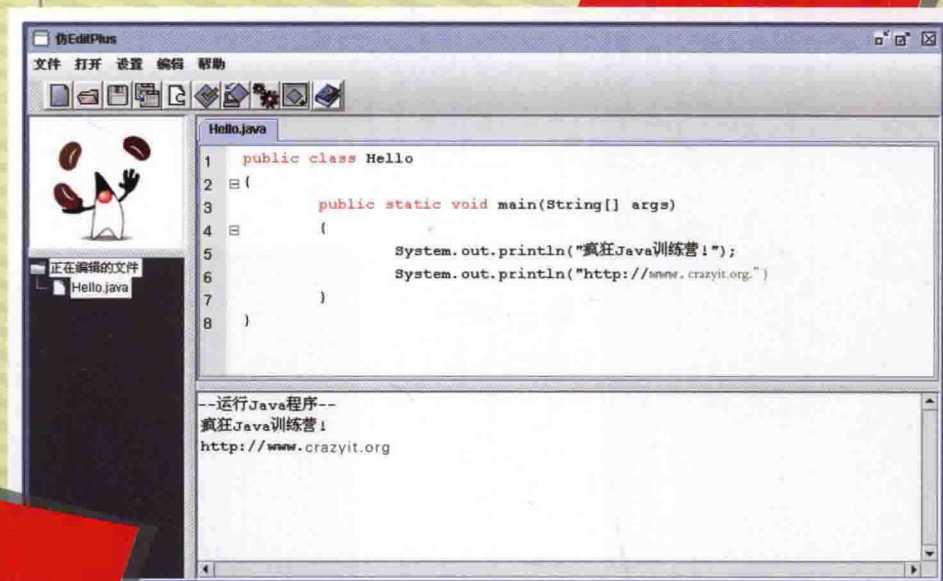


Socket网络弹球游戏的
对战界面

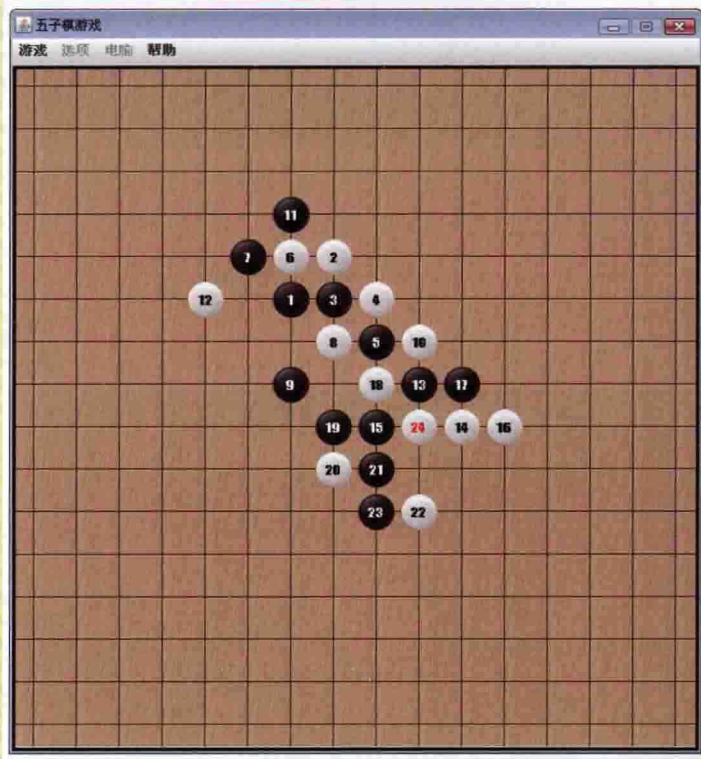


疯狂Java习题效果图

仿EditPlus编辑器



单机五子棋



单机俄罗斯方块



八年沉淀，王者归来

疯狂软件教育专注高级软件编程，以“十年磨一剑”的心态打造全国最强疯狂 Java 学习体系，包括疯狂 Java 体系原创图书，疯狂 Java 学习路线图，这些深厚的知识沉淀已被大量高校、培训机构奉为经典。

经过八年的技术沉淀，疯狂 Java 创始人李刚携疯狂 Java 精英讲师团队潜心打造全国技术沉淀最厚的高端软件教育。疯狂软件教育怀抱“科技强国”的梦想，立志以务实的技术推动中国的软件教育，为推动中国软件行业的发展贡献自己的绵薄之力。

疯狂软件教育网址：www.fkjava.org

优惠券

为感谢广大 Java 学习者对疯狂 Java 体系图书的认同，博文视点与疯狂软件教育共同制作价值 200 元的优惠券。所有读者凭此优惠券可抵疯狂软件教育 200 元学费。

每个学员最多使用一张优惠券，请在报名缴费前出示该优惠券，本优惠券不能兑换现金。博文视点与疯狂软件教育保留对本优惠券的最终解释权。



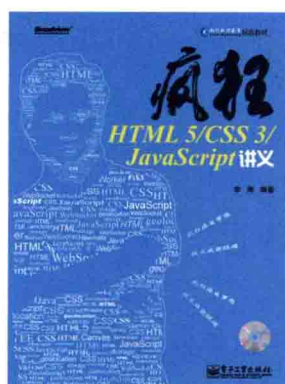
疯狂Java体系

疯狂源自梦想 技术成就辉煌



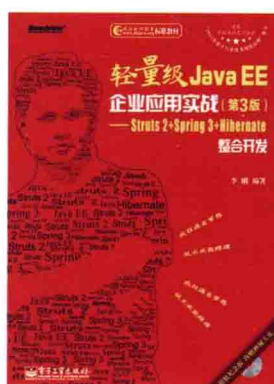
疯狂Java程序员的基本修养

作者: 李刚
定价: 59.00元
出版时间: 2013-01
书号: 978-7-121-19232-6



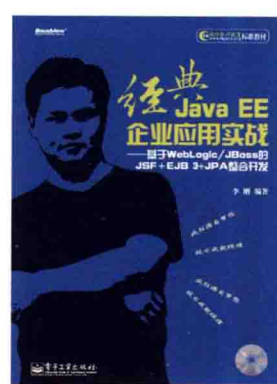
疯狂HTML 5/CSS 3/
JavaScript 讲义

作者: 李刚
定价: 69.00元 (含光盘1张)
出版时间: 2012-05
书号: 978-7-121-16863-5



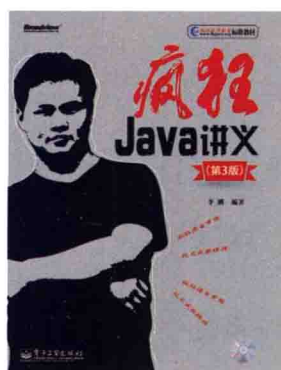
轻量级Java EE企业应用实战
(第3版)——Struts 2
+Spring 3+Hibernate整合开发

作者: 李刚
定价: 99.00元 (含光盘1张)
出版时间: 2012-04
书号: 978-7-121-16085-1



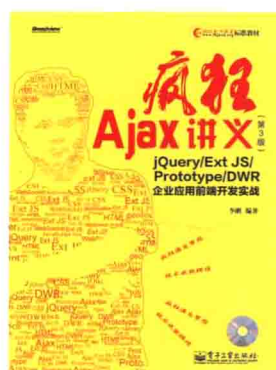
经典Java EE企业应用实战
——基于WebLogic/JBoss的
JSF+EJB 3+JPA整合开发

作者: 李刚
定价: 79.00元 (含光盘1张)
出版时间: 2010-08
书号: 978-7-121-11534-9



疯狂Java讲义 (第3版)

作者: 李刚
定价: 109.00元 (含光盘1张)
出版时间: 2014-07
书号: 978-7-121-23669-3



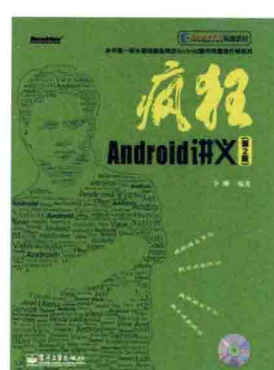
疯狂Ajax讲义 (第3版)
——jQuery/Ext JS/Prototype/
DWR企业应用前端开发实战

作者: 李刚
定价: 79.00元 (含光盘1张)
出版时间: 2013-01
书号: 978-7-121-19394-1



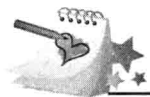
疯狂XML讲义 (第2版)

作者: 李刚
定价: 69.00元 (含光盘1张)
出版时间: 2011-08
书号: 978-7-121-14049-5



疯狂Android讲义
(第2版)

作者: 李刚
定价: 99.00元 (含光盘1张)
出版时间: 2013-03
书号: 978-7-121-19485-6



如何学习 Java

——谨以此文献给打算以编程为职业、并愿意为之疯狂的人

经常看到有些学生、求职者捧着一本类似 JBuilder 入门、Eclipse 指南之类的图书学习 Java，当他们学会了在这些工具中拖出窗体、安装按钮之后，就觉得自己掌握、甚至精通了 Java；又或是找来一本类似 JSP 动态网站编程之类的图书，学会使用 JSP 脚本编写一些页面后，就自我感觉掌握了 Java 开发。

还有一些学生、求职者听说 J2EE、Spring 或 EJB 很有前途，于是立即跑到书店或图书馆找来一本相关图书。希望立即学会它们，然后进入软件开发业、大显身手。

还有一些学生、求职者非常希望找到一本既速成、又大而全的图书，比如突击 J2EE 开发、一本书精通 J2EE 之类的图书（包括笔者曾出版的《轻量级 J2EE 企业应用实战》一书，据说销量不错），希望这样一本图书就可以打通自己的“任督二脉”，一跃成为 J2EE 开发高手。

也有些学生、求职者非常喜欢 J2EE 项目实战、项目大全之类的图书，他们的想法很单纯：我按照书上介绍，按图索骥、依葫芦画瓢，应该很快就可学会 J2EE，很快就能成为一个受人羡慕的 J2EE 程序员了。

.....

凡此种种，不一而足。但最后的结果往往是失败，因为这种学习没有积累、没有根基，学习过程中困难重重，每天都被一些相同、类似的问题所困扰，起初热情十足，经常上论坛询问，按别人的说法解决问题之后很高兴，既不知道为什么错？也不知道为什么对？只是盲目地抄袭别人的说法。最后的结果有两种：

① 久而久之，热情丧失，最后放弃学习。

② 大部分常见问题都问遍了，最后也可以从事一些重复性开发，但一旦遇到新问题，又将束手无策。

第二种情形在普通程序员中占了极大的比例，笔者多次听到、看到（在网络上）有些程序员抱怨：我做了 2 年多 Java 程序员了，工资还是 3000 多点。偶尔笔者会与他们聊聊工作相关内容，他们会告诉笔者：我也用 Spring 了啊，我也用 EJB 了啊……他们感到非常不平衡，为什么我的工资这么低？其实笔者很想告诉他们：你们太浮躁了！你们确实是用了 Spring、Hibernate 又或是 EJB，但你们未想过为什么要用这些技术？用这些技术有什么好处？如果不用这些技术行不行？

很多时候，我们的程序员把 Java 当成一种脚本，而不是一门面向对象的语言。他们习惯了在 JSP 脚本中使用 Java，但从不去想 JSP 如何运行，Web 服务器里的网络通信、多线程机制，为何一个 JSP 页面能同时向多个请求者提供服务？更不会想如何开发 Web 服务器；他们像代码机器一样编写 Spring Bean 代码，但从不去理解 Spring 容器的作用，更不会想如何开发 Spring 容器。

有时候，笔者的学生在编写五子棋、梭哈等作业感到困难时，会向他们的大学师兄、朋友求救，这些程序员告诉他：不用写了，网上有下载的！听到这样回答，笔者不禁感到哑然：网上还有 Windows 下载呢！网上下载和自己编写是两码事。偶尔，笔者会怀念以前黑色屏幕、绿荧荧字符时代，那时候程序员很单纯：当我们想偷懒时，习惯思维是写一个小工具；现在程序员很聪明：当他们想偷懒时，习惯思维是从网上下一个小工具。但是，谁更幸福？

当笔者的学生把他们完成的小作业放上互联网之后，然后就有许多人称他们为“高手”！这个称呼却让他们万分惭愧；惭愧之余，他们也感到万分欣喜，非常有成就感，这就是编程的快乐。编程的过程，与寻宝的过程完全一样：历经辛苦，终于找到心中的梦想，这是何等的快乐？

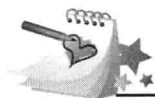
如果真的打算将编程当成职业，那就不应该如此浮躁，而是应该扎扎实实先学好 Java 语言，然后按 Java 本身的学习规律，踏踏实实一步一个脚印地学习，把基本功练扎实了才可获得更大的成功。

实际情况是，有多少程序员真正掌握了 Java 的面向对象？真正掌握了 Java 的多线程、网络通信、反射等内容？有多少 Java 程序员真正理解了类初始化时内存运行过程？又有多少程序员理解 Java 对象从创建到消失的全部细节？有几个程序员真正独立地编写过五子棋、梭哈、桌面弹球这种小游戏？又有几个 Java 程序员敢说：我可以开发 Struts？我可以开发 Spring？我可以开发 Tomcat？很多人又会说：这些都是许多人开发出来的！实际情况是：许多开源框架的核心最初完全是由一个人开发的。现在这些优秀程序已经出来了！你，是否深入研究过它们，是否深入掌握了它们？

如果要真正掌握 Java，包括后期的 Java EE 相关技术（例如 Struts、Spring、Hibernate 和 EJB 等），一定要记住笔者的话：绝不要从 IDE（如 JBuilder、Eclipse 和 NetBeans）工具开始学习！IDE 工具的功能很强大，初学者学起来也很容易上手，但也非常危险：因为 IDE 工具已经为我们做了许多事情，而软件开发要全部了解软件开发的全部步骤。



2011 年 12 月 17 日



光盘说明

一、光盘内容



本光盘是《疯狂 Java 讲义（第 3 版）》一书的配书光盘，书中的代码按章、按节存放，即第 2 章、第 3 节所使用的代码放在 codes 文件夹的 02\2.2 文件夹下，依此类推。

另：书中每份源代码也给出与光盘源文件的对应关系，方便读者查找。



本光盘 codes 目录下有 17 个文件夹，其内容和含义说明如下：

(1) 01~18 文件夹名对应于《疯狂 Java 讲义（第 3 版）》中的章名，即第 3 章所使用的代码放在 codes 文件夹的 03 文件夹下，依此类推。

(2) 本书所有代码都是 IDE 工具无关的程序，读者既可以在命令行窗口直接编译、运行这些代码，也可以导入 Eclipse、NetBeans 等 IDE 工具来运行它们。

(3) 本书第 12 章第 11 节的 TableModelTest.java 程序，以及第 13 章的绝大部分程序都需要连接数据库，所以读者需要先导入 *.sql 文件中的数据库脚本，并修改 mysql.ini 文件中的数据库连接信息。连接数据库时所用的驱动程序 JAR 文件为 mysql-connector-java-5.1.30-bin.jar 文件。这些需要连接数据库的程序里还提供了一个 *.cmd 文件，该文件是一个批处理文件，运行该文件可以运行相应的 Java 程序，例如 DatabaseMetaDataTest.java 对应的 *.cmd 文件为 runDatabaseMetaDataTest.cmd。



本光盘根目录下提供了一个“Java 设计模式（疯狂 Java 联盟版）.chm”文件，这是一份关于设计模式的电子教材，由疯狂 Java 联盟的杨恩雄亲自编写、制作，他同意广大读者阅读、传播这份开源文档。



本光盘根目录下包含一个“project_codes”文件夹，该文件夹里包含了疯狂 Java 联盟的杨恩雄编写的《疯狂 Java 实战演义》一书的光盘内容，该光盘中包含了大量实战性很强的项目，这些项目基本覆盖了《疯狂 Java 讲义（第 3 版）》课后习题的要求，读者可以参考相关案例来完成《疯狂 Java 讲义（第 3 版）》的课后习题。



本光盘根目录下包含一个“课件”文件夹，该文件夹里包含了《疯狂 Java 讲义（第 3 版）》各章配套的授课 PPT 教案，各高校教师、学生可在此基础上自由修改、传播，但请保留署名。



本光盘根目录下包含一个“视频”文件夹，里面包含了大约 14 个小时的基础授课视频。

二、运行环境



本书中的程序在以下环境调试通过：

(1) 安装 jdk-8u5-windows-x64.exe，安装完成后，添加 CLASSPATH 环境变量，该环境变量的值为：;%JAVA_HOME%/lib/tools.jar;%JAVA_HOME%/lib/dt.jar。如果为了可以编译和运行 Java 程序，还应该在 PATH 环境变量中增加 %JAVA_HOME%/bin。其中 JAVA_HOME 代表 JDK（不是 JRE）的安装路径。安装上面工具的详细步骤，请参考本书的第 1 章。

(2) 安装 MySQL 5.5 或更高版本，按第 13 章所介绍的方式安装。

三、注意事项

(1) 书中有大量代码需要连接数据库，读者应修改数据库 URL 以及用户名、密码，让这些代码与读者的运行环境一致。如果项目下有 SQL 脚本，则导入 SQL 脚本即可；如果没有 SQL 脚本，系统将在运行时自动建表，读者只需创建对应的数据库即可。

(2) 在使用本光盘中的程序时，请将程序拷贝到硬盘上，并去除文件的只读属性。

四、技术支持

如果您在使用本光盘中遇到不懂的技术问题，则可以登录如下网站与作者联系：

<http://www.crazyit.org>

我在 Java 编程教学中把《疯狂 Java 讲义》列为重要的中文参考资料。它覆盖了“够用”的 Java 语言和技术，作者有实际的编程和教学经验，也尽力把相关问题讲解明白、分析清楚，这在同类书籍中是比较难得的。



前言

2014 年 3 月 18 日，Oracle 发布了 Java 8 正式版。Java 8 是自 Java 5 以来最重要的版本更新，Java 8 引入了大量新特性——重新设计的接口语法、Lambda 表达式、方法引用、构造器引用、函数式编程、流式编程、新的日期、时间 API 等，这些新特性进一步增强了 Java 语言的功能。

为了向广大工作者、学习者介绍最新、最前沿的 Java 知识，在 Java 8 正式发布之前，笔者已经深入研究过 Java 8 绝大部分可能新增的功能；当 Java 8 正式发布之后，笔者在第一时间开始了《疯狂 Java 讲义（第 2 版）》的升级：使用 Java 8 改写了全书所有程序，全面介绍了 Java 8 的各种新特性。

在以“疯狂 Java 体系”图书为教材的疯狂软件教育中心（www.fkjava.org），经常有学生询问：为什么叫疯狂 Java 这个名字？也有一些读者通过网络、邮件来询问这个问题。其实这个问题的答案可以在本书第 1 版的前言中找到。疯狂的本质是一种“享受编程”的状态。在一些不了解编程的人看来：编程的人总面对着电脑，在键盘上敲打，这种生活实在太枯燥了，但实际上是因为他们并未真正了解编程，并未真正走进编程。在外人眼中：程序员不过是在敲打键盘；但在程序员心中：程序员敲出的每个字符，都是程序的一部分。

程序是什么呢？程序是对现实世界的数字化模拟。开发一个程序，实际是创建一个或大或小的“模拟世界”。在这个过程中，程序员享受着“创造”的乐趣，程序员沉醉在他所创造的“模拟世界”里：疯狂地设计、疯狂地编码实现。实现过程不断地遇到问题，然后解决它；不断地发现程序的缺陷，然后重新设计、修复它——这个过程本身就是一种享受。一旦完全沉浸到编程世界里，程序员是“物我两忘”的，眼中看到的、心中想到的，只有他正在创造的“模拟世界”。

在学会享受编程之前，编程学习者都应该采用“案例驱动”的方式，学习者需要明白程序的作用是：解决问题——如果你的程序不能解决你自己的问题，如何期望你的程序去解决别人的问题呢？那你的程序的价值何在？——知道一个知识点能解决什么问题，才去学这个知识点，而不是盲目学习！因此本书强调编程实战，强调以项目激发编程兴趣。

仅仅只是看完这本书，你不会成为高手！在编程领域里，没有所谓的“武林秘笈”，再好的书一定要配合大量练习，否则书里的知识依然属于作者，而读者则仿佛身入宝山而一无所获的笨汉。本书配合了大量高强度的练习，希望读者强迫自己去完成这些项目。这些习题的答案可以参考本书所附光盘中《疯

狂 Java 实战演义》的配套代码。如果需要获得编程思路和交流，可以登录 <http://www.crazyit.org> 与广大读者和笔者交流。

本书前两版面市的近 6 年时间里，无数读者已经通过本书步入了 Java 编程世界，而且第 2 版的年销量比第 1 版的年销量大幅提升，这说明“青山遮不住”，优秀的作品，经过时间的沉淀，往往历久弥新。

广大读者对疯狂 Java 的肯定，读者认同、赞誉既让笔者十分欣慰，也鞭策笔者以更高的热情、更严谨的方式创作图书。时至今日，每次笔者创作或升级图书时，总有一种诚惶诚恐、如履薄冰的感觉，惟恐辜负广大读者的厚爱。

笔者非常欢迎所有热爱编程、愿意推动中国软件业的学习者、工作者对本书提出宝贵的意见，非常乐意与大家交流。中国软件业还处于发展阶段，所有热爱编程、愿意推动中国软件业的人应该联合起来，共同为中国软件行业贡献自己的绵薄之力。

本书有什么特点



本书并不是一本简单的 Java 入门教材，也不是一门“闭门造车”式的 Java 读物。本书来自于笔者 8 年多的 Java 培训经历，凝结了笔者近 8000 个小时的授课经验，总结了几千个 Java 学员学习过程中的典型错误。

因此，本书具有如下三个特点：

1. 案例驱动，引爆编程激情

本书不再是知识点的铺陈，而是致力于将知识点融入实际项目的开发中，所以本书中涉及了大量 Java 案例：仿 QQ 的游戏大厅、MySQL 企业管理器、仿 EditPlus 的文本编辑器、多线程、断点下载工具……希望读者通过编写这些程序找到编程的乐趣。

2. 再现李刚老师课堂氛围

本书的内容是笔者 8 年多授课经历的总结，知识体系取自疯狂 Java 实战的课程体系。

本书力求再现笔者的课堂氛围：以浅显比喻代替乏味的讲解，以疯狂实战代替空洞的理论。

书中包含了大量“注意”、“学生提问”部分，这些正是几千个 Java 学员所犯错误的汇总。

3. 注释详细，轻松上手

为了降低读者阅读的难度，书中代码的注释非常详细，几乎每两行代码就有一行注释。不仅如此，本书甚至还把一些简单理论作为注释穿插到代码中，力求让读者能轻松上手。

本书所有程序中关键代码以粗体字标出，也是为了帮助读者能迅速找到这些程序的关键点。

本书写给谁看



如果你仅仅想对 Java 有所涉猎，那么本书并不适合你；如果你想全面掌握 Java 语言，并使用 Java 来解决问题、开发项目，或者希望以 Java 编程作为你的职业，那么本书将非常适合你。希望本书能引爆你内心潜在的编程激情，如果本书能让你产生废寝忘食的感觉，那笔者就非常欣慰了。

2014-06-15

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

以书为证彰显卓越品质

《C++ Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows 内核情景分析》、《加密与解密》、《编程之美》、《VC++ 深入详解》、《SEO 实战密码》、《PPT 演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉 Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与 IT 业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于 IT 专业出版之巅。

英雄帖

江湖风云起,代有才人出。

IT 界群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享 IT 心得

• 专业的作者服务 •

博文视点自成立以来一直专注于 IT 专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照 IT 技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身定制写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN 官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn





CONTENTS

第1章 Java语言概述与开发环境 1

1.1 Java语言的发展简史 2

1.2 Java的竞争对手及各自优势 4

1.2.1 C#简介和优势 4

1.2.2 Ruby简介和优势 5

1.2.3 Python简介和优势 5

1.3 Java程序运行机制 6

1.3.1 高级语言的运行机制 6

1.3.2 Java程序的运行机制和JVM 6

1.4 开发Java的准备 8

1.4.1 下载和安装Java 8的JDK 8

学生提问 不是说JVM是运行Java程序的虚拟机吗?那JRE和JVM的关系是怎样的呢? 8

学生提问 为什么不安装公共JRE呢? 9

1.4.2 设置PATH环境变量 10

为什么选择用户变量?用户变量与系统变量有什么区别? 11

1.5 第一个Java程序 12

1.5.1 编辑Java源代码 12

1.5.2 编译Java程序 12

学生提问 当编译C程序时,不仅需要指定存放目标文件的位置,也需要指定目标文件的文件名,这里使用javac编译Java程序时怎么不需要指定目标文件的文件名呢? 13

1.5.3 运行Java程序 13

1.5.4 根据CLASSPATH环境变量定位类 14

1.6 Java程序的基本规则 15

1.6.1 Java程序的组织形式 15

1.6.2 Java源文件的命名规则 16

1.6.3 初学者容易犯的错误 17

1.7 垃圾回收机制 19

1.8 何时开始使用IDE工具 20

学生提问 我想学习Java编程,到底是学习Eclipse好,还是学习NetBeans好呢? 21

1.9 本章小结 21

第2章 理解面向对象 22

2.1 面向对象 23

2.1.1 结构化程序设计简介 23

2.1.2 程序的三种基本结构 24

2.1.3 面向对象程序设计简介 26

2.1.4 面向对象的基本特征 27

2.2 UML(统一建模语言)介绍 28

2.2.1 用例图 30

2.2.2 类图 30

2.2.3 组件图 32

2.2.4 部署图 33

2.2.5 顺序图 33

2.2.6 活动图 34

2.2.7 状态机图 35

2.3 Java的面向对象特征 36

2.3.1 一切都是对象 36

2.3.2 类和对象 36

2.4 本章小结 37

第3章 数据类型和运算符 38

3.1 注释 39

3.1.1 单行注释和多行注释 39

3.1.2 文档注释 40

学生提问 API文档是什么? 40

学生提问 为什么要学习查看API文档的方法? 40

3.2 标识符和关键字 46

3.2.1 分隔符 46

3.2.2 标识符规则 47

3.2.3 Java关键字 47

3.3 数据类型分类 48

学生提问 什么是变量?变量有什么用? 48

3.4 基本数据类型 49

3.4.1 整型 49

3.4.2 字符型 51

学生提问 什么是字符集? 51

3.4.3 浮点型 53

3.4.4 数值中使用下划线分隔 54

3.4.5 布尔型 54

3.5 基本类型的类型转换 55

3.5.1 自动类型转换 55

3.5.2 强制类型转换 56

3.5.3 表达式类型的自动提升 58

3.6	直接量	59
3.6.1	直接量的类型	59
3.6.2	直接量的赋值	59
3.7	运算符	60
3.7.1	算术运算符	60
3.7.2	赋值运算符	63
3.7.3	位运算符	63
3.7.4	扩展后的赋值运算符	66
3.7.5	比较运算符	66
3.7.6	逻辑运算符	67
3.7.7	三目运算符	68
3.7.8	运算符的结合性和优先级	69
3.8	本章小结	70
第4章	流程控制与数组	71
4.1	顺序结构	72
4.2	分支结构	72
4.2.1	if 条件语句	72
4.2.2	Java 7 增强后的 switch 分支语句	76
4.3	循环结构	78
4.3.1	while 循环语句	78
4.3.2	do while 循环语句	79
4.3.3	for 循环	80
4.3.4	嵌套循环	83
4.4	控制循环结构	84
4.4.1	使用 break 结束循环	84
4.4.2	使用 continue 忽略本次循环剩下语句	85
4.4.3	使用 return 结束方法	86
4.5	数组类型	86
4.5.1	理解数组：数组也是一种类型	86
	int[] 是一种类型吗？怎么使用这种类型呢？	87
4.5.2	定义数组	87
4.5.3	数组的初始化	88
	能不能只分配内存空间，不赋初始值呢？	88
4.5.4	使用数组	89
	为什么要我记住这些异常信息？	89
4.5.5	foreach 循环	90
4.6	深入数组	91
4.6.1	内存中的数组	91
	为什么有栈内存和堆内存之分？	92
4.6.2	基本类型数组的初始化	94
4.6.3	引用类型数组的初始化	95
4.6.4	没有多维数组	97
	我是否可以图 4.13 中灰色覆盖的数组元素再次指向另一个数组？这样不就可以扩展成三维数组，甚至扩展成更多维的数组吗？	98
4.6.5	Java 8 增强的工具类：Arrays	99

4.6.6	数组的应用举例	102
4.7	本章小结	105
	本章练习	105
第5章	面向对象（上）	106
5.1	类和对象	107
5.1.1	定义类	107
	构造器不是没有返回值吗？为什么不能用 void 声明呢？	109
5.1.2	对象的产生和使用	110
5.1.3	对象、引用和指针	110
5.1.4	对象的 this 引用	111
5.2	方法详解	115
5.2.1	方法的所属性	115
5.2.2	方法的参数传递机制	116
5.2.3	形参个数可变的方法	119
5.2.4	递归方法	120
5.2.5	方法重载	122
	为什么方法的返回值类型不能用于区分重载的方法？	122
5.3	成员变量和局部变量	123
5.3.1	成员变量和局部变量	123
5.3.2	成员变量的初始化和内存中的运行机制	126
5.3.3	局部变量的初始化和内存中的运行机制	128
5.3.4	变量的使用规则	129
5.4	隐藏和封装	130
5.4.1	理解封装	130
5.4.2	使用访问控制符	130
5.4.3	package、import 和 import static	133
5.4.4	Java 的常用包	138
5.5	深入构造器	138
5.5.1	使用构造器执行初始化	138
	构造器是创建 Java 对象的途径，是不是说构造器完全负责创建 Java 对象？	139
5.5.2	构造器重载	139
	为什么要用 this 来调用另一个重载的构造器？我把另一个构造器里的代码复制、粘贴到这个构造器里不就可以了吗？	141
5.6	类的继承	141
5.6.1	继承的特点	141
5.6.2	重写父类的方法	142
5.6.3	super 限定	144
5.6.4	调用父类构造器	146
	为什么我创建 Java 对象时从未感觉到 java.lang. Object 类的构造器被调用过？	148
5.7	多态	148
5.7.1	多态性	148

5.7.2	引用变量的强制类型转换	150
5.7.3	instanceof 运算符	151
5.8	继承与组合	152
5.8.1	使用继承的注意点	152
5.8.2	利用组合实现复用	153
	使用组合关系来实现复用，需要创建两个 Animal 对象，是不是意味着使用组合关系时系统开销更大？	156
5.9	初始化块	156
5.9.1	使用初始化块	156
5.9.2	初始化块和构造器	158
5.9.3	静态初始化块	159
5.10	本章小结	161

第 6 章 面向对象（下）

6.1	Java 8 增强的包装类	163
	Java 为什么要对这些数据进行缓存呢？	166
6.2	处理对象	167
6.2.1	打印对象和 toString 方法	167
6.2.2	== 和 equals 方法	169
	上面程序中判断 obj 是否为 Person 类的实例时，为何不用 obj instanceof Person 来判断呢？	172
6.3	类成员	172
6.3.1	理解类成员	172
6.3.2	单例（Singleton）类	173
6.4	final 修饰符	174
6.4.1	final 成员变量	175
6.4.2	final 局部变量	176
6.4.3	final 修饰基本类型变量和引用类型变量的区别	177
6.4.4	可执行“宏替换”的 final 变量	178
6.4.5	final 方法	180
6.4.6	final 类	180
6.4.7	不可变类	181
6.4.8	缓存实例的不可变类	183
6.5	抽象类	186
6.5.1	抽象方法和抽象类	186
6.5.2	抽象类的作用	189
6.6	Java 8 改进的接口	190
6.6.1	接口的概念	190
6.6.2	Java 8 中接口的定义	190
6.6.3	接口的继承	193
6.6.4	使用接口	193
6.6.5	接口和抽象类	195
6.6.6	面向接口编程	195
6.7	内部类	199
6.7.1	非静态内部类	199
	非静态内部类对象和外部类对象	

	的关系是怎样的？	203
6.7.2	静态内部类	203
	为什么静态内部类的实例方法也不能访问外部类的实例属性呢？	204
	接口里是否能定义内部接口？	205
6.7.3	使用内部类	205
	既然内部类是外部类的成员，那么是否可以外部类定义子类，在子类中再定义一个内部类来重写其父类中的内部类呢？	208
6.7.4	局部内部类	208
6.7.5	Java 8 改进的匿名内部类	209
6.8	Java 8 新增的 Lambda 表达式	212
6.8.1	Lambda 表达式入门	212
6.8.2	Lambda 表达式与函数式接口	214
6.8.3	方法引用与构造器引用	216
6.8.4	Lambda 表达式与匿名内部类的联系和区别	218
6.8.5	使用 Lambda 表达式调用 Arrays 的类方法	219
6.9	枚举类	220
6.9.1	手动实现枚举类	220
6.9.2	枚举类入门	221
6.9.3	枚举类的成员变量、方法和构造器	222
6.9.4	实现接口的枚举类	224
	枚举类不是用 final 修饰了吗？怎么还能派生子类呢？	225
6.9.5	包含抽象方法的枚举类	225
6.10	对象与垃圾回收	226
6.10.1	对象在内存中的状态	227
6.10.2	强制垃圾回收	227
6.10.3	finalize 方法	229
6.10.4	对象的软、弱和虚引用	230
6.11	修饰符的适用范围	233
6.12	使用 JAR 文件	234
6.12.1	jar 命令详解	235
6.12.2	创建可执行的 JAR 包	236
6.12.3	关于 JAR 包的技巧	237
6.13	本章小结	238
	本章练习	238

第 7 章 Java 基础类库

7.1	与用户互动	240
7.1.1	运行 Java 程序的参数	240
7.1.2	使用 Scanner 获取键盘输入	241
7.2	系统相关	243
7.2.1	System 类	243
7.2.2	Runtime 类	245
7.3	常用类	246
7.3.1	Object 类	246

7.3.2	Java 7 新增的 Objects 类	247
7.3.3	String、StringBuffer 和 StringBuilder 类	248
7.3.4	Math 类	251
7.3.5	Java 7 的 ThreadLocalRandom 与 Random	253
7.3.6	BigDecimal 类	255
7.4	Java 8 的日期、时间类	257
7.4.1	Date 类	257
7.4.2	Calendar 类	258
7.4.3	Java 8 新增的日期、时间包	261
7.5	正则表达式	263
7.5.1	创建正则表达式	263
7.5.2	使用正则表达式	266
7.6	国际化与格式化	270
7.6.1	Java 国际化的思路	270
7.6.2	Java 支持的国家和语言	270
7.6.3	完成程序国际化	271
7.6.4	使用 MessageFormat 处理包含占 位符的字符串	273
7.6.5	使用类文件代替资源文件	274
7.6.6	使用 NumberFormat 格式化数字	274
7.6.7	使用 DateFormat 格式化日期、时间	276
7.6.8	使用 SimpleDateFormat 格式化日期	277
7.7	Java 8 新增的日期、时间格式器	278
7.7.1	使用 DateTimeFormatter 完成格式化	278
7.7.2	使用 DateTimeFormatter 解析字符串	279
7.8	本章小结	280
	本章练习	280

第 8 章 Java 集合 281

8.1	Java 集合概述	282
8.2	Collection 和 Iterator 接口	283
8.2.1	使用 Lambda 表达式遍历集合	285
8.2.2	使用 Java 8 增强的 Iterator 遍历集 合元素	286
8.2.3	使用 Lambda 表达式遍历 Iterator	287
8.2.4	使用 foreach 循环遍历集合元素	288
8.2.5	使用 Java 8 新增的 Predicate 操作集合	288
8.2.6	使用 Java 8 新增的 Stream 操作集合	289
8.3	Set 集合	291
8.3.1	HashSet 类	292
	 hashCode()方法对于 HashSet 是不 是十分重要?	293
8.3.2	LinkedHashSet 类	295
8.3.3	TreeSet 类	296
8.3.4	EnumSet 类	302
8.3.5	各 Set 实现类的性能分析	303
8.4	List 集合	304

8.4.1	Java 8 改进的 List 接口和 ListIterator 接口	304
8.4.2	ArrayList 和 Vector 实现类	307
8.4.3	固定长度的 List	308
8.5	Queue 集合	308
8.5.1	PriorityQueue 实现类	309
8.5.2	Deque 接口与 ArrayDeque 实现类	309
8.5.3	LinkedList 实现类	311
8.5.4	各种线性表的性能分析	312
8.6	Java 8 增强的 Map 集合	313
8.6.1	Java 8 为 Map 新增的方法	315
8.6.2	Java 8 改进的 HashMap 和 Hashtable 实现类	316
8.6.3	LinkedHashMap 实现类	319
8.6.4	使用 Properties 读写属性文件	319
8.6.5	SortedMap 接口和 TreeMap 实现类	320
8.6.6	WeakHashMap 实现类	323
8.6.7	IdentityHashMap 实现类	323
8.6.8	EnumMap 实现类	324
8.6.9	各 Map 实现类的性能分析	325
8.7	HashSet 和 HashMap 的性能选项	325
8.8	操作集合的工具类: Collections	326
8.8.1	排序操作	326
8.8.2	查找、替换操作	329
8.8.3	同步控制	330
8.8.4	设置不可变集合	330
8.9	烦琐的接口: Enumeration	331
8.10	本章小结	332
	本章练习	332

第 9 章 泛型 333

9.1	泛型入门	334
9.1.1	编译时不检查类型的异常	334
9.1.2	使用泛型	334
9.1.3	Java 7 泛型的“菱形”语法	335
9.2	深入泛型	336
9.2.1	定义泛型接口、类	336
9.2.2	从泛型类派生子类	338
9.2.3	并不存在泛型类	339
9.3	类型通配符	339
9.3.1	使用类型通配符	341
9.3.2	设定类型通配符的上限	341
9.3.3	设定类型形参的上限	343
9.4	泛型方法	344
9.4.1	定义泛型方法	344
9.4.2	泛型方法和类型通配符的区别	346
9.4.3	Java 7 的“菱形”语法与泛型构 造器	347

9.4.4	设定通配符下限	348	11.4	AWT 常用组件	397
9.4.5	泛型方法与重载	350	11.4.1	基本组件	397
9.4.6	Java 8 改进的类型推断	351	11.4.2	对话框 (Dialog)	399
9.5	擦除和转换	352	11.5	事件处理	401
9.6	泛型与数组	353	11.5.1	Java 事件模型的流程	401
9.7	本章小结	355	11.5.2	事件和事件监听器	403
第 10 章	异常处理	356	11.5.3	事件适配器	407
10.1	异常概述	357	11.5.4	使用内部类实现监听器	408
10.2	异常处理机制	358	11.5.5	使用外部类实现监听器	408
10.2.1	使用 try...catch 捕获异常	358	11.5.6	类本身作为事件监听器类	409
10.2.2	异常类的继承体系	360	11.5.7	匿名内部类实现监听器	410
10.2.3	Java 7 提供的多异常捕获	362	11.6	AWT 菜单	410
10.2.4	访问异常信息	363	11.6.1	菜单条、菜单和菜单项	410
10.2.5	使用 finally 回收资源	364	11.6.2	右键菜单	412
10.2.6	异常处理的嵌套	366		为什么即使我没有给多行文本域编写右键菜单, 但当我在多行文本域上单击右键时也一样会弹出右键菜单?	414
10.2.7	Java 7 的自动关闭资源的 try 语句	366	11.7	在 AWT 中绘图	414
10.3	Checked 异常和 Runtime 异常体系	368	11.7.1	画图的实现原理	414
10.3.1	使用 throws 声明抛出异常	368	11.7.2	使用 Graphics 类	415
10.4	使用 throw 抛出异常	370	11.8	处理位图	419
10.4.1	抛出异常	370	11.8.1	Image 抽象类和 BufferedImage 实现类	419
10.4.2	自定义异常类	371	11.8.2	使用 ImageIO 输入/输出位图	421
10.4.3	catch 和 throw 同时使用	372	11.9	剪贴板	425
10.4.4	Java 7 增强的 throw 语句	373	11.9.1	数据传递的类和接口	426
10.4.5	异常链	374	11.9.2	传递文本	426
10.5	Java 的异常跟踪栈	376	11.9.3	使用系统剪贴板传递图像	428
10.6	异常处理规则	377	11.9.4	使用本地剪贴板传递对象引用	430
10.6.1	不要过度使用异常	378	11.9.5	通过系统剪贴板传递 Java 对象	433
10.6.2	不要使用过于庞大的 try 块	379	11.10	拖放功能	435
10.6.3	避免使用 Catch All 语句	379	11.10.1	拖放目标	436
10.6.4	不要忽略捕获到的异常	379	11.10.2	拖放源	439
10.7	本章小结	380	11.11	本章小结	440
	本章练习	380		本章练习	440
第 11 章	AWT 编程	381	第 12 章	Swing 编程	441
11.1	GUI (图形用户界面) 和 AWT	382	12.1	Swing 概述	442
11.2	AWT 容器	383	12.2	Swing 基本组件的用法	443
11.3	布局管理器	386	12.2.1	Java 7 的 Swing 组件层次	443
11.3.1	FlowLayout 布局管理器	386	12.2.2	AWT 组件的 Swing 实现	444
11.3.2	BorderLayout 布局管理器	387		为什么单击 Swing 多行文本域时不是弹出像 AWT 多行文本域中的右键菜单?	450
	BorderLayout 最多只能放置 5 个组件吗? 那它也太不实用了吧?	388	12.2.3	为组件设置边框	450
11.3.3	GridLayout 布局管理器	389	12.2.4	Swing 组件的双缓冲和键盘驱动	452
11.3.4	GridBagLayout 布局管理器	390	12.2.5	使用 JToolBar 创建工具条	453
11.3.5	CardLayout 布局管理器	392	12.2.6	使用 JFileChooser 和 Java 7 增强的 JColorChooser	455
11.3.6	绝对定位	394	12.2.7	使用 JOptionPane	462
11.3.7	BoxLayout 布局管理器	395	12.3	Swing 中的特殊容器	467
	图 11.15 和图 11.16 显示的所有按钮都紧挨在一起, 如果希望像 FlowLayout、GridLayout 等布局管理器那样指定组件的间距应该怎么办?	396			

12.3.1	使用 JSplitPane	467
12.3.2	使用 JTabbedPane	469
12.3.3	使用 JLayeredPane、JdesktopPane 和 JInternalFrame.....	473
12.4	Swing 简化的拖放功能	480
12.5	Java 7 新增的 Swing 功能	481
12.5.1	使用 JLayer 装饰组件	481
12.5.2	创建透明、不规则形状窗口	487
12.6	使用 JProgressBar、ProgressMonitor 和 BoundedRangeModel 创建进度条	489
12.6.1	创建进度条	489
12.6.2	创建进度对话框	492
12.7	使用 JSlider 和 BoundedRangeModel 创建 滑动条	494
12.8	使用 JSpinner 和 SpinnerModel 创建微调控 制器	497
12.9	使用 JList、JComboBox 创建列表框	500
12.9.1	简单列表框	500
12.9.2	不强制存储列表项的 ListModel 和 ComboBoxModel.....	503
12.9.3	强制存储列表项的 DefaultListModel 和 DefaultComboBoxModel.....	506
	 为什么 JComboBox 提供了添加、删 除列表项的方法？而 JList 没有提供 添加、删除列表项的方法呢？	508
12.9.4	使用 ListCellRenderer 改变列表项 外观	508
12.10	使用 JTree 和 TreeModel 创建树	510
12.10.1	创建树	511
12.10.2	拖动、编辑树节点	513
12.10.3	监听节点事件	517
12.10.4	使用 DefaultTreeCellRenderer 改 变节点外观	519
12.10.5	扩展 DefaultTreeCellRenderer 改变 节点外观	520
12.10.6	实现 TreeCellRenderer 改变节点 外观	523
12.11	使用 JTable 和 TableModel 创建表格	524
12.11.1	创建表格	525
	 我们指定的表格数据、表格列标题 都是 Object 类型的数组，JTable 如 何显示这些 Object 对象？	525
12.11.2	TableModel 和监听器	530
12.11.3	TableColumnModel 和监听器	534
12.11.4	实现排序	537
12.11.5	绘制单元格内容	540
12.11.6	编辑单元格内容	543
12.12	使用 JFormattedTextField 和 JtextPane 创建格式文本	546
12.12.1	监听 Document 的变化	547
12.12.2	使用 JPasswordField	549
12.12.3	使用 JFormattedTextField	549

12.12.4	使用 JEditorPane	557
12.12.5	使用 JTextPane	557
12.13	本章小结	564
	本章练习	564

第 13 章 MySQL 数据库与 JDBC 编程

13.1	JDBC 基础	566
13.1.1	JDBC 简介	566
13.1.2	JDBC 驱动程序	567
13.2	SQL 语法	568
13.2.1	安装数据库	568
13.2.2	关系数据库基本概念和 MySQL 基本命令	570
13.2.3	SQL 语句基础	572
13.2.4	DDL 语句	573
13.2.5	数据库约束	577
13.2.6	索引	584
13.2.7	视图	585
13.2.8	DML 语句语法	585
13.2.9	单表查询	588
13.2.10	数据库函数	592
13.2.11	分组和组函数	594
13.2.12	多表连接查询	596
13.2.13	子查询	599
13.2.14	集合运算	601
13.3	JDBC 的典型用法	602
13.3.1	JDBC 4.2 常用接口和类简介	602
13.3.2	JDBC 编程步骤	604
	 前面给出的仅仅是 MySQL 和 Oracle 两种数据库的驱动，我看 不出驱动类字符串有什么规律啊。如 果我希望使用其他数据库，那怎么 找到其他数据库的驱动类呢？	604
13.4	执行 SQL 语句的方式	607
13.4.1	使用 Java 8 新增的 executeLargeUpdate 方法执行 DDL 和 DML 语句	607
13.4.2	使用 execute 方法执行 SQL 语句	608
13.4.3	使用 PreparedStatement 执行 SQL 语句	610
13.4.4	使用 CallableStatement 调用存储 过程	614
13.5	管理结果集	615
13.5.1	可滚动、可更新的结果集	615
13.5.2	处理 Blob 类型数据	617
13.5.3	使用 ResultSetMetaData 分析结 果集	622
13.6	Java 7 的 RowSet 1.1	624
13.6.1	Java 7 新增的 RowSetFactory 与 RowSet	625
13.6.2	离线 RowSet	627
13.6.3	离线 RowSet 的查询分页	629

13.7	事务处理.....	630	15.4.2	输入/输出流体系	674
13.7.1	事务的概念和 MySQL 事务支持	630	15.4.3	转换流	677
13.7.2	JDBC 的事务支持	632		怎么没有把字符流转换成字节流的转换流呢?	677
13.7.3	Java 8 增强的批量更新	634	15.4.4	推回输入流	678
13.8	分析数据库信息.....	635	15.5	重定向标准输入/输出.....	679
13.8.1	使用 DatabaseMetaData 分析数据库信息.....	635	15.6	Java 虚拟机读写其他进程的数据.....	680
13.8.2	使用系统表分析数据库信息	636	15.7	RandomAccessFile.....	682
13.8.3	选择合适的分析方式	637	15.8	对象序列化.....	686
13.9	使用连接池管理连接.....	638	15.8.1	序列化的含义和意义	686
13.9.1	DBCP 数据源.....	638	15.8.2	使用对象流实现序列化	686
13.9.2	C3PO 数据源	639	15.8.3	对象引用的序列化	688
13.10	本章小结.....	640	15.8.4	自定义序列化	692
	本章练习	640	15.8.5	另一种自定义序列化机制	696
			15.8.6	版本	698
第 14 章	Annotation (注释)	641	15.9	NIO.....	699
14.1	基本 Annotation.....	642	15.9.1	Java 新 IO 概述	699
14.1.1	限定重写父类方法: @Override	642	15.9.2	使用 Buffer	699
14.1.2	标示已过时: @Deprecated	643	15.9.3	使用 Channel	702
14.1.3	抑制编译器警告: @SuppressWarnings	644	15.9.4	字符集和 Charset	705
14.1.4	Java 7 的“堆污染”警告与 @SafeVarargs.....	644		二进制序列与字符之间如何对应呢?	706
14.1.5	Java 8 的函数式接口与 @FunctionalInterface	645	15.9.5	文件锁	707
14.2	JDK 的元 Annotation	646	15.10	Java 7 的 NIO.2.....	709
14.2.1	使用@Retention.....	646	15.10.1	Path、Paths 和 Files 核心 API	709
14.2.2	使用@Target	647	15.10.2	使用 FileVisitor 遍历文件和目录	710
14.2.3	使用@Documented	647	15.10.3	使用 WatchService 监控文件变化.....	711
14.2.4	使用@Inherited	648	15.10.4	访问文件属性	712
14.3	自定义 Annotation.....	649	15.11	本章小结	714
14.3.1	定义 Annotation	649		本章练习	714
14.3.2	提取 Annotation 信息	650	第 16 章	多线程	715
14.3.3	使用 Annotation 的示例	652	16.1	线程概述.....	716
14.3.4	Java 8 新增的重复注解	656	16.1.1	线程和进程	716
14.3.5	Java 8 新增的 Type Annotation	658	16.1.2	多线程的优势	717
14.4	编译时处理 Annotation.....	659	16.2	线程的创建和启动.....	718
14.5	本章小结.....	663	16.2.1	继承 Thread 类创建线程类	718
第 15 章	输入/输出	664	16.2.2	实现 Runnable 接口创建线程类	719
15.1	File 类	665	16.2.3	使用 Callable 和 Future 创建线程	720
15.1.1	访问文件和目录	665	16.2.4	创建线程的三种方式对比	722
15.1.2	文件过滤器	667	16.3	线程的生命周期.....	722
15.2	理解 Java 的 IO 流	668	16.3.1	新建和就绪状态	722
15.2.1	流的分类	668	16.3.2	运行和阻塞状态	724
15.2.2	流的概念模型	669	16.3.3	线程死亡	725
15.3	字节流和字符流.....	670	16.4	控制线程.....	726
15.3.1	InputStream 和 Reader	670	16.4.1	join 线程	726
15.3.2	OutputStream 和 Writer.....	672	16.4.2	后台线程	727
15.4	输入/输出流体系	673	16.4.3	线程睡眠: sleep	728
15.4.1	处理流的用法	674	16.4.4	线程让步: yield	729
			16.4.5	改变线程优先级	730

16.5	线程同步.....	731
16.5.1	线程安全问题.....	731
16.5.2	同步代码块.....	733
16.5.3	同步方法.....	735
16.5.4	释放同步监视器的锁定.....	737
16.5.5	同步锁 (Lock).....	737
16.5.6	死锁.....	739
16.6	线程通信.....	741
16.6.1	传统的线程通信.....	741
16.6.2	使用 Condition 控制线程通信.....	744
16.6.3	使用阻塞队列 (BlockingQueue) 控制线程通信.....	746
16.7	线程组和未处理的异常.....	749
16.8	线程池.....	752
16.8.1	Java 8 改进的线程池.....	752
16.8.2	Java 8 增强的 ForkJoinPool.....	754
16.9	线程相关类.....	757
16.9.1	ThreadLocal 类.....	757
16.9.2	包装线程不安全的集合.....	759
16.9.3	线程安全的集合类.....	759
16.10	本章小结.....	760

第 17 章 网络编程..... 761

17.1	网络编程的基础知识.....	762
17.1.1	网络基础知识.....	762
17.1.2	IP 地址和端口号.....	763
17.2	Java 的基本网络支持.....	764
17.2.1	使用 InetAddress.....	764
17.2.2	使用 URLDecoder 和 URLEncoder.....	765
17.2.3	URL、URLConnection 和 URLPermission.....	766
17.3	基于 TCP 协议的网络编程.....	772
17.3.1	TCP 协议基础.....	772
17.3.2	使用 ServerSocket 创建 TCP 服务器端.....	773
17.3.3	使用 Socket 进行通信.....	773
17.3.4	加入多线程.....	776
17.3.5	记录用户信息.....	778
17.3.6	半关闭的 Socket.....	785
17.3.7	使用 NIO 实现非阻塞 Socket 通信.....	786
17.3.8	使用 Java 7 的 AIO 实现非阻塞 通信.....	792



上面程序中好像没用到④⑤号代
码的 get()方法的返回值,这两个地
方不调用 get()方法行吗?..... 795

17.4	基于 UDP 协议的网络编程.....	798
17.4.1	UDP 协议基础.....	799
17.4.2	使用 DatagramSocket 发送、接收 数据.....	799
17.4.3	使用 MulticastSocket 实现多点广播.....	803
17.5	使用代理服务器.....	813
17.5.1	直接使用 Proxy 创建连接.....	813
17.5.2	使用 ProxySelector 自动选择代理 服务器.....	814
17.6	本章小结.....	817
	本章练习.....	817

第 18 章 类加载机制与反射..... 818

18.1	类的加载、连接和初始化.....	819
18.1.1	JVM 和类.....	819
18.1.2	类的加载.....	820
18.1.3	类的连接.....	821
18.1.4	类的初始化.....	821
18.1.5	类初始化的时机.....	822
18.2	类加载器.....	823
18.2.1	类加载器简介.....	823
18.2.2	类加载机制.....	824
18.2.3	创建并使用自定义的类加载器.....	826
18.2.4	URLClassLoader 类.....	829
18.3	通过反射查看类信息.....	830
18.3.1	获得 Class 对象.....	830
18.3.2	从 Class 中获取信息.....	831
18.3.3	Java 8 新增的方法参数反射.....	835
18.4	使用反射生成并操作对象.....	836
18.4.1	创建对象.....	836
18.4.2	调用方法.....	838
18.4.3	访问成员变量值.....	840
18.4.4	操作数组.....	841
18.5	使用反射生成 JDK 动态代理.....	842
18.5.1	使用 Proxy 和 InvocationHandler 创建动态代理.....	843
18.5.2	动态代理和 AOP.....	844
18.6	反射和泛型.....	848
18.6.1	泛型和 Class 类.....	848
18.6.2	使用反射来获取泛型信息.....	850
18.7	本章小结.....	851
	本章练习.....	851

第 1 章

Java 语言概述与开发环境

本章要点

- ✎ Java 语言的发展简史
- ✎ Java 语言各竞争对手的情况
- ✎ 编译型语言和解释型语言
- ✎ Java 语言的编译、解释运行机制
- ✎ 通过 JVM 实现跨平台
- ✎ 安装 JDK
- ✎ 设置 PATH 环境变量
- ✎ 编写、运行 Java 程序
- ✎ Java 程序的组织形式
- ✎ Java 程序的命名规则
- ✎ 初学者易犯的错误
- ✎ Java 的垃圾回收机制

Java 语言历时近二十年,已发展成为人类计算机史上影响深远的编程语言,从某种程度上来看,它甚至超出了编程语言的范畴,成为一种开发平台,一种开发规范。更甚至于:Java 已成为一种信仰,Java 语言所崇尚的开源、自由等精神,吸引了全世界无数优秀的程序员。事实是,从人类有史以来,从来没有一门编程语言能吸引这么多的程序员,也没有一门编程语言能衍生出如此之多的开源框架。

Java 语言是一门非常纯粹的面向对象编程语言,它吸收了 C++ 语言的各种优点,又摒弃了 C++ 里难以理解的多继承、指针等概念,因此 Java 语言具有功能强大和简单易用两个特征。Java 语言作为静态面向对象编程语言的代表,极好地实现了面向对象理论,允许程序员以优雅的思维方式进行复杂的编程开发。

不仅如此,Java 语言相关的 Java EE 规范里包含了时下最流行的各种软件工程理念,各种先进的设计思想总能在 Java EE 规范、平台以及相关框架里找到相应实现。从某种程度上来看,学精了 Java 语言的相关方面,相当于系统地学习了软件开发相关知识,而不是仅仅学完了一门编程语言。

时至今日,大部分银行、电信、证券、电子商务、电子政务等系统或者已经采用 Java EE 平台构建,或者正在逐渐过渡到采用 Java EE 平台来构建,Java EE 规范是目前最成熟的,也是应用最广的企业级应用开发规范。

1.1 Java 语言的发展简史

Java 语言的诞生具有一定的戏剧性,它并不是经过精心策划、制作,最后产生的划时代产品,从某个角度来看,Java 语言的诞生完全是一种误会。

1990 年年末,Sun 公司预料嵌入式系统将在未来家用电器领域大显身手。于是 Sun 公司成立了一个由 James Gosling 领导的“Green 计划”,准备为下一代智能家电(如电视机、微波炉、电话)编写一个通用控制系统。

该团队最初考虑使用 C++ 语言,但是很多成员包括 Sun 的首席科学家 Bill Joy,发现 C++ 和可用的 API 在某些方面存在很大问题。而且工作小组使用的是嵌入式平台,可用的系统资源极其有限。并且很多成员都发现 C++ 太复杂,以致很多开发者经常错误使用。而且 C++ 缺少垃圾回收系统、可移植性、分布式和多线程等功能。

根据可用的资金,Bill Joy 决定开发一种新语言,他提议在 C++ 的基础上,开发一种面向对象的环境。于是,Gosling 试图通过修改和扩展 C++ 的功能来满足这个要求,但是后来他放弃了。他决定创造一种全新的语言:Oak。

到了 1992 年的夏天,Green 计划已经完成了新平台的部分功能,包括 Green 操作系统、Oak 的程序设计语言、类库等。同年 11 月,Green 计划被转化成“FirstPerson 有限公司”,一个 Sun 公司的全资子公司。

FirstPerson 团队致力于创建一种高度互动的设备。当时代华纳公司发布了一个关于电视机顶盒的征求提议书时,FirstPerson 改变了他们的目标,作为对征求提议书的响应,提出了一个机顶盒平台的提议。但有线电视业界觉得 FirstPerson 的平台给予用户过多的控制权,因此 FirstPerson 的投标败给了 SGI。同时,与 3DO 公司的另外一笔关于机顶盒的交易也没有成功。此时,可怜的 Green 项目几乎接近夭折,甚至 Green 项目组的一半成员也被调到了其他项目组。

正如中国古代的寓言所言:塞翁失马,焉知非福?如果 Green 项目在机顶盒平台投标成功,也许就不会诞生 Java 这门伟大的语言了。

1994 年夏天,互联网和浏览器的出现不仅给广大互联网的用户带来了福音,也给 Oak 语言带来了新的生机。Gosling 立即意识到,这是一个机会,于是对 Oak 进行了小规模的改造,到了 1994 年秋,小组中的 Naughton 和 Jonathan Payne 完成了第一个 Java 语言的网页浏览器:WebRunner。Sun 公司实验室主任 Bert Sutherland 和技术总监 Eric Schmidt 观看了该浏览器的演示,对该浏览器的效果给予了高度评价。当时 Oak 这个商标已被别人注册,于是只得将 Oak 更名为 Java。

Sun 公司在 1995 年年初发布了 Java 语言，Sun 公司直接把 Java 放到互联网上，免费给大家使用。甚至连源代码也不保密，也放在互联网上向所有人公开。

几个月后，让所有人都大吃一惊的事情发生了：Java 成了互联网上最热门的宝贝。竟然有 10 万多人次访问了 Sun 公司的网页，下载了 Java 语言。然后，互联网上立即就有数不清的 Java 小程序（也就是 Applet），演示着各种小动画、小游戏等。

Java 语言终于扬眉吐气了，成为了一种广为人知的编程语言。

在 Java 语言出现之前，互联网的网页实质上就像是一张纸，不会有任何动态的内容。有了 Java 语言之后，浏览器的功能被扩大了，Java 程序可以直接在浏览器里运行，可以直接与远程服务器交互：用 Java 语言编程，可以在互联网上像传送电子邮件一样方便地传送程序文件！

1995 年，Sun 虽然推出了 Java，但这只是一种语言，如果想开发复杂的应用程序，必须要有一个强大的开发类库。因此，Sun 在 1996 年年初发布了 JDK 1.0。这个版本包括两部分：运行环境（即 JRE）和开发环境（即 JDK）。运行环境包括核心 API、集成 API、用户界面 API、发布技术、Java 虚拟机（JVM）5 个部分；开发环境包括编译 Java 程序的编译器（即 javac 命令）。

接着，Sun 在 1997 年 2 月 18 日发布了 JDK 1.1。JDK 1.1 增加了 JIT（即时编译）编译器。JIT 和传统的编译器不同，传统的编译器是编译一条，运行完后将其扔掉；而 JIT 会将经常用到的指令保存在内存中，当下次调用时就不需要重新编译了，通过这种方式让 JDK 在效率上有了较大提升。

但一直以来，Java 主要的应用就是网页上的 Applet 以及一些移动设备。到了 1996 年年底，Flash 面世了，这是一种更加简单的动画设计软件：使用 Flash 几乎无须任何编程语言知识，就可以做出丰富多彩的动画。随后 Flash 增加了 ActionScript 编程脚本，Flash 逐渐蚕食了 Java 在网页上的应用。

从 1995 年 Java 的诞生到 1998 年年底，Java 语言虽然成为了互联网上广泛使用的编程语言，但它并没有找到一个准确的定位，也没有找到它必须存在的理由：Java 语言可以编写 Applet，而 Flash 一样可以做到，而且更快，开发成本更低。

直到 1998 年 12 月，Sun 发布了 Java 历史上最重要的 JDK 版本：JDK 1.2，伴随 JDK 1.2 一同发布的还有 JSP/Servlet、EJB 等规范，并将 Java 分成了 J2EE、J2SE 和 J2ME 三个版本。

- J2ME：主要用于控制移动设备和信息家电等有限存储的设备。
- J2SE：整个 Java 技术的核心和基础，它是 J2ME 和 J2EE 编程的基础，也是这本书主要介绍的内容。
- J2EE：Java 技术中应用最广泛的部分，J2EE 提供了企业应用开发相关的完整解决方案。

这标志着 Java 已经吹响了向企业、桌面和移动三个领域进军的号角，标志着 Java 已经进入 Java 2 时代，这个时期也是 Java 飞速发展的时期。

在 Java 2 中，Java 发生了很多革命性的变化，而这些革命性的变化一直沿用到现在，对 Java 的发展形成了深远的影响。直到今天还经常看到 J2EE、J2ME 等名称。

不仅如此，JDK 1.2 还把它的 API 分成了三大类。

- 核心 API：由 Sun 公司制定的基本的 API，所有的 Java 平台都应该提供。这就是平常所说的 Java 核心类库。
- 可选 API：这是 Sun 为 JDK 提供的扩充 API，这些 API 因平台的不同而不同。
- 特殊 API：用于满足特殊要求的 API。如用于 JCA 和 JCE 的第三方加密类库。

2002 年 2 月，Sun 发布了 JDK 历史上最为成熟的版本：JDK 1.4。此时由于 Compaq、Fujitsu、SAS、Symbian、IBM 等公司的参与，使 JDK 1.4 成为发展最快的一个 JDK 版本。JDK 1.4 已经可以使用 Java 实现大多数的应用了。

在此期间，Java 语言在企业应用领域大放异彩，涌现出大量基于 Java 语言的开源框架：Struts、WebWork、Hibernate、Spring 等；大量企业应用服务器也开始涌现：WebLogic、WebSphere、JBoss 等，这些都标志着 Java 语言进入了飞速发展时期。

2004 年 10 月，Sun 发布了万众期待的 JDK 1.5，同时，Sun 将 JDK 1.5 改名为 Java SE 5.0，J2EE、

J2ME 也相应地改名为 Java EE 和 Java ME。JDK 1.5 增加了诸如泛型、增强的 for 语句、可变数量的形参、注释 (Annotations)、自动拆箱和装箱等功能;同时,也发布了新的企业级平台规范,如通过注释等新特性来简化 EJB 的复杂性,并推出了 EJB 3.0 规范。还推出了自己的 MVC 框架规范:JSF,JSF 规范类似于 ASP.NET 的服务器端控件,通过它可以快速地构建复杂的 JSP 界面。

2006 年 12 月, Sun 公司发布了 JDK 1.6 (也被称为 Java SE 6)。一直以来, Sun 公司维持着大约 2 年发布一次 JDK 新版本的习惯。

但在 2009 年 4 月 20 日, Oracle 宣布将以每股 9.5 美元的价格收购 Sun, 该交易的总价值约为 74 亿美元。而 Oracle 通过收购 Sun 公司获得了两项软件资产: Java 和 Solaris。

于是曾经代表一个时代的公司: Sun 终于被“雨打风吹”去,“江湖”上再也没有了 Sun 的身影。多年以后,在新一代的程序员心中可能会遗忘曾经的 Sun 公司,但老一辈的程序员们将永久地怀念 Sun 公司的传奇。

Sun 倒下了,不过 Java 的大旗依然“猎猎”作响。2007 年 11 月, Google 宣布推出一款基于 Linux 平台的开源手机操作系统: Android。Android 的出现顺应了即将出现的移动互联网潮流,而且 Android 系统的用户体验非常好,因此迅速成为手机操作系统的中坚力量。Android 平台使用了 Dalvik 虚拟机来运行 .dex 文件, Dalvik 虚拟机的作用类似于 JVM 虚拟机,只是它并未遵守 JVM 规范而已。Android 使用 Java 语言来开发应用程序,这也给了 Java 语言一个新的机会。在过去的岁月中, Java 语言作为服务器端编程语言,已经取得了极大的成功;而 Android 平台的流行,则让 Java 语言获得了在客户端程序上大展拳脚的机会。

2011 年 7 月 28 日, Oracle 公司终于“如约”发布了 Java SE 7——这次版本升级经过了将近 5 年时间。Java SE 7 也是 Oracle 发布的第一个 Java 版本,引入了二进制整数、支持字符串的 switch 语句、菱形语法、多异常捕捉、自动关闭资源的 try 语句等新特性。

2014 年 3 月 18 日, Oracle 公司发布了 Java SE 8,这次版本升级为 Java 带来了全新的 Lambda 表达式。除此之外, Java 8 还增加了大量新特性,这些新特性使得 Java 变得更加强大,本书后面将会详细介绍这些新特性。

1.2 Java 的竞争对手及各自优势

Java 语言目前是最流行的面向对象编程语言,与 Java 类似的程序设计语言还有 C#、Ruby 和 Python 等,它们在某些方面有自己的独特优势,因此都是 Java 语言有力的竞争者。

➤➤ 1.2.1 C#简介和优势

当年 Microsoft 也一度加入到 Java 语言阵营中, Microsoft 曾经在 Visual Studio 中提供了 Visual J++。正当 Microsoft 尽力在 Visual J++基础上拓展 Java 功能,并使之与 Windows 操作系统紧密结合在一起时, Sun 公司对 Microsoft 提出法律诉讼,称其违反了 Java 许可证协议中的条款,最终的结果是微软公司不得不停止 Visual J++产品的开发。1998 年 10 月以后, Microsoft 就不再发布新的 Visual J++版本,而且导致 Microsoft 一直站在 Java 阵营的对立面,甚至在 Windows XP 系统中不再提供 Java 运行时环境的支持。

接下来, Microsoft 推出了 .NET 平台,并发布了 C#语言,无论从哪个角度来看, C#程序设计语言都是 Microsoft 对 Java 语言的反击。自 C#诞生之日起,关于 C#与 Java 之间的论战便此起彼伏,至今不辍。从技术的角度来看, C#与 Java 都是对传统面向对象程序设计的组件化软件时代的革新成果,可谓殊途同归,两种编程语言甚至有“90%的重叠”。

Java 和 C#都对传统 C++艰深、晦涩的语法和语义进行了改进。在语法方面,两者都摒弃了 C++中函数及其参数的 const 修饰、宏替换、全局变量和全局函数等华而不实的地方;在继承方面,两者都采用了更易于理解的单继承和多接口的实现方案;在源代码组织方面,都提出了声明与实现于一体的逻辑

封装。

相对于 Java 语言, C#也有其独特的优势: Microsoft 提供的 Visual Studio 开发平台可以极好地提高 C#程序的开发效率, 而且 Microsoft 要比 Java 更善于利用 Windows 平台, 当使用 C#创建 Windows 服务、记录 Windows 事件日志、访问 Windows 注册表时, .NET 确实更方便。Microsoft 提供了 Windows 任务相关的大量基类, 允许程序员通过向导、拖放等操作来快速开发应用, 因此比较容易使用。

Java 的设计宗旨独立于任何平台, 自然不会提供太多的 Windows 特性。但这也正是 Java 语言的优势: 跨平台。对于一个企业应用而言, 永远无法确定这个应用需要在怎样的平台上运行, 如果你一旦选择了 C#语言, 那么你的应用就只能局限在 Windows 平台上。因此, 对于一个开放式的企业应用而言, 通常会选择 Java 作为开发语言, 而不是选择 C#。

1.2.2 Ruby 简介和优势

Ruby 语言由日本人松本行弘(まつもとゆきひろ, 英译: Yukihiro Matsumoto, 外号 Matz)于 1993 年起开始着手研发, 经历 2 年时间, 发布了 Ruby 语言的第一个版本: 0.95 版。据松本行弘的描述: 他一直想发明一种语言, 这种语言既能进行高效开发, 又能让开发人员享受编程的快乐。

事实上, Ruby 确实是一种非常简洁的解释型语言, 一种纯粹的面向对象编程语言, 甚至比 Java 更纯粹(在 Java 语言里, 还有基本数据类型等不是对象的变量, 但在 Ruby 语言里, 一切都是对象)。除此之外, Ruby 还提供了许多额外的便捷功能, 比如闭包、迭代和集合等, 这些都是为了达到 Ruby 语言创始人的梦想: 让 Ruby 开发者能享受编程的快乐。

相对于 Java 语言而言, Ruby 语言最大的特征就是简洁。首先, 它是一种弱类型的语言, 变量无须声明, 变量没有类型, 因此 Ruby 的变量可以保存任何类型的数据; 其次, 它还提供了强大的正则表达式支持, 并支持运算符重载; 除此之外, Ruby 也提供了许多额外的便捷功能, 比如闭包、代码块、迭代器和集合等。

Ruby 语言还有一个重要的优点: 它也是完全跨平台的, 可以在任何操作系统上解释执行。

2004 年, Ruby 语言阵营里出现了一个优秀的 MVC 框架: Ruby On Rails, 这个开发框架被宣传成现有企业框架的一个替代品。Ruby On Rails 框架是一个真正意义上的敏捷开发框架, 它提供了大量代码生成器, 通过使用这些代码生成器可以极好地提高应用的开发效率。

相对于 Java 领域的众多开发框架而言, Ruby on Rails 提供了一个“一站式”的解决方案, Ruby On Rails 框架提供了 Web 层的 MVC 框架、持久层的 ORM 等解决方案。借助于 Ruby 这种动态语言的优势, 整个应用的代码相当简洁, 因而使得 Ruby On Rails 应用的开发非常快速。

关于 Rails 框架和 Java EE 平台, 其实是各有优势的。Rails 平台的简洁性、易用性会在中小型应用上发挥出更大的吸引力; 但 Java EE 平台则提供了更多的选择, 适合对技术有精准把握的开发者, 用于解决有复杂需求的大型企业级应用。

1.2.3 Python 简介和优势

Python 由 Guido 于 1989 年年底开发, Python 语言是基于 ABC 教学语言的。ABC 这种语言非常优美和强大, 是专门为非专业程序员设计的。但是, ABC 语言并没有获得广泛的应用, Guido 认为是非开放造成的。

Guido 决心在 Python 中避免这一错误, 因此 Guido 加强了 Python 和其他语言如 C、C++和 Java 的结合性。同时, 他还实现了许多 ABC 中闪过过但未曾实现的东西。Python 的第一个实现是基于 Mac 机的。Python 由 ABC 语言发展而来, 主要受到了 Modula-3(另一种相当优美且强大的语言, 为小型团体所设计)的影响, 并且结合了 UNIX shell 和 C 的习惯。

Python 是一种面向对象的解释型编程语言, 也是一种功能强大而完善的通用型语言, 已经具有二十多年的发展历史, 成熟且稳定。Python 具有脚本语言中最丰富和强大的类库, 足以支持绝大多数日常应用。Python 语言具有简洁而清晰的语法特点, 适合完成各种高层任务, 几乎可以在所有的主流操

作系统上运行。

虽然 Python 也是一种解释型的脚本语言,但一些大规模的软件开发计划,比如 Zope、Mnet、BitTorrent 和 Google 都广泛地使用了该语言,而 Python 的支持者喜欢称它为高级动态编程语言,因此 Python 绝非 JavaScript 等只能处理简单任务的脚本语言所能比拟。

Python 的两大特色是可扩展性和清晰的语法。Python 新的内置模块 (module) 可以用 C 或 C++ 写成,也可为现成的模块加上 Python 的接口。Python 的设计者在设计它的时候认为:对于一个特定的问题,只要有一种最好的方法来解决就好了。因此,Python 甚至不是一种格式自由的语言,例如,它要求 if 语句的下一行必须向右缩进,否则不能通过编译。

1.3 Java 程序运行机制

Java 语言是一种特殊的高级语言,它既具有解释型语言的特征,也具有编译型语言的特征,因为 Java 程序要经过先编译,后解释两个步骤。

►► 1.3.1 高级语言的运行机制

计算机高级语言按程序的执行方式可以分为编译型和解释型两种。

编译型语言是指使用专门的编译器,针对特定平台(操作系统)将某种高级语言源代码一次性“翻译”成可被该平台硬件执行的机器码(包括机器指令和操作数),并包装成该平台所能识别的可执行性程序的格式,这个转换过程称为编译(Compile)。编译生成的可执行性程序可以脱离开发环境,在特定的平台上独立运行。

有些程序编译结束后,还可能需要对其他编译好的目标代码进行链接,即组装两个以上的目标代码模块生成最终的可执行性程序,通过这种方式实现低层次的代码复用。

因为编译型语言是一次性地编译成机器码,所以可以脱离开发环境独立运行,而且通常运行效率较高;但因为编译型语言的程序被编译成特定平台上的机器码,因此编译生成的可执行性程序通常无法移植到其他平台上运行;如果需要移植,则必须将源代码复制到特定平台上,针对特定平台进行修改,至少也需要采用特定平台上的编译器重新编译。

现有的 C、C++、Objective-C、Pascal 等高级语言都属于编译型语言。

解释型语言是指使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行的语言。解释型语言通常不会进行整体性的编译和链接处理,解释型语言相当于把编译型语言中的编译和解释过程混合到一起同时完成。

可以认为:每次执行解释型语言的程序都需要进行一次编译,因此解释型语言的程序运行效率通常较低,而且不能脱离解释器独立运行。但解释型语言有一个优势:跨平台比较容易,只需提供特定平台的解释器即可,每个特定平台上的解释器负责将源程序解释成特定平台的机器指令即可。解释型语言可以方便地实现源程序级的移植,但这是以牺牲程序执行效率为代价的。

现有的 Ruby、Python 等语言都属于解释型语言。

除此之外,还有一种伪编译型语言,如 Visual Basic,它属于半编译型语言,并不是真正的编译型语言。它首先被编译成 P-代码,并将解释引擎封装在可执行性程序内,当运行程序时,P-代码会被解析成真正的二进制代码。表面上看起来,Visual Basic 可以编译生成可执行性的 EXE 文件,而且这个 EXE 文件也可以脱离开发环境,在特定平台上运行,非常像编译型语言。实际上,在这个 EXE 文件中,既有程序的启动代码,也有链接解释程序的代码,而这部分代码负责启动 Visual Basic 解释程序,再对 Visual Basic 代码进行解释并执行。

►► 1.3.2 Java 程序的运行机制和 JVM

Java 语言比较特殊,由 Java 语言编写的程序需要经过编译步骤,但这个编译步骤并不会生成特定

平台的机器码，而是生成一种与平台无关的字节码（也就是*.class 文件）。当然，这种字节码不是可执行性的，必须使用 Java 解释器来解释执行。因此可以认为：Java 语言既是编译型语言，也是解释型语言。或者说，Java 语言既不是纯粹的编译型语言，也不是纯粹的解释型语言。Java 程序的执行过程必须经过先编译、后解释两个步骤，如图 1.1 所示。

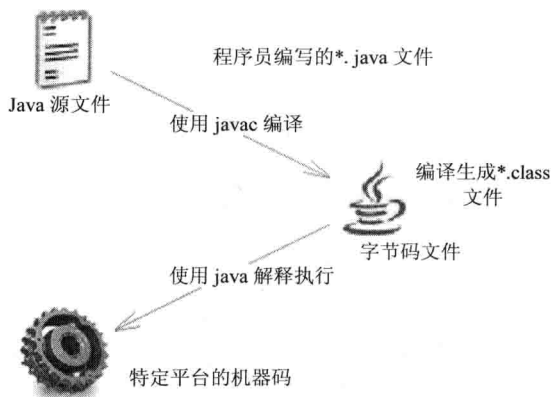


图 1.1 执行 Java 程序的两个步骤

Java 语言里负责解释执行字节码文件的是 Java 虚拟机，即 JVM（Java Virtual Machine）。JVM 是可运行 Java 字节码文件的虚拟计算机。所有平台上的 JVM 向编译器提供相同的编程接口，而编译器只需要面向虚拟机，生成虚拟机能理解的代码，然后由虚拟机来解释执行。在一些虚拟机的实现中，还会将虚拟机代码转换成特定系统的机器码执行，从而提高执行效率。

当使用 Java 编译器编译 Java 程序时，生成的是与平台无关的字节码，这些字节码不面向任何具体平台，只面向 JVM。不同平台上的 JVM 都是不同的，但它们都提供了相同的接口。JVM 是 Java 程序跨平台的关键部分，只要为不同平台实现了相应的虚拟机，编译后的 Java 字节码就可以在该平台上运行。显然，相同的字节码程序需要在不同的平台上运行，这几乎是“不可能的”，只有通过中间的转换器才可以实现，JVM 就是这个转换器。

JVM 是一个抽象的计算机，和实际的计算机一样，它具有指令集并使用不同的存储区域。它负责执行指令，还要管理数据、内存和寄存器。



提示：

JVM 的作用很容易理解，就像有两支不同的笔，但需要把同一个笔帽套在两支不同的笔上，只有为这两支笔分别提供一个转换器，这个转换器向上的接口相同，用于适应同一个笔帽；向下的接口不同，用于适应两支不同的笔。在这个类比中，可以近似地理解两支不同的笔就是不同的操作系统，而同一个笔帽就是 Java 字节码程序，转换器角色则对应 JVM。类似地，也可以认为 JVM 分为向上和向下两个部分，所有平台上的 JVM 向上提供给 Java 字节码程序的接口完全相同，但向下适应不同平台的接口则互不相同。

Oracle 公司制定的 Java 虚拟机规范在技术上规定了 JVM 的统一标准，具体定义了 JVM 的如下细节：

- 指令集
- 寄存器
- 类文件的格式
- 栈
- 垃圾回收堆
- 存储区

Oracle 公司制定这些规范的目的是为了提供统一的标准，最终实现 Java 程序的平台无关性。

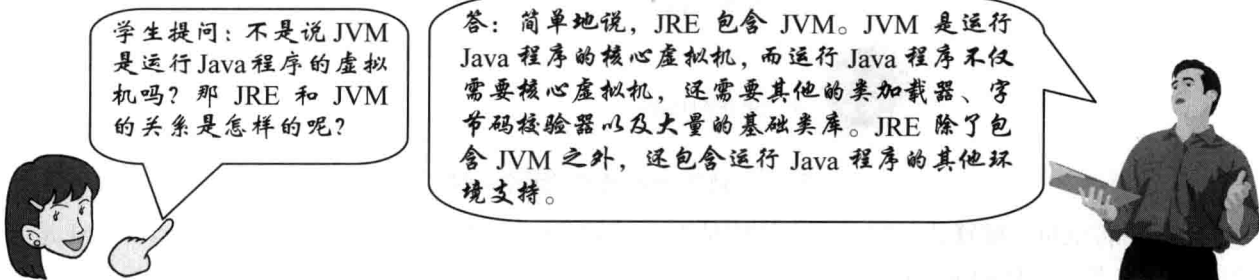
1.4 开发 Java 的准备

在开发 Java 程序之前,必须先完成一些准备工作,也就是在计算机上安装并配置 Java 开发环境,开发 Java 程序需要安装和配置 JDK。

1.4.1 下载和安装 Java 8 的 JDK

JDK 的全称是 Java SE Development Kit,即 Java 标准版开发包,是 Sun 提供的一套用于开发 Java 应用程序的开发包,它提供了编译、运行 Java 程序所需的各种工具和资源,包括 Java 编译器、Java 运行时环境,以及常用的 Java 类库等。

这里又涉及一个概念:Java 运行时环境,它的全称是 Java Runtime Environment,因此也被称为 JRE,它是运行 Java 程序的必需条件。



一般而言,如果只是运行 Java 程序,可以只安装 JRE,无须安装 JDK。



注意:

如果需要开发 Java 程序,则应该选择安装 JDK;当然,安装了 JDK 之后,就包含了 JRE,也可以运行 Java 程序。但如果只是运行 Java 程序,则需要在计算机上安装 JRE,仅安装 JVM 是不够的。实际上,Oracle 网站上提供的就是 JRE 的下载,并不提供单独 JVM 的下载。



Oracle 把 Java 分为 Java SE、Java EE 和 Java ME 三个部分,而且为 Java SE 和 Java EE 分别提供了 JDK 和 Java EE SDK (Software Development Kit) 两个开发包,如果读者只需要学习 Java SE 的编程知识,则可以下载标准的 JDK;如果读者学完 Java SE 之后,还需要继续学习 Java EE 相关内容,也可以选择下载 Java EE SDK,有一个 Java EE SDK 版本里已经包含了最新版的 JDK,安装 Java EE SDK 就包含了 JDK。

本书的内容主要是介绍 Java SE 的知识,因此下载标准的 JDK 即可。下载和安装 JDK 请按如下步骤进行。

① 登录 <http://www.oracle.com/technetwork/java/javase/downloads/index.html>,即可看到如图 1.2 所示的页面,下载 Java SE Development Kit 的最新版本。本书成书之时,JDK 的最新版本是 JDK 8u5,本书所有的案例也是基于该版本 JDK 的。

② 单击如图 1.2 所示页面中的链接,进入 JDK 8 的下载页面。读者应根

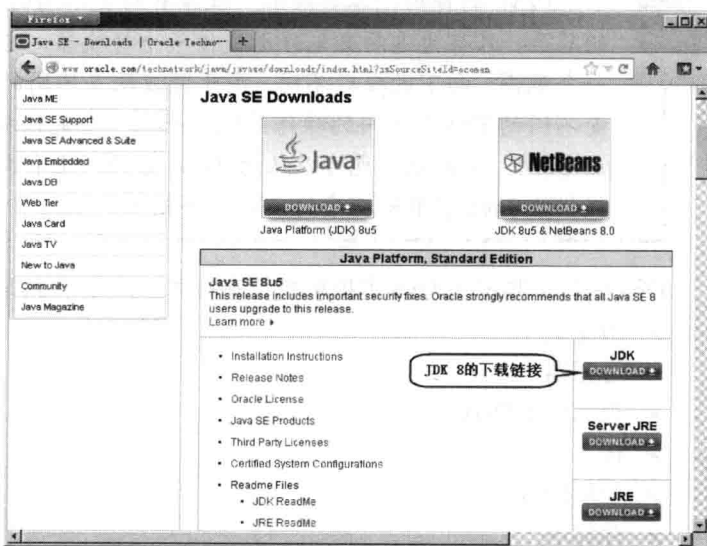


图 1.2 下载 JDK 的页面

据自己的平台选择合适的 JDK 版本：对于 Windows 平台，可以选择 Windows x86 或 Windows x64 版本（32 位的 Windows 系统选择 Windows x86 版本，64 位的 Windows 系统则选择 Windows x64 版本）；对于 Linux 平台，则下载 Linux 平台的 JDK。

**提示：**

在如图 1.2 所示页面上还可以看到 Server JRE 8 和 JRE 8 两个下载链接，这两个下载链接分别用于下载服务器版 JRE 和普通版 JRE，其中 Server JRE 包含 JVM 监控工具，以及服务器应用常用的工具，但是不包括浏览器插件；而普通版 JRE 则与之相反。

③ 64 位 Windows 系统的 JDK 下载成功后，得到一个 jdk-8u5-windows-x64.exe 文件，这是一个标准的 EXE 文件，可以通过双击该文件来运行安装程序。对于 Linux 平台上的 JDK 安装文件，只需为该文件添加可执行的属性，然后执行该安装文件即可。

④ 开始安装后，第一个对话框询问用户是否准备开始安装 JDK，单击“下一步”按钮，进入如图 1.3 所示的组件选择窗口。

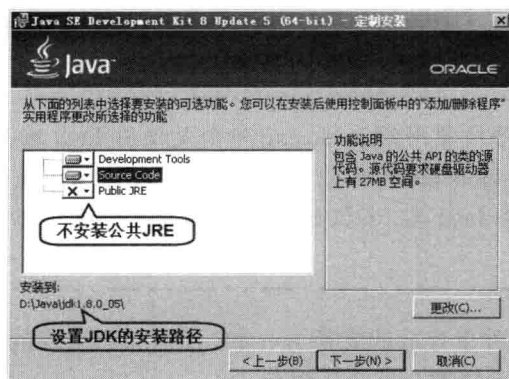


图 1.3 安装 JDK 的必需组件

大部分时候，并不需要安装所有的组件。在图 1.3 中，只需选择安装 JDK 的两个组件即可。

➤ **Development Tools**：这是 JDK 的核心，包括编译 Java 程序必需的命令工具。实际上，这个选项里已经包含了运行 Java 程序的 JRE，这个 JRE 会安装在 JDK 安装目录的子目录里，这也是无须安装公共 JRE 的原因。

学生提问：为什么不安装公共 JRE 呢？

答：公共 JRE 是一个独立的 JRE 系统，会单独安装在系统的其他路径下。公共 JRE 会向 IE 浏览器和系统中注册 Java 运行时环境。通过这种方式，系统中任何应用程序都可以使用公共 JRE。由于现在在网页上执行 Applet 的机会越来越少，而且完全可以选择使用 JDK 目录下的 JRE 来运行 Java 程序，因此没有太大必要安装公共 JRE。

➤ **Source Code**：安装这个选项将会安装 Java 所有核心类库的源代码。

⑤ 选择 JDK 的安装路径，系统默认安装在 C:\Program Files\Java 路径下，但不推荐安装在有空格的路径下，这样可能导致一些未知的问题，建议直接安装在根路径下，例如图 1.3 所示的 D:\Java\jdk1.8.0_05\。单击“下一步”按钮，等待安装完成。

安装完成后，可在 JDK 安装路径下看到如下的文件路径。

- **bin**：该路径下存放了 JDK 的各种工具命令，常用的 javac、java 等命令就放在该路径下。
- **db**：该路径是安装 Java DB 的路径。

- include: 一些平台特定的头文件。
- jre: 该路径下安装的就是运行 Java 程序所必需的 JRE 环境。
- lib: 该路径下存放的是 JDK 工具命令的实际执行程序, 如果使用 WinRAR 打开 lib 路径下的 tools.jar 文件, 将看到如图 1.4 所示的文件结构。



图 1.4 tools.jar 压缩文件的文件结构



提示:

用于编译 Java 程序所使用的 javac.exe 命令是使用 Java 编写的, 这个类就是 lib 路径下 tools.jar 文件中 sun\tools\javac 路径下的 Main 类, JDK 的 bin 路径下的 javac.exe 命令实际上仅仅是包装了这个 Java 类。不仅如此, bin 路径下的绝大部分命令都是包装了 tools.jar 文件里的工具类。

- javafx-src.zip: 该压缩文件里存放的就是 Java FX 所有核心类库的源代码, 本书不会涉及 Java FX 的相关内容, 因此读者无须理会这个压缩包。
- src.zip: 该压缩文件里存放的是 Java 所有核心类库的源代码。
- README 和 LICENSE 等说明性文档。

在上面路径中, bin 路径是一个非常有用的路径, 这个路径下包含了编译和运行 Java 程序的 javac 和 java 两个命令。除此之外, 还包含了 appletviewer、jar 等大量工具命令。本书的后面章节将会介绍该路径下的常用命令的用法。

➤➤ 1.4.2 设置 PATH 环境变量

前面已经介绍过了, 编译和运行 Java 程序必须经过两个步骤。

- ① 将源文件编译成字节码。
- ② 解释执行平台无关的字节码程序。

上面这两个步骤分别需要使用 java 和 javac 两个命令。启动 Windows 操作系统的命令行窗口 (在“开始”菜单里运行 cmd 命令即可), 在命令行窗口里依次输入 java 和 javac 命令, 将看到如下输出:

```
'java' 不是内部或外部命令, 也不是可运行的程序
或批处理文件。
'javac' 不是内部或外部命令, 也不是可运行的程序
或批处理文件。
```

这意味着还不能使用 java 和 javac 两个命令。这是因为: 虽然已经在计算机里安装了 JDK, 而 JDK 的安装路径下也包含了 java 和 javac 两个命令, 但计算机不知道到哪里去找这两个命令。

计算机如何查找命令呢? Windows 操作系统根据 Path 环境变量来查找命令。Path 环境变量的值是一系列路径, Windows 操作系统将在这一系列的路径中依次查找命令, 如果能找到这个命令, 则该命令是可执行的; 否则将出现“'xxx'不是内部或外部命令, 也不是可运行的程序或批处理文件”的提示。而 Linux 操作系统则根据 PATH 环境变量来查找命令, PATH 环境变量的值也是一系列路径。因为 Windows

操作系统不区分大小写，设置 Path 和 PATH 并没有区别；而 Linux 系统是区分大小写的，设置 Path 和 PATH 是有区别的，因此只需要设置 PATH 环境变量即可。

不管是 Linux 平台还是 Windows 平台，只需把 java 和 javac 两个命令所在的路径添加到 PATH 环境变量中，就可以编译和运行 Java 程序了。

1.4.2.1 在 Windows XP、Windows 7 等平台上设置环境变量

右击桌面上的“计算机”图标，出现右键菜单；单击“属性”菜单项，系统显示“控制面板\所有控制面板项\系统”窗口，单击该窗口左边栏中的“高级系统设置”链接，出现“系统属性”对话框；单击该对话框中的“高级”Tab 页，出现如图 1.5 所示的对话框。

单击“环境变量”按钮，将看到如图 1.6 所示的“环境变量”对话框，通过该对话框可以修改或添加环境变量。

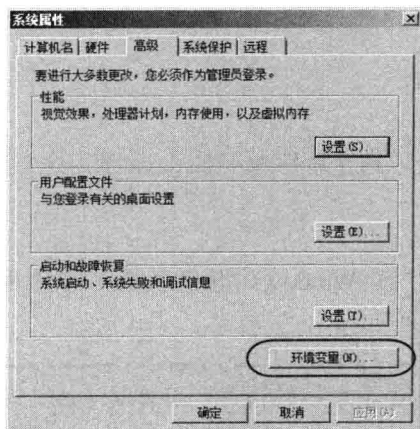


图 1.5 “系统属性”对话框

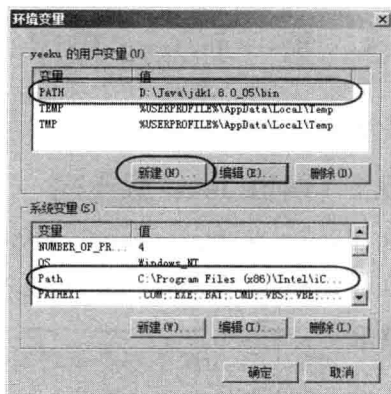


图 1.6 “环境变量”对话框

如图 1.6 所示的对话框上面的“用户变量”部分用于设置当前用户的环境变量，下面的“系统变量”部分用于设置整个系统的环境变量。对于 Windows 系统而言，名为 Path 的系统环境变量已经存在，可以直接修改该环境变量，在该环境变量值后追加 D:\Java\jdk1.8.0_05\bin（其中 D:\Java\jdk1.8.0_05\是本书 JDK 的安装路径）。实际上通常建议添加用户变量，单击“新建”按钮，添加名为 PATH 的环境变量，设置 PATH 环境变量的值为 D:\Java\jdk1.8.0_05\bin。

学生提问：为什么选择用户变量？用户变量与系统变量有什么区别？

答：用户变量和系统变量并没有太大的差别，只是用户变量只对当前用户有效，而系统变量对所有用户有效。为了减少自己所做的修改对其他人的影响，故设置用户变量避免影响其他人。对于当前用户而言，设置用户变量和系统变量的效果大致相同，只是系统变量的路径排在用户变量的路径之前。这可能出现一种情况：如果 Path 系统变量的路径里包含了 java 命令，而 PATH 用户变量的路径里也包含了 java 命令，则优先执行 Path 系统变量路径里包含的 java 命令。

1.4.2.2 在 Linux 上设置环境变量

进入当前用户的 home 路径下，然后在 home 路径下输入如下命令：

```
ls -a
```

该命令将列出当前路径下所有的文件，包括隐藏文件，Linux 平台的环境变量是通过 .bash_profile 文件来设置的。使用无格式编辑器打开该文件，在该文件的 PATH 变量后添加：/home/yeeku/Java/jdk1.8.0_05/bin，其中/home/yeeku/Java/jdk1.8.0_05/是本书的 JDK 安装路径。修改后的 PATH 变量

设置如下:

```
# 设置 PATH 环境变量
PATH=.:$PATH:$HOME/bin:/home/yeeku/Java/jdk1.8.0_05/bin
```

Linux 平台与 Windows 平台不一样, 多个路径之间以冒号(:)作为分隔符, 而\$PATH 则用于引用原有的 PATH 变量值。

完成了 PATH 变量值的设置后, 在.bash_profile 文件最后添加导出 PATH 变量的语句, 如下所示:

```
# 导出 PATH 环境变量
export PATH
```

重新登录 Linux 平台, 或者执行如下命令:

```
source .bash_profile
```

两种方式都是为了运行该文件, 让文件中设置的 PATH 变量值生效。

1.5 第一个 Java 程序

本节将编写编程语言里最“著名”的程序: HelloWorld, 以这个程序来开始 Java 学习之旅。

1.5.1 编辑 Java 源代码

编辑 Java 源代码可以使用任何无格式的文本编辑器, 在 Windows 操作系统上可使用记事本(NotePad)、EditPlus 等程序, 在 Linux 平台上可使用 VI 工具等。



提示:

编写 Java 程序不要使用写字板, 更不可使用 Word 等文档编辑器。因为写字板、Word 等工具有格式的编辑器, 当使用它们编辑一份文档时, 这个文档中会包含一些隐藏的格式化字符, 这些隐藏字符会导致程序无法正常编译、运行。

在记事本中新建一个文本文件, 并在该文件中输入如下代码。

程序清单: codes\01\1.5\HelloWorld.java

```
public class HelloWorld
{
    // Java 程序的入口方法, 程序将从这里开始执行
    public static void main(String[] args)
    {
        // 向控制台打印一条语句
        System.out.println("Hello World!");
    }
}
```

编辑上面的 Java 文件时, 注意程序中粗体字标识的单词, Java 程序严格区分大小写。将上面文本文件保存为 HelloWorld.java, 该文件就是 Java 程序的源程序。

编写好 Java 程序的源代码后, 接下来就应该编译该 Java 源文件来生成字节码了。

1.5.2 编译 Java 程序

编译 Java 程序需要使用 javac 命令, 因为前面已经把 javac 命令所在的路径添加到了系统的 PATH 环境变量中, 因此现在可以使用 javac 命令来编译 Java 程序了。

如果直接在命令行窗口里输入 javac, 不跟任何选项和参数, 系统将会输出大量提示信息, 用以提示 javac 命令的用法, 读者可以参考该提示信息来使用 javac 命令。

对于初学者而言, 先掌握 javac 命令的如下用法:

```
javac -d destdir srcFile
```

在上面命令中, `-d destdir` 是 `javac` 命令的选项, 用以指定编译生成的字节码文件的存放路径, `destdir` 只需是本地磁盘上的一个有效路径即可; 而 `srcFile` 是 Java 源文件所在的位置, 这个位置既可以是绝对路径, 也可以是相对路径。

通常, 总是将生成的字节码文件放在当前路径下, 当前路径可以用一点 (.) 来表示。在命令行窗口进入 `HelloWorld.java` 文件所在路径, 在该路径下输入如下命令:

```
javac -d . HelloWorld.java
```

运行该命令后, 在该路径下生成一个 `HelloWorld.class` 文件。

学生提问: 当编译 C 程序时, 不仅需要指定存放目标文件的位置, 也需要指定目标文件的文件名, 这里使用 `javac` 编译 Java 程序时怎么不需要指定目标文件的文件名呢?

答: 使用 `javac` 编译文件只需要指定存放目标文件的位置即可, 无须指定字节码文件的文件名。因为 `javac` 编译后生成的字节码文件有默认的文件名: 文件名总是以源文件所定义类的类名作为主文件名, 以 `.class` 作为后缀名。这意味着如果一个源文件里定义了多个类, 将编译生成多个字节码文件。事实上, 指定目标文件存放位置的 `-d` 选项也是可省略的, 如果省略该项, 则意味着将生成的字节码文件放在当前路径下。



如果读者喜欢用 `EditPlus` 作为无格式编辑器, 则可以使用 `EditPlus` 把 `javac` 命令集成进来, 从而直接在 `EditPlus` 编辑器中编译 Java 程序, 而无须每次启动命令行窗口。

在 `EditPlus` 中集成 `javac` 命令按如下步骤进行。

- ① 选择 `EditPlus` 的“工具”→“配置用户工具”菜单, 弹出如图 1.7 所示的对话框。
- ② 单击“组名称”按钮来设置工具组的名称, 例如输入“编译运行 Java”。单击“添加工具”按钮, 并选择“程序”选项, 然后输入 `javac` 命令的用法和参数, 输入成功后看到如图 1.8 所示的界面。

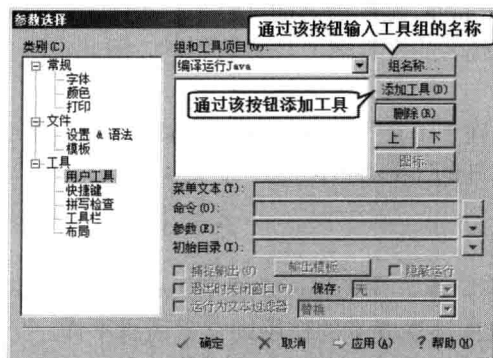


图 1.7 集成用户工具的对话框

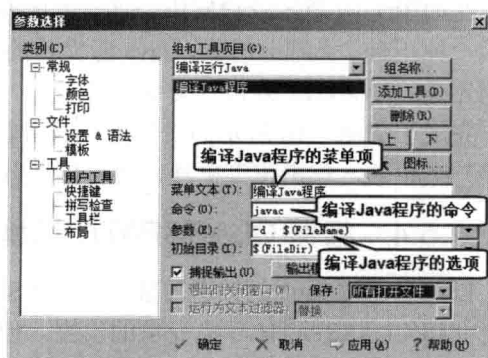


图 1.8 集成编译 Java 程序的工具

- ③ 单击“确定”按钮, 返回 `EditPlus` 主界面。再次选择 `EditPlus` 的“工具”菜单, 将看到该菜单中增加了“编译 Java 程序”菜单项, 单击该菜单项即可编译 `EditPlus` 当前打开的 Java 源程序代码。

1.5.3 运行 Java 程序

运行 Java 程序使用 `java` 命令, 启动命令行窗口, 进入 `HelloWorld.class` 所在的位置, 在命令行窗口里直接输入 `java` 命令, 不带任何参数或选项, 将看到系统输出大量提示, 告诉开发者如何使用 `java` 命令。

对于初学者而言, 当前只要掌握 `java` 命令的如下用法即可:

```
java Java 类名
```

值得注意的是, `java` 命令后的参数是 Java 类名, 而不是字节码文件的文件名, 也不是 Java 源文件名。

通过命令行窗口进入 HelloWorld.class 所在的路径, 输入如下命令:

```
java HelloWorld
```

运行上面命令, 将看到如下输出:

```
Hello World!
```

这表明 Java 程序运行成功。

如果运行 java helloworld 或者 java helloWorld 等命令, 将会看到如图 1.9 所示的错误提示。

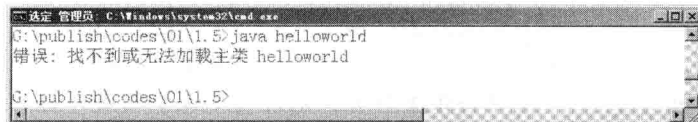


图 1.9 类名大小写不正确的提示

因为 Java 是区分大小写的语言, 所以 java 命令后的类名必须严格区分大小写。

与编译 Java 程序类似的是, 也可以在 EditPlus 里集成运行 Java 程序的工具, 集成运行 Java 程序的设置界面如图 1.10 所示。

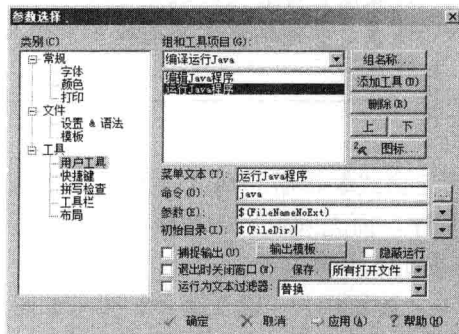


图 1.10 集成运行 Java 程序的设置界面

在如图 1.10 所示的设置中, 似乎运行 Java 程序的命令是 java 无扩展名的文件名, 实际上这只是利用了一种巧合: 大部分时候, Java 源文件的主文件名 (无扩展名的文件名) 与类名相同, 因此实际上执行的还是 “java Java 类名” 命令。

完成了如图 1.10 所示的设置后, 返回 EditPlus 主界面, 在 “工具” 菜单中将会增加一个 “运行 Java 程序” 菜单项, 单击该菜单项, 将可以运行 EditPlus 当前打开的 Java 程序。

1.5.4 根据 CLASSPATH 环境变量定位类

以前学习过 Java 的读者可能对 CLASSPATH 环境变量不陌生, 几乎每一本介绍 Java 入门的图书里都会介绍 CLASSPATH 环境变量的设置, 但对于 CLASSPATH 环境变量的作用则常常语焉不详。

实际上, 如果使用 1.5 以上版本的 JDK, 完全可以不用设置 CLASSPATH 环境变量——正如上面编译、运行 Java 程序所见到的, 即使不设置 CLASSPATH 环境变量, 完全可以正常编译和运行 Java 程序。

那么 CLASSPATH 环境变量的作用是什么呢? 当使用 “java Java 类名” 命令来运行 Java 程序时, JRE 到哪里去搜索 Java 类呢? 可能有读者会回答, 在当前路径下搜索啊。这个回答很聪明, 但 1.4 以前版本的 JDK 都没有设计这个功能, 这意味着即使当前路径已经包含了 HelloWorld.class, 并在当前路径下执行 “java HelloWorld”, 系统将一样提示找不到 HelloWorld 类。

如果使用 1.4 以前版本的 JDK, 则需要在 CLASSPATH 环境变量中添加一点 (.), 用以告诉 JRE 需要在当前路径下搜索 Java 类。

除此之外, 编译和运行 Java 程序还需要 JDK 的 lib 路径下 dt.jar 和 tools.jar 文件中的 Java 类, 因此还需要把这两个文件添加到 CLASSPATH 环境变量里。

因此, 如果使用 1.4 以前版本的 JDK 来编译和运行 Java 程序, 常常需要设置 CLASSPATH 环境变

量的值为`.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar` (其中`%JAVA_HOME%`代表 JDK 的安装目录)。

后来的 JRE 会自动搜索当前路径下的类文件, 而且使用 Java 的编译和运行工具时, 系统可以自动加载 `dt.jar` 和 `tools.jar` 文件中的 Java 类, 因此不再需要设置 `CLASSPATH` 环境变量。



注意：只有使用早期版本的 JDK 时, 才需要设置 `CLASSPATH` 环境变量。



当然, 即使使用 JDK 1.5 以上版本的 JDK, 也可以设置 `CLASSPATH` 环境变量, 一旦设置了该环境变量, JRE 将会按该环境变量指定的路径来搜索 Java 类。这意味着如果 `CLASSPATH` 环境变量中不包括一点 (`.`), 也就是没有包含当前路径, JRE 不会在当前路径下搜索 Java 类。

如果想在运行 Java 程序时临时指定 JRE 搜索 Java 类的路径, 则可以使用 `-classpath` 选项, 即按如下格式来运行 `java` 命令:

```
java -classpath dir1;dir2;dir3...;dirN Java 类
```

`-classpath` 选项的值可以是一系列的路径, 多个路径之间在 Windows 平台上以分号 (`;`) 隔开, 在 Linux 平台上则以冒号 (`:`) 隔开。

如果在运行 Java 程序时指定了 `-classpath` 选项的值, JRE 将严格按 `-classpath` 选项所指定的路径来搜索 Java 类, 既不会在当前路径下搜索 Java 类, `CLASSPATH` 环境变量所指定的搜索路径也不再有效。

如果想使 `CLASSPATH` 环境变量指定的搜索路径有效, 而且还会在当前路径下搜索 Java 类, 则可以按如下格式来运行 Java 程序:

```
java -classpath %CLASSPATH%;.;dir1;dir2;dir3...;dirN Java 类
```

上面命令通过 `%CLASSPATH%` 来引用 `CLASSPATH` 环境变量的值, 并在 `-classpath` 选项的值里添加了一点, 强制 JRE 在当前路径下搜索 Java 类。

1.6 Java 程序的基本规则

前面已经编写了 Java 学习之旅的第一个程序, 下面对这个简单的 Java 程序进行一些解释, 解释 Java 程序必须满足的基本规则。

1.6.1 Java 程序的组织形式

Java 程序是一种纯粹的面向对象的程序设计语言, 因此 Java 程序必须以类 (`class`) 的形式存在, 类 (`class`) 是 Java 程序的最小程序单位。Java 程序不允许可执行性语句、方法等成分独立存在, 所有的程序部分都必须放在类定义里。

上面的 `HelloWorld.java` 程序是一个简单的程序, 但还不是最简单的 Java 程序, 最简单的 Java 程序是只包含一个空类定义的程序。下面将编写一个最简单的 Java 程序。

程序清单: `codes\01\1.6\Test.java`

```
class Test
{
}
```

这是一个最简单的 Java 程序, 这个程序定义了一个 `Test` 类, 这个类里没有任何的类成分, 是一个空类, 但这个 Java 程序是绝对正确的, 如果使用 `javac` 命令来编译这个程序, 就知道这个程序可以通过编译, 没有任何问题。

但如果使用 `java` 命令来运行上面的 `Test` 类, 则会得到如下错误提示:

```
错误: 在类 Test 中找不到 main 方法, 请将 main 方法定义为:
public static void main(String[] args)
```


上面的错误提示仅仅表明：这个类不能被 java 命令解释执行，并不表示这个类是错误的。实际上，Java 解释器规定：如需某个类能被解释器直接解释执行，则这个类里必须包含 main 方法，而且 main 方法必须使用 public static void 来修饰，且 main 方法的形参必须是字符串数组类型（String[] args 是字符串数组的形式）。也就是说，main 方法的写法几乎是固定的。Java 虚拟机就从这个 main 方法开始解释执行，因此，main 方法是 Java 程序的入口。至于 main 方法为何要采用这么“复杂”的写法，后面章节会有更详细的解释，读者现在只能把这个方法死记下来。

对于那些不包含 main 方法的类，也是有用的类。对于一个大型的 Java 程序而言，往往只需要一个入口，也就是只有一个类包含 main 方法，而其他类都是用于被 main 方法直接或间接调用的。

1.6.2 Java 源文件的命名规则

Java 程序源文件的命名不是随意的，Java 文件的命名必须满足如下规则。

- Java 程序源文件的后缀必须是 .java，不能是其他文件后缀名。
- 在通常情况下，Java 程序源文件的主文件名可以是任意的。但有一种情况例外：如果 Java 程序源代码里定义了一个 public 类，则该源文件的主文件名必须与该 public 类（也就是该类定义使用了 public 关键字修饰）的类名相同。

由于 Java 程序源文件的文件名必须与 public 类的类名相同，因此，一个 Java 源文件里最多只能定义一个 public 类。



注意：一个 Java 源文件可以包含多个类定义，但最多只能包含一个 public 类定义；如果 Java 源文件里包含 public 类定义，则该源文件的文件名必须与这个 public 类的类名相同。



虽然 Java 源文件里没有包含 public 类定义时，这个源文件的文件名可以是随意的，但推荐让 Java 源文件的主文件名与类名相同，这可以提供更好的可读性。通常有如下建议：

- 一个 Java 源文件只定义一个类，不同的类使用不同的源文件定义。
- 让 Java 源文件的主文件名与该源文件中定义的 public 类同名。

在疯狂软件的教学过程中，发现很多学员经常犯一个错误，他们在保存一个 Java 文件时，常常保存成形如 *.java.txt 的文件名，而且这种文件名看起来非常像是 *.java。这是 Windows 的默认设置所引起的，Windows 默认会“隐藏已知文件类型的扩展名”。为了避免这个问题，通常推荐关闭 Windows 的“隐藏已知文件类型的扩展名”功能。

为了关闭“隐藏已知文件类型的扩展名”功能，在 Windows 的资源管理器窗口打开“组织”菜单，然后单击“文件夹和搜索选项”菜单项，将弹出“文件夹选项”对话框，单击该对话框里的“查看”Tab 页，看到如图 1.11 所示的对话框。

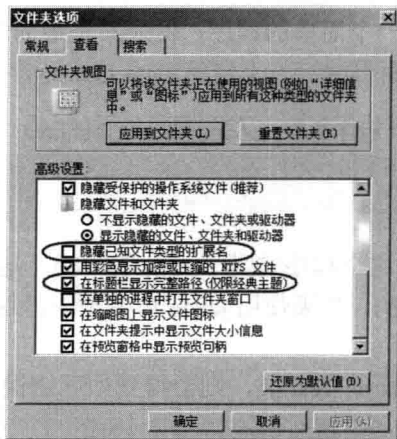


图 1.11 “文件夹选项”对话框

去掉“隐藏已知文件类型的扩展名”选项之前的钩,则可以让所有文件显示真实的文件名,从而避免 HelloWorld.java.txt 这样的错误。

另外,图 1.11 中还显示勾选了“在标题栏显示完整路径(仅限经典主题)”选项,这对于开发中准确定位 Java 源文件也很有帮助。

1.6.3 初学者容易犯的错误

万事开头难,Java 编程的初学者常常会遇到各种各样的问题,对于在学校跟着老师学习的读者而言,可以直接通过询问老师来解决这些问题;但对于自学的读者而言,则需要花更多时间、精力来解决这些问题,而且一旦遇到的问题几天都得不到解决,往往会带给他们很大的挫败感。

下面介绍一些初学者经常出现的错误,希望减少读者在学习中的障碍。

1. CLASSPATH 环境变量的问题

由于历史原因,几乎所有的图书和资料中都介绍必须设置这个环境变量。实际上,正如前面所介绍的,如果使用 1.5 以上版本的 JDK,完全可以不用设置这个环境变量。如果不设置这个环境变量,将可以正常编译和运行 Java 程序。

相反,如果有的读者看过其他 Java 入门书籍,或者参考过网上的各种资料(网络是一个最大的资源库,但网络上的资料又是鱼龙混杂、良莠不齐的。网络上的资料很多都是转载的,只要一个人提出一个错误的说法,这个错误的说法可能被成千上万的人转载,从而看到成千上万的错误说法),可能总是习惯设置 CLASSPATH 环境变量。

设置 CLASSPATH 环境变量没有错,关键是设置错了就比较麻烦了。正如前面所介绍的,如果没有设置 CLASSPATH 环境变量,Java 解释器将会在当前路径下搜索 Java 类,因此在 HelloWorld.class 文件所在的路径运行 java HelloWorld 将没有任何问题;但如果设置了 CLASSPATH 环境变量,Java 解释器将只在 CLASSPATH 环境变量所指定的系列路径中搜索 Java 类,这样就容易出现问题了。

由于很多资料上提到 CLASSPATH 环境变量中应该添加 dt.jar 和 tools.jar 两个文件,因此很多读者会设置 CLASSPATH 环境变量的值为: D:\Java\jdk1.8.0_05\lib\dt.jar;D:\Java\jdk1.8.0_05\lib\tools.jar,这将导致 Java 解释器不在当前路径下搜索 Java 类。如果此时在 HelloWorld.class 文件所在的路径运行 java HelloWorld,将出现如下错误提示:

错误: 找不到或无法加载主类 HelloWorld

上面的错误是一个典型错误:找不到类定义的错误,通常都是由 CLASSPATH 环境变量设置不正确造成的。因此,如果读者要设置 CLASSPATH 环境变量,一定不要忘记在 CLASSPATH 环境变量中增加一点(.),强制 Java 解释器在当前路径下搜索 Java 类。



提示:

如果指定了 CLASSPATH 环境变量,一定不要忘记在 CLASSPATH 环境变量中增加一点(.),一点代表当前路径,用以强制 Java 解释器在当前路径下搜索 Java 类。

除此之外,有的读者在设置 CLASSPATH 环境变量时总是仗着自己记忆很好,往往选择手动输入 CLASSPATH 环境变量的值,这非常容易引起错误:偶然的手误,或者多一个空格,或者少一个空格,都有可能引起错误。

实际上,有更好的方法来解决这个错误,完全可以在文件夹的地址栏里看到某个文件或文件夹的完整路径,就可以直接通过复制、粘贴来设置 CLASSPATH 环境变量了。

通过资源管理器打开 JDK 安装路径,将可以看到如图 1.12 所示的界面。

读者可以通过复制地址栏里的字符串来设置环境变量,而不是采用手动输入,从而减少出错的可能。

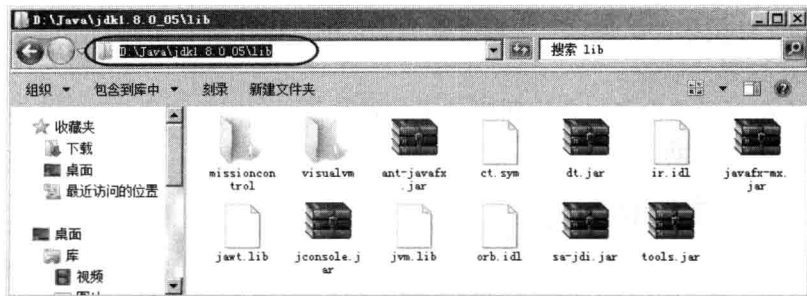


图 1.12 在地址栏中显示完整路径

2. 大小写问题

前面已经提到: Java 语言是严格区分大小写的语言。但由于大部分读者都是 Windows 操作系统的忠实拥护者, 因此对大小写问题往往都不够重视 (Linux 平台是区分大小写的)。

例如, 有的读者编写的 Java 程序里的类是 HelloWorld, 但当他运行 Java 程序时, 运行的则是 java helloworld 这种形式——这种错误的形式有很多种(对的道路只有一条, 但错误的道路则有成千上万条)。总之, 就是 java 命令后的类名没有严格按 Java 程序中编写的来写, 可能引起系统提示如图 1.9 所示的错误。

因此必须提醒读者注意: 在 Java 程序里, HelloWorld 和 helloworld 是完全不同的, 必须严格注意 Java 程序里的大小写问题。

不仅如此, 读者按书中所示的程序编写 Java 程序时, 必须严格注意 Java 程序中每个单词的大小写, 不要随意编写。例如 class 和 Class 是不同的两个词, class 是正确的, 但如果写成 Class, 则程序无法编译通过。实际上, Java 程序中的关键字全部是小写的, 无须大写任何字母。

3. 路径里包含空格的问题

这是一个更容易引起错误的问题。由于 Windows 系统的很多路径都包含了空格, 典型的例如 Program Files 文件夹, 而且这个文件夹是 JDK 的默认安装路径。

如果 CLASSPATH 环境变量里包含的路径中存在空格, 则可能引发错误。因此, 推荐大家安装 JDK 以及 Java 相关程序、工具时, 不要安装在包含空格的路径下, 否则可能引发错误。

4. main 方法的问题

如果需要用 java 命令直接运行一个 Java 类, 这个 Java 类必须包含 main 方法, 这个 main 方法必须使用 public 和 static 来修饰, 必须使用 void 声明该方法的返回值, 而且该方法的参数类型只能是一个字符串数组, 而不能是其他形式的参数。对于这个 main 方法而言, 前面的 public 和 static 修饰符的位置可以互换, 但其他部分则是固定的。

定义 main 方法时, 不要写成 Main 方法, 如果不小心把方法名的首字母写成了大写, 编译时不会出现任何问题, 但运行该程序时将给出如下错误提示:

错误: 在类 Xxx 中找不到 main 方法, 请将 main 方法定义为:
public static void main(String[] args)

这个错误提示找不到 main 方法, 因为 Java 虚拟机只会选择从 main 方法开始执行; 对于 Main 方法, Java 虚拟机会把该方法当成一个普通方法, 而不是程序的入口。

main 方法里可以放置程序员需要执行的可执行性语句, 例如 System.out.println("Hello Java!"), 这行语句是 Java 里的输出语句, 用于向控制台输出 "Hello Java!" 这个字符串内容, 输出结束后还输出一个换行符。

在 Java 程序里执行输出有两种简单的方式: System.out.print(需要输出的内容)和 System.out.println(需要输出的内容), 其中前者在输出结束后不会换行, 而后者在输出结束后会换行。后面会有关于这两个方法更详细的解释, 此处读者只能把这两个方法先记下来。



1.7 垃圾回收机制

传统的 C/C++ 等编程语言, 需要程序员负责回收已经分配的内存。显式进行垃圾回收是一件比较困难的事情, 因为程序员并不总是知道内存应该何时被释放。如果一些分配出去的内存得不到及时回收, 就会引起系统运行速度下降, 甚至导致程序瘫痪, 这种现象被称为内存泄漏。总体而言, 显式进行垃圾回收主要有如下两个缺点。

- 程序忘记及时回收无用内存, 从而导致内存泄漏, 降低系统性能。
- 程序错误地回收程序核心类库的内存, 从而导致系统崩溃。

与 C/C++ 程序不同, Java 语言不需要程序员直接控制内存回收, Java 程序的内存分配和回收都是由 JRE 在后台自动进行的。JRE 会负责回收那些不再使用的内存, 这种机制被称为垃圾回收 (Garbage Collection, GC)。通常 JRE 会提供一个后台线程来进行检测和控制, 一般都是在 CPU 空闲或内存不足时自动进行垃圾回收, 而程序员无法精确控制垃圾回收的时间和顺序等。

Java 的堆内存是一个运行时数据区, 用以保存类的实例 (对象), Java 虚拟机的堆内存中存储着正在运行的应用程序所建立的所有对象, 这些对象不需要程序通过代码来显式地释放。一般来说, 堆内存的回收由垃圾回收来负责, 所有的 JVM 实现都有一个由垃圾回收器管理的堆内存。垃圾回收是一种动态存储管理技术, 它自动释放不再被程序引用的对象, 按照特定的垃圾回收算法来实现内存资源的自动回收功能。

在 C/C++ 中, 对象所占用的内存不会被自动释放, 如果程序没有显式释放对象所占用的内存, 对象所占用的内存就不能分配给其他对象, 该内存存在程序结束运行之前将一直被占用; 而在 Java 中, 当没有引用变量指向原先分配给某个对象的内存时, 该内存便成为垃圾。JVM 的一个超级线程会自动释放该内存区。垃圾回收意味着程序不再需要的对象是“垃圾信息”, 这些信息将被丢弃。

当一个对象不再被引用时, 内存回收它占领的空间, 以便空间被后来的新对象使用。事实上, 除了释放没用的对象外, 垃圾回收也可以清除内存记录碎片。由于创建对象和垃圾回收器释放丢弃对象所占的内存空间, 内存会出现碎片。碎片是分配给对象的内存块之间的空闲内存区, 碎片整理将所占用的堆内存移到堆的一端, JVM 将整理出的内存分配给新的对象。

垃圾回收能自动释放内存空间, 减轻编程的负担。这使 Java 虚拟机具有两个显著的优点。

- 垃圾回收机制可以很好地提高编程效率。在没有垃圾回收机制时, 可能要花许多时间来解决一个难懂的存储器问题。在用 Java 语言编程时, 依靠垃圾回收机制可大大缩短时间。
- 垃圾回收机制保护程序的完整性, 垃圾回收是 Java 语言安全性策略的一个重要部分。

垃圾回收的一个潜在缺点是它的开销影响程序性能。Java 虚拟机必须跟踪程序中有用的对象, 才可以确定哪些对象是无用的对象, 并最终释放这些无用的对象。这个过程需要花费处理器的时间。其次是垃圾回收算法的不完备性, 早先采用的某些垃圾回收算法就不能保证 100% 收集到所有的废弃内存。当然, 随着垃圾回收算法的不断改进, 以及软硬件运行效率的不断提升, 这些问题都可以迎刃而解。

Java 语言规范没有明确地说明 JVM 使用哪种垃圾回收算法, 但是任何一种垃圾回收算法一般要做两件基本的事情: 发现无用的对象; 回收被无用对象占用的内存空间, 使该空间可被程序再次使用。

通常, 垃圾回收具有如下几个特点。

- 垃圾回收机制的工作目标是回收无用对象的内存空间, 这些内存空间都是 JVM 堆内存里的内存空间, 垃圾回收只能回收内存资源, 对其他物理资源, 如数据库连接、磁盘 I/O 等资源则无能为力。
- 为了更快地让垃圾回收机制回收那些不再使用的对象, 可以将该对象的引用变量设置为 null, 通过这种方式暗示垃圾回收机制可以回收该对象。

- 垃圾回收发生的不可预知性。由于不同 JVM 采用了不同的垃圾回收机制和不同的垃圾回收算法, 因此它有可能是定时发生的, 有可能是当 CPU 空闲时发生的, 也有可能和原始的垃圾回收一样, 等到内存消耗出现极限时发生, 这和垃圾回收实现机制的选择及具体的设置都有关系。虽然程序员可以通过调用 Runtime 对象的 gc() 或 System.gc() 等方法来建议系统进行垃圾回收, 但这种调用仅仅是建议, 依然不能精确控制垃圾回收机制的执行。
- 垃圾回收的精确性主要包括两个方面: 一是垃圾回收机制能够精确地标记活着的对象; 二是垃圾回收器能够精确地定位对象之间的引用关系。前者是完全回收所有废弃对象的前提, 否则就可能造成内存泄漏; 而后者则是实现归并和复制等算法的必要条件, 通过这种引用关系, 可以保证所有对象都能被可靠地回收, 所有对象都能被重新分配, 从而有效地减少内存碎片的产生。
- 现在的 JVM 有多种不同的垃圾回收实现, 每种回收机制因其算法差异可能表现各异, 有的当垃圾回收开始时就停止应用程序的运行, 有的当垃圾回收运行时允许应用程序的线程运行, 还有的在同一时间允许垃圾回收多线程运行。

当编写 Java 程序时, 一个基本原则是: 对于不再需要的对象, 不要引用它们。如果保持对这些对象的引用, 垃圾回收机制暂时不会回收该对象, 则会导致系统可用内存越来越少; 当系统可用内存越来越少时, 垃圾回收执行的频率就越来越高, 从而导致系统的性能下降。

2011 年 7 月发布的 Java 7 提供了 G1 垃圾回收器来代替原有的并行标记/清除垃圾回收器 (简称 CMS)。在未来的日子里, G1 垃圾回收器将会逐渐取代原有的 CMS 垃圾回收器。相信 G1 的性能、稳定性将会比 CMS 更胜一筹。

2014 年 3 月发布的 Java 8 删除了 HotSpot JVM 中的永生代内存 (PermGen, 永生代内存主要用于存储一些需要常驻内存、通常不会被回收的信息), 而是改为使用本地内存来存储类的元数据信息, 并将之称为: 元空间 (Metaspace), 这意味着以后不会再遇到 java.lang.OutOfMemoryError:PermGen 错误 (曾经令许多 Java 程序员头痛的错误)。

1.8 何时开始使用 IDE 工具

对于 Java 语言的初学者而言, 这里给出一个忠告: 不要使用任何 IDE 工具来学习 Java 编程, Windows 平台上可以选择记事本, Linux 平台上可以选择使用 VI 工具。如果嫌 Windows 上的记事本的颜色太单调, 可以选择使用 EditPlus 或者 UltraEdit。

在多年的程序开发生涯中, 常常见到一些所谓的 Java 程序员, 他们怀揣一本 Eclipse 从入门到精通, 只会单击几个“下一步”按钮就敢说自己精通 Java 了, 实际上他们连动手建一个 Web 应用都不会, 连 Java 的 Web 应用的文件结构都搞不清楚, 这也许不是他们的错, 可能他们习惯了在 Eclipse 或者 NetBeans 工具里通过单击鼠标来新建 Web 应用, 而从来不去看这些工具为我们做了什么。

曾经看到一个在某培训机构已经学习了 2 个月的学生, 连 extends 这个关键字都拼不出来, 不禁令人哑然, 这就是依赖 IDE 工具的后果。

还见过许多所谓的技术经理, 他们来应聘时往往滔滔不绝, 口若悬河。他们知道很多新名词、新概念, 但机试往往很不乐观: 说没有 IDE 工具, 提供了 IDE 工具后, 又说没文档, 提供了文档又说不能上网, 提供了上网又说不是在自己的电脑上, 没有代码参考……他们的理由比他们的技术强!

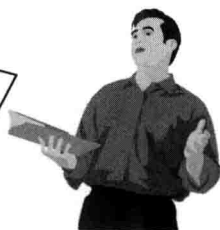
可能有读者会说, 程序员是不需要记那些简单语法的! 关于这一点也有一定的道理。但问题是: 没有一个人会在遇到 1+1=? 的问题时说, 我要查一下文档! 对于一个真正的程序员而言, 大部分代码就在手边, 还需要记忆?

当然, IDE 工具也有其优势, 在项目管理、团队开发方面都有不可比拟的优势。但并不是每个人都可以使用 IDE 工具的。



学生提问：我想学习 Java 编程，到底是学习 Eclipse 好，还是学习 NetBeans 好呢？

答：你学习的是 Java 语言，而不是任何工具。如果你一开始就从工具学起，可能导致你永远都不会 Java 语言。虽说“工欲善其事，必先利其器”，但这个前提是你已经会做这件事情了——如果你还不会做这件事情，那么再利的器对你都没有任何作用。再者，你现在知道的可能只有 Eclipse 和 NetBeans，实际上，Java 的 IDE 工具多如牛毛，除了 Eclipse 和 NetBeans 之外，还有 IBM 提供的 WSAD 和 VisualAge、Oracle 提供的 Jdeveloper 等等，每个 IDE 都各有特色，各有优势。如果从工具学起，势必造成对工具的依赖，当换用其他 IDE 工具时极为困难。如果你从 Java 语言本身学起，把 Java 语言本身的相关方面掌握到熟练，那么使用任何 IDE 工具都会得心应手。



那么何时开始使用 IDE 工具呢？标准是：如果你还离不开这个 IDE 工具，那么你就不能使用这个 IDE 工具；只有当你十分清楚在 IDE 工具里单击每一个菜单，单击每一个按钮……IDE 工具在底层为你做的每个细节时，才可以使用 IDE 工具！

如果读者有志于成为一名优秀的 Java 程序员，那么，到了更高层次后，就不可避免地需要自己开发 IDE 工具的插件（例如开发 Eclipse 插件），定制自己的 IDE 工具，甚至负责开发整个团队的开发平台，这些都要求开发者对 Java 开发的细节非常熟悉。因此，不要从 IDE 工具开始学习。

1.9 本章小结

本章简单介绍了 Java 语言的发展历史，并详细介绍了 Java 语言的编译、解释运行机制，也大致讲解了 Java 语言的垃圾回收机制。除此之外，还大致介绍了 Java 语言的几个竞争对手的情况。本章的重点是讲解如何搭建 Java 开发环境，包括安装 JDK，设置 PATH 环境变量，并详细阐述了 CLASSPATH 环境变量的作用，并向读者指出应该如何处理 CLASSPATH 环境变量。本章还详细介绍了如何开发和运行第一个 Java 程序，并总结出了初学者容易出现的几个错误。最后针对 Java 学习者是否应该使用 IDE 工具给出了一些过来人的建议。

本章练习

1. 搭建自己的 Java 开发环境。
2. 编写 Java 语言的 HelloWorld。

第2章 理解面向对象

本章要点

- 结构化程序设计
- 顺序结构
- 分支结构
- 循环结构
- 面向对象程序设计
- 继承、封装、多态
- UML 简介
- 掌握常用的 UML 图形
- 理解 Java 的面向对象特征

Java 语言是纯粹的面向对象的程序设计语言，这主要表现为 Java 完全支持面向对象的三种基本特征：继承、封装和多态。Java 语言完全以对象为中心，Java 程序的最小程序单位是类，整个 Java 程序由一个一个的类组成。

Java 完全支持使用对象、类、继承、封装、消息等基本概念来进行程序设计，允许从现实世界中客观存在的事物（即对象）出发来构造软件系统，在系统构造中尽可能运用人类的自然思维方式。实际上，这些优势是所有面向对象编程语言共同特征。面向对象的方式实际上由 OOA（面向对象分析）、OOD（面向对象设计）和 OOP（面向对象编程）三个部分有机组成，其中，OOA 和 OOD 的结构需要使用一种方式来描述并记录，目前业界统一采用 UML（统一建模语言）来描述并记录 OOA 和 OOD 的结果。

目前 UML 的最新版本是 2.0，它一共包括 13 种类型的图形，使用这 13 种图形中的某些就可以很好地描述并记录软件分析、设计的结果。通常而言，没有必要为软件系统绘制 13 种 UML 图形，常用的 UML 图形有用例图、类图、组件图、部署图、顺序图、活动图和状态机图。本章将会介绍 UML 图的相关概念，也会详细介绍这 7 种常用的 UML 图的绘制方法。

2.1 面向对象

在目前的软件开发领域有两种主流的开发方法：结构化开发方法和面向对象开发方法。早期的编程语言如 C、Basic、Pascal 等都是结构化编程语言；随着软件开发技术的逐渐发展，人们发现面向对象可以提供更好的可重用性、可扩展性和可维护性，于是催生了大量的面向对象的编程语言，如 C++、Java、C# 和 Ruby 等。

➤➤ 2.1.1 结构化程序设计简介

结构化程序设计方法主张按功能来分析系统需求，其主要原则可概括为自顶向下、逐步求精、模块化等。结构化程序设计首先采用结构化分析（Structured Analysis, SA）方法对系统进行需求分析，然后使用结构化设计（Structured Design, SD）方法对系统进行概要设计、详细设计，最后采用结构化编程（Structured Program, SP）方法来实现系统。使用这种 SA、SD 和 SP 的方式可以较好地保证软件系统的开发进度和质量。

因为结构化程序设计方法主张按功能把软件系统逐步细分，因此这种方法也被称为面向功能的程序设计方法；结构化程序设计的每个功能都负责对数据进行一次处理，每个功能都接受一些数据，处理完后输出一些数据，这种处理方式也被称为面向数据流的处理方式。

结构化程序设计里最小的程序单元是函数，每个函数都负责完成一个功能，用以接收一些输入数据，函数对这些输入数据进行处理，处理结束后输出一些数据。整个软件系统由一个个函数组成，其中作为程序入口的函数被称为主函数，主函数依次调用其他普通函数，普通函数之间依次调用，从而完成整个软件系统的功能。图 2.1 显示了结构化软件的逻辑结构示意图。

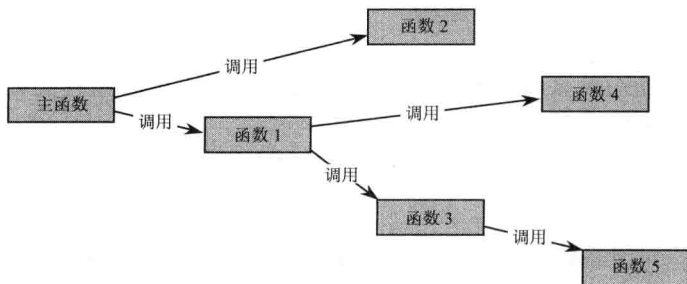


图 2.1 结构化软件的逻辑结构示意图

从图 2.1 可以看出，结构化设计需要采用自顶向下的设计方式，在设计阶段就需要考虑每个模块应该分解成哪些子模块，每个子模块又分解成哪些更小的模块……依此类推，直至将模块细化成一个个函数。

每个函数都是具有输入、输出的子系统，函数的输入数据包括函数形参、全局变量和常量等，函数的输出数据包括函数返回值以及传出参数等。结构化程序设计方式有如下两个局限性。

- 设计不够直观，与人类习惯思维不一致。采用结构化程序分析、设计时，开发者需要将客观世界模型分解成一个个功能，每个功能用以完成一定的数据处理。
- 适应性差，可扩展性不强。由于结构化设计采用自顶向下的设计方式，所以当用户的需求发生改变，或需要修改现有的实现方式时，都需要自顶向下地修改模块结构，这种方式的维护成本相当高。



提示：

采用结构化方式设计的软件系统，整个软件系统就由一个个函数组成，这个软件的运行入口往往由一个“主函数”代表，而主函数负责把系统中的所有函数“串起来”。

2.1.2 程序的三种基本结构

在过去的日子里，很多编程语言都提供了 GOTO 语句，GOTO 语句非常灵活，可以让程序的控制流程任意流转——如果大量使用 GOTO 语句，程序完全不需要使用循环。但 GOTO 语句实在太随意了，如果程序随意使用 GOTO 语句，将会导致程序流程难以理解，并且容易出错。在实际软件开发过程中，更注重软件的可读性和可修改性，因此 GOTO 语句逐渐被抛弃了。



提示：

Java 语言拒绝使用 GOTO 语句，但它将 goto 作为保留字，意思是目前 Java 版本还未使用 GOTO 语句，但也许在未来的日子里，当 Java 不得不使用 GOTO 语句时，Java 还是可能使用 GOTO 语句的。

结构化程序设计非常强调实现某个功能的算法，而算法的实现过程是由一系列操作组成的，这些操作之间的执行次序就是程序的控制结构。1996 年，计算机科学家 Bohm 和 Jacopini 证明了这样的事实：任何简单或复杂的算法都可以由顺序结构、选择结构和循环结构这三种基本结构组合而成。所以，这三种结构就被称为程序设计的三种基本结构，也是结构化程序设计必须采用的结构。

1. 顺序结构

顺序结构表示程序中的各操作是按照它们在源代码中的排列顺序依次执行的，其流程如图 2.2 所示。

图中的 S1 和 S2 表示两个处理步骤，这些处理步骤可以是一个非转移操作或多个非转移操作，甚至可以是空操作，也可以是三种基本结构中的任一结构。整个顺序结构只有一个入口点 a 和一个出口点 b。这种结构的特点是：程序从入口点 a 处开始，按顺序执行所有操作，直到出口点 b 处，所以称为顺序结构。

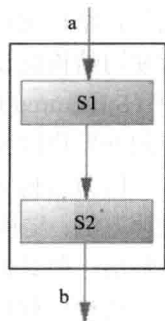


图 2.2 顺序结构



提示：

虽然 Java 是面向对象的编程语言，但 Java 的方法类似于结构化程序设计的函数，因此方法中代码的执行也是顺序结构。

2. 选择结构

选择结构表示程序的处理需要根据某个特定的条件选择其中的一个分支执行。选择结构有单选择、双选择和多选择三种形式。

双选择是典型的选择结构形式，其流程如图 2.3 所示，图中的 S1 和 S2 与顺序结构中的说明相同。由图中可见，在结构的入口点 a 处有一个判断条件，表示程序流程出现了两个可供选择的分支，如果判断条件为真则执行 S1 处理，否则执行 S2 处理。值得注意的是，这两个分支中只能选择一个且必须选

择一个执行，但不论选择了哪一个分支执行，最后流程都一定到达结构的出口点 b 处。

当 S1 和 S2 中的任意一个处理为空时，说明结构中只有一个可供选择的分支，如果判断条件为真，则执行 S1 处理，否则直接执行到结构出口点 b 处。也就是说，如果判断条件为假时，则什么也没执行，所以称为单选择结构，如图 2.4 所示。

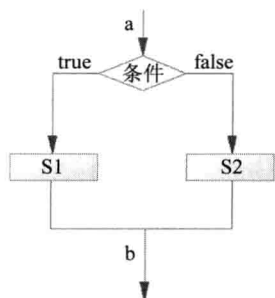


图 2.3 双选择结构

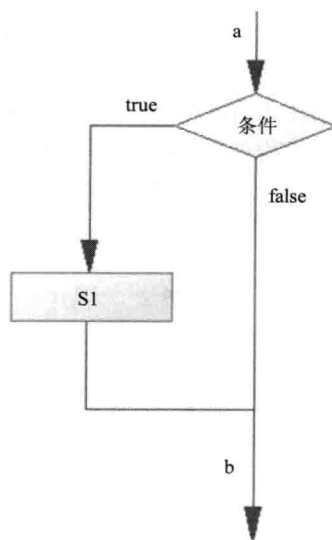


图 2.4 单选择结构

多选择结构是指程序流程中遇到如图 2.5 所示的 S1、S2、S3、S4…多个分支，程序执行方向根据判断条件来确定。如果条件 1 为真，则执行 S1 处理；如果条件 1 为假，条件 2 为真，则执行 S2 处理；如果条件 1 为假，条件 2 为假，条件 3 为真，则执行 S3 处理……依此类推。从图 2.5 中可以看出， S_n 处理的 n 值越大，则需要满足的条件越苛刻。

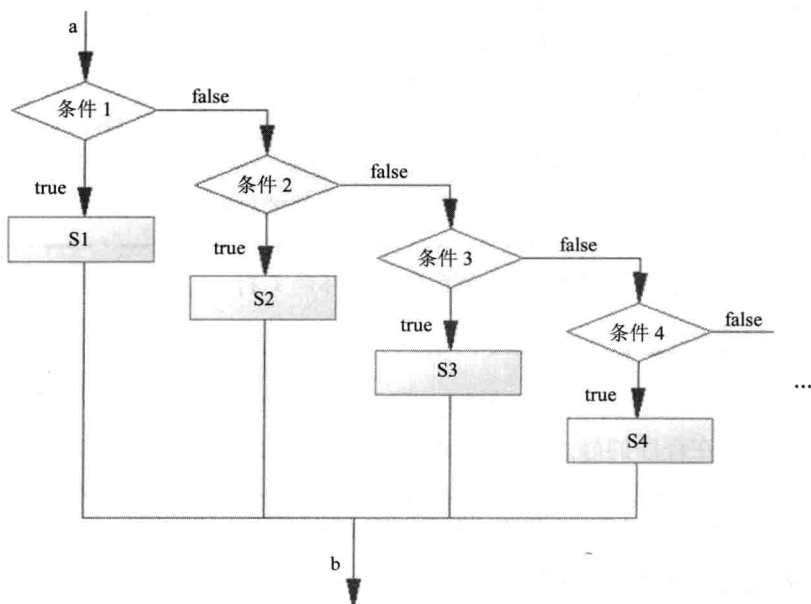


图 2.5 多选择结构

对于图 2.5 所示的多选择结构，不论选择了哪一个分支，最后流程都要到达同一个出口点 b 处。如果所有分支的条件都不满足，则直接到达出口点 b 处。有些程序语言不支持多选择结构，但所有的结构化程序设计语言都是支持的。



提示:

Java 语言对此处介绍的三种选择结构都有很好的支持, 本书的第 4 章将介绍 Java 的这三种选择结构。

3. 循环结构

循环结构表示程序反复执行某个或某些操作, 直到某条件为假 (或为真) 时才停止循环。在循环结构中最主要的是: 在什么情况下执行循环? 哪些操作需要重复执行? 循环结构的基本形式有两种: 当型循环和直到型循环, 其流程如图 2.6 所示。图中带 S 标识的矩形内的操作称为循环体, 即循环入口点 a 到循环出口点 b 之间的处理步骤, 这就是需要循环执行的部分。而在什么情况下执行循环则要根据条件判断。

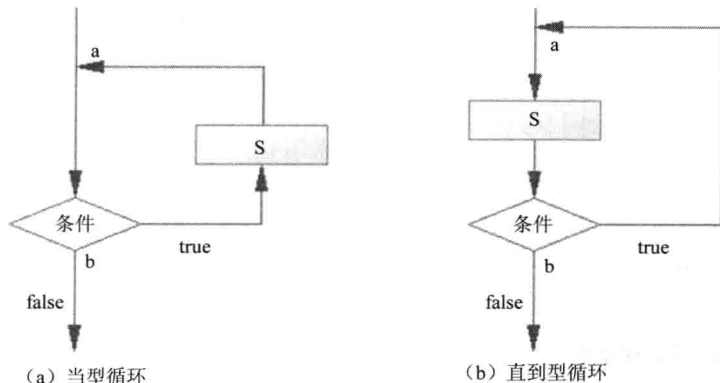


图 2.6 循环结构

- **当型结构:** 先判断条件, 当条件为真时执行循环体, 并且在循环体结束时自动返回到循环入口处, 再次判断循环条件; 如果条件为假, 则退出循环体到达流程出口处。因为是“当条件为真时执行循环”, 即先判断后执行, 所以被称为当型循环。其流程如图 2.6 (a) 所示。
- **直到型循环:** 从入口处直接执行循环体, 循环体结束时判断条件, 如果条件为真, 则返回入口处继续执行循环体, 直到条件为假时退出循环体到达流程出口处, 是先执行后判断。因为是“直到条件为假时结束循环”, 所以被称为直到型循环。其流程如图 2.6 (b) 所示。

同样, 循环结构也只有一个入口点 a 和一个出口点 b, 循环终止是指流程执行到循环的出口点。图 2.6 中所表示的 S 处理可以是一个或多个操作, 也可以是一个完整的结构或过程。



提示:

Java 语言同样提供了对当型循环和直到型循环的支持, 本书第 4 章在介绍循环时将会深入介绍这些内容。

通过三种基本控制结构可以看到, 结构化程序设计中的任何结构都具有唯一的入口和唯一的出口, 并且程序不会出现死循环。在程序的静态形式与动态执行流程之间具有良好的对应关系。本书之所以详细介绍这些程序结构, 主要因为 Java 语言的方法体内同样是由这三种程序结构组成的, 换句话说, 虽然 Java 是面向对象的, 但 Java 的方法里则是一种结构化的程序流。

2.1.3 面向对象程序设计简介

面向对象是一种更优秀的程序设计方法, 它的基本思想是使用类、对象、继承、封装、消息等基本概念进行程序设计。它从现实世界中客观存在的事物 (即对象) 出发来构造软件系统, 并在系统构造中尽可能运用人类的自然思维方式, 强调直接以现实世界中的事物 (即对象) 为中心来思考, 认识问题, 并根据这些事物的本质特点, 把它们抽象地表示为系统中的类, 作为系统的基本构成单元 (而不是用一些与现实世界中的事物相关比较远, 并且没有对应关系的过程来构造系统), 这使得软件系统的组件可以直接映像到客观世界, 并保持客观世界中事物及其相互关系的本来面貌。

采用面向对象方式开发的软件系统，其最小的程序单元是类，这些类可以生成系统中的多个对象，而这些对象则直接映像成客观世界的各种事物。采用面向对象方式开发的软件系统逻辑上的组成结构如图 2.7 所示。

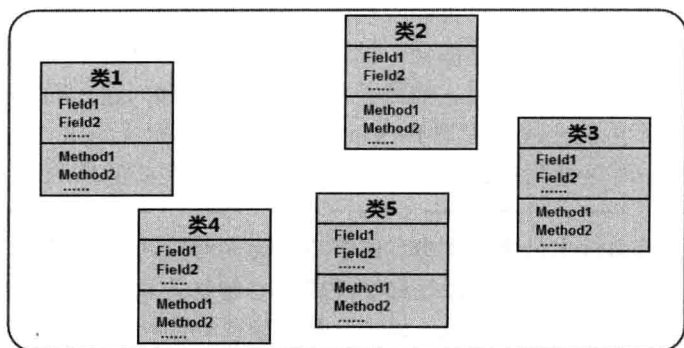


图 2.7 面向对象方式开发的软件系统逻辑上的组成结构

从图 2.7 可以看出，面向对象的软件系统由多个类组成，类代表了客观世界中具有某种特征的一类事物，这类事物往往有一些内部的状态数据，比如人有身高、体重、年龄、爱好等各种状态数据——当然程序没必要记录该事物所有的状态数据，程序只要记录业务关心的状态数据即可。

面向对象的语言不仅使用类来封装一类事物的内部状态数据，这种状态数据就对应于图 2.7 中的成员变量（由 Field 翻译而来，有些资料将其直译为字段；还有些资料将其翻译为属性，但这个说法非常不准确，Java 的属性指的是 Property）；而且类会提供操作这些状态数据的方法，还会为这类事物的行为特征提供相应的实现，这种实现也是方法。因此可以得到如下基本等式：

成员变量（状态数据）+ 方法（行为）= 类定义

从这个等式来看，面向对象比面向过程的编程粒度要大：面向对象的程序单位是类；而面向过程的程序单位是函数（相当于方法），因此面向对象比面向过程更简单、易用。



提示：

假设需要组装一台电脑，如果拿到手的是主板、CPU、内存条、硬盘等这种大粒度的组件，随便找个人就可以把它们组装成一台电脑；但如果拿到手的是一些二极管、三极管、集成电路等小粒度的组件，要想把它们组装成电脑，恐怕没那么容易。如果把数据以及操作数据的方法都封装成对象，这就相当于提供了大粒度的组件，因此编程更容易。

从面向对象的眼光来看，开发者希望从自然的认识、使用角度来定义和使用类。也就是说，开发者希望直接对客观世界进行模拟：定义一个类，对应客观世界的哪种事物；业务需要关心这个事物的哪些状态，程序就为这些状态定义成员变量；业务需要关心这个事物的哪些行为，程序就为这些行为定义方法。

不仅如此，面向对象程序设计与人类习惯的思维方法有较好的一致性，比如希望完成“猪八戒吃西瓜”这样一件事情。

在面向过程的程序世界里，一切以函数为中心，函数最大，因此这件事情会用如下语句来表达：

```
吃(猪八戒, 西瓜);
```

在面向对象的程序世界里，一切以对象为中心，对象最大，因此这件事情会用如下语句来表达：

```
猪八戒.吃(西瓜);
```

对比两条语句不难发现，面向对象的语句更接近自然语言的语法：主语、谓语、宾语一目了然，十分直观，因此程序员更易理解。

2.1.4 面向对象的基本特征

面向对象方法具有三个基本特征：封装（Encapsulation）、继承（Inheritance）和多态（Polymorphism），

其中封装指的是将对象的实现细节隐藏起来, 然后通过一些公用方法来暴露该对象的功能; 继承是面向对象实现软件复用的重要手段, 当子类继承父类后, 子类作为一种特殊的父类, 将直接获得父类的属性和方法; 多态指的是子类对象可以直接赋给父类变量, 但运行时依然表现出子类的行为特征, 这意味着同一个类型的对象在执行同一个方法时, 可能表现出多种行为特征。

除此之外, 抽象也是面向对象的重要部分, 抽象就是忽略一个主题中与当前目标无关的那些方面, 以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题, 而只是考虑部分问题。例如, 需要考察 Person 对象时, 不可能在程序中把 Person 的所有细节都定义出来, 通常只能定义 Person 的部分数据、部分行为特征——而这些数据、行为特征是软件系统所关心的部分。



提示:

虽然抽象是面向对象的重要部分, 但它不是面向对象的特征之一, 因为所有的编程语言都需要抽象。当开发者进行抽象时应该考虑哪些特征是软件系统所需要的, 那么这些特征就应该使用程序记录并表现出来。因此, 需要抽象哪些特征没有必然的规定, 而是取决于软件系统的功能需求。

面向对象还支持如下几个功能。

- 对象是面向对象方法中最基本的概念, 它的基本特点有: 标识唯一性、分类性、多态性、封装性、模块独立性好。
- 类是具有共同属性、共同方法的一类事物。类是对象的抽象; 对象则是类的实例。而类是整个软件系统最小的程序单元, 类的封装性将各种信息细节隐藏起来, 并通过公用方法来暴露该类对外所提供的功能, 从而提高了类的内聚性, 降低了对象之间的耦合性。
- 对象间的这种相互合作需要一个机制协助进行, 这样的机制称为“消息”。消息是一个实例与另一个实例之间相互通信的机制。
- 在面向对象方法中, 类之间共享属性和操作的机制称为继承。继承具有传递性。继承可分为单继承 (一个继承只允许有一个直接父类, 即类等级为树形结构) 与多继承 (一个类允许有多个直接父类)。



注意:

由于多继承可能引起继承结构的混乱, 而且会大大降低程序的可理解性, 所以 Java 不支持多继承。



在编程语言领域, 还有一个“基于对象”的概念, 这两个概念极易混淆。通常而言, “基于对象”也使用了对象, 但是无法利用现有的对象模板产生新的对象类型, 继而产生新的对象, 也就是说, “基于对象”没有继承的特点; 而“多态”则更需要继承, 没有了继承的概念也就无从谈论“多态”。面向对象方法的三大基本特征 (封装、继承、多态) 缺一不可。例如, JavaScript 语言就是基于对象的, 它使用一些封装好的对象, 调用对象的方法, 设置对象的属性; 但是它们无法让开发者派生新的类, 开发者只能使用现有对象的方法和属性。

判断一门语言是否是面向对象的, 通常可以使用继承和多态来加以判断。“面向对象”和“基于对象”都实现了“封装”的概念, 但是面向对象实现了“继承和多态”, 而“基于对象”没有实现这些。

面向对象编程的程序员按照分工分为“类库的创建者”和“类库的使用者”。使用类库的人并不都是具备了面向对象思想的人, 通常知道如何继承和派生新对象就可以使用类库了, 然而他们的思维并没有真正地转过来, 使用类库只是在形式上是面向对象的, 而实质上只是库函数的一种扩展。

2.2 UML (统一建模语言) 介绍

面向对象软件开发需要经过 OOA (面向对象分析)、OOD (面向对象设计) 和 OOP (面向对象编

程)三个阶段, OOA 对目标系统进行分析, 建立分析模型, 并将之文档化; OOD 用面向对象的思想对 OOA 的结果进行细化, 得出设计模型。OOA 和 OOD 的分析、设计结果需要统一的符号来描述、交流并记录, UML 就是这种用于描述、记录 OOA 和 OOD 结果的符号表示法。

面向对象的分析与设计方法在 20 世纪 80 年代末至 90 年代中出现了一个高潮, UML 是这个高潮的产物。在此期间出现了三种具有代表性的表示方法。

Booch 是面向对象方法最早的倡导者之一, 他提出了面向对象软件工程的概念。Booch 1993 表示法(由 Booch 提出)比较适合于系统的设计和构造。

Rumbaugh 等人提出了面向对象的建模技术(OMT)方法, 采用面向对象的概念, 并引入了各种独立于语言的表示符。这种方法用对象模型、动态模型、功能模型和用例模型共同完成对整个系统的建模, 所定义的概念和符号可用于软件开发的分析、设计和实现的全过程, 软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2 特别适用于分析和描述以数据为中心的信息系统。

Jacobson 于 1994 年提出了 OOSE 方法, 其最大特点是面向用例(Use-Case), 并在用例的描述中引入了外部角色的概念。用例的概念是精确描述需求的重要武器, 但用例贯穿于整个开发过程, 包括对系统的测试和验证。OOSE 比较适合支持商业工程和需求分析。

UML 统一了 Booch、Rumbaugh 和 Jacobson 的表示方法, 而且对其进行了进一步的发展, 并最终统一为大众所接受的标准建模语言。UML 是一种定义良好、易于表达、功能强大且普遍适用的建模语言, 它的作用域不限于支持面向对象的分析与设计, 还支持从需求分析开始的软件开发全过程。

截至 1996 年 10 月, UML 获得了工业界、科技界和应用界的广泛支持, 已有 700 多家公司表示支持采用 UML 作为建模语言。1996 年年底, UML 已稳占面向对象技术市场的 85%, 成为可视化建模语言事实上的工业标准。1997 年年底, OMG 组织(Object Management Group, 对象管理组织)采纳 UML 1.1 作为基于面向对象技术的标准建模语言。UML 代表了面向对象方法的软件开发技术的发展方向, 目前 UML 的最新版本是 2.0, UML 的大致发展过程如图 2.8 所示。

图 2.8 中的 UML 1.1 和 UML 2.0 是 UML 历史上两个具有里程碑意义的版本, 其中, UML 1.1 是 OMG 正式发布的第一个标准版本, 而 UML 2.0 是目前最成熟、稳定的 UML 版本。

UML 图大致上可分为静态图和动态图两种, UML 2.0 的组成如图 2.9 所示。

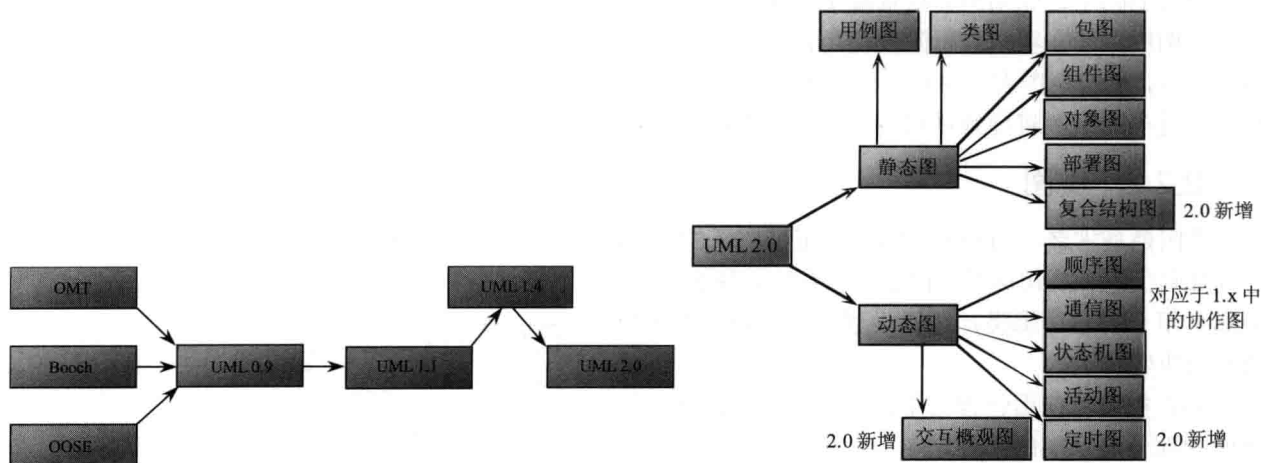


图 2.8 UML 的发展历史

图 2.9 UML 2.0 的组成

从图 2.9 可以看出, UML 2.0 一共包括 13 种正式图形: 活动图(activity diagram)、类图(class diagram)、通信图(communication diagram, 对应于 UML 1.x 中的协作图)、组件图(component diagram)、复合结构图(composite structure diagram, UML 2.0 新增)、部署图(deployment diagram)、交互概观图(interactive overview diagram, UML 2.0 新增)、对象图(object diagram)、包图(package diagram)、顺序图(sequence diagram)、状态机图(state machine diagram)、定时图(timing diagram, UML 2.0 新增)、用例图(use case diagram)。

当读者看到这 13 种 UML 图形时，可能会对 UML 产生恐惧的感觉，实际上正如大家所想：很少有一个软件系统在分析、设计阶段对每个细节都使用 13 种图形来表现。永远记住一点：不要把 UML 表示法当成一种负担，而应该把它当成一种工具，一种用于描述、记录软件分析设计的工具。最常用的 UML 图包括用例图、类图、组件图、部署图、顺序图、活动图和状态机图等。

2.2.1 用例图

用例图用于描述系统提供的系列功能，而每个用例则代表系统的一个功能模块。用例图的主要目的是帮助开发团队以一种可视化的方式理解系统的需求功能，用例图对系统的实现不做任何说明，仅仅是系统功能的描述。

用例图包括用例（以一个椭圆表示，用例的名称放在椭圆的中心或椭圆下面）、角色（Actor，也就是与系统交互的其他实体，以一个人形符号表示）、角色和用例之间的关系（以简单的线段来表示），以及系统内用例之间的关系。用例图一般表示出用例的组织关系——要么是整个系统的全部用例，要么是完成具体功能的一组用例。图 2.10 是一个简单的 BBS 系统的部分用例示意图。

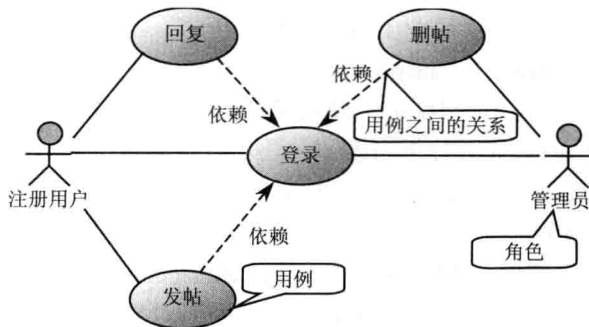


图 2.10 用例图

用例图通常用于表达系统或者系统范畴的高级功能。如图 2.10 所示，可以很容易看出该系统所提供的功能。这个系统允许注册用户登录、发帖和回复，其中发帖和回复需要依赖于登录；允许管理员删除其他人的帖子，删帖也需要依赖于登录。

用例图主要在需求分析阶段使用，主要用于描述系统实现的功能，方便与客户交流，保证系统需求的无二性，用实例图表示系统外观，不要指望用例图和系统的各个类之间有任何联系。不要把用例做得过多，过多的用例将导致难以阅读，难以理解；尽可能多地使用文字说明。

2.2.2 类图

类图是最古老、功能最丰富、使用最广泛的 UML 图。类图表示系统中应该包含哪些实体，各实体之间如何关联；换句话说，它显示了系统的静态结构，类图可用于表示逻辑类，逻辑类通常就是业务人员所谈及的事物种类。

类在类图上使用包含三个部分的矩形来描述，最上面的部分显示类的名称，中间部分包含类的属性，最下面的部分包含类的方法。图 2.11 显示了类图中类的表示方法。

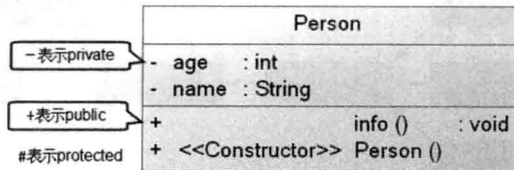


图 2.11 类图中类的表示方法

类图除了可以表示实体的静态内部结构之外，还可以表示实体之间的相互关系。类之间有三种基本关系：

- 关联（包括聚合、组合）
- 泛化（与继承同一个概念）
- 依赖

1. 关联

客观世界中的两个实体之间总是存在千丝万缕的关系，当把这两个实体抽象到软件系统中时，两个类之间必然存在关联关系。关联具有一定的方向性：如果仅能从一个类单方向地访问另一个类，则被称为单向关联；如果两个类可以互相访问对象，则被称为双向关联。一个对象能访问关联对象的数目被称为多重性，例如，建立学生和老师之间的单向关联，则可以从学生访问老师，但从老师不能访问学生。关联使用一条实线来表示，带箭头的实线表示单向关联。

在很多时候，关联和属性很像，关联和属性的关键区别在于：类里的某个属性引用到另外一个实体时，则变成了关联。

关联关系包括两种特例：聚合和组合，它们都有部分和整体的关系，但通常认为组合比聚合更加严格。当某个实体聚合成另一个实体时，该实体还可以同时是另一个实体的部分，例如，学生既可以是篮球俱乐部的成员，也可以是书法俱乐部的成员；当某个实体组合成另一个实体时，该实体则不能同时是一个实体的部分。聚合使用带空心菱形框的实线表示，组合则使用带实心菱形框的实线表示。图 2.12 显示了几个类之间的关联关系。



图 2.12 中的 Student、Teacher 等类都没有表现其属性、方法等特性，因为本图的重点在于表现类之间的关系。实际的类图中可能会为 Student、Teacher 每个类都添加属性、方法等细节。

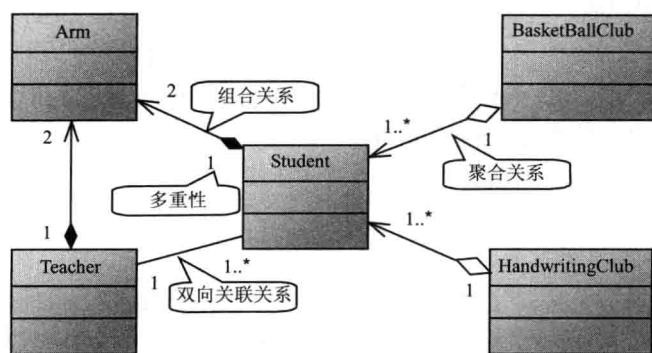


图 2.12 类之间的关联关系

图 2.12 中描述 Teacher 和 Student 之间的关联关系：它们是双向关联关系，而且使用了多重性来表示 Teacher 和 Student 之间存在 1 : N 的关联关系（1..* 表示可以是一个到多个），即一个 Teacher 实体可以有 1 个或多个关联的 Student 实体；Student 和 BasketballClub 存在聚合关系，即 1 个或多个 Student 实体可以聚合成一个 BasketballClub 实体；而 Arm（手臂）和 Student 之间存在组合关系，2 个 Arm 实体组合成一个 Student 实体。

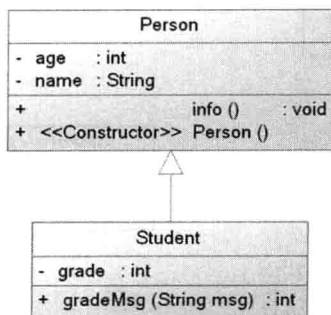


图 2.13 类之间的继承关系

2. 泛化

泛化与继承是同一个概念，都是指子类是一种特殊的父类，类与类之间的继承关系是非常普遍的，继承关系使用带空心三角形的实线表示。图 2.13 显示了 Student 和 Person 类之间的继承关系。

从图 2.13 可以看出，Student 是 Person 的子类，即 Student 类是一种特殊的 Person 类。



提示：

还有一种与继承类似的关系，类实现接口可视为一种特殊的继承，这种实现用带空心三角形的虚线表示。

3. 依赖

如果一个类的改动会导致另一个类的改动,则称两个类之间存在依赖。依赖关系使用带箭头的虚线表示,其中箭头指向被依赖的实体。依赖的常见可能原因如下:

- 改动的类将消息发给另一个类。
- 改动的类以另一个类作为数据部分。
- 改动的类以另一个类作为操作参数。

通常而言,依赖是单向的,尤其是当数据表现和数据模型分开设计时,数据表现依赖于数据模型。例如, JDK 基础类库中的 `JTable` 和 `DefaultTableModel`,关于这两个类的介绍请参考本书 12.11 节的介绍。图 2.14 显示了它们之间的依赖关系。

对于图 2.14 中表述的 `JTable` 和 `DefaultTableModel` 两个类,其中 `DefaultTableModel` 是 `JTable` 的数据模型,当 `DefaultTableModel` 发生改变时, `JTable` 将相应地发生改变。

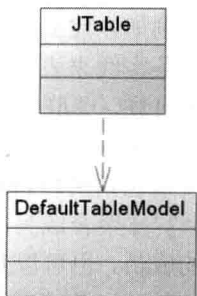


图 2.14 `JTable` 和 `DefaultTableModel` 之间的依赖关系

2.2.3 组件图

对于现代的大型应用程序而言,通常不只是单独一个类或单独一组类所能完成的,通常会由一个或多个可部署的组件组成。对 Java 程序而言,可复用的组件通常打包成一个 JAR、WAR 等文件;对 C/C++ 应用而言,可复用的组件通常是一个函数库,或者是一个 DLL (动态链接库) 文件。

组件图提供系统的物理视图,它的用途是显示系统中的软件对其他软件组件(例如,库函数)的依赖关系。组件图可以在一个非常高的层次上显示,仅显示系统中粗粒度的组件,也可以在组件包层次上显示。

组件图通常包含组件、接口和 Port 等图元, UML 使用带  符号的矩形来表示组件,使用圆圈代表接口,使用位于组件边界上的小矩形代表 Port。

组件的接口表示它能对外提供的服务规范,这个接口通常有两种表现形式。

- 用一条实线连接到组件边界的圆圈表示。
- 使用位于组件内部的圆圈表示。

组件除了可以对外提供服务接口之外,组件还可能依赖于某个接口,组件依赖于某个接口使用一条带半圆的实线来表示。图 2.15 显示了组件的接口和组件依赖的接口。

图 2.15 显示了一个简单的 `Order` 组件,该组件对外提供一个 `Payable` 接口,该组件需要依赖于一个 `CustomerLookup` 接口——通常这个 `CustomerLookup` 接口也是系统中已有的接口。

图 2.16 显示了包含组件关系的组件图。

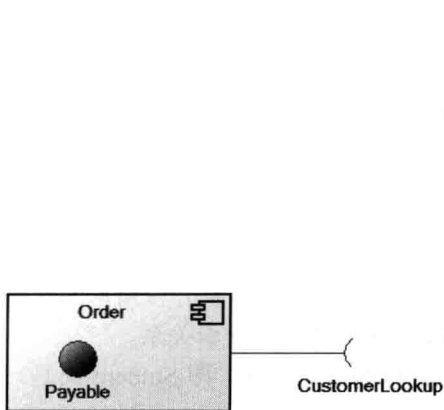


图 2.15 组件与接口

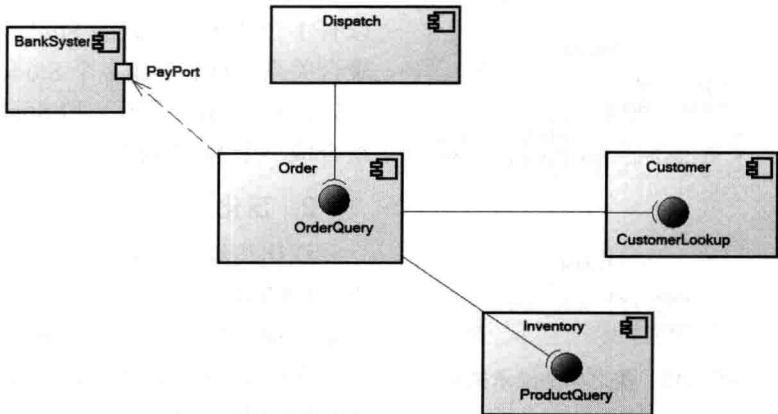


图 2.16 组件图

从图 2.16 可以看出,本系统绘制电子购物平台的几个核心组件,其中 `Order` 组件提供 `OrderQuery` 接口,该接口允许 `Dispatch` 组件查询系统中的订单及其状态, `Order` 组件又需要依赖于 `Customer` 组件的

CustomerLookup 接口, 通过该接口查询系统中的顾客信息; Order 组件也需要依赖于 Inventory 组件的 ProductQuery 接口, 通过该接口查询系统中的产品信息。

2.2.4 部署图

现代的软件工程早已超出早期的单机程序, 整个软件系统可能是跨国家、跨地区的分布式软件, 软件的不同部分可能需要部署在不同地方、不同平台之上。部署图用于描述软件系统如何部署到硬件环境中, 它的用途是显示软件系统不同的组件将在何处物理运行, 以及它们将如何彼此通信。

因为部署图是对物理运行情况进行建模, 所以系统的生产人员就可以很好地利用这种图来安装、部署软件系统。

部署图中的符号包括组件图中所使用的符号元素, 另外还增加了节点的概念: 节点是各种计算资源的通用名称, 主要包括处理器和设备两种类型, 两者的区别是处理器能够执行程序硬件构件(如计算机主机), 而设备是一种不具备计算能力的硬件构件(如打印机)。UML 中使用三维立方体来表示节点, 节点的名称位于立方体的顶部。图 2.17 显示了一个简单的部署图。

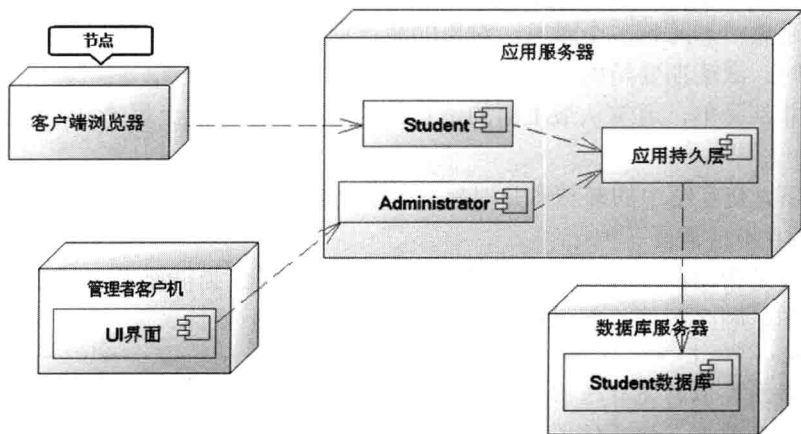


图 2.17 部署图

从图 2.17 可以看出, 整个应用分为 5 个组件: Student、Administrator、应用持久层、Student 数据库和 UI 界面组件, 部署图准确地表现了各组件之间的依赖关系。除此之外, 部署图的重点在物理节点上, 图 2.17 反映该应用需要部署在 4 个物理节点上, 其中普通客户端无须部署任何组件, 直接使用客户端浏览器即可; 管理者客户机上需要部署 UI 界面; 应用服务器上需要部署 Student、Administrator 和应用持久层三个组件; 而数据库服务器上需要部署 Student 数据库。

2.2.5 顺序图

顺序图显示具体用例(或者用例的一部分)的详细流程, 并且显示流程中不同对象之间的调用关系, 同时还可以很详细地显示对不同对象的不同调用。顺序图描述了对对象之间的交互(顺序图和通信图都被称为交互图), 重点在于描述消息及其时间顺序。

顺序图有两个维度: 垂直维度, 以发生的时间顺序显示消息/调用的序列; 水平维度, 显示消息被发送到的对象实例。顺序图的关键在于对象之间的消息, 对象之间的信息传递就是所谓的消息发送, 消息通常表现为对象调用另一个对象的方法或方法的返回值, 发送者和接收者之间的箭头表示消息。

顺序图的绘制非常简单。顺序图的顶部每个框表示每个类的实例(对象), 框中的类实例名称和类名称之间用冒号或空格来分隔, 例如 myReportGenerator : ReportGenerator。如果某个类实例向另一个类实例发送一条消息, 则绘制一条指向接收类实例的带箭头的连线, 并把消息/方法的名称放在连线上。

对于某些特别重要的消息, 还可以绘制一条带箭头的指向发起类实例的虚线, 将返回值标注在虚线上, 绘制带返回值的信息可以使得序列图更易于阅读。图 2.18 显示了用户登录顺序图。

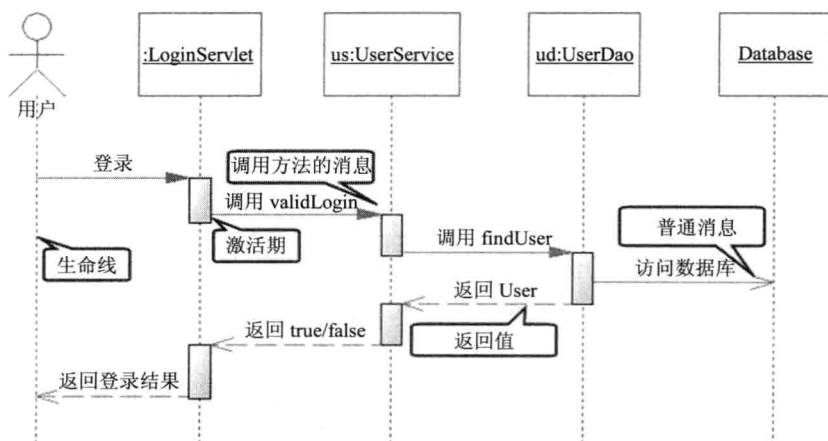


图 2.18 用户登录顺序图

当绘制顺序图时，消息可以向两个方向扩展，消息穿梭在顺序图中，通常应该把消息发送者与接收者相邻摆放，尽量避免消息跨越多个对象。对象的激活期不是其存在的时间，而是它占据 CPU 的执行时间，绘制顺序图时，激活期要精确。

阅读顺序图也非常简单，通常从最上面的消息开始（也就是时间上最先开始的消息），然后沿消息方向依次阅读。

在大多数情况下，交互图中的参与者是对象，所以也可以直接在方框中放置对象名，UML 1.x 要求对象名有下划线；2.0 不再需要。

绘制顺序图主要是帮助开发者对某个用例的内部执行清晰化，当需要考察某个用例内部若干对象行为时，应使用顺序图，顺序图擅长表现对象之间的协作顺序，不擅长表现行为的精确定义。



提示：

与顺序图类似的还有通信图（以前也被称为协作图），通信图同样可以准确地描述对象之间的交互关系，但通信图没有精确的时间概念。一般来说，通信图可以描述的内容，顺序图都可以描述，但顺序图比通信图多了时间的概念。

2.2.6 活动图

活动图和状态机图都被称为演化图，其区别和联系如下。

- 活动图：用于描述用例内部的活动或方法的流程，如果除去活动图中的并行活动描述，它就变成流程图。
- 状态机图：描述某一对象生命周期中需要关注的不同状态，并会详细描述刺激对象状态改变的事件，以及对对象状态改变时所采取的动作。

演化图的 5 要素如下。

- 状态：状态是对象响应事件前后的不同面貌，状态是某个时间段对象所保持的稳定态，目前的软件计算都是基于稳定态的，对象的稳定态是对象的固有特征，一个对象的状态一般是有限的。有限状态的对象是容易计算的，对象的状态越多，对象的状态迁移越复杂，对象状态可以想象成对象演化过程中的快照。
- 事件：来自对象外界的刺激，通常的形式是消息的传递，只是相对对象而言发生了事件。事件是对象状态发生改变的原动力。
- 动作：动作是对象针对所发生事件所做的处理，实际上通常表现为某个方法被执行。
- 活动：活动是动作激发的后续系统行为。
- 条件：条件指事件发生所需要具备的条件。

对于激发对象状态改变的事件，通常有如下两种类型。

- 内部事件：从系统内部激发的事件，一个对象的方法（动作）调用（通过事件激活）另一个对象的方法（动作）。
- 外部事件：从系统边界外激发的事件，例如用户的鼠标、键盘动作。

活动图主要用于描述过程原理、业务逻辑以及 workflow 技术。活动图非常类似于传统的流程图，它也使用圆角矩形表示活动，使用带箭头的实线表示事件；区别是活动图支持并发。图 2.19 显示了简单的活动图。

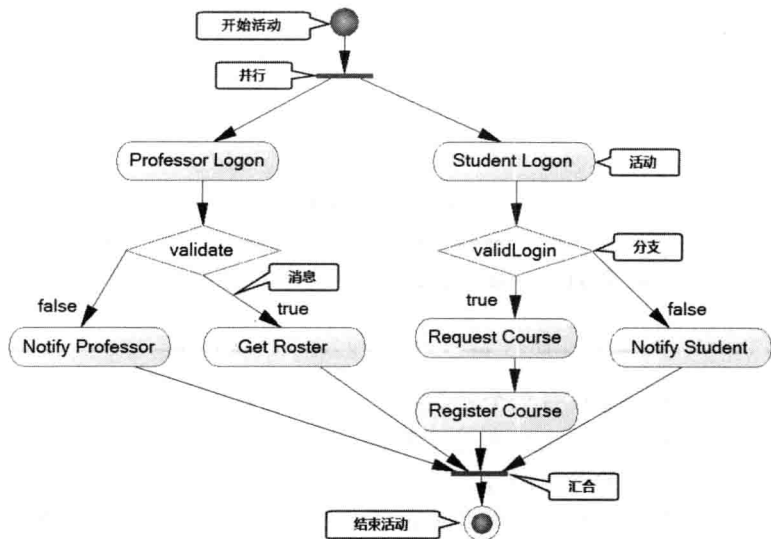


图 2.19 活动图

从图 2.19 可以看出，如果将这个活动图的两支分开，每支就是一个传统的流程图，每个活动依次向下，遇到条件分支使用菱形框来表示条件。与传统的流程图不同的是，活动图可以使用并行分支分出多条并行活动。

绘制活动图时以活动为中心，整个活动图只有一个开始活动，可以有多个结束活动。活动图需要将并行活动和串行活动分离，遇到分支和循环时最好像传统的流程图那样将分支、循环条件明确表示。活动图最大优点在于支持并行行为，并行对于 workflow 建模和过程建模非常重要。所以有了并行，因此需要进行同步，同步通过汇合来指明。

2.2.7 状态机图

状态机图表示某个对象所处的不同状态和该类的状态转换信息。实际上，通常只对“感兴趣的”对象绘制状态机图。也就是说，在系统活动期间具有三个或更多潜在状态的对象才需要考虑使用状态机图进行描述。

状态机图的符号集包括 5 个基本元素。

- 初始状态，使用实心圆来绘制。
- 状态之间的转换，使用具有带箭头的线段来绘制。
- 状态，使用圆角矩形来绘制。
- 判断点，使用空心圆来绘制。
- 一个或者多个终止点，使用内部包含实心圆的圆来绘制。

要绘制状态机图，首先绘制起点和一条指向该类的初始状态的转换线段。状态本身可以在图中的任意位置绘制，然后使用状态转换线段将它们连接起来。图 2.20 显示了 Hibernate 实体的状态机图。

图 2.20 描绘了 Hibernate 实体具有三个状态：瞬态、持久化和脱管。当程序通过 new 直接创建一个对象时，该对象处于瞬态；对一个瞬态的对象执行 save()、saveOrUpdate() 方法后该对象将会变成持久化状态；对一个持久化状态的实体执行 delete() 方法后该对象将变成瞬态；持久化状态和脱管状态也可以相互转换。

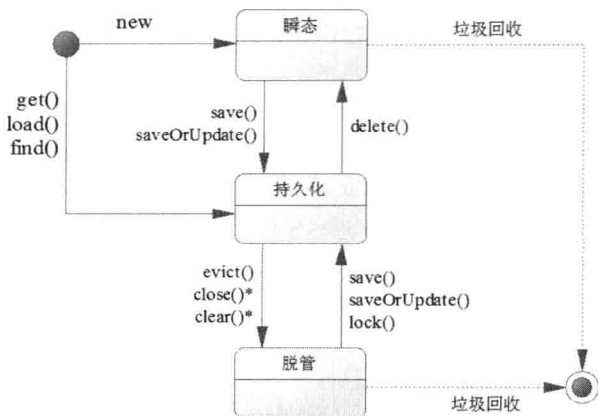


图 2.20 状态机图



提示:

阅读本书时无须理会 Hibernate 相关知识, 读者只需要明白图 2.20 所绘制的状态机图即可。

绘制状态机图时应该保证对象只有一个初始状态, 可以有多个终结状态。状态要表示对象的关键快照, 有重要的实际意义, 无关紧要的状态则无须考虑, 绘制状态机图时事件和方法要明确。

状态机图擅长表现单个对象的跨用例行为, 对于多个对象的交互行为应该考虑采用顺序图, 不要对系统的每个对象都画状态机图, 只对真正需要关心各个状态的对象才绘制状态机图。

2.3 Java 的面向对象特征

Java 是纯粹的面向对象编程语言, 完全支持面向对象的三大基本特征: 封装、继承和多态。Java 程序的组成单位就是类, 不管多大的 Java 应用程序, 都是由一个个类组成的。

2.3.1 一切都是对象

在 Java 语言中, 除了 8 个基本数据类型值之外, 一切都是对象, 而对象就是面向对象程序设计的中心。对象是人們要进行研究的任何事物, 从最简单的整数到复杂的飞机等均可看作对象, 它不仅能表示具体的事物, 还能表示抽象的规则、计划或事件。

对象具有状态, 一个对象用数据值来描述它的状态。Java 通过为对象定义成员变量来描述对象的状态; 对象还有操作, 这些操作可以改变对象的状态, 对象的操作也被称为对象的行为, Java 通过为对象定义方法来描述对象的行为。

对象实现了数据和操作的结合, 对象把数据和对数据的操作封装成一个有机的整体, 因此面向对象提供了更大的编程粒度, 对程序员来说, 更易于掌握和使用。

对象是 Java 程序的核心, 所以 Java 里的对象具有唯一性, 每个对象都有一个标识来引用它, 如果某个对象失去了标识, 这个对象将变成垃圾, 只能等着系统垃圾回收机制来回收它。Java 语言不允许直接访问对象, 而是通过对对象的引用来操作对象。

2.3.2 类和对象

具有相同或相似性质的一组对象的抽象就是类, 类是对一类事物的描述, 是抽象的、概念上的定义; 对象是实际存在的该类事物的个体, 因而也称为实例 (instance)。

对象的抽象化是类, 类的具体化就是对象, 也可以说类的实例是对象。类用来描述一系列对象, 类概述每个对象应包括的数据, 类概述每个对象的行为特征。因此, 可以把类理解成某种概念、定义, 它规定了某类对象所共同具有的数据和行为特征。

Java 语言使用 `class` 关键字定义类，定义类时可使用成员变量来描述该类对象的数据，可使用方法来描述该类对象的行为特征。

在客观世界中有若干类，这些类之间有一定的结构关系。通常有如下两种主要的结构关系。

- 一般→特殊关系：这种关系就是典型的继承关系，Java 语言使用 `extends` 关键字来表示这种继承关系，Java 的子类是一种特殊的父类。因此，这种一般→特殊的关系其实是一种“is a”关系。



提示：

在讲授面向对象时经常提的一个概念——一般→特殊的关系也可代表大类和小类的关系。比如水果→苹果，就是典型的一般→特殊的关系，苹果 is a 水果，水果的范围是不是比苹果的范围大呢？所以可以认为：父类也可被称为大类，子类也可被称为小类。

- 整体→部分结构关系：也被称为组装结构，这是典型的组合关系，Java 语言通过在一个类里保存另一个对象的引用来实现这种组合关系。因此，这种整体→部分结构关系其实是一种“has a”关系。

开发者定义了 Java 类之后，就可以使用 `new` 关键字来创建指定类的对象了，每个类可以创建任意多个对象，多个对象的成员变量值可以不同——这表现为不同对象的数据存在差异。

2.4 本章小结

本章主要介绍了面向对象的相关概念，也简要介绍了结构化程序设计的相关知识，包括结构化程序设计的基本特征以及存在的缺陷，还详细介绍了结构化程序设计的三种基本结构。本章重点介绍了面向对象程序设计的相关概念，以及面向对象程序设计的三个基本特征，并简要介绍了 Java 语言对面向对象特征的支持。本章详细介绍了 UML 的概念以及相关知识，并通过示例讲解了常用 UML 图形的绘制方法，这些 UML 图形是读者进行面向对象分析的重要方法，也是读者阅读本书后面章节的基础知识。

第 3 章

数据类型和运算符

本章要点

- 注释的重要性和用途
- 单行注释语法和多行注释语法
- 文档注释的语法和常用的 javadoc 标记
- javadoc 命令的用法
- 掌握查看 API 文档的方法
- 数据类型的两大类
- 8 种基本类型及各自的注意点
- 自动类型转换
- 强制类型转换
- 表达式类型的自动提升
- 直接量的类型和赋值
- Java 提供的基本运算符
- 运算符的结合性和优先级

Java 语言是一门强类型语言。强类型包含两方面的含义：① 所有的变量必须先声明、后使用；② 指定类型的变量只能接受类型与之匹配的值。强类型语言可以在编译过程中发现源代码的错误，从而保证程序更加健壮。Java 语言提供了丰富的基本数据类型，例如整型、字符型、浮点型和布尔型等。基本类型大致上可以分为两类：数值类型和布尔类型，其中数值类型包括整型、字符型和浮点型，所有数值类型之间可以进行类型转换，这种类型转换包括自动类型转换和强制类型转换。

Java 语言还提供了一系列功能丰富的运算符，这些运算符包括所有的算术运算符，以及功能丰富的位运算符、比较运算符、逻辑运算符，这些运算符是 Java 编程的基础。将运算符和操作数连接在一起就形成了表达式。

3.1 注释

编写程序时总需要为程序添加一些注释，用以说明某段代码的作用，或者说明某个类的用途、某个方法的功能，以及该方法的参数和返回值的数据类型及意义等。

程序注释的作用非常大，很多初学者在开始学习 Java 语言时，会很努力写程序，但不大会注意添加注释，他们认为添加注释是一件浪费时间，而且没有意义的事情。经过一段时间的学习，他们写出了一些不错的小程序，如一些游戏、工具软件等。再经过一段时间的学习，他们开始意识到当初写的程序在结构上有很多不足，需要重构。于是打开源代码，他们以为可以很轻松地改写原有的代码，但这时发现理解原来写的代码非常困难，很难理解原有的编程思路。

为什么要添加程序注释？至少有如下三方面的考虑。

- 永远不要过于相信自己的理解力！当你思路通畅，进入编程境界时，你可以很流畅地实现某个功能，但这种流畅可能是因为你当时正处于这种开发思路中。为了在再次阅读这段代码时，还能找回当初编写这段代码的思路，建议添加注释！
- 可读性第一，效率第二！在那些“古老”的岁月里，编程是少数人的专利，他们随心所欲地写程序，他们以追逐程序执行效率为目的。但随着软件行业的发展，人们发现仅有少数技术极客编程满足不了日益增长的软件需求，越来越多的人加入了编程队伍，并引入了工程化的方式来管理软件开发。这个时候，软件开发变成团队协作作战，团队成员的沟通变得很重要，因此，一个人写的代码，需要被整个团队的其他人所理解；而且，随着硬件设备的飞速发展，程序的可读性取代执行效率变成了第一考虑的要素。
- 代码即文档！很多刚刚学完学校软件工程课程的学生会以为：文档就是 Word 文档！实际上，程序源代码是程序文档的重要组成部分，在想着把各种软件相关文档写规范的同时，不要忘了把软件里最重要的文档——源代码写规范！

程序注释是源代码的一个重要部分，对于一份规范的程序源代码而言，注释应该占到源代码的 1/3 以上。几乎所有的编程语言都提供了添加注释的方法。一般的编程语言都提供了基本的单行注释和多行注释，Java 语言也不例外，除此之外，Java 语言还提供了一种文档注释。Java 语言的注释一共有三种类型。

- 单行注释。
- 多行注释。
- 文档注释。

➤➤ 3.1.1 单行注释和多行注释

单行注释就是在程序中注释一行代码，在 Java 语言中，将双斜线（//）放在需要注释的内容之前就可以了；多行注释是指一次性地将程序中多行代码注释掉，在 Java 语言中，使用“/*”和“*/”将程序中需要注释的内容包含起来，“/*”表示注释开始，而“*/”表示注释结束。

下面代码中增加了单行注释和多行注释。

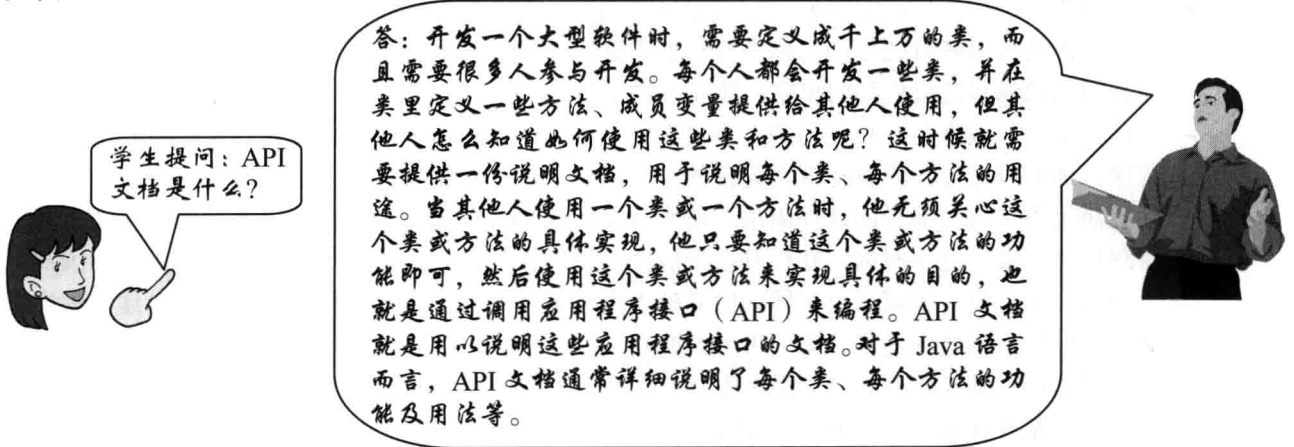
程序清单: codes\03\3.1\CommentTest.java

```
public class CommentTest
{
    /*
    这里面的内容全部是多行注释
    Java 语言真的很有趣
    */
    public static void main(String[] args)
    {
        // 这是一行简单的注释
        System.out.println("Hello World!");
        // System.out.println("这行代码被注释了, 将不会被编译、执行!");
    }
}
```

除此之外, 添加注释也是调试程序的一个重要方法。如果觉得某段代码可能有问题, 可以先把这段代码注释起来, 让编译器忽略这段代码, 再次编译、运行, 如果程序可以正常执行, 则可以说明错误就是由这段代码引起的, 这样就缩小了错误所在的范围, 有利于排错; 如果依然出现相同的错误, 则可以说明错误不是由这段代码引起的, 同样也缩小了错误所在的范围。

3.1.2 文档注释

Java 语言还提供了一种功能更强大的注释形式: 文档注释。如果编写 Java 源代码时添加了合适的文档注释, 然后通过 JDK 提供的 javadoc 工具可以直接将源代码里的文档注释提取成一份系统的 API 文档。



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

下载 Java 8 的 API 文档很简单, 登录 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 站点, 将页面上的滚动条向下滚动, 找到 “Additional Resources” 部分, 看到如图 3.1 所示的页面。

单击如图 3.1 所示的链接即可下载得到 Java SE 8 文档, 这份文档里包含了 JDK 的 API 文档。下载成功后得到一个 jdk-8-apidocs.zip 文件。

将 jdk-8-apidocs.zip 文件解压缩到任意路径, 将会得到一个 docs 文件夹, 这个文件夹下的内容就是 JDK 文档, JDK 文档不仅包含 API 文档, 还包含 JDK 的其他说明文档。

进入 docs/api 路径下, 打开 index.html 文件, 可以

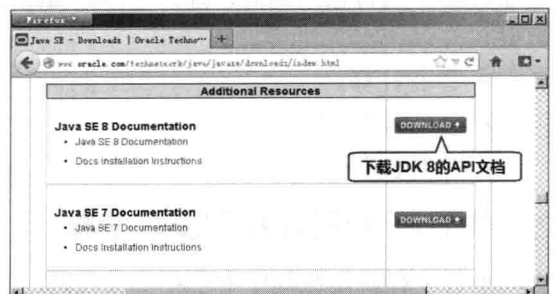


图 3.1 下载 JDK 8 的 API 文档

看到 JDK 8 API 文档首页，这个首页就是一个典型的 Java API 文档首页，如图 3.2 所示。

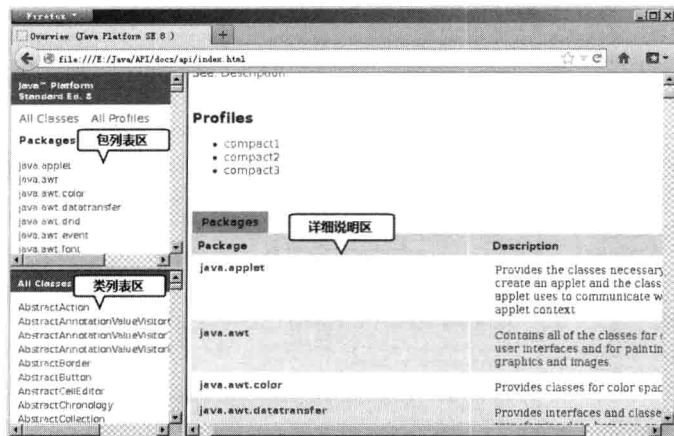


图 3.2 API 文档首页

从图 3.2 所示的首页中可以看出，API 文档页面被分为三个部分，左上角部分是 API 文档“包列表区”，在该区域内可以查看 Java 类的所有包（至于什么是包，本书将在后面章节介绍）；左下角是 API 文档的“类列表区”，用于查看 Java 的所有类；右边页面是“详细说明区”，默认显示的是各包空间的说明信息。



提示：

在图 3.2 所示的“详细说明区”的 Profiles 下看到 compact1、compact2、compact3 这三个 profile，这是 JDK 8 新增的功能。每个 compact profile 都是 Java SE 平台的预定义子集，通过使用特定的 compact profile，可以让应用无须整合 Java 平台，这样可以方便地在小型设备上部署、运行 Java 程序。

如果单击“类列表区”里列出的某个类，将看到右边页面变成如图 3.3 所示的格局。



图 3.3 类说明区格局（一）

当单击了左边“类列表区”中的 Button 类后，即可看到右边页面显示了 Button 类的详细信息，这些信息是使用 Button 类的重要资料。把图 3.3 所示窗口右边的滚动条向下滚动，将在“详细说明区”看到如图 3.4 所示的格局。

从图 3.4 所示的类说明区中可以看出，API 文档中详细列出了该类里包含的所有成分，通过查看该文档，开发者就可以掌握该类的用法。从图 3.4 所看到的内部类列表、成员变量（由 Field 意译而来）列表、构造器列表和方法列表只给出了一些简单描述，如果开发者需要获得更详细的信息，则可以单击具体的内部类、成员变量、构造器和方法的链接，从而看到对应项的详细用法说明。

对于内部类、成员变量、方法列表区都可以分为左右两格，其中左边一格是该项的修饰符、类型说

明, 右边一格是该项的简单说明。

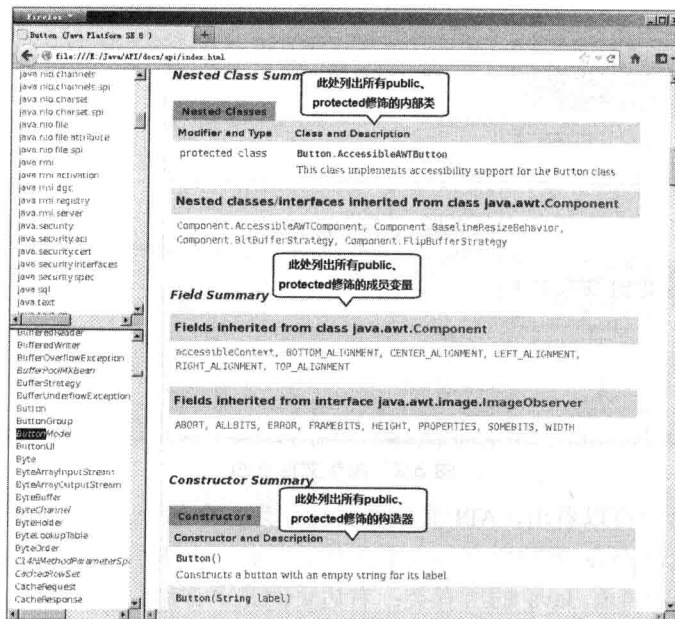


图 3.4 类说明区格局 (二)

同样, 在开发中定义类、方法时也可以先添加文档注释, 然后使用 javadoc 工具来生成自己的 API 文档。



注意：

此处介绍的成员变量、构造器、方法等可能有点超前, 读者可以参考后面的知识来理解如何定义成员变量、构造器、方法等, 此处的重点只是学习使用文档注释。



由于文档注释是用于生成 API 文档的, 而 API 文档主要用于说明类、方法、成员变量的功能。因此, javadoc 工具只处理文档源文件在类、接口、方法、成员变量、构造器和内部类之前的注释, 忽略其他地方的文档注释。而且 javadoc 工具默认只处理以 public 或 protected 修饰的类、接口、方法、成员变量、构造器和内部类之前的文档注释。



注意：

API 文档类似于产品的使用说明书, 通常使用说明书只需要介绍那些暴露的、供用户使用的部分。Java 类中只有以 public 或 protected 修饰的内容才是希望暴露给别人使用的内容, 因此 javadoc 默认只处理 public 或 protected 修饰的内容。如果开发者确实希望 javadoc 工具可以提取 private 修饰的内容, 则可以在使用 javadoc 工具时增加 -private 选项。



文档注释以斜线后紧跟两个星号 (/**) 开始, 以星号后紧跟一个斜线 (*/) 结束, 中间部分全部

都是文档注释，会被提取到 API 文档中。

下面先编写一个 JavadocTest 类，这个类里包含了对类、方法、成员变量的文档注释。

程序清单：codes\03\3.1\JavadocTest.java

```
package lee;
/**
 * Description:
 * <br>网站: <a href="http://www.crazyit.org">疯狂 Java 联盟</a>
 * <br>Copyright (C), 2001-2015, Yeeku.H.Lee
 * <br>This program is protected by copyright laws.
 * <br>Program Name:
 * <br>Date:
 * @author Yeeku.H.Lee kongyeeku@163.com
 * @version 1.0
 */
public class JavadocTest
{
    /**
     * 简单测试成员变量
     */
    protected String name;
    /**
     * 主方法，程序的入口
     */
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

再编写一个 Test 类，这个类里包含了对类、构造器、成员变量的文档注释。

程序清单：codes\03\3.1\Test.java

```
package yeeku;
/**
 * Description:
 * <br>网站: <a href="http://www.crazyit.org">疯狂 Java 联盟</a>
 * <br>Copyright (C), 2001-2015, Yeeku.H.Lee
 * <br>This program is protected by copyright laws.
 * <br>Program Name:
 * <br>Date:
 * @author Yeeku.H.Lee kongyeeku@163.com
 * @version 1.0
 */
public class Test
{
    /**
     * 简单测试成员变量
     */
    public int age;
    /**
     * Test 类的测试构造器
     */
    public Test()
    {
    }
}
```

上面 Java 程序中粗体字标识部分就是文档注释。编写好上面的 Java 程序后，就可以使用 javadoc 工具提取这两个程序中的文档注释来生成 API 文档了。javadoc 命令的基本用法如下：

javadoc 选项 Java 源文件|包

javadoc 命令可对源文件、包生成 API 文档，在上面的语法格式中，Java 源文件可以支持通配符，例如，使用 *.java 来代表当前路径下所有的 Java 源文件。javadoc 的常用选项有如下几个。

- -d <directory>: 该选项指定一个路径，用于将生成的 API 文档放到指定目录下。
- -windowtitle <text>: 该选项指定一个字符串，用于设置 API 文档的浏览器窗口标题。

- `-doctitle <html-code>`: 该选项指定一个 HTML 格式的文本, 用于指定概述页面的标题。

注意:

只有对处于多个包下的源文件来生成 API 文档时, 才有概述页面。



- `-header <html-code>`: 该选项指定一个 HTML 格式的文本, 包含每个页面的页眉。

除此之外, `javadoc` 命令还包含了大量其他选项, 读者可以通过在命令行窗口执行 `javadoc -help` 来查看 `javadoc` 命令的所有选项。

在命令行窗口执行如下命令来为刚刚编写的两个 Java 程序生成 API 文档:

```
javadoc -d apidoc -windowtitle 测试 -doctitle 学习 javadoc 工具的测试 API 文档 -header 我的类 *Test.java
```

在 `JavadocTest.java` 和 `Test.java` 所在路径下执行上面命令, 可以看到生成 API 文档的提示信息。进入 `JavadocTest.java` 和 `Test.java` 所在路径, 可以看到一个 `apidoc` 文件夹, 该文件夹下的内容就是刚刚生成的 API 文档, 进入 `apidoc` 路径, 打开 `index.html` 文件, 将看到如图 3.5 所示的页面。

同样, 如果单击如图 3.5 所示页面的左下角类列表区中的某个类, 则可以看到该类的详细说明, 如图 3.3 和图 3.4 所示。

除此之外, 如果希望 `javadoc` 工具生成更详细的文档信息, 例如为方法参数、方法返回值等生成详细的说明信息, 则可利用 `javadoc` 标记。常用的 `javadoc` 标记如下。

- `@author`: 指定 Java 程序的作者。
- `@version`: 指定源文件的版本。
- `@deprecated`: 不推荐使用的方法。
- `@param`: 方法的参数说明信息。
- `@return`: 方法的返回值说明信息。
- `@see`: “参见”, 用于指定交叉参考的内容。
- `@exception`: 抛出异常的类型。
- `@throws`: 抛出的异常, 和 `@exception` 同义。

需要指出的是, 这些标记的使用是有位置限制的。上面这些标记可以出现在类或者接口文档注释中的有 `@see`、`@deprecated`、`@author`、`@version` 等; 可以出现在方法或构造器文档注释中的有 `@see`、`@deprecated`、`@param`、`@return`、`@throws` 和 `@exception` 等; 可以出现在成员变量的文档注释中的有 `@see` 和 `@deprecated` 等。

下面的 `JavadocTagTest` 程序包含了一个 `hello` 方法, 该方法的文档注释使用了 `@param` 和 `@return` 等文档标记。

程序清单: `codes\03\3.1\JavadocTagTest.java`

```
package yeeku;
/**
 * Description:
 * <br>网站: <a href="http://www.crazyit.org">疯狂 Java 联盟</a>
 * <br>Copyright (C), 2001-2015, Yeeku.H.Lee
 * <br>This program is protected by copyright laws.
 * <br>Program Name:
 * <br>Date:
```

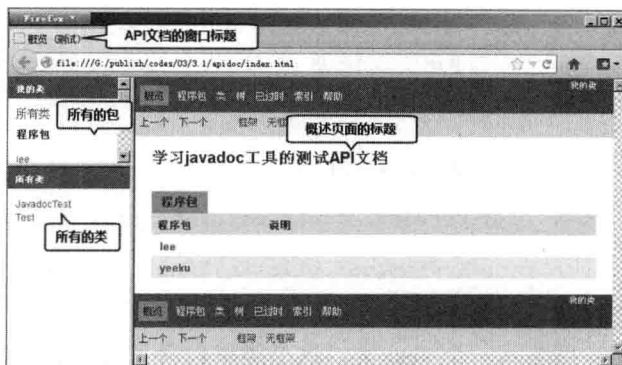


图 3.5 自己生成的 API 文档

```

* @author Yeeku.H.Lee kongyeeku@163.com
* @version 1.0
*/
public class JavadocTagTest
{
    /**
     * 一个得到打招呼字符串的方法
     * @param name 该参数指定向谁打招呼
     * @return 返回打招呼的字符串
     */
    public String hello(String name)
    {
        return name + ", 你好!";
    }
}

```

上面程序中粗体字标识出使用 javadoc 标记的示范。再次使用 javadoc 工具来生成 API 文档，这次为了提取到文档中的 @author 和 @version 等标记信息，在使用 javadoc 工具时增加 -author 和 -version 两个选项，即按如下格式来运行 javadoc 命令：

```

javadoc -d apidoc -windowtitle 测试 -doctitle 学习 javadoc 工具的测试 API 文档 -header 我的类
-version -author *Test.java

```

上面命令将会提取 Java 源程序中的 -author 和 -version 两个标记的信息。除此之外，还会提取 @param 和 @return 标记的信息，因而将会看到如图 3.6 所示的 API 文档页面。



注意：

javadoc 工具默认不会提取 @author 和 @version 两个标记的信息，如果需要提取这两个标记的信息，应该在使用 javadoc 工具时指定 -author 和 -version 两个选项。

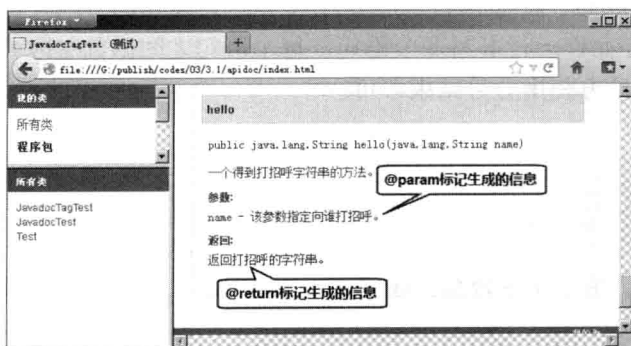


图 3.6 使用文档标记设置更丰富的 API 信息

对比图 3.2 和图 3.5，两个图都显示了 API 文档的首页，但图 3.2 显示的 API 文档首页里包含了对每个包的详细说明，而图 3.5 显示的文档首页里每个包的说明部分都是空白。这是因为 API 文档中的包注释并不是直接放在 Java 源文件中的，而是必须另外指定，通常通过一个标准的 HTML 文件来提供包注释，这个文件被称为包描述文件。包描述文件的文件名通常是 package.html，并与该包下所有的 Java 源文件放在一起，javadoc 工具会自动寻找对应的包描述文件，并提取

该包描述文件中的 <body/> 元素里的内容，作为该包的描述信息。

接下来还是使用上面编写的三个 Java 文件，但把这三个 Java 文件按包结构分开组织存放，并提供对应的包描述文件，源文件和对应包描述文件的组织结构如下（该示例位于光盘的 codes\03\3.1\package 路径下）。

- lee 文件夹：包含 JavadocTest.java 文件（该 Java 类的包为 lee），对应包描述文件 package.html。
- yeeku 文件夹：包含 Test.java 文件和 JavadocTagTest.java 文件（这两个 Java 类的包为 yeeku），对应包描述文件 package.html。

在命令行窗口进入 lee 和 yeeku 所在路径，执行如下命令：

```

javadoc -d apidoc -windowtitle 测试 -doctitle 学习 javadoc 工具的测试 API 文档 -header 我的类
-version -author lee yeeku

```

上面命令指定对 lee 包和 yeeku 包来生成 API 文档，而不是对 Java 源文件来生成 API 文档，这也是允许的。其中 lee 包和 yeeku 包下面都提供了对应的包描述文件。

打开上面命令生成的 API 文档首页, 将可以看到如图 3.7 所示的页面。

可能有读者会发现, 如果需要设置包描述信息, 则需要将 Java 源文件按包结构来组织存放, 这不是问题。实际上, 当编写 Java 源文件时, 通常总会按包结构来组织存放 Java 源文件, 这样更有利于项目的管理。

现在生成的 API 文档已经非常“专业”了, 和系统提供的 API 文档基本类似。关于 Java 文档注释和 javadoc 工具使用的介绍也基本告一段落了。

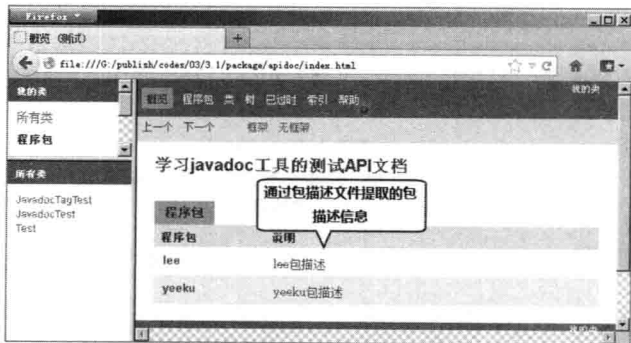


图 3.7 设置包描述信息

3.2 标识符和关键字

Java 语言也和其他编程语言一样, 使用标识符作为变量、对象的名字, 也提供了系列的关键字用以实现特别的功能。本节详细介绍 Java 语言的标识符和关键字等内容。

3.2.1 分隔符

Java 语言里的分号 (;)、花括号 ({})、方括号([])、圆括号 ()、空格、圆点 (.) 都具有特殊的分隔作用, 因此被统称为分隔符。

1. 分号

Java 语言里对语句的分隔不是使用回车来完成的, Java 语言采用分号 (;) 作为语句的分隔, 因此每个 Java 语句必须使用分号作为结尾。Java 程序允许一行书写多个语句, 每个语句之间以分号隔开即可; 一个语句也可以跨多行, 只要在最后结束的地方使用分号结束即可。

例如, 下面语句都是合法的 Java 语句。

```
int age = 25; String name = "李刚";
String hello = "你好!" +
    "Java";
```

值得指出的是, Java 语句可以跨越多行书写, 但一个字符串、变量名不能跨越多行。例如, 下面的 Java 语句是错误的。

```
// 字符串不能跨越多行
String a = "dddddd
xxxxxxx";
// 变量名不能跨越多行
String na
me = "李刚";
```

不仅如此, 虽然 Java 语法允许一行书写多个语句, 但从程序可读性角度来看, 应该避免在一行书写多个语句。

2. 花括号

花括号的作用就是定义一个代码块, 一个代码块指的就是“{”和“}”所包含的一段代码, 代码块在逻辑上是一个整体。对 Java 语言而言, 类定义部分必须放在一个代码块里, 方法体部分也必须放在一个代码块里。除此之外, 条件语句中的条件执行体和循环语句中的循环体通常也放在代码块里。

花括号一般是成对出现的, 有一个“{”则必然有一个“}”, 反之亦然。

3. 方括号

方括号的主要作用是用于访问数组元素，方括号通常紧跟数组变量名，而方括号里指定希望访问的数组元素的索引。

例如，如下代码：

```
// 下面代码试图为名为 a 的数组的第四个元素赋值  
a[3] = 3;
```

4. 圆括号

圆括号是一个功能非常丰富的分隔符：定义方法时必须使用圆括号来包含所有的形参声明，调用方法时也必须使用圆括号来传入实参值；不仅如此，圆括号还可以将表达式中某个部分括成一个整体，保证这个部分优先计算；除此之外，圆括号还可以作为强制类型转换的运算符。

关于圆括号分隔符在后面还有更进一步的介绍，此处不再赘述。

5. 空格

Java 语言使用空格分隔一条语句的不同部分。Java 语言是一门格式自由的语言，所以空格几乎可以出现在 Java 程序的任何地方，也可以出现任意多个空格，但不要使用空格把一个变量名隔开成两个，这将导致程序出错。

Java 语言中的空格包含空格符（Space）、制表符（Tab）和回车（Enter）等。

除此之外，Java 源程序还会使用空格来合理缩进 Java 代码，从而提供更好的可读性。

6. 圆点

圆点（.）通常用作类/对象和它的成员（包括成员变量、方法和内部类）之间的分隔符，表明调用某个类或某个实例的指定成员。关于圆点分隔符的用法，后面还会有更进一步的介绍，此处不再赘述。

3.2.2 标识符规则

标识符就是用于给程序中变量、类、方法命名的符号。Java 语言的标识符必须以字母、下画线（_）、美元符（\$）开头，后面可以跟任意数目的字母、数字、下画线（_）和美元符（\$）。此处的字母并不局限于 26 个英文字母，而且可以包含中文字符、日文字符等。

由于 Java 语言支持 Unicode 6.2.0 字符集，因此 Java 的标识符可以使用 Unicode 6.0.0 所能表示的多种语言的字符。Java 语言是区分大小写的，因此 abc 和 Abc 是两个不同的标识符。

使用标识符时，需要注意如下规则。

- 标识符可以由字母、数字、下画线（_）和美元符（\$）组成，其中数字不能打头。
- 标识符不能是 Java 关键字和保留字，但可以包含关键字和保留字。
- 标识符不能包含空格。
- 标识符只能包含美元符（\$），不能包含@、#等其他特殊字符。

3.2.3 Java 关键字

Java 语言中有一些具有特殊用途的单词被称为关键字（keyword），当定义标识符时，不要让标识符和关键字相同，否则将引起错误。例如，下面代码将无法通过编译。

```
// 试图定义一个名为 boolean 的变量，但 boolean 是关键字，不能作为标识符  
int boolean;
```

Java 的所有关键字都是小写的，TRUE、FALSE 和 NULL 都不是 Java 关键字。

Java 一共包含 50 个关键字, 如表 3.1 所示。

表 3.1 Java 关键字

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

上面的 50 个关键字中, enum 是从 Java 5 新增的关键字, 用于定义一个枚举。而 goto 和 const 这两个关键字也被称为保留字 (reserved word), 保留字的意思是, Java 现在还未使用这两个关键字, 但可能在未来的 Java 版本中使用这两个关键字; 不仅如此, Java 还提供了三个特殊的直接量 (literal): true、false 和 null; Java 语言的标识符也不能使用这三个特殊的直接量。

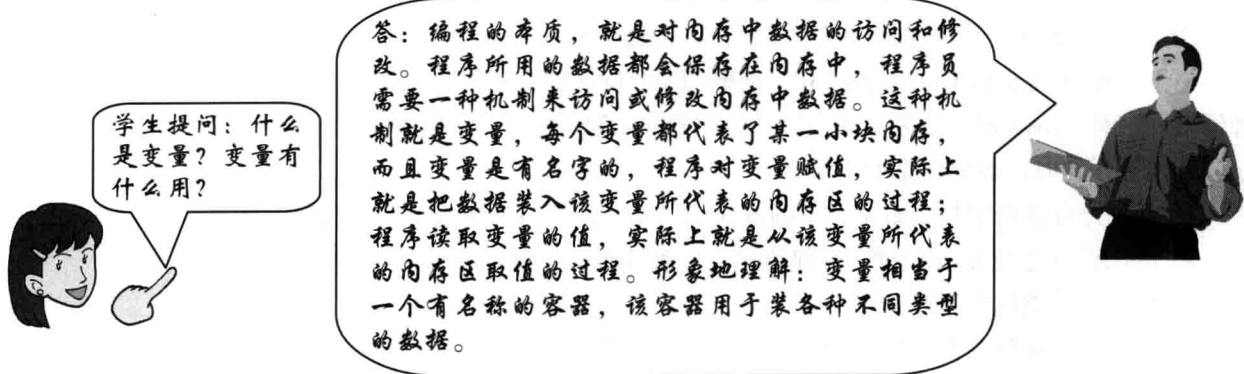
3.3 数据类型分类

Java 语言是强类型 (strongly typed) 语言, 强类型包含两方面的含义: ① 所有的变量必须先声明、后使用; ② 指定类型的变量只能接受类型与之匹配的值。这意味着每个变量和每个表达式都有一个在编译时就确定的类型。类型限制了一个变量能被赋的值, 限制了一个表达式可以产生的值, 限制了在这些值上可以进行的操作, 并确定了这些操作的含义。

强类型语言可以在编译时进行更严格的语法检查, 从而减少编程错误。

声明变量的语法非常简单, 只要指定变量的类型和变量名即可, 如下所示:

```
type varName[ = 初始值];
```



上面语法中, 定义变量时既可指定初始值, 也可不指定初始值。随着变量的作用范围的不同 (变量有成员变量和局部变量之分, 具体请参考本书 5.3 节内容), 变量还可能使用其他修饰符。但不管是哪种变量, 定义变量至少需要指定变量类型和变量名两个部分。定义变量时的变量类型可以是 Java 语言支持的所有类型。

Java 语言支持的类型分为两类: 基本类型 (Primitive Type) 和引用类型 (Reference Type)。

基本类型包括 boolean 类型和数值类型。数值类型有整数类型和浮点类型。整数类型包括 byte、short、int、long、char, 浮点类型包括 float 和 double。

**提示：**

char 代表字符型，实际上字符型也是一种整数类型，相当于无符号整数类型。

引用类型包括类、接口和数组类型，还有一种特殊的 null 类型。所谓引用数据类型就是对一个对象的引用，对象包括实例和数组两种。实际上，引用类型变量就是一个指针，只是 Java 语言里不再使用指针这个说法。

空类型 (null type) 就是 null 值的类型，这种类型没有名称。因为 null 类型没有名称，所以不可能声明一个 null 类型的变量或者转换到 null 类型。空引用 (null) 是 null 类型变量唯一的值。空引用 (null) 可以转换为任何引用类型。

在实际开发中，程序员可以忽略 null 类型，假定 null 只是引用类型的一个特殊直接量。

**注意：**

空引用 (null) 只能被转换成引用类型，不能转换成基本类型，因此不要把一个 null 值赋给基本数据类型的变量。



3.4 基本数据类型

Java 的基本数据类型分为两大类：boolean 类型和数值类型。而数值类型又可以分为整数类型和浮点类型，整数类型里的字符类型也可被单独对待。因此常把 Java 里的基本数据类型分为 4 类，如图 3.8 所示。

Java 只包含这 8 种基本数据类型，值得指出的是，字符串不是基本数据类型，字符串是一个类，也就是一个引用数据类型。

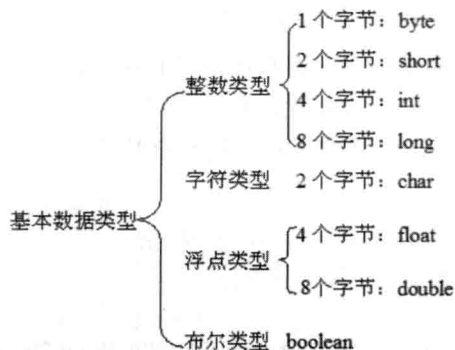


图 3.8 Java 的基本类型

3.4.1 整型

通常所说的整型，实际指的是如下 4 种类型。

- byte: 一个 byte 类型整数在内存里占 8 位，表数范围是： $-128(-2^7) \sim 127(2^7-1)$ 。
- short: 一个 short 类型整数在内存里占 16 位，表数范围是： $-32768(-2^{15}) \sim 32767(2^{15}-1)$ 。
- int: 一个 int 类型整数在内存里占 32 位，表数范围是： $-2147483648(-2^{31}) \sim 2147483647(2^{31}-1)$ 。
- long: 一个 long 类型整数在内存里占 64 位，表数范围是： $(-2^{63}) \sim (2^{63}-1)$ 。

int 是最常用的整数类型，因此在通常情况下，直接给出一个整数值默认就是 int 类型。除此之外，有如下两种情形必须指出。

- 如果直接将一个较小的整数值（在 byte 或 short 类型的表数范围内）赋给一个 byte 或 short 变量，系统会自动把这个整数值当成 byte 或者 short 类型来处理。
- 如果使用一个巨大的整数值（超出了 int 类型的表数范围）时，Java 不会自动把这个整数值当成 long 类型来处理。如果希望系统把一个整数值当成 long 类型来处理，应在这个整数值后增加 l 或者 L 作为后缀。通常推荐使用 L，因为英文字母 l 很容易跟数字 1 搞混。

下面的代码片段验证了上面的结论。

程序清单：codes\03\3.4\IntegerValTest.java

```
// 下面代码是正确的，系统会自动把 56 当成 byte 类型处理
byte a = 56;
/*
下面代码是错误的，系统不会把 99999999999999 当成 long 类型处理
long b = 99999999999999;
*/
```

```

所以超出 int 的表数范围，从而引起错误
*/
// long bigValue = 999999999999999;
// 下面代码是正确的，在巨大的整数值后使用 L 后缀，强制使用 long 类型
long bigValue2 = 9223372036854775807L;
    
```



注意：

可以把一个较小的整数值（在 int 类型的表数范围以内）直接赋给一个 long 类型的变量，这并不是因为 Java 会把这个较小的整数值当成 long 类型来处理，Java 依然把这个整数值当成 int 类型来处理，只是因为 int 类型的值会自动类型转换到 long 类型。



Java 中整数值有 4 种表示方式：十进制、二进制、八进制和十六进制，其中二进制的整数以 0b 或 0B 开头；八进制的整数以 0 开头；十六进制的整数以 0x 或者 0X 开头，其中 10~15 分别以 a~f（此处的 a~f 不区分大小写）来表示。

下面的代码片段分别使用八进制和十六进制的数。

程序清单：codes\03\3.4\IntegerValTest.java

```

// 以 0 开头的整数值是八进制的整数
int octalValue = 013;
// 以 0x 或 0X 开头的整数值是十六进制的整数
int hexValue1 = 0x13;
int hexValue2 = 0XaF;
    
```

在某些时候，程序需要直接使用二进制整数，二进制整数更“真实”，更能表达整数在内存中的存在形式。不仅如此，有些程序（尤其在开发一些游戏时）使用二进制整数会更便捷。

Java 7 新增了对二进制整数的支持，二进制的整数以 0b 或者 0B 开头。程序片段如下。

程序清单：codes\03\3.4\IntegerValTest.java

```

// 定义两个 8 位的二进制整数
int binVal1 = 0b11010100;
byte binVal2 = 0B01101001;
// 定义一个 32 位的二进制整数，最高位是符号位
int binVal3 = 0B10000000000000000000000000000011;
System.out.println(binVal1); // 输出 212
System.out.println(binVal2); // 输出 105
System.out.println(binVal3); // 输出 -2147483645
    
```

从上面粗体字可以看出，当定义 32 位的二进制整数时，最高位其实是符号位，当符号位是 1 时，表明它是一个负数，负数在计算机里是以补码的形式存在的，因此还需要换算成原码。



提示：

所有数字在计算机底层都是以二进制形式存在的，原码是直接将一个数值换算成二进制数。但计算机以补码的形式保存所有的整数。补码的计算规则：正数的补码和原码完全相同，负数的补码是其反码加 1；反码是对原码按位取反，只是最高位（符号位）保持不变。

将上面的二进制整数 binVal3 转换成十进制数的过程如图 3.9 所示。

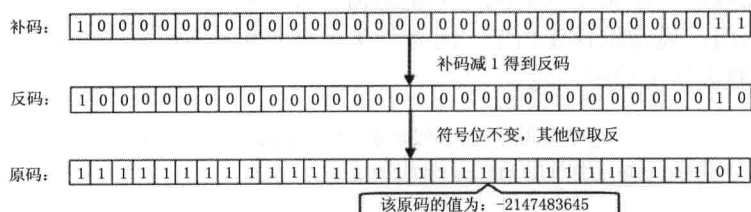


图 3.9 二进制整数转换成十进制数

正如前面所指出的, 整数值默认就是 `int` 类型, 因此使用二进制形式定义整数时, 二进制整数默认占 32 位, 其中第 32 位是符号位; 如果在二进制整数后添加 `l` 或 `L` 后缀, 那么这个二进制整数默认占 64 位, 其中第 64 位是符号位。

例如如下程序。

程序清单: codes\03\3.4\IntegerValTest.java

```
/*
  定义一个 8 位的二进制整数, 该数值默认占 32 位, 因此它是一个正数
  只是强制类型转换成 byte 时产生了溢出, 最终导致 binVal4 变成了 -23
*/
byte binVal4 = (byte)0b11101001;
/*
  定义一个 32 位的二进制整数, 最高位是 1
  但由于数值后添加了 L 后缀, 因此该整数实际占 64 位, 第 32 位的 1 不是符号位
  因此 binVal5 的值等于 2 的 31 次方 + 2 + 1
*/
long binVal5 = 0b10000000000000000000000000000011L;
System.out.println(binVal4); // 输出 -23
System.out.println(binVal5); // 输出 2147483651
```

上面程序中粗体字代码与前面程序片段的粗体字代码基本相同, 只是在定义二进制整数时添加了“L”后缀, 这就表明把它当成 `long` 类型处理, 因此该整数实际占 64 位。此时的第 32 位不再是符号位, 因此它依然是一个正数。

至于程序中的 `byte binVal4 = (byte)0b11101001;` 代码, 其中 `0b11101001` 依然是一个 32 位的正整数, 只是程序进行强制类型转换时发生了溢出, 导致它变成了负数。关于强制类型转换的知识请参考本章 3.5 节。

3.4.2 字符型

字符型通常用于表示单个的字符, 字符型值必须使用单引号 (') 括起来。Java 语言使用 16 位的 Unicode 字符集作为编码方式, 而 Unicode 被设计成支持世界上所有书面语言的字符, 包括中文字符, 因此 Java 程序支持各种语言的字符。

学生提问:
什么是字符集?

答: 严格来说, 计算机无法保存电影、音乐、图片、字符……计算机只能保存二进制码。因此电影、音乐、图片、字符都需要先转换为二进制码, 然后才能保存。因此平时会听到 `avi`、`mov` 等各种电影格式; `mp3`、`wma` 等各种音乐格式; `gif`、`png` 等各种图片格式; 之所以需要这些格式, 就是因为计算机需要先将电影、音乐、图片等转换为二进制码, 然后才能保存。对于保存字符就简单多了, 直接把所有需要保存的字符编号, 当计算机要保存某个字符时, 只要将该字符的编号转换为二进制码, 然后保存起来即可。所谓字符集, 就是给所有字符的编号组成总和。早期美国人给英文字符、数字、标点符号等字符进行了编号, 他们认为所有字符顶多 100 多个, 只要一个字节 (8 位, 支持 256 个字符编号) 即可为所有字符编号——这就是 ASCII 字符集。后来, 亚洲国家纷纷为本国文字进行编号——即制订本国的字符集, 但这些字符集并不兼容。于是美国人又为世界上所有书面语言的字符进行了统一编号, 这次他们用了两个字节 (16 位, 支持 65536 个字符编号), 这就是 Unicode 字符集。

字符型值有如下三种表示形式。

➤ 直接通过单个字符来指定字符型值, 例如 `'A'`、`'9'` 和 `'0'` 等。

- 通过转义字符表示特殊字符型值，例如'\n'、'\t'等。
- 直接使用 Unicode 值来表示字符型值，格式是'\uXXXX'，其中 XXXX 代表一个十六进制的整数。Java 语言中常用的转义字符如表 3.2 所示。

表 3.2 Java 语言中常用的转义字符

转义字符	说 明	Unicode 表示方式
\b	退格符	\u0008
\n	换行符	\u000a
\r	回车符	\u000d
\t	制表符	\u0009
\"	双引号	\u0022
\'	单引号	\u0027
\\	反斜线	\u005c

字符型值也可以采用十六进制编码方式来表示，范围是'\u0000'~'\uFFFF'，一共可以表示 65536 个字符，其中前 256 个 ('\u0000'~'\u00FF') 字符和 ASCII 码中的字符完全重合。

由于计算机底层保存字符时，实际是保存该字符对应的编号，因此 char 类型的值也可直接作为整型值来使用，它相当于一个 16 位的无符号整数，表数范围是 0~65535。



提示：

char 类型的变量、值完全可以参与加、减、乘、除等数学运算，也可以比较大小——实际上都是用该字符对应的编码参与运算。

如果把 0~65535 范围内的一个 int 整数赋给 char 类型变量，系统会自动把这个 int 整数当成 char 类型来处理。

下面程序简单示范了字符型变量的用法。

程序清单：codes\03\3.4\CharTest.java

```
public class CharTest
{
    public static void main(String[] args)
    {
        // 直接指定单个字符作为字符值
        char aChar = 'a';
        // 使用转义字符来作为字符值
        char enterChar = '\r';
        // 使用 Unicode 编码值来指定字符值
        char ch = '\u9999';
        // 将输出一个'香'字符
        System.out.println(ch);
        // 定义一个'疯'字符值
        char zhong = '疯';
        // 直接将一个 char 变量当成 int 类型变量使用
        int zhongValue = zhong;
        System.out.println(zhongValue);
        // 直接把一个 0~65535 范围内的 int 整数赋给一个 char 变量
        char c = 97;
        System.out.println(c);
    }
}
```

Java 没有提供表示字符串的基本数据类型，而是通过 String 类来表示字符串，由于字符串由多个字符组成，因此字符串要使用双引号括起来。如下代码：

```
// 下面代码定义了一个 s 变量，它是一个字符串实例的引用，它是一个引用类型的变量
String s = "沧海月明珠有泪，蓝田玉暖日生烟。";
```

读者必须注意: `char` 类型使用单引号括起来, 而字符串使用双引号括起来。关于 `String` 类的用法以及对应的各种方法, 读者应该通过查阅 API 文档来掌握, 以此来练习使用 API 文档。

值得指出的是, Java 语言中的单引号、双引号和反斜线都有特殊的用途, 如果一个字符串中包含了这些特殊字符, 则应该使用转义字符的表示形式。例如, 在 Java 程序中表示一个绝对路径: `"c:\codes"`, 但这种写法得不到期望的结果, 因为 Java 会把反斜线当成转义字符, 所以应该写成这种形式: `"c:\\codes"`, 只有同时写两个反斜线, Java 才会把第一个反斜线当成转义字符, 和后一个反斜线组成真正的反斜线。

3.4.3 浮点型

Java 的浮点类型有两种: `float` 和 `double`。Java 的浮点类型有固定的表数范围和字段长度, 字段长度和表数范围与机器无关。Java 的浮点数遵循 IEEE 754 标准, 采用二进制数据的科学计数法来表示浮点数, 对于 `float` 型数值, 第 1 位是符号位, 接下来 8 位表示指数, 再接下来的 23 位表示尾数; 对于 `double` 类型数值, 第 1 位也是符号位, 接下来的 11 位表示指数, 再接下来的 52 位表示尾数。



注意: 因为 Java 浮点数使用二进制数据的科学计数法来表示浮点数, 因此可能不能精确表示一个浮点数。例如把 5.2345556f 值赋给一个 `float` 类型变量, 接着输出这个变量时看到这个变量的值已经发生了改变。使用 `double` 类型的浮点数比 `float` 类型的浮点数更精确, 但如果浮点数的精度足够高 (小数点后的数字很多时), 依然可能发生这种情况。如果开发者需要精确保存一个浮点数, 则可以考虑使用 `BigDecimal` 类。



`double` 类型代表双精度浮点数, `float` 类型代表单精度浮点数。一个 `double` 类型的数值占 8 字节、64 位, 一个 `float` 类型的数值占 4 字节、32 位。

Java 语言的浮点数有两种表示形式。

- 十进制数形式: 这种形式就是简单的浮点数, 例如 5.12、512.0、.512。浮点数必须包含一个小数点, 否则会被当成 `int` 类型处理。
- 科学计数法形式: 例如 5.12e2 (即 5.12×10^2), 5.12E2 (也是 5.12×10^2)。

必须指出的是, 只有浮点类型的数值才可以使用科学计数法形式表示。例如, 51200 是一个 `int` 类型的值, 但 512E2 则是浮点类型的值。

Java 语言的浮点类型默认是 `double` 类型, 如果希望 Java 把一个浮点类型值当成 `float` 类型处理, 应该在这个浮点类型值后紧跟 `f` 或 `F`。例如, 5.12 代表一个 `double` 类型的值, 占 64 位的内存空间; 5.12f 或者 5.12F 才表示一个 `float` 类型的值, 占 32 位的内存空间。当然, 也可以在一个浮点数后添加 `d` 或 `D` 后缀, 强制指定是 `double` 类型, 但通常没必要。

Java 还提供了三个特殊的浮点数值: 正无穷大、负无穷大和非数, 用于表示溢出和出错。例如, 使用一个正数除以 0 将得到正无穷大, 使用一个负数除以 0 将得到负无穷大, 0.0 除以 0.0 或对一个负数开方将得到一个非数。正无穷大通过 `Double` 或 `Float` 类的 `POSITIVE_INFINITY` 表示; 负无穷大通过 `Double` 或 `Float` 类的 `NEGATIVE_INFINITY` 表示, 非数通过 `Double` 或 `Float` 类的 `NaN` 表示。

必须指出的是, 所有的正无穷大数值都是相等的, 所有的负无穷大数值都是相等的; 而 `NaN` 不与任何数值相等, 甚至和 `NaN` 都不相等。



注意: 只有浮点数除以 0 才可以得到正无穷大或负无穷大, 因为 Java 语言会自动把和浮点数运算的 0 (整数) 当成 0.0 (浮点数) 处理。如果一个整数值除以 0, 则会抛出一个异常: `ArithmeticException: / by zero` (除以 0 异常)。



下面程序示范了上面介绍的关于浮点数的各个知识点。

程序清单: codes\03\3.4\FloatTest.java

```
public class FloatTest
{
    public static void main(String[] args)
    {
        float af = 5.2345556f;
        // 下面将看到 af 的值已经发生了改变
        System.out.println(af);
        double a = 0.0;
        double c = Double.NEGATIVE_INFINITY;
        float d = Float.NEGATIVE_INFINITY;
        // 看到 float 和 double 的负无穷大是相等的
        System.out.println(c == d);
        // 0.0 除以 0.0 将出现非数
        System.out.println(a / a);
        // 两个非数之间是不相等的
        System.out.println(a / a == Float.NaN);
        // 所有正无穷大都是相等的
        System.out.println(6.0 / 0 == 555.0/0);
        // 负数除以 0.0 得到负无穷大
        System.out.println(-8 / a);
        // 下面代码将抛出除以 0 的异常
        // System.out.println(0 / 0);
    }
}
```

3.4.4 数值中使用下画线分隔

正如前面程序中看到的,当程序中用到的数值位数特别多时,程序员眼睛“看花”了都看不清到底有多少位数。为了解决这种问题,Java 7 引入了一个新功能:程序员可以在数值中使用下画线,不管是整型数值,还是浮点型数值,都可以自由地使用下画线。通过使用下画线分隔,可以更直观地分辨数值中到底包含多少位。如下面程序所示。

程序清单: codes\03\3.4\UnderscoreTest.java

```
public class UnderscoreTest
{
    public static void main(String[] args)
    {
        // 定义一个 32 位的二进制数,最高位是符号位
        int binVal = 0B1000_0000_0000_0000_0000_0000_0011;
        double pi = 3.14_15_92_65_36;
        System.out.println(binVal);
        System.out.println(pi);
        double height = 8_8_4_8.23;
        System.out.println(height);
    }
}
```

3.4.5 布尔型

布尔型只有一个 boolean 类型,用于表示逻辑上的“真”或“假”。在 Java 语言中,boolean 类型的数值只能是 true 或 false,不能用 0 或者非 0 来代表。其他基本数据类型的值也不能转换成 boolean 类型。



提示:

Java 规范并没有强制指定 boolean 类型的变量所占用的内存空间。虽然 boolean 类型的变量或值只要 1 位即可保存,但由于大部分计算机在分配内存时允许分配的最小内存单元是字节(8 位),因此 bit 大部分时候实际上占用 8 位。

例如,下面代码定义了两个 boolean 类型的变量,并指定初始值。

程序清单: codes\03\3.4\BooleanTest.java

```
// 定义b1的值为true
boolean b1 = true;
// 定义b2的值为false
boolean b2 = false;
```

字符串"true"和"false"不会直接转换成 boolean 类型,但如果使用一个 boolean 类型的值和字符串进行连接运算,则 boolean 类型的值将会自动转换成字符串。看下面代码(程序清单同上)。

```
// 使用 boolean 类型的值和字符串进行连接运算,boolean 类型的值会自动转换成字符串
String str = true + "";
// 下面将输出 true
System.out.println(str);
```

boolean 类型的值或变量主要用做旗标来进行流程控制,Java 语言中使用 boolean 类型的变量或值控制的流程主要有如下几种。

- if 条件控制语句
- while 循环控制语句
- do 循环控制语句
- for 循环控制语句

除此之外,boolean 类型的变量和值还可在三目运算符(?:)中使用。这些内容在后面将会有更详细的介绍。

3.5 基本类型的类型转换

在 Java 程序中,不同的基本类型的值经常需要进行相互转换。Java 语言所提供的 7 种数值类型之间可以相互转换,有两种类型转换方式:自动类型转换和强制类型转换。

3.5.1 自动类型转换

Java 所有的数值型变量可以相互转换,如果系统支持把某种基本类型的值直接赋给另一种基本类型的变量,则这种方式被称为自动类型转换。当把一个表数范围小的数值或变量直接赋给另一个表数范围大的变量时,系统将可以进行自动类型转换;否则就需要强制转换。

表数范围小的可以向表数范围大的进行自动类型转换,就如同有两瓶水,当把小瓶里的水倒入大瓶中时不会有任何问题。Java 支持自动类型转换的类型如图 3.10 所示。

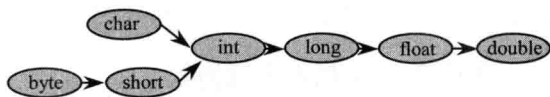


图 3.10 自动类型转换图

图 3.10 中所示的箭头左边的数值类型可以自动类型转换为箭头右边的数值类型。下面程序示范了自动类型转换。

程序清单: codes\03\3.5\AutoConversion.java

```
public class AutoConversion
{
    public static void main(String[] args)
    {
        int a = 6;
        // int 类型可以自动转换为 float 类型
        float f = a;
        // 下面将输出 6.0
        System.out.println(f);
        // 定义一个 byte 类型的整数变量
        byte b = 9;
        // 下面代码将出错, byte 类型不能自动类型转换为 char 类型
        // char c = b;
```



```

// byte 类型变量可以自动类型转换为 double 类型
double d = b;
// 下面将输出 9.0
System.out.println(d);
}

```

不仅如此,当把任何基本类型的值和字符串值进行连接运算时,基本类型的值将自动类型转换为字符串类型,虽然字符串类型不是基本类型,而是引用类型。因此,如果希望把基本类型的值转换为对应的字符串时,可以把基本类型的值和一个空字符串进行连接。



提示:

+ 不仅可作为加法运算符使用,还可作为字符串连接运算符使用。

看如下代码。

程序清单: codes\03\3.5\PrimitiveAndString.java

```

public class PrimitiveAndString
{
    public static void main(String[] args)
    {
        // 下面代码是错误的,因为 5 是一个整数,不能直接赋给一个字符串
        // String str1 = 5;
        // 一个基本类型的值和字符串进行连接运算时,基本类型的值自动转换为字符串
        String str2 = 3.5f + "";
        // 下面输出 3.5
        System.out.println(str2);
        // 下面语句输出 7Hello!
        System.out.println(3 + 4 + "Hello! ");
        // 下面语句输出 Hello!34, 因为 Hello! + 3 会把 3 当成字符串处理
        // 而后再把 4 当成字符串处理
        System.out.println("Hello! " + 3 + 4);
    }
}

```

上面程序中有一个“3 + 4 + “Hello!””表达式,这个表达式先执行“3 + 4”运算,这是执行两个整数之间的加法,得到 7,然后进行“7 + “Hello!””运算,此时会把 7 当成字符串进行处理,从而得到 7Hello!。反之,对于““Hello!” + 3 + 4”表达式,先进行““Hello!” + 3”运算,得到一个 Hello!3 字符串,再和 4 进行连接运算,4 也被转换成字符串进行处理。

3.5.2 强制类型转换

如果希望把图 3.10 中箭头右边的类型转换为左边的类型,则必须进行强制类型转换,强制类型转换的语法格式是:(targetType)value,强制类型转换的运算符是圆括号()。当进行强制类型转换时,类似于把一个大瓶子里的水倒入一个小瓶子,如果大瓶子里的水不多还好,但如果大瓶子里的水很多,将会引起溢出,从而造成数据丢失。这种转换也被称为“缩小转换(Narrow Conversion)”。

下面程序示范了强制类型转换。

程序清单: codes\03\3.5\NarrowConversion.java

```

public class NarrowConversion
{
    public static void main(String[] args)
    {
        int iValue = 233;
        // 强制把一个 int 类型的值转换为 byte 类型的值
        byte bValue = (byte)iValue;
        // 将输出-23
        System.out.println(bValue);
        double dValue = 3.98;
        // 强制把一个 double 类型的值转换为 int 类型的值
        int tol = (int)dValue;
    }
}

```

```
// 将输出 3
System.out.println(tol);
}
}
```

在上面程序中,把一个浮点数强制类型转换为整数时,Java 将直接截断浮点数的小数部分。除此之外,上面程序还把 233 强制类型转换为 byte 类型的整数,从而变成了 23,这就是典型的溢出。图 3.11 示范了这个转换过程。

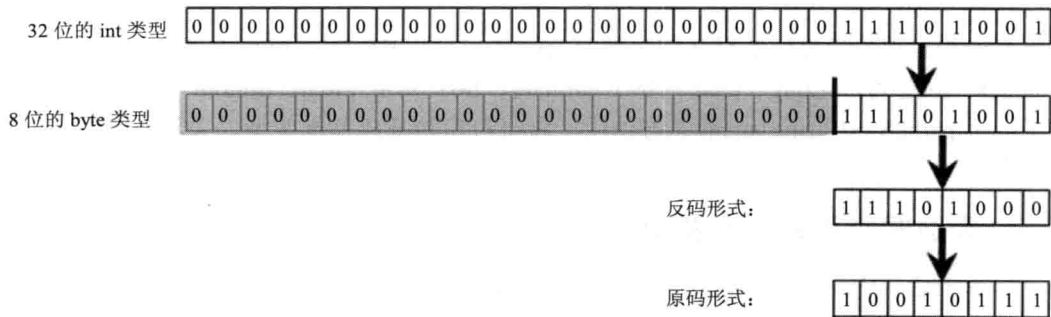


图 3.11 int 类型向 byte 类型强制类型转换

从图 3.11 可以看出,32 位 int 类型的 233 在内存中如图 3.11 上面所示,强制类型转换为 8 位的 byte 类型,则需要截断前面的 24 位,只保留右边 8 位,最左边的 1 是一个符号位,此处表明这是一个负数,负数在计算机里是以补码形式存在的,因此还需要换算成原码。

将补码减 1 得到反码形式,再将反码取反就可以得到原码。

最后的二进制原码为 10010111,这个 byte 类型的值为 $-(16 + 4 + 2 + 1)$,也就是 -23。

从图 3.11 很容易看出,当试图强制把表数范围大的类型转换为表数范围小的类型时,必须格外小心,因为非常容易引起信息丢失。

经常上网的读者可能会发现有些网页上会包含临时生成的验证字符串,那么这个随机字符串是如何生成的呢?可以先随机生成一个在指定范围内的 int 数字(如果希望生成小写字母,就在 97~122 之间),然后将其强制转换成 char 类型,再将多次生成的字符连缀起来即可。

下面程序示范了如何生成一个 6 位的随机字符串,这个程序中用到了后面的循环控制,不理解循环的读者可以参考后面章节的介绍。

程序清单: codes\03\3.5\RandomStr.java

```
public class RandomStr
{
    public static void main(String[] args)
    {
        // 定义一个空字符串
        String result = "";
        // 进行 6 次循环
        for(int i = 0 ; i < 6 ; i ++ )
        {
            // 生成一个 97~122 之间的 int 类型整数
            int intVal = (int) (Math.random() * 26 + 97);
            // 将 intValue 强制转换为 char 类型后连接到 result 后面
            result = result + (char)intVal;
        }
        // 输出随机字符串
        System.out.println(result);
    }
}
```

还有下面一行容易出错的代码:

```
// 直接把 5.6 赋值给 float 类型变量将出现错误,因为 5.6 默认是 double 类型
float a = 5.6
```

上面代码中的 5.6 默认是一个 double 类型的浮点数,因此将 5.6 赋值给一个 float 类型变量将导致

错误，必须使用强制类型转换才可以，即将上面代码改为如下形式：

```
float a = (float)5.6
```

在通常情况下，字符串不能直接转换为基本类型，但通过基本类型对应的包装类则可以实现把字符串转换成基本类型。例如，把字符串转换成 `int` 类型，则可通过如下代码实现：

```
String a = "45";
// 使用 Integer 的方法将一个字符串转换成 int 类型
int iValue = Integer.parseInt(a);
```

Java 为 8 种基本类型都提供了对应的包装类：`boolean` 对应 `Boolean`、`byte` 对应 `Byte`、`short` 对应 `Short`、`int` 对应 `Integer`、`long` 对应 `Long`、`char` 对应 `Character`、`float` 对应 `Float`、`double` 对应 `Double`，8 个包装类都提供了一个 `parseXxx(String str)` 静态方法用于将字符串转换成基本类型。关于包装类的介绍，请参考本书第 6 章。

3.5.3 表达式类型的自动提升

当一个算术表达式中包含多个基本类型的值时，整个算术表达式的数据类型将发生自动提升。Java 定义了如下的自动提升规则。

- 所有的 `byte` 类型、`short` 类型和 `char` 类型将被提升到 `int` 类型。
- 整个算术表达式的数据类型自动提升到与表达式中最高等级操作数同样的类型。操作数的等级排列如图 3.10 所示，位于箭头右边类型的等级高于位于箭头左边类型的等级。

下面程序示范了一个典型的错误。

程序清单：codes\03\3.5\AutoPromote.java

```
// 定义一个 short 类型变量
short sValue = 5;
// 表达式中的 sValue 将自动提升到 int 类型，则右边的表达式类型为 int
// 将一个 int 类型值赋给 short 类型变量将发生错误
sValue = sValue - 2;
```

上面的“`sValue - 2`”表达式的类型将被提升到 `int` 类型，这样就把右边的 `int` 类型值赋给左边的 `short` 类型变量，从而引起错误。

下面代码是表达式类型自动提升的正确示例代码（程序清单同上）。

```
byte b = 40;
char c = 'a';
int i = 23;
double d = .314;
// 右边表达式中最高等级操作数为 d (double 类型)
// 则右边表达式的类型为 double 类型，故赋给一个 double 类型变量
double result = b + c + i * d;
// 将输出 144.222
System.out.println(result);
```

必须指出，表达式的类型将严格保持和表达式中最高等级操作数相同的类型。下面代码中两个 `int` 类型整数进行除法运算，即使无法除尽，也将得到一个 `int` 类型结果（程序清单同上）。

```
int val = 3;
// 右边表达式中两个操作数都是 int 类型，故右边表达式的类型为 int
// 虽然 23/3 不能除尽，但依然得到一个 int 类型整数
int intResult = 23 / val;
System.out.println(intResult); // 将输出 7
```

从上面程序中可以看出，当两个整数进行除法运算时，如果不能整除，得到的结果将是把小数部分截断取整后的整数。

如果表达式中包含了字符串，则又是另一番情形了。因为当把加号（+）放在字符串和基本类型值之间时，这个加号是一个字符串连接运算符，而不是进行加法运算。看如下代码：

```
// 输出字符串 Hello!a7
System.out.println("Hello!" + 'a' + 7);
// 输出字符串 104Hello!
System.out.println('a' + 7 + "Hello!");
```

对于第一个表达式 “`"Hello!" + 'a' + 7`”，先进行 “`"Hello!" + 'a'`” 运算，把 `'a'` 转换成字符串，拼接成字符串 `Hello!a`，接着进行 “`"Hello!a" + 7`” 运算，这也是一个字符串连接运算，得到结果是 `Hello!a7`。对于第二个表达式，先进行 “`'a' + 7`” 加法运算，其中 `'a'` 自动提升到 `int` 类型，变成 `a` 对应的 ASCII 值：97，从 “`97 + 7`” 将得到 104，然后进行 “`104 + "Hello!"`” 运算，104 会自动转换成字符串，将变成两个字符串的连接运算，从而得到 `104Hello!`。

3.6 直接量

直接量是指在程序中通过源代码直接给出的值，例如在 `int a = 5`；这行代码中，为变量 `a` 所分配的初始值 5 就是一个直接量。

➤➤ 3.6.1 直接量的类型

并不是所有的数据类型都可以指定直接量，能指定直接量的通常只有三种类型：基本类型、字符串类型和 `null` 类型。具体而言，Java 支持如下 8 种类型的直接量。

- `int` 类型的直接量：在程序中直接给出的整型数值，可分为二进制、十进制、八进制和十六进制 4 种，其中二进制需要以 `0B` 或 `0b` 开头，八进制需要以 `0` 开头，十六进制需要以 `0x` 或 `0X` 开头。例如 123、012（对应十进制的 10）、0x12（对应十进制的 18）等。
- `long` 类型的直接量：在整型数值后添加 `l` 或 `L` 后就变成了 `long` 类型的直接量。例如 3L、0x12L（对应十进制的 18L）。
- `float` 类型的直接量：在一个浮点数后添加 `f` 或 `F` 就变成了 `float` 类型的直接量，这个浮点数可以是标准小数形式，也可以是科学计数法形式。例如 5.34F、3.14E5f。
- `double` 类型的直接量：直接给出一个标准小数形式或者科学计数法形式的浮点数就是 `double` 类型的直接量。例如 5.34、3.14E5。
- `boolean` 类型的直接量：这个类型的直接量只有 `true` 和 `false`。
- `char` 类型的直接量：`char` 类型的直接量有三种形式，分别是用单引号括起来的字符、转义字符和 Unicode 值表示的字符。例如 `'a'`、`'\n'` 和 `'\u0061'`。
- `String` 类型的直接量：一个用双引号括起来的字符序列就是 `String` 类型的直接量。
- `null` 类型的直接量：这个类型的直接量只有一个值，即 `null`。

在上面的 8 种类型的直接量中，`null` 类型是一种特殊类型，它只有一个值：`null`，而且这个直接量可以赋给任何引用类型的变量，用以表示这个引用类型变量中保存的地址为空，即还未指向任何有效对象。

➤➤ 3.6.2 直接量的赋值

通常总是把一个直接量赋值给对应类型的变量，例如下面代码都是合法的。

```
int a = 5;
char c = 'a';
boolean b = true;
float f = 5.12f;
double d = 4.12;
String author = "李刚";
String book = "疯狂 Android 讲义";
```

除此之外，Java 还支持数值之间的自动类型转换，因此允许把一个数值直接量直接赋给另一种类型的变量，这种赋值必须是系统所支持的自动类型转换，例如把 `int` 类型的直接量赋给一个 `long` 类型的变量。Java 所支持的数值之间的自动类型转换图如图 3.10 所示，箭头左边类型的直接量可以直接赋给箭

头右边类型的变量；如果需要把图 3.10 中箭头右边类型的直接量赋给箭头左边类型的变量，则需要强制类型转换。

String 类型的直接量不能赋给其他类型的变量，null 类型的直接量可以直接赋给任何引用类型的变量，包括 String 类型。boolean 类型的直接量只能赋给 boolean 类型的变量，不能赋给其他任何类型的变量。

关于字符串直接量有一点需要指出，当程序第一次使用某个字符串直接量时，Java 会使用常量池（constant pool）来缓存该字符串直接量，如果程序后面的部分需要用到该字符串直接量时，Java 会直接使用常量池（constant pool）中的字符串直接量。



提示：

由于 String 类是一个典型的不可变类，因此 String 对象创建出来就不可能被改变，因此无须担心共享 String 对象会导致混乱。关于不可变类的概念参考本书第 6 章。



提示：

常量池（constant pool）指的是在编译期被确定，并被保存在已编译的.class 文件中的一些数据。它包括关于类、方法、接口中的常量，也包括字符串直接量。

看如下程序：

```
String s0 = "hello";
String s1 = "hello";
String s2 = "he" + "llo";
System.out.println( s0 == s1 );
System.out.println( s0 == s2 );
```

运行结果为：

```
true
true
```

Java 会确保每个字符串常量只有一个，不会产生多个副本。例子中的 s0 和 s1 中的"hello"都是字符串常量，它们在编译期就被确定了，所以 s0 == s1 返回 true；而"he"和"llo"也都是字符串常量，当一个字符串由多个字符串常量连接而成时，它本身也是字符串常量，s2 同样在编译期就被解析为一个字符串常量，所以 s2 也是常量池中"hello"的引用。因此，程序输出 s0 == s1 返回 true，s1 == s2 也返回 true。



3.7 运算符

运算符是一种特殊的符号，用以表示数据的运算、赋值和比较等。Java 语言使用运算符将一个或多个操作数连缀成执行性语句，用以实现特定功能。

Java 语言中的运算符可分为如下几种。

- 算术运算符
- 赋值运算符
- 比较运算符
- 逻辑运算符
- 位运算符
- 类型相关运算符

➤➤ 3.7.1 算术运算符

Java 支持所有的基本算术运算符，这些算术运算符用于执行基本的数学运算：加、减、乘、除和求余等。下面是 7 个基本的算术运算符。

+: 加法运算符。例如如下代码：

```
double a = 5.2;
double b = 3.1;
double sum = a + b;
// sum 的值为 8.3
System.out.println(sum);
```

除此之外，+还可以作为字符串的连接运算符。

-: 减法运算符。例如如下代码：

```
double a = 5.2;
double b = 3.1;
double sub = a - b;
// sub 的值为 2.1
System.out.println(sub);
```

*: 乘法运算符。例如如下代码：

```
double a = 5.2;
double b = 3.1;
double multiply = a * b;
// multiply 的值为 16.12
System.out.println(multiply);
```

/: 除法运算符。除法运算符有些特殊，如果除法运算符的两个操作数都是整数类型，则计算结果也是整数，就是将自然除法的结果截断取整，例如 19/4 的结果是 4，而不是 5。如果除法运算符的两个操作数都是整数类型，则除数不可以是 0，否则将引发除以零异常。

但如果除法运算符的两个操作数有一个是浮点数，或者两个都是浮点数，则计算结果也是浮点数，这个结果就是自然除法的结果。而且此时允许除数是 0，或者 0.0，得到结果是正无穷大或负无穷大。看下面代码。

程序清单：codes\03\3.7\DivTest.java

```
public class DivTest
{
    public static void main(String[] args)
    {
        double a = 5.2;
        double b = 3.1;
        double div = a / b;
        // div 的值将是 1.6774193548387097
        System.out.println(div);
        // 输出正无穷大: Infinity
        System.out.println("5 除以 0.0 的结果是:" + 5 / 0.0);
        // 输出负无穷大: -Infinity
        System.out.println("-5 除以 0.0 的结果是:" + - 5 / 0.0);
        // 下面代码将出现异常
        // java.lang.ArithmeticException: / by zero
        System.out.println("-5 除以 0 的结果是:" + -5 / 0);
    }
}
```

?: 求余运算符。求余运算的结果不一定总是整数，它的计算结果是使用第一个操作数除以第二个操作数，得到一个整除的结果后剩下的值就是余数。由于求余运算也需要进行除法运算，因此如果求余运算的两个操作数都是整数类型，则求余运算的第二个操作数不能是 0，否则将引发除以零异常。如果求余运算的两个操作数中有一个或者两个都是浮点数，则允许第二个操作数是 0 或 0.0，只是求余运算的结果是非数：NaN。0 或 0.0 对零以外的任何数求余都将得到 0 或 0.0。看如下程序。

程序清单：codes\03\3.7\ModTest.java

```
public class ModTest
{
    public static void main(String[] args)
    {
        double a = 5.2;
        double b = 3.1;
        double mod = a % b;
        System.out.println(mod); // mod 的值为 2.1
    }
}
```

```
System.out.println("5 对 0.0 求余的结果是:" + 5 % 0.0); // 输出非数: NaN
System.out.println("-5.0 对 0 求余的结果是:" + -5.0 % 0); // 输出非数: NaN
System.out.println("0 对 5.0 求余的结果是:" + 0 % 5.0); // 输出 0.0
System.out.println("0 对 0.0 求余的结果是:" + 0 % 0.0); // 输出非数: NaN
// 下面代码将出现异常: java.lang.ArithmeticException: / by zero
System.out.println("-5 对 0 求余的结果是:" + -5 % 0);
```

++: 自加。该运算符有两个要点: ① 自加是单目运算符, 只能操作一个操作数; ② 自加运算符只能操作单个数值型(整型、浮点型都行)的变量, 不能操作常量或表达式。运算符既可以出现在操作数的左边, 也可以出现在操作数的右边。但出现在左边和右边的效果是不一样的。如果把++放在左边, 则先把操作数加 1, 然后才把操作数放入表达式中运算; 如果把++放在右边, 则先把操作数放入表达式中运算, 然后才把操作数加 1。看如下代码:

```
int a = 5;
// 让 a 先执行算术运算, 然后自加
int b = a++ + 6;
// 输出 a 的值为 6, b 的值为 11
System.out.println(a + "\n" + b);
```

执行完后, a 的值为 6, 而 b 的值为 11。当++在操作数右边时, 先执行 a + 6 的运算(此时 a 的值为 5), 然后对 a 加 1。对比下面代码:

```
int a = 5;
// 让 a 先自加, 然后执行算术运算
int b = ++a + 6;
// 输出 a 的值为 6, b 的值为 12
System.out.println(a + "\n" + b);
```

执行的结果是 a 的值为 6, b 的值为 12。当++在操作数左边时, 先对 a 加 1, 然后执行 a + 6 的运算(此时 a 的值为 6), 因此 b 为 12。

一: 自减。也是单目运算符, 用法与++基本相似, 只是将操作数的值减 1。



注意:

自加和自减只能用于操作变量, 不能用于操作数值直接量、常量或表达式。例如, 5++, 6--等写法都是错误的。



Java 并没有提供其他更复杂的运算符, 如果需要完成乘方、开方等运算, 则可借助于 java.lang.Math 类的工具方法完成复杂的数学运算, 见如下代码。

程序清单: codes\03\3.7\MathTest.java

```
public class MathTest
{
    public static void main(String[] args)
    {
        double a = 3.2; // 定义变量 a 为 3.2
        // 求 a 的 5 次方, 并将计算结果赋给 b
        double b = Math.pow(a, 5);
        System.out.println(b); // 输出 b 的值
        // 求 a 的平方根, 并将结果赋给 c
        double c = Math.sqrt(a);
        System.out.println(c); // 输出 c 的值
        // 计算随机数, 返回一个 0~1 之间的伪随机数
        double d = Math.random();
        System.out.println(d); // 输出随机数 d 的值
        // 求 1.57 的 sin 函数值: 1.57 被当成弧度数
        double e = Math.sin(1.57);
        System.out.println(e); // 输出接近 1
    }
}
```

Math 类下包含了丰富的静态方法，用于完成各种复杂的数学运算。



注意：

+除了可以作为数学的加法运算符之外，还可以作为字符串的连接运算符。-除了可以作为减法运算符之外，还可以作为求负的运算符。



-作为求负运算符的例子请看如下代码：

```
// 定义 double 变量 x，其值为-5.0
double x = -5.0;
x = -x; // 将 x 求负，其值变成 5.0
```

3.7.2 赋值运算符

赋值运算符用于为变量指定变量值，与 C 类似，Java 也使用=作为赋值运算符。通常，使用赋值运算符将一个直接量值赋给变量。例如如下代码。

程序清单：codes\03\3.7\AssignOperatorTest.java

```
String str = "Java"; // 为变量 str 赋值 Java
double pi = 3.14; // 为变量 pi 赋值 3.14
boolean visited = true; // 为变量 visited 赋值 true
```

除此之外，也可使用赋值运算符将一个变量的值赋给另一个变量。如下代码是正确的（程序清单同上）。

```
String str2 = str; //将变量 str 的值赋给 str2
```



提示：

按前面关于变量的介绍，可以把变量当成一个可盛装数据的容器。而赋值运算就是将“装入”变量的过程。赋值运算符是从右向左执行计算的，程序先计算得到=右边的值，然后将该值“装入”=左边的变量，因此赋值运算符(=)左边只能是变量。

值得指出的是，赋值表达式是有值的，赋值表达式的值就是右边被赋的值。例如 `String str2 = str` 表达式的值就是 `str`。因此，赋值运算符支持连续赋值，通过使用多个赋值运算符，可以一次为多个变量赋值。如下代码是正确的（程序清单同上）。

```
int a;
int b;
int c;
// 通过为 a,b,c 赋值，三个变量的值都是 7
a = b = c = 7;
// 输出三个变量的值
System.out.println(a + "\n" + b + "\n" + c);
```



注意：

虽然 Java 支持这种一次为多个变量赋值的写法，但这种写法导致程序的可读性降低，因此不推荐这样写。



赋值运算符还可用于将表达式的值赋给变量。如下代码是正确的。

```
double d1 = 12.34;
double d2 = d1 + 5; // 将表达式的值赋给 d2
System.out.println(d2); // 输出 d2 的值，将输出 17.34
```

赋值运算符还可与其他运算符结合，扩展成功能更加强大的赋值运算符，参考 3.7.4 节。

3.7.3 位运算符

Java 支持的位运算符有如下 7 个。

- &: 按位与。当两位同时为 1 时才返回 1。
- |: 按位或。只要有一位为 1 即可返回 1。
- ~: 按位非。单目运算符, 将操作数的每个位 (包括符号位) 全部取反。
- ^: 按位异或。当两位相同时返回 0, 不同时返回 1。
- <<: 左移运算符。
- >>: 右移运算符。
- >>>: 无符号右移运算符。

一般来说, 位运算符只能操作整数类型的变量或值。位运算符的运算法则如表 3.3 所示。

表 3.3 位运算符的运算法则

第一个操作数	第二个操作数	按位与	按位或	按位异或
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

按位非只需要一个操作数, 这个运算符将把操作数在计算机底层的二进制码按位 (包括符号位) 取反。如下代码测试了按位与和按位或运算的运行结果。

程序清单: codes\03\3.7\BitOperatorTest.java

```
System.out.println(5 & 9); // 将输出 1
System.out.println(5 | 9); // 将输出 13
```

程序执行的结果是: 5&9 的结果是 1, 5|9 的结果是 13。下面介绍运算原理。

5 的二进制码是 00000101 (省略了前面的 24 个 0), 而 9 的二进制码是 00001001 (省略了前面的 24 个 0)。运算过程如图 3.12 所示。

下面是按位异或和按位取反的执行代码 (程序清单同上)。

```
System.out.println(~5); // 将输出 4
System.out.println(5 ^ 9); // 将输出 12
```

程序执行 ~5 的结果是 4, 执行 5 ^ 9 的结果是 12。下面通过图 3.13 来介绍运算原理。

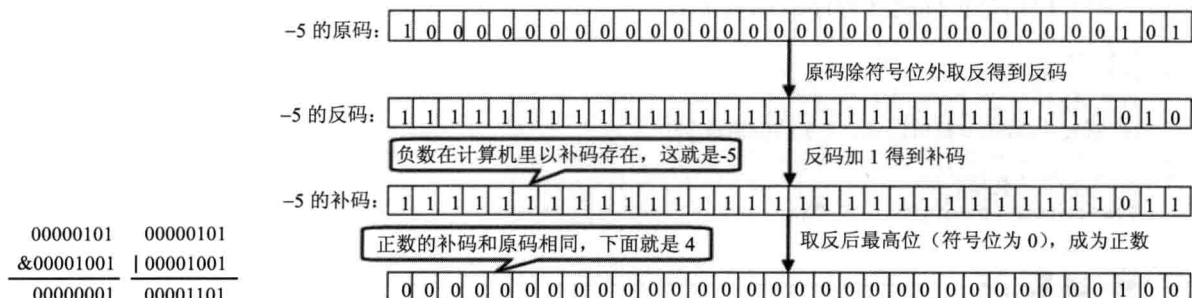


图 3.12 按位与和按位或运算过程



图 3.13 ~-5 的运算过程

而 5 ^ 9 的运算过程如图 3.14 所示。



图 3.14 5 ^ 9 的运算过程

左移运算符是将操作数的二进制码整体左移指定位数，左移后右边空出来的位以 0 填充。例如如下代码（程序清单同上）：

```
System.out.println(5 << 2); // 输出 20
System.out.println(-5 << 2); // 输出 -20
```

下面以 -5 为例来介绍左移运算的运算过程，如图 3.15 所示。

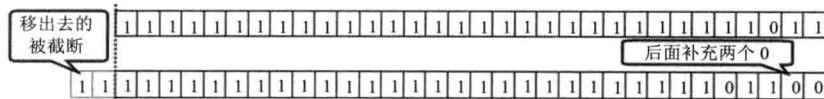


图 3.15 -5 左移两位的运算过程

在图 3.15 中，上面的 32 位数是 -5 的补码，左移两位后得到一个二进制补码，这个二进制补码的最高位是 1，表明是一个负数，换算成十进制数就是 -20。

Java 的右移运算符有两个：>>和>>>，对于>>运算符而言，把第一个操作数的二进制码右移指定位数后，左边空出来的位以原来的符号位填充，即如果第一个操作数原来是正数，则左边补 0；如果第一个操作数是负数，则左边补 1。>>>是无符号右移运算符，它把第一个操作数的二进制码右移指定位数后，左边空出来的位总是以 0 填充。

看下面代码（程序清单同上）：

```
System.out.println(-5 >> 2); // 输出 -2
System.out.println(-5 >>> 2); // 输出 1073741822
```

下面用示意图来说明>>和>>>运算符的运算过程。

从图 3.16 来看，-5 右移 2 位后左边空出 2 位，空出来的 2 位以符号位补充。从图中可以看出，右移运算后得到的结果的正负与第一个操作数的正负相同。右移后的结果依然是一个负数，这是一个二进制补码，换算成十进制数就是 -2。

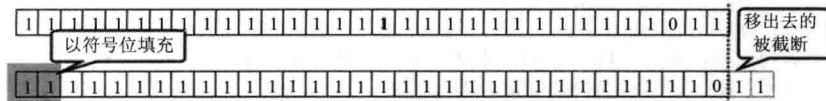


图 3.16 -5>>2 的运算过程

从图 3.17 来看，-5 无符号右移 2 位后左边空出 2 位，空出来的 2 位以 0 补充。从图中可以看出，无符号右移运算后的结果总是得到一个正数。图 3.17 中下面的正数是 1073741822 ($2^{30}-2$)。

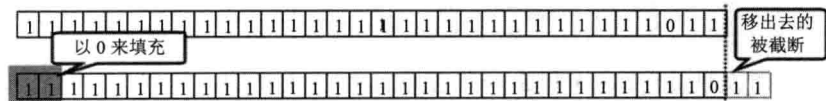


图 3.17 -5>>>2 的运算过程

进行移位运算时还要遵循如下规则。

- 对于低于 int 类型（如 byte、short 和 char）的操作数总是先自动类型转换为 int 类型后再移位。
- 对于 int 类型的整数移位 $a \gg b$ ，当 $b > 32$ 时，系统先用 b 对 32 求余（因为 int 类型只有 32 位），得到的结果才是真正移位的位数。例如， $a \gg 33$ 和 $a \gg 1$ 的结果完全一样，而 $a \gg 32$ 的结果和 a 相同。
- 对于 long 类型的整数移位 $a \gg b$ ，当 $b > 64$ 时，总是先用 b 对 64 求余（因为 long 类型是 64 位），得到的结果才是真正移位的位数。

❖ 注意 ❖

当进行移位运算时，只要被移位的二进制码没有发生有效位的数字丢失（对于正数而言，通常指被移出的位全部都是 0），不难发现左移 n 位就相当于乘以 2 的 n 次方，右移 n 位则是除以 2 的 n 次方。不仅如此，进行移位运算不会改变操作数本身，只是得到了一个新的运算结果，而原来的操作数本身是不会改变的。



3.7.4 扩展后的赋值运算符

赋值运算符可与算术运算符、位移运算符结合，扩展成功能更加强大的运算符。扩展后的赋值运算符如下。

- +=: 对于 $x += y$ ，即对应于 $x = x + y$ 。
- -=: 对于 $x -= y$ ，即对应于 $x = x - y$ 。
- *=: 对于 $x *= y$ ，即对应于 $x = x * y$ 。
- /=: 对于 $x /= y$ ，即对应于 $x = x / y$ 。
- %=: 对于 $x \% = y$ ，即对应于 $x = x \% y$ 。
- &=: 对于 $x \& = y$ ，即对应于 $x = x \& y$ 。
- |=: 对于 $x |= y$ ，即对应于 $x = x | y$ 。
- ^=: 对于 $x \wedge = y$ ，即对应于 $x = x \wedge y$ 。
- <<=: 对于 $x << = y$ ，即对应于 $x = x << y$ 。
- >>=: 对于 $x >> = y$ ，即对应于 $x = x >> y$ 。
- >>>=: 对于 $x >>> = y$ ，即对应于 $x = x >>> y$ 。

只要能使用这种扩展后的赋值运算符，通常都推荐使用它们。因为这种运算符不仅具有更好的性能，而且程序会更加健壮。下面程序示范了 += 运算符的用法。

程序清单：codes\03\3.7\EnhanceAssignTest.java

```
public class EnhanceAssignTest
{
    public static void main(String[] args)
    {
        // 定义一个 byte 类型的变量
        byte a = 5;
        // 下面语句出错，因为 5 默认是 int 类型，a + 5 就是 int 类型
        // 把 int 类型赋给 byte 类型的变量，所以出错
        // a = a + 5;
        // 定义一个 byte 类型的变量
        byte b = 5;
        // 下面语句不会出现错误
        b += 5;
    }
}
```

运行上面程序，不难发现 $a = a + 5$ 和 $a += 5$ 虽然运行结果相同，但底层的运行机制还是存在一定差异的。因此，如果可以使用这种扩展后的运算符，则推荐使用它们。

3.7.5 比较运算符

比较运算符用于判断两个变量或常量的大小，比较运算的结果是一个布尔值（true 或 false）。Java 支持的比较运算符如下。

- >: 大于，只支持左右两边操作数是数值类型。如果前面变量的值大于后面变量的值，则返回 true。
- >=: 大于等于，只支持左右两边操作数是数值类型。如果前面变量的值大于等于后面变量的值，则返回 true。
- <: 小于，只支持左右两边操作数是数值类型。如果前面变量的值小于后面变量的值，则返回 true。
- <=: 小于等于，只支持左右两边操作数是数值类型。如果前面变量的值小于等于后面变量的值，则返回 true。
- ==: 等于，如果进行比较的两个操作数都是数值类型，即使它们的数据类型不相同，只要它们的值相等，也都将返回 true。例如 $97 == 'a'$ 返回 true， $5.0 == 5$ 也返回 true。如果两个操作数都是引用类型，那么只有当两个引用变量的类型具有父子关系时才可以比较，而且这两个引用必

须指向同一个对象才会返回 true。Java 也支持两个 boolean 类型的值进行比较,例如, `true == false` 将返回 false。



注意:

基本类型的变量、值不能和引用类型的变量、值使用 `==` 进行比较; boolean 类型的变量、值不能与其他任意类型的变量、值使用 `==` 进行比较; 如果两个引用类型之间没有父子继承关系,那么它们的变量也不能使用 `==` 进行比较。



➤ `!=`: 不等于,如果进行比较的两个操作数都是数值类型,无论它们的数据类型是否相同,只要它们的值不相等,也都将返回 true。如果两个操作数都是引用类型,只有当两个引用变量的类型具有父子关系时才可以比较,只要两个引用指向的不是同一个对象就会返回 true。

下面程序示范了比较运算符的使用。

程序清单: codes\03\3.7\ComparableOperatorTest.java

```
public class ComparableOperatorTest
{
    public static void main(String[] args)
    {
        System.out.println("5 是否大于 4.0: " + (5 > 4.0)); // 输出 true
        System.out.println("5 和 5.0 是否相等: " + (5 == 5.0)); // 输出 true
        System.out.println("97 和 'a' 是否相等: " + (97 == 'a')); // 输出 true
        System.out.println("true 和 false 是否相等: " + (true == false)); // 输出 false
        // 创建 2 个 ComparableOperatorTest 对象, 分别赋给 t1 和 t2 两个引用
        ComparableOperatorTest t1 = new ComparableOperatorTest();
        ComparableOperatorTest t2 = new ComparableOperatorTest();
        // t1 和 t2 是同一个类的两个实例的引用, 所以可以比较
        // 但 t1 和 t2 引用不同的对象, 所以返回 false
        System.out.println("t1 是否等于 t2: " + (t1 == t2));
        // 直接将 t1 的值赋给 t3, 即让 t3 指向 t1 指向的对象
        ComparableOperatorTest t3 = t1;
        // t1 和 t3 指向同一个对象, 所以返回 true
        System.out.println("t1 是否等于 t3: " + (t1 == t3));
    }
}
```

值得注意的是, Java 为所有的基本数据类型都提供了对应的包装类, 关于包装类实例的比较有些特殊, 具体介绍可以参考 6.1 节。

3.7.6 逻辑运算符

逻辑运算符用于操作两个布尔型的变量或常量。逻辑运算符主要有如下 6 个。

- `&&`: 与, 前后两个操作数必须都是 true 才返回 true, 否则返回 false。
- `&`: 不短路与, 作用与 `&&` 相同, 但不会短路。
- `||`: 或, 只要两个操作数中有一个是 true, 就可以返回 true, 否则返回 false。
- `|`: 不短路或, 作用与 `||` 相同, 但不会短路。
- `!`: 非, 只需要一个操作数, 如果操作数为 true, 则返回 false; 如果操作数为 false, 则返回 true。
- `^`: 异或, 当两个操作数不同时才返回 true, 如果两个操作数相同则返回 false。

下面代码示范了或、与、非、异或 4 个逻辑运算符的执行示意。

程序清单: codes\03\3.7\LogicOperatorTest.java

```
// 直接对 false 求非运算, 将返回 true
System.out.println(!false);
// 5>3 返回 true, '6' 转换为整数 54, '6'>10 返回 true, 求与后返回 true
System.out.println(5 > 3 && '6' > 10);
```



```
// 4>=5 返回 false, 'c'>'a'返回 true。求或后返回 true
System.out.println(4 >= 5 || 'c' > 'a');
// 4>=5 返回 false, 'c'>'a'返回 true。两个不同的操作数求异或返回 true
System.out.println(4 >= 5 ^ 'c' > 'a');
```

对于|与||的区别, 参见如下代码 (程序清单同上)。

```
// 定义变量 a,b, 并为两个变量赋值
int a = 5;
int b = 10;
// 对 a > 4 和 b++ > 10 求或运算
if (a > 4 | b++ > 10)
{
    // 输出 a 的值是 5, b 的值是 11
    System.out.println("a 的值是:" + a + ", b 的值是:" + b);
}
```

执行上面程序, 看到输出 a 的值为 5, b 的值为 11, 这表明 b++ > 10 表达式得到了计算, 但实际上没有计算的必要, 因为 a > 4 已经返回了 true, 则整个表达式一定返回 true。

再看如下代码, 只是将上面示例的不短路逻辑或改成了短路逻辑或 (程序清单同上)。

```
// 定义变量 c,d, 并为两个变量赋值
int c = 5;
int d = 10;
// c > 4 || d++ > 10 求或运算
if (c > 4 || d++ > 10)
{
    // 输出 c 的值是 5, d 的值是 10
    System.out.println("c 的值是:" + c + ", d 的值是:" + d);
}
```

上面代码执行的结果是: c 的值为 5, 而 d 的值为 10。

对比两段代码, 后面的代码仅仅将不短路或改成短路或, 程序最后输出的 d 值不再是 11, 这表明表达式 d++ > 10 没有获得执行的机会。因为对于短路逻辑或||而言, 如果第一个操作数返回 true, || 将不再对第二个操作数求值, 直接返回 true。不会计算 d++ > 10 这个逻辑表达式, 因而 d++没有获得执行的机会。因此, 最后输出的 d 值为 10。而不短路或 | 总是执行前后两个操作数。

&与&&的区别与此类似: &总会计算前后两个操作数, 而&&先计算左边的操作数, 如果左边的操作数为 false, 则直接返回 false, 根本不会计算右边的操作数。

▶▶ 3.7.7 三目运算符

三目运算符只有一个: ? :, 三目运算符的语法格式如下:

```
(expression) ? if-true-statement : if-false-statement;
```

三目运算符的规则是: 先对逻辑表达式 expression 求值, 如果逻辑表达式返回 true, 则返回第二个操作数的值, 如果逻辑表达式返回 false, 则返回第三个操作数的值。看如下代码。

程序清单: codes\03\3.7\ThreeTest.java

```
String str = 5 > 3 ? "5 大于 3" : "5 不大于 3";
System.out.println(str); // 输出"5 大于 3"
```

大部分时候, 三目运算符都是作为 if else 的精简写法。因此, 如果将上面代码换成 if else 的写法, 则代码如下 (程序清单同上)。

```
String str2 = null;
if (5 > 3)
{
    str2 = "5 大于 3";
}
else
{
}
```

```

        str2 = "5 不大于 3";
    }

```

这两种代码写法的效果是完全相同的。三目运算符和 if else 写法的区别在于：if 后的代码块可以有多个语句，但三目运算符是不支持多个语句的。

三目运算符可以嵌套，嵌套后的三目运算符可以处理更复杂的情况，如下程序所示(程序清单同上)。

```

int a = 11;
int b = 12;
// 三目运算符支持嵌套
System.out.println(a > b ?
    "a 大于 b" : (a < b ? "a 小于 b" : "a 等于 b"));

```

上面程序中粗体字代码是一个由三目运算符构成的表达式，这个表达式本身又被嵌套在三目运算符中。通过使用嵌套的三目运算符，即可让三目运算符处理更复杂的情况。

3.7.8 运算符的结合性和优先级

所有的数学运算都认为是从左向右运算的，Java 语言中大部分运算符也是从左向右结合的，只有单目运算符、赋值运算符和三目运算符例外，其中，单目运算符、赋值运算符和三目运算符是从右向左结合的，也就是从右向左运算。

乘法和加法是两个可结合的运算，也就是说，这两个运算符左右两边的操作数可以互换位置而不会影响结果。

运算符有不同的优先级，所谓优先级就是在表达式运算中的运算顺序。表 3.4 列出了包括分隔符在内的所有运算符的优先级顺序，上一行中的运算符总是优先于下一行的。

表 3.4 运算符优先级

运算符说明	Java 运算符
分隔符	. [] {} , ;
单目运算符	++ -- ~ !
强制类型转换运算符	(type)
乘法/除法/求余	* / %
加法/减法	+ -
移位运算符	<< >> >>>
关系运算符	< <= >= > instanceof
等价运算符	== !=
按位与	&
按位异或	^
按位或	
条件与	&&
条件或	
三目运算符	?:
赋值	= += -= *= /= &= = ^= %= <<= >>= >>>=

根据表 3.4 中运算符的优先级，下面分析一下 `int a = 3; int b = a + 2 * a` 语句的执行过程。程序先执行 `2 * a` 得到 6，再执行 `a + 6` 得到 9。如果使用 `()` 就可以改变程序的执行顺序，例如 `int b = (a + 2) * a`，则先执行 `a + 2` 得到结果 5，再执行 `5 * a` 得到 15。

在表 3.4 中还提到了两个类型相关的运算符 `instanceof` 和 `(type)`，这两个运算符与类、继承有关，此处不作介绍，在第 5 章将有更详细的介绍。

因为 Java 运算符存在这种优先级的关系，因此经常看到有些学生在做 SCJP，或者某些公司的面试题，有如下 Java 代码：`int a = 5; int b = 4; int c = a++ - --b * ++a / b-- >> 2 % a--`，c 的值是多少？这样

的语句实在太恐怖了，即使多年的老程序员看到这样的语句也会眩晕。

这样的代码只能在考试中出现，如果笔者带过的 team 里有 member 写这样的代码，恐怕他马上就得走人了，因为他完全不懂程序开发：源代码就是一份文档，源代码的可读性比代码运行效率更重要。因此在这里要提醒读者：

- 不要把一个表达式写得过于复杂，如果一个表达式过于复杂，则把它分成几步来完成；
- 不要过多地依赖运算符的优先级来控制表达式的执行顺序，这样可读性太差，尽量使用()来控制表达式的执行顺序。



提示：

有些学员喜欢做一些千奇百怪的 Java 题目，例如刚刚提到的题目，还有如“在 &abc、_、\$xx、!abc 中，哪几个标识符是合法的标识符？”，这也是一个相当糟糕的题目。实际上在写一个 Java 程序时，根本不允许使用这些千奇百怪的标识符！

想起一个寓言：有人问一个有多年航海经验的船长，这条航线的暗礁你都非常清楚吧？船长的回答是：我不知道，我只知道哪里是深水航线。这是很有哲理的故事，它告诉我们写程序时，尽量采用良好的编码风格，养成良好的习惯；不要随心所欲地乱写，不要把所有的错误都犯完！世界上对的路可能只有一条，错的路却可能有成千上万条，不要成为别人的前车之鉴！

国内的编程者与国外的编程者有一个很大的差别，国外的编程者往往关心我能写什么程序？而国内的编程者往往更关心我能考什么证书？特别是一些大学生，非常热衷于考证！有时候很想告诉他们：你们的大学毕业证是国家教育部发的，难道还不够好吗？为什么还要去考一些杂七杂八的证？因为有人要考证，所以就会出现这些乱七八糟的 Java 考题。请大家记住学习编程的最终目的：是用来编写程序解决实际问题，而不是用来考证的。

3.8 本章小结

本章详细介绍了 Java 语言的各种基础知识，包括 Java 代码的三种注释语法，并讲解了如何查阅 JDK API 文档，这是学习 Java 编程必须掌握的基本技能。本章讲解了 Java 程序的标识符规则和数据类型的相关知识，包括基本类型的强制类型转换和自动类型转换。除此之外，本章还详细介绍了 Java 语言提供的各种运算符，包括算术、位、赋值、比较、逻辑等常用运算符，并详细列出了各种运算符的结合性和优先级。

本章练习

1. 定义学生、老师、教室三个类，为三个类编写文档注释，并使用 javadoc 工具来生成 API 文档。
2. 使用 8 种基本数据类型声明多个变量，并使用不同方式为 8 种基本类型的变量赋值，熟悉每种数据类型的赋值规则和表示方式。
3. 在数值型的变量之间进行类型转换，包括低位向高位的自动转换、高位向低位的强制转换。
4. 使用数学运算符、逻辑运算符编写 40 个表达式，先自行计算各表达式的值，然后通过程序输出这些表达式的值进行对比，看看能否做到一切尽在掌握。

第4章

流程控制与数组

本章要点

- 顺序结构
- if 分支语句
- switch 分支语句
- while 循环
- do while 循环
- for 循环
- 嵌套循环
- 控制循环结构
- 理解数组
- 数组的定义和初始化
- 使用数组元素
- 数组作为引用类型的运行机制
- 多维数组的实质
- 操作数组的工具类
- 数组的实际应用场景

不论哪一种编程语言，都会提供两种基本的流程控制结构：分支结构和循环结构。其中分支结构用于实现根据条件来选择性地执行某段代码，循环结构则用于实现根据循环条件重复执行某段代码。Java 同样提供了这两种流程控制结构的语法，Java 提供了 `if` 和 `switch` 两种分支语句，并提供了 `while`、`do while` 和 `for` 三种循环语句。除此之外，JDK 5 还提供了一种新的循环：`foreach` 循环，能以更简单的方式来遍历集合、数组的元素。Java 还提供了 `break` 和 `continue` 来控制程序的循环结构。

数组也是大部分编程语言都支持的数据结构，Java 也不例外。Java 的数组类型是一种引用类型的变量，Java 程序通过数组引用变量来操作数组，包括获得数组的长度，访问数组元素的值等。本章将会详细介绍 Java 数组的相关知识，包括如何定义、初始化数组等基础知识，并会深入介绍数组在内存中的运行机制。

4.1 顺序结构

任何编程语言中最常见的程序结构就是顺序结构。顺序结构就是程序从上到下逐行地执行，中间没有任何判断和跳转。

如果 `main` 方法的多行代码之间没有任何流程控制，则程序总是从上向下依次执行，排在前面的代码先执行，排在后面的代码后执行。这意味着：如果没有流程控制，Java 方法里的语句是一个顺序执行流，从上向下依次执行每条语句。

4.2 分支结构

Java 提供了两种常见的分支控制结构：`if` 语句和 `switch` 语句，其中 `if` 语句使用布尔表达式或布尔值作为分支条件来进行分支控制；而 `switch` 语句则用于对多个整型值进行匹配，从而实现分支控制。

▶▶ 4.2.1 if 条件语句

`if` 语句使用布尔表达式或布尔值作为分支条件来进行分支控制。`if` 语句有如下三种形式。

第一种形式：

```
if ( logic expression )
{
    statement...
}
```

第二种形式：

```
if (logic expression)
{
    statement...
}
else
{
    statement...
}
```

第三种形式：

```
if (logic expression)
{
    statement...
}
else if(logic expression)
{
    statement...
}
...// 可以有零个或多个 else if 语句
else// 最后的 else 语句也可以省略
{
    statement...
}
```

在上面 if 语句的三种形式中, 放在 if 之后括号里的只能是一个逻辑表达式, 即这个表达式的返回值只能是 true 或 false。第二种形式和第三种形式是相通的, 如果第三种形式中 else if 块不出现, 就变成了第二种形式。

在上面的条件语句中, if(logic expression)、else if(logic expression)和 else 后花括号括起来的多行代码被称为代码块, 一个代码块通常被当成一个整体来执行 (除非运行过程中遇到 return、break、continue 等关键字, 或者遇到了异常), 因此这个代码块也被称为条件执行体。例如如下程序。

程序清单: codes\04\4.2\IfTest.java

```
public class IfTest
{
    public static void main(String[] args)
    {
        int age = 30;
        if (age > 20)
            // 只有当 age > 20 时, 下面花括号括起来的代码块才会执行
            // 花括号括起来的语句是一个整体, 要么一起执行, 要么一起不执行
            {
                System.out.println("年龄已经大于 20 岁了");
                System.out.println("20 岁以上的人应该学会承担责任...");
            }
    }
}
```

如果 if(logic expression)、else if(logic expression)和 else 后的代码块只有一行语句时, 则可以省略花括号, 因为单行语句本身就是一个整体, 无须用花括号来把它们定义成一个整体。下面代码完全可以正常执行 (程序清单同上)。

```
// 定义变量 a, 并为其赋值
int a = 5;
if (a > 4)
    // 如果 a>4, 则执行下面的执行体, 只有一行代码作为代码块
    System.out.println("a 大于 4");
else
    // 否则, 执行下面的执行体, 只有一行代码作为代码块
    System.out.println("a 不大于 4");
```

通常建议不要省略 if、else、else if 后执行体的花括号, 即使条件执行体只有一行代码, 也保留花括号会有更好的可读性, 而且保留花括号会减少发生错误的可能。例如如下代码, 则不能正常执行 (程序清单同上)。

```
//定义变量 b, 并为其赋值
int b = 5;
if (b > 4)
    // 如果 b>4, 则执行下面的执行体, 只有一行代码作为代码块
    System.out.println("b 大于 4");
else
    // 否则, 执行下面的执行体, 只有一行代码作为代码块
    b--;
    //对于下面代码而言, 它已经不再是条件执行体的一部分, 因此总会执行
    System.out.println("b 不大于 4");
```

上面代码中以粗体字标识的代码行: System.out.println("b 不大于 4"); 总会执行, 因为这行代码并不属于 else 后的条件执行体, else 后的条件执行体就是 b--;这行代码。

❖ 注意: ❖

if、else、else if 后的条件执行体要么是一个花括号括起来的代码块, 则这个代码块整体作为条件执行体; 要么是以分号为结束符的一行语句, 甚至可能是一个空语句 (空语句是一个分号), 那么就只是这条语句作为条件执行体。如果省略了 if 条件后条件执行体的花括号, 那么 if 条件只控制到紧跟该条件语句的第一个分号处。



如果 if 后有多条语句作为条件执行体，若省略了这个条件执行体的花括号，则会引起编译错误。看下面代码（程序清单同上）：

```
// 定义变量 c，并为其赋值
int c = 5;
if (c > 4)
// 如果 c>4，则执行下面的执行体，将只有 c--; 一行代码为执行体
    c--;
// 下面是一行普通代码，不属于执行体
    System.out.println("c 大于 4");
// 此处的 else 将没有 if 语句，因此编译出错
else
// 否则，执行下面的执行体，只有一行代码作为代码块
    System.out.println("c 不大于 4");
```

在上面代码中，因为 if 后的条件执行体省略了花括号，则系统只把 c--; 一行代码作为条件执行体，当 c--; 语句结束后，if 语句也就结束了。后面的 System.out.println("c 大于 4"); 代码已经是一行普通代码了，不再属于条件执行体，从而导致 else 语句没有 if 语句，从而引起编译错误。

对于 if 语句，还有一个很容易出现的逻辑错误，这个逻辑错误并不属于语法问题，但引起错误的的可能性更大。看下面程序。

程序清单：codes\04\4.2\IfErrorTest.java

```
public class IfErrorTest
{
    public static void main(String[] args)
    {
        int age = 45;
        if (age > 20)
        {
            System.out.println("青年人");
        }
        else if (age > 40)
        {
            System.out.println("中年人");
        }
        else if (age > 60)
        {
            System.out.println("老年人");
        }
    }
}
```

表面上看起来，上面的程序没有任何问题：人的年龄大于 20 岁是青年人，年龄大于 40 岁是中年人，年龄大于 60 岁是老年人。但运行上面程序，发现打印结果是：青年人，而实际上希望 45 岁应判断为中年人——这显然出现了一个问题。

对于任何的 if else 语句，表面上看起来 else 后没有任何条件，或者 else if 后只有一个条件——但这不是真相：因为 else 的含义是“否则”——else 本身就是一个条件！这也是把 if、else 后代码块统称为条件执行体的原因，else 的隐含条件是对前面条件取反。因此，上面代码实际上可改写为如下形式。

程序清单：codes\04\4.2\IfErrorTest2.java

```
public class IfErrorTest2
{
    public static void main(String[] args)
    {
        int age = 45;
        if (age > 20)
        {
            System.out.println("青年人");
        }
        // 在原本的 if 条件中增加了 else 的隐含条件
        if (age > 40 && !(age > 20))
        {
```



```

        System.out.println("中年人");
    }
    // 在原本的 if 条件中增加了 else 的隐含条件
    if (age > 60 && !(age > 20) && !(age > 40 && !(age > 20)))
    {
        System.out.println("老年人");
    }
}

```

此时就比较容易看出为什么发生上面的错误了。对于 `age > 40 && !(age > 20)` 这个条件，又可改写成 `age > 40 && age <= 20`，这样永远也不会发生了。对于 `age > 60 && !(age > 20) && !(age > 40 && !(age > 20))` 这个条件，则更不可能发生了。因此，程序永远都不会判断中年人和老年人的情形。

为了达到正确的目的，可以把程序改为如下形式。

程序清单：codes\04\4.2\IfCorrectTest.java

```

public class IfCorrectTest
{
    public static void main(String[] args)
    {
        int age = 45;
        if (age > 60)
        {
            System.out.println("老年人");
        }
        else if (age > 40)
        {
            System.out.println("中年人");
        }
        else if (age > 20)
        {
            System.out.println("青年人");
        }
    }
}

```

运行程序，得到了正确结果。实际上，上面程序等同于下面代码。

程序清单：codes\04\4.2\IfCorrectTest2.java

```

public class TestIfCorrect2
{
    public static void main(String[] args)
    {
        int age = 45;
        if (age > 60)
        {
            System.out.println("老年人");
        }
        // 在原本的 if 条件中增加了 else 的隐含条件
        if (age > 40 && !(age > 60))
        {
            System.out.println("中年人");
        }
        // 在原本的 if 条件中增加了 else 的隐含条件
        if (age > 20 && !(age > 60) && !(age > 40 && !(age > 60)))
        {
            System.out.println("青年人");
        }
    }
}

```

上面程序的判断逻辑即转为如下三种情形。

- age 大于 60 岁，判断为“老年人”。
- age 大于 40 岁，且 age 小于等于 60 岁，判断为“中年人”。
- age 大于 20 岁，且 age 小于等于 40 岁，判断为“青年人”。

上面的判断逻辑才是实际希望的判断逻辑。因此，当使用 `if...else` 语句进行流程控制时，一定不要忽略了 `else` 所带的隐含条件。

如果每次都去计算 `if` 条件和 `else` 条件的交集也是一件非常烦琐的事情，为了避免出现上面的错误，在使用 `if...else` 语句时有一条基本规则：总是优先把包含范围小的条件放在前面处理。如 `age>60` 和 `age>20` 两个条件，明显 `age>60` 的范围更小，所以应该先处理 `age>60` 的情况。



注意：

使用 `if...else` 语句时，一定要先处理包含范围更小的情况。



4.2.2 Java 7 增强后的 switch 分支语句

`switch` 语句由一个控制表达式和多个 `case` 标签组成，和 `if` 语句不同的是，`switch` 语句后面的控制表达式的数据类型只能是 `byte`、`short`、`char`、`int` 四种整数类型，枚举类型和 `java.lang.String` 类型（从 Java 7 才允许），不能是 `boolean` 类型。

`switch` 语句往往需要在 `case` 标签后紧跟一个代码块，`case` 标签作为这个代码块的标识。`switch` 语句的语法格式如下：

```
switch (expression)
{
    case condition1:
    {
        statement(s)
        break;
    }
    case condition2:
    {
        statement(s)
        break;
    }
    ...
    case conditionN:
    {
        statement(s)
        break;
    }
    default:
    {
        statement(s)
    }
}
```

这种分支语句的执行是先对 `expression` 求值，然后依次匹配 `condition1`、`condition2`、...、`conditionN` 等值，遇到匹配的值即执行对应的执行体；如果所有 `case` 标签后的值都不与 `expression` 表达式的值相等，则执行 `default` 标签后的代码块。

和 `if` 语句不同的是，`switch` 语句中各 `case` 标签后代码块的开始点和结束点非常清晰，因此完全可以省略 `case` 后代码块的花括号。与 `if` 语句中的 `else` 类似，`switch` 语句中的 `default` 标签看似没有条件，其实是有条件的，条件就是 `expression` 表达式的值不能与前面任何一个 `case` 标签后的值相等。

下面程序示范了 `switch` 语句的用法。

程序清单：codes\04\4.2\SwitchTest.java

```
public class SwitchTest
{
    public static void main(String[] args)
    {
        // 声明变量 score，并为其赋值为'C'
        char score = 'C';
        // 执行 switch 分支语句
        switch (score)
        {
            case 'A':
```

```

        System.out.println("优秀");
        break;
    case 'B':
        System.out.println("良好");
        break;
    case 'C':
        System.out.println("中");
        break;
    case 'D':
        System.out.println("及格");
        break;
    case 'F':
        System.out.println("不及格");
        break;
    default:
        System.out.println("成绩输入错误");
    }
}
}

```

运行上面程序，看到输出“中”，这个结果完全正常，字符表达式 `score` 的值为'C'，对应结果为“中”。

在 `case` 标签后的每个代码块后都有一条 `break;` 语句，这个 `break;` 语句有极其重要的意义，Java 的 `switch` 语句允许 `case` 后代码块没有 `break;` 语句，但这种做法可能引入一个陷阱。如果把上面程序中的 `break;` 语句都注释掉，将看到如下运行结果：

```

中
及格
不及格
成绩输入错误

```

这个运行结果看起来比较奇怪，但这正是由 `switch` 语句的运行流程决定的：`switch` 语句会先求出 `expression` 表达式的值，然后拿这个表达式和 `case` 标签后的值进行比较，一旦遇到相等的值，程序就开始执行这个 `case` 标签后的代码，不再判断与后面 `case`、`default` 标签的条件是否匹配，除非遇到 `break;` 才会结束。

Java 7 增强了 `switch` 语句的功能，允许 `switch` 语句的控制表达式是 `java.lang.String` 类型的变量或表达式——只能是 `java.lang.String` 类型，不能是 `StringBuffer` 或 `StringBuilder` 这两种字符串类型。

如下程序也是正确的。

程序清单：codes\04\4.2\StringSwitchTest.java

```

public class StringSwitchTest
{
    public static void main(String[] args)
    {
        // 声明变量 season
        String season = "夏天";
        // 执行 switch 分支语句
        switch (season)
        {
            case "春天":
                System.out.println("春暖花开.");
                break;
            case "夏天":
                System.out.println("夏日炎炎.");
                break;
            case "秋天":
                System.out.println("秋高气爽.");
                break;
            case "冬天":
                System.out.println("冬雪皑皑.");
                break;
            default:

```

```
System.out.println("季节输入错误");
```

❗ 注意: ❗

使用 switch 语句时,有两个值得注意的地方:第一个地方是 switch 语句后的 expression 表达式的数据类型只能是 byte、short、char、int 四种整数类型, String (Java 7 才支持) 和枚举类型;第二个地方是如果省略了 case 后代码块的 break;, 将引入一个陷阱。



📁 4.3 循环结构

循环语句可以在满足循环条件的情况下,反复执行某一段代码,这段被重复执行的代码被称为循环体。当反复执行这个循环体时,需要在合适的时候把循环条件改为假,从而结束循环,否则循环将一直执行下去,形成死循环。循环语句可能包含如下 4 个部分。

- 初始化语句 (init_statement): 一条或多条语句,这些语句用于完成一些初始化工作,初始化语句在循环开始之前执行。
- 循环条件 (test_expression): 这是一个 boolean 表达式,这个表达式能决定是否执行循环体。
- 循环体 (body_statement): 这个部分是循环的主体,如果循环条件允许,这个代码块将被重复执行。如果这个代码块只有一行语句,则这个代码块的花括号是可以省略的。
- 迭代语句 (iteration_statement): 这个部分在一次循环体执行结束后,对循环条件求值之前执行,通常用于控制循环条件中的变量,使得循环在合适的时候结束。

上面 4 个部分只是一般性的分类,并不是每个循环中都非常清晰地分出了这 4 个部分。

➤➤ 4.3.1 while 循环语句

while 循环的语法格式如下:

```
[init_statement]
while(test_expression)
{
    statement;
    [iteration_statement]
}
```

while 循环每次执行循环体之前,先对 test_expression 循环条件求值,如果循环条件为 true,则运行循环体部分。从上面的语法格式来看,迭代语句 iteration_statement 总是位于循环体的最后,因此只有当循环体能成功执行完成时,while 循环才会执行 iteration_statement 迭代语句。

从这个意义上来看,while 循环也可被当成条件语句——如果 test_expression 条件一开始就为 false,则循环体部分将永远不会获得执行。

下面程序示范了一个简单的 while 循环。

程序清单: codes\04\4.3\WhileTest.java

```
public class WhileTest
{
    public static void main(String[] args)
    {
        // 循环的初始化条件
        int count = 0;
        // 当 count 小于 10 时,执行循环体
        while (count < 10)
        {
            System.out.println(count);
            // 迭代语句
            count++;
        }
        System.out.println("循环结束!");
    }
}
```


如果 while 循环的循环体部分和迭代语句合并在一起,且只有一行代码,则可以省略 while 循环后的花括号。但这种省略花括号的做法,可能降低程序的可读性。



注意：

如果省略了循环体的花括号,那么 while 循环条件仅控制到紧跟该循环条件的第一个分号处。



使用 while 循环时,一定要保证循环条件有变成 false 的时候,否则这个循环将成为一个死循环,永远无法结束这个循环。例如如下代码(程序清单同上):

```
// 下面是一个死循环
int count = 0;
while (count < 10)
{
    System.out.println("不停执行的死循环 " + count);
    count--;
}
System.out.println("永远无法跳出的循环体");
```

在上面代码中,count 的值越来越小,这将导致 count 值永远小于 10,count < 10 循环条件一直为 true,从而导致这个循环永远无法结束。

除此之外,对于许多初学者而言,使用 while 循环时还有一个陷阱:while 循环的循环条件后紧跟一个分号。比如有如下程序片段(程序清单同上):

```
int count = 0;
// while 后紧跟一个分号,表明循环体是一个分号(空语句)
while (count < 10);
// 下面的代码块与 while 循环已经没有任何关系
{
    System.out.println("-----" + count);
    count++;
}
```

乍一看,这段代码片段没有任何问题,但仔细看一下这个程序,不难发现 while 循环的循环条件表达式后紧跟了一个分号。在 Java 程序中,一个单独的分号表示一个空语句,不做任何事情的空语句,这意味着这个 while 循环的循环体是空语句。空语句作为循环体也不是最大的问题,问题是当 Java 反复执行这个循环体时,循环条件的返回值没有任何改变,这就成了一个死循环。分号后面的代码块则与 while 循环没有任何关系。

4.3.2 do while 循环语句

do while 循环与 while 循环的区别在于:while 循环是先判断循环条件,如果条件为真则执行循环体;而 do while 循环则先执行循环体,然后才判断循环条件,如果循环条件为真,则执行下一次循环,否则中止循环。do while 循环的语法格式如下:

```
[init_statement]
do
{
    statement;
    [iteration_statement]
}while (test_expression);
```

与 while 循环不同的是,do while 循环的循环条件后必须有一个分号,这个分号表明循环结束。下面程序示范了 do while 循环的用法。

程序清单: codes\04\4.3\DoWhileTest.java

```
public class DoWhileTest
{
    public static void main(String[] args)
    {
```



```
// 定义变量 count
int count = 1;
// 执行 do while 循环
do
{
    System.out.println(count);
    // 循环迭代语句
    count++;
    // 循环条件紧跟 while 关键字
}while (count < 10);
System.out.println("循环结束!");
}
```

即使 `test_expression` 循环条件的值开始就是假, `do while` 循环也会执行循环体。因此, `do while` 循环的循环体至少执行一次。下面的代码片段验证了这个结论(程序清单同上)。

```
// 定义变量 count2
int count2 = 20;
// 执行 do while 循环
do
    // 这行代码把循环体和迭代部分合并成了一行代码
    System.out.println(count2++);
while (count2 < 10);
System.out.println("循环结束!");
```

从上面程序来看, 虽然开始 `count2` 的值就是 20, `count2 < 10` 表达式返回 `false`, 但 `do while` 循环还是会把循环体执行一次。

4.3.3 for 循环

`for` 循环是更加简洁的循环语句, 大部分情况下, `for` 循环可以代替 `while` 循环、`do while` 循环。`for` 循环的基本语法格式如下:

```
for ([init_statement]; [test_expression]; [iteration_statement])
{
    statement
}
```

程序执行 `for` 循环时, 先执行循环的初始化语句 `init_statement`, 初始化语句只在循环开始前执行一次。每次执行循环体之前, 先计算 `test_expression` 循环条件的值, 如果循环条件返回 `true`, 则执行循环体, 循环体执行结束后执行循环迭代语句。因此, 对于 `for` 循环而言, 循环条件总比循环体要多执行一次, 因为最后一次执行循环条件返回 `false`, 将不再执行循环体。

值得指出的是, `for` 循环的循环迭代语句并没有与循环体放在一起, 因此即使在执行循环体时遇到 `continue` 语句结束本次循环, 循环迭代语句也一样会得到执行。

✱ 注意: ✱

`for` 循环和 `while`、`do while` 循环不一样: 由于 `while`、`do while` 循环的循环迭代语句紧跟着循环体, 因此如果循环体不能完全执行, 如使用 `continue` 语句来结束本次循环, 则循环迭代语句不会被执行。但 `for` 循环的循环迭代语句并没有与循环体放在一起, 因此不管是否使用 `continue` 语句来结束本次循环, 循环迭代语句一样会获得执行。



与前面循环类似的是, 如果循环体只有一行语句, 那么循环体的花括号可以省略。下面使用 `for` 循环代替前面的 `while` 循环, 代码如下。

程序清单: codes\04\4.3\ForTest.java

```
public class ForTest
{
    public static void main(String[] args)
    {
        // 循环的初始化条件、循环条件、循环迭代语句都在下面一行
    }
}
```

```
for (int count = 0 ; count < 10 ; count++)
{
    System.out.println(count);
}
System.out.println("循环结束!");
}
```

在上面的循环语句中，for 循环的初始化语句只有一个，循环条件也只是一个简单的 boolean 表达式。实际上，for 循环允许同时指定多个初始化语句，循环条件也可以是一个包含逻辑运算符的表达式。例如如下程序：

程序清单：codes\04\4.3\ForTest2.java

```
public class ForTest2
{
    public static void main(String[] args)
    {
        // 同时定义了三个初始化变量，使用&&来组合多个 boolean 表达式
        for (int b = 0, s = 0, p = 0
            ; b < 10 && s < 4 && p < 10; p++)
        {
            System.out.println(b++);
            System.out.println(++s + p);
        }
    }
}
```

上面代码中初始化变量有三个，但是只能有一个声明语句，因此如果需要在初始化表达式中声明多个变量，那么这些变量应该具有相同的数据类型。

初学者使用 for 循环时也容易犯一个错误，他们以为只要在 for 后的括号内控制了循环迭代语句就万无一失，但实际情况则不是这样的。例如下面的程序：

程序清单：codes\04\4.3\ForErrorTest.java

```
public class ForErrorTest
{
    public static void main(String[] args)
    {
        // 循环的初始化条件、循环条件、循环迭代语句都在下面一行
        for (int count = 0 ; count < 10 ; count++)
        {
            System.out.println(count);
            // 再次修改了循环变量
            count *= 0.1;
        }
        System.out.println("循环结束!");
    }
}
```

在上面的 for 循环中，表面上看起来控制了 count 变量的自加，count < 10 有变成 false 的时候。但实际上程序中粗体字标识的代码行在循环体内修改了 count 变量的值，并且把这个变量的值乘以了 0.1，这也会导致 count 的值永远都不能超过 10，因此上面程序也是一个死循环。

❖ 注意：❖

建议不要在循环体内修改循环变量（也叫循环计数器）的值，否则会增加程序出错的可能性。万一程序真的需要访问、修改循环变量的值，建议重新定义一个临时变量，先将循环变量的值赋给临时变量，然后对临时变量的值进行修改。



for 循环圆括号中只有两个分号是必需的，初始化语句、循环条件、迭代语句部分都是可以省略的，如果省略了循环条件，则这个循环条件默认为 true，将会产生一个死循环。例如下面程序。

程序清单: codes\04\4.3\DeadForTest.java

```
public class DeadForTest
{
    public static void main(String[] args)
    {
        // 省略了 for 循环三个部分, 循环条件将一直为 true
        for ( ; ; )
        {
            System.out.println("=====");
        }
    }
}
```

运行上面程序, 将看到程序一直输出=====字符串, 这表明此程序是一个死循环。

使用 for 循环时, 还可以把初始化条件定义在循环体之外, 把循环迭代语句放在循环体内, 这种做法就非常类似于前面的 while 循环了。下面的程序再次使用 for 循环来代替前面的 while 循环。

程序清单: codes\04\4.3\ForInsteadWhile.java

```
public class ForInsteadWhile
{
    public static void main(String[] args)
    {
        // 把 for 循环的初始化条件提出来独立定义
        int count = 0;
        // for 循环里只放循环条件
        for ( ; count < 10 ; )
        {
            System.out.println(count);
            // 把循环迭代部分放在循环体之后定义
            count++;
        }
        System.out.println("循环结束!");
        // 此处将还可以访问 count 变量
    }
}
```

上面程序的执行过程和前面的 WhileTest.java 程序的执行过程完全相同。因为把 for 循环的循环迭代部分放在循环体之后, 则会出现与 while 循环类似的情形, 如果循环体部分使用 continue 语句来结束本次循环, 将会导致循环迭代语句得不到执行。

把 for 循环的初始化语句放在循环之前定义还有一个作用: 可以扩大初始化语句中所定义变量的作用域。在 for 循环里定义的变量, 其作用域仅在该循环内有效, for 循环终止以后, 这些变量将不可被访问。如果需要在 for 循环以外的地方使用这些变量的值, 就可以采用上面的做法。除此之外, 还有一种做法也可以满足这种要求: 额外定义一个变量来保存这个循环变量的值。例如下面代码片段:

```
int tmp = 0;
// 循环的初始化条件、循环条件、循环迭代语句都在下面一行
for (int i = 0 ; i < 10 ; i++)
{
    System.out.println(count);
    // 使用 tmp 来保存循环变量 i 的值
    tmp = i;
}
System.out.println("循环结束!");
// 此处还可通过 tmp 变量来访问 i 变量的值
```

相比前面的代码, 通常更愿意选择这种解决方案。使用一个变量 tmp 来保存循环变量 i 的值, 使得程序更加清晰, 变量 i 和变量 tmp 的责任更加清晰。反之, 如果采用前一种方法, 则变量 i 的作用域被扩大了, 功能也被扩大了。作用域扩大的后果是: 如果该方法还有另一个循环也需要定义循环变量, 则不能再次使用 i 作为循环变量。



提示:

选择循环变量时, 习惯选择 i、j、k 来作为循环变量。

4.3.4 嵌套循环

如果把一个循环放在另一个循环体内，那么就可以形成嵌套循环，嵌套循环既可以是 for 循环嵌套 while 循环，也可以是 while 循环嵌套 do while 循环……即各种类型的循环都可以作为外层循环，也可以作为内层循环。

当程序遇到嵌套循环时，如果外层循环的循环条件允许，则开始执行外层循环的循环体，而内层循环将被外层循环的循环体来执行——只是内层循环需要反复执行自己的循环体而已。当内层循环执行结束，且外层循环的循环体执行结束时，则再次计算外层循环的循环条件，决定是否再次开始执行外层循环的循环体。

根据上面分析，假设外层循环的循环次数为 n 次，内层循环的循环次数为 m 次，那么内层循环的循环体实际上需要执行 $n \times m$ 次。嵌套循环的执行流程如图 4.1 所示。

从图 4.1 来看，嵌套循环就是把内层循环当成外层循环的循环体。当只有内层循环的循环条件为 false 时，才会完全跳出内层循环，才可以结束外层循环的当次循环，开始下一次循环。下面是一个嵌套循环的示例代码。

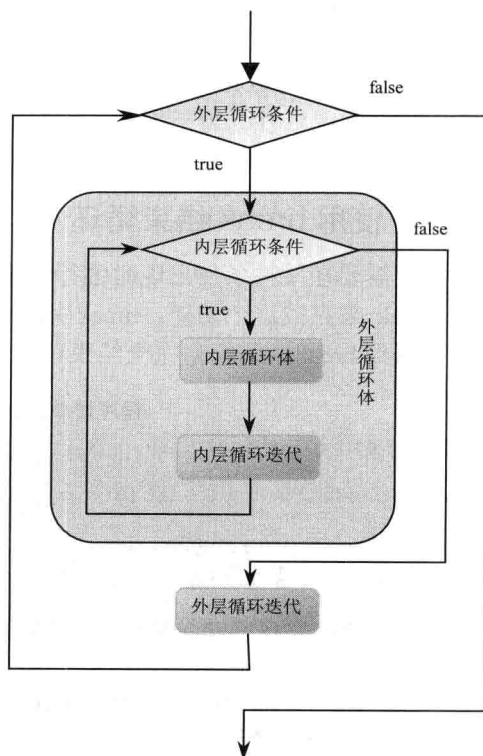


图 4.1 嵌套循环的执行流程

程序清单: codes\04\4.3\NestedLoopTest.java

```
public class NestedLoopTest
{
    public static void main(String[] args)
    {
        // 外层循环
        for (int i = 0 ; i < 5 ; i++ )
        {
            // 内层循环
            for (int j = 0; j < 3 ; j++ )
            {
                System.out.println("i 的值为:" + i + " j 的值为:" + j);
            }
        }
    }
}
```

运行上面程序，看到如下运行结果：

```
i 的值为:0 j 的值为:0
i 的值为:0 j 的值为:1
i 的值为:0 j 的值为:2
.....
```

从上面运行结果可以看出，进入嵌套循环时，循环变量 i 开始为 0，这时即进入了外层循环。进入外层循环后，内层循环把 i 当成一个普通变量，其值为 0。在外层循环的当次循环里，内层循环就是一个普通循环。

实际上，嵌套循环不仅可以是两层嵌套，而且可以是三层嵌套、四层嵌套……不论循环如何嵌套，总可以把内层循环当成外层循环的循环体来对待，区别只是这个循环体里包含了需要反复执行的代码。

4.4 控制循环结构

Java 语言没有提供 `goto` 语句来控制程序的跳转，这种做法提高了程序流程控制的可读性，但降低了程序流程控制的灵活性。为了弥补这种不足，Java 提供了 `continue` 和 `break` 来控制循环结构。除此之外，`return` 可以结束整个方法，当然也就结束了一次循环。

4.4.1 使用 `break` 结束循环

某些时候需要在某种条件出现时强行终止循环，而不是等到循环条件为 `false` 时才退出循环。此时，可以使用 `break` 来完成这个功能。`break` 用于完全结束一个循环，跳出循环体。不管是哪种循环，一旦在循环体中遇到 `break`，系统将完全结束该循环，开始执行循环之后的代码。例如如下程序。

程序清单：codes\04\4.4\BreakTest.java

```
public class BreakTest
{
    public static void main(String[] args)
    {
        // 一个简单的 for 循环
        for (int i = 0; i < 10 ; i++ )
        {
            System.out.println("i 的值是" + i);
            if (i == 2)
            {
                // 执行该语句时将结束循环
                break;
            }
        }
    }
}
```

运行上面程序，将看到 `i` 循环到 2 时即结束，当 `i` 等于 2 时，循环体内遇到 `break` 语句，程序跳出该循环。

`break` 语句不仅可以结束其所在的循环，还可以直接结束其外层循环。此时需要在 `break` 后紧跟一个标签，这个标签用于标识一个外层循环。

Java 中的标签就是一个紧跟着英文冒号 (:) 的标识符。与其他语言不同的是，Java 中的标签只有放在循环语句之前才有作用。例如下面代码。

程序清单：codes\04\4.4\BreakTest2.java

```
public class BreakTest2
{
    public static void main(String[] args)
    {
        // 外层循环，outer 作为标识符
        outer:
        for (int i = 0 ; i < 5 ; i++ )
        {
            // 内层循环
            for (int j = 0; j < 3 ; j++ )
            {
                System.out.println("i 的值为:" + i + " j 的值为:" + j);
                if (j == 1)
                {
                    // 跳出 outer 标签所标识的循环
                    break outer;
                }
            }
        }
    }
}
```

运行上面程序，看到如下运行结果：

```
i 的值为:0 j 的值为:0  
i 的值为:0 j 的值为:1
```

程序从外层循环进入内层循环后，当j等于1时，程序遇到一个 `break outer;` 语句，这行代码将会导致结束 `outer` 标签指定的循环，不是结束 `break` 所在的循环，而是结束 `break` 循环的外层循环。所以看到上面的运行结果。

值得指出的是，`break` 后的标签必须是一个有效的标签，即这个标签必须在 `break` 语句所在的循环之前定义，或者在其所在循环的外层循环之前定义。当然，如果把这个标签放在 `break` 语句所在的循环之前定义，也就失去了标签的意义，因为 `break` 默认就是结束其所在的循环。



通常紧跟 `break` 之后的标签，必须在 `break` 所在循环的外层循环之前定义才有意义。



4.4.2 使用 `continue` 忽略本次循环剩下语句

`continue` 的功能和 `break` 有点类似，区别是 `continue` 只是忽略本次循环剩下语句，接着开始下一次循环，并不会终止循环；而 `break` 则是完全终止循环本身。如下程序示范了 `continue` 的用法。

程序清单：codes\04\4.4\ContinueTest.java

```
public class ContinueTest  
{  
    public static void main(String[] args)  
    {  
        // 一个简单的 for 循环  
        for (int i = 0; i < 3; i++)  
        {  
            System.out.println("i 的值是" + i);  
            if (i == 1)  
            {  
                // 忽略本次循环的剩下语句  
                continue;  
            }  
            System.out.println("continue 后的输出语句");  
        }  
    }  
}
```

运行上面程序，看到如下运行结果：

```
i 的值是 0  
continue 后的输出语句  
i 的值是 1  
i 的值是 2  
continue 后的输出语句
```

从上面运行结果来看，当i等于1时，程序没有输出“`continue` 后的输出语句”字符串，因为程序执行到 `continue` 时，忽略了当次循环中 `continue` 语句后的代码。从这个意义上来看，如果把一个 `continue` 语句放在单次循环的最后一行，这个 `continue` 语句是没有任何意义的——因为它仅仅忽略了一片空白，没有忽略任何程序语句。

与 `break` 类似的是，`continue` 后也可以紧跟一个标签，用于直接跳过标签所标识循环的当次循环的剩下语句，重新开始下一次循环。例如下面代码。

程序清单：codes\04\4.4\ContinueTest2.java

```
public class ContinueTest2  
{  
    public static void main(String[] args)  
    {  
        // 外层循环
```

```
outer:
for (int i = 0 ; i < 5 ; i++ )
{
    // 内层循环
    for (int j = 0; j < 3 ; j++ )
    {
        System.out.println("i 的值为:" + i + " j 的值为:" + j);
        if (j == 1)
        {
            // 忽略 outer 标签所指定的循环中本次循环所剩下语句
            continue outer;
        }
    }
}
```

运行上面程序可以看到, 循环变量 `j` 的值将无法超过 1, 因为每当 `j` 等于 1 时, `continue outer;` 语句就结束了外层循环的当次循环, 直接开始下一次循环, 内层循环没有机会执行完成。

与 `break` 类似的是, `continue` 后的标签也必须是一个有效标签, 即这个标签通常应该放在 `continue` 所在循环的外层循环之前定义。

4.4.3 使用 return 结束方法

`return` 关键字并不是专门用于结束循环的, `return` 的功能是结束一个方法。当一个方法执行到一个 `return` 语句时 (`return` 关键字后还可以跟变量、常量和表达式, 这将在方法介绍中有更详细的解释), 这个方法将被结束。

Java 程序中大部分循环都被放在方法中执行, 例如前面介绍的所有循环示范程序。一旦在循环体内执行到一个 `return` 语句, `return` 语句就会结束该方法, 循环自然也随之结束。例如下面程序。

程序清单: codes\04\4.4\ReturnTest.java

```
public class ReturnTest
{
    public static void main(String[] args)
    {
        // 一个简单的 for 循环
        for (int i = 0; i < 3 ; i++ )
        {
            System.out.println("i 的值是" + i);
            if (i == 1)
            {
                return;
            }
            System.out.println("return 后的输出语句");
        }
    }
}
```

运行上面程序, 循环只能执行到 `i` 等于 1 时, 当 `i` 等于 1 时程序将完全结束 (当 `main` 方法结束时, 也就是 Java 程序结束时)。从这个运行结果来看, 虽然 `return` 并不是专门用于循环结构控制的关键字, 但通过 `return` 语句确实可以结束一个循环。与 `continue` 和 `break` 不同的是, `return` 直接结束整个方法, 不管这个 `return` 处于多少层循环之内。

4.5 数组类型

数组是编程语言中最常见的一种数据结构, 可用于存储多个数据, 每个数组元素存放一个数据, 通常可通过数组元素的索引来访问数组元素, 包括为数组元素赋值和取出数组元素的值。Java 语言的数组则具有其特有的特征, 下面将详细介绍 Java 语言的数组。

4.5.1 理解数组: 数组也是一种类型

Java 的数组要求所有的数组元素具有相同的数据类型。因此, 在一个数组中, 数组元素的类型是唯

一的, 即一个数组里只能存储一种数据类型的数据, 而不能存储多种数据类型的数据。



注意：

因为 Java 语言是面向对象的语言, 而类与类之间可以支持继承关系, 这样可能产生一个数组里可以存放多种数据类型的假象。例如有一个水果数组, 要求每个数组元素都是水果, 实际上数组元素既可以是苹果, 也可以是香蕉(苹果、香蕉都继承了水果, 都是一种特殊的水果), 但这个数组的数组元素的类型还是唯一的, 只能是水果类型。



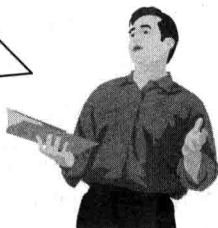
一旦数组的初始化完成, 数组在内存中所占的空间将被固定下来, 因此数组的长度将不可改变。即使把某个数组元素的数据清空, 但它所占的空间依然被保留, 依然属于该数组, 数组的长度依然不变。

Java 的数组既可以存储基本类型的数据, 也可以存储引用类型的数据, 只要所有的数组元素具有相同的类型即可。

值得指出的是, 数组也是一种数据类型, 它本身是一种引用类型。例如 `int` 是一个基本类型, 但 `int[]` (这是定义数组的一种方式) 就是一种引用类型了。

学生提问: `int[]` 是一种类型吗? 怎么使用这种类型呢?

答: 没错, `int[]` 就是一种数据类型, 与 `int` 类型、`String` 类型类似, 一样可以使用该类型来定义变量, 也可以使用该类型进行类型转换等。使用 `int[]` 类型来定义变量、进行类型转换时与使用其他普通类型没有任何区别。`int[]` 类型是一种引用类型, 创建 `int[]` 类型的对象也就是创建数组, 需要使用创建数组的语法。



4.5.2 定义数组

Java 语言支持两种语法格式来定义数组:

```
type[] arrayName;  
type arrayName[];
```

对这两种语法格式而言, 通常推荐使用第一种格式。因为第一种格式不仅具有更好的语意, 而且具有更好的可读性。对于 `type[] arrayName;` 方式, 很容易理解这是定义一个变量, 其中变量名是 `arrayName`, 而变量类型是 `type[]`。前面已经指出: `type[]` 确实是一种新类型, 与 `type` 类型完全不同 (例如 `int` 类型是基本类型, 但 `int[]` 是引用类型)。因此, 这种方式既容易理解, 也符合定义变量的语法。但第二种格式 `type arrayName[]` 的可读性就差了, 看起来好像定义了一个类型为 `type` 的变量, 而变量名是 `arrayName[]`, 这与真实的含义相去甚远。

可能有些读者非常喜欢 `type arrayName[]`; 这种定义数组的方式, 这可能是因为早期某些计算机读物的误导, 从现在开始就不要再使用这种糟糕的方式了。



提示：

Java 的模仿者 C# 就不再支持 `type arrayName[]` 这种语法, 它只支持第一种定义数组的语法。越来越多的语言不再支持 `type arrayName[]` 这种数组定义语法。

数组是一种引用类型的变量, 因此使用它定义一个变量时, 仅仅表示定义了一个引用变量 (也就是定义了一个指针), 这个引用变量还未指向任何有效的内存, 因此定义数组时不能指定数组的长度。而且由于定义数组只是定义了一个引用变量, 并未指向任何有效的内存空间, 所以还没有内存空间来存储数组元素, 因此这个数组也不能使用, 只有对数组进行初始化后才可以使

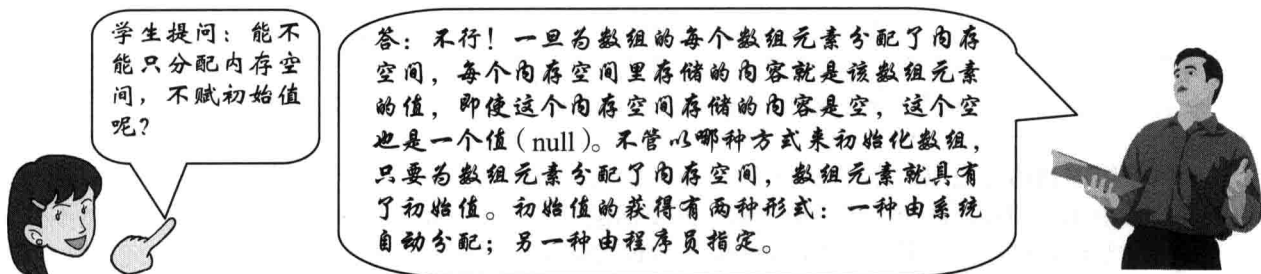


注意： 定义数组时不能指定数组的长度。



4.5.3 数组的初始化

Java 语言中数组必须先初始化,然后才可以使用。所谓初始化,就是为数组的数组元素分配内存空间,并为每个数组元素赋初始值。



数组的初始化有如下两种方式。

- 静态初始化: 初始化时由程序员显式指定每个数组元素的初始值, 由系统决定数组长度。
- 动态初始化: 初始化时程序员只指定数组长度, 由系统为数组元素分配初始值。

1. 静态初始化

静态初始化的语法格式如下:

```
arrayName = new type[]{element1, element2, element3, element4 ...}
```

在上面的语法格式中, 前面的 `type` 就是数组元素的数据类型, 此处的 `type` 必须与定义数组变量时所使用的 `type` 相同, 也可以是定义数组时所指定的 `type` 的子类, 并使用花括号把所有的数组元素括起来, 多个数组元素之间以英文逗号 (,) 隔开, 定义初始化值的花括号紧跟 `[]` 之后。值得指出的是, 执行静态初始化时, 显式指定的数组元素值的类型必须与 `new` 关键字后的 `type` 类型相同, 或者是其子类的实例。下面代码定义了使用这三种形式来进行静态初始化。

程序清单: codes\04\4.5\ArrayTest.java

```
// 定义一个 int 数组类型的变量, 变量名为 intArr
int[] intArr;
// 使用静态初始化, 初始化数组时只指定数组元素的初始值, 不指定数组长度
intArr = new int[]{5, 6, 8, 20};
// 定义一个 Object 数组类型的变量, 变量名为 objArr
Object[] objArr;
// 使用静态初始化, 初始化数组时数组元素的类型是
// 定义数组时所指定的数组元素类型的子类
objArr = new String[] {"Java", "李刚"};
Object[] objArr2;
// 使用静态初始化
objArr2 = new Object[] {"Java", "李刚"};
```

因为 Java 语言是面向对象的编程语言, 能很好地支持子类和父类的继承关系: 子类实例是一种特殊的父类实例。在上面程序中, `String` 类型是 `Object` 类型的子类, 即字符串是一种特殊的 `Object` 实例。关于继承更详细的介绍, 请参考本书第 5 章。

除此之外, 静态初始化还有如下简化的语法格式:

```
type[] arrayName = {element1, element2, element3, element4 ...}
```

在这种语法格式中, 直接使用花括号来定义一个数组, 花括号把所有的数组元素括起来形成一个数组。只有在定义数组的同时执行数组初始化才支持使用简化的静态初始化。

在实际开发过程中, 可能更习惯将数组定义和数组初始化同时完成, 代码如下 (程序清单同上):

```
// 数组的定义和初始化同时完成, 使用简化的静态初始化写法
int[] a = {5, 6, 7, 9};
```

2. 动态初始化

动态初始化只指定数组的长度，由系统为每个数组元素指定初始值。动态初始化的语法格式如下：

```
arrayName = new type[length];
```

在上面语法中，需要指定一个 `int` 类型的 `length` 参数，这个参数指定了数组的长度，也就是可以容纳数组元素的个数。与静态初始化相似的是，此处的 `type` 必须与定义数组时使用的 `type` 类型相同，或者是定义数组时使用的 `type` 类型的子类。下面代码示范了如何进行动态初始化（程序清单同上）。

```
// 数组的定义和初始化同时完成，使用动态初始化语法
int[] prices = new int[5];
// 数组的定义和初始化同时完成，初始化数组时元素的类型是定义数组时元素类型的子类
Object[] books = new String[4];
```

执行动态初始化时，程序员只需指定数组的长度，即为每个数组元素指定所需的内存空间，系统将负责为这些数组元素分配初始值。指定初始值时，系统按如下规则分配初始值。

- 数组元素的类型是基本类型中的整数类型（`byte`、`short`、`int` 和 `long`），则数组元素的值是 0。
- 数组元素的类型是基本类型中的浮点类型（`float`、`double`），则数组元素的值是 0.0。
- 数组元素的类型是基本类型中的字符类型（`char`），则数组元素的值是 `'\u0000'`。
- 数组元素的类型是基本类型中的布尔类型（`boolean`），则数组元素的值是 `false`。
- 数组元素的类型是引用类型（类、接口和数组），则数组元素的值是 `null`。



注意：不要同时使用静态初始化和动态初始化，也就是说，不要在进行数组初始化时，既指定数组的长度，也为每个数组元素分配初始值。



数组初始化完成后，就可以使用数组了，包括为数组元素赋值、访问数组元素值和获得数组长度等。

4.5.4 使用数组

数组最常用的用法就是访问数组元素，包括对数组元素进行赋值和取出数组元素的值。访问数组元素都是通过在数组引用变量后紧跟一个方括号（`[]`），方括号里是数组元素的索引值，这样就可以访问数组元素了。访问到数组元素后，就可以把一个数组元素当成一个普通变量使用了，包括为该变量赋值和取出该变量的值，这个变量的类型就是定义数组时使用的类型。

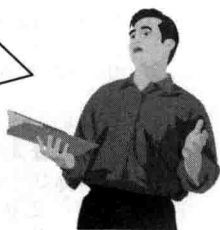
Java 语言的数组索引是从 0 开始的，也就是说，第一个数组元素的索引值为 0，最后一个数组元素的索引值为数组长度减 1。下面代码示范了输出数组元素的值，以及为指定数组元素赋值（程序清单同上）。

```
// 输出 objArr 数组的第二个元素，将输出字符串"李刚"
System.out.println(objArr[1]);
// 为 objArr2 的第一个数组元素赋值
objArr2[0] = "Spring";
```

如果访问数组元素时指定的索引值小于 0，或者大于等于数组的长度，编译程序不会出现任何错误，但运行时出现异常：`java.lang.ArrayIndexOutOfBoundsException: N`（数组索引越界异常），异常信息后的 `N` 就是程序员试图访问的数组索引。

学生提问：为什么要我记住这些异常信息？

答：编写程序，并不是单单指在电脑里敲出这些代码，还包括调试这个程序，使之可以正常运行。没有任何人可以保证自己写的程序总是正确的，因此调试程序是写程序的重要组成部分，调试程序的工作量往往超过编写代码的工作量。如何根据错误提示信息，准确定位错误位置，以及排除错误是程序员的基本功。培养这些基本功需要记住常见的异常信息，以及对应的出错原因。



下面代码试图访问的数组元素索引值等于数组长度，将引发数组索引越界异常（程序清单同上）。

```
// 访问数组元素指定的索引值等于数组长度，所以下面代码将在运行时出现异常
System.out.println(objArr2[2]) ;
```

所有的数组都提供了一个 `length` 属性，通过这个属性可以访问到数组的长度，一旦获得了数组的长度，就可以通过循环来遍历该数组的每个数组元素。下面代码示范了输出 `prices` 数组（动态初始化的 `int[]` 数组）的每个数组元素的值（程序清单同上）。

```
// 使用循环输出 prices 数组的每个数组元素的值
for (int i = 0; i < prices.length ; i ++ )
{
    System.out.println(prices[i]);
}
```

执行上面代码将输出 5 个 0，因为 `prices` 数组执行的是默认初始化，数组元素是 `int` 类型，系统为 `int` 类型的数组元素赋值为 0。

下面代码示范了为动态初始化的数组元素进行赋值，并通过循环方式输出每个数组元素（程序清单同上）。

```
// 对动态初始化后的数组元素进行赋值
books[0] = "疯狂 Java 讲义";
books[1] = "轻量级 Java EE 企业应用实战";
// 使用循环输出 books 数组的每个数组元素的值
for (int i = 0 ; i < books.length ; i++ )
{
    System.out.println(books[i]);
}
```

上面代码将先输出字符串“疯狂 Java 讲义”和“轻量级 Java EE 企业应用实战”，然后输出两个 `null`，因为 `books` 使用了动态初始化，系统为所有数组元素都分配一个 `null` 作为初始值，后来程序又为前两个元素赋值，所以看到了这样的程序输出结果。

从上面代码中不难看出，初始化一个数组后，相当于同时初始化了多个相同类型的变量，通过数组元素的索引就可以自由访问这些变量（实际上都是数组元素）。使用数组元素与使用普通变量并没有什么不同，一样可以对数组元素进行赋值，或者取出数组元素的值。

4.5.5 foreach 循环

从 Java 5 之后，Java 提供了一种更简单的循环：`foreach` 循环，这种循环遍历数组和集合（关于集合的介绍请参考本书第 8 章）更加简洁。使用 `foreach` 循环遍历数组和集合元素时，无须获得数组和集合长度，无须根据索引来访问数组元素和集合元素，`foreach` 循环自动遍历数组和集合的每个元素。

`foreach` 循环的语法格式如下：

```
for(type variableName : array | collection)
{
    // variableName 自动迭代访问每个元素...
}
```

在上面语法格式中，`type` 是数组元素或集合元素的类型，`variableName` 是一个形参名，`foreach` 循环将自动将数组元素、集合元素依次赋给该变量。下面程序示范了如何使用 `foreach` 循环来遍历数组元素。

程序清单：codes\04\4.5\ForEachTest.java

```
public class ForEachTest
{
    public static void main(String[] args)
    {
        String[] books = {"轻量级 Java EE 企业应用实战" ,
            "疯狂 Java 讲义",
            "疯狂 Android 讲义"};
        // 使用 foreach 循环来遍历数组元素
        // 其中 book 将会自动迭代每个数组元素
```

```
        for (String book : books)
        {
            System.out.println(book);
        }
    }
}
```

从上面程序可以看出，使用 `foreach` 循环遍历数组元素时无须获得数组长度，也无须根据索引来访问数组元素。`foreach` 循环和普通循环不同的是，它无须循环条件，无须循环迭代语句，这些部分都由系统来完成，`foreach` 循环自动迭代数组的每个元素，当每个元素都被迭代一次后，`foreach` 循环自动结束。

当使用 `foreach` 循环来迭代输出数组元素或集合元素时，通常不要对循环变量进行赋值，虽然这种赋值在语法上是允许的，但没有太大的实际意义，而且极容易引起错误。例如下面程序。

程序清单：codes\04\4.5\ForeachErrorTest.java

```
public class ForEachErrorTest
{
    public static void main(String[] args)
    {
        String[] books = {"轻量级 Java EE 企业应用实战",
                           "疯狂 Java 讲义",
                           "疯狂 Android 讲义"};
        // 使用 foreach 循环来遍历数组元素，其中 book 将会自动迭代每个数组元素
        for (String book : books)
        {
            book = "疯狂 Ajax 讲义";
            System.out.println(book);
        }
        System.out.println(books[0]);
    }
}
```

运行上面程序，将看到如下运行结果：

```
疯狂 Ajax 讲义
疯狂 Ajax 讲义
疯狂 Ajax 讲义
轻量级 Java EE 企业应用实战
```

从上面运行结果来看，由于在 `foreach` 循环中对数组元素进行赋值，结果导致不能正确遍历数组元素，不能正确地取出每个数组元素的值。而且当再次访问第一个数组元素时，发现数组元素的值依然没有改变。不难看出，当使用 `foreach` 来迭代访问数组元素时，`foreach` 中的循环变量相当于一个临时变量，系统会把数组元素依次赋给这个临时变量，而这个临时变量并不是数组元素，它只是保存了数组元素的值。因此，如果希望改变数组元素的值，则不能使用这种 `foreach` 循环。



注意：

使用 `foreach` 循环迭代数组元素时，并不能改变数组元素的值，因此不要对 `foreach` 的循环变量进行赋值。



4.6 深入数组

数组是一种引用数据类型，数组引用变量只是一个引用，数组元素和数组变量在内存里是分开存放的。下面将深入介绍数组在内存中的运行机制。

4.6.1 内存中的数组

数组引用变量只是一个引用，这个引用变量可以指向任何有效的内存，只有当该引用指向有效内存后，才可通过该数组变量来访问数组元素。

与所有引用变量相同的是，引用变量是访问真实对象的根本方式。也就是说，如果希望在程序中访

问数组对象本身, 则只能通过这个数组的引用变量来访问它。

实际的数组对象被存储在堆(heap)内存中; 如果引用该数组对象的数组引用变量是一个局部变量, 那么它被存储在栈(stack)内存中。数组在内存中的存储示意图如图 4.2 所示。

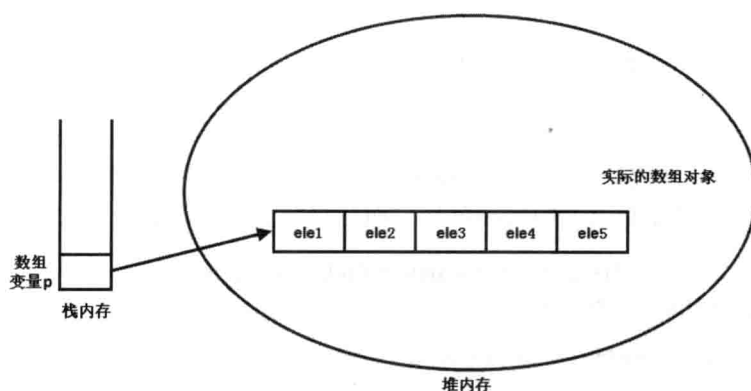
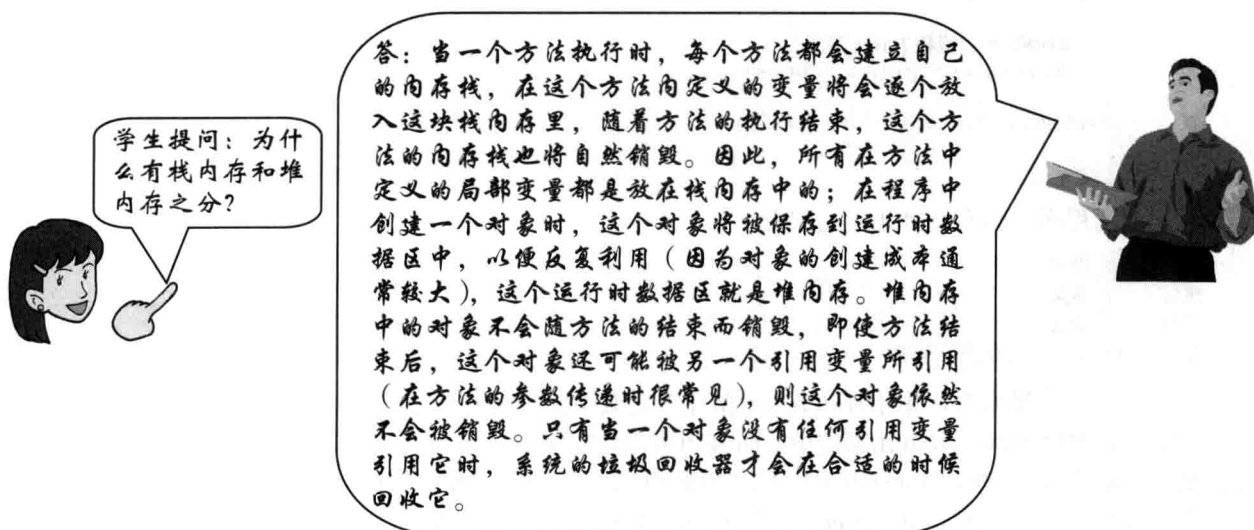


图 4.2 数组在内存中的存储示意图

如果需要访问如图 4.2 所示堆内存中的数组元素, 则程序中只能通过 `p[index]` 的形式实现。也就是说, 数组引用变量是访问堆内存中数组元素的根本方式。



如果堆内存中数组不再有任何引用变量指向自己, 则这个数组将成为垃圾, 该数组所占的内存将会被系统的垃圾回收机制回收。因此, 为了让垃圾回收机制回收一个数组所占的内存空间, 可以将该数组变量赋为 `null`, 也就切断了数组引用变量和实际数组之间的引用关系, 实际的数组也就成了垃圾。

只要类型相互兼容, 就可以让一个数组变量指向另一个实际的数组, 这种操作会让人产生数组的长度可变的错觉。如下代码所示。

程序清单: codes\04\4.6\ArrayInRam.java

```
public class ArrayInRam
{
    public static void main(String[] args)
    {
        // 定义并初始化数组, 使用静态初始化
        int[] a = {5, 7, 20};
        // 定义并初始化数组, 使用动态初始化
        int[] b = new int[4];
        // 输出 b 数组的长度
        System.out.println("b 数组的长度为: " + b.length);
        // 循环输出 a 数组的元素
        for (int i = 0, len = a.length; i < len; i++)
```

```

    {
        System.out.println(a[i]);
    }
    // 循环输出 b 数组的元素
    for (int i = 0, len = b.length; i < len; i++)
    {
        System.out.println(b[i]);
    }
    // 因为 a 是 int[] 类型, b 也是 int[] 类型, 所以可以将 a 的值赋给 b
    // 也就是让 b 引用指向 a 引用指向的数组
    b = a;
    // 再次输出 b 数组的长度
    System.out.println("b 数组的长度为: " + b.length);
}

```

运行上面代码后, 将可以看到先输出 b 数组的长度为 4, 然后依次输出 a 数组和 b 数组的每个数组元素, 接着会输出 b 数组的长度为 3。看起来似乎数组的长度是可变的, 但这只是一个假象。必须牢记: 定义并初始化一个数组后, 在内存中分配了两个空间, 一个用于存放数组的引用变量, 另一个用于存放数组本身。下面将结合示意图来说明上面程序的运行过程。

当程序定义并初始化了 a、b 两个数组后, 系统内存中实际上产生了 4 块内存区, 其中栈内存中有两个引用变量: a 和 b; 堆内存中也有两块内存区, 分别用于存储 a 和 b 引用所指向的数组本身。此时计算机内存的存储示意图如图 4.3 所示。

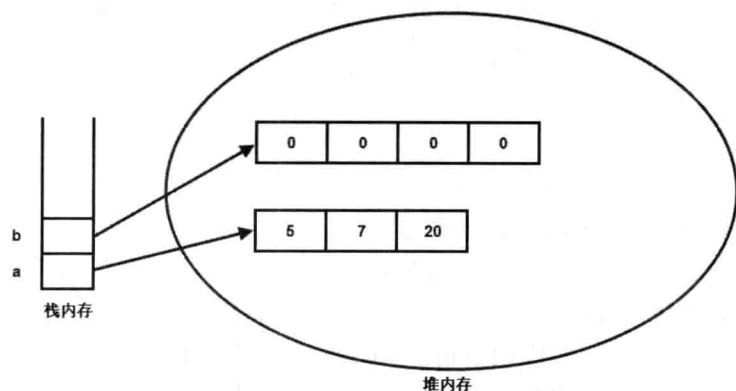


图 4.3 定义并初始化 a、b 两个数组后的存储示意图

从图 4.3 中可以非常清楚地看出 a 引用和 b 引用各自所引用的数组对象, 并可以很清楚地看出 a 变量所引用的数组长度是 3, b 变量所引用的数组长度是 4。

当执行上面的粗体字标识代码 **b = a;** 时, 系统将会把 a 的值赋给 b, a 和 b 都是引用类型变量, 存储的是地址。因此把 a 的值赋给 b 后, 就是让 b 指向 a 所指向的地址。此时计算机内存的存储示意图如图 4.4 所示。

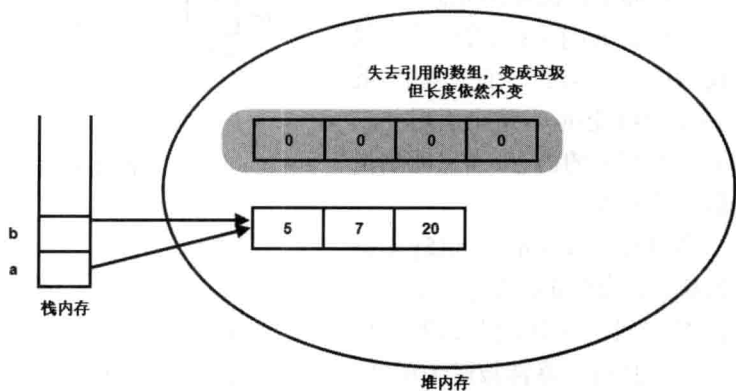


图 4.4 让 b 引用指向 a 引用所指向数组后的存储示意图

从图 4.4 中可以看出, 当执行了 **b = a;** 之后, 堆内存中的第一个数组具有了两个引用: a 变量和 b 变量都引用了第一个数组。此时第二个数组失去了引用, 变成垃圾, 只有等待垃圾回收机制来回收它——但它的长度依然不会改变, 直到它彻底消失。



提示:

程序员进行程序开发时, 不要仅仅停留在代码表面, 而要深入底层的运行机制, 才可以对程序的运行机制有更准确的把握。看待一个数组时, 一定要把数组看成两个部分: 一部分是数组引用, 也就是在代码中定义的数组引用变量; 还有一部分是实际的数组对象, 这部分是在堆内存里运行的, 通常无法直接访问它, 只能通过数组引用变量来访问。

4.6.2 基本类型数组的初始化

对于基本类型数组而言,数组元素的值直接存储在对应的数组元素中,因此,初始化数组时,先为该数组分配内存空间,然后将数组元素的值存入对应数组元素中。

下面程序定义了一个 `int[]` 类型的数组变量,采用动态初始化的方式初始化了该数组,并显式为每个数组元素赋值。

程序清单: `codes\04\4.6\PrimitiveArrayTest.java`

```
public class PrimitiveArrayTest
{
    public static void main(String[] args)
    {
        // 定义一个 int[] 类型的数组变量
        int[] iArr;
        // 动态初始化数组, 数组长度为 5
        iArr = new int[5];
        // 采用循环方式为每个数组元素赋值
        for (int i = 0; i < iArr.length; i++)
        {
            iArr[i] = i + 10;
        }
    }
}
```

上面代码的执行过程代表了基本类型数组初始化的典型过程。下面将结合示意图详细介绍这段代码的执行过程。

执行第一行代码 `int[] iArr;` 时,仅定义一个数组变量,此时内存中的存储示意图如图 4.5 所示。

执行了 `int[] iArr;` 代码后,仅在栈内存中定义了一个空引用(就是 `iArr` 数组变量),这个引用并未指向任何有效的内存,当然无法指定数组的长度。

当执行 `iArr = new int[5];` 动态初始化后,系统将负责为该数组分配内存空间,并分配默认的初始值:所有数组元素都被赋值为 0,此时内存中的存储示意图如图 4.6 所示。

此时 `iArr` 数组的每个数组元素的值都是 0,当循环为该数组的每个数组元素依次赋值后,此时每个数组元素的值都变成程序显式指定的值。显式指定每个数组元素值后的存储示意图如图 4.7 所示。

从图 4.7 中可以看到基本类型数组的存储示意图,每个数组元素的值直接存储在对应的内存中。操作基本类型数组的数组元素时,实际上相当于操作基本类型的变量。

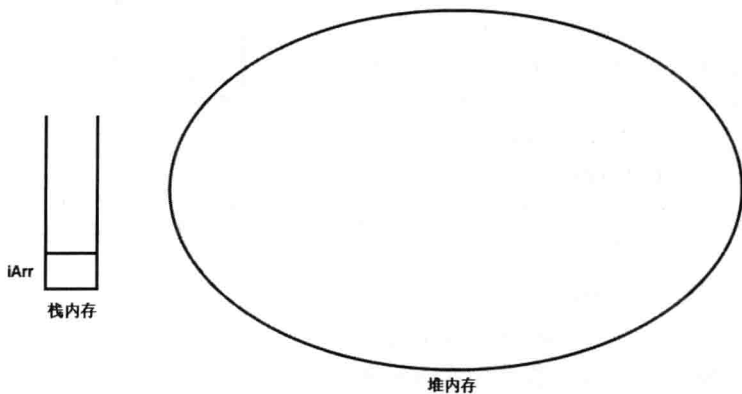


图 4.5 定义 `iArr` 数组变量后的存储示意图

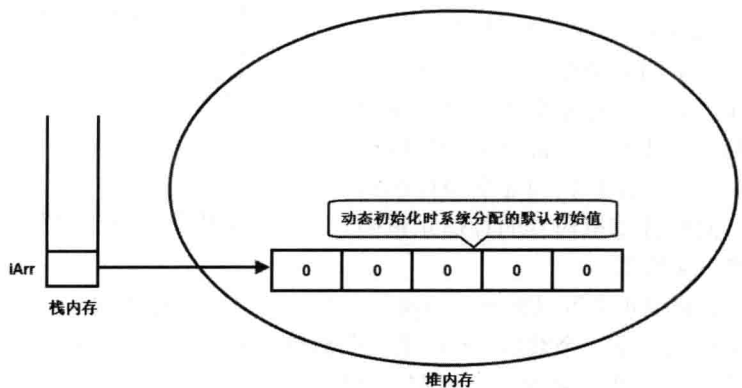


图 4.6 动态初始化 `iArr` 数组后的存储示意图

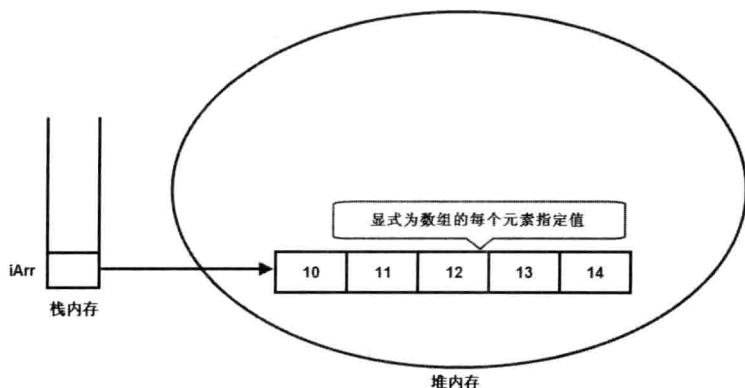


图 4.7 显式指定每个数组元素值后的存储示意图

4.6.3 引用类型数组的初始化

引用类型数组的数组元素是引用，因此情况变得更加复杂。每个数组元素里存储的还是引用，它指向另一块内存，这块内存里存储了有效数据。

为了更好地说明引用类型数组的运行过程，下面先定义一个 `Person` 类（所有类都是引用类型）。关于定义类、对象和引用的详细介绍请参考第 5 章。`Person` 类的代码如下：

程序清单：codes\04\4.6\ReferenceArrayTest.java

```
class Person
{
    public int age; // 年龄
    public double height; // 身高
    // 定义一个 info 方法
    public void info()
    {
        System.out.println("我的年龄是: " + age
            + ", 我的身高是: " + height);
    }
}
```

下面程序将定义一个 `Person[]` 数组，接着动态初始化这个 `Person[]` 数组，并为这个数组的每个数组元素指定值。程序代码如下（程序清单同上）：

```
public class ReferenceArrayTest
{
    public static void main(String[] args)
    {
        // 定义一个 students 数组变量，其类型是 Person[]
        Person[] students;
        // 执行动态初始化
        students = new Person[2];
        // 创建一个 Person 实例，并将这个 Person 实例赋给 zhang 变量
        Person zhang = new Person();
        // 为 zhang 所引用的 Person 对象的 age、height 赋值
        zhang.age = 15;
        zhang.height = 158;
        // 创建一个 Person 实例，并将这个 Person 实例赋给 lee 变量
        Person lee = new Person();
        // 为 lee 所引用的 Person 对象的 age、height 赋值
        lee.age = 16;
        lee.height = 161;
        // 将 zhang 变量的值赋给第一个数组元素
        students[0] = zhang;
        // 将 lee 变量的值赋给第二个数组元素
        students[1] = lee;
        // 下面两行代码的结果完全一样，因为 lee
        // 和 students[1] 指向的是同一个 Person 实例
        lee.info();
    }
}
```



```

        students[1].info();
    }
}
    
```

上面代码的执行过程代表了引用类型数组初始化的典型过程。下面将结合示意图详细介绍这段代码的执行过程。

执行 `Person[] students;` 代码时, 这行代码仅仅在栈内存中定义了一个引用变量, 也就是一个指针, 这个指针并未指向任何有效的内存区。此时内存中存储示意图如图 4.8 所示。

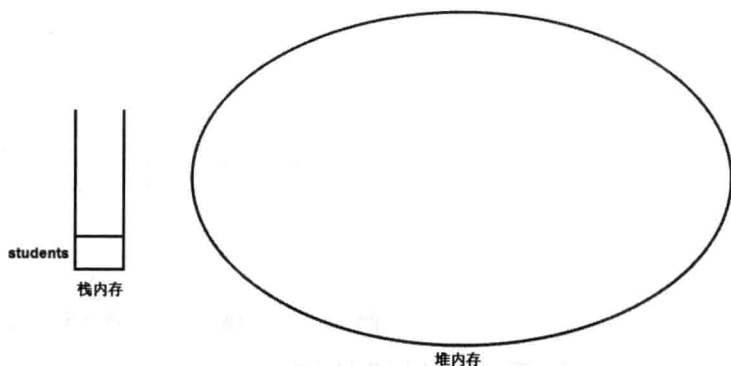


图 4.8 定义一个 students 数组变量后的存储示意图

在如图 4.8 所示的栈内存中定义了一个 `students` 变量, 它仅仅是一个引用, 并未指向任何有效的内存。直到执行初始化, 本程序对 `students` 数组执行动态初始化, 动态初始化由系统为数组元素分配默认的初始值: `null`, 即每个数组元素的值都是 `null`。执行动态初始化后的存储示意图如图 4.9 所示。

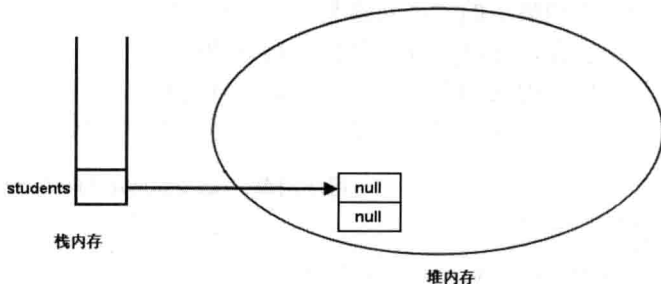


图 4.9 动态初始化 students 数组后的存储示意图

从图 4.9 中可以看出, `students` 数组的两个数组元素都是引用, 而且这个引用并未指向任何有效的内存, 因此每个数组元素的值都是 `null`。这意味着依然不能直接使用 `students` 数组元素, 因为每个数组元素都是 `null`, 这相当于定义了两个连续的 `Person` 变量, 但这个变量还未指向任何有效的内存区, 所以这两个连续的 `Person` 变量 (`students` 数组的数组元素) 还不能使用。

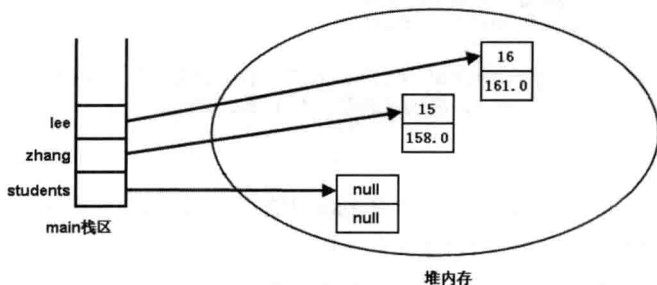


图 4.10 创建两个 Person 实例后的存储示意图

接着的代码定义了 `zhang` 和 `lee` 两个 `Person` 实例, 定义这两个实例实际上分配了 4 块内存, 在栈内存中存储了 `zhang` 和 `lee` 两个引用变量, 还在堆内存中存储了两个 `Person` 实例。此时的内存存储示意图如图 4.10 所示。

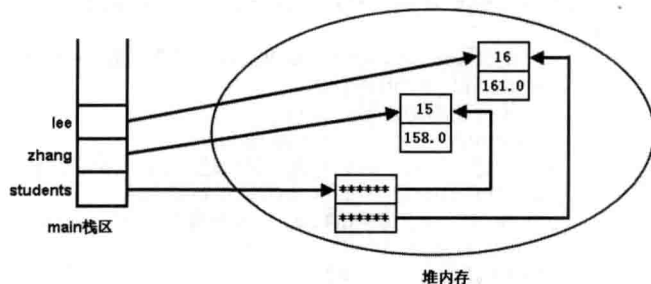


图 4.11 为数组元素赋值后的存储示意图

此时 `students` 数组的两个数组元素依然是 `null`, 直到程序依次将 `zhang`

赋给 `students` 数组的第一个元素, 把 `lee` 赋给 `students` 数组的第二个元素, `students` 数组的两个数组元素将会指向有效的内存区。此时的内存存储示意图如图 4.11 所示。

从图 4.11 中可以看出, 此时 `zhang` 和 `students[0]` 指向同一个内存区, 而且它们都是引用类型变量, 因此通过 `zhang` 和 `students[0]` 来访问 `Person` 实例的实例变量和方法的效果完全一样, 不论修改 `students[0]` 所指向的 `Person` 实例的实例变量, 还是修改 `zhang` 变量所指向的 `Person` 实例的实例变量, 所修改的其实是同一个内存区, 所以必然互相影响。同理, `lee` 和 `students[1]` 也是引用同一个 `Person` 对象, 也具有

相同的效果。

4.6.4 没有多维数组

Java 语言里提供了支持多维数组的语法。但本书还是想说，没有多维数组——如果从数组底层的运行机制上来看。

Java 语言里的数组类型是引用类型，因此数组变量其实是一个引用，这个引用指向真实的数组内存。数组元素的类型也可以是引用，如果数组元素的引用再次指向真实的数组内存，这种情形看上去很像多维数组。

回到前面定义数组类型的语法：`type[] arrName;`，这是典型的一维数组的定义语法，其中 `type` 是数组元素的类型。如果希望数组元素也是一个引用，而且是指向 `int` 数组的引用，则可以把 `type` 具体成 `int[]`（前面已经指出，`int[]` 就是一种类型，`int[]` 类型的用法与普通类型并无任何区别），那么上面定义数组的语法就是 `int[][] arrName`。

如果把 `int` 这个类型扩大到 Java 的所有类型（不包括数组类型），则出现了定义二维数组的语法：

```
type[][] arrName;
```

Java 语言采用上面的语法格式来定义二维数组，但它的实质还是一维数组，只是其数组元素也是引用，数组元素里保存的引用指向一维数组。

接着对这个“二维数组”执行初始化，同样可以把这个数组当成一维数组来初始化，把这个“二维数组”当成一个一维数组，其元素的类型是 `type[]` 类型，则可以采用如下语法进行初始化：

```
arrName = new type[length][]
```

上面的初始化语法相当于初始化了一个一维数组，这个一维数组的长度是 `length`。同样，因为这个一维数组的数组元素是引用类型（数组类型）的，所以系统为每个数组元素都分配初始值：`null`。

这个二维数组实际上完全可以当成一维数组使用：使用 `new type[length]` 初始化一维数组后，相当于定义了 `length` 个 `type` 类型的变量；类似的，使用 `new type[length][]` 初始化这个数组后，相当于定义了 `length` 个 `type[]` 类型的变量，当然，这些 `type[]` 类型的变量都是数组类型，因此必须再次初始化这些数组。

下面程序示范了如何把二维数组当成一维数组处理。

程序清单：codes\04\4.6\TwoDimensionTest.java

```
public class TwoDimensionTest
{
    public static void main(String[] args)
    {
        // 定义一个二维数组
        int[][] a;
        // 把 a 当成一维数组进行初始化，初始化 a 是一个长度为 4 的数组
        // a 数组的数组元素又是引用类型
        a = new int[4][];
        // 把 a 数组当成一维数组，遍历 a 数组的每个数组元素
        for (int i = 0, len = a.length; i < len; i++)
        {
            System.out.println(a[i]);
        }
        // 初始化 a 数组的第一个元素
        a[0] = new int[2];
        // 访问 a 数组的第一个元素所指数组的第二个元素
        a[0][1] = 6;
        // a 数组的第一个元素是一个一维数组，遍历这个一维数组
        for (int i = 0, len = a[0].length; i < len; i++)
        {
            System.out.println(a[0][i]);
        }
    }
}
```

上面程序中粗体字代码部分把 `a` 这个二维数组当成一维数组处理，只是每个数组元素都是 `null`，所

以看到输出结果都是 null。下面结合示意图来说明这个程序的执行过程。

程序的第一行 `int[][] a;`，将在栈内存中定义一个引用变量，这个变量并未指向任何有效的内存空间，此时的堆内存中还未为这行代码分配任何存储区。

程序对 `a` 数组执行初始化：`a = new int[4][];`，这行代码让 `a` 变量指向一块长度为 4 的数组内存，这个长度为 4 的数组里每个数组元素都是引用类型（数组类型），系统为这些数组元素分配默认的初始值：`null`。此时 `a` 数组在内存中的存储示意图如图 4.12 所示。

从图 4.12 来看，虽然声明 `a` 是一个二维数组，但这里丝毫看不出它是一个二维数组的样子，完全是一维数组的样子。这个一维数组的长度是 4，只是这 4 个数组元素都是引用类型，它们的默认值是 `null`。所以程序中可以把 `a` 数组当成一维数组处理，依次遍历 `a` 数组的每个元素，将看到每个数组元素的值都是 `null`。

由于 `a` 数组的元素必须是 `int[]` 数组，所以接下来的程序对 `a[0]` 元素执行初始化，也就是让图 4.12 右边堆内存中的第一个数组元素指向一个有效的数组内存，指向一个长度为 2 的 `int` 数组。因为程序采用动态初始化 `a[0]` 数组，因此系统将为 `a[0]` 所引用数组的每个元素分配默认的初始值：`0`，然后程序显式为 `a[0]` 数组的第二个元素赋值为 6。此时在内存中的存储示意图如图 4.13 所示。

图 4.13 中灰色覆盖的数组元素就是程序显式指定的数组元素值。`TwoDimensionTest.java` 接着迭代输出 `a[0]` 数组的每个数组元素，将看到输出 0 和 6。

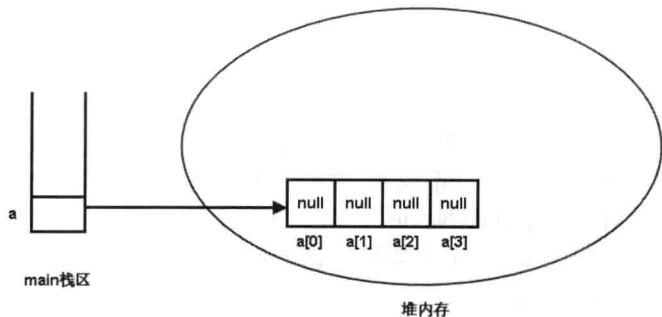


图 4.12 将二维数组当成一维数组初始化的存储示意图

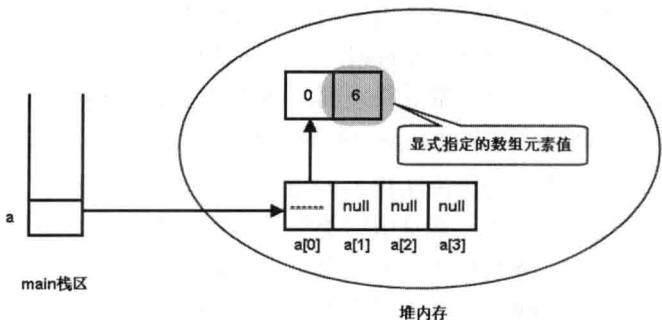
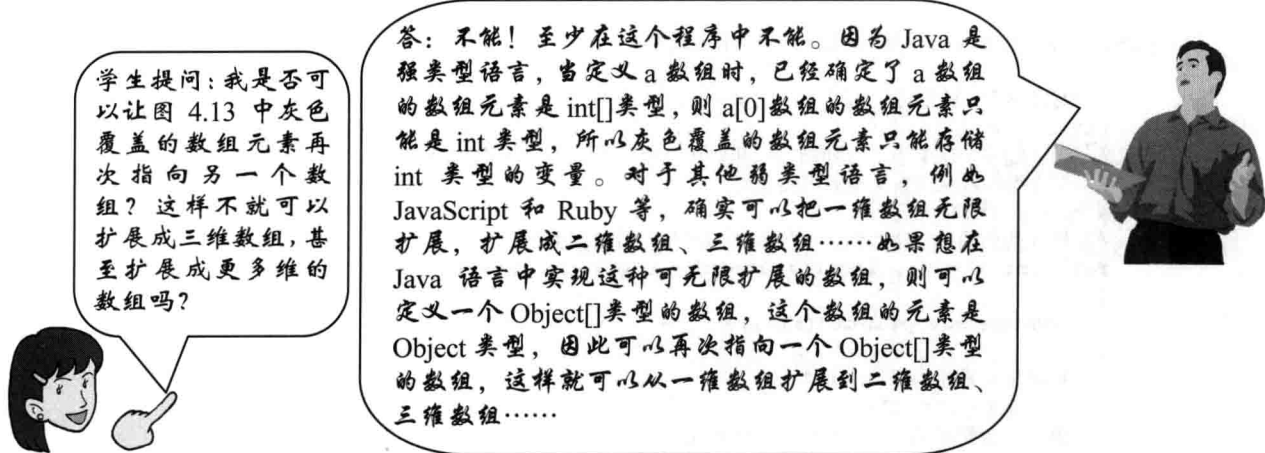


图 4.13 初始化 `a[0]` 后的存储示意图



从上面程序中可以看出，初始化多维数组时，可以只指定最左边维的大小；当然，也可以一次指定每一维的大小。例如下面代码（程序清单同上）：

```
// 同时初始化二维数组的两个维数
int[][] b = new int[3][4];
```

上面代码将定义一个 `b` 数组变量，这个数组变量指向一个长度为 3 的数组，这个数组的每个数组元素又是一个数组类型，它们各指向对应的长度为 4 的 `int[]` 数组，每个数组元素的值为 0。这行代码执行

后在内存中的存储示意图如图 4.14 所示。

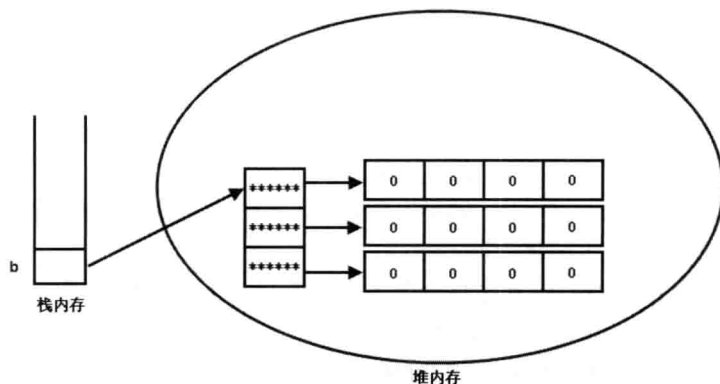


图 4.14 同时初始化二维数组的两个维数后的存储示意图

还可以使用静态初始化方式来初始化二维数组。使用静态初始化方式来初始化二维数组时，二维数组的每个数组元素都是一维数组，因此必须指定多个一维数组作为二维数组的初始化值。如下代码所示（程序清单同上）：

```
// 使用静态初始化语法来初始化一个二维数组
String[][] str1 = new String[][]{new String[3]
    , new String[]{"hello"}};
// 使用简化的静态初始化语法来初始化二维数组
String[][] str2 = {new String[3]
    , new String[]{"hello"}};
```

上面代码执行后内存中的存储示意图如图 4.15 所示。

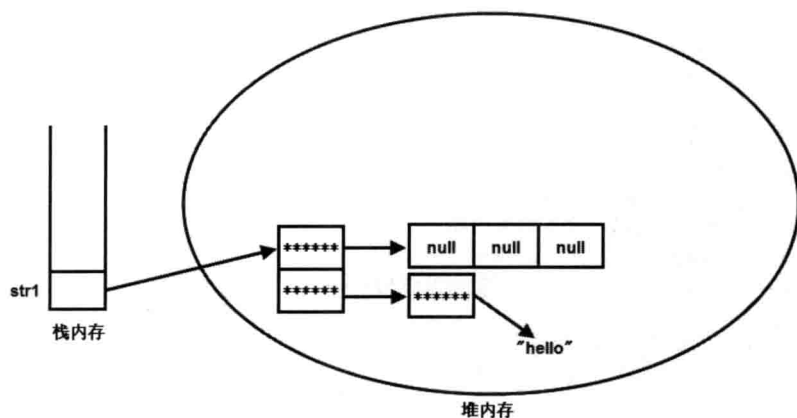


图 4.15 采用静态初始化语法初始化二维数组的存储示意图

通过上面讲解可以得到一个结论：二维数组是一维数组，其数组元素是一维数组；三维数组也是一维数组，其数组元素是二维数组……从这个角度来看，Java 语言里没有多维数组。

4.6.5 Java 8 增强的工具类：Arrays

Java 提供的 Arrays 类里包含的一些 static 修饰的方法可以直接操作数组，这个 Arrays 类里包含了如下几个 static 修饰的方法（static 修饰的方法可以直接通过类名调用）。

- `int binarySearch(type[] a, type key)`: 使用二分法查询 key 元素值在 a 数组中出现的索引；如果 a 数组不包含 key 元素值，则返回负数。调用该方法时要求数组中元素已经按升序排列，这样才能得到正确结果。
- `int binarySearch(type[] a, int fromIndex, int toIndex, type key)`: 这个方法与前一个方法类似，但它只搜索 a 数组中 fromIndex 到 toIndex 索引的元素。调用该方法时要求数组中元素已经按升序排列，这样才能得到正确结果。

- `type[] copyOf(type[] original, int length)`: 这个方法将会把 `original` 数组复制成一个新的数组, 其中 `length` 是新数组的长度。如果 `length` 小于 `original` 数组的长度, 则新数组就是原数组的前面 `length` 个元素; 如果 `length` 大于 `original` 数组的长度, 则新数组的前面元素就是原数组的所有元素, 后面补充 0 (数值类型)、`false` (布尔类型) 或者 `null` (引用类型)。
- `type[] copyOfRange(type[] original, int from, int to)`: 这个方法与前面方法相似, 但这个方法只复制 `original` 数组的 `from` 索引到 `to` 索引的元素。
- `boolean equals(type[] a, type[] a2)`: 如果 `a` 数组和 `a2` 数组的长度相等, 而且 `a` 数组和 `a2` 数组的数组元素也一一相同, 该方法将返回 `true`。
- `void fill(type[] a, type val)`: 该方法将会把 `a` 数组的所有元素都赋值为 `val`。
- `void fill(type[] a, int fromIndex, int toIndex, type val)`: 该方法与前一个方法的作用相同, 区别只是该方法仅仅将 `a` 数组的 `fromIndex` 到 `toIndex` 索引的数组元素赋值为 `val`。
- `void sort(type[] a)`: 该方法对 `a` 数组的数组元素进行排序。
- `void sort(type[] a, int fromIndex, int toIndex)`: 该方法与前一个方法相似, 区别是该方法仅仅对 `fromIndex` 到 `toIndex` 索引的元素进行排序。
- `String toString(type[] a)`: 该方法将一个数组转换成一个字符串。该方法按顺序把多个数组元素连缀在一起, 多个数组元素使用英文逗号 (,) 和空格隔开。

下面程序示范了 `Arrays` 类的用法。

程序清单: `codes\04\4.6\ArraysTest.java`

```
public class ArraysTest
{
    public static void main(String[] args)
    {
        // 定义一个 a 数组
        int[] a = new int[]{3, 4, 5, 6};
        // 定义一个 a2 数组
        int[] a2 = new int[]{3, 4, 5, 6};
        // a 数组和 a2 数组的长度相等, 每个元素依次相等, 将输出 true
        System.out.println("a 数组和 a2 数组是否相等: "
            + Arrays.equals(a, a2));
        // 通过复制 a 数组, 生成一个新的 b 数组
        int[] b = Arrays.copyOf(a, 6);
        System.out.println("a 数组和 b 数组是否相等: "
            + Arrays.equals(a, b));
        // 输出 b 数组的元素, 将输出 [3, 4, 5, 6, 0, 0]
        System.out.println("b 数组的元素为: "
            + Arrays.toString(b));
        // 将 b 数组的第 3 个元素 (包括) 到第 5 个元素 (不包括) 赋值为 1
        Arrays.fill(b, 2, 4, 1);
        // 输出 b 数组的元素, 将输出 [3, 4, 1, 1, 0, 0]
        System.out.println("b 数组的元素为: "
            + Arrays.toString(b));
        // 对 b 数组进行排序
        Arrays.sort(b);
        // 输出 b 数组的元素, 将输出 [0, 0, 1, 1, 3, 4]
        System.out.println("b 数组的元素为: "
            + Arrays.toString(b));
    }
}
```

★ 注意: ★

`Arrays` 类处于 `java.util` 包下, 为了在程序中使用 `Arrays` 类, 必须在程序中导入 `java.util.Arrays` 类。关于如何导入指定包下的类, 请参考本书第 5 章。为了篇幅考虑, 本书中的程序代码都没有包含 `import` 语句, 读者可参考光盘里对应程序来阅读书中代码。



除此之外，在 `System` 类里也包含了一个 `static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` 方法，该方法可以将 `src` 数组里的元素值赋给 `dest` 数组的元素，其中 `srcPos` 指定从 `src` 数组的第几个元素开始赋值，`length` 参数指定将 `src` 数组的多少个元素值赋给 `dest` 数组的元素。

Java 8 增强了 `Arrays` 类的功能，为 `Arrays` 类增加了一些工具方法，这些工具方法可以充分利用多 CPU 并行的能力来提高设置、排序的性能。下面是 Java 8 为 `Arrays` 类增加的工具方法。

**提示：**

由于计算机硬件的飞速发展，目前几乎所有家用 PC 都是 4 核、8 核的 CPU，而服务器的 CPU 则具有更好的性能，因此 Java 8 与时俱进地增加了并发支持，并发支持可以充分利用硬件设备来提高程序的运行性能。

- `void parallelPrefix(xxx[] array, XxxBinaryOperator op)`: 该方法使用 `op` 参数指定的计算公式计算得到的结果作为新的元素。`op` 计算公式包括 `left`、`right` 两个形参，其中 `left` 代表数组中前一个索引处的元素，`right` 代表数组中当前索引处的元素，当计算第一个新数组元素时，`left` 的值默认为 1。
- `void parallelPrefix(xxx[] array, int fromIndex, int toIndex, XxxBinaryOperator op)`: 该方法与上一个方法相似，区别是该方法仅重新计算 `fromIndex` 到 `toIndex` 索引的元素。
- `void setAll(xxx[] array, IntToXxxFunction generator)`: 该方法使用指定的生成器 (`generator`) 为所有数组元素设置值，该生成器控制数组元素的值的生成算法。
- `void parallelSetAll(xxx[] array, IntToXxxFunction generator)`: 该方法的功能与上一个方法相同，只是该方法增加了并行能力，可以利用多 CPU 并行来提高性能。
- `void parallelSort(xxx[] a)`: 该方法的功能与 `Arrays` 类以前就有的 `sort()` 方法相似，只是该方法增加了并行能力，可以利用多 CPU 并行来提高性能。
- `void parallelSort(xxx[] a, int fromIndex, int toIndex)`: 该方法与上一个方法相似，区别是该方法仅对 `fromIndex` 到 `toIndex` 索引的元素进行排序。
- `Spliterator.OfXxx spliterator(xxx[] array)`: 将该数组的所有元素转换成对应的 `Spliterator` 对象。
- `Spliterator.OfXxx spliterator(xxx[] array, int startInclusive, int endExclusive)`: 该方法与上一个方法相似，区别是该方法仅转换 `startInclusive` 到 `endExclusive` 索引的元素。
- `XxxStream stream(xxx[] array)`: 该方法将数组转换为 `Stream`，`Stream` 是 Java 8 新增的流式编程的 API。
- `XxxStream stream(xxx[] array, int startInclusive, int endExclusive)`: 该方法与上一个方法相似，区别是该方法仅将 `fromIndex` 到 `toIndex` 索引的元素转换为 `Stream`。

上面方法列表中，所有以 `parallel` 开头的方法都表示该方法可利用 CPU 并行的能力来提高性能。上面方法中的 `xxx` 代表不同的数据类型，比如处理 `int[]` 型数组时应将 `xxx` 换成 `int`，处理 `long[]` 型数组时应将 `xxx` 换成 `long`。

下面程序示范了 Java 8 为 `Arrays` 类新增的方法。

**提示：**

下面程序用到了接口、匿名内部类的知识，读者阅读起来可能有一定的困难，此处只要大致知道 `Arrays` 新增的这些新方法就行，暂时并不需要读者立即掌握该程序，可以等到掌握了接口、匿名内部类后再来学习下面程序。

程序清单: codes\04\4.6\ArraysTest2.java

```
public class ArraysTest2
{
    public static void main(String[] args)
    {
        int[] arr1 = new int[]{3, -4, 25, 16, 30, 18};
        // 对数组 arr1 进行并发排序
        Arrays.parallelSort(arr1);
    }
}
```

```

System.out.println(Arrays.toString(arr1));
int[] arr2 = new int[]{3, -4, 25, 16, 30, 18};
Arrays.parallelPrefix(arr2, new IntBinaryOperator()
{
    // left 代表数组中前一个索引处的元素, 计算第一个元素时, left 为 1
    // right 代表数组中当前索引处的元素
    public int applyAsInt(int left, int right)
    {
        return left * right;
    }
});
System.out.println(Arrays.toString(arr2));
int[] arr3 = new int[5];
Arrays.parallelSetAll(arr3, new IntUnaryOperator()
{
    // operand 代表正在计算的元素索引
    public int applyAsInt(int operand)
    {
        return operand * 5;
    }
});
System.out.println(Arrays.toString(arr3));
}

```

上面程序中第一行粗体字代码调用了 `parallelSort()` 方法对数组执行排序, 该方法的功能与传统 `sort()` 方法大致相似, 只是在多 CPU 机器上会有更好的性能。第二段粗体字代码使用的计算公式为 `left * right`, 其中 `left` 代表数组中前一个索引处的元素, `right` 代表数组中当前索引处的元素。程序使用的数组为:

```
{3, -4, 25, 16, 30, 18}
```

计算新的数组元素的方式为:

```
{1*3=3, 3*-4=-12, -12*25=-300, -300*16=-48000, -48000*30=-144000, -144000*18=-2592000}
```

因此将会得到如下新的数组元素:

```
{3, -12, -300, -4800, -144000, -2592000}
```

第三段粗体字代码使用 `operand * 5` 公式来设置数组元素, 该公式中 `operand` 代表正在计算的数组元素的索引。因此第三段粗体字代码计算得到的数组为:

```
{0, 5, 10, 15, 20}
```



提示:

上面两段粗体字代码都可以使用 Lambda 表达式进行简化, 关于 Lambda 表达式的知识请参考本书 6.8 节。

4.6.6 数组的应用举例

数组的用途是很广泛的, 如果程序中有多个类型相同的变量, 而且它们具有逻辑的整体性, 则可以把它们定义成一个数组。

例如, 在实际开发中的一个常用工具函数: 需要将一个浮点数转换成人民币读法字符串, 这个程序就需要使用数组。实现这个函数的思路是, 首先把这个浮点数分成整数部分和小数部分。提取整数部分很容易, 直接将这个浮点数强制类型转换成一个整数即可, 这个整数就是浮点数的整数部分; 再使用浮点数减去整数将可以得到这个浮点数的小数部分。

然后分开处理整数部分和小数部分, 其中小数部分的处理比较简单, 直接截断到保留 2 位数字, 转换成几角几分的字符串。整数部分的处理则稍微复杂一点, 但只要认真分析不难发现, 中国的数字习惯是 4 位一节的, 一个 4 位的数字可被转成几千几百几十, 至于后面添加什么单位则不确定, 如果这节 4 位数字出现在 1~4 位, 则后面添加单位元; 如果这节 4 位数字出现在 5~8 位, 则后面添加单位万; 如果这节 4 位数字出现在 9~12 位, 则后面添加单位亿; 多于 12 位就暂不考虑了。

因此实现这个程序的关键就是把一个 4 位数字字符串转换成一个中文读法。下面程序把这个需求实现了一部分。

程序清单: codes\04\4.6\Num2Rmb.java

```

public class Num2Rmb
{
    private String[] hanArr = {"零", "壹", "贰", "叁", "肆",
        "伍", "陆", "柒", "捌", "玖"};
    private String[] unitArr = {"十", "百", "千"};
    /**
     * 把一个浮点数分解成整数部分和小数部分字符串
     * @param num 需要被分解的浮点数
     * @return 分解出来的整数部分和小数部分。第一个数组元素是整数部分，第二个数组元素是小数部分
     */
    private String[] divide(double num)
    {
        // 将一个浮点数强制类型转换为 long 型，即得到它的整数部分
        long zheng = (long)num;
        // 浮点数减去整数部分，得到小数部分，小数部分乘以 100 后再取整得到 2 位小数
        long xiao = Math.round((num - zheng) * 100);
        // 下面用了 2 种方法把整数转换为字符串
        return new String[]{zheng + "", String.valueOf(xiao)};
    }
    /**
     * 把一个四位的数字字符串变成汉字字符串
     * @param numStr 需要被转换的四位的数字字符串
     * @return 四位的数字字符串被转换成汉字字符串
     */
    private String toHanStr(String numStr)
    {
        String result = "";
        int numLen = numStr.length();
        // 依次遍历数字字符串的每一位数字
        for (int i = 0 ; i < numLen ; i++ )
        {
            // 把 char 型数字转换成 int 型数字，因为它们的 ASCII 码值恰好相差 48
            // 因此把 char 型数字减去 48 得到 int 型数字，例如 '4' 被转换成 4
            int num = numStr.charAt(i) - 48;
            // 如果不是最后一位数字，而且数字不是零，则需要添加单位（千、百、十）
            if ( i != numLen - 1 && num != 0 )
            {
                result += hanArr[num] + unitArr[numLen - 2 - i];
            }
            // 否则不要添加单位
            else
            {
                result += hanArr[num];
            }
        }
        return result;
    }
    public static void main(String[] args)
    {
        Num2Rmb nr = new Num2Rmb();
        // 测试把一个浮点数分解成整数部分和小数部分
        System.out.println(Arrays.toString(nr.divide(236711125.123)));
        // 测试把一个四位的数字字符串变成汉字字符串
        System.out.println(nr.toHanStr("6109"));
    }
}

```

运行上面程序，看到如下运行结果：

```
[236711125, 12]
```

```
陆仟壹佰零玖
```

从上面程序的运行结果来看，初步实现了所需功能，但这个程序并不是这么简单，对零的处理比较复杂。例如，有两个零连在一起时该如何处理呢？如果最高位是零如何处理呢？最低位是零又如何处理呢？因此这个程序还需要继续完善，希望读者能把这个程序写完。

除此之外，还可以利用二维数组来完成五子棋、连连看、俄罗斯方块、扫雷等常见小游戏。下面简单介绍利用二维数组实现五子棋。先定义一个二维数组作为下棋的棋盘，每当一个棋手下一步棋后，也就是为二维数组的一个数组元素赋值。下面程序完成了这个程序的初步功能。

程序清单：codes\04\4.6\Gobang.java

```
public class Gobang
{
    // 定义棋盘的大小
    private static int BOARD_SIZE = 15;
    // 定义一个二维数组来充当棋盘
    private String[][] board;
    public void initBoard()
    {
        // 初始化棋盘数组
        board = new String[BOARD_SIZE][BOARD_SIZE];
        // 把每个元素赋为"十"，用于在控制台画出棋盘
        for (int i = 0 ; i < BOARD_SIZE ; i++)
        {
            for ( int j = 0 ; j < BOARD_SIZE ; j++)
            {
                board[i][j] = "十";
            }
        }
    }
    // 在控制台输出棋盘的方法
    public void printBoard()
    {
        // 打印每个数组元素
        for (int i = 0 ; i < BOARD_SIZE ; i++)
        {
            for ( int j = 0 ; j < BOARD_SIZE ; j++)
            {
                // 打印数组元素后不换行
                System.out.print(board[i][j]);
            }
            // 每打印完一行数组元素后输出一个换行符
            System.out.print("\n");
        }
    }
    public static void main(String[] args) throws Exception
    {
        Gobang gb = new Gobang();
        gb.initBoard();
        gb.printBoard();
        // 这是用于获取键盘输入的方法
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String inputStr = null;
        // br.readLine(): 每当在键盘上输入一行内容后按回车键，刚输入的内容将被 br 读取到
        while ((inputStr = br.readLine()) != null)
        {
            // 将用户输入的字符串以逗号(,)作为分隔符，分隔成2个字符串
            String[] posStrArr = inputStr.split(",");
            // 将2个字符串转换为用户下棋的坐标
            int xPos = Integer.parseInt(posStrArr[0]);
            int yPos = Integer.parseInt(posStrArr[1]);
            // 把对应的数组元素赋为"●"。
            gb.board[yPos - 1][xPos - 1] = "●";
            /*
            电脑随机生成2个整数，作为电脑下棋的坐标，赋给 board 数组
            还涉及
            1. 坐标的有效性，只能是数字，不能超出棋盘范围
            2. 下的棋的点，不能重复下棋
            3. 每次下棋后，需要扫描谁赢了
            */
            gb.printBoard();
            System.out.println("请输入您下棋的坐标，应以 x,y 的格式：");
        }
    }
}
```


第5章

面向对象（上）

本章要点

- 定义类、成员变量和方法
- 创建并使用对象
- 对象和引用
- 方法必须属于类或对象
- Java 方法的参数传递机制
- 递归方法
- 方法的重载
- 实现良好的封装
- 使用 `package` 和 `import`
- 构造器的作用和构造器重载
- 继承的特点和用法
- 重写父类方法
- `super` 关键字的用法
- 继承和多态
- 向上转型和强制类型转换
- 继承和组合的关系
- 使用组合来实现复用
- 构造器和初始化块的作用及区别
- 静态初始化块

Java 是面向对象的程序设计语言, Java 语言提供了定义类、成员变量、方法等最基本的功能。类可被认为是一种自定义的数据类型, 可以使用类来定义变量, 所有使用类定义的变量都是引用变量, 它们将会引用到类的对象。类用于描述客观世界里某一类对象的共同特征, 而对象则是类的具体存在, Java 程序使用类的构造器来创建该类的对象。

Java 也支持面向对象的三大特征: 封装、继承和多态, Java 提供了 `private`、`protected` 和 `public` 三个访问控制修饰符来实现良好的封装, 提供了 `extends` 关键字来让子类继承父类, 子类继承父类就可以继承到父类的成员变量和方法, 如果访问控制允许, 子类实例可以直接调用父类里定义的方法。继承是实现类复用的重要手段, 除此之外, 也可通过组合关系来实现这种复用, 从某种程度上来看, 继承和组合具有相同的功能。使用继承关系来实现复用时, 子类对象可以直接赋给父类变量, 这个变量具有多态性, 编程更加灵活; 而利用组合关系来实现复用时, 则不具备这种灵活性。

构造器用于对类实例进行初始化操作, 构造器支持重载, 如果多个重载的构造器里包含了相同的初始化代码, 则可以把这些初始化代码放置在普通初始化块里完成, 初始化块总在构造器执行之前被调用。除此之外, Java 还提供了一种静态初始化块, 静态初始化块用于初始化类, 在类初始化阶段被执行。如果继承树里的某一个类需要被初始化时, 系统将会同时初始化该类的所有父类。

5.1 类和对象

Java 是面向对象的程序设计语言, 类是面向对象的重要内容, 可以把类当成一种自定义类型, 可以使用类来定义变量, 这种类型的变量统称为引用变量。也就是说, 所有类是引用类型。

5.1.1 定义类

面向对象的程序设计过程中有两个重要概念: 类(class) 和对象(object, 也被称为实例, instance), 其中类是某一批对象的抽象, 可以把类理解成某种概念; 对象才是一个具体存在的实体, 从这个意义上来看, 日常所说的人, 其实都是人的实例, 而不是人类。

Java 语言是面向对象的程序设计语言, 类和对象是面向对象的核心。Java 语言提供了对创建类和创建对象简单的语法支持。

Java 语言里定义类的简单语法如下:

```
[修饰符] class 类名
{
    零个到多个构造器定义...
    零个到多个成员变量...
    零个到多个方法...
}
```

在上面的语法格式中, 修饰符可以是 `public`、`final`、`abstract`, 或者完全省略这三个修饰符, 类名只要是一个合法的标识符即可, 但这仅仅满足的是 Java 的语法要求; 如果从程序的可读性方面来看, Java 类名必须是由一个或多个有意义的单词连缀而成的, 每个单词首字母大写, 其他字母全部小写, 单词与单词之间不要使用任何分隔符。

对一个类定义而言, 可以包含三种最常见的成员: 构造器、成员变量和方法, 三种成员都可以定义零个或多个, 如果三种成员都只定义零个, 就是定义了一个空类, 这没有太大的实际意义。

类里各成员之间的定义顺序没有任何影响, 各成员之间可以相互调用, 但需要指出的是, `static` 修饰的成员不能访问没有 `static` 修饰的成员。

成员变量用于定义该类或该类的实例所包含的状态数据, 方法则用于定义该类或该类的实例的行为特征或者功能实现。构造器用于构造该类的实例, Java 语言通过 `new` 关键字来调用构造器, 从而返回该类的实例。

构造器是一个类创建对象的根本途径, 如果一个类没有构造器, 这个类通常无法创建实例。因此, Java 语言提供了一个功能: 如果程序员没有为一个类编写构造器, 则系统会为该类提供一个默认的构造器。一旦程序员为一个类提供了构造器, 系统将不再为该类提供构造器。

定义成员变量的语法格式如下:

```
[修饰符] 类型 成员变量名 [= 默认值];
```

对定义成员变量语法格式的详细说明如下。

- 修饰符: 修饰符可以省略, 也可以是 `public`、`protected`、`private`、`static`、`final`, 其中 `public`、`protected`、`private` 三个最多只能出现其中之一, 可以与 `static`、`final` 组合起来修饰成员变量。
- 类型: 类型可以是 Java 语言允许的任何数据类型, 包括基本类型和现在介绍的引用类型。
- 成员变量名: 成员变量名只要是一个合法的标识符即可, 但这只是从语法角度说的; 如果从程序可读性角度来看, 成员变量名应该由一个或多个有意义的单词连缀而成, 第一个单词首字母小写, 后面每个单词首字母大写, 其他字母全部小写, 单词与单词之间不要使用任何分隔符。成员变量用于描述类或对象包含的状态数据, 因此成员变量名建议使用英文名词。
- 默认值: 定义成员变量还可以指定一个可选的默认值。



注意:

成员变量由英文单词 `field` 意译而来, 早期有些书籍将成员变量称为属性。但实际上在 Java 世界里属性 (由 `property` 翻译而来) 指的是一组 `setter` 方法和 `getter` 方法。比如说某个类有 `age` 属性, 意味着该类包含 `setAge()` 和 `getAge()` 两个方法。另外, 也有些资料、书籍将 `field` 翻译为字段、域。



定义方法的语法格式如下:

```
[修饰符] 方法返回值类型 方法名(形参列表)
{
    // 由零条到多条可执行性语句组成的方法体
}
```

对定义方法语法格式的详细说明如下。

- 修饰符: 修饰符可以省略, 也可以是 `public`、`protected`、`private`、`static`、`final`、`abstract`, 其中 `public`、`protected`、`private` 三个最多只能出现其中之一; `abstract` 和 `final` 最多只能出现其中之一, 它们可以与 `static` 组合起来修饰方法。
- 方法返回值类型: 返回值类型可以是 Java 语言允许的任何数据类型, 包括基本类型和引用类型; 如果声明了方法返回值类型, 则方法体内必须有一个有效的 `return` 语句, 该语句返回一个变量或一个表达式, 这个变量或者表达式的类型必须与此处声明的类型匹配。除此之外, 如果一个方法没有返回值, 则必须使用 `void` 来声明没有返回值。
- 方法名: 方法名的命名规则与成员变量的命名规则基本相同, 但由于方法用于描述该类或该类的实例的行为特征或功能实现, 因此通常建议方法名以英文动词开头。
- 形参列表: 形参列表用于定义该方法可以接受的参数, 形参列表由零组到多组“参数类型 形参名”组合而成, 多组参数之间以英文逗号 (,) 隔开, 形参类型和形参名之间以英文空格隔开。一旦在定义方法时指定了形参列表, 则调用该方法时必须传入对应的参数值——谁调用方法, 谁负责为形参赋值。

方法体里多条可执行性语句之间有严格的执行顺序, 排在方法体前面的语句总是先执行, 排在方法体后面的语句总是后执行。

`static` 是一个特殊的关键字, 它可用于修饰方法、成员变量等成员。`static` 修饰的成员表明它属于这个类本身, 而不属于该类的单个实例, 因为通常把 `static` 修饰的成员变量和方法也称为类变量、类方法。不使用 `static` 修饰的普通方法、成员变量则属于该类的单个实例, 而不属于该类。因为通常把不使用 `static` 修饰的成员变量和方法也称为实例变量、实例方法。

由于 `static` 的英文直译就是静态的意思, 因此有时也把 `static` 修饰的成员变量和方法称为静态变量和静态方法, 把不使用 `static` 修饰的成员变量和方法称为非静态变量和非静态方法。静态成员不能直接访问非静态成员。

**提示:**

虽然绝大部分资料都喜欢把 static 称为静态,但实际上这种说法很模糊,完全无法说明 static 的真正作用。static 的真正作用就是用于区分成员变量、方法、内部类、初始化块(本书后面会介绍后两种成员)这四种成员到底属于类本身还是属于实例。在类中定义的成员,static 相当于一个标志,有 static 修饰的成员属于类本身,没有 static 修饰的成员属于该类的实例。

构造器是一个特殊的方法,定义构造器的语法格式与定义方法的语法格式很像,定义构造器的语法格式如下:

```
[修饰符] 构造器名(形参列表)
{
    // 由零条到多条可执行性语句组成的构造器执行体
}
```

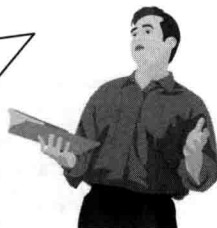
对定义构造器语法格式的详细说明如下。

- 修饰符: 修饰符可以省略,也可以是 public、protected、private 其中之一。
- 构造器名: 构造器名必须和类名相同。
- 形参列表: 和定义方法形参列表的格式完全相同。

值得指出的是,构造器既不能定义返回值类型,也不能使用 void 声明构造器没有返回值。如果为构造器定义了返回值类型,或使用 void 声明构造器没有返回值,编译时不会出错,但 Java 会把这个所谓的构造器当成方法来处理——它就不再是构造器。

学生提问: 构造器不是没有返回值吗? 为什么不能用 void 声明呢?

答: 简单地说,这是 Java 的语法规定。实际上,类的构造器是有返回值的,当使用 new 关键字来调用构造器时,构造器返回该类的实例,可以把这个类的实例当成构造器的返回值,因此构造器的返回值类型总是当前类,无须定义返回值类型。但必须注意: 不要在构造器里显式使用 return 来返回当前类的对象,因为构造器的返回值是隐式的。



下面程序将定义一个 Person 类。

程序清单: codes\05\5.1\Person.java

```
public class Person
{
    // 下面定义了两个成员变量
    public String name;
    public int age;
    // 下面定义了一个 say 方法
    public void say(String content)
    {
        System.out.println(content);
    }
}
```

上面的 Person 类代码里没有定义构造器,系统将为它提供一个默认的构造器,系统提供的构造器总是没有参数的。

定义类之后,接下来即可使用该类了,Java 的类大致有如下作用。

- 定义变量。
- 创建对象。
- 调用类的类方法或访问类的类变量。

下面先介绍使用类来定义变量和创建对象。

5.1.2 对象的产生和使用

创建对象的根本途径是构造器，通过 `new` 关键字来调用某个类的构造器即可创建这个类的实例。

程序清单：codes\05\5.1\PersonTest.java

```
// 使用 Person 类定义一个 Person 类型的变量
Person p;
// 通过 new 关键字调用 Person 类的构造器，返回一个 Person 实例
// 将该 Person 实例赋给 p 变量
p = new Person();
```

上面代码也可简写成如下形式：

```
// 定义 p 变量的同时并为 p 变量赋值
Person p = new Person();
```

创建对象之后，接下来即可使用该对象了，Java 的对象大致有如下作用。

- 访问对象的实例变量。
- 调用对象的方法。

如果访问权限允许，类里定义的方法和成员变量都可以通过类或实例来调用。类或实例访问方法或成员变量的语法是：类.类变量|方法，或者实例.实例变量|方法，在这种方式中，类或实例是主调者，用于访问该类或该实例的成员变量或方法。

`static` 修饰的方法和成员变量，既可通过类来调用，也可通过实例来调用；没有使用 `static` 修饰的普通方法和成员变量，只可通过实例来调用。下面代码中通过 `Person` 实例来调用 `Person` 的成员变量和方法（程序清单同上）。

```
// 访问 p 的 name 实例变量，直接为该变量赋值
p.name = "李刚";
// 调用 p 的 say() 方法，声明 say() 方法时定义了一个形参
// 调用该方法必须为形参指定一个值
p.say("Java 语言很简单，学习很容易！");
// 直接输出 p 的 name 实例变量，将输出 李刚
System.out.println(p.name);
```

上面代码中通过 `Person` 实例调用了 `say()` 方法，调用方法时必须为方法的形参赋值。因此在这行代码中调用 `Person` 对象的 `say()` 方法时，必须为 `say()` 方法传入一个字符串作为形参的参数值，这个字符串将被给 `content` 参数。

大部分时候，定义一个类就是为了重复创建该类的实例，同一个类的多个实例具有相同的特征，而类则是定义了多个实例的共同特征。从某个角度来看，类定义的是多个实例的特征，因此类不是一种具体存在，实例才是具体存在。完全可以这样说：你不是人这个类，我也不是人这个类，我们都只是人的实例。

5.1.3 对象、引用和指针

在前面 `PersonTest.java` 代码中，有这样一行代码：`Person p = new Person();`，这行代码创建了一个 `Person` 实例，也被称为 `Person` 对象，这个 `Person` 对象被赋给 `p` 变量。

在这行代码中实际产生了两个东西：一个是 `p` 变量，一个是 `Person` 对象。

从 `Person` 类定义来看，`Person` 对象应包含两个实例变量，而变量是需要内存来存储的。因此，当创建 `Person` 对象时，必然需要有对应的内存来存储 `Person` 对象的实例变量。图 5.1 显示了 `Person` 对象在内存中的存储示意图。

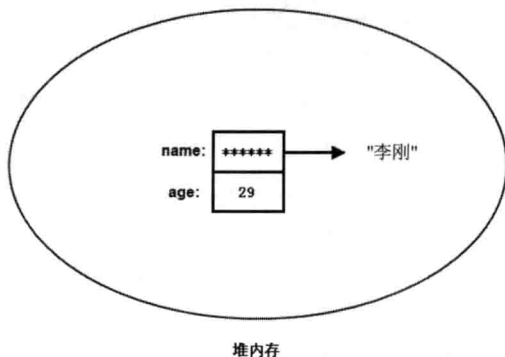


图 5.1 `Person` 对象的内存存储示意图

从图 5.1 中可以看出, Person 对象由多块内存组成,不同内存块分别存储了 Person 对象的不同成员变量。当把这个 Person 对象赋值给一个引用变量时,系统如何处理呢?难道系统会把这个 Person 对象在内存里重新复制一份吗?显然不会,Java 没有这么笨,Java 让引用变量指向这个对象即可。也就是说,引用变量里存放的仅仅是一个引用,它指向实际的对象。

与前面介绍的数组类型类似,类也是一种引用数据类型,因此程序中定义的 Person 类型的变量实际上是一个引用,它被存放在栈内存里,指向实际的 Person 对象;而真正的 Person 对象则存放在堆(heap)内存中。图 5.2 显示了将 Person 对象赋给一个引用变量的示意图。

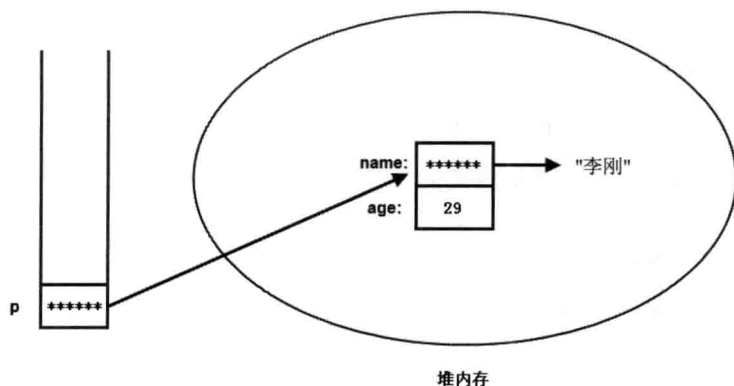


图 5.2 引用变量指向实际对象的示意图

栈内存里的引用变量并未真正存储对象的成员变量,对象的成员变量数据实际存放在堆内存里;而引用变量只是指向该堆内存里的对象。从这个角度来看,引用变量与 C 语言里的指针很像,它们都是存储一个地址值,通过这个地址来引用到实际对象。实际上,Java 里的引用就是 C 里的指针,只是 Java 语言把这个指针封装起来,避免开发者进行烦琐的指针操作。

当一个对象被创建成功以后,这个对象将保存在堆内存中,Java 程序不允许直接访问堆内存中的对象,只能通过该对象的引用操作该对象。也就是说,不管是数组还是对象,都只能通过引用来访问它们。

如图 5.2 所示,p 引用变量本身只存储了一个地址值,并未包含任何实际数据,但它指向实际的 Person 对象,当访问 p 引用变量的成员变量和方法时,实际上是访问 p 所引用对象的成员变量和方法。



提示:
不管是数组还是对象,当程序访问引用变量的成员变量或方法时,实际上是访问该引用变量所引用的数组、对象的成员变量或方法。
:.....

堆内存里的对象可以有多个引用,即多个引用变量指向同一个对象,代码如下(程序清单同上):

```
// 将 p 变量的值赋值给 p2 变量  
Person p2 = p;
```

上面代码把 p 变量的值赋值给 p2 变量,也就是将 p 变量保存的地址值赋给 p2 变量,这样 p2 变量和 p 变量将指向堆内存里的同一个 Person 对象。不管访问 p2 变量的成员变量和方法,还是访问 p 变量的成员变量和方法,它们实际上是访问同一个 Person 对象的成员变量和方法,将会返回相同的访问结果。

如果堆内存里的对象没有任何变量指向该对象,那么程序将无法再访问该对象,这个对象也就变成了垃圾,Java 的垃圾回收机制将回收该对象,释放该对象所占的内存区。

因此,如果希望通知垃圾回收机制回收某个对象,只需切断该对象的所有引用变量和它之间的关系即可,也就是把这些引用变量赋值为 null。

5.1.4 对象的 this 引用

Java 提供了一个 this 关键字, this 关键字总是指向调用该方法的对象。根据 this 出现位置的不同, this 作为对象的默认引用有两种情形。

- 构造器中引用该构造器正在初始化的对象。
- 在方法中引用调用该方法的对象。

this 关键字最大的作用就是让类中一个方法，访问该类里的另一个方法或实例变量。假设定义了一个 Dog 类，这个 Dog 对象的 run() 方法需要调用它的 jump() 方法，那么应该如何做？是否应该定义如下的 Dog 类呢？

程序清单：codes\05\5.1\Dog.java

```
public class Dog
{
    // 定义一个 jump() 方法
    public void jump()
    {
        System.out.println("正在执行 jump 方法");
    }
    // 定义一个 run() 方法，run() 方法需要借助 jump() 方法
    public void run()
    {
        Dog d = new Dog();
        d.jump();
        System.out.println("正在执行 run 方法");
    }
}
```

使用这种方式来定义这个 Dog 类，确实可以实现在 run() 方法中调用 jump() 方法。那么这种做法是否够好呢？下面再提供一个程序来创建 Dog 对象，并调用该对象的 run() 方法。

程序清单：codes\05\5.1\DogTest.java

```
public class DogTest
{
    public static void main(String[] args)
    {
        // 创建 Dog 对象
        Dog dog = new Dog();
        // 调用 Dog 对象的 run() 方法
        dog.run();
    }
}
```

在上面的程序中，一共产生了两个 Dog 对象，在 Dog 类的 run() 方法中，程序创建了一个 Dog 对象，并使用名为 d 的引用变量来指向该 Dog 对象；在 DogTest 的 main() 方法中，程序再次创建了一个 Dog 对象，并使用名为 dog 的引用变量来指向该 Dog 对象。

这里产生了两个问题。第一个问题：在 run() 方法中调用 jump() 方法时是否一定需要一个 Dog 对象？第二个问题：是否一定需要重新创建一个 Dog 对象？第一个问题的答案是肯定的，因为没有使用 static 修饰的成员变量和方法都必须使用对象来调用。第二个问题的答案是否定的，因为当程序调用 run() 方法时，一定会提供一个 Dog 对象，这样就可以直接使用这个已经存在的 Dog 对象，而无须重新创建新的 Dog 对象了。

因此需要在 run() 方法中获得调用该方法的对象，通过 this 关键字就可以满足这个要求。

this 可以代表任何对象，当 this 出现在某个方法体中时，它所代表的对象是不确定的，但它的类型是确定的，它所代表的对象只能是当前类；只有当这个方法被调用时，它所代表的对象才被确定下来：谁在调用这个方法，this 就代表谁。

将前面的 Dog 类的 run() 方法改为如下形式会更加合适。

程序清单：codes\05\5.1\Dog.java

```
// 定义一个 run() 方法，run() 方法需要借助 jump() 方法
public void run()
{
    // 使用 this 引用调用 run() 方法的对象
    this.jump();
}
```

```
        System.out.println("正在执行 run 方法");  
    }
```

采用上面方法定义的 Dog 类更符合实际意义。从前一种 Dog 类定义来看,在 Dog 对象的 run()方法内重新创建了一个新的 Dog 对象,并调用它的 jump()方法,这意味着一个 Dog 对象的 run()方法需要依赖于另一个 Dog 对象的 jump()方法,这不符合逻辑。上面的代码更符合实际情形:当一个 Dog 对象调用 run()方法时,run()方法需要依赖它自己的 jump()方法。

在现实世界里,对象的一个方法依赖于另一个方法的情形如此常见:例如,吃饭方法依赖于拿筷子方法,写程序方法依赖于敲键盘方法,这种依赖都是同一个对象两个方法之间的依赖。因此,Java 允许对象的一个成员直接调用另一个成员,可以省略 this 前缀。也就是说,将上面的 run()方法改为如下形式也完全正确。

```
public void run()  
{  
    jump();  
    System.out.println("正在执行 run 方法");  
}
```

大部分时候,一个方法访问该类中定义的其他方法、成员变量时加不加 this 前缀的效果是完全一样的。

对于 static 修饰的方法而言,则可以使用类来直接调用该方法,如果在 static 修饰的方法中使用 this 关键字,则这个关键字就无法指向合适的对象。所以,static 修饰的方法中不能使用 this 引用。由于 static 修饰的方法不能使用 this 引用,所以 static 修饰的方法不能访问不使用 static 修饰的普通成员,因此 Java 语法规则:静态成员不能直接访问非静态成员。



提示:

省略 this 前缀只是一种假象,虽然程序员省略了调用 jump()方法之前的 this,但实际上这个 this 依然是存在的。根据汉语语法习惯:完整的语句至少包括主语、谓语、宾语,在面向对象的世界里,主、谓、宾的结构完全成立,例如“猪八戒吃西瓜”是一条汉语语句,转换为面向对象的语法,就可以写成“猪八戒.吃(西瓜);”,因此本书常常把调用成员变量、方法的对象称为“主调(主语调用者的简称)”。对于 Java 语言来说,调用成员变量、方法时,主调是必不可少的,即使代码中省略了主调,但实际的主调依然存在。一般来说,如果调用 static 修饰的成员(包括方法、成员变量)时省略了前面的主调,那么默认使用该类作为主调;如果调用没有 static 修饰的成员(包括方法、成员变量)时省略了前面的主调,那么默认使用 this 作为主调。

下面程序演示了静态方法直接访问非静态方法时引发的错误。

程序清单: codes\05\5.1\StaticAccessNonStatic.java

```
public class StaticAccessNonStatic  
{  
    public void info()  
    {  
        System.out.println("简单的 info 方法");  
    }  
    public static void main(String[] args)  
    {  
        // 因为 main()方法是静态方法,而 info()是非静态方法  
        // 调用 main()方法的是该类本身,而不是该类的实例  
        // 因此省略的 this 无法指向有效的对象  
        info();  
    }  
}
```

编译上面的程序,系统提示在 info();代码行出现如下错误:

无法从静态上下文中引用非静态 方法 info()

上面错误正是因为 `info()` 方法是属于实例的方法，而不是属于类的方法，因此必须使用对象来调用该方法。在上面的 `main()` 方法中直接调用 `info()` 方法时，系统相当于使用 `this` 作为该方法的调用者，而 `main()` 方法是一个 `static` 修饰的方法，`static` 修饰的方法属于类，而不属于对象，因此调用 `static` 修饰的方法的主调总是类本身；如果允许在 `static` 修饰的方法中出现 `this` 引用，那将导致 `this` 无法引用有效的对象，因此上面程序出现编译错误。



Java 有一个让人极易“混淆”的语法，它允许使用对象来调用 `static` 修饰的成员变量、方法，但实际上这是不应该的。前面已经介绍过，`static` 修饰的成员属于类本身，而不属于该类的实例，既然 `static` 修饰的成员完全不属于该类的实例，那么就不应该允许使用实例去调用 `static` 修饰的成员变量和方法！所以请读者牢记一点：Java 编程时不要使用对象去调用 `static` 修饰的成员变量、方法，而是应该使用类去调用 `static` 修饰的成员变量、方法！如果在其他 Java 代码中看到对象调用 `static` 修饰的成员变量、方法的情形，则完全可以把这种用法当成假象，将其替换成用类来调用 `static` 修饰的成员变量、方法的代码。



如果确实需要在静态方法中访问另一个普通方法，则只能重新创建一个对象。例如，将上面的 `info()` 调用改为如下形式：

```
// 创建一个对象作为调用者来调用 info() 方法
new StaticAccessNonStatic().info();
```

大部分时候，普通方法访问其他方法、成员变量时无须使用 `this` 前缀，但如果方法里有个局部变量和成员变量同名，但程序又需要在该方法里访问这个被覆盖的成员变量，则必须使用 `this` 前缀。关于局部变量覆盖成员变量的情形，参见 5.3 节的内容。

除此之外，`this` 引用也可以用于构造器中作为默认引用，由于构造器是直接使用 `new` 关键字来调用，而不是使用对象来调用的，所以 `this` 在构造器中代表该构造器正在初始化的对象。

程序清单：codes\05\5.1\ThisInConstructor.java

```
public class ThisInConstructor
{
    // 定义一个名为 foo 的成员变量
    public int foo;
    public ThisInConstructor()
    {
        // 在构造器里定义一个 foo 变量
        int foo = 0;
        // 使用 this 代表该构造器正在初始化的对象
        // 下面的代码将会把该构造器正在初始化的对象的 foo 成员变量设为 6
        this.foo = 6;
    }
    public static void main(String[] args)
    {
        // 所有使用 ThisInConstructor 创建的对象的 foo 成员变量
        // 都将被设为 6，所以下面代码将输出 6
        System.out.println(new ThisInConstructor().foo);
    }
}
```

在 `ThisInConstructor` 构造器中使用 `this` 引用时，`this` 总是引用该构造器正在初始化的对象。程序粗体字标识代码行将正在执行初始化的 `ThisInConstructor` 对象的 `foo` 成员变量设为 6，这意味着该构造器返回的所有对象的 `foo` 成员变量都等于 6。

与普通方法类似的是，大部分时候，在构造器中访问其他成员变量和方法时都可以省略 `this` 前缀，但如果构造器中有一个与成员变量同名的局部变量，又必须在构造器中访问这个被覆盖的成员变量，则必须使用 `this` 前缀。如上面的 `ThisInConstructor.java` 所示。

当 `this` 作为对象的默认引用使用时，程序可以像访问普通引用变量一样来访问这个 `this` 引用，甚至

可以把 `this` 当成普通方法的返回值。看下面程序：

程序清单：codes\05\5.1\ReturnThis.java

```
public class ReturnThis
{
    public int age;
    public ReturnThis grow()
    {
        age++;
        // return this 返回调用该方法的对象
        return this;
    }
    public static void main(String[] args)
    {
        ReturnThis rt = new ReturnThis();
        // 可以连续调用同一个方法
        rt.grow()
           .grow()
           .grow();
        System.out.println("rt 的 age 成员变量值是：" + rt.age);
    }
}
```

从上面程序中可以看出，如果在某个方法中把 `this` 作为返回值，则可以多次连续调用同一个方法，从而使得代码更加简洁。但是，这种把 `this` 作为返回值的方法可能造成实际意义的模糊，例如上面的 `grow` 方法，用于表示对象的生长，即 `age` 成员变量的值加 1，实际上不应该有返回值。

注意：

使用 `this` 作为方法的返回值可以让代码更加简洁，但可能造成实际意义的模糊。



5.2 方法详解

方法是类或对象的行为特征的抽象，方法是类或对象最重要的组成部分。但从功能上来看，方法完全类似于传统结构化程序设计里的函数。值得指出的是，Java 里的方法不能独立存在，所有的方法都必须定义在类里。方法在逻辑上要么属于类，要么属于对象。

5.2.1 方法的所属性

不论是从定义方法的语法来看，还是从方法的功能来看，都不难发现方法和函数之间的相似性。实际上，方法确实是由传统的函数发展而来的，方法与传统的函数有着显著不同：在结构化编程语言里，函数是一等公民，整个软件由一个个的函数组成；在面向对象编程语言里，类才是一等公民，整个系统由一个个的类组成。因此在 Java 语言里，方法不能独立存在，方法必须属于类或对象。

因此，如果需要定义方法，则只能在类体内定义，不能独立定义一个方法。一旦将一个方法定义在某个类的类体内，如果这个方法使用了 `static` 修饰，则这个方法属于这个类，否则这个方法属于这个类的实例。

Java 语言是静态的。一个类定义完成后，只要不再重新编译这个类文件，该类和该类的对象所拥有的方法是固定的，永远都不会改变。

因为 Java 里的方法不能独立存在，它必须属于一个类或一个对象，因此方法也不能像函数那样被独立执行，执行方法时必须使用类或对象来作为调用者，即所有方法都必须使用“类.方法”或“对象.方法”的形式来调用。这里可能产生一个问题：同一个类里不同方法之间相互调用时，不就可以直接调用吗？这里需要指出：同一个类的一个方法调用另外一个方法时，如果被调方法是普通方法，则默认使用 `this` 作为调用者；如果被调方法是静态方法，则默认使用类作为调用者。也就是说，表面上看起来某些方法可以被独立执行，但实际上还是使用 `this` 或者类来作为调用者。

永远不要把方法当成独立存在的实体,正如现实世界由类和对象组成,而方法只能作为类和对象的附属,Java 语言里的方法也是一样。Java 语言里方法的所属性主要体现在如下几个方面。

- 方法不能独立定义,方法只能在类体里定义。
- 从逻辑意义上来看,方法要么属于该类本身,要么属于该类的一个对象。
- 永远不能独立执行方法,执行方法必须使用类或对象作为调用者。

使用 `static` 修饰的方法属于这个类本身,使用 `static` 修饰的方法既可以使用类作为调用者来调用,也可以使用对象作为调用者来调用。但值得指出的是,因为使用 `static` 修饰的方法还是属于这个类的,因此使用该类的任何对象来调用这个方法时将会得到相同的执行结果,这是由于底层依然是使用这些实例所属的类作为调用者。

没有 `static` 修饰的方法则属于该类的对象,不属于这个类本身。因此没有 `static` 修饰的方法只能使用对象作为调用者来调用,不能使用类作为调用者来调用。使用不同对象作为调用者来调用同一个普通方法,可能得到不同的结果。

➤➤ 5.2.2 方法的参数传递机制

前面已经介绍了 Java 里的方法是不能独立存在的,调用方法也必须使用类或对象作为主调者。如果声明方法时包含了形参声明,则调用方法时必须给这些形参指定参数值,调用方法时实际传给形参的参数值也被称为实参。

那么,Java 的实参值是如何传入方法的呢?这是由 Java 方法的参数传递机制来控制的,Java 里方法的参数传递方式只有一种:值传递。所谓值传递,就是将实际参数值的副本(复制品)传入方法内,而参数本身不会受到任何影响。



提示:

Java 里的参数传递类似于《西游记》里的孙悟空,孙悟空复制了一个假孙悟空,这个假孙悟空具有和孙悟空相同的能力,可除妖或被砍头。但不管这个假孙悟空遇到什么事,真孙悟空不会受到任何影响。与此类似,传入方法的是实际参数值的复制品,不管方法中对这个复制品如何操作,实际参数值本身不会受到任何影响。

下面程序演示了方法参数传递的效果。

程序清单: codes\05\5.2\PrimitiveTransferTest.java

```
public class PrimitiveTransferTest
{
    public static void swap(int a , int b)
    {
        // 下面三行代码实现 a、b 变量的值交换
        // 定义一个临时变量来保存 a 变量的值
        int tmp = a;
        // 把 b 的值赋给 a
        a = b;
        // 把临时变量 tmp 的值赋给 a
        b = tmp;
        System.out.println("swap 方法里, a 的值是"
            + a + "; b 的值是" + b);
    }
    public static void main(String[] args)
    {
        int a = 6;
        int b = 9;
        swap(a , b);
        System.out.println("交换结束后, 变量 a 的值是"
            + a + "; 变量 b 的值是" + b);
    }
}
```

运行上面程序,看到如下运行结果:

swap 方法里, a 的值是 9; b 的值是 6
 交换结束后, 变量 a 的值是 6; 变量 b 的值是 9

从上面运行结果来看, swap()方法里 a 和 b 的值是 9、6, 交换结束后, 变量 a 和 b 的值依然是 6、9。从这个运行结果可以看出, main()方法里的变量 a 和 b, 并不是 swap()方法里的 a 和 b。正如前面讲的, swap()方法的 a 和 b 只是 main()方法里变量 a 和 b 的复制品。下面通过示意图来说明上面程序的执行过程。Java 程序总是从 main()方法开始执行, main()方法开始定义了 a、b 两个局部变量, 两个变量在内存中的存储示意图如图 5.3 所示。

当程序执行 swap()方法时, 系统进入 swap()方法, 并将 main()方法中的 a、b 变量作为参数值传入 swap()方法, 传入 swap()方法的只是 a、b 的副本, 而不是 a、b 本身, 进入 swap()方法后系统中产生了 4 个变量, 这 4 个变量在内存中的存储示意图如图 5.4 所示。

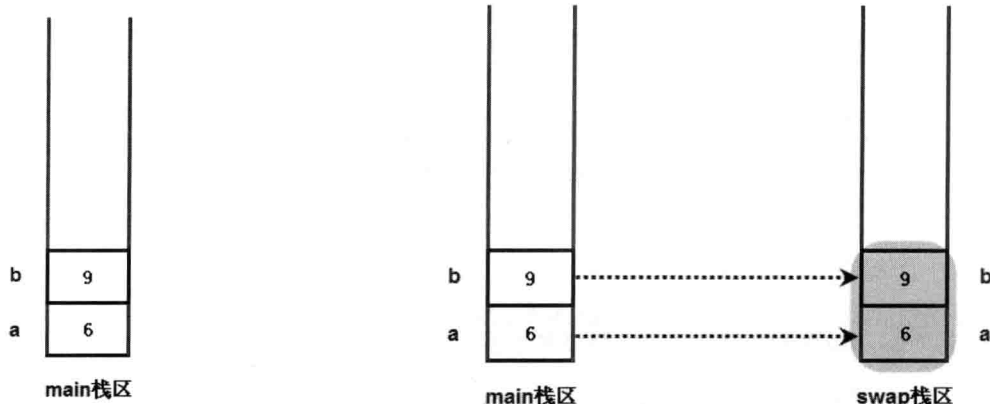


图 5.3 main()方法中定义了 a、b 变量存储示意图 图 5.4 main()方法中的变量作为参数值传入 swap()方法存储示意图

在 main()方法中调用 swap()方法时, main()方法还未结束。因此, 系统分别为 main()方法和 swap()方法分配两块栈区, 用于保存 main()方法和 swap()方法的局部变量。main()方法中的 a、b 变量作为参数值传入 swap()方法, 实际上是在 swap()方法栈区中重新产生了两个变量 a、b, 并将 main()方法栈区中 a、b 变量的值分别赋给 swap()方法栈区中的 a、b 参数 (就是对 swap()方法的 a、b 形参进行了初始化)。此时, 系统存在两个 a 变量、两个 b 变量, 只是存在于不同的方法栈区中而已。

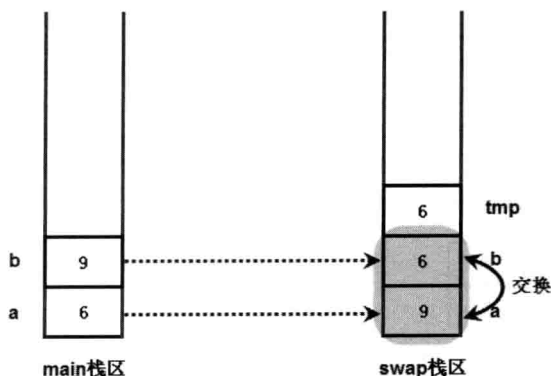


图 5.5 swap()方法中 a、b 交换之后的存储示意图

程序在 swap()方法中交换 a、b 两个变量的值, 实际上是对图 5.4 中灰色覆盖区域的 a、b 变量进行交换, 交换结束后 swap 方法中输出 a、b 变量的值, 看到 a 的值为 9, b 的值为 6, 此时内存中的存储示意图如图 5.5 所示。

对比图 5.5 与图 5.3, 两个示意图中 main()方法栈区中 a、b 的值并未有任何改变, 程序改变的只是 swap()方法栈区中的 a、b。这就是值传递的实质: 当系统开始执行方法时, 系统为形参执行初始化, 就是把实参变量的值赋给方法的形参变量, 方法里操作的并不是实际的实参变量。

前面看到的是基本类型的参数传递, Java 对于引用类型的参数传递, 一样采用的是值传递方式。但许多初学者可能对引用类型的参数传递会产生一些误会。下面程序示范了引用类型的参数传递的效果。

程序清单: codes\05\5.2\ReferenceTransferTest.java

```
class DataWrap
{
    int a;
    int b;
}
```

```
public class ReferenceTransferTest
{
    public static void swap(DataWrap dw)
    {
        // 下面三行代码实现 dw 的 a、b 两个成员变量的值交换
        // 定义一个临时变量来保存 dw 对象的 a 成员变量的值
        int tmp = dw.a;
        // 把 dw 对象的 b 成员变量的值赋给 a 成员变量
        dw.a = dw.b;
        // 把临时变量 tmp 的值赋给 dw 对象的 b 成员变量
        dw.b = tmp;
        System.out.println("swap 方法里, a 成员变量的值是"
            + dw.a + "; b 成员变量的值是" + dw.b);
    }
    public static void main(String[] args)
    {
        DataWrap dw = new DataWrap();
        dw.a = 6;
        dw.b = 9;
        swap(dw);
        System.out.println("交换结束后, a 成员变量的值是"
            + dw.a + "; b 成员变量的值是" + dw.b);
    }
}
```

执行上面程序, 看到如下运行结果:

```
swap 方法里, a 成员变量的值是 9; b 成员变量的值是 6
交换结束后, a 成员变量的值是 9; b 成员变量的值是 6
```

从上面运行结果来看, 在 `swap()` 方法里, `a`、`b` 两个成员变量的值被交换成功。不仅如此, 当 `swap()` 方法执行结束后, `main()` 方法里 `a`、`b` 两个成员变量的值也被交换了。这很容易造成一种错觉: 调用 `swap()` 方法时, 传入 `swap()` 方法的就是 `dw` 对象本身, 而不是它的复制品。但这只是一种错觉, 下面还是结合示意图来说明程序的执行过程。

程序从 `main()` 方法开始执行, `main()` 方法开始创建了一个 `DataWrap` 对象, 并定义了一个 `dw` 引用变量来指向 `DataWrap` 对象, 这是一个与基本类型不同的地方。创建一个对象时, 系统内存中有两个东西: 堆内存中保存了对象本身, 栈内存中保存了引用该对象的引用变量。接着程序通过引用来操作 `DataWrap` 对象, 把该对象的 `a`、`b` 两个成员变量分别赋值为 6、9。此时系统内存中的存储示意图如图 5.6 所示。

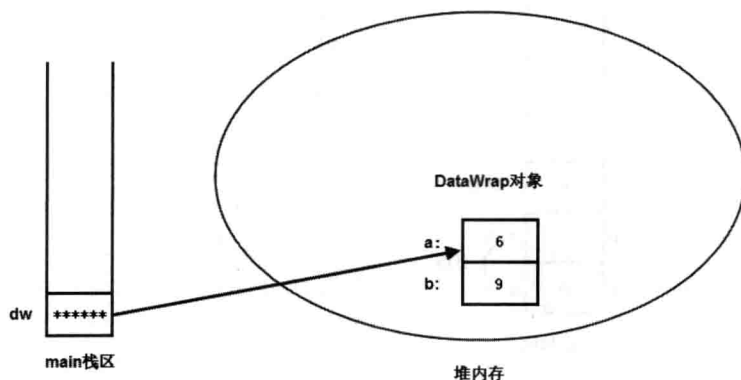


图 5.6 `main()` 方法中创建了 `DataWrap` 对象后存储示意图

接下来, `main()` 方法中开始调用 `swap()` 方法, `main()` 方法并未结束, 系统会分别为 `main()` 和 `swap()` 开辟出两个栈区, 用于存放 `main()` 和 `swap()` 方法的局部变量。调用 `swap()` 方法时, `dw` 变量作为实参传入 `swap()` 方法, 同样采用值传递方式: 把 `main()` 方法里 `dw` 变量的值赋给 `swap()` 方法里的 `dw` 形参, 从而完成 `swap()` 方法的 `dw` 形参的初始化。值得指出的是, `main()` 方法中的 `dw` 是一个引用 (也就是一个指针), 它保存了 `DataWrap` 对象的地址值, 当把 `dw` 的值赋给 `swap()` 方法的 `dw` 形参后, 即让 `swap()` 方法的 `dw` 形参也保存这个地址值, 即也会引用到堆内存中的 `DataWrap` 对象。图 5.7 显示了 `dw` 传入 `swap()`

方法后的存储示意图。

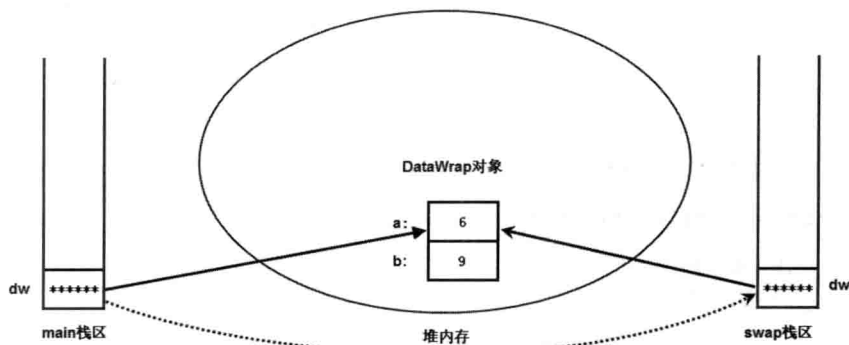


图 5.7 main()方法中的 dw 传入 swap()方法后存储示意图

从图 5.7 来看,这种参数传递方式是不折不扣的值传递方式,系统一样复制了 dw 的副本传入 swap()方法,但关键在于 dw 只是一个引用变量,所以系统复制了 dw 变量,但并未复制 DataWrap 对象。

当程序在 swap()方法中操作 dw 形参时,由于 dw 只是一个引用变量,故实际操作的是堆内存中的 DataWrap 对象。此时,不管是操作 main()方法里的 dw()变量,还是操作 swap 方法里的 dw()参数,其实都是操作它所引用的 DataWrap 对象,它们操作的是同一个对象。因此,当 swap()方法中交换 dw 参数所引用 DataWrap 对象的 a、b 两个成员变量的值后,可以看到 main()方法中 dw 变量所引用 DataWrap 对象的 a、b 两个成员变量的值也被交换了。

为了更好地证明 main()方法中的 dw 和 swap()方法中的 dw 是两个变量,在 swap()方法的最后一行增加如下代码:

```
// 把 dw 直接赋值为 null, 让它不再指向任何有效地址
dw = null;
```

执行上面代码的结果是 swap()方法中的 dw 变量不再指向任何有效内存,程序其他地方不做任何修改。main()方法调用了 swap()方法后,再次访问 dw()变量的 a、b 两个成员变量,依然可以输出 9、6。可见 main()方法中的 dw 变量没有受到任何影响。实际上,当 swap()方法中增加 dw = null;代码后,内存中的存储示意图如图 5.8 所示。

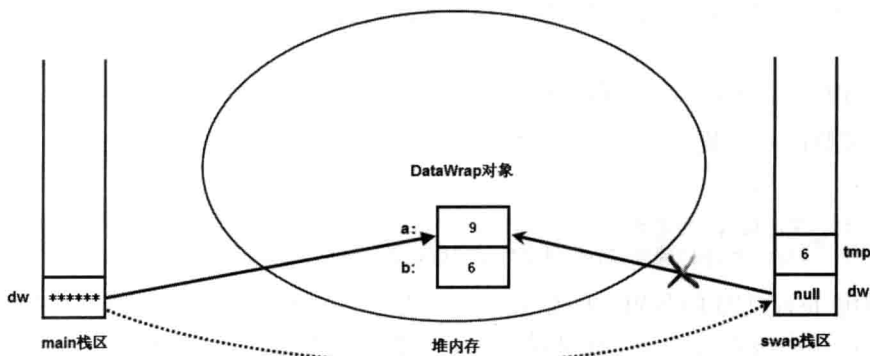


图 5.8 将 swap()方法的 dw 赋值为 null 后存储示意图

从图 5.8 来看,把 swap()方法中的 dw 赋值为 null 后,swap()方法中失去了 DataWrap 的引用,不可再访问堆内存中的 DataWrapper 对象。但 main()方法中的 dw 变量不受任何影响,依然引用 DataWrap 对象,所以依然可以输出 DataWrap 对象的 a、b 成员变量的值。

►► 5.2.3 形参个数可变的方法

从 JDK 1.5 之后,Java 允许定义形参个数可变的参数,从而允许为方法指定数量不确定的形参。如果在定义方法时,在最后一个形参的类型后增加三点 (...),则表明该形参可以接受多个参数值,多个参数值被当成数组传入。下面程序定义了一个形参个数可变的方法。

程序清单: codes\05\5.2\Varargs.java

```
public class Varargs
{
    // 定义了形参数可变的方法
    public static void test(int a , String... books)
    {
        // books 被当成数组处理
        for (String tmp : books)
        {
            System.out.println(tmp);
        }
        // 输出整数变量 a 的值
        System.out.println(a);
    }
    public static void main(String[] args)
    {
        // 调用 test 方法
        test(5 , "疯狂 Java 讲义" , "轻量级 Java EE 企业应用实战");
    }
}
```

运行上面程序, 看到如下运行结果:

```
疯狂 Java 讲义
轻量级 Java EE 企业应用实战
5
```

从上面运行结果可以看出, 当调用 `test()` 方法时, `books` 参数可以传入多个字符串作为参数值。从 `test()` 的方法体代码来看, 形参数可变的参数本质就是一个数组参数, 也就是说, 下面两个方法签名的效果完全一样。

```
// 以可变个数形参来定义方法
public static void test(int a , String... books);
```

下面采用数组形参来定义方法

```
public static void test(int a , String[] books);
```

这两种形式都包含了一个名为 `books` 的形参, 在两个方法的方法体内都可以把 `books` 当成数组处理。但区别是调用两个方法时存在差别, 对于以可变形参的形式定义的方法, 调用方法时更加简洁, 如下面代码所示。

```
test(5 , "疯狂 Java 讲义" , "轻量级 Java EE 企业应用实战");
```

传给 `books` 参数的实参数值无须是一个数组, 但如果采用数组形参来声明方法, 调用时则必须传给该形参一个数组, 如下所示。

```
// 调用 test() 方法时传入一个数组
test(23 , new String[]{"疯狂 Java 讲义" , "轻量级 Java EE 企业应用实战"});
```

对比两种调用 `test()` 方法的代码, 明显第一种形式更加简洁。实际上, 即使是采用形参数可变的形式来定义方法, 调用该方法时也一样可以为个数可变的形参传入一个数组。

最后还要指出的是, 数组形式的形参可以处于形参列表的任意位置, 但个数可变的形参只能处于形参列表的最后。也就是说, 一个方法中最多只能有一个长度可变的形参。



注意:

长度可变的形参只能处于形参列表的最后。一个方法中最多只能包含一个长度可变的形参。长度可变的形参本质就是一个数组类型的形参, 因此调用包含一个长度可变形参的方法时, 这个长度可变的形参既可以传入多个参数, 也可以传入一个数组。



5.2.4 递归方法

一个方法体内调用它自身, 被称为方法递归。方法递归包含了一种隐式的循环, 它会重复执行某段

代码,但这种重复执行无须循环控制。

例如有如下数学题。已知有一个数列: $f(0)=1$, $f(1)=4$, $f(n+2)=2*f(n+1)+f(n)$, 其中 n 是大于 0 的整数, 求 $f(10)$ 的值。这个题可以使用递归来求得。下面程序将定义一个 `fn` 方法, 用于计算 $f(10)$ 的值。

程序清单: codes\05\5.2\Recursive.java

```
public class Recursive
{
    public static int fn(int n)
    {
        if (n == 0)
        {
            return 1;
        }
        else if (n == 1)
        {
            return 4;
        }
        else
        {
            // 方法中调用它自身, 就是方法递归
            return 2 * fn(n - 1) + fn(n - 2);
        }
    }
    public static void main(String[] args)
    {
        // 输出 fn(10) 的结果
        System.out.println(fn(10));
    }
}
```

在上面的 `fn` 方法体中, 再次调用了 `fn` 方法, 这就是方法递归。注意 `fn` 方法里调用 `fn` 的形式:

```
return 2 * fn(n - 1) + fn(n - 2)
```

对于 `fn(10)`, 即等于 $2 * fn(9) + fn(8)$, 其中 `fn(9)` 又等于 $2 * fn(8) + fn(7)$ ……依此类推, 最终会计算到 `fn(2)` 等于 $2 * fn(1) + fn(0)$, 即 `fn(2)` 是可计算的, 然后一路反算回去, 就可以最终得到 `fn(10)` 的值。

仔细看上面递归的过程, 当一个方法不断地调用它本身时, 必须在某个时刻方法的返回值是确定的, 即不再调用它本身, 否则这种递归就变成了无穷递归, 类似于死循环。因此定义递归方法时有一条最重要的规定: 递归一定要向已知方向递归。

例如, 如果把上面数学题改为如此。已知有一个数列: $f(20)=1$, $f(21)=4$, $f(n+2)=2 * f(n+1) + f(n)$, 其中 n 是大于 0 的整数, 求 $f(10)$ 的值。那么 `fn` 的方法体就应该改为如下:

```
public static int fn(int n)
{
    if (n == 20)
    {
        return 1;
    }
    else if (n == 21)
    {
        return 4;
    }
    else
    {
        // 方法中调用它自身, 就是方法递归
        return fn(n + 2) - 2 * fn(n + 1);
    }
}
```

从上面的 `fn` 方法来看, 需要计算 `fn(10)` 的值时, `fn(10)` 等于 `fn(12) - 2 * fn(11)`, 而 `fn(11)` 等于 `fn(13) - 2 * fn(12)`……依此类推, 直到 `fn(19)` 等于 `fn(21) - 2 * fn(20)`, 此时就可以得到 `fn(19)` 的值了, 然后依次反算到 `fn(10)` 的值。这就是递归的重要规则: 对于求 `fn(10)` 而言, 如果 `fn(0)` 和 `fn(1)` 是已知的, 则应该采用 `fn(n) = 2 * fn(n - 1) + fn(n - 2)` 的形式递归, 因为小的一端已知; 如果 `fn(20)` 和 `fn(21)` 是已知的, 则应该

采用 $fn(n) = fn(n+2) - 2 * fn(n+1)$ 的形式递归，因为大的一端已知。

递归是非常有用的。例如希望遍历某个路径下的所有文件，但这个路径下文件夹的深度是未知的，那么就可以使用递归来实现这个需求。系统可定义一个方法，该方法接受一个文件路径作为参数，该方法可遍历当前路径下的所有文件和文件路径——该方法中再次调用该方法本身来处理该路径下的所有文件路径。

总之，只要一个方法的方法体实现中再次调用了方法本身，就是递归方法。递归一定要向已知方向递归。

5.2.5 方法重载

Java 允许同一个类里定义多个同名方法，只要形参列表不同就行。如果同一个类中包含了两个或两个以上方法的方法名相同，但形参列表不同，则被称为方法重载。

从上面介绍可以看出，在 Java 程序中确定一个方法需要三个要素。

- 调用者，也就是方法的所属者，既可以是类，也可以是对象。
- 方法名，方法的标识。
- 形参列表，当调用方法时，系统将会根据传入的实参列表匹配。

方法重载的要求就是两同一不同：同一个类中方法名相同，参数列表不同。至于方法的其他部分，如方法返回值类型、修饰符等，与方法重载没有任何关系。

下面程序中包含了方法重载的示例。

程序清单：codes\05\5.2\Overload.java

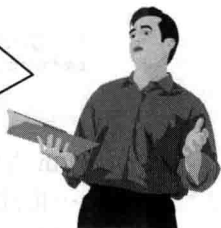
```
public class Overload
{
    // 下面定义了两个 test() 方法，但方法的形参列表不同
    // 系统可以区分这两个方法，这被称为方法重载
    public void test()
    {
        System.out.println("无参数");
    }
    public void test(String msg)
    {
        System.out.println("重载的 test 方法 " + msg);
    }
    public static void main(String[] args)
    {
        Overload ol = new Overload();
        // 调用 test() 时没有传入参数，因此系统调用上面没有参数的 test() 方法
        ol.test();
        // 调用 test() 时传入了一个字符串参数
        // 因此系统调用上面带一个字符串参数的 test() 方法
        ol.test("hello");
    }
}
```

编译、运行上面程序完全正常，虽然两个 test() 方法的方法名相同，但因为它们的形参列表不同，所以系统可以正常区分出这两个方法。



学生提问：为什么方法的返回值类型不能用于区分重载的方法？

答：对于 `int f(){} 和 void f(){} 两个方法，如果这样调用 int result = f();，系统可以识别是调用返回值类型为 int 的方法；但 Java 调用方法时可以忽略方法返回值，如果采用如下方法来调用 f()，你能判断是调用哪个方法吗？如果你尚且不能判断，那么 Java 系统也会糊涂。在编程过程中有一条重要规则：不要让系统糊涂，系统一糊涂，肯定就是你错了。因此，Java 里不能使用方法返回值类型作为区分方法重载的依据。`



不仅如此,如果被重载的方法里包含了长度可变的形参,则需要注意。看下面程序里定义的两个重载的方法。

程序清单: codes\05\5.2\OverloadVarargs.java

```
public class OverloadVarargs
{
    public void test(String msg)
    {
        System.out.println("只有一个字符串参数的 test 方法 ");
    }
    // 因为前面已经有了一个 test() 方法, test() 方法里有一个字符串参数
    // 此处的长度可变形参里不包含一个字符串参数的形式
    public void test(String... books)
    {
        System.out.println("****形参长度可变的 test 方法****");
    }
    public static void main(String[] args)
    {
        OverloadVarargs olv = new OverloadVarargs();
        // 下面两次调用将执行第二个 test() 方法
        olv.test();
        olv.test("aa" , "bb");
        // 下面调用将执行第一个 test() 方法
        olv.test("aa");
        // 下面调用将执行第二个 test() 方法
        olv.test(new String[]{"aa"});
    }
}
```

编译、运行上面程序,将看到 `olv.test()`和 `olv.test("aa", "bb")`;两次调用的是 `test(String... books)`方法,而 `olv.test("aa")`;则调用的是 `test(String msg)`方法。通过这个程序可以看出,如果同一个类中定义了 `test(String... books)`方法,同时还定义了一个 `test(String)`方法,则 `test(String... books)`方法的 `books`不可能通过直接传入一个字符串参数来调用,如果只传入一个参数,系统会执行重载的 `test(String)`方法。如果需要调用 `test(String... books)`方法,又只想传入一个字符串参数,则可采用传入字符串数组的形式,如下代码所示。

```
olv.test(new String[]{"aa"});
```

大部分时候并不推荐重载形参长度可变的方法,因为这样做确实没有太大的意义,而且容易降低程序的可读性。

5.3 成员变量和局部变量

在 Java 语言中,根据定义变量位置的不同,可以将变量分成两大类:成员变量和局部变量。成员变量和局部变量的运行机制存在较大差异,本节将详细介绍这两种变量的运行差异。

5.3.1 成员变量和局部变量

成员变量指的是在类里定义的变量,也就是前面所介绍的 `field`;局部变量指的是在方法里定义的变量。不管是成员变量还是局部变量,都应该遵守相同的命名规则:从语法角度来看,只要是一个合法的标识符即可;但从程序可读性角度来看,应该是多个有意义的单词连缀而成,其中第一个单词首字母小写,后面每个单词首字母大写。Java 程序中的变量划分如图 5.9 所示。

成员变量被分为类变量和实例变量两种,定义成员变量时没有 `static` 修饰的就是实例变量,有 `static` 修饰

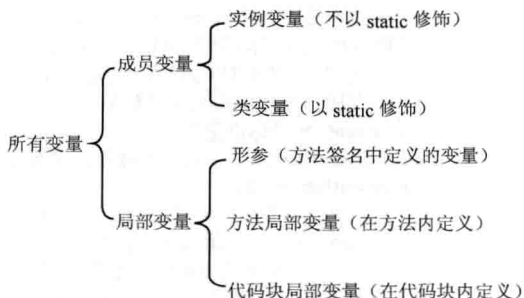


图 5.9 变量分类图

的就是类变量。其中类变量从该类的准备阶段起开始存在,直到系统完全销毁这个类,类变量的作用域与这个类的生存范围相同;而实例变量则从该类的实例被创建起开始存在,直到系统完全销毁这个实例,实例变量的作用域与对应实例的生存范围相同。



提示:
一个类在使用之前要经过类加载、类验证、类准备、类解析、类初始化等几个阶段,关于类的生命周期的介绍,读者可以参考本书第 18 章。
.....

正是基于这个原因,可以把类变量和实例变量统称为成员变量,其中类变量可以理解为类成员变量,它作为类本身的一个成员,与类本身共存亡;实例变量则可理解为实例成员变量,它作为实例的一个成员,与实例共存亡。

只要类存在,程序就可以访问该类的类变量。在程序中访问类变量通过如下语法:

类.类变量

只要实例存在,程序就可以访问该实例的实例变量。在程序中访问实例变量通过如下语法:

实例.实例变量

当然,类变量也可以让该类的实例来访问。通过实例来访问类变量的语法如下:

实例.类变量

但由于这个实例并不拥有这个类变量,因此它访问的并不是这个实例的变量,依然是访问它对应类的类变量。也就是说,如果通过一个实例修改了类变量的值,由于这个类变量并不属于它,而是属于它对应的类。因此,修改的依然是类的类变量,与通过该类来修改类变量的结果完全相同,这会导致该类的其他实例来访问这个类变量时也将获得这个被修改过的值。

下面程序定义了一个 **Person** 类,在这个 **Person** 类中定义两个成员变量,一个实例变量: **name**, 以及一个类变量: **eyeNum**。程序还通过 **PersonTest** 类来创建 **Person** 实例,并分别通过 **Person** 类和 **Person** 实例来访问实例变量和类变量。

程序清单: codes\05\5.3\PersonTest.java

```
class Person
{
    // 定义一个实例变量
    public String name;
    // 定义一个类变量
    public static int eyeNum;
}
public class PersonTest
{
    public static void main(String[] args)
    {
        // 第一次主动使用 Person 类,该类自动初始化,则 eyeNum 变量开始起作用,输出 0
        System.out.println("Person 的 eyeNum 类变量值:"
            + Person.eyeNum);
        // 创建 Person 对象
        Person p = new Person();
        // 通过 Person 对象的引用 p 来访问 Person 对象 name 实例变量
        // 并通过实例访问 eyeNum 类变量
        System.out.println("p 变量的 name 变量值是: " + p.name
            + " p 对象的 eyeNum 变量值是: " + p.eyeNum);
        // 直接为 name 实例变量赋值
        p.name = "孙悟空";
        // 通过 p 访问 eyeNum 类变量,依然是访问 Person 的 eyeNum 类变量
        p.eyeNum = 2;
        // 再次通过 Person 对象来访问 name 实例变量和 eyeNum 类变量
        System.out.println("p 变量的 name 变量值是: " + p.name
            + " p 对象的 eyeNum 变量值是: " + p.eyeNum);
        // 前面通过 p 修改了 Person 的 eyeNum,此处的 Person.eyeNum 将输出 2
        System.out.println("Person 的 eyeNum 类变量值:" + Person.eyeNum);
        Person p2 = new Person();
    }
}
```

```
// p2 访问的 eyeNum 类变量依然引用 Person 类的, 因此依然输出 2
System.out.println("p2 对象的 eyeNum 类变量值:" + p2.eyeNum);
}
}
```

从上面程序来看, 成员变量无须显式初始化, 只要为一个类定义了类变量或实例变量, 系统就会在这个类的准备阶段或创建该类的实例时进行默认初始化, 成员变量默认初始化时的赋值规则与数组动态初始化时数组元素的赋值规则完全相同。

从上面程序运行结果来看, 不难发现类变量的作用域比实例变量的作用域更大: 实例变量随实例的存在而存在, 而类变量则随类的存在而存在。实例也可访问类变量, 同一个类的所有实例访问类变量时, 实际上访问的是该类本身的同一个变量, 也就是说, 访问了同一片内存区。



提示:

正如前面提到的, Java 允许通过实例来访问 `static` 修饰的成员变量本身就是一个错误, 因此读者以后看到通过实例来访问 `static` 成员变量的情形, 都可以将它替换成通过类本身来访问 `static` 成员变量的情形, 这样程序的可读性、明确性都会大大提高。

局部变量根据定义形式的不同, 又可以被分为如下三种。

- 形参: 在定义方法签名时定义的变量, 形参的作用域在整个方法内有效。
- 方法局部变量: 在方法体内定义的局部变量, 它的作用域是从定义该变量的地方生效, 到该方法结束时失效。
- 代码块局部变量: 在代码块中定义的局部变量, 这个局部变量的作用域从定义该变量的地方生效, 到该代码块结束时失效。

与成员变量不同的是, 局部变量除了形参之外, 都必须显式初始化。也就是说, 必须先给方法局部变量和代码块局部变量指定初始值, 否则不可以访问它们。

下面代码是定义代码块局部变量的实例程序。

程序清单: codes\05\5.3\BlockTest.java

```
public class BlockTest
{
    public static void main(String[] args)
    {
        {
            // 定义一个代码块局部变量 a
            int a;
            // 下面代码将出现错误, 因为 a 变量还未初始化
            // System.out.println("代码块局部变量 a 的值: " + a);
            // 为 a 变量赋初始值, 也就是进行初始化
            a = 5;
            System.out.println("代码块局部变量 a 的值: " + a);
        }
        // 下面试图访问的 a 变量并不存在
        // System.out.println(a);
    }
}
```

从上面代码中可以看出, 只要离开了代码块局部变量所在的代码块, 这个局部变量就立即被销毁, 变为不可见。

对于方法局部变量, 其作用域从定义该变量开始, 直到该方法结束。下面代码示范了方法局部变量的作用域。

程序清单: codes\05\5.3\MethodLocalVariableTest.java

```
public class MethodLocalVariableTest
{
    public static void main(String[] args)
    {
        // 定义一个方法局部变量 a
```

```
int a;
// 下面代码将出现错误, 因为 a 变量还未初始化
// System.out.println("方法局部变量 a 的值: " + a);
// 为 a 变量赋初始值, 也就是进行初始化
a = 5;
System.out.println("方法局部变量 a 的值: " + a);
}
}
```

形参的作用域是整个方法体内有效, 而且形参也无须显式初始化, 形参的初始化在调用该方法时由系统完成, 形参的值由方法的调用者负责指定。

当通过类或对象调用某个方法时, 系统会在该方法栈区内为所有的形参分配内存空间, 并将实参的值赋给对应的形参, 这就完成了形参的初始化。关于形参的传递机制请参阅 5.2.2 节的介绍。

在同一个类里, 成员变量的作用范围是整个类内有效, 一个类里不能定义两个同名的成员变量, 即使一个是类变量, 一个是实例变量也不行; 一个方法里不能定义两个同名的方法局部变量, 方法局部变量与形参也不能同名; 同一个方法中不同代码块内的代码块局部变量可以同名; 如果先定义代码块局部变量, 后定义方法局部变量, 前面定义的代码块局部变量与后面定义的方法局部变量也可以同名。

Java 允许局部变量和成员变量同名, 如果方法里的局部变量和成员变量同名, 局部变量会覆盖成员变量, 如果需要在这个方法里引用被覆盖的成员变量, 则可使用 `this` (对于实例变量) 或类名 (对于类变量) 作为调用者来限定访问成员变量。

程序清单: codes\05\5.3\VariableOverrideTest.java

```
public class VariableOverrideTest
{
    // 定义一个 name 实例变量
    private String name = "李刚";
    // 定义一个 price 类变量
    private static double price = 78.0;
    // 主方法, 程序的入口
    public static void main(String[] args)
    {
        // 方法里的局部变量, 局部变量覆盖成员变量
        int price = 65;
        // 直接访问 price 变量, 将输出 price 局部变量的值: 65
        System.out.println(price);
        // 使用类名作为 price 变量的限定,
        // 将输出 price 类变量的值: 78.0
        System.out.println(VariableOverrideTest.price);
        // 运行 info 方法
        new VariableOverrideTest().info();
    }
    public void info()
    {
        // 方法里的局部变量, 局部变量覆盖成员变量
        String name = "孙悟空";
        // 直接访问 name 变量, 将输出 name 局部变量的值: "孙悟空"
        System.out.println(name);
        // 使用 this 来作为 name 变量的限定
        // 将输出 name 实例变量的值: "李刚"
        System.out.println(this.name);
    }
}
```

从上面代码可以清楚地看出局部变量覆盖成员变量时, 不过依然可以在方法中显式指定类名和 `this` 作为调用者来访问被覆盖的成员变量, 这使得编程更加自由。不过大部分时候还是应该尽量避免这种局部变量和成员变量同名的情形。

5.3.2 成员变量的初始化和内存中的运行机制

当系统加载类或创建该类的实例时, 系统自动为成员变量分配内存空间, 并在分配内存空间后, 自

动为成员变量指定初始值。

下面以 codes\05\5.3\PersonTest.java 代码中定义的 Person 类来创建两个实例,配合示意图来说明 Java 的成员变量的初始化和内存中的运行机制。看下面几行代码:

```
// 创建第一个 Person 对象
Person p1 = new Person();
// 创建第二个 Person 对象
Person p2 = new Person();
// 分别为两个 Person 对象的 name 实例变量赋值
p1.name = "张三";
p2.name = "孙悟空";
// 分别为两个 Person 对象的 eyeNum 类变量赋值
p1.eyeNum = 2;
p2.eyeNum = 3;
```

当程序执行第一行代码 `Person p1 = new Person();` 时,如果这行代码是第一次使用 Person 类,则系统通常会在第一次使用 Person 类时加载这个类,并初始化这个类。在类的准备阶段,系统将会为该类的类变量分配内存空间,并指定默认初始值。当 Person 类初始化完成后,系统内存中的存储示意图如图 5.10 所示。

从图 5.10 中可以看出,当 Person 类初始化完成后,系统将在堆内存中为 Person 类分配一块内存区(当 Person 类初始化完成后,系统会为 Person 类创建一个类对象,具体参考本书第 18 章),在这块内存区里包含了保存 eyeNum 类变量的内存,并设置 eyeNum 的默认初始值: 0。

系统接着创建了一个 Person 对象,并把这个人对象赋给 p1 变量, Person 对象里包含了名为 name 的实例变量,实例变量是在创建实例时分配内存空间并指定初始值的。当创建了第一个 Person 对象后,系统内存中的存储示意图如图 5.11 所示。

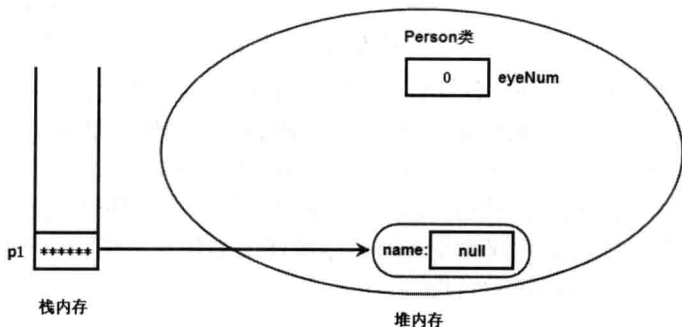
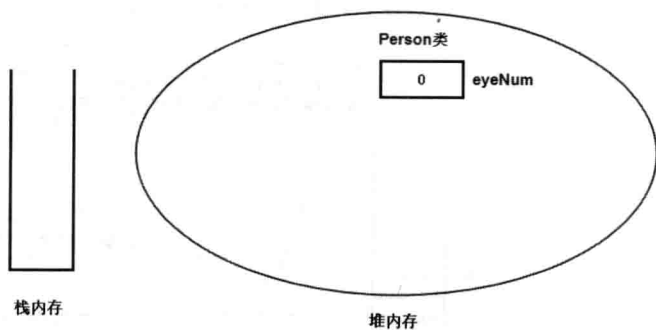
从图 5.11 中可以看出, eyeNum 类变量并不属于 Person 对象,它是属于 Person 类的,所以创建第一个 Person 对象时并不需要为 eyeNum 类变量分配内存,系统只是为 name 实例变量分配了内存空间,并指定默认初始值: null。

接着执行 `Person p2 = new Person();` 代码创建第二个 Person 对象,此时因为 Person 类已经存在于堆内存中了,所以不再需要对 Person 类进行初始化。创建第二个 Person 对象与创建第一个 Person 对象并没有什么不同。

当程序执行 `p1.name = "张三";` 代码时,将为 p1 的 name 实例变量赋值,也就是让图 5.11 中堆内存中的 name 指向"张三"字符串。执行完成后,两个 Person 对象在内存中的存储示意图如图 5.12 所示。

从图 5.12 中可以看出, name 实例变量是属于单个 Person 实例的,因此修改第一个 Person 对象的 name 实例变量时仅仅与该对象有关,与 Person 类和其他 Person 对象没有任何关系。同样,修改第二个 Person 对象的 name 实例变量时,也与 Person 类和其他 Person 对象无关。

直到执行 `p1.eyeNum = 2;` 代码时,此时通过 Person 对象来修改 Person 的类变量,从图 5.12 中不难看出, Person 对象根本没有保存 eyeNum 这个变量,通过 p1 访问的 eyeNum 类变量,其实还是 Person



类的 eyeNum 类变量。因此，此时修改的是 Person 类的 eyeNum 类变量。修改成功后，内存中的存储示意图如图 5.13 所示。

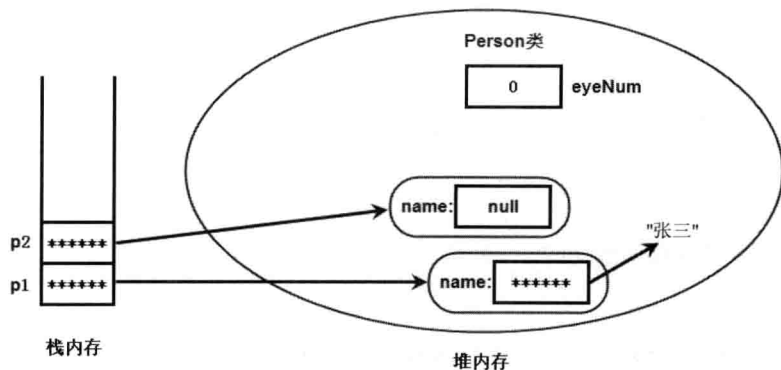


图 5.12 为第一个 Person 对象的 name 实例变量赋值后的存储示意图

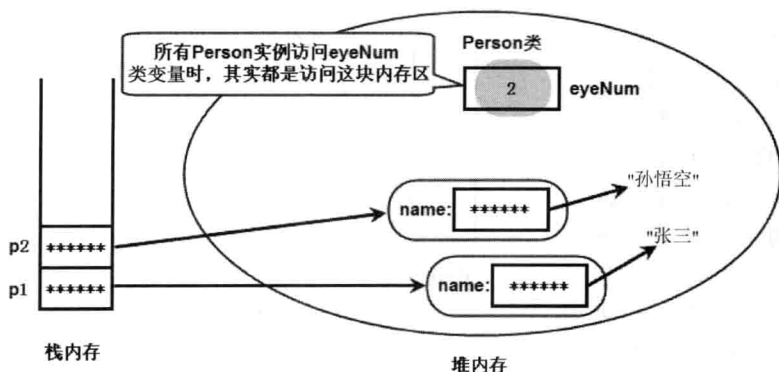


图 5.13 设置 p1 的 eyeNum 类变量之后的存储示意图

从图 5.13 中可以看出，当通过 p1 来访问类变量时，实际上访问的是 Person 类的 eyeNum 类变量。事实上，所有的 Person 实例访问 eyeNum 类变量时都将访问到 Person 类的 eyeNum 类变量，也就是图 5.13 中灰色覆盖的区域。换句话说，不管通过哪个 Person 实例来访问 eyeNum 类变量，本质其实还是通过 Person 类来访问 eyeNum 类变量时，它们所访问的是同一块内存。基于这个理由，本书建议读者，当程序需要访问类变量时，尽量使用类作为主调，而不要使用对象作为主调，这样可以避免程序产生歧义，提高程序的可读性。



注意

遗憾的是，经常见到有些公司的招聘笔试题，或者有些某某试题（比如 SCJP 等），其中常常就有通过不同对象来访问类变量的情形。Java 语法中允许通过对象来访问类成员（包括类变量、方法）可以说完全是一个缺陷，聪明的开发者应该学会避开这个陷阱，而不是天天在这个陷阱旁边绕来绕去！



5.3.3 局部变量的初始化和内存中的运行机制

局部变量定义后，必须经过显式初始化后才能使用，系统不会为局部变量执行初始化。这意味着定义局部变量后，系统并未为这个变量分配内存空间，直到等到程序为这个变量赋初始值时，系统才会为局部变量分配内存，并将初始值保存在这块内存中。

与成员变量不同，局部变量不属于任何类或实例，因此它总是保存在其所在方法的栈内存中。如果局部变量是基本类型的变量，则直接把这个变量的值保存在该变量对应的内存中；如果局部变量是一个引用类型的变量，则这个变量里存放的是地址，通过该地址引用到该变量实际引用的对象或数组。

栈内存中的变量无须系统垃圾回收，往往随方法或代码块的运行结束而结束。因此，局部变量的作

用域是从初始化该变量开始,直到该方法或该代码块运行完成而结束。因为局部变量只保存基本类型的值或者对象的引用,因此局部变量所占的内存区通常比较小。

5.3.4 变量的使用规则

对Java初学者而言,何时应该使用类变量?何时应该使用实例变量?何时应该使用方法局部变量?何时应该使用代码块局部变量?这种选择比较困难,如果仅就程序的运行结果来看,大部分时候都可以直接使用类变量或者实例变量来解决问题,无须使用局部变量。但实际上这种做法相当错误,因为定义一个成员变量时,成员变量将被放置到堆内存中,成员变量的作用域将扩大到类存在范围或者对象存在范围,这种范围的扩大有两个害处。

- 增大了变量的生存时间,这将导致更大的内存开销。
- 扩大了变量的作用域,这不利于提高程序的内聚性。

对比下面三个程序。

程序清单: codes\05\5.3\ScopeTest1.java

```
public class ScopeTest1
{
    // 定义一个类成员变量作为循环变量
    static int i;
    public static void main(String[] args)
    {
        for ( i = 0 ; i < 10 ; i++)
        {
            System.out.println("Hello");
        }
    }
}
```

程序清单: codes\05\5.3\ScopeTest2.java

```
public class ScopeTest2
{
    public static void main(String[] args)
    {
        // 定义一个方法局部变量作为循环变量
        int i;
        for ( i = 0 ; i < 10 ; i++)
        {
            System.out.println("Hello");
        }
    }
}
```

程序清单: codes\05\5.3\ScopeTest3.java

```
public class ScopeTest3
{
    public static void main(String[] args)
    {
        // 定义一个代码块局部变量作为循环变量
        for (int i = 0 ; i < 10 ; i++)
        {
            System.out.println("Hello");
        }
    }
}
```

这三个程序的运行结果完全相同,但程序的效果则大有差异。第三个程序最符合软件开发规范:对于一个循环变量而言,只需要它在循环体内有效,因此只需要把这个变量放在循环体内(也就是在代码块内定义),从而保证这个变量的作用域仅在该代码块内。

如果有如下几种情形,则应该考虑使用成员变量。

- 如果需要定义的变量是用于描述某个类或某个对象的固有信息的,例如人的身高、体重等信息,它们是人对象的固有信息,每个人对象都具有这些信息。这种变量应该定义为成员变量。如果

这种信息对这个类的所有实例完全相同，或者说它是类相关的，例如人类的眼睛数量，目前所有人的眼睛数量都是 2，如果人类进化了，变成了 3 个眼睛，则所有人的眼睛数量都是 3，这种类相关的信息应该定义成类变量；如果这种信息是实例相关的，例如人的身高、体重等，每个人实例的身高、体重可能互不相同，这种信息是实例相关的，因此应该定义成实例变量。

- 如果在某个类中需要以一个变量来保存该类或者实例运行时的状态信息，例如上面五子棋程序中的棋盘数组，它用以保存五子棋实例运行时的状态信息。这种用于保存某个类或某个实例状态信息的变量通常应该使用成员变量。
- 如果某个信息需要在某个类的多个方法之间进行共享，则这个信息应该使用成员变量来保存。例如，在把浮点数转换为人民币读法字符串的程序中，数字的大写字符和单位字符等是多个方法的共享信息，因此应设置为成员变量。

即使在程序中使用局部变量，也应该尽可能地缩小局部变量的作用范围，局部变量的作用范围越小，它在内存里停留的时间就越短，程序运行性能就越好。因此，能用代码块局部变量的地方，就坚决不要使用方法局部变量。

5.4 隐藏和封装

在前面程序中经常出现通过某个对象的直接访问其成员变量的情形，这可能引起一些潜在的问题，比如将某个 `Person` 的 `age` 成员变量直接设为 1000，这在语法上没有任何问题，但显然违背了现实。因此，Java 程序推荐将类和对象的成员变量进行封装。

5.4.1 理解封装

封装 (Encapsulation) 是面向对象的三大特征之一 (另外两个是继承和多态)，它指的是将对象的状态信息隐藏在对象内部，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问。

封装是面向对象编程语言对客观世界的模拟，在客观世界里，对象的状态信息都被隐藏在对象内部，外界无法直接操作和修改。就如刚刚说的 `Person` 对象的 `age` 变量，只能随着岁月的流逝，`age` 才会增加，通常不能随意修改 `Person` 对象的 `age`。对一个类或对象实现良好的封装，可以实现以下目的。

- 隐藏类的实现细节。
- 让使用者只能通过事先预定的方法来访问数据，从而可以在该方法里加入控制逻辑，限制对成员变量的不合理访问。
- 可进行数据检查，从而有利于保证对象信息的完整性。
- 便于修改，提高代码的可维护性。

为了实现良好的封装，需要从两个方面考虑。

- 将对象的成员变量和实现细节隐藏起来，不允许外部直接访问。
- 把方法暴露出来，让方法来控制对这些成员变量进行安全的访问和操作。

因此，封装实际上有两个方面的含义：把该隐藏的隐藏起来，把该暴露的暴露出来。这两个方面都需要通过使用 Java 提供的访问控制符来实现。

5.4.2 使用访问控制符

Java 提供了 3 个访问控制符：`private`、`protected` 和 `public`，分别代表了 3 个访问控制级别，另外还有一个不加任何访问控制符的访问控制级别，提供了 4 个访问控制级别。Java 的访问控制级别由小到大如图 5.14 所示。

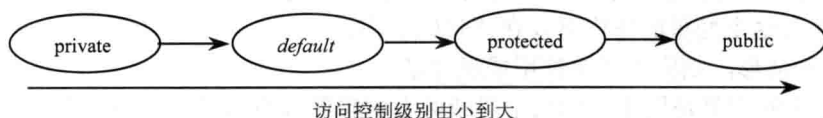


图 5.14 访问控制级别图

图 5.14 中的 4 个访问控制级别中的 **default** 并没有对应的访问控制符, 当不使用任何访问控制符来修饰类或类成员时, 系统默认使用该访问控制级别。这 4 个访问控制级别的详细介绍如下。

- **private** (当前类访问权限): 如果类里的一个成员 (包括成员变量、方法和构造器等) 使用 **private** 访问控制符来修饰, 则这个成员只能在当前类的内部被访问。很显然, 这个访问控制符用于修饰成员变量最合适, 使用它来修饰成员变量就可以把成员变量隐藏在该类的内部。
- **default** (包访问权限): 如果类里的一个成员 (包括成员变量、方法和构造器等) 或者一个外部类不使用任何访问控制符修饰, 就称它是包访问权限的, **default** 访问控制的成员或外部类可以被相同包下的其他类访问。关于包的介绍请看 5.4.3 节。
- **protected** (子类访问权限): 如果一个成员 (包括成员变量、方法和构造器等) 使用 **protected** 访问控制符修饰, 那么这个成员既可以被同一个包中的其他类访问, 也可以被不同包中的子类访问。在通常情况下, 如果使用 **protected** 来修饰一个方法, 通常是希望其子类来重写这个方法。关于父类、子类的介绍请参考 5.6 节的内容。
- **public** (公共访问权限): 这是一个最宽松的访问控制级别, 如果一个成员 (包括成员变量、方法和构造器等) 或者一个外部类使用 **public** 访问控制符修饰, 那么这个成员或外部类就可以被所有类访问, 不管访问类和被访问类是否处于同一个包中, 是否具有父子继承关系。

最后使用表 5.1 来总结上述的访问控制级别。

表 5.1 访问控制级别表

	private	default	protected	public
同一个类中	✓	✓	✓	✓
同一个包中		✓	✓	✓
子类中			✓	✓
全局范围内				✓

通过上面关于访问控制符的介绍不难发现, 访问控制符用于控制一个类的成员是否可以被其他类访问, 对于局部变量而言, 其作用域就是它所在的方法, 不可能被其他类访问, 因此不能使用访问控制符来修饰。

对于外部类而言, 它也可以使用访问控制符修饰, 但外部类只能有两种访问控制级别: **public** 和默认, 外部类不能使用 **private** 和 **protected** 修饰, 因为外部类没有处于任何类的内部, 也就没有其所在类的内部、所在类的子类两个范围, 因此 **private** 和 **protected** 访问控制符对外部类没有意义。

外部类可以使用 **public** 和包访问控制权限, 使用 **public** 修饰的外部类可以被所有类使用, 如声明变量、创建实例; 不使用任何访问控制符修饰的外部类只能被同一个包中的其他类使用。



提示:

如果一个 Java 源文件里定义的所有类都没有使用 **public** 修饰, 则这个 Java 源文件的文件名可以是一切合法的文件名; 但如果一个 Java 源文件里定义了一个 **public** 修饰的类, 则这个源文件的文件名必须与 **public** 修饰的类的类名相同。

掌握了访问控制符的用法之后, 下面通过使用合理的访问控制符来定义一个 **Person** 类, 这个 **Person** 类实现了良好的封装。

程序清单: codes\05\5.4\Person.java

```
public class Person
{
    // 使用 private 修饰成员变量, 将这些成员变量隐藏起来
    private String name;
    private int age;
    // 提供方法来操作 name 成员变量
    public void setName(String name)
    {
        // 执行合理性校验, 要求用户名必须在 2~6 位之间
    }
}
```



```

        if (name.length() > 6 || name.length() < 2)
        {
            System.out.println("您设置的人名不符合要求");
            return;
        }
        else
        {
            this.name = name;
        }
    }
    public String getName()
    {
        return this.name;
    }
    // 提供方法来操作 age 成员变量
    public void setAge(int age)
    {
        // 执行合理性校验, 要求用户年龄必须在 0~100 之间
        if (age > 100 || age < 0)
        {
            System.out.println("您设置的年龄不合法");
            return;
        }
        else
        {
            this.age = age;
        }
    }
    public int getAge()
    {
        return this.age;
    }
}

```

定义了上面的 Person 类之后, 该类的 name 和 age 两个成员变量只有在 Person 类内才可以操作和访问, 在 Person 类之外只能通过各自对应的 setter 和 getter 方法来操作和访问它们。



提示:

Java 类里实例变量的 setter 和 getter 方法有非常重要的意义。例如, 某个类里包含了一个名为 abc 的实例变量, 则其对应的 setter 和 getter 方法名应为 setAbc() 和 getAbc() (即将原实例变量名的首字母大写, 并在前面分别增加 set 和 get 动词, 就变成 setter 和 getter 方法名)。如果一个 Java 类的每个实例变量都被使用 private 修饰, 并为每个实例变量都提供了 public 修饰 setter 和 getter 方法, 那么这个类就是一个符合 JavaBean 规范的类。因此, JavaBean 总是一个封装良好的类。

下面程序在 main() 方法中创建一个 Person 对象, 并尝试操作和访问该对象的 age 和 name 两个实例变量。

程序清单: codes\05\5.4\PersonTest.java

```

public class PersonTest
{
    public static void main(String[] args)
    {
        Person p = new Person();
        // 因为 age 成员变量已被隐藏, 所以下面语句将出现编译错误
        // p.age = 1000;
        // 下面语句编译不会出现错误, 但运行时将提示"您设置的年龄不合法"
        // 程序不会修改 p 的 age 成员变量
        p.setAge(1000);
        // 访问 p 的 age 成员变量也必须通过其对应的 getter 方法
        // 因为上面从未成功设置 p 的 age 成员变量, 故此处输出 0
        System.out.println("未能设置 age 成员变量时: "
            + p.getAge());
        // 成功修改 p 的 age 成员变量
    }
}

```

```
p.setAge(30);
// 因为上面成功设置了 p 的 age 成员变量, 故此处输出 30
System.out.println("成功设置 age 成员变量后: "
    + p.getAge());
// 不能直接操作 p 的 name 成员变量, 只能通过其对应的 setter 方法
// 因为 "李刚" 字符串长度满足 2~6, 所以可以成功设置
p.setName("李刚");
System.out.println("成功设置 name 成员变量后: "
    + p.getName());
}
```

正如上面程序中注释的, `PersonTest` 类的 `main()` 方法不可再直接修改 `Person` 对象的 `name` 和 `age` 两个实例变量, 只能通过各自对应的 `setter` 方法来操作这两个实例变量的值。因为使用 `setter` 方法来操作 `name` 和 `age` 两个实例变量, 就允许程序员在 `setter` 方法中增加自己的控制逻辑, 从而保证 `Person` 对象的 `name` 和 `age` 两个实例变量不会出现与实际不符的情形。



提示：

一个类常常就是一个小的模块, 应该只让这个模块公开必须让外界知道的内容, 而隐藏其他一切内容。进行程序设计时, 应尽量避免一个模块直接操作和访问另一个模块的数据, 模块设计追求高内聚 (尽可能把模块的内部数据、功能实现细节隐藏在模块内部独立完成, 不允许外部直接干预)、低耦合 (仅暴露少量的方法给外部使用)。正如日常常见的内存条, 内存条里的数据及其实现细节被完全隐藏在内存条里面, 外部设备 (如主机板) 只能通过内存条的金手指 (提供一些方法供外部调用) 来和内存条进行交互。

关于访问控制符的使用, 存在如下几条基本原则。

- 类里的绝大部分成员变量都应该使用 `private` 修饰, 只有一些 `static` 修饰的、类似全局变量的成员变量, 才可能考虑使用 `public` 修饰。除此之外, 有些方法只用于辅助实现该类的其他方法, 这些方法被称为工具方法, 工具方法也应该使用 `private` 修饰。
- 如果某个类主要用做其他类的父类, 该类里包含的大部分方法可能仅希望被其子类重写, 而不想被外界直接调用, 则应该使用 `protected` 修饰这些方法。
- 希望暴露出来给其他类自由调用的方法应该使用 `public` 修饰。因此, 类的构造器通过使用 `public` 修饰, 从而允许在其他地方创建该类的实例。因为外部类通常都希望被其他类自由使用, 所以大部分外部类都使用 `public` 修饰。



注意：

本书在写作过程中, 有些类并没有提供良好的封装, 这只是为了更好地演示某个知识点, 或为了突出某些用法, 读者不必模仿这种不好的做法。



5.4.3 package、import 和 import static

前面提到了包范围这个概念, 那么什么是包呢? 关于这个问题, 先来回忆一个场景: 在我们漫长的求学、工作生涯中可曾遇到过与自己同名的同学或同事? 因为笔者姓名的缘故, 笔者经常会遭遇此类事情。如果同一个班级里出现两个叫“李刚”的同学, 那老师怎么处理呢? 老师通常会在我们的名字前增加一个限定, 例如大李刚、小李刚以示区分。

类似地, Oracle 公司的 JDK、各种系统软件厂商、众多的软件开发商, 他们会提供成千上万、具有各种用途的类, 不同软件公司在开发过程中也要提供大量的类, 这些类会不会发生同名的情形呢? 答案是肯定的。那么如何处理这种重名问题呢? Oracle 也允许在类名前增加一个前缀来限定这个类。Java 引入了包 (package) 机制, 提供了类的多层命名空间, 用于解决类的命名冲突、类文件管理等问题。

Java 允许将一组功能相关的类放在同一个 package 下, 从而组成逻辑上的类库单元。如果希望把一个类放在指定的包结构下, 应该在 Java 源程序的第一个非注释行放置如下格式的代码:

```
package packageName;
```

一旦在 Java 源文件中使用了这个 `package` 语句,就意味着该源文件里定义的所有类都属于这个包。位于包中的每个类的完整类名都应该是包名和类名的组合,如果其他人需要使用该包下的类,也应该使用包名加类名的组合。

下面程序在 `lee` 包下定义了一个简单的 Java 类。

程序清单: `codes\05\5.4\Hello.java`

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

上面程序中粗体字代码行表明把 `Hello` 类放在 `lee` 包空间下。把上面源文件保存在任意位置,使用如下命令来编译这个 Java 文件:

```
javac -d . Hello.java
```

前面已经介绍过, `-d` 选项用于设置编译生成 `class` 文件的保存位置,这里指定将生成的 `class` 文件放在当前路径(就代表当前路径)下。使用该命令编译该文件后,发现当前路径下并没有 `Hello.class` 文件,而是在当前路径下多了一个名为 `lee` 的文件夹,该文件夹下则有一个 `Hello.class` 文件。

这是怎么回事呢?这与 Java 的设计有关。假设某个应用中包含两个 `Hello` 类,Java 通过引入包机制来区分两个不同的 `Hello` 类。不仅如此,这两个 `Hello` 类还对应两个 `Hello.class` 文件,它们在文件系统中也必须分开存放才不会引起冲突。所以 Java 规定:位于包中的类,在文件系统中也必须有与包名层次相同的目录结构。

对于上面的 `Hello.class`,它必须放在 `lee` 文件夹下才是有效的,当使用带 `-d` 选项的 `javac` 命令来编译 Java 源文件时,该命令会自动建立对应的文件结构来存放相应的 `class` 文件。

如果直接使用 `javac Hello.java` 命令来编译这个文件,将会在当前路径下生成一个 `Hello.class` 文件,而不会生成 `lee` 文件夹。也就是说,如果编译 Java 文件时不使用 `-d` 选项,编译器不会为 Java 源文件生成相应的文件结构。鉴于此,本书推荐编译 Java 文件时总是使用 `-d` 选项,即使想把生成的 `class` 文件放在当前路径下,也应使用 `-d` 选项,而不省略 `-d` 选项。

进入编译器生成的 `lee` 文件夹所在路径,执行如下命令:

```
java lee.Hello
```

看到上面程序正常输出。

如果进入 `lee` 路径下使用 `java Hello` 命令来运行 `Hello` 类,系统将提示错误。正如前面讲的, `Hello` 类处于 `lee` 包下,因此必须把 `Hello.class` 文件放在 `lee` 路径下。

当虚拟机要装载 `lee.Hello` 类时,它会依次搜索 `CLASSPATH` 环境变量所指定的系列路径,查找这些路径下是否包含 `lee` 路径,并在 `lee` 路径下查找是否包含 `Hello.class` 文件。虚拟机在装载带包名的类时,会先搜索 `CLASSPATH` 环境变量指定的目录,然后在这些目录中按与包层次对应的目录结构去查找 `class` 文件。

同一个包中的类不必位于相同的目录下,例如有 `lee.Person` 和 `lee.PersonTest` 两个类,它们完全可以一个位于 C 盘下某个位置,一个位于 D 盘下某个位置,只要让 `CLASSPATH` 环境变量里包含这两个路径即可。虚拟机会自动搜索 `CLASSPATH` 下的子路径,把它们当成同一个包下的类来处理。

不仅如此,也应该把 Java 源文件放在与包名一致的目录结构下。与前面介绍的理由相似,如果系统中存在两个 `Hello` 类,通常也对应两个 `Hello.java` 源文件,如果把它们的源文件也放在对应的文件结构下,就可以解决源文件在文件系统中的存储冲突。

例如,可以把上面的 `Hello.java` 文件也放在与包层次相同的文件夹下面,即放在 `lee` 路径下。如果将源文件和 `class` 文件统一存放,也可能造成混乱,通常建议将源文件和 `class` 文件也分开存放,以便管理。例如,上面定义的位于 `lee` 包下的 `Hello.java` 及其生成的 `Hello.class` 文件,建议以图 5.15 所示的形

式来存放。

注意：

很多初学者以为只要把生成的 class 文件放在某个目录下，这个目录名就成了这个类的包名。这是一个错误的看法，不是有了目录结构，就等于有了包名。为 Java 类添加包必须在 Java 源文件中通过 package 语句指定，单靠目录名是没法指定的。Java 的包机制需要两个方面保证：① 源文件里使用 package 语句指定包名；② class 文件必须放在对应的路径下。

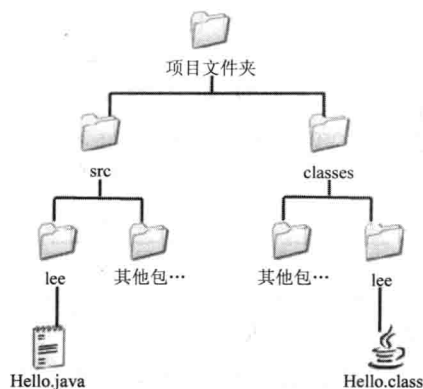


图 5.15 项目里源文件和 class 文件的组织

Java 语法只要求包名是有效的标识符即可，但从可读性规范角度来看，包名应该全部是小写字母，而且应该由一个或多个有意义的单词连缀而成。

当系统越来越大时，是否会发生包名、类名同时重复的情形呢？这个可能性不大，但在实际开发中，还是应该选择合适的包名，用以更好地组织系统中类库。为了避免不同公司之间类名的重复，Oracle 建议使用公司 Internet 域名倒写来作为包名，例如公司的 Internet 域名是 crazyit.org，则该公司的所有类都建议放在 org.crazyit 包及其子包下。



提示：

在实际企业开发中，还会在 org.crazyit 包下以项目名建立子包；如果该项目足够大，则还会在项目名子包下以模块名来建立模块子包；如果该模块下还包括多种类型的组件，则还会建立对应的子包。假设有一个 eLearning 系统，对于该系统下学生模块的 DAO 组件，则通常会放在 org.crazyit.elearning.student.dao 包下，其中 elearning 是项目名，student 是模块名，dao 用于组织一类组件。

package 语句必须作为源文件的第一条非注释性语句，一个源文件只能指定一个包，即只能包含一条 package 语句，该源文件中可以定义多个类，则这些类将全部位于该包下。

如果没有显式指定 package 语句，则处于默认包下。在实际企业开发中，通常不会把类定义在默认包下，但本书中的大量示例程序为了简单起见，都没有显式指定 package 语句。

同一个包下的类可以自由访问，例如下面的 HelloTest 类，如果把它也放在 lee 包下，则这个 HelloTest 类可以直接访问 Hello 类，无须添加包前缀。

程序清单：codes\05\5.4\HelloTest.java

```
package lee;
public class HelloTest
{
    public static void main(String[] args)
    {
        // 直接访问相同包下的另一个类，无须使用包前缀
        Hello h = new Hello();
    }
}
```

下面代码在 lee 包下再定义一个 sub 子包，并在该包下定义一个 Apple 空类。

```
package lee.sub;
public class Apple{}
```

对于上面的 lee.sub.Apple 类，位于 lee.sub 包下，与 lee.HelloTest 类和 lee.Hello 类不再处于同一个包下，因此使用 lee.sub.Apple 类时就需要使用该类的全名（即包名加类名），即必须使用 lee.sub.Apple 写法来使用该类。

虽然 lee.sub 包是 lee 包的子包，但在 lee.Hello 或 lee.HelloTest 中使用 lee.sub.Apple 类时，依然不能省略前面的 lee 包路径，即在 lee.HelloTest 类和 lee.Hello 类中使用该类时不可写成 sub.Apple，必须写成

完整包路径加类名: lee.sub.Apple。



提示：

父包和子包之间确实表示了某种内在的逻辑关系，例如前面介绍的 org.crazyit.elearning 父包和 org.crazyit.elearning.student 子包，确实可以表明后者是前者的一个模块。但父包和子包在用法上则不存在任何关系，如果父包中的类需要使用子包中的类，则必须使用子包的全名，而不能省略父包部分。

如果创建处于其他包下类的实例，则在调用构造器时也需要使用包前缀。例如在 lee.HelloTest 类中创建 lee.sub.Apple 类的对象，则需要采用如下代码：

```
// 调用构造器时需要在构造器前增加包前缀
lee.sub.Apple a = new lee.sub.Apple();
```

正如上面看到的，如果需要使用不同包中的其他类时，总是需要使用该类的全名，这是一件很烦琐的事情。

为了简化编程，Java 引入了 import 关键字，import 可以向某个 Java 文件中导入指定包层次下某个类或全部类，import 语句应该出现在 package 语句（如果有的话）之后、类定义之前。一个 Java 源文件只能包含一个 package 语句，但可以包含多个 import 语句，多个 import 语句用于导入多个包层次下的类。

使用 import 语句导入单个类的用法如下：

```
import package.subpackage...ClassName;
```

上面语句用于直接导入指定 Java 类。例如导入前面提到的 lee.sub.Apple 类，应该使用下面的代码：

```
import lee.sub.Apple;
```

使用 import 语句导入指定包下全部类的用法如下：

```
import package.subpackage...*;
```

上面 import 语句中的星号 (*) 只能代表类，不能代表包。因此使用 import lee.*; 语句时，它表明导入 lee 包下的所有类，即 Hello 类和 HelloTest 类，而 lee 包下 sub 子包内的类则不会被导入。如需导入 lee.sub.Apple 类，则可以使用 import lee.sub.*; 语句来导入 lee.sub 包下的所有类。

一旦在 Java 源文件中使用 import 语句来导入指定类，在该源文件中使用这些类时就可以省略包前缀，不再需要使用类全名。修改上面的 HelloTest.java 文件，在该文件中使用 import 语句来导入 lee.sub.Apple 类（程序清单同上）。

```
package lee;
//使用 import 导入 lee.sub.Apple 类
import lee.sub.Apple;
public class HelloTest
{
    public static void main(String[] args)
    {
        Hello h = new Hello();
        // 使用类全名的写法
        lee.sub.Apple a = new lee.sub.Apple();
        // 如果使用 import 语句来导入 Apple 类，就可以不再使用类全名了
        Apple aa = new Apple();
    }
}
```

正如上面代码中看到的，通过使用 import 语句可以简化编程。但 import 语句并不是必需的，只要坚持在类里使用其他类的全名，则可以无须使用 import 语句。



注意：

Java 默认为所有源文件导入 java.lang 包下的所有类，因此前面在 Java 程序中使用 String、System 类时都无须使用 import 语句来导入这些类。但对于前面介绍数组时提到的 Arrays 类，其位于 java.util 包下，则必须使用 import 语句来导入该类。



在一些极端的情况下, import 语句也帮不了我们, 此时只能在源文件中使用类全名。例如, 需要在程序中使用 java.sql 包下的类, 也需要使用 java.util 包下的类, 则可以使用如下两行 import 语句:

```
import java.util.*;
import java.sql.*;
```

如果接下来在程序中需要使用 Date 类, 则会引起如下编译错误:

```
HelloTest.java:25: 对 Date 的引用不明确,
java.sql 中的类 java.sql.Date 和 java.util 中的类 java.util.Date 都匹配
```

上面错误提示: 在 HelloTest.java 文件的第 25 行使用了 Date 类, 而 import 语句导入的 java.sql 和 java.util 包下都包含了 Date 类, 系统糊涂了! 再次提醒读者: 不要把系统搞糊涂, 系统一糊涂就是你错了。在这种情况下, 如果需要指定包下的 Date 类, 则只能使用该类的全名。

```
// 为了让引用更加明确, 即使使用了 import 语句, 也还是需要使用类的全名
java.sql.Date d = new java.sql.Date();
```

import 语句可以简化编程, 可以导入指定包下某个类或全部类。

JDK 1.5 以后更是增加了一种静态导入的语法, 它用于导入指定类的某个静态成员变量、方法或全部的静态成员变量、方法。

静态导入使用 import static 语句, 静态导入也有两种语法, 分别用于导入指定类的单个静态成员变量、方法和全部静态成员变量、方法, 其中导入指定类的单个静态成员变量、方法的语法格式如下:

```
import static package.subpackage...ClassName.fieldName|methodName;
```

上面语法导入 package.subpackage...ClassName 类中名为 fieldName 的静态成员变量或者名为 methodName 的静态方法。例如, 可以使用 import static java.lang.System.out; 语句来导入 java.lang.System 类的 out 静态成员变量。

导入指定类的全部静态成员变量、方法的语法格式如下:

```
import static package.subpackage...ClassName.*;
```

上面语法中的星号只能代表静态成员变量或方法名。

import static 语句也放在 Java 源文件的 package 语句 (如果有的话) 之后、类定义之前, 即放在与普通 import 语句相同的位置, 而且 import 语句和 import static 语句之间没有任何顺序要求。

所谓静态成员变量、静态方法其实就是前面介绍的类变量、类方法, 它们都需要使用 static 修饰, 而 static 在很多地方都被翻译为静态, 因此 import static 也就被翻译成了“静态导入”。其实完全可以抛开这个翻译, 用一句话来归纳 import 和 import static 的作用: 使用 import 可以省略写包名; 而使用 import static 则可以连类名都省略。

下面程序使用 import static 语句来导入 java.lang.System 类下的全部静态成员变量, 从而可以将程序简化成如下形式。

程序清单: codes\05\5.4\StaticImportTest.java

```
import static java.lang.System.*;
import static java.lang.Math.*;
public class StaticImportTest
{
    public static void main(String[] args)
    {
        // out 是 java.lang.System 类的静态成员变量, 代表标准输出
        // PI 是 java.lang.Math 类的静态成员变量, 表示  $\pi$  常量
        out.println(PI);
        // 直接调用 Math 类的 sqrt 静态方法
        out.println(sqrt(256));
    }
}
```

从上面程序不难看出, import 和 import static 的功能非常相似, 只是它们导入的对象不一样而已。import 语句和 import static 语句都是用于减少程序中代码编写量的。

现在可以总结出 Java 源文件的大体结构如下:

```
package 语句 // 0个或1个, 必须放在文件开始
import | import static 语句 // 0个或多个, 必须放在所有类定义之前
public classDefinition | interfaceDefinition | enumDefinition
// 0个或1个public类、接口或枚举定义
classDefinition | interfaceDefinition | enumDefinition // 0个或多个普通类、接口或枚举定义
```

上面提到了接口定义、枚举定义, 读者可以暂时把接口、枚举都当成一种特殊的类。

5.4.4 Java 的常用包

Java 的核心类都放在 java 包以及其子包下, Java 扩展的许多类都放在 javax 包以及其子包下。这些实用类也就是前面所说的 API (应用程序接口), Oracle 按这些类的功能分别放在不同的包下。下面几个包是 Java 语言中的常用包。

- java.lang: 这个包下包含了 Java 语言的核心类, 如 String、Math、System 和 Thread 类等, 使用这个包下的类无须使用 import 语句导入, 系统会自动导入这个包下的所有类。
- java.util: 这个包下包含了 Java 的大量工具类/接口和集合框架类/接口, 例如 Arrays 和 List、Set 等。
- java.net: 这个包下包含了一些 Java 网络编程相关的类/接口。
- java.io: 这个包下包含了一些 Java 输入/输出编程相关的类/接口。
- java.text: 这个包下包含了一些 Java 格式化相关的类。
- java.sql: 这个包下包含了 Java 进行 JDBC 数据库编程的相关类/接口。
- java.awt: 这个包下包含了抽象窗口工具集 (Abstract Window Toolkits) 的相关类/接口, 这些类主要用于构建图形用户界面 (GUI) 程序。
- java.swing: 这个包下包含了 Swing 图形用户界面编程的相关类/接口, 这些类可用于构建平台无关的 GUI 程序。

读者现在只需对这些包有一个大致印象即可, 随着本书后面的介绍, 读者会逐渐熟悉这些包下各类和接口的用法。

5.5 深入构造器

构造器是一个特殊的方法, 这个特殊方法用于创建实例时执行初始化。构造器是创建对象的重要途径 (即使使用工厂模式、反射等方式创建对象, 其实质依然是依赖于构造器), 因此, Java 类必须包含一个或一个以上的构造器。

5.5.1 使用构造器执行初始化

构造器最大的用处就是在创建对象时执行初始化。前面已经介绍过了, 当创建一个对象时, 系统为这个对象的实例变量进行默认初始化, 这种默认的初始化把所有基本类型的实例变量设为 0 (对数值型实例变量) 或 false (对布尔型实例变量), 把所有引用类型的实例变量设为 null。

如果想改变这种默认的初始化, 想让系统创建对象时就为该对象的实例变量显式指定初始值, 就可以通过构造器来实现。



注意:

如果程序员没有为 Java 类提供任何构造器, 则系统会为这个类提供一个无参数的构造器, 这个构造器的执行体为空, 不做任何事情。无论如何, Java 类至少包含一个构造器。



下面类提供了一个自定义的构造器, 通过这个构造器就可以让程序员进行自定义的初始化操作。

程序清单: codes\05\5.5\ConstructorTest.java

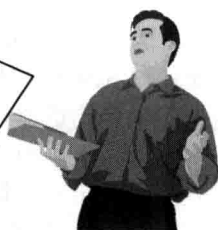
```
public class ConstructorTest
{
```

```
public String name;
public int count;
// 提供自定义的构造器, 该构造器包含两个参数
public ConstructorTest(String name , int count)
{
    // 构造器里的 this 代表它进行初始化的对象
    // 下面两行代码将传入的 2 个参数赋给 this 代表对象的 name 和 count 实例变量
    this.name = name;
    this.count = count;
}
public static void main(String[] args)
{
    // 使用自定义的构造器来创建对象
    // 系统将会对该对象执行自定义的初始化
    ConstructorTest tc = new ConstructorTest("疯狂 Java 讲义" , 90000);
    // 输出 ConstructorTest 对象的 name 和 count 两个实例变量
    System.out.println(tc.name);
    System.out.println(tc.count);
}
```

运行上面程序, 将看到输出 ConstructorTest 对象时, 它的 name 实例变量不再是 null, 而且 count 实例变量也不再是 0, 这就是提供自定义构造器的作用。

学生提问: 构造器是创建 Java 对象的途径, 是不是说构造器完全负责创建 Java 对象?

答: 不是! 构造器是创建 Java 对象的重要途径, 通过 new 关键字调用构造器时, 构造器也确实返回了该类的对象, 但这个对象并不是完全由构造器负责创建的。实际上, 当程序员调用构造器时, 系统会先为该对象分配内存空间, 并为这个对象执行默认初始化, 这个对象已经产生了——这些操作在构造器执行之前就都完成了。也就是说, 当系统开始执行构造器的执行体之前, 系统已经创建了一个对象, 只是这个对象还不能被外部程序访问, 只能在该构造器中通过 this 来引用。当构造器的执行体执行结束后, 这个对象作为构造器的返回值被返回, 通常还会赋给另一个引用类型的变量, 从而让外部程序可以访问该对象。



一旦程序员提供了自定义的构造器, 系统就不再提供默认的构造器, 因此上面的 ConstructorTest 类不能再通过 new ConstructorTest(); 代码来创建实例, 因为该类不再包含无参数的构造器。

如果用户希望该类保留无参数的构造器, 或者希望有多个初始化过程, 则可以为该类提供多个构造器。如果一个类里提供了多个构造器, 就形成了构造器的重载。

因为构造器主要用于被其他方法调用, 用以返回该类的实例, 因而通常把构造器设置成 public 访问权限, 从而允许系统中任何位置的类来创建该类的对象。除非在一些极端的情况下, 业务需要限制创建该类的对象, 可以把构造器设置成其他访问权限, 例如设置为 protected, 主要用于被其子类调用; 把其设置为 private, 阻止其他类创建该类的实例。

5.5.2 构造器重载

同一个类里具有多个构造器, 多个构造器的形参列表不同, 即被称为构造器重载。构造器重载允许 Java 类里包含多个初始化逻辑, 从而允许使用不同的构造器来初始化 Java 对象。

构造器重载和方法重载基本相似: 要求构造器的名字相同, 这一点无须特别要求, 因为构造器必须与类名相同, 所以同一个类的所有构造器名肯定相同。为了让系统能区分不同的构造器, 多个构造器的参数列表必须不同。

下面的 Java 类示范了构造器重载, 利用构造器重载就可以通过不同的构造器来创建 Java 对象。

程序清单: codes\05\5.5\ConstructorOverload.java

```
public class ConstructorOverload
{
    public String name;
    public int count;
    // 提供无参数的构造器
    public ConstructorOverload(){ }
    // 提供带两个参数的构造器
    // 对该构造器返回的对象执行初始化
    public ConstructorOverload(String name , int count)
    {
        this.name = name;
        this.count = count;
    }
    public static void main(String[] args)
    {
        // 通过无参数构造器创建 ConstructorOverload 对象
        ConstructorOverload oc1 = new ConstructorOverload();
        // 通过有参数构造器创建 ConstructorOverload 对象
        ConstructorOverload oc2 = new ConstructorOverload(
            "轻量级 Java EE 企业应用实战", 300000);
        System.out.println(oc1.name + " " + oc1.count);
        System.out.println(oc2.name + " " + oc2.count);
    }
}
```

上面的 ConstructorOverload 类提供了两个重载的构造器,两个构造器的名字相同,但形参列表不同。系统通过 new 调用构造器时,系统将根据传入的实参列表来决定调用哪个构造器。

如果系统中包含了多个构造器,其中一个构造器的执行体里完全包含另一个构造器的执行体,如图 5.16 所示。

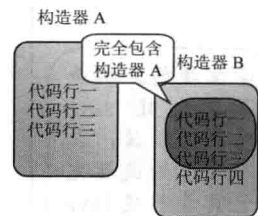


图 5.16 构造器 B 完全包含构造器 A

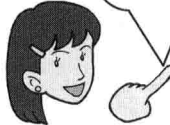
从图 5.16 中可以看出,构造器 B 完全包含了构造器 A。对于这种完全包含的情况,如果是两个方法之间存在这种关系,则可在方法 B 中调用方法 A。但构造器不能被直接调用,构造器必须使用 new 关键字来调用。但一旦使用 new 关键字来调用构造器,将会导致系统重新创建一个对象。为了在构造器 B 中调用构造器 A 中的初始化代码,又不会重新创建一个 Java 对象,可以使用 this 关键字来调用相应的构造器。下面代码实现了在一个构造器中直接使用另一个构造器的初始化代码。

程序清单: codes\05\5.5\Apple.java

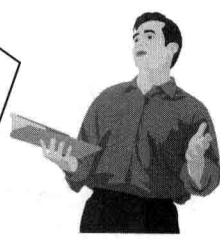
```
public class Apple
{
    public String name;
    public String color;
    public double weight;
    public Apple(){ }
    // 两个参数的构造器
    public Apple(String name , String color)
    {
        this.name = name;
        this.color = color;
    }
    // 三个参数的构造器
    public Apple(String name , String color , double weight)
    {
        // 通过 this 调用另一个重载的构造器的初始化代码
        this(name , color);
        // 下面 this 引用该构造器正在初始化的 Java 对象
        this.weight = weight;
    }
}
```

上面的 Apple 类里包含了三个构造器,其中第三个构造器通过 this 来调用另一个重载构造器的初始化代码。程序中 this(name, color);调用表明调用该类另一个带两个字符串参数的构造器。

使用 this 调用另一个重载的构造器只能在构造器中使用,而且必须作为构造器执行体的第一条语句。使用 this 调用重载的构造器时,系统会根据 this 后括号里的实参来调用形参列表与之对应的构造器。



学生提问:为什么要用 this 来调用另一个重载的构造器?我把另一个构造器里的代码复制、粘贴到这个构造器里不就可以了么?



答:如果仅仅从软件功能实现上来看,这样复制、粘贴确实可以实现这个效果;但从软件工程的角度来看,这样做是相当糟糕的。在软件开发里有一个规则:不要把相同的代码段书写两次以上!因为软件是一个需要不断更新的产品,如果有一天需要更新图 5.16 中构造器 A 的初始化代码,假设构造器 B、构造器 C……里都包含了相同的初始化代码,则需要同时打开构造器 A、构造器 B、构造器 C……的代码进行修改;反之,如果构造器 B、构造器 C……是通过 this 调用了构造器 A 的初始化代码,则只需要打开构造器 A 进行修改即可。因此,尽量避免相同的代码重复出现,充分复用每一段代码,既可以让程序代码更加简洁,也可以降低软件的维护成本。

5.6 类的继承

继承是面向对象的三大特征之一,也是实现软件复用的重要手段。Java 的继承具有单继承的特点,每个子类只有一个直接父类。

5.6.1 继承的特点

Java 的继承通过 extends 关键字来实现,实现继承的类被称为子类,被继承的类被称为父类,有的也称其为基类、超类。父类和子类的关系,是一种一般和特殊的关系。例如果和苹果的关系,苹果继承了水果,苹果是水果的子类,则苹果是一种特殊的水果。

因为子类是一种特殊的父类,因此父类包含的范围总比子类包含的范围要大,所以可以认为父类是大类,而子类是小类。

Java 里子类继承父类的语法格式如下:

```
修饰符 class SubClass extends SuperClass
{
    //类定义部分
}
```

从上面语法格式来看,定义子类的语法非常简单,只需在原来的类定义上增加 extends SuperClass 即可,即表明该子类继承了 SuperClass 类。

Java 使用 extends 作为继承的关键字,extends 关键字在英文中是扩展,而不是继承!这个关键字很好地体现了子类和父类的关系:子类是对父类的扩展,子类是一种特殊的父类。从这个意义上来看,使用继承来描述子类和父类的关系是错误的,用扩展更恰当。因此这样的说法更加准确:Apple 类扩展了 Fruit 类。

为什么国内把 extends 翻译为“继承”呢?除了与历史原因有关之外,把 extends 翻译为“继承”也是有其理由的:子类扩展了父类,将可以获得父类的全部成员变量和方法,这与汉语中的继承(子辈从父辈那里获得一笔财富称为继承)具有很好的类似性。值得指出的是,Java 的子类不能获得父类的构造器。

下面程序示范了子类继承父类的特点。下面是 Fruit 类的代码。

程序清单: codes\05\5.6\Fruit.java

```
public class Fruit
{
    public double weight;
    public void info()
    {
        System.out.println("我是一个水果! 重"
            + weight + "g! ");
    }
}
```

接下来再定义该 Fruit 类的子类 Apple, 程序如下。

程序清单: codes\05\5.6\Apple.java

```
public class Apple extends Fruit
{
    public static void main(String[] args)
    {
        // 创建 Apple 对象
        Apple a = new Apple();
        // Apple 对象本身没有 weight 成员变量
        // 因为 Apple 的父类有 weight 成员变量, 也可以访问 Apple 对象的 weight 成员变量
        a.weight = 56;
        // 调用 Apple 对象的 info() 方法
        a.info();
    }
}
```

上面的 Apple 类基本只是一个空类, 它只包含了一个 main() 方法, 但程序中创建了 Apple 对象之后, 可以访问该 Apple 对象的 weight 实例变量和 info() 方法, 这表明 Apple 对象也具有了 weight 实例变量和 info() 方法, 这就是继承的作用。

Java 语言摒弃了 C++ 中难以理解的多继承特征, 即每个类最多只有一个直接父类。例如下面代码将会引起编译错误。

```
class SubClass extends Base1 , Base2 , Base3{...}
```

很多书在介绍 Java 的单继承时, 可能会说 Java 类只能有一个父类, 严格来讲, 这种说法是错误的, 应该换成如下说法: Java 类只能有一个直接父类, 实际上, Java 类可以有无限多个间接父类。例如:

```
class Fruit extends Plant{...}
class Apple extends Fruit{...}
```

上面的类定义中 Fruit 是 Apple 类的父类, Plant 类也是 Apple 类的父类。区别是 Fruit 是 Apple 的直接父类, 而 Plant 则是 Apple 类的间接父类。

如果定义一个 Java 类时并未显式指定这个类的直接父类, 则这个类默认扩展 java.lang.Object 类。因此, java.lang.Object 类是所有类的父类, 要么是其直接父类, 要么是其间接父类。因此所有的 Java 对象都可调用 java.lang.Object 类所定义的实例方法。关于 java.lang.Object 类的介绍请参考 7.3.1 节。

从子类角度来看, 子类扩展 (extends) 了父类; 但从父类的角度来看, 父类派生 (derive) 出了子类。也就是说, 扩展和派生所描述的是同一个动作, 只是观察角度不同而已。

5.6.2 重写父类的方法

子类扩展了父类, 子类是一个特殊的父类。大部分时候, 子类总是以父类为基础, 额外增加新的成员变量和方法。但有一种情况例外: 子类需要重写父类的方法。例如鸟类都包含了飞翔方法, 其中鸵鸟是一种特殊的鸟类, 因此鸵鸟应该是鸟的子类, 因此它也将从鸟类获得飞翔方法, 但这个飞翔方法明显不适合鸵鸟, 为此, 鸵鸟需要重写鸟类的方法。

下面程序先定义了一个 Bird 类。

程序清单: codes\05\5.6\Bird.java

```
public class Bird
```

```

{
    // Bird 类的 fly() 方法
    public void fly()
    {
        System.out.println("我在天空里自由自在地飞翔...");
    }
}

```

下面再定义一个 Ostrich 类, 这个类扩展了 Bird 类, 重写了 Bird 类的 fly() 方法。

程序清单: codes\05\5.6\Ostrich.java

```

public class Ostrich extends Bird
{
    // 重写 Bird 类的 fly() 方法
    public void fly()
    {
        System.out.println("我只能在地上奔跑...");
    }
    public static void main(String[] args)
    {
        // 创建 Ostrich 对象
        Ostrich os = new Ostrich();
        // 执行 Ostrich 对象的 fly() 方法, 将输出"我只能在地上奔跑..."
        os.fly();
    }
}

```

执行上面程序, 将看到执行 os.fly() 时执行的不再是 Bird 类的 fly() 方法, 而是执行 Ostrich 类的 fly() 方法。

这种子类包含与父类同名方法的现象被称为方法重写 (Override), 也被称为方法覆盖。可以说子类重写了父类的方法, 也可以说子类覆盖了父类的方法。

方法的重写要遵循“两同两小一大”规则, “两同”即方法名相同、形参列表相同; “两小”指的是子类方法返回值类型应比父类方法返回值类型更小或相等, 子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等; “一大”指的是子类方法的访问权限应比父类方法的访问权限更大或相等。尤其需要指出的是, 覆盖方法和被覆盖方法要么都是类方法, 要么都是实例方法, 不能一个是类方法, 一个是实例方法。例如, 如下代码将会引发编译错误。

```

class BaseClass
{
    public static void test(){...}
}
class SubClass extends BaseClass
{
    public void test(){...}
}

```

当子类覆盖了父类方法后, 子类的对象将无法访问父类中被覆盖的方法, 但可以在子类方法中调用父类中被覆盖的方法。如果需要在子类方法中调用父类中被覆盖的方法, 则可以使用 super (被覆盖的是实例方法) 或者父类类名 (被覆盖的是类方法) 作为调用者来调用父类中被覆盖的方法。

如果父类方法具有 private 访问权限, 则该方法对其子类是隐藏的, 因此其子类无法访问该方法, 也就是无法重写该方法。如果子类中定义了一个与父类 private 方法具有相同的方法名、相同的形参列表、相同的返回值类型的方法, 依然不是重写, 只是在子类中重新定义了一个新方法。例如, 下面代码是完全正确的。

```

class BaseClass
{
    // test() 方法是 private 访问权限, 子类不可访问该方法
    private void test(){...}
}
class SubClass extends BaseClass
{

```



```
// 此处并不是方法重写，所以可以增加 static 关键字
public static void test(){...}
}
```

方法重载和方法重写在英语中分别是 **overload** 和 **override**，经常看到有些初学者或一些低水平的公司喜欢询问重载和重写的区别？其实把重载和重写放在一起比较本身没有太大的意义，因为重载主要发生在同一个类的多个同名方法之间，而重写发生在子类和父类的同名方法之间。它们之间的联系很少，除了二者都是发生在方法之间，并要求方法名相同之外，没有太大的相似之处。当然，父类方法和子类方法之间也可能发生重载，因为子类会获得父类方法，如果子类定义了一个与父类方法有相同的方法名，但参数列表不同的方法，就会形成父类方法和子类方法的重载。

5.6.3 super 限定

如果需要在子类方法中调用父类被覆盖的实例方法，则可使用 **super** 限定来调用父类被覆盖的实例方法。为上面的 **Ostrich** 类添加一个方法，在这个方法中调用 **Bird** 类中被覆盖的 **fly** 方法。

```
public void callOverridedMethod()
{
    // 在子类方法中通过 super 显式调用父类被覆盖的实例方法
    super.fly();
}
```

借助 **callOverridedMethod()** 方法的帮助，就可以让 **Ostrich** 对象既可以调用自己重写的 **fly()** 方法，也可以调用 **Bird** 类中被覆盖的 **fly()** 方法（调用 **callOverridedMethod()** 方法即可）。

super 是 Java 提供的一个关键字，**super** 用于限定该对象调用它从父类继承得到的实例变量或方法。正如 **this** 不能出现在 **static** 修饰的方法中一样，**super** 也不能出现在 **static** 修饰的方法中。**static** 修饰的方法是属于类的，该方法的调用者可能是一个类，而不是对象，因而 **super** 限定也就失去了意义。

如果在构造器中使用 **super**，则 **super** 用于限定该构造器初始化的是该对象从父类继承得到的实例变量，而不是该类自己定义的实例变量。

如果子类定义了和父类同名的实例变量，则会发生子类实例变量隐藏父类实例变量的情形。在正常情况下，子类里定义的方法直接访问该实例变量默认会访问到子类中定义的实例变量，无法访问到父类中被隐藏的实例变量。在子类定义的实例方法中可以通过 **super** 来访问父类中被隐藏的实例变量，如下代码所示。

程序清单：codes\05\5.6\SubClass.java

```
class BaseClass
{
    public int a = 5;
}
public class SubClass extends BaseClass
{
    public int a = 7;
    public void accessOwner()
    {
        System.out.println(a);
    }
    public void accessBase()
    {
        // 通过 super 来限定访问从父类继承得到的 a 实例变量
        System.out.println(super.a);
    }
    public static void main(String[] args)
    {
        SubClass sc = new SubClass();
        sc.accessOwner(); // 输出 7
        sc.accessBase(); // 输出 5
    }
}
```

上面程序的 **BaseClass** 和 **SubClass** 中都定义了名为 **a** 的实例变量，则 **SubClass** 的 **a** 实例变量将会隐藏 **BaseClass** 的 **a** 实例变量。当系统创建了 **SubClass** 对象时，实际上会为 **SubClass** 对象分配两块内存，

一块用于存储在 SubClass 类中定义的 a 实例变量,一块用于存储从 BaseClass 类继承得到的 a 实例变量。

程序中粗体字代码访问 super.a 时,此时使用 super 限定访问该实例从父类继承得到的 a 实例变量,而不是在当前类中定义的 a 实例变量。

如果子类里没有包含和父类同名的成员变量,那么在子类实例方法中访问该成员变量时,则无须显式使用 super 或父类名作为调用者。如果在某个方法中访问名为 a 的成员变量,但没有显式指定调用者,则系统查找 a 的顺序为:

- (1) 查找该方法中是否有名为 a 的局部变量。
- (2) 查找当前类中是否包含名为 a 的成员变量。
- (3) 查找 a 的直接父类中是否包含名为 a 的成员变量,依次上溯 a 的所有父类,直到 java.lang.Object 类,如果最终不能找到名为 a 的成员变量,则系统出现编译错误。

如果被覆盖的是类变量,在子类的方法中则可以通过父类名作为调用者来访问被覆盖的类变量。



提示:

当程序创建一个子类对象时,系统不仅会为该类中定义的实例变量分配内存,也会为它从父类继承得到的所有实例变量分配内存,即使子类定义了与父类中同名的实例变量。也就是说,当系统创建一个 Java 对象时,如果该 Java 类有两个父类(一个直接父类 A,一个间接父类 B),假设 A 类中定义了 2 个实例变量,B 类中定义了 3 个实例变量,当前类中定义了 2 个实例变量,那么这个 Java 对象将会保存 2+3+2 个实例变量。

如果在子类里定义了与父类中已有变量同名的变量,那么子类中定义的变量会隐藏父类中定义的变量。注意不是完全覆盖,因此系统在创建子类对象时,依然会为父类中定义的、被隐藏的变量分配内存空间。



注意:

为了在子类方法中访问父类中定义的、被隐藏的实例变量,或为了在子类方法中调用父类中定义的、被覆盖(Override)的方法,可以通过 super 作为限定来调用这些实例变量和实例方法。



因为子类中定义与父类中同名的实例变量并不会完全覆盖父类中定义的实例变量,它只是简单地隐藏了父类中的实例变量,所以会出现如下特殊的情形。

程序清单: codes\05\5.6\HideTest.java

```
class Parent
{
    public String tag = "疯狂 Java 讲义";           // ①
}
class Derived extends Parent
{
    // 定义一个私有的 tag 实例变量来隐藏父类的 tag 实例变量
    private String tag = "轻量级 Java EE 企业应用实战"; // ②
}
public class HideTest
{
    public static void main(String[] args)
    {
        Derived d = new Derived();
        // 程序不可访问 d 的私有变量 tag, 所以下面语句将引起编译错误
        // System.out.println(d.tag);           // ③
        // 将 d 变量显式地向上转型为 Parent 后, 即可访问 tag 实例变量
        // 程序将输出: "疯狂 Java 讲义"
        System.out.println(((Parent)d).tag);      // ④
    }
}
```

上面程序的①行粗体字代码为父类 Parent 定义了一个 tag 实例变量,②行粗体字代码为其子类定义

了一个 `private` 的 `tag` 实例变量，子类中定义的这个实例变量将会隐藏父类中定义的 `tag` 实例变量。

程序的入口 `main()` 方法中先创建了一个 `Derived` 对象。这个 `Derived` 对象将会保存两个 `tag` 实例变量，一个是在 `Parent` 类中定义的 `tag` 实例变量，一个是在 `Derived` 类中定义的 `tag` 实例变量。此时程序中包括一个 `d` 变量，它引用一个 `Derived` 对象，内存中的存储示意图如图 5.17 所示。

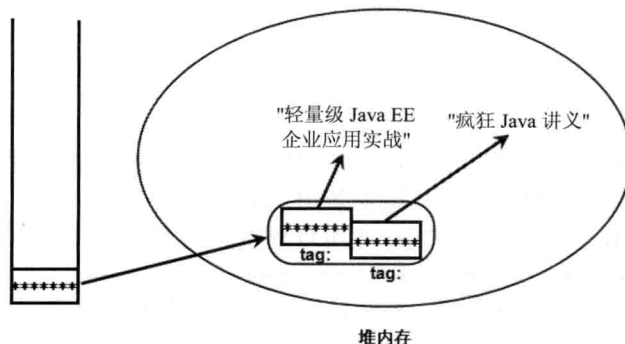


图 5.17 子类的实例变量隐藏父类的实例变量存储示意图

接着，程序将 `Derived` 对象赋给 `d` 变量，当在③行粗体字代码处试图通过 `d` 来访问 `tag` 实例变量时，程序将提示访问权限不允许。这是因为访问哪个实例变量由声明该变量的类型决定，所以系统将会试图访问在②行粗体代码处定义的 `tag` 实例变量；程序在④行粗体字代码处先将 `d` 变量强制向上转型为 `Parent` 类型，再通过它来访问 `tag` 实例变量是允许的，因为此时系统将会访问在①行粗体字代码处定义的 `tag` 实例变量，也就是输出“疯狂 Java 讲义”。

5.6.4 调用父类构造器

子类不会获得父类的构造器，但子类构造器里可以调用父类构造器的初始化代码，类似于前面所介绍的一个构造器调用另一个重载的构造器。

在一个构造器中调用另一个重载的构造器使用 `this` 调用来完成，在子类构造器中调用父类构造器使用 `super` 调用来完成。

看下面程序定义了 `Base` 类和 `Sub` 类，其中 `Sub` 类是 `Base` 类的子类，程序在 `Sub` 类的构造器中使用 `super` 来调用 `Base` 构造器的初始化代码。

程序清单：codes\05\5.6\Sub.java

```
class Base
{
    public double size;
    public String name;
    public Base(double size , String name)
    {
        this.size = size;
        this.name = name;
    }
}

public class Sub extends Base
{
    public String color;
    public Sub(double size , String name , String color)
    {
        // 通过 super 调用来调用父类构造器的初始化过程
        super(size , name);
        this.color = color;
    }

    public static void main(String[] args)
    {
        Sub s = new Sub(5.6 , "测试对象" , "红色");
        // 输出 Sub 对象的三个实例变量
        System.out.println(s.size + "--" + s.name
```

```

        + "--" + s.color);
    }
}

```

从上面程序中不难看出, 使用 `super` 调用和使用 `this` 调用也很像, 区别在于 `super` 调用的是其父类的构造器, 而 `this` 调用的是同一个类中重载的构造器。因此, 使用 `super` 调用父类构造器也必须出现在子类构造器执行体的第一行, 所以 `this` 调用和 `super` 调用不会同时出现。

不管是否使用 `super` 调用来执行父类构造器的初始化代码, 子类构造器总会调用父类构造器一次。子类构造器调用父类构造器分如下几种情况。

- 子类构造器执行体的第一行使用 `super` 显式调用父类构造器, 系统将根据 `super` 调用里传入的实参列表调用父类对应的构造器。
- 子类构造器执行体的第一行代码使用 `this` 显式调用本类中重载的构造器, 系统将根据 `this` 调用里传入的实参列表调用本类中的另一个构造器。执行本类中另一个构造器时即会调用父类构造器。
- 子类构造器执行体中既没有 `super` 调用, 也没有 `this` 调用, 系统将会在执行子类构造器之前, 隐式调用父类无参数的构造器。

不管上面哪种情况, 当调用子类构造器来初始化子类对象时, 父类构造器总会在子类构造器之前执行; 不仅如此, 执行父类构造器时, 系统会再次上溯执行其父类构造器……依此类推, 创建任何 Java 对象, 最先执行的总是 `java.lang.Object` 类的构造器。

对于如图 5.18 所示的继承树: 如果创建 `ClassB` 的对象, 系统将先执行 `java.lang.Object` 类的构造器, 再执行 `ClassA` 类的构造器, 然后才执行 `ClassB` 类的构造器, 这个执行过程还是最基本的情况。如果 `ClassB` 显式调用 `ClassA` 的构造器, 而该构造器又调用了 `ClassA` 类中重载的构造器, 则会看到 `ClassA` 两个构造器先后执行的情形。

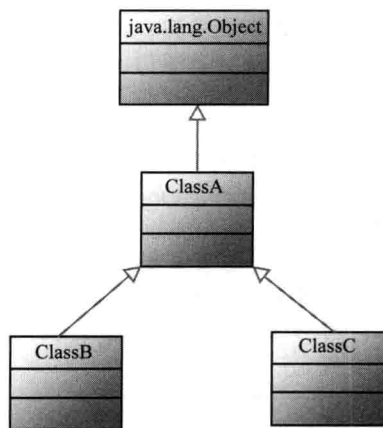


图 5.18 继承树

下面程序定义了三个类, 它们之间有严格的继承关系, 通过这种继承关系让读者看到构造器之间的调用关系。

程序清单: codes\05\5.6\Wolf.java

```

class Creature
{
    public Creature()
    {
        System.out.println("Creature 无参数的构造器");
    }
}
class Animal extends Creature
{
    public Animal(String name)
    {
        System.out.println("Animal 带一个参数的构造器, "
            + "该动物的 name 为" + name);
    }
}

```



```

    }
    public Animal(String name , int age)
    {
        // 使用 this 调用同一个重载的构造器
        this(name);
        System.out.println("Animal 带两个参数的构造器, "
            + "其 age 为" + age);
    }
}
public class Wolf extends Animal
{
    public Wolf()
    {
        // 显式调用父类有两个参数的构造器
        super("灰太狼", 3);
        System.out.println("Wolf 无参数的构造器");
    }
    public static void main(String[] args)
    {
        new Wolf();
    }
}

```

上面程序的 main 方法只创建了一个 Wolf 对象，但系统在底层完成了复杂的操作。运行上面程序，看到如下运行结果：

```

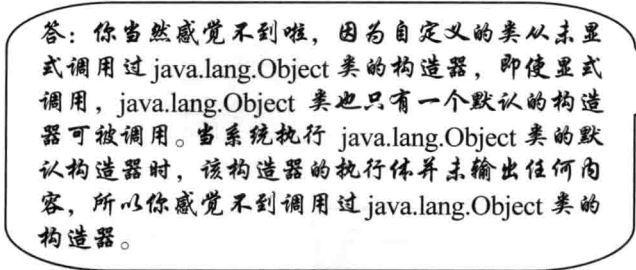
Creature 无参数的构造器
Animal 带一个参数的构造器，该动物的 name 为灰太狼
Animal 带两个参数的构造器，其 age 为 3
Wolf 无参数的构造器

```

从上面运行过程来看，创建任何对象总是从该类所在继承树最顶层类的构造器开始执行，然后依次向下执行，最后才执行本类的构造器。如果某个父类通过 this 调用了同类中重载的构造器，就会依次执行此父类的多个构造器。



学生提问：为什么我创建 Java 对象时从未感觉到 java.lang.Object 类的构造器被调用过？



答：你当然感觉不到啦，因为自定义的类从未显式调用过 java.lang.Object 类的构造器，即使显式调用，java.lang.Object 类也只有一个默认的构造器可被调用。当系统执行 java.lang.Object 类的默认构造器时，该构造器的执行体并未输出任何内容，所以你感觉不到调用过 java.lang.Object 类的构造器。

5.7 多态

Java 引用变量有两个类型：一个是编译时类型，一个是运行时类型。编译时类型由声明该变量时使用的类型决定，运行时类型由实际赋给该变量的对象决定。如果编译时类型和运行时类型不一致，就可能出现所谓的多态（Polymorphism）。

>> 5.7.1 多态性

先看下面程序。

程序清单：codes\05\5.7\SubClass.java

```

class BaseClass
{
    public int book = 6;
    public void base()
    {
    }
}

```

```

        System.out.println("父类的普通方法");
    }
    public void test()
    {
        System.out.println("父类的被覆盖的方法");
    }
}
public class SubClass extends BaseClass
{
    // 重新定义一个 book 实例变量隐藏父类的 book 实例变量
    public String book = "轻量级 Java EE 企业应用实战";
    public void test()
    {
        System.out.println("子类的覆盖父类的方法");
    }
    public void sub()
    {
        System.out.println("子类的普通方法");
    }
    public static void main(String[] args)
    {
        // 下面编译时类型和运行时类型完全一样, 因此不存在多态
        BaseClass bc = new BaseClass();
        // 输出 6
        System.out.println(bc.book);
        // 下面两次调用将执行 BaseClass 的方法
        bc.base();
        bc.test();
        // 下面编译时类型和运行时类型完全一样, 因此不存在多态
        SubClass sc = new SubClass();
        // 输出"轻量级 Java EE 企业应用实战"
        System.out.println(sc.book);
        // 下面调用将执行从父类继承到的 base() 方法
        sc.base();
        // 下面调用将执行当前类的 test() 方法
        sc.test();
        // 下面编译时类型和运行时类型不一样, 多态发生
        BaseClass polymorphicBc = new SubClass();
        // 输出 6 —— 表明访问的是父类对象的实例变量
        System.out.println(polymorphicBc.book);
        // 下面调用将执行从父类继承到的 base() 方法
        polymorphicBc.base();
        // 下面调用将执行当前类的 test() 方法
        polymorphicBc.test();
        // 因为 polymorphicBc 的编译时类型是 BaseClass
        // BaseClass 类没有提供 sub() 方法, 所以下面代码编译时会出现错误
        // polymorphicBc.sub();
    }
}

```

上面程序的 main() 方法中显式创建了三个引用变量, 对于前两个引用变量 bc 和 sc, 它们编译时类型和运行时类型完全相同, 因此调用它们的成员变量和方法非常正常, 完全没有任何问题。但第三个引用变量 polymorphicBc 则比较特殊, 它的编译时类型是 BaseClass, 而运行时类型是 SubClass, 当调用该引用变量的 test() 方法 (BaseClass 类中定义了该方法, 子类 SubClass 覆盖了父类的该方法) 时, 实际执行的是 SubClass 类中覆盖后的 test() 方法, 这就可能出现多态了。

因为子类其实是一种特殊的父类, 因此 Java 允许把一个子类对象直接赋给一个父类引用变量, 无须任何类型转换, 或者被称为向上转型 (upcasting), 向上转型由系统自动完成。

当把一个子类对象直接赋给父类引用变量时, 例如上面的 BaseClass polymorphicBc = new SubClass();, 这个 polymorphicBc 引用变量的编译时类型是 BaseClass, 而运行时类型是 SubClass, 当运行时调用该引用变量的方法时, 其方法行为总是表现出子类方法的行为特征, 而不是父类方法的行为特征, 这就可能出现: 相同类型的变量、调用同一个方法时呈现出多种不同的行为特征, 这就是多态。

上面的 main() 方法中注释了 polymorphicBc.sub();, 这行代码会在编译时引发错误。虽然 polymorphicBc 引用变量实际上确实包含 sub() 方法 (例如, 可以通过反射来执行该方法), 但因为它的编译时类型为

BaseClass, 因此编译时无法调用 sub()方法。

与方法不同的是, 对象的实例变量则不具备多态性。比如上面的 polymorphicBc 引用变量, 程序中输出它的 book 实例变量时, 并不是输出 SubClass 类里定义的实例变量, 而是输出 BaseClass 类的实例变量。



注意：

引用变量在编译阶段只能调用其编译时类型所具有的方法, 但运行时则执行它运行时类型所具有的方法。因此, 编写 Java 代码时, 引用变量只能调用声明该变量时所用类里包含的方法。例如, 通过 Object p = new Person() 代码定义一个变量 p, 则这个 p 只能调用 Object 类的方法, 而不能调用 Person 类里定义的方法。



注意：

通过引用变量来访问其包含的实例变量时, 系统总是试图访问它编译时类型所定义的成员变量, 而不是它运行时类型所定义的成员变量。



►► 5.7.2 引用变量的强制类型转换

编写 Java 程序时, 引用变量只能调用它编译时类型的方法, 而不能调用它运行时类型的方法, 即使它实际所引用的对象确实包含该方法。如果需要让这个引用变量调用它运行时类型的方法, 则必须把它强制类型转换成运行时类型, 强制类型转换需要借助于类型转换运算符。

类型转换运算符是小括号, 类型转换运算符的用法是: (type)variable, 这种用法可以将 variable 变量转换成一个 type 类型的变量。前面在介绍基本类型的强制类型转换时, 已经看到了使用这种类型转换运算符的用法, 类型转换运算符可以将一个基本类型变量转换成另一个类型。

除此之外, 这个类型转换运算符还可以将一个引用类型变量转换成其子类类型。这种强制类型转换不是万能的, 当进行强制类型转换时需要注意:

- 基本类型之间的转换只能在数值类型之间进行, 这里所说的数值类型包括整数型、字符型和浮点型。但数值类型和布尔类型之间不能进行类型转换。
- 引用类型之间的转换只能在具有继承关系的两个类型之间进行, 如果是两个没有任何继承关系的类型, 则无法进行类型转换, 否则编译时就会出现错误。如果试图把一个父类实例转换成子类类型, 则这个对象必须实际上是子类实例才行 (即编译时类型为父类类型, 而运行时类型是子类类型), 否则将在运行时引发 ClassCastException 异常。

下面是进行强制类型转换的示范程序。下面程序详细说明了哪些情况可以进行类型转换, 哪些情况不可以进行类型转换。

程序清单: codes\05\5.7\ConversionTest.java

```
public class ConversionTest
{
    public static void main(String[] args)
    {
        double d = 13.4;
        long l = (long)d;
        System.out.println(l);
        int in = 5;
        // 试图把一个数值类型的变量转换为 boolean 类型, 下面代码编译出错
        // 编译时会提示: 不可转换的类型
        // boolean b = (boolean)in;
        Object obj = "Hello";
        // obj 变量的编译时类型为 Object, Object 与 String 存在继承关系, 可以强制类型转换
        // 而且 obj 变量的实际类型是 String, 所以运行时也可通过
        String objStr = (String)obj;
        System.out.println(objStr);
    }
}
```



```
// 定义一个 objPri 变量, 编译时类型为 Object, 实际类型为 Integer
Object objPri = new Integer(5);
// objPri 变量的编译时类型为 Object, objPri 的运行时类型为 Integer
// Object 与 Integer 存在继承关系
// 可以强制类型转换, 而 objPri 变量的实际类型是 Integer
// 所以下面代码运行时引发 ClassCastException 异常
String str = (String)objPri;
}
}
```

考虑到进行强制类型转换时可能出现异常, 因此进行类型转换之前应先通过 `instanceof` 运算符来判断是否可以成功转换。例如, 上面的 `String str = (String)objPri;` 代码运行时会引发 `ClassCastException` 异常, 这是因为 `objPri` 不可转换成 `String` 类型。为了让程序更加健壮, 可以将代码改为如下:

```
if (objPri instanceof String)
{
    String str = (String)objPri;
}
```

在进行强制类型转换之前, 先用 `instanceof` 运算符判断是否可以成功转换, 从而避免出现 `ClassCastException` 异常, 这样可以保证程序更加健壮。



注意:

当把子类对象赋给父类引用变量时, 被称为向上转型 (upcasting), 这种转型总是可以成功的, 这也从另一个侧面证实了子类是一种特殊的父类。这种转型只是表明这个引用变量的编译时类型是父类, 但实际执行它的方法时, 依然表现出子类对象的行为方式。但把一个父类对象赋给子类引用变量时, 就需要进行强制类型转换, 而且还可能在运行时产生 `ClassCastException` 异常, 使用 `instanceof` 运算符可以让强制类型转换更安全。



`instanceof` 和类型转换运算符一样, 都是 Java 提供的运算符, 与 +、- 等算术运算符的用法大致相似, 下面具体介绍该运算符的用法。

►► 5.7.3 instanceof 运算符

`instanceof` 运算符的前一个操作数通常是一个引用类型变量, 后一个操作数通常是一个类 (也可以是接口, 可以把接口理解成一种特殊的类), 它用于判断前面的对象是否是后面的类, 或者其子类、实现类的实例。如果是, 则返回 `true`, 否则返回 `false`。

在使用 `instanceof` 运算符时需要注意: `instanceof` 运算符前面操作数的编译时类型要么与后面的类相同, 要么与后面的类具有父子继承关系, 否则会引起编译错误。下面程序示范了 `instanceof` 运算符的用法。

程序清单: codes\05\5.7\InstanceofTest.java

```
public class InstanceofTest
{
    public static void main(String[] args)
    {
        // 声明 hello 时使用 Object 类, 则 hello 的编译类型是 Object
        // Object 是所有类的父类, 但 hello 变量的实际类型是 String
        Object hello = "Hello";
        // String 与 Object 类存在继承关系, 可以进行 instanceof 运算。返回 true
        System.out.println("字符串是否是 Object 类的实例: "
            + (hello instanceof Object));
        System.out.println("字符串是否是 String 类的实例: "
            + (hello instanceof String)); // 返回 true
        // Math 与 Object 类存在继承关系, 可以进行 instanceof 运算。返回 false
        System.out.println("字符串是否是 Math 类的实例: "
            + (hello instanceof Math));
        // String 实现了 Comparable 接口, 所以返回 true
        System.out.println("字符串是否是 Comparable 接口的实例: "
            + (hello instanceof Comparable));
    }
}
```



```
String a = "Hello";
// String 类与 Math 类没有继承关系, 所以下面代码编译无法通过
System.out.println("字符串是否是 Math 类的实例: "
    + (a instanceof Math));
}
```

上面程序通过 `Object hello = "Hello";` 代码定义了一个 `hello` 变量, 这个变量的编译时类型是 `Object` 类, 但实际类型是 `String`。因为 `Object` 类是所有类、接口的父类, 因此可以执行 `hello instanceof String` 和 `hello instanceof Math` 等。

但如果使用 `String a = "Hello";` 代码定义的变量 `a`, 就不能执行 `a instanceof Math`, 因为 `a` 的编译类型是 `String`, `String` 类型既不是 `Math` 类型, 也不是 `Math` 类型的父类, 所以这行代码编译就会出错。

`instanceof` 运算符的作用是: 在进行强制类型转换之前, 首先判断前一个对象是否是后一个类的实例, 是否可以成功转换, 从而保证代码更加健壮。

`instanceof` 和 `(type)` 是 Java 提供的两个相关的运算符, 通常先用 `instanceof` 判断一个对象是否可以强制类型转换, 然后再使用 `(type)` 运算符进行强制类型转换, 从而保证程序不会出现错误。

5.8 继承与组合

继承是实现类复用的重要手段, 但继承带来了一个最大的坏处: 破坏封装。相比之下, 组合也是实现类复用的重要方式, 而采用组合方式来实现类复用则能提供更好的封装性。下面将详细介绍继承和组合之间的联系与区别。

➤➤ 5.8.1 使用继承的注意事项

子类扩展父类时, 子类可以从父类继承得到成员变量和方法, 如果访问权限允许, 子类可以直接访问父类的成员变量和方法, 相当于子类可以直接复用父类的成员变量和方法, 确实非常方便。

继承带来了高度复用的同时, 也带来了一个严重的问题: 继承严重地破坏了父类的封装性。前面介绍封装时提到: 每个类都应该封装它内部信息和实现细节, 而只暴露必要的方法给其他类使用。但在继承关系中, 子类可以直接访问父类的成员变量 (内部信息) 和方法, 从而造成子类和父类的严重耦合。

从这个角度来看, 父类的实现细节对子类不再透明, 子类可以访问父类的成员变量和方法, 并可以改变父类方法的实现细节 (例如, 通过方法重写的方式来改变父类的方法实现), 从而导致子类可以肆意篡改父类的方法。例如前面提到的 `Ostrich` 类, 它就重写了 `Bird` 类的 `fly()` 方法, 从而改变了 `fly()` 方法的实现细节。有如下代码:

```
Bird b = new Ostrich();
b.fly();
```

对于上面代码声明的 `Bird` 引用变量, 因为实际引用一个 `Ostrich` 对象, 所以调用 `b` 的 `fly()` 方法时执行的不再是 `Bird` 类提供的 `fly()` 方法, 而是 `Ostrich` 类重写后的 `fly()` 方法。

为了保证父类有良好的封装性, 不会被子类随意改变, 设计父类通常应该遵循如下规则。

- 尽量隐藏父类的内部数据。尽量把父类的所有成员变量都设置成 `private` 访问类型, 不要让子类直接访问父类的成员变量。
- 不要让子类可以随意访问、修改父类的方法。父类中那些仅为辅助其他的工具方法, 应该使用 `private` 访问控制符修饰, 让子类无法访问该方法; 如果父类中的方法需要被外部类调用, 则必须以 `public` 修饰, 但又不希望子类重写该方法, 可以使用 `final` 修饰符 (该修饰符后面会有更详细的介绍) 来修饰该方法; 如果希望父类的某个方法被子类重写, 但不希望被其他类自由访问, 则可以使用 `protected` 来修饰该方法。
- 尽量不要在父类构造器中调用将要被子类重写的方法。

看如下程序。

程序清单: codes\05\5.8\Sub.java

```
class Base
{
```

```
public Base()
{
    test();
}
public void test()           // ①号 test() 方法
{
    System.out.println("将被子类重写的方法");
}
}
public class Sub extends Base
{
    private String name;
    public void test()        // ②号 test() 方法
    {
        System.out.println("子类重写父类的方法, "
            + "其 name 字符串长度" + name.length());
    }
    public static void main(String[] args)
    {
        // 下面代码会引发空指针异常
        Sub s = new Sub();
    }
}
```

当系统试图创建 `Sub` 对象时, 同样会先执行其父类构造器, 如果父类构造器调用了被其子类重写的方法, 则变成调用被子类重写后的方法。当创建 `Sub` 对象时, 会先执行 `Base` 类中的 `Base` 构造器, 而 `Base` 构造器中调用了 `test()` 方法——并不是调用①号 `test()` 方法, 而是调用②号 `test()` 方法, 此时 `Sub` 对象的 `name` 实例变量是 `null`, 因此将引发空指针异常。

如果想把某些类设置成最终类, 即不能被当成父类, 则可以使用 `final` 修饰这个类, 例如 JDK 提供的 `java.lang.String` 类和 `java.lang.System` 类。除此之外, 使用 `private` 修饰这个类的所有构造器, 从而保证子类无法调用该类的构造器, 也就无法继承该类。对于把所有的构造器都使用 `private` 修饰的父类而言, 可另外提供一个静态方法, 用于创建该类的实例。

对很多初学者而言, 何时使用继承关系是一个难以把握的问题, 他们常常可能根据属性值的不同来派生子类。例如对于 `Animal` 类, 有的初学者可能派生出 `BigAnimal` 和 `SmallAnimal` 两个子类, 如果从一般到特殊的角度来看, 确实可以把 `BigAnimal` 和 `SmallAnimal` 两个类当成 `Animal` 类的子类。但从程序角度来看, 完全没有必要设计这样两个类: 主要在 `Animal` 类中增加一个 `size` 属性, 用于表示不同的 `Animal` 对象到底是 `BigAnimal` 还是 `SmallAnimal`, 完全没有必要重新派生出两个新类。

到底何时需要从父类派生新的子类呢? 不仅需要保证子类是一种特殊的父类, 而且需要具备以下两个条件之一。

- 子类需要额外增加属性, 而不仅仅是属性值的改变。例如从 `Person` 类派生出 `Student` 子类, `Person` 类里没有提供 `grade` (年级) 属性, 而 `Student` 类需要 `grade` 属性来保存 `Student` 对象就读的年级, 这种父类到子类的派生, 就符合 Java 继承的前提。
- 子类需要增加自己独特的行为方式 (包括增加新的方法或重写父类的方法)。例如从 `Person` 类派生出 `Teacher` 类, 其中 `Teacher` 类需要增加一个 `teaching()` 方法, 该方法用于描述 `Teacher` 对象独有的行为方式: 教学。

上面详细介绍了继承关系可能存在的问题, 以及如何处理这些问题。如果只是出于类复用的目的, 并不一定需要使用继承, 完全可以使用组合来实现。

5.8.2 利用组合实现复用

如果需要复用一个类, 除了把这个类当成基类来继承之外, 还可以把该类当成另一个类的组合成分, 从而允许新类直接复用该类的 `public` 方法。不管是继承还是组合, 都允许在新类 (对于继承就是子类) 中直接复用旧类的方法。

对于继承而言, 子类可以直接获得父类的 `public` 方法, 程序使用子类时, 将可以直接访问该子类从父类那里继承到的方法; 而组合则是把旧类对象作为新类的成员变量组合进来, 用以实现新类的功能,

用户看到的是新类的方法，而不能看到被组合对象的方法。因此，通常需要在新类里使用 **private** 修饰被组合的旧类对象。

仅从类复用的角度来看，不难发现父类的功能等同于被组合的类，都将自身的方法提供给新类使用；子类和组合关系里的整体类，都可复用原有类的方法，用于实现自身的功能。

假设有下面三个类：**Animal**、**Wolf** 和 **Bird**，它们之间有如图 5.19 所示的继承树。

图 5.19 所示三个类的代码如下。

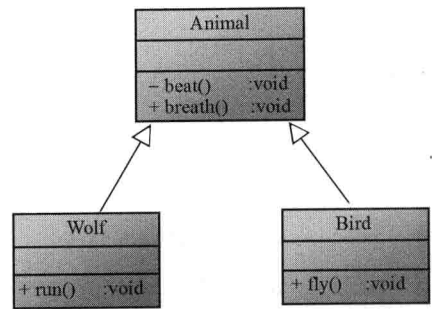


图 5.19 继承实现类复用

程序清单：codes\05\5.8\InheritTest.java

```

class Animal
{
    private void beat()
    {
        System.out.println("心脏跳动...");
    }
    public void breath()
    {
        beat();
        System.out.println("吸一口气，吐一口气，呼吸中...");
    }
}
// 继承 Animal，直接复用父类的 breath() 方法
class Bird extends Animal
{
    public void fly()
    {
        System.out.println("我在天空自在的飞翔...");
    }
}
// 继承 Animal，直接复用父类的 breath() 方法
class Wolf extends Animal
{
    public void run()
    {
        System.out.println("我在陆地上的快速奔跑...");
    }
}
public class InheritTest
{
    public static void main(String[] args)
    {
        Bird b = new Bird();
        b.breath();
        b.fly();
        Wolf w = new Wolf();
        w.breath();
        w.run();
    }
}
    
```

正如上面代码所示，通过让 **Bird** 和 **Wolf** 继承 **Animal**，从而允许 **Wolf** 和 **Bird** 获得 **Animal** 的方法，从而复用了 **Animal** 提供的 **breath()** 方法。通过这种方式，相当于让 **Wolf** 类和 **Bird** 类同时拥有其父类 **Animal** 的 **breath()** 方法，从而让 **Wolf** 对象和 **Bird** 对象都可直接复用 **Animal** 里定义的 **breath()** 方法。

如果仅仅从软件复用的角度来看，将上面三个类的定义改为如下形式也可实现相同的复用。

程序清单：codes\05\5.8\CompositeTest.java

```

class Animal
{
    private void beat()
    {
        System.out.println("心脏跳动...");
    }
    public void breath()
    
```



```

    {
        beat();
        System.out.println("吸一口气,吐一口气,呼吸中...");
    }
}
class Bird
{
    // 将原来的父类组合到原来的子类,作为子类的一个组合成分
    private Animal a;
    public Bird(Animal a)
    {
        this.a = a;
    }
    // 重新定义一个自己的 breath() 方法
    public void breath()
    {
        // 直接复用 Animal 提供的 breath() 方法来实现 Bird 的 breath() 方法
        a.breath();
    }
    public void fly()
    {
        System.out.println("我在天空自在的飞翔...");
    }
}
class Wolf
{
    // 将原来的父类组合到原来的子类,作为子类的一个组合成分
    private Animal a;
    public Wolf(Animal a)
    {
        this.a = a;
    }
    // 重新定义一个自己的 breath() 方法
    public void breath()
    {
        // 直接复用 Animal 提供的 breath() 方法来实现 Wolf 的 breath() 方法
        a.breath();
    }
    public void run()
    {
        System.out.println("我在陆地上的快速奔跑...");
    }
}
public class CompositeTest
{
    public static void main(String[] args)
    {
        // 此时需要显式创建被组合的对象
        Animal a1 = new Animal();
        Bird b = new Bird(a1);
        b.breath();
        b.fly();
        // 此时需要显式创建被组合的对象
        Animal a2 = new Animal();
        Wolf w = new Wolf(a2);
        w.breath();
        w.run();
    }
}

```

对于上面定义的两个类: `Animal`、`Wolf` 和 `Bird`, 它们对应的 UML 图如图 5.20 所示。

从图 5.20 中可以看出, 此时的 `Wolf` 对象和 `Bird` 对象由 `Animal` 对象组合而成, 因此上面程序中创建 `Wolf` 对象和 `Bird` 对象之前先创建 `Animal` 对象, 并利用这个 `Animal` 对象来创建 `Wolf` 对象和 `Bird` 对象。运行该程序时, 可以看到与前面程序相同的执行结果。

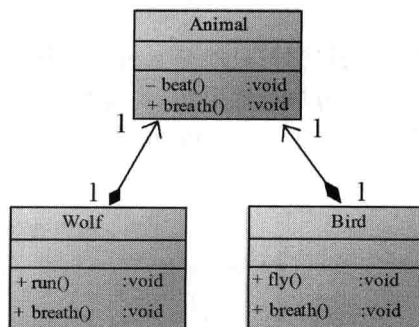
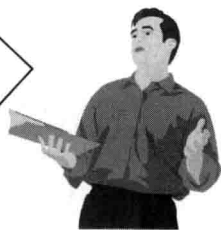


图 5.20 组合实现复用



学生提问：使用组合关系来实现复用，需要创建两个 Animal 对象，是不是意味着使用组合关系时系统开销更大？

答：不会。回忆前面介绍继承时所讲的内容，当创建一个子类对象时，系统不仅需要为子类定义的实例变量分配内存空间，而且需要为它的父类所定义的实例变量分配内存空间。如果采用继承的设计方式，假设父类定义了2个实例变量，子类定义了3个实例变量，当创建子类实例时，系统需要为子类实例分配5块内存空间；如果采用组合的设计方式，先创建被嵌入类实例，此时需要分配2块内存空间，再创建整体类实例，也需要分配3块内存空间，只是需要多一个引用变量来引用被嵌入的对象。通过这个分析来看，继承设计与组合设计的系统开销不会有本质的差别。



大部分时候，继承关系中从多个子类里抽象出共有父类的过程，类似于组合关系中从多个整体类里提取被组合类的过程；继承关系中从父类派生子类的过程，则类似于组合关系中把被组合类组合到整体类的过程。

到底该用继承？还是该用组合呢？继承是对已有的类做一番改造，以此获得一个特殊的版本。简而言之，就是将一个较为抽象的类改造成能适用于某些特定需求的类。因此，对于上面的 Wolf 和 Animal 的关系，使用继承更能表达其现实意义。用一个动物来合成一匹狼毫无意义：狼并不是由动物组成的。反之，如果两个类之间有明确的整体、部分的关系，例如 Person 类需要复用 Arm 类的方法（Person 对象由 Arm 对象组合而成），此时就应该采用组合关系来实现复用，把 Arm 作为 Person 类的组合成员变量，借助于 Arm 的方法来实现 Person 的方法，这是一个不错的选择。

总之，继承要表达的是一种“是 (is-a)”的关系，而组合表达的是“有 (has-a)”的关系。

5.9 初始化块

Java 使用构造器来对单个对象进行初始化操作，使用构造器先完成整个 Java 对象的状态初始化，然后将 Java 对象返回给程序，从而让该 Java 对象的信息更加完整。与构造器作用非常类似的是初始化块，它也可以对 Java 对象进行初始化操作。

5.9.1 使用初始化块

初始化块是 Java 类里可出现的第 4 种成员（前面依次有成员变量、方法和构造器），一个类里可以有多个初始化块，相同类型的初始化块之间有顺序：前面定义的初始化块先执行，后面定义的初始化块后执行。初始化块的语法格式如下：

```
[修饰符] {
    // 初始化块的可执行性代码
    ...
}
```

初始化块的修饰符只能是 static，使用 static 修饰的初始化块被称为静态初始化块。初始化块里的代码可以包含任何可执行性语句，包括定义局部变量、调用其他对象的方法，以及使用分支、循环语句等。

下面程序定义了一个 Person 类，它既包含了构造器，也包含了初始化块。下面看看在程序中创建 Person 对象时发生了什么。

程序清单：code\05\5.9\Person.java

```
public class Person
{
    // 下面定义一个初始化块
    {
        int a = 6;
```

```
        if (a > 4)
        {
            System.out.println("Person 初始化块: 局部变量 a 的值大于 4");
        }
        System.out.println("Person 的初始化块");
    }
    // 定义第二个初始化块
    {
        System.out.println("Person 的第二个初始化块");
    }
    // 定义无参数的构造器
    public Person()
    {
        System.out.println("Person 类的无参数构造器");
    }
    public static void main(String[] args)
    {
        new Person();
    }
}
```

上面程序的 main() 方法只创建了一个 Person 对象, 程序的输出如下:

```
Person 初始化块: 局部变量 a 的值大于 4
Person 的初始化块
Person 的第二个初始化块
Person 类的无参数构造器
```

从运行结果可以看出, 当创建 Java 对象时, 系统总是先调用该类里定义的初始化块, 如果一个类里定义了 2 个普通初始化块, 则前面定义的初始化块先执行, 后面定义的初始化块后执行。

初始化块虽然也是 Java 类的一种成员, 但它没有名字, 也就没有标识, 因此无法通过类、对象来调用初始化块。初始化块只在创建 Java 对象时隐式执行, 而且在执行构造器之前执行。

✱ 注意: ✱

虽然 Java 允许一个类里定义 2 个普通初始化块, 但这没有任何意义。因为初始化块是在创建 Java 对象时隐式执行的, 而且它们总是全部执行, 因此完全可以把多个普通初始化块合并成一个初始化块, 从而可以让程序更加简洁, 可读性更强。



从上面代码可以看出, 初始化块和构造器的作用非常相似, 它们都用于对 Java 对象执行指定的初始化操作, 但它们之间依然存在一些差异, 下面具体分析初始化块和构造器之间的差异。

普通初始化块、声明实例变量指定的默认值都可认为是对象的初始化代码, 它们的执行顺序与源程序中的排列顺序相同。看如下代码。

程序清单: codes\05\5.9\InstanceInitTest.java

```
public class InstanceInitTest
{
    // 先执行初始化块将 a 实例变量赋值为 6
    {
        a = 6;
    }
    // 再执行将 a 实例变量赋值为 9
    int a = 9;
    public static void main(String[] args)
    {
        // 下面代码将输出 9
        System.out.println(new InstanceInitTest().a);
    }
}
```

上面程序中定义了两次对 a 实例变量赋值, 执行结果是 a 实例变量的值为 9, 这表明 **int a = 9** 这行代码比初始化块后执行。但如果将粗体字初始化块代码与 **int a = 9;** 的顺序调换一下, 将可以看到程序输

出 InstanceInitTest 的实例变量 a 的值为 6，这是由于初始化块中代码再次将 a 实例变量的值设为 6。



注意：

当 Java 创建一个对象时，系统先为该对象的所有实例变量分配内存（前提是该类已经被加载过了），接着程序开始对这些实例变量执行初始化，其初始化顺序是：先执行初始化块或声明实例变量时指定的初始值（这两个地方指定初始值的执行允许与它们在源代码中的排列顺序相同），再执行构造器里指定的初始值。



5.9.2 初始化块和构造器

从某种程度上来看，初始化块是构造器的补充，初始化块总是在构造器执行之前执行。系统同样可以使用初始化块来进行对象的初始化操作。

与构造器不同的是，初始化块是一段固定执行的代码，它不能接收任何参数。因此初始化块对同一个类的所有对象所进行的初始化处理完全相同。基于这个原因，不难发现初始化块的基本用法，如果有一段初始化处理代码对所有对象完全相同，且无须接收任何参数，就可以把这段初始化处理代码提取到初始化块中。图 5.21 显示了把两个构造器中的代码提取成初始化块示意图。

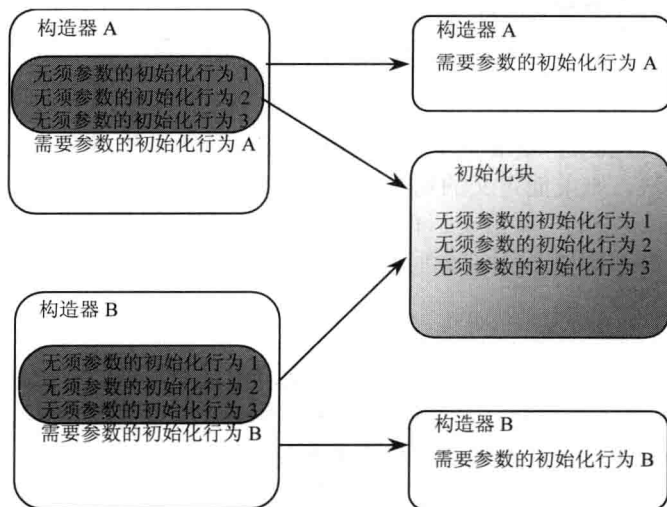


图 5.21 将构造器代码块提取成初始化块

从图 5.21 中可以看出，如果两个构造器中有相同的初始化代码，且这些初始化代码无须接收参数，就可以把它们放在初始化块中定义。通过把多个构造器中的相同代码提取到初始化块中定义，能更好地提高初始化代码的复用，提高整个应用的可维护性。



注意：

实际上初始化块是一个假象，使用 javac 命令编译 Java 类后，该 Java 类中的初始化块会消失——初始化块中代码会被“还原”到每个构造器中，且位于构造器所有代码的前面。



与构造器类似，创建一个 Java 对象时，不仅会执行该类的普通初始化块和构造器，而且系统会一直上溯到 java.lang.Object 类，先执行 java.lang.Object 类的初始化块，开始执行 java.lang.Object 的构造器，依次向下执行其父类的初始化块，开始执行其父类的构造器……最后才执行该类的初始化块和构造器，返回该类的对象。

除此之外，如果希望类加载后对整个类进行某些初始化操作，例如当 Person 类加载后，则需要把 Person 类的 eyeNumber 类变量初始化为 2，此时需要使用 static 关键字来修饰初始化块，使用 static 修饰的初始化块被称为静态初始化块。

5.9.3 静态初始化块

如果定义初始化块时使用了 `static` 修饰符, 则这个初始化块就变成了静态初始化块, 也被称为类初始化块(普通初始化块负责对对象执行初始化, 类初始化块则负责对类进行初始化)。静态初始化块是类相关的, 系统将在类初始化阶段执行静态初始化块, 而不是在创建对象时才执行。因此静态初始化块总是比普通初始化块先执行。

静态初始化块是类相关的, 用于对整个类进行初始化处理, 通常用于对类变量执行初始化处理。静态初始化块不能对实例变量进行初始化处理。



注意:

静态初始化块也被称为类初始化块, 也属于类的静态成员, 同样需要遵循静态成员不能访问非静态成员的规则, 因此静态初始化块不能访问非静态成员, 包括不能访问实例变量和实例方法。



与普通初始化块类似的是, 系统在类初始化阶段执行静态初始化块时, 不仅会执行本类的静态初始化块, 而且还会一直上溯到 `java.lang.Object` 类(如果它包含静态初始化块), 先执行 `java.lang.Object` 类的静态初始化块(如果有), 然后执行其父类的静态初始化块……最后才执行该类的静态初始化块, 经过这个过程, 才完成了该类的初始化过程。只有当类初始化完成后, 才可以在系统中使用这个类, 包括访问这个类的类方法、类变量或者用这个类来创建实例。

下面程序创建三个类: `Root`、`Mid` 和 `Leaf`, 这三个类都提供了静态初始化块和普通初始化块, 而且 `Mid` 类里还使用 `this` 调用重载的构造器, 而 `Leaf` 使用 `super` 显式调用其父类指定的构造器。

程序清单: `codes\05\5.9\Test.java`

```
class Root
{
    static{
        System.out.println("Root 的静态初始化块");
    }
    {
        System.out.println("Root 的普通初始化块");
    }
    public Root()
    {
        System.out.println("Root 的无参数的构造器");
    }
}
class Mid extends Root
{
    static{
        System.out.println("Mid 的静态初始化块");
    }
    {
        System.out.println("Mid 的普通初始化块");
    }
    public Mid()
    {
        System.out.println("Mid 的无参数的构造器");
    }
    public Mid(String msg)
    {
        // 通过 this 调用同一类中重载的构造器
        this();
        System.out.println("Mid 的带参数构造器, 其参数值: "
            + msg);
    }
}
class Leaf extends Mid
{
    static{
```



```

        System.out.println("Leaf 的静态初始化块");
    }
    {
        System.out.println("Leaf 的普通初始化块");
    }
    public Leaf()
    {
        // 通过 super 调用父类中有一个字符串参数的构造器
        super("疯狂 Java 讲义");
        System.out.println("执行 Leaf 的构造器");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        new Leaf();
        new Leaf();
    }
}

```

上面定义了三类，其继承树如图 5.22 所示。

在上面主程序中两次执行 new Leaf(); 代码，创建两个 Leaf 对象，将可看到如图 5.23 所示的输出。

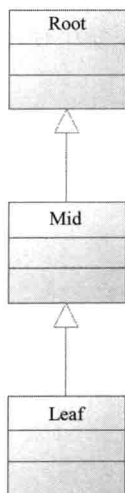


图 5.22 继承结构

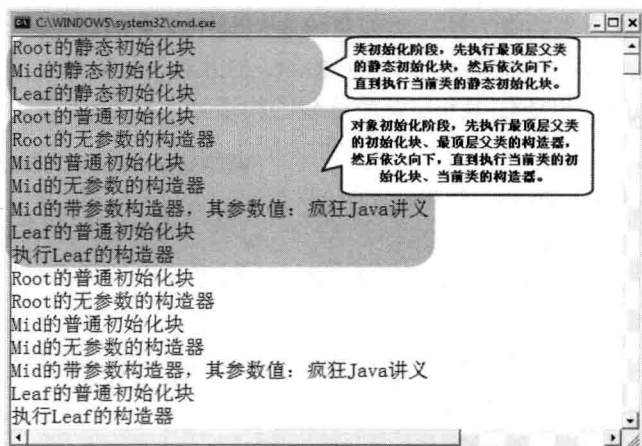


图 5.23 创建 Leaf 对象的执行过程

从图 5.23 来看，第一次创建一个 Leaf 对象时，因为系统中还不存在 Leaf 类，因此需要先加载并初始化 Leaf 类，初始化 Leaf 类时会先执行其顶层父类的静态初始化块，再执行其直接父类的静态初始化块，最后才执行 Leaf 本身的静态初始化块。

一旦 Leaf 类初始化成功后，Leaf 类在该虚拟机里将一直存在，因此当第二次创建 Leaf 实例时无须再次对 Leaf 类进行初始化。

普通初始化块和构造器的执行顺序与前面介绍的一致，每次创建一个 Leaf 对象时，都需要先执行最顶层父类的初始化块、构造器，然后执行其父类的初始化块、构造器……最后才执行 Leaf 类的初始化块和构造器。



注意：

Java 系统加载并初始化某个类时，总是保证该类的所有父类（包括直接父类和间接父类）全部加载并初始化。关于类初始化的知识可参阅本书 18.1 节的介绍。



静态初始化块和声明静态成员变量时所指定的初始值都是该类的初始化代码，它们的执行顺序与源程序中的排列顺序相同。看如下代码。

程序清单: codes\05\5.9\StaticInitTest.java

```
public class StaticInitTest
{
    // 先执行静态初始化块将 a 静态成员变量赋值为 6
    static{
        a = 6;
    }
    // 再将 a 静态成员变量赋值为 9
    static int a = 9;
    public static void main(String[] args)
    {
        // 下面代码将输出 9
        System.out.println(StaticInitTest.a);
    }
}
```

上面程序中定义了两次对 `a` 静态成员变量进行赋值, 执行结果是 `a` 值为 9, 这表明 `static int a = 9;` 这行代码位于静态初始化块之后执行。如果将上面程序中粗体字静态初始化块与 `static int a = 9;` 调换顺序, 将可以看到程序输出 6, 这是由于静态初始化块中代码再次将 `a` 的值设为 6。

**提示:**

当 JVM 第一次主动使用某个类时, 系统会在类准备阶段为该类的所有静态成员变量分配内存; 在初始化阶段则负责初始化这些静态成员变量, 初始化静态成员变量就是执行类初始化代码或者声明类成员变量时指定的初始值, 它们的执行顺序与源代码中的排列顺序相同。



5.10 本章小结

本章主要介绍了 Java 面向对象的基本知识, 包括如何定义类, 如何为类定义成员变量、方法, 以及如何创建类的对象。本章还深入分析了对象和引用变量之间的关系。方法也是本章介绍的重点, 本章详细介绍了方法的参数传递机制、递归方法、重载方法、可变长度形参的方法等内容, 并详细对比了成员变量和局部变量在用法上的差别, 并深入对比了成员变量和局部变量在运行机制上的差别。

本章详细讲解了如何使用访问控制符来设计封装良好的类, 并使用 `package` 语句来组合系统中大量的类, 以及如何使用 `import` 语句来导入其他包中的类。

本章着重讲解了 Java 的继承和多态, 包括如何利用 `extends` 关键字来实现继承, 以及把一个子类对象赋给父类变量时产生的多态行为。本章还深入比较了继承、组合两种类复用机制各自的优缺点和适用场景。

►► 本章练习

1. 编写一个学生类, 提供 `name`、`age`、`gender`、`phone`、`address`、`email` 成员变量, 且为每个成员变量提供 `setter`、`getter` 方法。为学生类提供默认的构造器和带所有成员变量的构造器。为学生类提供方法, 用于描绘吃、喝、玩、睡等行为。

2. 利用第 1 题定义的 `Student` 类, 定义一个 `Student[]` 数组保存多个 `Student` 对象作为通讯录数据。程序可通过 `name`、`email`、`address` 查询, 如果找不到数据, 则进行友好提示。

3. 定义普通人、老师、班主任、学生、学校这些类, 提供适当的成员变量、方法用于描述其内部数据和行为特征, 并提供主类使之运行。要求有良好的封装性, 将不同类放在不同的包下面, 增加文档注释, 生成 API 文档。

4. 改写第 1 题的程序, 利用组合来实现类复用。

5. 定义交通工具、汽车、火车、飞机这些类, 注意它们的继承关系, 为这些类提供超过 3 个不同的构造器, 并通过初始化块提取构造器中的通用代码。

第6章

面向对象（下）

本章要点

- ✎ 包装类及其用法
- ✎ toString 方法的用法
- ✎ ==和 equals 的区别
- ✎ static 关键字的用法
- ✎ 实现单例类
- ✎ final 关键字的用法
- ✎ 不可变类和可变类
- ✎ 缓存实例的不可变类
- ✎ abstract 关键字的用法
- ✎ 实现模板模式
- ✎ 接口的概念和作用
- ✎ 定义接口的语法
- ✎ 实现接口
- ✎ 接口和抽象类的联系与区别
- ✎ 面向接口编程的优势
- ✎ 内部类的概念和定义语法
- ✎ 非静态内部类和静态内部类
- ✎ 创建内部类的对象
- ✎ 扩展内部类
- ✎ 匿名内部类和局部内部类
- ✎ Lambda 表达式与函数式接口
- ✎ 方法引用和构造器引用
- ✎ 枚举类概念和作用
- ✎ 手动实现枚举类
- ✎ JDK 1.5 提供的枚举类
- ✎ 枚举类的成员变量、方法和构造器
- ✎ 实现接口的枚举类
- ✎ 包含抽象方法的枚举类
- ✎ 垃圾回收和对象的 finalize 方法
- ✎ 强制垃圾回收的方法
- ✎ 对象的软、弱和虚引用
- ✎ JAR 文件的用途
- ✎ 使用 jar 命令创建 JAR 文件

除了前一章所介绍的关于类、对象的基本语法之外，本章将会继续介绍 Java 面向对象的特性。Java 为 8 个基本类型提供了对应的包装类，通过这些包装类可以把 8 个基本类型的值包装成对象使用，JDK 1.5 提供了自动装箱和自动拆箱功能，允许把基本类型值直接赋给对应的包装类引用变量，也允许把包装类对象直接赋给对应的基本类型变量。

Java 提供了 `final` 关键字来修饰变量、方法和类，系统不允许为 `final` 变量重新赋值，子类不允许覆盖父类的 `final` 方法，`final` 类不能派生子类。通过使用 `final` 关键字，允许 Java 实现不可变类，不可变类会让系统更加安全。

`abstract` 和 `interface` 两个关键字分别用于定义抽象类和接口，抽象类和接口都是从多个子类中抽象出来的共同特征。但抽象类主要作为多个类的模板，而接口则定义多类应该遵守的规范。Lambda 表达式是 Java 8 的重要更新，本章将会详细介绍 Lambda 表达式的相关内容。`enum` 关键字用于创建枚举类，枚举类是一种不能自由创建对象的类，枚举类的对象在定义类时已经固定下来。枚举类特别适合定义像行星、季节这样的类，它们能创建的实例是有限且确定的。

本章将进一步介绍对象在内存中的运行机制，并深入介绍对象的几种引用方式，以及垃圾回收机制如何处理具有不同引用的对象，并详细介绍如何使用 `jar` 命令来创建 JAR 包。

6.1 Java 8 增强的包装类

Java 是面向对象的编程语言，但它也包含了 8 种基本数据类型，这 8 种基本数据类型不支持面向对象的编程机制，基本数据类型的数据也不具备“对象”的特性：没有成员变量、方法可以被调用。Java 之所以提供这 8 种基本数据类型，主要是为了照顾程序员的传统习惯。

这 8 种基本数据类型带来了一定的方便性，例如可以进行简单、有效的常规数据处理。但在某些时候，基本数据类型会有一些制约，例如所有引用类型的变量都继承了 `Object` 类，都可当成 `Object` 类型变量使用。但基本数据类型的变量就不可以，如果有个方法需要 `Object` 类型的参数，但实际需要的值却是 2、3 等数值，这可能就比较难以处理。

为了解决 8 种基本数据类型的变量不能当成 `Object` 类型变量使用的问题，Java 提供了包装类（Wrapper Class）的概念，为 8 种基本数据类型分别定义了相应的引用类型，并称之为基本数据类型的包装类。

表 6.1 基本数据类型和包装类的对应关系

基本数据类型	包 装 类
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean

从表 6.1 可以看出，除了 `int` 和 `char` 有点例外之外，其他的基本数据类型对应的包装类都是将其首字母大写即可。

在 JDK 1.5 以前，把基本数据类型变量变成包装类实例需要通过对应包装类的构造器来实现，不仅如此，8 个包装类中除了 `Character` 之外，还可以通过传入一个字符串参数来构建包装类对象。

在 JDK 1.5 以前，如果希望获得包装类对象中包装的基本类型变量，则可以使用包装类提供的 `xxxValue()` 实例方法。由于这种用法已经过时，故此处不再给出示例代码。

通过上面介绍不难看出，基本类型变量和包装类对象之间的转换关系如图 6.1 所示。

从图 6.1 中可以看出，Java 提供的基本类型变量和包装类对象之间的转换有点烦琐，但从 JDK 1.5 之后这种烦琐就消除了，JDK 1.5 提供了自动装箱（Autoboxing）和自动拆箱（AutoUnboxing）功能。

所谓自动装箱，就是可以把一个基本类型变量直接赋给对应的包装类变量，或者赋给 Object 变量（Object 是所有类的父类，子类对象可以直接赋给父类变量）；自动拆箱则与之相反，允许直接把包装类对象直接赋给一个对应的基本类型变量。

下面程序示范了自动装箱和自动拆箱的用法。

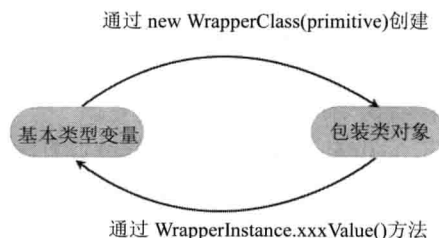


图 6.1 JDK 1.5 以前基本类型变量与包装类实例之间的转换

程序清单：codes\06\6.1\AutoBoxingUnboxing.java

```

public class AutoBoxingUnboxing
{
    public static void main(String[] args)
    {
        // 直接把一个基本类型变量赋给 Integer 对象
        Integer inObj = 5;
        // 直接把一个 boolean 类型变量赋给一个 Object 类型的变量
        Object boolObj = true;
        // 直接把一个 Integer 对象赋给 int 类型的变量
        int it = inObj;
        if (boolObj instanceof Boolean)
        {
            // 先把 Object 对象强制类型转换为 Boolean 类型，再赋给 boolean 变量
            boolean b = (Boolean)boolObj;
            System.out.println(b);
        }
    }
}
    
```

当 JDK 提供了自动装箱和自动拆箱功能后，大大简化了基本类型变量和包装类对象之间的转换过程。值得指出的是，进行自动装箱和自动拆箱时必须注意类型匹配，例如 Integer 只能自动拆箱成 int 类型变量，不要试图拆箱成 boolean 类型变量；与之类似的是，int 类型变量只能自动装箱成 Integer 对象（即使赋给 Object 类型变量，那也只是利用了 Java 的向上自动转型特性），不要试图装箱成 Boolean 对象。

借助于包装类的帮助，再加上 JDK 1.5 提供的自动装箱、自动拆箱功能，开发者可以把基本类型的变量“近似”地当成对象使用（所有装箱、拆箱过程都由系统自动完成，无须程序员理会）；反过来，开发者也可以把包装类的实例近似地当成基本类型的变量使用。

除此之外，包装类还可实现基本类型变量和字符串之间的转换。把字符串类型的值转换为基本类型的值有两种方式。

- 利用包装类提供的 parseXxx(String s) 静态方法（除了 Character 之外的所有包装类都提供了该方法。
- 利用包装类提供的 Xxx(String s) 构造器。

String 类提供了多个重载 valueOf() 方法，用于将基本类型变量转换成字符串，下面程序示范了这种转换关系。

程序清单：codes\06\6.1\Primitive2String.java

```

public class Primitive2String
{
    public static void main(String[] args)
    {
        String intStr = "123";
        // 把一个特定字符串转换成 int 变量
        int it1 = Integer.parseInt(intStr);
        int it2 = new Integer(intStr);
        System.out.println(it2);
        String floatStr = "4.56";
        // 把一个特定字符串转换成 float 变量
        float ft1 = Float.parseFloat(floatStr);
    }
}
    
```

```

float ft2 = new Float(floatStr);
System.out.println(ft2);
// 把一个 float 变量转换成 String 变量
String ftStr = String.valueOf(2.345f);
System.out.println(ftStr);
// 把一个 double 变量转换成 String 变量
String dbStr = String.valueOf(3.344);
System.out.println(dbStr);
// 把一个 boolean 变量转换成 String 变量
String boolStr = String.valueOf(true);
System.out.println(boolStr.toUpperCase());
}
}

```

通过上面程序可以看出基本类型变量和字符串之间的转换关系,如图 6.2 所示。

如果希望把基本类型变量转换成字符串,还有一种更简单的方法:将基本类型变量和""进行连接运算,系统会自动把基本类型变量转换成字符串。例如下面代码:

```

// intStr 的值为"5"
String intStr = 5 + "";

```

此处要指出的是,虽然包装类型的变量是引用数据类型,但包装类的实例可以与数值类型的值进行比较,这种比较是直接取出包装类实例所包装的数值来进行比较的。

看下面代码。

程序清单: codes\06\6.1\WrapperClassCompare.java

```

Integer a = new Integer(6);
// 输出 true
System.out.println("6 的包装类实例是否大于 5.0" + (a > 5.0));

```

两个包装类的实例进行比较的情况就比较复杂,因为包装类的实例实际上是引用类型,只有两个包装类引用指向同一个对象时才会返回 true。下面代码示范了这种效果(程序清单同上)。

```

System.out.println("比较 2 个包装类的实例是否相等: "
    + (new Integer(2) == new Integer(2))); //输出 false

```

但 JDK 1.5 以后支持所谓的自动装箱,自动装箱就是可以直接把一个基本类型值赋给一个包装类实例,在这种情况下可能会出现一些特别的情形。看如下代码(程序清单同上)。

```

// 通过自动装箱,允许把基本类型值赋值给包装类实例
Integer ina = 2;
Integer inb = 2;
System.out.println("两个 2 自动装箱后是否相等: " + (ina == inb)); // 输出 true
Integer biga = 128;
Integer bigb = 128;
System.out.println("两个 128 自动装箱后是否相等: " + (biga == bigb)); //输出 false

```

上面程序让人比较费解:同样是两个 int 类型的数值自动装箱成 Integer 实例后,如果是两个 2 自动装箱后就相等;但如果是两个 128 自动装箱后就不相等,这是为什么呢?这与 Java 的 Integer 类的设计有关,查看 Java 系统中 java.lang.Integer 类的源代码,如下所示。

```

// 定义一个长度为 256 的 Integer 数组
static final Integer[] cache = new Integer[-(-128) + 127 + 1];
static {
    // 执行初始化,创建-128 到 127 的 Integer 实例,并放入 cache 数组中
    for(int i = 0; i < cache.length; i++)
        cache[i] = new Integer(i - 128);
}

```

从上面代码可以看出,系统把一个-128~127 之间的整数自动装箱成 Integer 实例,并放入了一个名为 cache 的数组中缓存起来。如果以后把一个-128~127 之间的整数自动装箱成一个 Integer 实例时,实

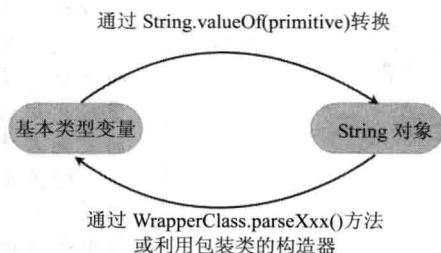


图 6.2 基本类型变量和字符串之间的转换关系

际上是直接指向对应的数组元素，因此-128~127之间的同一个整数自动装箱成 Integer 实例时，永远都是引用 cache 数组的同一个数组元素，所以它们全部相等；但每次把一个不在-128~127范围内的整数自动装箱成 Integer 实例时，系统总是重新创建一个 Integer 实例，所以出现程序中的运行结果。



学生提问：Java 为什么要对这些数据进行缓存呢？

答：缓存是一种非常优秀的设计模式，在 Java、Java EE 平台的很多地方都会通过缓存来提高系统的运行性能。简单地说，如果你需要一台电脑，那么你就去买了一台电脑。但你不可能一直使用这台电脑，你总会离开这台电脑——在你离开电脑的这段时间内，你如何做？你会不会立即把电脑扔掉？当然不会，你会把电脑放在房间里，等下次又需要电脑时直接开机使用，而不是再次去购买一台。假设电脑是内存中的对象，而你的房间是内存，如果房间足够大，则可以把所有曾经用过的各种东西都缓存起来，但这不可能，房间的空间是有限制的，因此有些东西你用过一次就扔掉了。你只会把一些购买成本大、需要频繁使用的东西保存下来。类似地，Java 也把一些创建成本大、需要频繁使用的对象缓存起来，从而提高程序的运行性能。



Java 7 增强了包装类的功能，Java 7 为所有的包装类都提供了一个静态的 `compare(xxx val1, xxx val2)` 方法，这样开发者就可以通过包装类提供的 `compare(xxx val1, xxx val2)` 方法来比较两个基本类型值的大小，包括比较两个 boolean 类型值，两个 boolean 类型值进行比较时，`true > false`。例如如下代码：

```
System.out.println(Boolean.compare(true, false)); // 输出 1
System.out.println(Boolean.compare(true, true)); // 输出 0
System.out.println(Boolean.compare(false, true)); // 输出 -1
```

不仅如此，Java 7 还为 Character 包装类增加了大量的工具方法来对一个字符进行判断。关于 Character 中可用的方法请参考 Character 的 API 文档。

Java 8 再次增强了这些包装类的功能，其中一个重要的增强就是支持无符号算术运算。Java 8 为整型包装类增加了支持无符号运算的方法。Java 8 为 Integer、Long 增加了如下方法。

- `static String toUnsignedString(int/long i)`：该方法将指定 int 或 long 型整数转换为无符号整数对应的字符串。
- `static String toUnsignedString(int i/long, int radix)`：该方法将指定 int 或 long 型整数转换为指定进制的无符号整数对应的字符串。
- `static xxx parseUnsignedXxx(String s)`：该方法将指定字符串解析成无符号整数。当调用类为 Integer 时，xxx 代表 int；当调用类是 Long 时，xxx 代表 long。
- `static xxx parseUnsignedXxx(String s, int radix)`：该方法将指定字符串按指定进制解析成无符号整数。当调用类为 Integer 时，xxx 代表 int；当调用类是 Long 时，xxx 代表 long。
- `static int compareUnsigned(xxx x, xxx y)`：该方法将 x、y 两个整数转换为无符号整数后比较大小。当调用类为 Integer 时，xxx 代表 int；当调用类是 Long 时，xxx 代表 long。
- `static long divideUnsigned(long dividend, long divisor)`：该方法将 x、y 两个整数转换为无符号整数后计算它们相除的商。当调用类为 Integer 时，xxx 代表 int；当调用类是 Long 时，xxx 代表 long。
- `static long remainderUnsigned(long dividend, long divisor)`：该方法将 x、y 两个整数转换为无符号整数后计算它们相除的余数。当调用类为 Integer 时，xxx 代表 int；当调用类是 Long 时，xxx 代表 long。

Java 8 还为 Byte、Short 增加了 `toUnsignedInt(xxx x)`、`toUnsignedLong(yyy x)` 两个方法，这两个方法用于将指定 byte 或 short 类型的变量或值转换成无符号的 int 或 long 值。

下面程序示范了这些包装类的无符号算术运算功能。

程序清单: codes\06\6.1\UnsignedTest.java

```
public class UnsignedTest
{
    public static void main(String[] args)
    {
        byte b = -3;
        // 将byte类型的-3转换为无符号整数
        System.out.println("byte类型的-3对应的无符号整数: "
            + Byte.toUnsignedInt(b)); // 输出 253
        // 指定使用十六进制解析无符号整数
        int val = Integer.parseUnsignedInt("ab", 16);
        System.out.println(val); // 输出 171
        // 将-12转换为无符号int型, 然后转换为十六进制的字符串
        System.out.println(Integer.toUnsignedString(-12, 16)); // 输出 ffffffff4
        // 将两个数转换为无符号整数后相除
        System.out.println(Integer.divideUnsigned(-2, 3));
        // 将两个数转换为无符号整数相除后求余
        System.out.println(Integer.remainderUnsigned(-2, 7));
    }
}
```

无符号整数最大的特点是最高位不再被当成符号位, 因此无符号整数不支持负数, 其最小值为 0。上面程序的运算结果可能不太直观。理解该程序的关键是先把操作数转换为无符号整数, 然后再进行计算。以 byte 类型的 -3 为例, 其原码为 10000011 (最高位 1 代表负数), 其反码为 11111100, 补码为 11111101, 如果将该数当成无符号整数处理, 那么最高位的 1 就不再是符号位, 也是数值位, 该数就对应为 253, 即上面程序的输出结果。读者只要先将上面表达式中的操作数转换为无符号整数, 然后再进行运算, 即可得到程序的输出结果。

6.2 处理对象

Java 对象都是 Object 类的实例, 都可直接调用该类中定义的方法, 这些方法提供了处理 Java 对象的通用方法。

➤➤ 6.2.1 打印对象和 toString 方法

先看下面程序。

程序清单: codes\06\6.2\PrintObject.java

```
class Person
{
    private String name;
    public Person(String name)
    {
        this.name = name;
    }
}
public class PrintObject
{
    public static void main(String[] args)
    {
        // 创建一个 Person 对象, 将之赋给 p 变量
        Person p = new Person("孙悟空");
        // 打印 p 所引用的 Person 对象
        System.out.println(p);
    }
}
```

上面程序创建了一个 Person 对象, 然后使用 System.out.println() 方法输出 Person 对象。编译、运行上面程序, 看到如下运行结果:

```
Person@15db9742
```


当读者运行上面程序时，可能看到不同的输出结果：@符号后的8位十六进制数字可能发生改变。但这个输出结果是怎么来的呢？System.out的println()方法只能在控制台输出字符串，而Person实例是一个内存中的对象，怎么能直接转换为字符串输出呢？当使用该方法输出Person对象时，实际上输出的是Person对象的toString()方法的返回值。也就是说，下面两行代码的效果完全一样。

```
System.out.println(p);
System.out.println(p.toString());
```

toString()方法是Object类里的一个实例方法，所有的Java类都是Object类的子类，因此所有的Java对象都具有toString()方法。

不仅如此，所有的Java对象都可以和字符串进行连接运算，当Java对象和字符串进行连接运算时，系统自动调用Java对象toString()方法的返回值和字符串进行连接运算，即下面两行代码的结果也完全相同。

```
String pStr = p + "";
String pStr = p.toString() + "";
```

toString()方法是一个非常特殊的方法，它是一个“自我描述”方法，该方法通常用于实现这样一个功能：当程序员直接打印该对象时，系统将会输出该对象的“自我描述”信息，用以告诉外界该对象具有的状态信息。

Object类提供的toString()方法总是返回该对象实现类的“类名+@+hashCode”值，这个返回值并不能真正实现“自我描述”的功能，因此如果用户需要自定义类能实现“自我描述”的功能，就必须重写Object类的toString()方法。例如下面程序。

程序清单：codes\06\6.2\ToStringTest.java

```
class Apple
{
    private String color;
    private double weight;
    public Apple(){ }
    // 提供有参数的构造器
    public Apple(String color , double weight)
    {
        this.color = color;
        this.weight = weight;
    }
    // 省略 color、weight 的 setter 和 getter 方法
    ...
    // 重写 toString() 方法，用于实现 Apple 对象的“自我描述”
    public String toString()
    {
        return "一个苹果，颜色是：" + color
            + "，重量是：" + weight;
    }
}

public class ToStringTest
{
    public static void main(String[] args)
    {
        Apple a = new Apple("红色" , 5.68);
        // 打印 Apple 对象
        System.out.println(a);
    }
}
```

编译、运行上面程序，看到如下运行结果：

```
一个苹果，颜色是：红色，重量是：5.68
```

从上面运行结果可以看出，通过重写Apple类的toString()方法，就可以让系统在打印Apple对象时打印出该对象的“自我描述”信息。

大部分时候，重写toString()方法总是返回该对象的所有令人感兴趣的信息所组成的字符串。通常可返回如下格式的字符串：

```
类名[field1=值1, field2=值2, ...]
```

因此, 可以将上面 Apple 类的 toString()方法改为如下:

```
public String toString()
{
    return "Apple[color=" + color + ",weight=" + weight + "];";
}
```

这个 toString()方法提供了足够的有效信息来描述 Apple 对象, 也就实现了 toString()方法的功能。

6.2.2 ==和 equals 方法

Java 程序中测试两个变量是否相等有两种方式: 一种是利用==运算符, 另一种是利用 equals()方法。当使用==来判断两个变量是否相等时, 如果两个变量是基本类型变量, 且都是数值类型(不一定要要求数据类型严格相同), 则只要两个变量的值相等, 就将返回 true。

但对于两个引用类型变量, 只有它们指向同一个对象时, ==判断才会返回 true。==不可用于比较类型上没有父子关系的两个对象。下面程序示范了使用==来判断两种类型变量是否相等的结果。

程序清单: codes\06\6.2\EqualTest.java

```
public class EqualTest
{
    public static void main(String[] args)
    {
        int it = 65;
        float fl = 65.0f;
        // 将输出 true
        System.out.println("65 和 65.0f 是否相等? " + (it == fl));
        char ch = 'A';
        // 将输出 true
        System.out.println("65 和 'A' 是否相等? " + (it == ch));
        String str1 = new String("hello");
        String str2 = new String("hello");
        // 将输出 false
        System.out.println("str1 和 str2 是否相等? "
            + (str1 == str2));
        // 将输出 true
        System.out.println("str1 是否 equals str2? "
            + (str1.equals(str2)));
        // 由于 java.lang.String 与 EqualTest 类没有继承关系
        // 所以下面语句导致编译错误
        System.out.println("hello" == new EqualTest());
    }
}
```

运行上面程序, 可以看到 65、65.0f 和 'A' 相等。但对于 str1 和 str2, 因为它们都是引用类型变量, 它们分别指向两个通过 new 关键字创建的 String 对象, 因此 str1 和 str2 两个变量不相等。

对初学者而言, String 还有一个非常容易迷惑的地方: "hello"直接量和 new String("hello")有什么区别呢? 当 Java 程序直接使用形如"hello"的字符串直接量(包括可以在编译时就计算出来的字符串值)时, JVM 将会使用常量池来管理这些字符串; 当使用 new String("hello")时, JVM 会先使用常量池来管理"hello"直接量, 再调用 String 类的构造器来创建一个新的 String 对象, 新创建的 String 对象被保存在堆内存中。换句话说, new String("hello")一共产生了两个字符串对象。



提示:

常量池(constant pool)专门用于管理在编译时被确定并被保存在已编译的.class 文件中的一些数据。它包括了关于类、方法、接口中的常量, 还包括字符串常量。

下面程序示范了 JVM 使用常量池管理字符串直接量的情形。

程序清单: codes\06\6.2\StringCompareTest.java

```
public class StringCompareTest
{
```

```
public static void main(String[] args)
{
    // s1 直接引用常量池中的"疯狂 Java"
    String s1 = "疯狂 Java";
    String s2 = "疯狂";
    String s3 = "Java";
    // s4 后面的字符串值可以在编译时就确定下来
    // s4 直接引用常量池中的"疯狂 Java"
    String s4 = "疯狂" + "Java";
    // s5 后面的字符串值可以在编译时就确定下来
    // s5 直接引用常量池中的"疯狂 Java"
    String s5 = "疯" + "狂" + "Java";
    // s6 后面的字符串值不能在编译时就确定下来
    // 不能引用常量池中的字符串
    String s6 = s2 + s3;
    // 使用 new 调用构造器将会创建一个新的 String 对象
    // s7 引用堆内存中新创建的 String 对象
    String s7 = new String("疯狂 Java");
    System.out.println(s1 == s4); // 输出 true
    System.out.println(s1 == s5); // 输出 true
    System.out.println(s1 == s6); // 输出 false
    System.out.println(s1 == s7); // 输出 false
}
```

JVM 常量池保证相同的字符串直接量只有一个, 不会产生多个副本。例子中的 s1、s4、s5 所引用的字符串可以在编译期就确定下来, 因此它们都将引用常量池中的同一个字符串对象。

使用 new String() 创建的字符串对象是运行时创建出来的, 它被保存在运行时内存区 (即堆内存) 内, 不会放入常量池中。

但在很多时候, 程序判断两个引用变量是否相等时, 也希望有一种类似于“值相等”的判断规则, 并不严格要求两个引用变量指向同一个对象。例如对于两个字符串变量, 可能只是要求它们引用字符串对象里包含的字符序列相同即可认为相等。此时就可以利用 String 对象的 equals() 方法来进行判断, 例如上面程序中的 str1.equals(str2) 将返回 true。

equals() 方法是 Object 类提供的一个实例方法, 因此所有引用变量都可调用该方法来判断是否与其他引用变量相等。但使用这个方法判断两个对象相等的标准与使用 == 运算符没有区别, 同样要求两个引用变量指向同一个对象才会返回 true。因此这个 Object 类提供的 equals() 方法没有太大的实际意义, 如果希望采用自定义的相等标准, 则可采用重写 equals 方法来实现。



提示:

String 已经重写了 Object 的 equals() 方法, String 的 equals() 方法判断两个字符串相等的标准是: 只要两个字符串所包含的字符序列相同, 通过 equals() 比较将返回 true, 否则将返回 false。

注意:

很多书上经常说 equals() 方法是判断两个对象的值相等。这个说法并不准确, 什么叫对象的值呢? 对象的值如何相等? 实际上, 重写 equals() 方法就是提供自定义的相等标准, 你认为怎样是相等, 那就怎样是相等, 一切都是你做主! 在极端的情况下, 你可以让 Person 对象和 Dog 对象相等。



下面程序示范了重写 equals 方法产生 Person 对象和 Dog 对象相等的情形。

程序清单: codes\06\6.2\OverrideEqualsError.java

```
// 定义一个 Person 类
class Person
{
    // 重写 equals() 方法, 提供自定义的相等标准
    public boolean equals(Object obj)
```



```

    {
        // 不加判断, 总是返回 true, 即 Person 对象与任何对象都相等
        return true;
    }
}
// 定义一个 Dog 空类
class Dog{}
public class OverrideEqualsError
{
    public static void main(String[] args)
    {
        Person p = new Person();
        System.out.println("Person 对象是否 equals Dog 对象? "
            + p.equals(new Dog()));
        System.out.println("Person 对象是否 equals String 对象? "
            + p.equals(new String("Hello")));
    }
}

```

编译、运行上面程序, 可以看到 Person 对象和 Dog 对象相等, Person 对象和 String 对象也相等的“荒唐结果”, 造成这种结果的原因是由于重写 Person 类的 equals()方法时没有任何判断, 无条件地返回 true。实际上这种结果也不算太荒唐, 因为 Dog 对象和 Person 对象也不是完全不可能相等, 这要看关心的角度, 比如仅仅关心 Person 对象和 Dog 对象的年龄, 从年纪相等的角度来看, 那就可以认为年龄相等, Person 对象和 Dog 对象就是相等。

大部分时候, 并不希望看到 Person 对象和 Dog 对象相等的“荒唐局面”, 还是希望两个类型相同的对象才可能相等, 并且关键的成员变量相等才能相等。看下面重写 Person 类的 equals()方法, 更符合实际情况。

程序清单: code\06\6.2\OverrideEqualsRight.java

```

class Person
{
    private String name;
    private String idStr;
    public Person(){}
    public Person(String name , String idStr)
    {
        this.name = name;
        this.idStr = idStr;
    }
    // 此处省略 name 和 idStr 的 setter 和 getter 方法
    ...
    // 重写 equals() 方法, 提供自定义的相等标准
    public boolean equals(Object obj)
    {
        // 如果两个对象为同一个对象
        if (this == obj)
            return true;
        // 只有当 obj 是 Person 对象
        if (obj != null && obj.getClass() == Person.class)
        {
            Person personObj = (Person)obj;
            // 并且当前对象的 idStr 与 obj 对象的 idStr 相等时才可判断两个对象相等
            if (this.getIdStr().equals(personObj.getIdStr()))
            {
                return true;
            }
        }
        return false;
    }
}
public class OverrideEqualsRight
{
    public static void main(String[] args)
    {
        Person p1 = new Person("孙悟空" , "12343433433");
    }
}

```



```

Person p2 = new Person("孙行者", "12343433433");
Person p3 = new Person("孙悟空", "99933433");
// p1 和 p2 的 idStr 相等, 所以输出 true
System.out.println("p1 和 p2 是否相等? "
    + p1.equals(p2));
// p2 和 p3 的 idStr 不相等, 所以输出 false
System.out.println("p2 和 p3 是否相等? "
    + p2.equals(p3));
    }
}

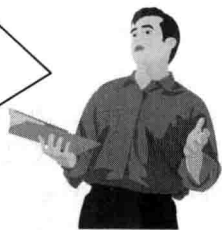
```

上面程序重写 Person 类的 equals() 方法, 指定了 Person 对象和其他对象相等的标准: 另一个对象必须是 Person 类的实例, 且两个 Person 对象的 idStr 相等, 即可判断两个 Person 对象相等。在这种判断标准下, 可认为只要两个 Person 对象的身份证字符串相等, 即可判断相等。



学生提问: 上面程序中判断 obj 是否为 Person 类的实例时, 为何不用 obj instanceof Person 来判断呢?

答: 对于 instanceof 运算符而言, 当前面对象是后面类的实例或其子类的实例时都将返回 true, 所以重写 equals() 方法判断两个对象是否为同一个类的实例时使用 instanceof 是有问题的。比如有一个 Teacher 类型的变量 t, 如果判断 t instanceof Person, 这也将返回 true。但对于重写 equals() 方法的要求而言, 通常要求两个对象是同一个类的实例, 因此使用 instanceof 运算符不太合适。改为使用 t.getClass() == Person.class 比较合适。这行代码用到了反射基础, 读者可参考第 18 章来理解此行代码。



通常而言, 正确地重写 equals() 方法应该满足下列条件。

- 自反性: 对任意 x, x.equals(x) 一定返回 true。
- 对称性: 对任意 x 和 y, 如果 y.equals(x) 返回 true, 则 x.equals(y) 也返回 true。
- 传递性: 对任意 x, y, z, 如果 x.equals(y) 返回 true, y.equals(z) 返回 true, 则 x.equals(z) 一定返回 true。
- 一致性: 对任意 x 和 y, 如果对象中用于等价比较的信息没有改变, 那么无论调用 x.equals(y) 多少次, 返回的结果应该保持一致, 要么一直是 true, 要么一直是 false。
- 对任何不是 null 的 x, x.equals(null) 一定返回 false。

Object 默认提供的 equals() 只是比较对象的地址, 即 Object 类的 equals() 方法比较的结果与 == 运算符比较的结果完全相同。因此, 在实际应用中常常需要重写 equals() 方法, 重写 equals 方法时, 相等条件是由业务要求决定的, 因此 equals() 方法的实现也是由业务要求决定的。

6.3 类成员

static 关键字修饰的成员就是类成员, 前面已经介绍的类成员有类变量、类方法、静态初始化块三个成分, static 关键字不能修饰构造器。static 修饰的类成员属于整个类, 不属于单个实例。

6.3.1 理解类成员

在 Java 类里只能包含成员变量、方法、构造器、初始化块、内部类 (包括接口、枚举) 5 种成员, 目前已经介绍了前面 4 种, 其中 static 可以修饰成员变量、方法、初始化块、内部类 (包括接口、枚举), 以 static 修饰的成员就是类成员。类成员属于整个类, 而不属于单个对象。

类变量属于整个类, 当系统第一次准备使用该类时, 系统会为该类变量分配内存空间, 类变量开始生效, 直到该类被卸载, 该类的类变量所占有的内存才被系统的垃圾回收机制回收。类变量生存范围几乎等同于该类的生存范围。当类初始化完成后, 类变量也被初始化完成。

类变量既可通过类来访问, 也可通过类的对象来访问。但通过类的对象来访问类变量时, 实际上并不是访问该对象所拥有的变量, 因为当系统创建该类的对象时, 系统不会为类变量分配内存, 也不会

再次对类变量进行初始化,也就是说,对象根本不拥有对应类的类变量。通过对象访问类变量只是一种假象,通过对象访问的依然是该类的类变量,可以这样理解:当通过对象来访问类变量时,系统会在底层转换为通过该类来访问类变量。

**提示:**

很多语言都不允许通过对象访问类变量,对象只能访问实例变量;类变量必须通过类来访问。

由于对象实际上并不持有类变量,类变量是由该类持有的,同一个类的所有对象访问类变量时,实际上访问的都是该类所持有的变量。因此,从程序运行表面来看,即可看到同一类的所有实例的类变量共享同一块内存区。

类方法也是类成员的一种,类方法也是属于类的,通常直接使用类作为调用者来调用类方法,但也可以使用对象来调用类方法。与类变量类似,即使使用对象来调用类方法,其效果也与采用类来调用类方法完全一样。

当使用实例来访问类成员时,实际上依然是委托给该类来访问类成员,因此即使某个实例为 `null`,它也可以访问它所属类的类成员。例如如下代码:

程序清单: codes\06\6.3\NullAccessStatic.java

```
public class NullAccessStatic
{
    private static void test()
    {
        System.out.println("static 修饰的类方法");
    }
    public static void main(String[] args)
    {
        // 定义一个 NullAccessStatic 变量, 其值为 null
        NullAccessStatic nas = null;
        // 使用 null 对象调用所属类的静态方法
        nas.test();
    }
}
```

编译、运行上面程序,一切正常,程序将打印出“static 修饰的类方法”字符串,这表明 `null` 对象可以访问它所属类的类成员。

**提示:**

如果一个 `null` 对象访问实例成员(包括实例变量和实例方法),将会引发 `NullPointerException` 异常,因为 `null` 表明该实例根本不存在,既然实例不存在,那么它的实例变量和实例方法自然也不存在。

静态初始化块也是类成员的一种,静态初始化块用于执行类初始化动作,在类的初始化阶段,系统会调用该类的静态初始化块来对类进行初始化。一旦该类初始化结束后,静态初始化块将永远不会获得执行的机会。

对 `static` 关键字而言,有一条非常重要的规则:类成员(包括方法、初始化块、内部类和枚举类)不能访问实例成员(包括成员变量、方法、初始化块、内部类和枚举类)。因为类成员是属于类的,类成员的作用域比实例成员的作用域更大,完全可能出现类成员已经初始化完成,但实例成员还不曾初始化的情况,如果允许类成员访问实例成员将会引起大量错误。

►► 6.3.2 单例 (Singleton) 类

大部分时候都把类的构造器定义成 `public` 访问权限,允许任何类自由创建该类的对象。但在某些时候,允许其他类自由创建该类的对象没有任何意义,还可能造成系统性能下降(因为频繁地创建对象、回收对象带来的系统开销问题)。例如,系统可能只有一个窗口管理器、一个假脱机打印设备或一个数据库引擎访问点,此时如果在系统中为这些类创建多个对象就没有太大的实际意义。

如果一个类始终只能创建一个实例，则这个类被称为单例类。

总之，在一些特殊场景下，要求不允许自由创建该类的对象，而只允许为该类创建一个对象。为了避免其他类自由创建该类的实例，应该把该类的构造器使用 `private` 修饰，从而把该类的所有构造器隐藏起来。

根据良好封装的原则：一旦把该类的构造器隐藏起来，就需要提供一个 `public` 方法作为该类的访问点，用于创建该类的对象，且该方法必须使用 `static` 修饰（因为调用该方法之前还不存在对象，因此调用该方法的不可可能是对象，只能是类）。

除此之外，该类还必须缓存已经创建的对象，否则该类无法知道是否曾经创建过对象，也就无法保证只创建一个对象。为此该类需要使用一个成员变量来保存曾经创建的对象，因为该成员变量需要被上面的静态方法访问，故该成员变量必须使用 `static` 修饰。

基于上面的介绍，下面程序创建了一个单例类。

程序清单：codes\06\6.3\SingletonTest.java

```
class Singleton
{
    // 使用一个类变量来缓存曾经创建的实例
    private static Singleton instance;
    // 对构造器使用 private 修饰，隐藏该构造器
    private Singleton() {}
    // 提供一个静态方法，用于返回 Singleton 实例
    // 该方法可以加入自定义控制，保证只产生一个 Singleton 对象
    public static Singleton getInstance()
    {
        // 如果 instance 为 null，则表明还不曾创建 Singleton 对象
        // 如果 instance 不为 null，则表明已经创建了 Singleton 对象
        // 将不会重新创建新的实例
        if (instance == null)
        {
            // 创建一个 Singleton 对象，并将其缓存起来
            instance = new Singleton();
        }
        return instance;
    }
}

public class SingletonTest
{
    public static void main(String[] args)
    {
        // 创建 Singleton 对象不能通过构造器
        // 只能通过 getInstance 方法来得到实例
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2); // 将输出 true
    }
}
```

正是通过上面 `getInstance` 方法提供的自定义控制（这也是封装的优势：不允许自由访问类的成员变量和实现细节，而是通过方法来控制合适暴露），保证 `Singleton` 类只能产生一个实例。所以，在 `SingletonTest` 类的 `main()` 方法中，看到两次产生的 `Singleton` 对象实际上是同一个对象。

6.4 final 修饰符

`final` 关键字可用于修饰类、变量和方法，`final` 关键字有点类似 C# 里的 `sealed` 关键字，用于表示它修饰的类、方法和变量不可改变。

`final` 修饰变量时，表示该变量一旦获得了初始值就不可被改变，`final` 既可以修饰成员变量（包括类变量和实例变量），也可以修饰局部变量、形参。有的书上介绍说 `final` 修饰的变量不能被赋值，这种说法是错误的！严格的说法是，`final` 修饰的变量不可被改变，一旦获得了初始值，该 `final` 变量的值就不能被重新赋值。

由于 `final` 变量获得初始值之后不能被重新赋值, 因此 `final` 修饰成员变量和修饰局部变量时有一定的不同。

6.4.1 final 成员变量

成员变量是随类初始化或对象初始化而初始化的。当类初始化时, 系统会为该类的类变量分配内存, 并分配默认值; 当创建对象时, 系统会为该对象的实例变量分配内存, 并分配默认值。也就是说, 当执行静态初始化块时可以对类变量赋初始值; 当执行普通初始化块、构造器时可对实例变量赋初始值。因此, 成员变量的初始值可以在定义该变量时指定默认值, 也可以在初始化块、构造器中指定初始值。

对于 `final` 修饰的成员变量而言, 一旦有了初始值, 就不能被重新赋值, 如果既没有在定义成员变量时指定初始值, 也没有在初始化块、构造器中为成员变量指定初始值, 那么这些成员变量的值将一直是系统默认分配的 `0`、`'\u0000'`、`false` 或 `null`, 这些成员变量也就完全失去了存在的意义。因此 Java 语法规则规定: **`final` 修饰的成员变量必须由程序员显式地指定初始值。**

归纳起来, `final` 修饰的类变量、实例变量能指定初始值的地方如下。

- 类变量: 必须在静态初始化块中指定初始值或声明该类变量时指定初始值, 而且只能在两个地方的其中之一指定。
- 实例变量: 必须在非静态初始化块、声明该实例变量或构造器中指定初始值, 而且只能在三个地方的其中之一指定。

`final` 修饰的实例变量, 要么在定义该实例变量时指定初始值, 要么在普通初始化块或构造器中为该实例变量指定初始值。但需要注意的是, 如果普通初始化块已经为某个实例变量指定了初始值, 则不能再在构造器中为该实例变量指定初始值; `final` 修饰的类变量, 要么在定义该类变量时指定初始值, 要么在静态初始化块中为该类变量指定初始值。

实例变量不能在静态初始化块中指定初始值, 因为静态初始化块是静态成员, 不可访问实例变量——非静态成员; 类变量不能在普通初始化块中指定初始值, 因为类变量在类初始化阶段已经被初始化了, 普通初始化块不能对其重新赋值。

下面程序演示了 `final` 修饰成员变量的效果, 详细示范了 `final` 修饰成员变量的各种具体情况。

程序清单: codes\06\6.4\FinalVariableTest.java

```
public class FinalVariableTest
{
    // 定义成员变量时指定默认值, 合法
    final int a = 6;
    // 下面变量将在构造器或初始化块中分配初始值
    final String str;
    final int c;
    final static double d;
    // 既没有指定默认值, 又没有在初始化块、构造器中指定初始值
    // 下面定义的 ch 实例变量是不合法的
    // final char ch;
    // 初始化块, 可对没有指定默认值的实例变量指定初始值
    {
        // 在初始化块中为实例变量指定初始值, 合法
        str = "Hello";
        // 定义 a 实例变量时已经指定了默认值
        // 不能为 a 重新赋值, 因此下面赋值语句非法
        // a = 9;
    }
    // 静态初始化块, 可对没有指定默认值的类变量指定初始值
    static
    {
        // 在静态初始化块中为类变量指定初始值, 合法
        d = 5.6;
    }
    // 构造器, 可对既没有指定默认值, 又没有在初始化块中
```



```
// 指定初始值的实例变量指定初始值
public FinalVariableTest()
{
    // 如果在初始化块中已经对 str 指定了初始值
    // 那么在构造器中不能对 final 变量重新赋值, 下面赋值语句非法
    // str = "java";
    c = 5;
}
public void changeFinal()
{
    // 普通方法不能为 final 修饰的成员变量赋值
    // d = 1.2;
    // 不能在普通方法中为 final 成员变量指定初始值
    // ch = 'a';
}
public static void main(String[] args)
{
    FinalVariableTest ft = new FinalVariableTest();
    System.out.println(ft.a);
    System.out.println(ft.c);
    System.out.println(ft.d);
}
```

上面程序详细示范了初始化 final 成员变量的各种情形, 读者参考程序中的注释应该可以很清楚地看出 final 修饰成员变量的用法。



注意:

与普通成员变量不同的是, final 成员变量 (包括实例变量和类变量) 必须由程序员显式初始化, 系统不会对 final 成员进行隐式初始化。



如果打算在构造器、初始化块中对 final 成员变量进行初始化, 则不要在初始化之前就访问成员变量的值。例如下面程序将会引起错误。

程序清单: codes\06\6.4\FinalErrorTest.java

```
public class FinalErrorTest
{
    // 定义一个 final 修饰的实例变量
    // 系统不会对 final 成员变量进行默认初始化
    final int age;
    {
        // age 没有初始化, 所以此处代码将引起错误
        System.out.println(age);
        age = 6;
        System.out.println(age);
    }
    public static void main(String[] args)
    {
        new FinalErrorTest();
    }
}
```

上面程序中定义了一个 final 成员变量: age, 系统不会对 age 成员变量进行隐式初始化, 所以初始化块中粗体字标识的代码行将引起错误, 因为它试图访问一个未初始化的变量。只要把定义 age 时的 final 修饰符去掉, 上面程序就正确了。

➤➤ 6.4.2 final 局部变量

系统不会对局部变量进行初始化, 局部变量必须由程序员显式初始化。因此使用 final 修饰局部变量时, 既可以在定义时指定默认值, 也可以不指定默认值。

如果 final 修饰的局部变量在定义时没有指定默认值, 则可以在后面代码中对该 final 变量赋初始值, 但只能一次, 不能重复赋值; 如果 final 修饰的局部变量在定义时已经指定默认值, 则后面代码中不能

再对该变量赋值。下面程序示范了 `final` 修饰局部变量、形参的情形。

程序清单: codes\06\6.4\FinalLocalVariableTest.java

```
public class FinalLocalVariableTest
{
    public void test(final int a)
    {
        // 不能对 final 修饰的形参赋值, 下面语句非法
        // a = 5;
    }
    public static void main(String[] args)
    {
        // 定义 final 局部变量时指定默认值, 则 str 变量无法重新赋值
        final String str = "hello";
        // 下面赋值语句非法
        // str = "Java";
        // 定义 final 局部变量时没有指定默认值, 则 d 变量可被赋值一次
        final double d;
        // 第一次赋初始值, 成功
        d = 5.6;
        // 对 final 变量重复赋值, 下面语句非法
        // d = 3.4;
    }
}
```

在上面程序中还示范了 `final` 修饰形参的情形。因为形参在调用该方法时, 由系统根据传入的参数来完成初始化, 因此使用 `final` 修饰的形参不能被赋值。

▶▶ 6.4.3 `final` 修饰基本类型变量和引用类型变量的区别

当使用 `final` 修饰基本类型变量时, 不能对基本类型变量重新赋值, 因此基本类型变量不能被改变。但对于引用类型变量而言, 它保存的仅仅是一个引用, `final` 只保证这个引用类型变量所引用的地址不会改变, 即一直引用同一个对象, 但这个对象完全可以发生改变。

下面程序示范了 `final` 修饰数组和 `Person` 对象的情形。

程序清单: codes\06\6.4\FinalReferenceTest.java

```
class Person
{
    private int age;
    public Person(){}
    // 有参数的构造器
    public Person(int age)
    {
        this.age = age;
    }
    // 省略 age 的 setter 和 getter 方法
    // age 的 setter 和 getter 方法
    ...
}
public class FinalReferenceTest
{
    public static void main(String[] args)
    {
        // final 修饰数组变量, iArr 是一个引用变量
        final int[] iArr = {5, 6, 12, 9};
        System.out.println(Arrays.toString(iArr));
        // 对数组元素进行排序, 合法
        Arrays.sort(iArr);
        System.out.println(Arrays.toString(iArr));
        // 对数组元素赋值, 合法
        iArr[2] = -8;
        System.out.println(Arrays.toString(iArr));
        // 下面语句对 iArr 重新赋值, 非法
        // iArr = null;
        // final 修饰 Person 变量, p 是一个引用变量
    }
}
```

```
final Person p = new Person(45);
// 改变 Person 对象的 age 实例变量, 合法
p.setAge(23);
System.out.println(p.getAge());
// 下面语句对 p 重新赋值, 非法
// p = null;
}
```

从上面程序中可以看出, 使用 `final` 修饰的引用类型变量不能被重新赋值, 但可以改变引用类型变量所引用对象的内容。例如上面 `iArr` 变量所引用的数组对象, `final` 修饰后的 `iArr` 变量不能被重新赋值, 但 `iArr` 所引用数组的数组元素可以被改变。与此类似的是, `p` 变量也使用了 `final` 修饰, 表明 `p` 变量不能被重新赋值, 但 `p` 变量所引用 `Person` 对象的成员变量的值可以被改变。

6.4.4 可执行“宏替换”的 `final` 变量

对一个 `final` 变量来说, 不管它是类变量、实例变量, 还是局部变量, 只要该变量满足三个条件, 这个 `final` 变量就不再是一个变量, 而是相当于一个直接量。

- 使用 `final` 修饰符修饰。
- 在定义该 `final` 变量时指定了初始值。
- 该初始值可以在编译时就被确定下来。

看如下程序。

程序清单: codes\06\6.4\FinalLocalTest.java

```
public class FinalLocalTest
{
    public static void main(String[] args)
    {
        // 定义一个普通局部变量
        final int a = 5;
        System.out.println(a);
    }
}
```

上面程序中的粗体字代码定义了一个 `final` 局部变量, 并在定义该 `final` 变量时指定初始值为 5。对于这个程序来说, 变量 `a` 其实根本不存在, 当程序执行 `System.out.println(a);` 代码时, 实际转换为执行 `System.out.println(5)`。



注意： `final` 修饰符的一个重要用途就是定义“宏变量”。当定义 `final` 变量时就为该变量指定了初始值, 而且该初始值可以在编译时就确定下来, 那么这个 `final` 变量本质上就是一个“宏变量”, 编译器会把程序中所有用到该变量的地方直接替换成该变量的值。



除了上面那种为 `final` 变量赋值时赋直接量的情况外, 如果被赋的表达式只是基本的算术表达式或字符串连接运算, 没有访问普通变量, 调用方法, Java 编译器同样会将这种 `final` 变量当成“宏变量”处理。示例如下。

程序清单: codes\06\6.4\FinalReplaceTest.java

```
public class FinalReplaceTest
{
    public static void main(String[] args)
    {
        // 下面定义了 4 个 final “宏变量”
        final int a = 5 + 2;
        final double b = 1.2 / 3;
        final String str = "疯狂" + "Java";
        final String book = "疯狂 Java 讲义: " + 99.0;
        // 下面的 book2 变量的值因为调用了方法, 所以无法在编译时被确定下来
        final String book2 = "疯狂 Java 讲义: " + String.valueOf(99.0); //①
    }
}
```



```

        System.out.println(book == "疯狂 Java 讲义: 99.0");
        System.out.println(book2 == "疯狂 Java 讲义: 99.0");
    }
}

```

上面程序中粗体字代码定义了 4 个 `final` 变量, 程序为这 4 个变量赋初始值指定的初始值要么是算术表达式, 要么是字符串连接运算。即使字符串连接运算中包含隐式类型(将数值转换为字符串)转换, 编译器依然可以在编译时就确定 `a`、`b`、`str`、`book` 这 4 个变量的值, 因此它们都是“宏变量”。

从表面上看, ①行代码定义的 `book2` 与 `book` 没有太大的区别, 只是定义 `book2` 变量时显式将数值 99.0 转换为字符串, 但由于该变量的值需要调用 `String` 类的方法, 因此编译器无法在编译时确定 `book2` 的值, `book2` 不会被当成“宏变量”处理。

程序最后两行代码分别判断 `book`、`book2` 和“疯狂 Java 讲义: 99.0”是否相等。由于 `book` 是一个“宏变量”, 它将被直接替换成“疯狂 Java 讲义: 99.0”, 因此 `book` 和“疯狂 Java 讲义: 99.0”相等, 但 `book2` 和该字符串不相等。



提示:

Java 会使用常量池来管理曾经用过的字符串直接量, 例如执行 `String a = "java";` 语句之后, 常量池中就会缓存一个字符串“java”; 如果程序再次执行 `String b = "java";`, 系统将会让 `b` 直接指向常量池中的“java”字符串, 因此 `a==b` 将会返回 `true`。

为了加深对 `final` 修饰符的印象, 下面再看一个程序。

程序清单: codes\06\6.4\StringJoinTest.java

```

public class StringJoinTest
{
    public static void main(String[] args)
    {
        String s1 = "疯狂 Java";
        // s2 变量引用的字符串可以在编译时就确定下来
        // 因此 s2 直接引用常量池中已有的“疯狂 Java”字符串
        String s2 = "疯狂" + "Java";
        System.out.println(s1 == s2);
        // 定义 2 个字符串直接量
        String str1 = "疯狂";    //①
        String str2 = "Java";    //②
        // 将 str1 和 str2 进行连接运算
        String s3 = str1 + str2;
        System.out.println(s1 == s3);
    }
}

```

上面程序中两行粗体字代码分别判断 `s1` 和 `s2` 是否相等, 以及 `s1` 和 `s3` 是否相等。`s1` 是一个普通的字符串直接量“疯狂 Java”, `s2` 的值是两个字符串直接量进行连接运算, 由于编译器可以在编译阶段就确定 `s2` 的值为“疯狂 Java”, 所以系统会让 `s2` 直接指向常量池中缓存的“疯狂 Java”字符串。因此 `s1==s2` 将输出 `true`。

对于 `s3` 而言, 它的值由 `str1` 和 `str2` 进行连接运算后得到。由于 `str1`、`str2` 只是两个普通变量, 编译器不会执行“宏替换”, 因此编译器无法在编译时确定 `s3` 的值, 也就无法让 `s3` 指向字符串池中缓存的“疯狂 Java”。由此可见, `s1==s3` 将输出 `false`。

让 `s1==s3` 输出 `true` 也很简单, 只要让编译器可以对 `str1`、`str2` 两个变量执行“宏替换”, 这样编译器即可在编译阶段就确定 `s3` 的值, 就会让 `s3` 指向字符串池中缓存的“疯狂 Java”。也就是说, 只要将①、②两行代码所定义的 `str1`、`str2` 使用 `final` 修饰即可。

注意:

对于实例变量而言, 既可以在定义该变量时赋初始值, 也可以在非静态初始化块、构造器中对它赋初始值, 在这三个地方指定初始值的效果基本一样。但对于 `final` 实例变量而言, 只有在定义该变量时指定初始值才会有“宏变量”的效果。



6.4.5 final 方法

final 修饰的方法不可被重写, 如果出于某些原因, 不希望子类重写父类的某个方法, 则可以使用 **final** 修饰该方法。

Java 提供的 **Object** 类里就有一个 **final** 方法: **getClass()**, 因为 Java 不希望任何类重写这个方法, 所以使用 **final** 把这个方法密封起来。但对于该类提供的 **toString()** 和 **equals()** 方法, 都允许子类重写, 因此没有使用 **final** 修饰它们。

下面程序试图重写 **final** 方法, 将会引发编译错误。

程序清单: codes\06\6.4\FinalMethodTest.java

```
public class FinalMethodTest
{
    public final void test() {}
}
class Sub extends FinalMethodTest
{
    // 下面方法定义将出现编译错误, 不能重写 final 方法
    public void test() {}
}
```

上面程序中父类是 **FinalMethodTest**, 该类里定义的 **test()** 方法是一个 **final** 方法, 如果其子类试图重写该方法, 将会引发编译错误。

对于一个 **private** 方法, 因为它仅在当前类中可见, 其子类无法访问该方法, 所以子类无法重写该方法——如果子类中定义一个与父类 **private** 方法有相同方法名、相同形参列表、相同返回值类型的方法, 也不是方法重写, 只是重新定义了一个新方法。因此, 即使使用 **final** 修饰一个 **private** 访问权限的方法, 依然可以在其子类中定义与该方法具有相同方法名、相同形参列表、相同返回值类型的方法。

下面程序示范了如何在子类中“重写”父类的 **private final** 方法。

程序清单: codes\06\6.4\PrivateFinalMethodTest.java

```
public class PrivateFinalMethodTest
{
    private final void test() {}
}
class Sub extends PrivateFinalMethodTest
{
    // 下面的方法定义不会出现问题
    public void test() {}
}
```

上面程序没有任何问题, 虽然子类和父类同样包含了同名的 **void test()** 方法, 但子类并不是重写父类的方法, 因此即使父类的 **void test()** 方法使用了 **final** 修饰, 子类中依然可以定义 **void test()** 方法。

final 修饰的方法仅仅是不能被重写, 并不是不能被重载, 因此下面程序完全没有问题。

```
public class FinalOverload
{
    // final 修饰的方法只是不能被重写, 完全可以被重载
    public final void test() {}
    public final void test(String arg) {}
}
```

6.4.6 final 类

final 修饰的类不可以有子类, 例如 **java.lang.Math** 类就是一个 **final** 类, 它不可以有子类。

当子类继承父类时, 将可以访问到父类内部数据, 并可通过重写父类方法来改变父类方法的实现细节, 这可能导致一些不安全的因素。为了保证某个类不可被继承, 则可以使用 **final** 修饰这个类。下面代码示范了 **final** 修饰的类不可被继承。

```
public final class FinalClass {}
// 下面的类定义将出现编译错误
class Sub extends FinalClass {}
```

因为 FinalClass 类是一个 final 类, 而 Sub 试图继承 FinalClass 类, 这将会引起编译错误。

6.4.7 不可变类

不可变 (immutable) 类的意思是创建该类的实例后, 该实例的实例变量是不可改变的。Java 提供的 8 个包装类和 java.lang.String 类都是不可变类, 当创建它们的实例后, 其实例的实例变量不可改变。例如如下代码:

```
Double d = new Double(6.5);
String str = new String("Hello");
```

上面程序创建了一个 Double 对象和一个 String 对象, 并为这个两对象传入了 6.5 和 "Hello" 字符串作为参数, 那么 Double 类和 String 类肯定需要提供实例变量来保存这两个参数, 但程序无法修改这两个实例变量的值, 因此 Double 类和 String 类没有提供修改它们的方法。

如果需要创建自定义的不可变类, 可遵守如下规则。

- 使用 private 和 final 修饰符来修饰该类的成员变量。
- 提供带参数构造器, 用于根据传入参数来初始化类里的成员变量。
- 仅为该类的成员变量提供 getter 方法, 不要为该类的成员变量提供 setter 方法, 因为普通方法无法修改 final 修饰的成员变量。
- 如果有必要, 重写 Object 类的 hashCode() 和 equals() 方法。equals() 方法根据关键成员变量来作为两个对象是否相等的标准, 除此之外, 还应该保证两个用 equals() 方法判断为相等的对象的 hashCode() 也相等。

例如, java.lang.String 这个类就做得很好, 它就是根据 String 对象里的字符序列来作为相等的标准, 其 hashCode() 方法也是根据字符序列计算得到的。下面程序测试了 java.lang.String 类的 equals() 和 hashCode() 方法。

程序清单: codes\06\6.4\ImmutableStringTest.java

```
public class ImmutableStringTest
{
    public static void main(String[] args)
    {
        String str1 = new String("Hello");
        String str2 = new String("Hello");
        System.out.println(str1 == str2); // 输出 false
        System.out.println(str1.equals(str2)); // 输出 true
        // 下面两次输出的 hashCode 相同
        System.out.println(str1.hashCode());
        System.out.println(str2.hashCode());
    }
}
```

下面定义一个不可变的 Address 类, 程序把 Address 类的 detail 和 postCode 成员变量都使用 private 隐藏起来, 并使用 final 修饰这两个成员变量, 不允许其他方法修改这两个成员变量的值。

程序清单: codes\06\6.4\Address.java

```
public class Address
{
    private final String detail;
    private final String postCode;
    // 在构造器里初始化两个实例变量
    public Address()
    {
        this.detail = "";
        this.postCode = "";
    }
    public Address(String detail, String postCode)
    {
        this.detail = detail;
        this.postCode = postCode;
    }
    // 仅为两个实例变量提供 getter 方法
```

```

public String getDetail()
{
    return this.detail;
}
public String getPostCode()
{
    return this.postCode;
}
// 重写 equals() 方法, 判断两个对象是否相等
public boolean equals(Object obj)
{
    if (this == obj)
    {
        return true;
    }
    if (obj != null && obj.getClass() == Address.class)
    {
        Address ad = (Address)obj;
        // 当 detail 和 postCode 相等时, 可认为两个 Address 对象相等
        if (this.getDetail().equals(ad.getDetail())
            && this.getPostCode().equals(ad.getPostCode()))
        {
            return true;
        }
    }
    return false;
}
public int hashCode()
{
    return detail.hashCode() + postCode.hashCode() * 31;
}
}

```

对于上面的 `Address` 类, 当程序创建了 `Address` 对象后, 同样无法修改该 `Address` 对象的 `detail` 和 `postCode` 实例变量。

与不可变类对应的是可变类, 可变类的含义是该类的实例变量是可变的。大部分时候所创建的类都是可变类, 特别是 `JavaBean`, 因为总是为其实例变量提供了 `setter` 和 `getter` 方法。

与可变类相比, 不可变类的实例在整个生命周期中永远处于初始化状态, 它的实例变量不可改变。因此对不可变类的实例的控制将更加简单。

前面介绍 `final` 关键字时提到, 当使用 `final` 修饰引用类型变量时, 仅表示这个引用类型变量不可被重新赋值, 但引用类型变量所指向的对象依然可改变。这就产生了一个问题: 当创建不可变类时, 如果它包含成员变量的类型是可变的, 那么其对象的成员变量的值依然是可改变的——这个不可变类其实是失败的。

下面程序试图定义一个不可变的 `Person` 类, 但因为 `Person` 类包含一个引用类型的成员变量, 且这个引用类是可变类, 所以导致 `Person` 类也变成了可变类。

程序清单: codes\06\6.4\Person.java

```

class Name
{
    private String firstName;
    private String lastName;
    public Name() {}
    public Name(String firstName, String lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    // 省略 firstName、lastName 的 setter 和 getter 方法
    ...
}
public class Person
{
    private final Name name;
    public Person(Name name)

```



```

    {
        this.name = name;
    }
    public Name getName()
    {
        return name;
    }
    public static void main(String[] args)
    {
        Name n = new Name("悟空", "孙");
        Person p = new Person(n);
        // Person对象的name的firstName值为"悟空"
        System.out.println(p.getName().getFirstName());
        // 改变Person对象的name的firstName值
        n.setFirstName("八戒");
        // Person对象的name的firstName值被改为"八戒"
        System.out.println(p.getName().getFirstName());
    }
}

```

上面程序中粗体字代码修改了 Name 对象（可变类的实例）的 firstName 的值，但由于 Person 类的 name 实例变量引用了该 Name 对象，这就会导致 Person 对象的 name 的 firstName 会被改变，这就破坏了设计 Person 类的初衷。

为了保持 Person 对象的不可变性，必须保护好 Person 对象的引用类型的成员变量：name，让程序无法访问到 Person 对象的 name 成员变量，也就无法利用 name 成员变量的可变性来改变 Person 对象了。为此将 Person 类改为如下：

```

public class Person
{
    private final Name name;
    public Person(Name name)
    {
        // 设置 name 实例变量为临时创建的 Name 对象，该对象的 firstName 和 lastName
        // 与传入的 name 参数的 firstName 和 lastName 相同
        this.name = new Name(name.getFirstName(), name.getLastName());
    }
    public Name getName()
    {
        // 返回一个匿名对象，该对象的 firstName 和 lastName
        // 与该对象里的 name 的 firstName 和 lastName 相同
        return new Name(name.getFirstName(), name.getLastName());
    }
}

```

注意阅读上面代码中的粗体字部分，Person 类改写了设置 name 实例变量的方法，也改写了 name 的 getter 方法。当程序向 Person 构造器里传入一个 Name 对象时，该构造器创建 Person 对象时并不是直接利用已有的 Name 对象（利用已有的 Name 对象有风险，因为这个已有的 Name 对象是可变的，如果程序改变了这个 Name 对象，将会导致 Person 对象也发生变化），而是重新创建了一个 Name 对象来赋给 Person 对象的 name 实例变量。当 Person 对象返回 name 变量时，它并没有直接把 name 实例变量返回，直接返回 name 实例变量的值也可能导致它所引用的 Name 对象被修改。

如果将 Person 类定义改为上面形式，再次运行 codes\06\6.4\Person.java 程序，将看到 Person 对象的 name 的 firstName 不会被修改。

因此，如果需要设计一个不可变类，尤其要注意其引用类型的成员变量，如果引用类型的成员变量的类是可变的，就必须采取必要的措施来保护该成员变量所引用的对象不会被修改，这样才能创建真正的不可变类。

➤➤ 6.4.8 缓存实例的不可变类

不可变类的实例状态不可改变，可以很方便地被多个对象所共享。如果程序经常需要使用相同的不可变类实例，则应该考虑缓存这种不可变类的实例。毕竟重复创建相同的对象没有太大的意义，而且加大系统开销。如果可能，应该将已经创建的不可变类的实例进行缓存。

缓存是软件设计中一个非常有用的模式,缓存的实现方式有很多种,不同的实现方式可能存在较大的性能差别,关于缓存的性能问题此处不做深入讨论。

本节将使用一个数组来作为缓存池,从而实现一个缓存实例的不可变类。

程序清单: codes\06\6.4\CacheImmutableTest.java

```
class CacheImmutable
{
    private static int MAX_SIZE = 10;
    // 使用数组来缓存已有的实例
    private static CacheImmutable[] cache
        = new CacheImmutable[MAX_SIZE];
    // 记录缓存实例在缓存中的位置, cache[pos-1]是最新缓存的实例
    private static int pos = 0;
    private final String name;
    private CacheImmutable(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return name;
    }
    public static CacheImmutable valueOf(String name)
    {
        // 遍历已缓存的对象,
        for (int i = 0 ; i < MAX_SIZE; i++)
        {
            // 如果已有相同实例, 则直接返回该缓存的实例
            if (cache[i] != null
                && cache[i].getName().equals(name))
            {
                return cache[i];
            }
        }
        // 如果缓存池已满
        if (pos == MAX_SIZE)
        {
            // 把缓存的第一个对象覆盖, 即把刚刚生成的对象放在缓存池的最开始位置
            cache[0] = new CacheImmutable(name);
            // 把 pos 设为 1
            pos = 1;
        }
        else
        {
            // 把新创建的对象缓存起来, pos 加 1
            cache[pos++] = new CacheImmutable(name);
        }
        return cache[pos - 1];
    }
    public boolean equals(Object obj)
    {
        if(this == obj)
        {
            return true;
        }
        if (obj != null && obj.getClass() == CacheImmutable.class)
        {
            CacheImmutable ci = (CacheImmutable)obj;
            return name.equals(ci.getName());
        }
        return false;
    }
    public int hashCode()
    {
        return name.hashCode();
    }
}
```

```

public class CacheImmutableTest
{
    public static void main(String[] args)
    {
        CacheImmutable c1 = CacheImmutable.valueOf("hello");
        CacheImmutable c2 = CacheImmutable.valueOf("hello");
        // 下面代码将输出 true
        System.out.println(c1 == c2);
    }
}

```

上面 `CacheImmutable` 类使用一个数组来缓存该类的对象, 这个数组长度为 `MAX_SIZE`, 即该类共可以缓存 `MAX_SIZE` 个 `CacheImmutable` 对象。当缓存池已满时, 缓存池采用“先进先出”规则来决定哪个对象将被移出缓存池。图 6.3 示范了缓存实例的不可变类示意图。

从图 6.3 中不难看出, 当使用 `CacheImmutable` 类的 `valueOf()` 方法来生成对象时, 系统是否重新生成新的对象, 取决于图 6.3 中被灰色覆盖的数组内是否已经存在该对象。如果该数组中已经缓存了该类的对象, 系统将不会重新生成对象。

`CacheImmutable` 类能控制系统生成 `CacheImmutable` 对象的个数, 需要程序使用该类的 `valueOf()` 方法来得到其对象, 而且程序使用 `private` 修饰符隐藏该类的构造器, 因此程序只能通过该类提供的 `valueOf()` 方法来获取实例。

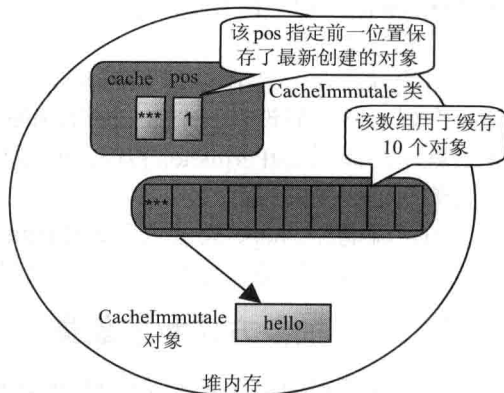


图 6.3 缓存实例的不可变类示意图



提示:

是否需要隐藏 `CacheImmutable` 类的构造器完全取决于系统需求。盲目乱用缓存也可能导致系统性能下降, 缓存的对象会占用系统内存, 如果某个对象只使用一次, 重复使用的概率不大, 缓存该实例就弊大于利; 反之, 如果某个对象需要频繁地重复使用, 缓存该实例就利大于弊。

例如 Java 提供的 `java.lang.Integer` 类, 它就采用了与 `CacheImmutable` 类相同的处理策略, 如果采用 `new` 构造器来创建 `Integer` 对象, 则每次返回全新的 `Integer` 对象; 如果采用 `valueOf()` 方法来创建 `Integer` 对象, 则会缓存该方法创建的对象。下面程序示范了 `Integer` 类构造器和 `valueOf()` 方法存在的差异。

程序清单: codes\06\6.4\IntegerCacheTest.java

```

public class IntegerCacheTest
{
    public static void main(String[] args)
    {
        // 生成新的 Integer 对象
        Integer in1 = new Integer(6);
        // 生成新的 Integer 对象, 并缓存该对象
        Integer in2 = Integer.valueOf(6);
        // 直接从缓存中取出 Integer 对象
        Integer in3 = Integer.valueOf(6);
        System.out.println(in1 == in2); // 输出 false
        System.out.println(in2 == in3); // 输出 true
        // 由于 Integer 只缓存 -128~127 之间的值
        // 因此 200 对应的 Integer 对象没有被缓存
        Integer in4 = Integer.valueOf(200);
        Integer in5 = Integer.valueOf(200);
        System.out.println(in4 == in5); // 输出 false
    }
}

```

运行上面程序, 即可发现两次通过 `Integer.valueOf(6);` 方法生成的 `Integer` 对象是同一个对象。但由于 `Integer` 只缓存 `-128~127` 之间的 `Integer` 对象, 因此两次通过 `Integer.valueOf(200);` 方法生成的 `Integer`

对象不是同一个对象。

6.5 抽象类

当编写一个类时，常常会为该类定义一些方法，这些方法用以描述该类的行为方式，那么这些方法都有具体的方法体。但在某些情况下，某个父类只是知道其子类应该包含怎样的方法，但无法准确地知道这些子类如何实现这些方法。例如定义了一个 `Shape` 类，这个类应该提供一个计算周长的方法 `calPerimeter()`，但不同 `Shape` 子类对周长的计算方法是不同的，即 `Shape` 类无法准确地知道其子类计算周长的方法。

可能有读者会提出，既然 `Shape` 类不知道如何实现 `calPerimeter()` 方法，那就干脆不要管它了！这不是一个好思路：假设有一个 `Shape` 引用变量，该变量实际上引用到 `Shape` 子类的实例，那么这个 `Shape` 变量就无法调用 `calPerimeter()` 方法，必须将其强制类型转换为其子类类型，才可调用 `calPerimeter()` 方法，这就降低了程序的灵活性。

如何既能让 `Shape` 类里包含 `calPerimeter()` 方法，又无须提供其方法实现呢？使用抽象方法即可满足该要求：抽象方法是只有方法签名，没有方法实现的方法。

6.5.1 抽象方法和抽象类

抽象方法和抽象类必须使用 `abstract` 修饰符来定义，有抽象方法的类只能被定义成抽象类，抽象类里可以没有抽象方法。

抽象方法和抽象类的规则如下。

- 抽象类必须使用 `abstract` 修饰符来修饰，抽象方法也必须使用 `abstract` 修饰符来修饰，抽象方法不能有方法体。
- 抽象类不能被实例化，无法使用 `new` 关键字来调用抽象类的构造器创建抽象类的实例。即使抽象类里不包含抽象方法，这个抽象类也不能创建实例。
- 抽象类可以包含成员变量、方法（普通方法和抽象方法都可以）、构造器、初始化块、内部类（接口、枚举）5 种成分。抽象类的构造器不能用于创建实例，主要是用于被其子类调用。
- 含有抽象方法的类（包括直接定义了一个抽象方法；或继承了一个抽象父类，但没有完全实现父类包含的抽象方法；或实现了一个接口，但没有完全实现接口包含的抽象方法三种情况）只能被定义成抽象类。

注意：

归纳起来，抽象类可用“有得有失”4 个字来描述。“得”指的是抽象类多了一个能力：抽象类可以包含抽象方法；“失”指的是抽象类失去了一个能力：抽象类不能用于创建实例。



定义抽象方法只需在普通方法上增加 `abstract` 修饰符，并把普通方法的方法体（也就是方法后花括号括起来的部分）全部去掉，并在方法后增加分号即可。

注意：

抽象方法和空方法体的方法不是同一个概念。例如，`public abstract void test();` 是一个抽象方法，它根本没有方法体，即方法定义后面没有一对花括号；但 `public void test(){}` 方法是一个普通方法，它已经定义了方法体，只是方法体为空，即它的方法体什么也不做，因此这个方法不可使用 `abstract` 来修饰。



定义抽象类只需在普通类上增加 `abstract` 修饰符即可。甚至一个普通类（没有包含抽象方法的类）增加 `abstract` 修饰符后也将变成抽象类。

下面定义一个 `Shape` 抽象类。

程序清单: codes\06\6.5\Shape.java

```

public abstract class Shape
{
    {
        System.out.println("执行 Shape 的初始化块...");
    }
    private String color;
    // 定义一个计算周长的抽象方法
    public abstract double calPerimeter();
    // 定义一个返回形状的抽象方法
    public abstract String getType();
    // 定义 Shape 的构造器, 该构造器并不是用于创建 Shape 对象
    // 而是用于被子类调用
    public Shape() {}
    public Shape(String color)
    {
        System.out.println("执行 Shape 的构造器...");
        this.color = color;
    }
    // 省略 color 的 setter 和 getter 方法
    ...
}

```

上面的 Shape 类里包含了两个抽象方法: calPerimeter() 和 getType(), 所以这个 Shape 类只能被定义成抽象类。Shape 类里既包含了初始化块, 也包含了构造器, 这些都不是在创建 Shape 对象时被调用的, 而是在创建其子类的实例时被调用。

抽象类不能用于创建实例, 只能当作父类被其他子类继承。

下面定义一个三角形类, 三角形类被定义成普通类, 因此必须实现 Shape 类里的所有抽象方法。

程序清单: codes\06\6.5\Triangle.java

```

public class Triangle extends Shape
{
    // 定义三角形的三边
    private double a;
    private double b;
    private double c;
    public Triangle(String color, double a, double b, double c)
    {
        super(color);
        this.setSides(a, b, c);
    }
    public void setSides(double a, double b, double c)
    {
        if (a >= b + c || b >= a + c || c >= a + b)
        {
            System.out.println("三角形两边之和必须大于第三边");
            return;
        }
        this.a = a;
        this.b = b;
        this.c = c;
    }
    // 重写 Shape 类的计算周长的抽象方法
    public double calPerimeter()
    {
        return a + b + c;
    }
    // 重写 Shape 类的返回形状的抽象方法
    public String getType()
    {
        return "三角形";
    }
}

```

上面的 Triangle 类继承了 Shape 抽象类, 并实现了 Shape 类中两个抽象方法, 是一个普通类, 因此可以创建 Triangle 类的实例, 可以让一个 Shape 类型的引用变量指向 Triangle 对象。

下面再定义一个 Circle 普通类, Circle 类也是 Shape 类的一个子类。

程序清单: codes\06\6.5\Circle.java

```
public class Circle extends Shape
{
    private double radius;
    public Circle(String color , double radius)
    {
        super(color);
        this.radius = radius;
    }
    public void setRadius(double radius)
    {
        this.radius = radius;
    }
    // 重写 Shape 类的计算周长的抽象方法
    public double calPerimeter()
    {
        return 2 * Math.PI * radius;
    }
    // 重写 Shape 类的返回形状的抽象方法
    public String getType()
    {
        return getColor() + "圆形";
    }
    public static void main(String[] args)
    {
        Shape s1 = new Triangle("黑色" , 3 , 4, 5);
        Shape s2 = new Circle("黄色" , 3);
        System.out.println(s1.getType());
        System.out.println(s1.calPerimeter());
        System.out.println(s2.getType());
        System.out.println(s2.calPerimeter());
    }
}
```

上面 main()方法中定义了两个 Shape 类型的引用变量, 它们分别指向 Triangle 对象和 Circle 对象。由于在 Shape 类中定义了 calPerimeter()方法和 getType()方法, 所以程序可以直接调用 s1 变量和 s2 变量的 calPerimeter()方法和 getType()方法, 无须强制类型转换为其子类类型。

利用抽象类和抽象方法的优势, 可以更好地发挥多态的优势, 使得程序更加灵活。

当使用 abstract 修饰类时, 表明这个类只能被继承; 当使用 abstract 修饰方法时, 表明这个方法必须由子类提供实现 (即重写)。而 final 修饰的类不能被继承, final 修饰的方法不能被重写。因此 final 和 abstract 永远不能同时使用。

★ 注意: ★

abstract 不能用于修饰成员变量, 不能用于修饰局部变量, 即没有抽象变量、没有抽象成员变量等说法; abstract 也不能用于修饰构造器, 没有抽象构造器, 抽象类里定义的构造器只能是普通构造器。



除此之外, 当使用 static 修饰一个方法时, 表明这个方法属于该类本身, 即通过类就可调用该方法, 但如果该方法被定义成抽象方法, 则将导致通过该类来调用该方法时出现错误 (调用了一个没有方法体的方法肯定会引起错误)。因此 static 和 abstract 不能同时修饰某个方法, 即没有所谓的类抽象方法。

★ 注意: ★

static 和 abstract 并不是绝对互斥的, static 和 abstract 虽然不能同时修饰某个方法, 但它们可以同时修饰内部类。



★ 注意: ★

abstract 关键字修饰的方法必须被其子类重写才有意义, 否则这个方法将永远不会有方法体, 因此 abstract 方法不能定义为 private 访问权限, 即 private 和 abstract 不能同时修饰方法。



6.5.2 抽象类的作用

从前面的示例程序可以看出, 抽象类不能创建实例, 只能当成父类来被继承。从语义的角度来看, 抽象类是从多个具体类中抽象出来的父类, 它具有更高层次的抽象。从多个具有相同特征的类中抽象出一个抽象类, 以这个抽象类作为其子类的模板, 从而避免了子类设计的随意性。

抽象类体现的就是一种模板模式的设计, 抽象类作为多个子类的通用模板, 子类在抽象类的基础上进行扩展、改造, 但子类总体上会大致保留抽象类的行为方式。

如果编写一个抽象父类, 父类提供了多个子类的通用方法, 并把一个或多个方法留给其子类实现, 这就是一种模板模式, 模板模式也是十分常见且简单的设计模式之一。例如前面介绍的 `Shape`、`Circle` 和 `Triangle` 三个类, 已经使用了模板模式。下面再介绍一个模板模式的范例, 在这个范例的抽象父类中, 父类的普通方法依赖于一个抽象方法, 而抽象方法则推迟到子类中提供实现。

程序清单: codes\06\6.5\SpeedMeter.java

```
public abstract class SpeedMeter
{
    // 转速
    private double turnRate;
    public SpeedMeter()
    {
    }
    // 把返回车轮半径的方法定义成抽象方法
    public abstract double getRadius();
    public void setTurnRate(double turnRate)
    {
        this.turnRate = turnRate;
    }
    // 定义计算速度的通用算法
    public double getSpeed()
    {
        // 速度等于 车轮半径 * 2 * PI * 转速
        return java.lang.Math.PI * 2 * getRadius() * turnRate;
    }
}
```

上面程序定义了一个抽象的 `SpeedMeter` 类(车速表), 该表里定义了一个 `getSpeed()` 方法, 该方法用于返回当前车速, `getSpeed()` 方法依赖于 `getRadius()` 方法的返回值。对于一个抽象的 `SpeedMeter` 类而言, 它无法确定车轮的半径, 因此 `getRadius()` 方法必须推迟到其子类中实现。

下面是其子类 `CarSpeedMeter` 的代码, 该类实现了其抽象父类的 `getRadius()` 方法, 既可创建 `CarSpeedMeter` 类的对象, 也可通过该对象来取得当前速度。

程序清单: codes\06\6.5\CarSpeedMeter.java

```
public class CarSpeedMeter extends SpeedMeter
{
    public double getRadius()
    {
        return 0.28;
    }
    public static void main(String[] args)
    {
        CarSpeedMeter csm = new CarSpeedMeter();
        csm.setTurnRate(15);
        System.out.println(csm.getSpeed());
    }
}
```

`SpeedMeter` 类里提供了速度表的通用算法, 但一些具体的实现细节则推迟到其子类 `CarSpeedMeter` 类中实现。这也是一种典型的模板模式。

模板模式在面向对象的软件中很常用, 其原理简单, 实现也很简单。下面是使用模板模式的一些简单规则。

➤ 抽象父类可以只定义需要使用的某些方法, 把不能实现的部分抽象成抽象方法, 留给其子类去

实现。

- 父类中可能包含需要调用其他系列方法的方法，这些被调方法既可以由父类实现，也可以由其子类实现。父类里提供的方法只是定义了一个通用算法，其实现也许并不完全由自身实现，而必须依赖于其子类的辅助。

6.6 Java 8 改进的接口

抽象类是从多个类中抽象出来的模板，如果将这种抽象进行得更彻底，则可以提炼出一种更加特殊的“抽象类”——接口（interface），接口里不能包含普通方法，接口里的所有方法都是抽象方法。Java 8 对接口进行了改进，允许在接口中定义默认方法，默认方法可以提供方法实现。

6.6.1 接口的概念

读者可能经常听说接口，比如 PCI 接口、AGP 接口等，因此很多读者认为接口等同于主机板上的插槽，这其实是一种错误的认识。当说 PCI 接口时，指的是主机板上那个插槽遵守了 PCI 规范，而具体的 PCI 插槽只是 PCI 接口的实例。

对于不同型号的主机板而言，它们各自的 PCI 插槽都需要遵守一个规范，遵守这个规范就可以保证插入该插槽里的板卡能与主机板正常通信。对于同一个型号的主机板而言，它们的 PCI 插槽需要有相同的数据交换方式、相同的实现细节，它们都是同一个类的不同实例。图 6.4 显示了这种抽象过程。

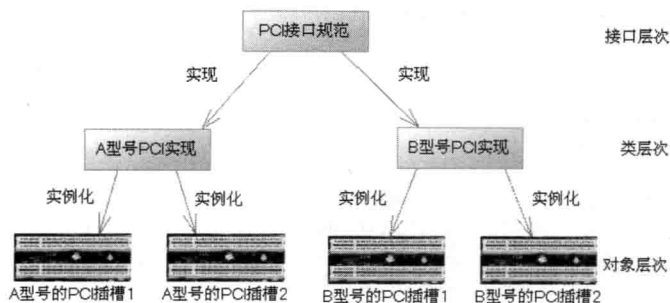


图 6.4 接口、类和实例的抽象示意图

从图 6.4 可以看出，同一个类的内部状态数据、各种方法的实现细节完全相同，类是一种具体实现体。而接口定义了一种规范，接口定义了某一批类所需要遵守的规范，接口不关心这些类的内部状态数据，也不关心这些类里方法的实现细节，它只规定这批类里必须提供某些方法，提供这些方法的类就可满足实际需要。

可见，接口是从多个相似类中抽象出来的规范，接口不提供任何实现。接口体现的是规范和实现分离的设计哲学。

让规范和实现分离正是接口的好处，让软件系统的各组件之间面向接口耦合，是一种松耦合的设计。例如主机板上提供了 PCI 插槽，只要一块显卡遵守 PCI 接口规范，就可以插入 PCI 插槽内，与该主机板正常通信。至于这块显卡是哪个厂家制造的，内部是如何实现的，主机板无须关心。

类似的，软件系统的各模块之间也应该采用这种面向接口的耦合，从而尽量降低各模块之间的耦合，为系统提供更好的可扩展性和可维护性。

因此，接口定义的是多个类共同的公共行为规范，这些行为是与外部交流的通道，这就意味着接口里通常是定义一组公用方法。

6.6.2 Java 8 中接口的定义

和类定义不同，定义接口不再使用 class 关键字，而是使用 interface 关键字。接口定义的基本语法如下：

```

[修饰符] interface 接口名 extends 父接口 1, 父接口 2...
{
    零个到多个常量定义...
}
    
```

```
    零个到多个抽象方法定义...  
    零个到多个内部类、接口、枚举定义...  
    零个到多个默认方法或类方法定义...  
}
```

对上面语法的详细说明如下。

- 修饰符可以是 **public** 或者省略, 如果省略了 **public** 访问控制符, 则默认采用包权限访问控制符, 即只有在相同包结构下才可以访问该接口。
- 接口名应与类名采用相同的命名规则, 即如果仅从语法角度来看, 接口名只要是合法的标识符即可; 如果要遵守 Java 可读性规范, 则接口名应由多个有意义的单词连缀而成, 每个单词首字母大写, 单词与单词之间无须任何分隔符。接口名通常能够使用形容词。
- 一个接口可以有多个直接父接口, 但接口只能继承接口, 不能继承类。



提示: 在上面语法定义中, 只有在 Java 8 以上的版本中才允许在接口中定义默认方法、类方法。关于内部类、内部接口、内部枚举的知识, 将在下一节详细介绍。

由于接口定义的是一种规范, 因此接口里不能包含构造器和初始化块定义。接口里可以包含成员变量 (只能是静态常量)、方法 (只能是抽象实例方法、类方法或默认方法)、内部类 (包括内部接口、枚举) 定义。

对比接口和类的定义方式, 不难发现接口的成员比类里的成员少了两种, 而且接口里的成员变量只能是静态常量, 接口里的方法只能是抽象方法、类方法或默认方法。

前面已经说过了, 接口里定义的是多个类公共的公共行为规范, 因此接口里的所有成员, 包括常量、方法、内部类和内部枚举都是 **public** 访问权限。定义接口成员时, 可以省略访问控制修饰符, 如果指定访问控制修饰符, 则只能使用 **public** 访问控制修饰符。

对于接口里定义的静态常量而言, 它们是接口相关的, 因此系统会自动为这些成员变量增加 **static** 和 **final** 两个修饰符。也就是说, 在接口中定义成员变量时, 不管是否使用 **public static final** 修饰符, 接口里的成员变量总是使用这三个修饰符来修饰。而且接口里没有构造器和初始化块, 因此接口里定义的成员变量只能在定义时指定默认值。

接口里定义成员变量采用如下两行代码的结果完全一样。

```
// 系统自动为接口里定义的成员变量增加 public static final 修饰符  
int MAX_SIZE = 50;  
public static final int MAX_SIZE = 50;
```

接口里定义的方法只能是抽象方法、类方法或默认方法, 因此如果不是定义默认方法, 系统将自动为普通方法增加 **abstract** 修饰符; 定义接口里的普通方法时不管是否使用 **public abstract** 修饰符, 接口里的普通方法总是使用 **public abstract** 来修饰。接口里的普通方法不能有方法实现 (方法体); 但类方法、默认方法都必须有方法实现 (方法体)。



注意: 接口里定义的内部类、内部接口、内部枚举默认都采用 **public static** 两个修饰符, 不管定义时是否指定这两个修饰符, 系统都会自动使用 **public static** 对它们进行修饰。



下面定义一个接口。

程序清单: codes\06\6.6\Output.java

```
package lee;  
public interface Output  
{  
    // 接口里定义的成员变量只能是常量  
    int MAX_CACHE_LINE = 50;  
    // 接口里定义的普通方法只能是 public 的抽象方法  
    void out();  
    void getData(String msg);  
}
```



```
// 在接口中定义默认方法, 需要使用 default 修饰
default void print(String... msgs)
{
    for (String msg : msgs)
    {
        System.out.println(msg);
    }
}
// 在接口中定义默认方法, 需要使用 default 修饰
default void test()
{
    System.out.println("默认的 test() 方法");
}
// 在接口中定义类方法, 需要使用 static 修饰
static String staticTest()
{
    return "接口里的类方法";
}
```

上面定义了一个 Output 接口, 这个接口里包含了一个成员变量: MAX_CACHE_LINE。除此之外, 这个接口还定义了两个普通方法: 表示取得数据的 getData()方法和表示输出的 out()方法。这就定义了 Output 接口的规范: 只要某个类能取得数据, 并可以将数据输出, 那它就是一个输出设备, 至于这个设备的实现细节, 这里暂时不关心。

Java 8 允许在接口中定义默认方法, 默认方法必须使用 default 修饰, 该方法不能使用 static 修饰, 无论程序是否指定, 默认方法总是使用 public 修饰——如果开发者没有指定 public, 系统会自动为默认方法添加 public 修饰符。由于默认方法并不没有 static 修饰, 因此不能直接使用接口来调用默认方法, 需要使用接口的实现类的实例来调用这些默认方法。

Java 8 允许在接口中定义类方法, 类方法必须使用 static 修饰, 该方法不能使用 default 修饰, 无论程序是否指定, 类方法总是使用 public 修饰——如果开发者没有指定 public, 系统会自动为类方法添加 public 修饰符。类方法可以直接使用接口来调用。

接口里的成员变量默认是使用 public static final 修饰的, 因此即使另一个类处于不同包下, 也可以通过接口来访问接口里的成员变量。例如下面程序。

程序清单: codes\06\6.6\OutputFieldTest.java

```
package yeeku;
public class OutputFieldTest
{
    public static void main(String[] args)
    {
        // 访问另一个包中的 Output 接口的 MAX_CACHE_LINE
        System.out.println(lee.Output.MAX_CACHE_LINE);
        // 下面语句将引发"为 final 变量赋值"的编译异常
        // lee.Output.MAX_CACHE_LINE = 20;
        // 使用接口来调用类方法
        System.out.println(lee.Output.staticTest());
    }
}
```

从上面 main()方法中可以看出, OutputFieldTest 与 Output 处于不同包下, 但可以访问 Output 的 MAX_CACHE_LINE 常量, 这表明该成员变量是 public 访问权限的, 而且可通过接口来访问该成员变量, 表明这个成员变量是一个类变量; 当为这个成员变量赋值时引发"为 final 变量赋值"的编译异常, 表明这个成员变量使用了 final 修饰。

✪ 注意: ✪

从某个角度来看, 接口可被当成一个特殊的类, 因此一个 Java 源文件里最多只能有一个 public 接口, 如果一个 Java 源文件里定义了一个 public 接口, 则该源文件的主文件名必须与该接口名相同。

