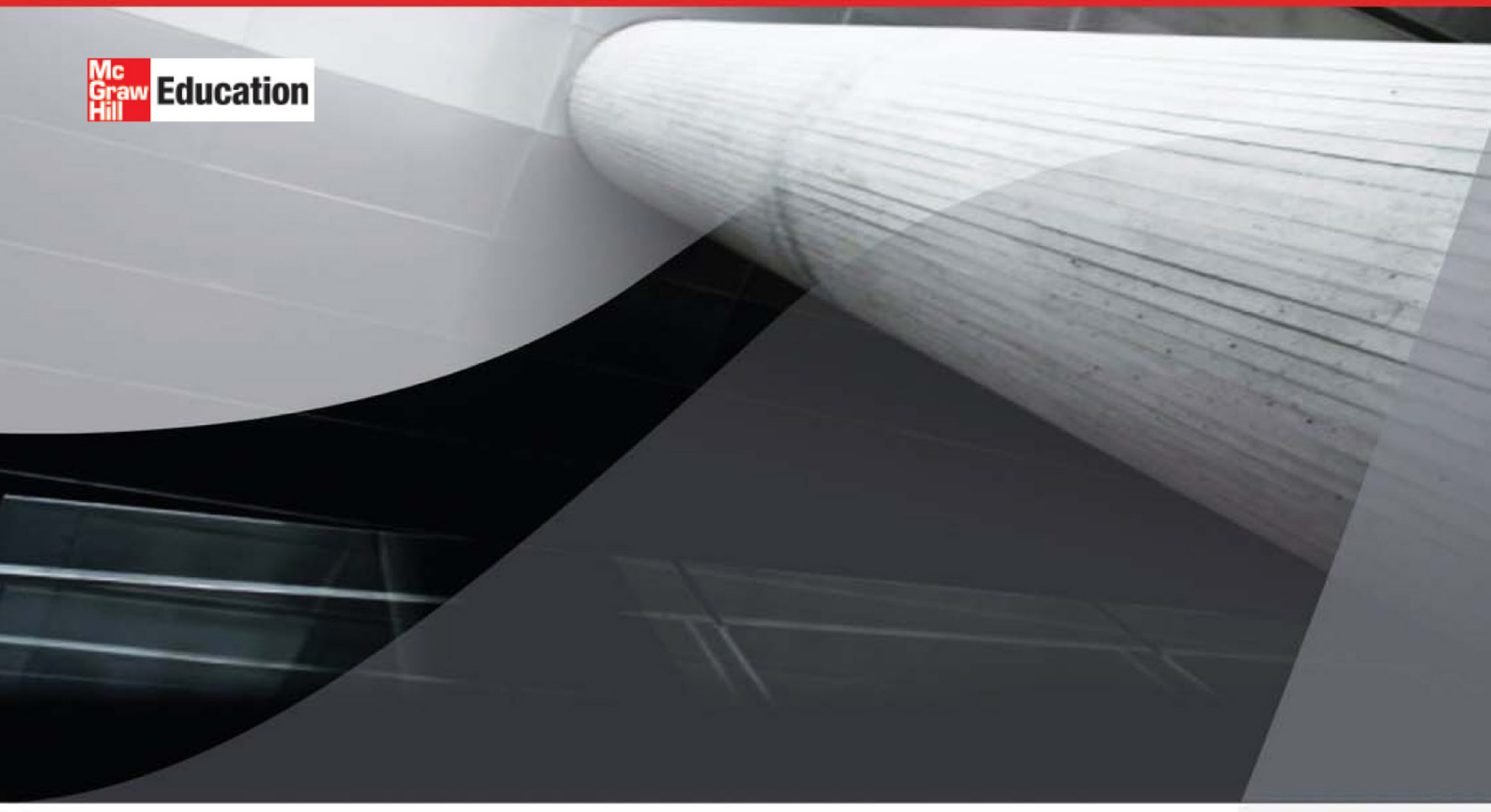
ORACLE



Java Programming

Java 7
编程高级进阶

向Java专家学习高级技能

[美] Poornachandra Sarang 著
曹如进 张方勇 译

清华大学出版社

Poornachandra Sarang
Java Programming
EISBN：978-0-07-163360-4

北京市版权局著作权合同登记号 图字：01-2012-7352

图书在版编目(CIP)数据

Java 7编程高级进阶/(美)萨朗(Sarang, P.) 著；曹如进，张方勇 译. —北京：清华大学出版社，2013.2
书名原文：Java Programming

ISBN 978-7-302-31362-5

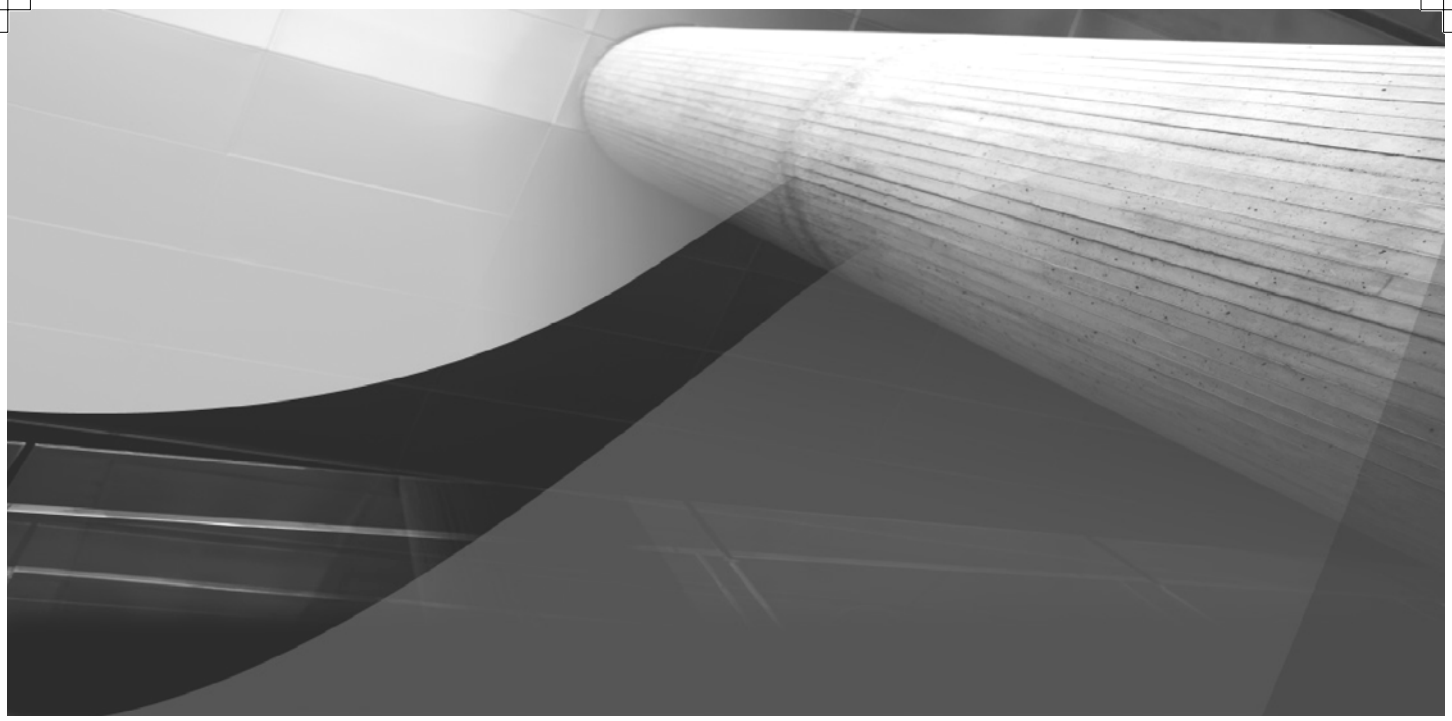
Ⅰ. ①J… Ⅱ. ①萨… ②曹… ③张… Ⅲ. ①JAVA语言—程序设计—教材 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第013983号

责任编辑：王 军 李维杰
装帧设计：牛静敏
责任校对：邱晓玉
责任印制：

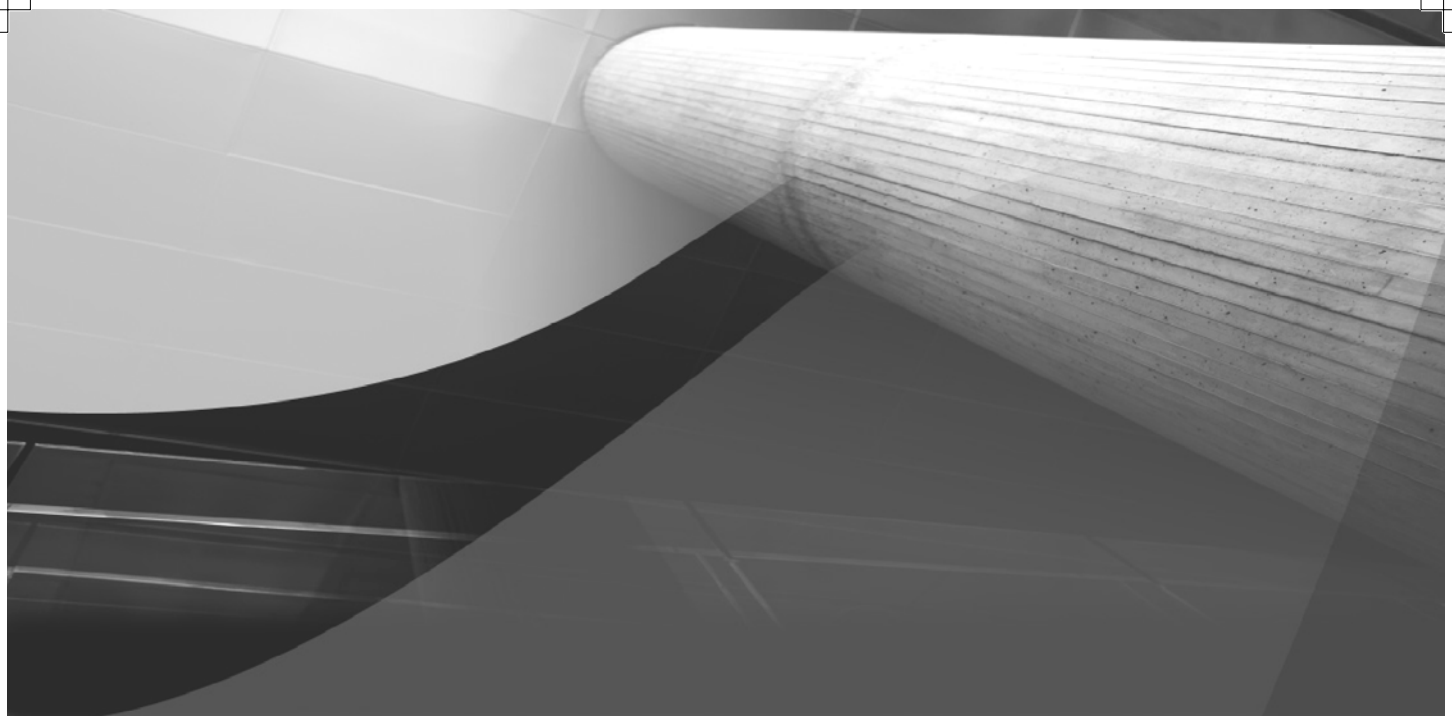
出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦A座 邮 编：100084
社 总 机：010-62770175 邮 购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经 销：全国新华书店
开 本：185mm×260mm **印 张**：35.5 **字 数**：886千字
版 次：2013年2月第1版 **印 次**：2013年2月第1次印刷
印 数：1~3000
定 价：78.00元

产品编号：



作 者 简 介

Poornachandra Sarang从1996年开始就已成为一名Java程序员。在过去的16年里，Sarang博士开展了多场基于Sun Microsystems官方课程的培训项目、讲师认证考核以及企业培训，他撰写了关于Java和其他类似话题的多本书籍与期刊文章，是许多国际会议的受邀演讲嘉宾，包括最近的“JavaOne 2011大会”。



致　　谢

自从1996年拥抱Java以来，我为Sun Microsystems(现为Oracle)公司开展了一些讲师培训计划和讲师认证考核。这些年里，我一直想写一本Java编程方面的书，为那些想要精通Java的专业人员提供准确、真实的语言知识。然而这一目标的优先级似乎一直较低，直到McGraw-Hill出版社对出版此书感兴趣，我的愿望才得以实现。首先，我要感谢McGraw-Hil出版社的组稿编辑。同时也感谢编辑和制作团队，感谢你们使我写书的梦想成真。这里，我想特别提及团队中与我直接交流的一些人的名字。我要感谢来自统稿部门的Joya Anthony、Jane Brownlow和Megg Morlin，同时感谢来自编辑与制作团队的Bart Reed、Harleen Chopra和Sapna Rastogi。还有，我要感谢Jody McKenzie，感谢她在制作和创造这本优秀且技术权威的书籍的过程中，帮助我解决了许多问题。

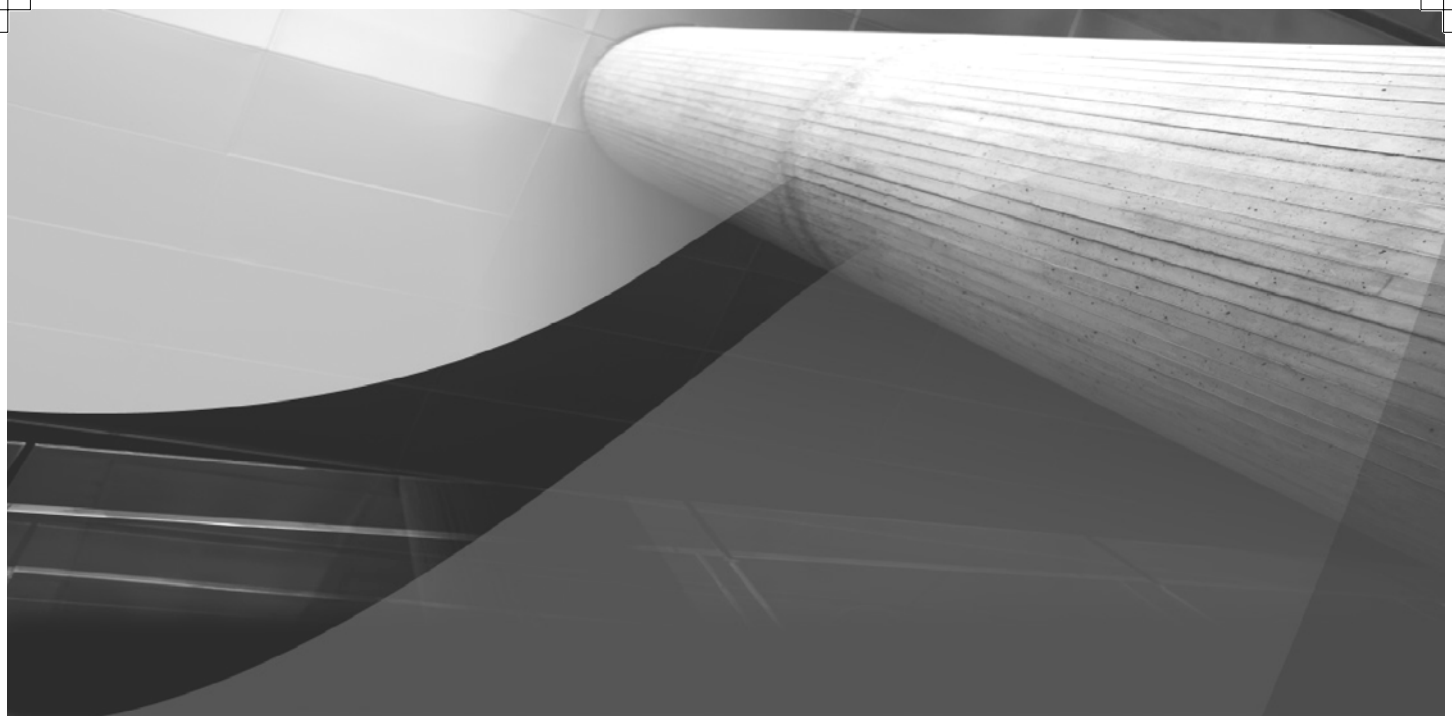
同时，我要感谢Oracle大家庭的成员。没有他们的帮助，这本书不可能达到今天的技术准确度。首先，我要感谢我的老朋友Pratik Parekh，他目前是Oracle公司Fusion中间件产品管理部门的总监。Pratik帮我介绍Oracle公司合适的联系人并与他们建立联系。我要感谢John Pampuch(Oracle公司Java虚拟机技术总监)，感谢他对我的请求立刻作出反应，并为我提供了全世界最好的技术审稿人。这名审稿人不是别人，正是Oracle公司客户端软件的首席架构师Danny Coward。Danny的评论如此见解深刻，以至于我重写了整本书。我要特别感谢Danny统一为修订后的手稿进行二次技术审核。有时，他甚至不厌其烦地指导我，为本书提供准确的技术信息。我必须承认，尽管我有多年的Java编程经验，但在许多地方也存在着误解，通过在本书写作的过程中不断与Danny沟通，这些误解都得到了澄清。Danny审校时非常仔细，甚至坚持严格遵守命名约定和遵守JLS标准的

代码格式化，从而使本书注意到了一些细节，例如必需的空行以及字符之间的空格。没有他一丝不苟地进行审校，这本书可能永远没法达到这个级别的技术准确度。再次感谢你，Danny。书中可能的其他错误都是我个人造成的。

我还要特别感谢我的学生们，尤其是Ishita Patel，他仔细阅读了手稿，发现了许多问题和疏漏之处。Ishita还帮助开发和测试了书中的所有代码示例。我也要感谢Rashmi Singh，他为书中的一些章节提供了建设性的意见。我还必须提到Steven Suting，他帮助我纠正了书中所有的Java语言语法规范。

我还要感谢Vijay Jadhav，他帮忙创作了书中的插图并且负责格式化、组织和跟踪手稿。

最后，我衷心感谢John Pampuch，感谢他欣然同意为本书作序。



序　言

Java从诞生之初，就成为我事业与生命中的一部分。我从1996年开始使用Java 1.0，并于当年年底加入了Sun公司的Java团队。尽管刚开始困难重重，但还是发生了一些特别的事情：为了创建能改变人们看待软件方式的平台，许多杰出的人才提出了数不清的想法。许多现有的公司采用Java作为一门核心技术，而许多新成立的公司则要么加入 Java 生态系统中，要么将Java用在另一领域。

Java不只是一门语言，并且还是一种由多个部分构成的平台，这些部分包括Java语言、JVMC(Java Virtual Machine，Java虚拟机)核心库以及许多其他组件。这些组件构成了一门灵活、强大且通用的技术，并用在了各种类型的应用程序中。

Java不只是平台，并且还是自身的生态系统。构建于Java平台之上的有IDE、监测与管理工具、库、应用程序服务器、测试与调试工具、开发工具，当然还有几乎所有类型的应用程序。

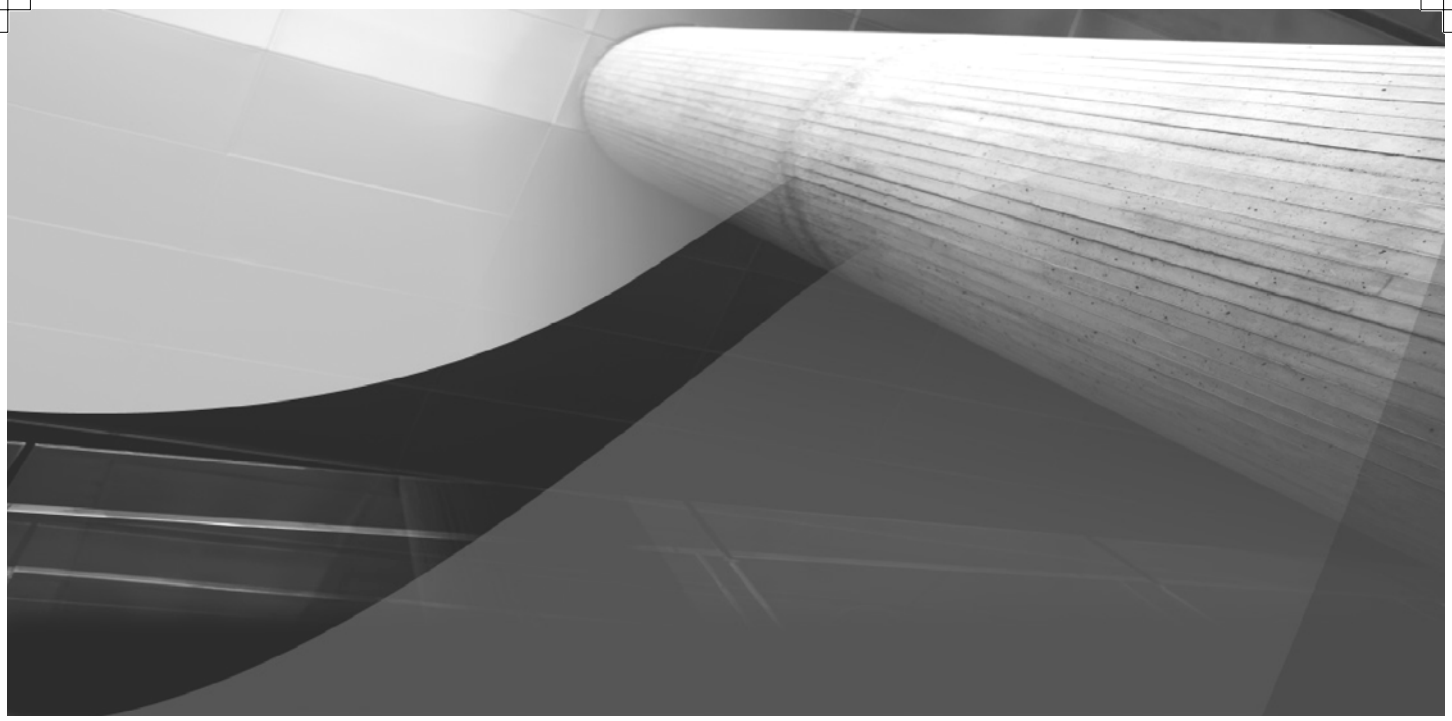
在我使用Java的这些年里，开发人员的数量有了很大增长。在20世纪90年代末，开发人员的数量有数十万。而如今，Oracle报告称有超过900万的开发人员在使用Java。Java的部署数量，算上消费者设备上的Java ME，要以数十亿计。

Java一直在进步。从某些指标上来看，Oracle JDK的性能从JDK 5起有了将近3倍的提升。这些提升反映了整个平台的变化。

除了性能改善以外，Java在每一次的发布版本中都会添加一系列功能特性。Java的演变过程包含了许多优秀技术专家的指导，他们中有许多人花费了很大心思关注维护兼容性与一致性。

即使这样，Java主要版本的发布，加上偶尔有些晦涩的特性，使得像本书这样的书籍对于开发人员至关重要。类似这样的书籍对数百万开发人员采用Java有着巨大的影响，包括我自己也是如此。很庆幸Poornachandra Sarang具有这份耐心和技术能力，感谢他花费时间有效地传达Java平台错综复杂的细节。

John Pampuch
Oracle公司Java虚拟机技术总监



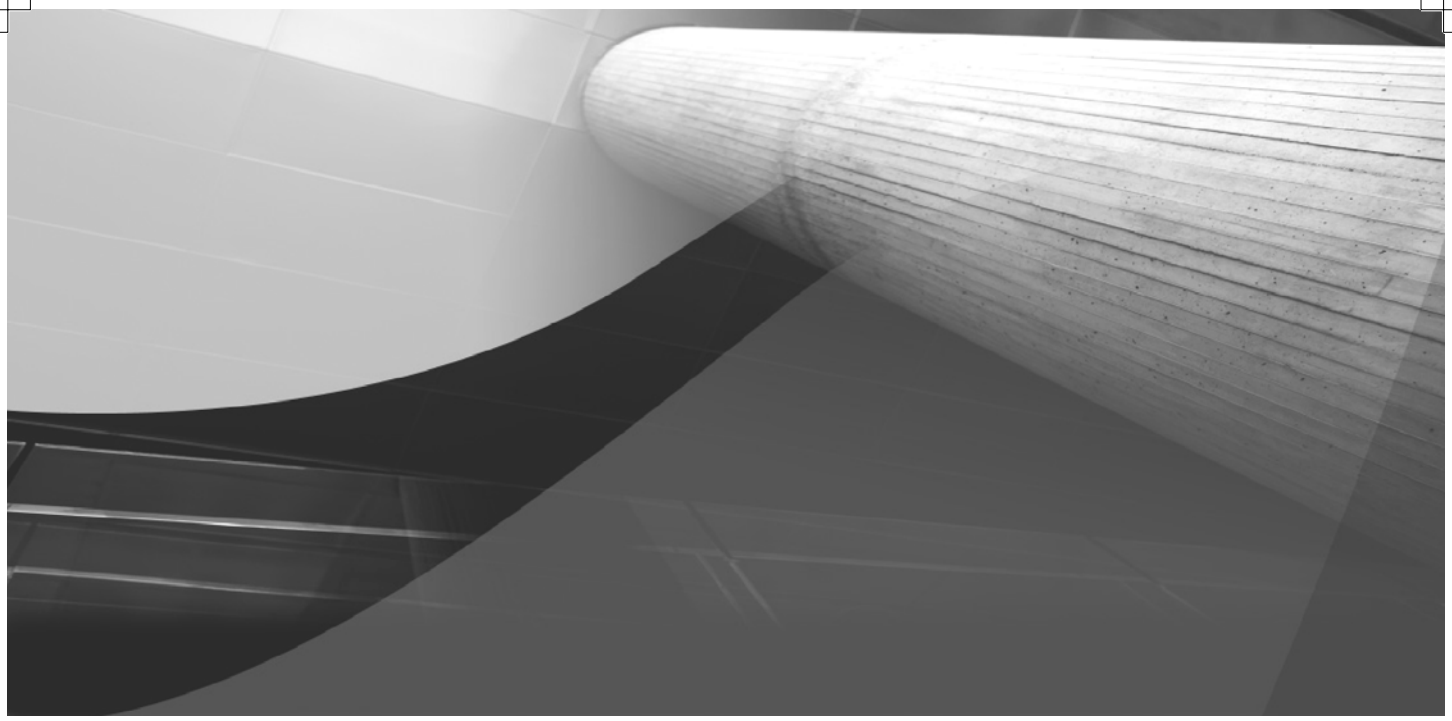
译 者 序

自从1996年发布以来，Java就一直风靡于世界范围内的开发人群中。使用Java开发的应用程序小至基于Web的Java applet，大至分布式企业级应用。与时俱进的新特性使得Java成为一门越来越流行的程序设计语言。

本书是为数不多的系统介绍Java的一本技术书籍。作者Poornachandra Sarang和技术审稿人John Pampuch，都是Java领域的领军人物，他们知道如何写书，更知道如何介绍专业技术。本书严谨、清晰、深入、细致地介绍了Java方方面面的技术，包括集合框架、I/O编程、泛型、Swing、多线程和并发工具、网络编程等，还介绍了Java SE 7新添加的特性，比如带资源的try语句、Fork/Join框架等。另外，此书还在互联网上免费发布了3章内容，分别介绍Java概况、Java语言中的基本概念以及Java中的运算符，欢迎广大读者下载阅读。

本书中文版得以发行，首先要感谢本书的作者，是他为我们撰写了一本好书。其次，要感谢清华大学出版社引进了本书的中文版权，使我们获得了翻译此书的机会。此外，要特别感谢清华大学出版社的李阳女士，她以巨大的热情和高度的责任心为本书的出版做了大量繁琐细致的出版业务工作。还要感谢赵宇、张霄、王玉等人在本书翻译过程中给予的大力帮助。

本书章节经过精心安排，内容承上启下，希望读者在阅读过程中能够亲手实践代码，从而更好、更充分地理解和掌握本书介绍的Java特性。本书由曹如进和张方勇翻译。尽管我们在翻译过程中力图信、达、雅，但限于自身水平，本书中文版必定存在错误、不足和不当之处，恳请广大读者将意见和建议发至drizzlecrj@gmail.com，我们不胜感激。



前　言

你手头的这本书由一名Java编程老手所著，经由Sun Microsystems(现为Oracle)公司审校。这本书将深入介绍Java语言的特性，包括Java SE 7中最新添加的特性。不论你是刚刚接触Java编程，还是考取Java证书的学生，抑或使用其他语言的专业程序员，你都会发现这本书在将你带入Java领域的过程中非常有帮助。

本书组织结构

本书共分24章，刚开始的前3章发布在互联网上(www.oraclepressbooks.com)。前3章介绍了传统的Hello World程序与基本的Java语法。它们之所以放在互联网上，是因为本书的大部分读者可能是专业程序员，他们希望直接跳入语言的高级话题。在第1章初步介绍完Java以及Java的演变历史后，我们在第2章直接跳到Java数组。一直到第9章，我们对Java语言进行了深入剖析，主要关注Java中的面向对象与许多复杂的细节，比如对象的创建过程，创建继承层次，final与static修饰符的恰当使用，public、protected与private修饰符的使用场景，定义内嵌类、本地类及匿名类，最后介绍如何在Java程序中有效地处理异常。

之后，我们暂缓介绍Java语言语法，开始讨论Java库，通过一些实际的代码示例介绍I/O编程。接下来，我们将介绍一些更加高级的语法话题，比如枚举、自动装箱、注解以及泛型。书中剩下的部分关注现实世界中的应用程序开发，包括构建GUI程序、事件与用户手势处理、理解数据结构、线程与网络编程以及一些重要的Java类。

本书所有章节的顺序都经过精心安排以避免前向引用。因此，本书可按章进行阅读。如果你熟悉Java并且希望深入了解某个特定话题，可以只阅读感兴趣的相关章节。

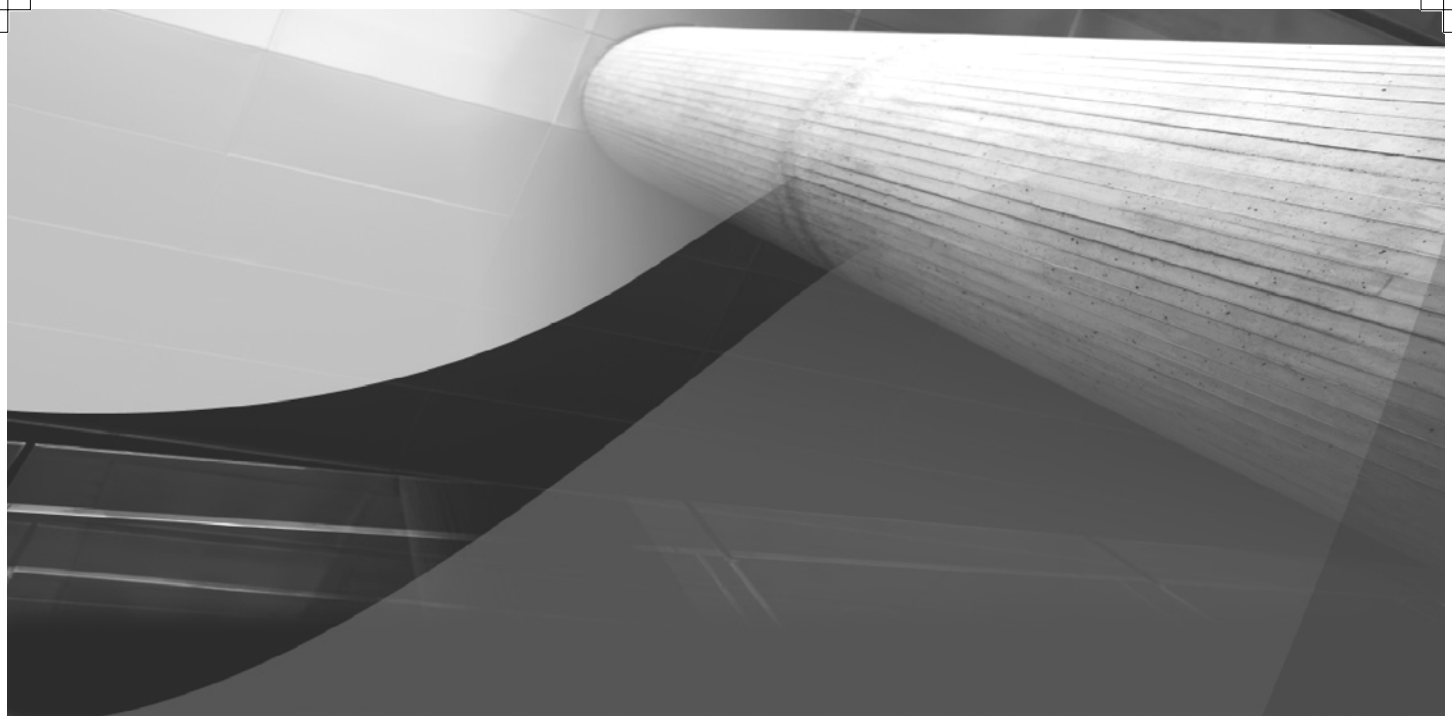
章节介绍

本书共包含精心组织的21章内容，另外还有关于Java语法的3章内容发布在www.oraclepressbooks.com上。以下是本书内容提要：

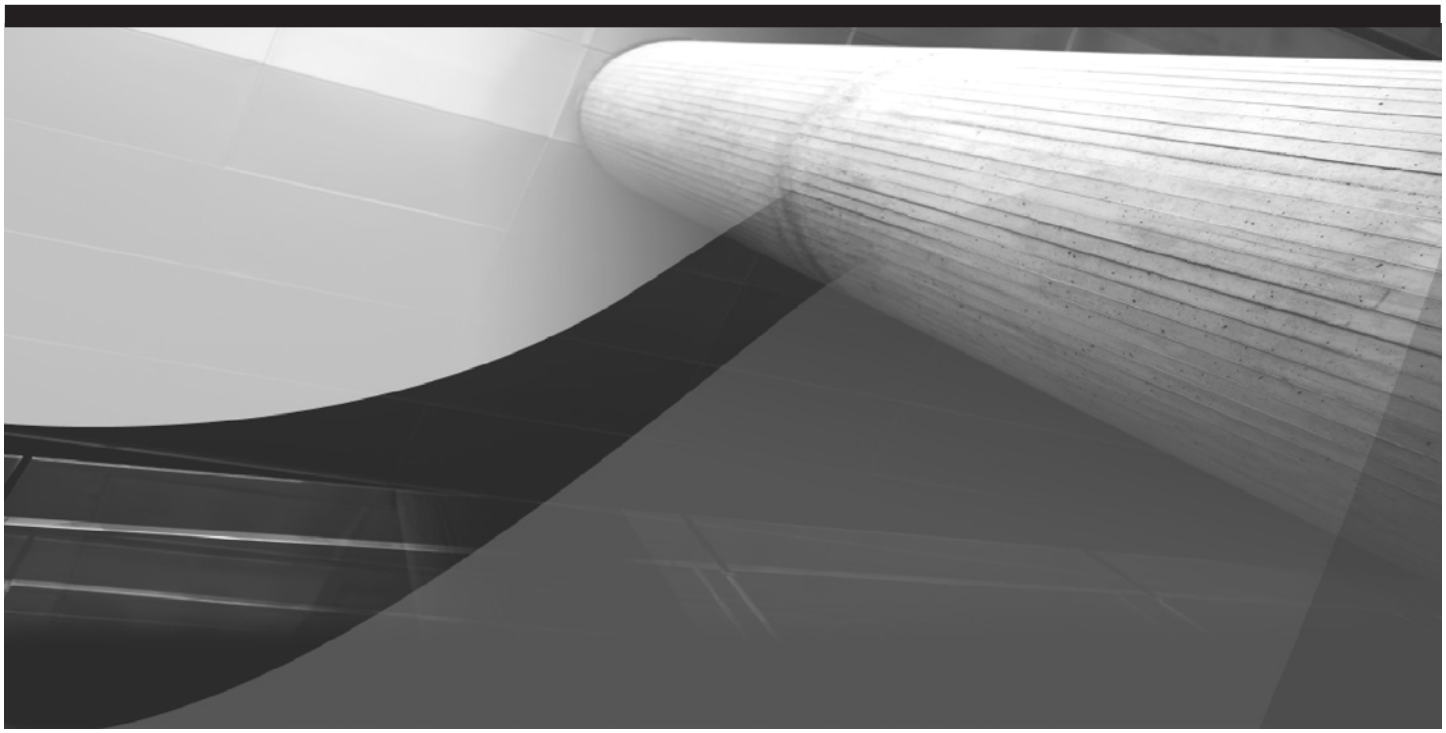
- 第1章简单介绍Java的历史，阐述为什么Java会被创建，什么是Java以及Java的一些主要特性。
- 第2章讨论如何声明和使用一维和多维数组。
- 第3章通过讨论什么是类来开始学习面向对象语言。这一章介绍面向对象语言的主要特性，如封装、继承与多态等。
- 第4章深入剖析Java面向对象中的继承特性。这一章会教你如何创建单一继承与多重继承。
- 第5章解释对象的创建过程以及对象创建过程中超类对象是如何构造的。
- 第6章通过讨论静态字段、方法与初始化器来带你深入面向对象编程的王国。
- 第7章通篇介绍Java类，包括内嵌类、本地类及匿名类。
- 第8章讨论Java中的异常处理。与前两章大量介绍Java语言的复杂特性相比，这一章会有所不同。
- 第9章与第10章介绍Java中的I/O编程，包括新的java.nio包。
- 第11章进一步讨论Java语言语法，并介绍枚举、自动装箱与注解。
- 第12章介绍泛型，这一章会让你对泛型的许多特性有更深入的技术理解。
- 第13章转向介绍Java中的实际应用程序开发，包括构建GUI应用程序，以及这类GUI应用程序中的事件处理方式。
- 第14章深入剖析创建复杂屏幕布局要用到的各种布局管理器。
- 第15章展示如何绘制图像以及处理用户手势。
- 第16章通篇讲解集合API，这是组织数据的一组重要API。
- 第17章是开始讲解线程编程的第一章。 这一章提供了关于JVM中线程实现的深度知识，并讨论基本的同步机制。
- 第18章讨论阻塞队列与同步器，如倒计数锁存、信号量等。
- 第19章讨论Callable、Future、Executor以及Java SE 7中最新引入的分支/合并(Fork/Join)框架。
- 第20章描述网络编程，这是Java应用程序非常重要的课题。
- 第21章通过讨论各式各样的类与API，为读者设立一条学习Java中丰富类的途径。

注意

放在网上的3章内容讨论了基本的Java语法，它们发布在www.oraclepressbooks.com上(读者也可以从本书中文版的支持网站www.tupwk.com.cn下载并阅读，或者参考《新手学Java 7编程(第5版)》(清华版，ISBN：978-7-302-29541-9)中的第1到3章内容)。这3章以传统的Hello World程序开篇，介绍了基本概念、操作符及控制流语句。它们还包含了Java SE 7中最新的Java语法。此外，这些章还讨论了许多容易漏掉的特性。



目　录



第1章

Java 简 介

自从1996年发布以来，Java语言就一直风靡于世界范围内的开发人员中。在本书写作之际，官方渠道(也就是Oracle)宣称世界范围内共有900万名活跃的Java开发人员。Java一直被广泛作为开发各种类型应用程序的一门语言接受，这些应用程序小至基于Web的Java applet，大至分布式企业级应用。你会发现Java既用在了小型嵌入式设备上，也用在了超大型的关键应用中。你会发现大学里的学生们在采用Java编写课程项目，同时也会发现业界的开发人员在使用Java开发商业项目，为政府开发可扩展应用程序，为银行开发对时间要求严格的可靠性应用程序以及为军队开发鲁棒性的关键应用程序。你会发现Java几乎用在了世界的每一个角落。

那么是什么原因让Java如此流行呢？Java是否仅仅是一门像Pascal、C和C++的语言呢？

Java是不是一种用于创建各种大小与复杂应用程序的工具呢？又是不是能囊括嵌入式乃至企业级应用程序运行的平台呢？在本章中，作者会尝试回答所有这些问题。你将了解到Java之所以流行的特别之处。

具体学习内容如下：

- 为什么Java会被创建
- 什么是Java
- Java特性介绍
- Java演变过程

1.1 为什么使用Java

Java公开发布于1996年。当时，C++占据了大部分市场，被广泛用于创建许多类型的软件应用程序。就在C++拥有如此市场规模的情况下，几乎没有人想到在这个计算时代引入一门新的编程语言，但是Sun Microsystems想到了。他们推出Java是出于当时某种特殊的原因。当时，他们急需一种适合开发嵌入式设备应用程序的语言，便想到开发一门新语言。

然而，当时的语言存在着众多问题。C++的“大量消耗资源”一直为人们所熟知。C++开发人员需要自行管理内存。使用C++编写(比如在Visual Studio中)普通的“Hello World”程序并运行需要好几MB的内存。虽然C++生成了高度优化后的代码，然而运行代码的需求通常非常高。这是当时市场上的其他开发环境和运行时系统，如Borland C++和Turbo Pascal也存在的问题。因此，大家迫切需要一门能够生成短代码且在运行时环境中不会占用目标设备过多内存空间的语言。

为嵌入式设备编程同样需要可移植性，因为这些设备一般使用各种不同的CPU及体系架构。C++应用程序在每一种体系架构中的行为并不一致，因此通常需要为新设备重写代码。管理多种构建环境和代码库的复杂性成为支持多设备开发的最大挑战。Java就是为了解决这些问题应运而生的一门高级语言。Java通过引入虚拟机(VM)与可移植的字节码架构解决了多设备开发的问题。Sun Microsystems就这样打着“编写一次，到处运行”的标语适宜地挺进了市场。

1.2 什么是Java

刚才介绍了创建Java的原因。Java作为一门编程语言，起初与C++、Smalltalk有着许多相似点，这也是为了吸引C++开发人员。Java被设计成一门简单且高级的语言，移除了内存管理、安全这类复杂特性，转而将这些任务交给虚拟机管理。

注意

如今的Java不再是一门简单的程序语言。Java既是平台，也是开发和运行时环境。到本章结束时，你将可以从如今的计算环境角度进一步了解Java。

Java的创建者希望为开发人员打造一条简单的迁移路线，因为当时大部分的开发人员都非常熟悉C++。尽管Java与C++有着许多相似性，例如它们都是面向对象编程语言，但是创建者为这门新语言移除了许多缺陷以及不太常用的特性。这不仅让Java体积更小，还附带提供了许多好处，例如让Java变得更加鲁棒、简单和可移植。

在介绍Java语言特性之前，让我们先看看Java的体系架构以及“虚拟机”的概念。

1.3 Java虚拟机

Java编译器将Java源程序翻译成所谓的字节码(bytecode)，这有点类似于C++或其他语言编译器生成的OBJ代码。唯一的不同之处在于：大部分编译器会为实际的CPU生成对象代码，而Java编译器生成的字节码则由“伪CPU”指令构成。换句话说，C++的OBJ代码包含了针对Intel 80x86 CPU、Motorola 68xxx CPU甚至Sparc工作站的指令集，而Java字节码指令集针对的CPU在现实中并不存在。Sun Microsystems所做的工作是在内存中创建虚拟的CPU，并为这个虚拟的CPU设计一套指令集，之后运行时在内存中进行仿真。

Java虚拟机(Java Virtual Machine，JVM)正是仿真了前面提到的虚拟CPU，为Java可执行文件(字节码)提供了运行时环境。另外还提供了字节码解释器与验证器，其中，验证器用于在翻译实际CPU指令以及运行前确认字节码的有效性。此外，JVM还有一些模块用于安全、内存、线程管理等其他用途。顾名思义，JVM本质上是能够运行Java可执行文件的机器。JVM及其相关核心组件见图1-1。

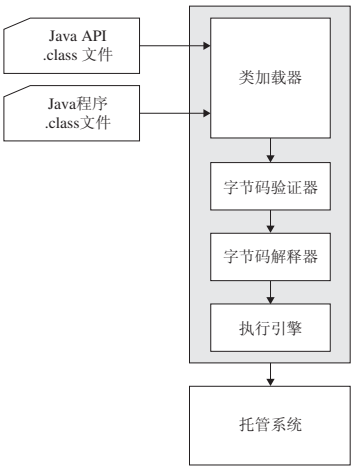


图1-1 Java虚拟机

Java编译器将生成的字节码存储在后缀为.class的文件中。你的程序也会用到Java开发工具包(Java Development Kit，JDK)提供的一些.class文件。JVM中的类加载器会在内存中加载这些.class文件。除了加载程序的.class文件之外，JVM还会加载一些程序在运行时需要的其他库。JVM会将加载的类提交并进行确认，进而确保它们不包含任何伪CPU指令集中未定义的指令。内部的.class文件无须经过这类验证。如果你的应用程序字节码包含无效指令，JVM会拒绝执行，并将它们从内存中卸载。在代码被验证后，内置的解释器会将它们转换为机器码。之后，代码会移交给执行单元以在主机中运行。

在理解了Java的体系架构这一基础之上，你将会欣赏到Java的诸多特性。

1.4 Java特性

Java提供了许多其他语言没有的好处。Java小巧、简单且面向对象，既可以当作编译语言，

也可以当作解释语言。Java编译器生成的可执行文件可以运行在任何提供JVM的平台上。Java平台本身是鲁棒且安全的。与众多其他语言一样，Java支持多线程，且表示形式更为简单。最后，与某些其他语言一样，Java天生是动态语言。下面我们将详细介绍每一个Java特性。

1.4.1 体积小

之前我们说过，创建Java背后的主要动机是为了给嵌入式系统编程开发一门语言。Java生成的可执行代码非常小；一般来说，“Hello World”应用程序的大小只有几个字节。如果同C++编译器生成的类似代码相比，后者的大小会轻松达到KB级别，此外后者在运行时还需要大量内存。运行Java编译后代码的运行时环境一般所需的内存空间少于1MB。如果再同运行C++应用程序相比，后者不仅需要MFC(Microsoft Foundation Classes，微软基础类库)或OWL(Borland’s Object Windowing Library，Borland对象窗口库)中嵌入的上百MB空间的代码，还需要成熟的操作系统及硬件才能部署。尽管这一点对于20世纪的独立应用程序或基于GUI的应用程序没错，但是微软目前的.NET平台提供了一套类似于Java及其可执行文件的体系架构，并且对运行时环境的需求与Java是具有可比性的。

注意

MFC与OWL在Java推出的时代都是非常流行的库。

1.4.2 简单易学

作为一门面向对象语言，Java简单易学。人们常说，如果没有学过像C和Pascal一样的面向过程语言，那么就直接跳过它们去学习面向对象语言。学习过面向过程语言的人通常会觉得迁移到面向对象语言很困难，因为学习时可能需要忘掉一些之前学习过的知识。Java非常简单易学， 我们推荐任何计算机科学课程都挑选Java作为首门编程语言。

1.4.3 面向对象

Java的重要特性是面向对象。C++源自C语言，允许声明全局变量，这意味着变量可以声明在任何对象范围之外，更确切地说，是可以声明在任何类定义之外。这违背了封装的准则——封装是面向对象语言的重要特性之一(我们会在第3章介绍面向对象编程)。Java不允许声明全局变量。类似的，Java中并没有像C和C++中的结构与联合，因为这些数据结构会将成员默认标记为公开，这违背了面向对象的准则。Java语言中这些特性的缺失使得Java成为一门更好的面向对象语言。

在Java中，整个代码包含的仅仅是完整的封装类。你可能想问，基础数据类型在Java中是否也以对象形式表示呢？为了保证效率，Java将基础数据类型声明为非对象；尽管如此，Java仍为这些原始类型提供了封装类，你可以使用这些封装对象保存基础数据类型条目。

1.4.4 兼具编译与解释特性

下面主要阐述Java与其他语言(比如C++、Pascal)之间的主要不同之处。类似C++这样的语言会将源程序编译成对象代码(一份.obj文件)。链接器通过将对象代码与目标库结合，将生成的对象代码转换为可执行文件(.exe文件)。当运行可执行文件时，操作系统中的加载器会在内存中加载可执行代码，解析出函数引用的绝对内存地址，继而执行代码。

编译和运行一段Java程序与之前的过程有着很大差别。之前也介绍过，Java编译器会将Java源程序翻译为所谓的字节码(面向虚拟CPU的指令集)。从Java源程序创建可执行文件并不包含链接器的过程。事实上，从Java源程序创建的可执行文件只有字节码。那么这类字节码是如何运行在实际CPU上的呢？当Java可执行文件(字节码)在机器上运行时，解释器会将每一个字节码指令转换为实际的CPU指令。这些指令会在接下来运行于实际CPU之上，因此编译和运行Java程序包含了编译与解释过程。正因为如此，Java被认为既是一门编译型语言，也是一门解释型语言。

由于Java字节码在运行时被解释为机器语言指令，Java代码的执行效率在性能上会略有损失。为了克服这个性能局限，大部分JVM都加入了大量的代码优化技巧，其中一个手段就是采用即时(Just-In-Time，JIT)编译，即在实际CPU开始程序执行前将字节码转换成机器语言代码。显然，JIT编译器没法像离线编译器(比如C++编译器)那样执行多种优化，因为需要实时地将字节码翻译为机器语言代码。如果JIT编译器试图进行过多优化，那么在用户从命令行敲下命令后需要很长时间才能启动程序。你需要在启动时间与吞吐量之间进行权衡。如今的VM采用了各种各样的优化技巧，大部分技巧偏向于选择更快速的启动而不是更大的吞吐量，它们还可以调节如何快速地适应这些优化。调优VM性能需要大量的专业知识。对于大部分应用而言，Java版本与C++版本一样快(有时更胜一筹)。

1.4.5 平台无关

引入JVM的一大好处在于平台无关。在前面你已经看到，JVM为Java可执行文件提供了运行时环境。同样，你看到JVM是运行在目标环境中的软件程序。因此，只要目标机器拥有JVM，编译后的Java代码(字节码)就可以在该机器上运行，而无须改动二进制文件或源程序。这样看来，Java程序在二进制层次是平台无关的。只要目标平台提供Java虚拟机，任何编译后的Java程序就可以在无须任何改动的情况下运行在任意平台上。这对于应用开发人员来说无疑是福音，因为这样可以轻松地创建出可以运行在多个操作系统上的应用程序而无须修改任何代码。在Java推出之前，创建可移植应用对于多数开发人员来说都是一项非常艰苦的工作。如果这些开发人员没法将他们流行的应用程序移植到其他平台，就会丢失大量利润；即使能够移植，他们也需要为新设备重新编码。关于Java应用程序的平台无关性可以参见图1-2。

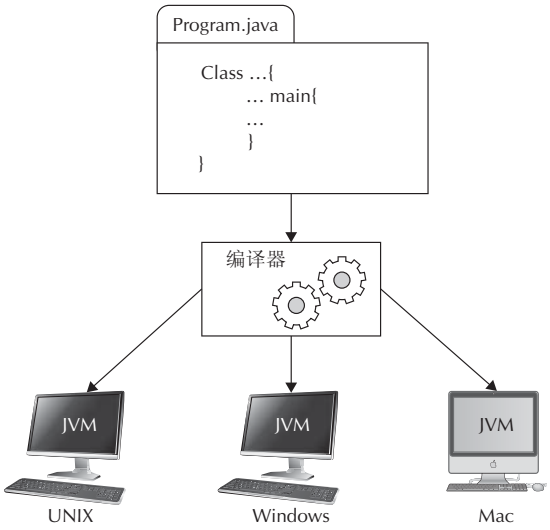


图1-2 演示Java的可移植性

如图1-2所示，Java编译器将给定的源程序编译成与机器无关的字节码。图1-2中的三台机器各自运行了一种不同的操作系统——分别是UNIX、Windows与Mac OS。三台机器运行的硬件可能完全不同。例如，UNIX可能运行在Sparc工作站上，Windows可能运行在Intel

80xx体系架构上，而Mac OS可能运行在诸如68xxx或Intel 80xx的体系架构上。尽管如此，三台机器上都有JVM。注意JVM本身是软件应用程序，并不是可移植的。因此，每个特定的目标平台都会有JVM。字节码在不同的JVM之间是可移植的，从而在支持JVM的平台上是可移植的。

1.4.6 鲁棒且安全

在体系架构中，JVM的引入同样能够帮助开发人员创建鲁棒且安全的Java应用程序。那么“鲁棒且安全”是什么意思呢？前面我们介绍过，字节码会由JVM解释，而JVM包含了字节码验证器。字节码验证器不仅会验证字节码的有效性，还会为每个运行其上的应用程序维护内存空间的完整性。如果字节码无效，JVM会简单地拒绝执行代码。类似的，如果字节码试图访问不属于当前应用程序的内存位置，JVM会拒绝执行代码。

那么怎样才能让编译后的Java应用程序拥有非法的字节码或无效的内存引用呢？字节码可能会被黑客在二进制编辑器中蓄意打开，或者可能会在网络传输过程中由于噪音的缘故遭到修改。相应的，内存引用甚至可能在运行时，在被JVM引用前被修改。所幸的是，JVM会在代码在真正的CPU上执行之前捕捉到这些有意或无意的错误，从而使应用程序非常鲁棒，而不会导致操作系统崩溃。

Java代码高度安全，这缘于Java并不能用于在目标机器上散播病毒。Java代码运行在JVM的严密监视之下，没有直接访问操作系统资源的权限。这类访问会通过JVM完成，JVM会为操作系统确保所有必需的安全性。

Java还丢弃了C++中的一种重要特性，那就是指针运算。Java不支持声明指针以及指针运算。

注意

Java支持的对象引用本质上就是指针，然而用户级别的指针运算是不受支持的。

缺少指针同样让Java程序更加鲁棒和安全。指针由于允许非法访问执行程序之外的内存空间而声名狼藉。它们还可以用于在系统中散播病毒。放弃指针同样有助于开发人员使用Java创建鲁棒且安全的应用程序。

Java既是一门静态类型语言，也是一门动态类型语言。静态是指编译时或源代码层面，而动态是指运行时或字节码层面。如果一门编程语言的类型检查发生在编译时而不是运行时，那么这门语言使用的是静态类型。在静态类型中，所有的表达式类型都在程序执行前的编译时确定。强类型是指在代码运行前实施类型系统规则。因此，Java被认为静态类型与强类型兼具。

另外一个让应用程序变得鲁棒的重要因素是，运行时针对内存及其他资源的正确分配与回收。在C++中，内存分配与回收是程序员的职责。开发人员可能会在使用完资源后忘记回收。最坏情况是回收了正在被应用程序其他部分使用的内存。这将会导致混乱以及难以跟踪的运行时错误。Java通过提供自动垃圾回收解决了这个问题。Java运行时承担了跟踪内存资源分配及释放(通常发生在资源不被应用程序中的任何部分引用时)的职责。自动垃圾回收让Java应用程序变得非常鲁棒。

Java程序之所以鲁棒，背后的另一个原因是异常处理机制。不管运行中的程序何时报错，都会被代码中提供的异常处理器处理。作为一名开发人员，你需要在程序中所有恰当

的位置提供异常处理器。Java有两种类型的异常处理机制——检查到的异常与未检查到的异常。编译器会让开发人员提供代码以处理检查到的异常，但不包括运行时异常(未检查到的异常)。Java强制性地对检查到的异常进行处理，这带来了更加鲁棒的应用程序以及更少的应用程序崩溃。

1.4.7 支持多线程

事实上，Java在1996年推出时就曾以多线程作为一大关键特性进行炒作(就好像其他语言不支持多线程一样)。尽管大部分其他语言都支持线程，但是Java提供了更为简单的线程语义。在C++中创建线程同在Java中创建线程一样简单，它们的不同之处在于共享资源的管理。为了在多个运行线程间共享资源，C++使用了若干概念，包括临界区和信号量。这些都是由操作系统本身提供的底层概念。为这些概念编码一般非常复杂。Java通过新设计的关键字synchronized隐藏了背后对这些概念的使用。你只需要声明一段代码或某个类为synchronized，之后Java运行时就会负责访问共享资源时产生的线程并发。这使得Java中的线程编程异常简单。

注意

Java早期版本中对开发人员隐藏的概念在新版本中都公开了。一般的同步概念，如信号量、倒计时锁存器(CountDownLatch)等，现在都可以被程序员访问了。

1.4.8 动态特性

Java中引入的另一个重要特性是动态特性。正如我们已经看到的，Java源程序会编译成字节码。这些字节码存储在后缀为.class的文件中。运行着的Java程序可以装载编译时未知的.class文件，了解类定义，实例化并按照自己的意图进行使用。这个过程在Java中叫做“内省(introspection)与反射(reflection)”。运行着的Java程序可以内省未知的类，了解其中定义的属性与操作，创建该类的实例，设置创建对象的属性，并调用创建对象的成员函数。Java还可以在运行时创建未知类型的对象数组。这就是Java的动态特性。

注意

Smalltalk从20世纪70年代就已具备这类动态特性，Objective-C从20世纪80年代具备这类特性，而现代语言(比如C#)也具备这类特性。

1.5 Java的演变过程

到目前为止，我们已经讨论了Java首个版本JDK 1.0的特性。随着时间的不断推移，Java也进行了许多扩展，并且增添了许多新的特性。你将在本书中学到其中的一些新特性。在这一节中，我们将讨论在过去15年里Java的演变过程。

虽然Java起初是设计用于嵌入式设备的一门程序语言，但现在已经发展为成熟的平台。下面我们将回顾Java演变至今的一些主要里程碑。Java以Java开发工具包(Java Development Kit，JDK)的形式发布。Java演化过程中的一些里程碑可通过JDK的版本号确定。下面将列出JDK的主要版本以及每个版本引入的特性。

1.5.1 JDK 1.0(1996年1月23日)：代号Oak

这是Java的首个版本，于1996年1月正式发布。在此之前，Java又名“Oak”,该名称主要用在Sun公司内部，用于嵌入式软件的开发。嵌入式设备需要跨越多种不同硬件的可移植性以及较少的空间占用。Java附带了这两种能力。JDK 1.0本身非常小，差不多212个类、8个用户包，以及Sun提供的一个用于调试的包。

注意

Java包为类与接口提供了逻辑分组功能。

因此，这个版本的Java仅提供了有限的能力，并且提供的库在当时没法与其他语言相提并论。java.awt(Abstract Windowing Toolkit，抽象窗口工具包)中提供的用户界面太过原始，甚至没有提供打印工具。尽管Java提供的功能集很少，但Java却在市场中迅速崛起，并且在推出的一年内变得非常流行。

Java成功的背后是因为当时互联网的兴起。那个时期的网页还不具备动态能力，网页只是静态的。Java applet为网页提供了动态内容以及交互能力。每份Netscape浏览器副本都会提供Java运行时。因此，开发人员通过编写Java applet能够很容易获得大量的用户。这使得Java变得非常流行并被快速接受。JDK 1.0 还提供了用于网络编程的类，Java曾一度被宣传为一门网络编程语言。

注意

applet是基于互联网的小型程序，采用Java编写，可以从任意计算机下载。

1.5.2 JDK 1.1(1997年2月19日)

Java的下一个主要版本发布于1997年2月。正如你所看到的，这个主要版本仅在1.0版本发布一年之后便推出了。你可以想到Java在那么短的时间内获得了多么大的欢迎度。那么在这个发布版本中，Java又添加了哪些主要特性呢？JDK 1.1共包含504个类、23个包，这些数字表明了Java API中新增插件的数目。因此，虽然这次发布只是改动了次版本号，但事实上这是一次很大的改动。

这一次Java语言的主要改动是引入了新的事件处理模型。JDK 1.0使用Windows操作系统中用到的层次化事件模型。JDK 1.1及后续版本使用代理事件模型，与旧模型相比，新模型更加高效。这帮助改善了Java性能，同时也是Java最需要的。因为Java编译与解释特性的缘故，那个时期的Java在性能上表现不佳。使用代理事件模型需要用到新添加的匿名类与内部类，用以提升效率。

JDK 1.1在之后的两年中有许多次版本发布，最后发布的一个版本是1999年4月发布的JDK 1.1.8。每一次发布都往库中添加了新的类与接口，Java就这样一天天成长起来。接下来，我们会详细介绍这些次发布版本中的一些重要里程碑。

1. Java Bean

Java Bean API是在Java中进行组件开发的那段时期引入的。Java Bean是可重用的软件组件，采用Java编写。开发人员可以在生成器工具中可视化地对Java Bean操作。因此，你

可以通过使用基于Java Bean的API可视化地创建复杂GUI(Graphical User Interface，图形用户界面)。

2. 远程方法调用

JDK 1.1还推出了远程方法调用(Remote Method Invocation，RMI)，使用RMI可以让客户端调用运行在远程服务器上的应用程序。RMI使用名为Java远程方法协议(Java Remote Method Protocol，JRMP)的专有二进制协议。之后，一种名为互联网内部请求代理协议(Internet InterORB Protocol，IIOP)的新协议被业内广泛接受，并逐渐取代JRMP。IIOP被设计成公共对象请求代理体系结构(Common Object Request Broker Architecture，CORBA)的一部分。为了填补这两种协议间的空白，Java在IIOP之上推出了RMI。

3. JAR文件格式

Java applet可能由多个源文件构成。当这类applet代码编译时，会为每个公开类或接口生成一份单独的对象代码(.class)文件。当applet代码下载到客户机上时，所有的这些.class文件必须在applet开始执行前全部下载好。HTTP 1.0协议需要为每一个下载文件创建单独的套接字连接。为了解决这个问题，JDK 1.1引入了JAR的概念，即使用PKZIP算法将所有的.class文件打包成单独的文件，生成的JAR文件也仍然会被HTML代码中的APPLET标签引用。这样一来，客户端只需要套接字连接以下载单个JAR文件即可。这带来了更快的下载速度，并且缩短了客户端应用程序的启动时间。

4. 数字签名

JDK 1.1中的另一个主要改动是引入了数字签名。在JDK 1.0中，Java applet需要运行在具有若干安全限制且访问系统资源受限的沙箱中；相反，独立的Java应用程序运行在沙箱之外，具有访问所有系统资源的完全访问权限。借助数字签名，部署到远程服务器上的applet可以在签名验证后被信任。客户端会将独立应用程序拥有的权限授予此类签名验证后的applet。

5. AWT增强

AWT包含了用于构建GUI的接口与类，JDK 1.1为之提供了若干增强。最值得注意的改动是之前提到过的新事件模型。除此之外，AWT包还增加了一些其他功能。GUI现在支持使用剪贴板进行数据传输，这样就可以将原生应用程序中的文档剪切/复制/粘贴到Java应用程序中，反之亦然。可以在Java应用程序中设置和使用桌面颜色，还可以为菜单项定义快捷键。同样，你还可以创建出像Windows系统中一样的弹出菜单。

最后，AWT类添加了打印支持。打印功能在JDK 1.0中根本不存在。AWT还添加了ScrollPane容器，用于显示窗口中带有滚动条的大型文档。要注意的是，尽管JDK 1.0没有提供这类基本功能，但还是在非常短暂的一年时间内被开发人员广泛接受。Java必定有着美好的未来，如今这一点也得到了验证。

6. 其他改动

JDK 1.1中还有一些值得注意的改动，包括引入对象序列化类、内省与反射API，以及提

供定义内部类的功能。此外，JDK 1.1还在多个地方进行了性能改善。

1.5.3 JDK 1.2(1998年12月8日)：代号 Playground

Java的下一个里程碑发生在1998年12月。这个版本拥有1520个类、59个包。你可以清晰地看到Java在这3年中有着多么快速的增长。在这段时期里，Sun还推出了描述Java技术的新名词——Java SE(标准版)。JDK这一名字依然用于描述这门技术的实现。

1. 引入Swing

Swing是Java 2中引入的主要特性——提供了新的基于Java的GUI类。Swing的出现使得早期的AWT类多少有点被淘汰(除那些Swing扩展的类以外)。AWT组件被认为过于笨重，因为它们使用了许多操作系统调用。新的基于Swing的GUI类完全基于Java，且非常轻量级。这些基于Java的类提供了在浏览器中按需安装的好处，而不需要浏览器支持新版的JDK。这对于IE浏览器同样适用(读者可能会想到那段时期内微软与Sun Microsystems之间著名的法律诉讼)。Swing类允许“插拔式观感(Pluggable Look And Feel，PLAF)”，允许开发人员随意改变应用程序的观感。Swing支持Windows、Motif以及Java原生平台。

注意

虽然这个Java平台的官方名称是J2SE 1.2，但当时仍被称为Java 2。

2. 2D API

JDK提供了支持2D API的类，可以让用户借助一些预定义的类与接口轻松地创建二维图表。2D API是Java SE平台中所有绘图的基础，而不只是用于图表。

3. 拖曳功能

J2SE 1.2推出了拖曳功能，使得可以在Java程序中选取内容，然后将它们拖曳到原生应用程序中，反之亦然。因此，你可以将Word文档中的内容拖曳到Java应用程序中，反过来操作同样行得通。整个过程不需要借助剪贴板进行复制与粘贴。

4. 音频加强

JDK提供了用于播放MIDI文件以及.wav、.aiff与.au文件的类。此外，JDK还提供了更高的音质。

5. Java IDL

Java IDL是CORBA在Java平台上的ORB(Object Request Broker，对象请求代理)实现。这项功能将Java应用程序和现有的CORBA客户端与服务端进行了整合。Java既提供CORBA IDL(Interface Definition Language，接口定义语言)向Java接口的映射，也提供Java向CORBA IDL的逆向映射。这使得对现有的Java客户端与基于远程调用的Java应用程序的投资得到了保障。

6. 安全增强

Java安全是基于策略的。JDK 1.1推出了数字签名，使用数字签名可以帮助鉴别applet源；尽管如此，但并不能根据applet源分配不同的权限。换句话说，两个被信任的不同applet之间没有任何区别，或者可以说信任的applet与独立应用程序之间没有任何区别。它们都会获得相同的权限。J2SE 1.2通过创建安全策略解决了这个问题。用户可以基于applet源定义安全策略。在这种情况下，不同的applet将受限于由它们的用户定义的不同策略，从而获得不同的权限。这同样适用于独立的Java应用程序，这取决于用户定义的策略。策略文件基于文本且易于配置。Java还提供了策略工具用于创建和维护策略。策略为系统资源带来了更加细粒度的访问控制。

7. 其他增强

除了前面提到的主要改动之外，J2SE 1.2还对性能进行了改善。加载类拥有更好的内存压缩。内存分配与垃圾回收拥有更快的算法以进行快速分配与回收。在这个版本的Java中还引入了前面提到的JIT编译器，添加了ArrayList、BufferedImage以及像集合这样的API，另外还添加了DSA代码签名。

注意

在Java 2平台推出之际，Java冒险地进入了另一个叫做服务端Java的领域。为此，Java推出了一套名为J2EE(Java 2 Enterprise Edition)的独立Java类。J2EE为服务端组件开发提供了诸如Servlet的类，J2EE也经历了一些主要的修订版，目前包括用于创建诸如EJB(Enterprise Java Bean)、Java Server Page(JSP)、Java数据库连接(Java Database Connectivity, JDBC)、J2EE连接器等服务端组件的类。到目前为止，Java明显地分为两块领域：服务端Java(J2EE，现在称为Java EE)与标准版Java(J2SE)。由于本书只专注Java标准版，因此后续我们不会讨论J2EE。

1.5.4 JDK 1.3(2000年5月8日)：代号 Kestrel

这次发布的Java主版本是J2SE(Java 2标准版)，发布于2000年5月。代号为Kestrel，也叫做Java 2版本1.3。这个版本并没有在之前版本的基础上添加过多特性。包含的类数量从1520增至1840，包数量从59增至76。值得注意的改动包括捆绑热点JVM(HotSpot JVM，首次发布于1999年4月，起因是J2SE 1.2 JVM)、JavaSound、Java命名与目录接口(Java Naming and Directory Interface，JNDI)以及Java平台调试器架构(Java Platform Debugger Architecture，JPDA)。JNDI为企业中的多种命名与目录服务提供了统一的接口，包括轻量级目录访问协议(Lightweight Directory Access Protocol，LDAP)、域名系统(Domain Name System，DNS)、网络信息服务(Network Information Service，NIS)、公共对象请求代理体系结构(CORBA)以及文件系统。与所有的Java API一样，JNDI独立于底层平台。服务提供者接口(Service Provider Interface，SPI)允许目录服务实现连入框架，这可能会用到服务器、平面文件或数据库。

JDK 1.3对RMI API进行了若干增强——长度超过64KB的字符串现在可以被序列化，rmid需要一份安全策略文件才能进行命名。DataFlavor类的拖曳API新增了两个方法。另外，

Java 2D API也添加了新的功能，包括支持PNG格式。除此之外，Swing、AWT、安全方面以及对象序列化API都有了许多改动。该版本中增强了java.math包，并添加了一些新的类，包括Timer类、StrictMath类、打印类以及java.media.sound类。该API引入了热点以及位于IIOP之上的RMI(前面已经介绍过)，此外还添加了RSA代码签名。

Java的下一个次发布版本是J2SE 1.3.1，代号为Ladybird，发布于2001年5月17日。

1.5.5 JDK 1.4(2002年2月6日)：代号 Merlin

这是首个在JCP(Java Community Process，Java社区过程)下使用JSR(Java Specification Request，Java规范请求)59开发得来的版本，包含2991个类、135个包。其中的主要改动包括从模仿Perl得来的正则表达式、异常链接、集成的XML解析器、XSLT处理器(JAXP)以及Java Web Start。为了支持正则表达式，JDK 1.4中添加了叫做java.util.regex的新包，其中包含用于使用正则表达式指定模式进行字符序列匹配的类。你将在第8章学习关于异常链接的知识。JAXP(Java API for XML Processing，用于处理XML的Java API)通过一系列标准的Java平台API为处理XML文档提供了基本支持。Java Web Start软件为基于Java技术的应用程序提供了一套灵活、鲁棒的部署解决方案。

注意

JCP是为Java技术开发标准技术规范的一种机制。JSR是对Java平台上提议或最终规范的实际描述。

除此之外，JDK 1.4还对AWT包进行了改动以提升鲁棒性、行为以及GUI程序的性能。同样，Swing也增加了许多新特性，包括新的spinner组件以及支持拖曳的格式化文本字段组件。JDK 1.4还将Popup与PopupFactory类开放给程序员，以允许他们创建自己的弹出菜单。JDBC为Java程序语言提供了统一的数据访问，并且在这个版本中被加强为JDBC 3.0 API。新特性包括可以在事务中设置存储点，在事务提交后保持结果集打开，重用prepared statement，获取自动生成的键，一次打开多个结果集。此外还包含了两个新的JDBC数据类型：BOOLEAN与DATALINK。DATALINK数据类型使得管理数据源之外的数据成为可能。新的I/O API提供了新特性并改善了性能。除了这些改动之外，JDK 1.4还为RMI、Math、集合框架、可访问性(Accessibility)以及Java原生接口(Java Native Interface，JNI)添加了一些功能。

之后发布的次版本有代号为Hopper的J2SE 1.4.1(发布于2002年9月16日)以及代号为Mantis的J2SE 1.4.2(发布于2003年6月26日)。

1.5.6 JDK 5.0(2004年9月30日)：代号 Tiger

代号为Tiger的这个版本在JSR 176下开发而成，增加了许多重要的语言特性，包括for-each循环、泛型、自动装箱以及可变参数(varargs)。由于本书专注于讲解J2SE，因此我们会介绍这个版本中的大部分新增特性。该版本共有3562个类、166个包。

泛型可以让类型或方法在提供编译器类型安全的同时，操作各种不同类型的对象。集合框架中的所有类都增加了泛型版本。你将在第12章学习泛型。增强的for循环在遍历集合和数组时消除了枯燥乏味的工作以及迭代器和索引变量可能造成的潜在错误。自动装箱/拆箱特性消除了基本类型与封装类型间手工转换的繁重工作，第11章会讲解该特性。可变参数可以

让你在程序调用时指定数量变化的参数，这个特性将在第9章讨论。除了这些以外，这个版本还引入了类型安全的枚举，我们将在第11章进行介绍。新添加的静态导入功能可以避免限定静态成员使用类名，从而克服了“常量接口反模式”带来的缺点。此外，国际化API也得到了增强。国际化是在无须进行工程性改动的情况下让应用程序适应多种/多地区语言的过程。有的时候，术语“国际化”会简写为“i18n”，这是因为首字母(i)与末字母(n)之间有18个字母。除此之外，以下API得到了增强：JavaSound与Java 2D技术、Image I/O、AWT以及Swing。java.lang与java.util获得了若干增强，包括加入新的Formatter与Scanner类，在本书中你会看到多个程序示例使用了这些类。并发工具与集合框架也得到了许多增强。我们将在第19章中介绍并发工具中所有的主要增强功能，在第16章介绍集合框架的主要增强功能。在硬件方面，JDK 5.0为SUSE Linux与Windows 2003上的服务器VM添加了对AMD皓龙处理器的支持。

1.5.7 JDK SE 6(2006年12月11日)：代号 Mustang

这个版本促进了JVM中脚本语言的使用(JavaScript使用Mozilla的Rhino引擎)，并提供了对Visual Basic语言的支持。在这个版本中，Sun使用Java SE替换了名称“J2SE”，并且丢掉了版本号后面的“.0”。这个版本对集合框架进行了许多改动并添加了许多功能，其中包括添加新的接口Deque、BlockingDeque、NavigableSet、NavigableMap以及ConcurrentNavigableMap。此外还添加了一些具体的实用类，并翻新了已有的类以实现新的接口。我们将在第16章讨论集合框架。java.lang.instrument包也获得了若干增强。Instrumentation API提供了可以让Java程序语言代理将程序instrument到JVM上运行的服务。在这本书中，我们不会讨论instrumentation。java.io包引入了名为Console的新类，我们将在第9章对其进行介绍。Console类包含了用于访问基于字符的控制台设备的方法；该类的readPassword方法通过禁用控制台上的字符回显功能来允许输入敏感数据，比如密码。File类现在拥有可以获取磁盘使用信息以及设置和查询文件权限的方法，我们将在第9章讨论该类。JAR与ZIP API均得到了增强。此外，Java Web Start与Java网络加载协议(Java Network Launching Protocol，JNLP)也获得了增强。JNLP为部署基于Java 2技术的应用程序到客户端桌面提供了浏览器无关的体系结构。

其他的主要改动包括支持插入式注解(JSR 269)、大量的GUI改善，包括增强原生UI以支持Windows Vista观感，以及增强JPDA与JVM工具接口以更好地进行监测与故障排除。

1.5.8 JDK SE 7(2011年7月7日)：代号 Dolphin

Java SE 7是Java SE平台的主要发布版本，在上一版本发布很久后才到来。Java SE 7为Java语言引入了许多增强功能：整数类型现在可以使用二进制数系统表示，数值字面量可以包含下划线字符以获得更好的可读性，可以在switch语句中使用字符串，在泛型实例的创建语法中引入了钻石运算符，添加了新的带资源的try语句，多种异常类型现在可以包含在单个catch块中。为了改善可变参数方法中传递非具体化参数(nonreifiable formal parameter)时的编译器警告与错误，这个版本添加了一个新的编译器选项以及两个注解。这里的许多特性都会在本书的相关章节中进行介绍。

Java SE 7引入了NIO.2 API，可以为管理文件系统对象而开发自定义的文件系统供应

程序。API中的新增功能为文件I/O与文件系统访问提供了广泛支持，我们将在第9章进行介绍。JDBC 4.1 API可以让你使用带资源的try语句来自动关闭类型为Connection、Resultset和Statement的资源。RowSet也添加了一些功能，可以创建JDBC驱动支持的所有行集类型。Java SE 7添加了对Solaris中流控制传输协议(Stream Control Transmission Protocol，SCTP)以及套接字连线协议(Socket Direct Protocol，SDP)的支持。SDP是为了在InfiniBand fabric结构上支持流连接的连线协议。在本书写作之际，SDP已经出现在Solaris与Linux平台上。新版本让你能够将富Internet应用(Rich Internet Application，RIA)开发和部署成applet或Java Web Start应用程序。由于这是一块完全不同的应用程序领域，因此需要单独一本书进行描述，本书将不再讨论这些API。

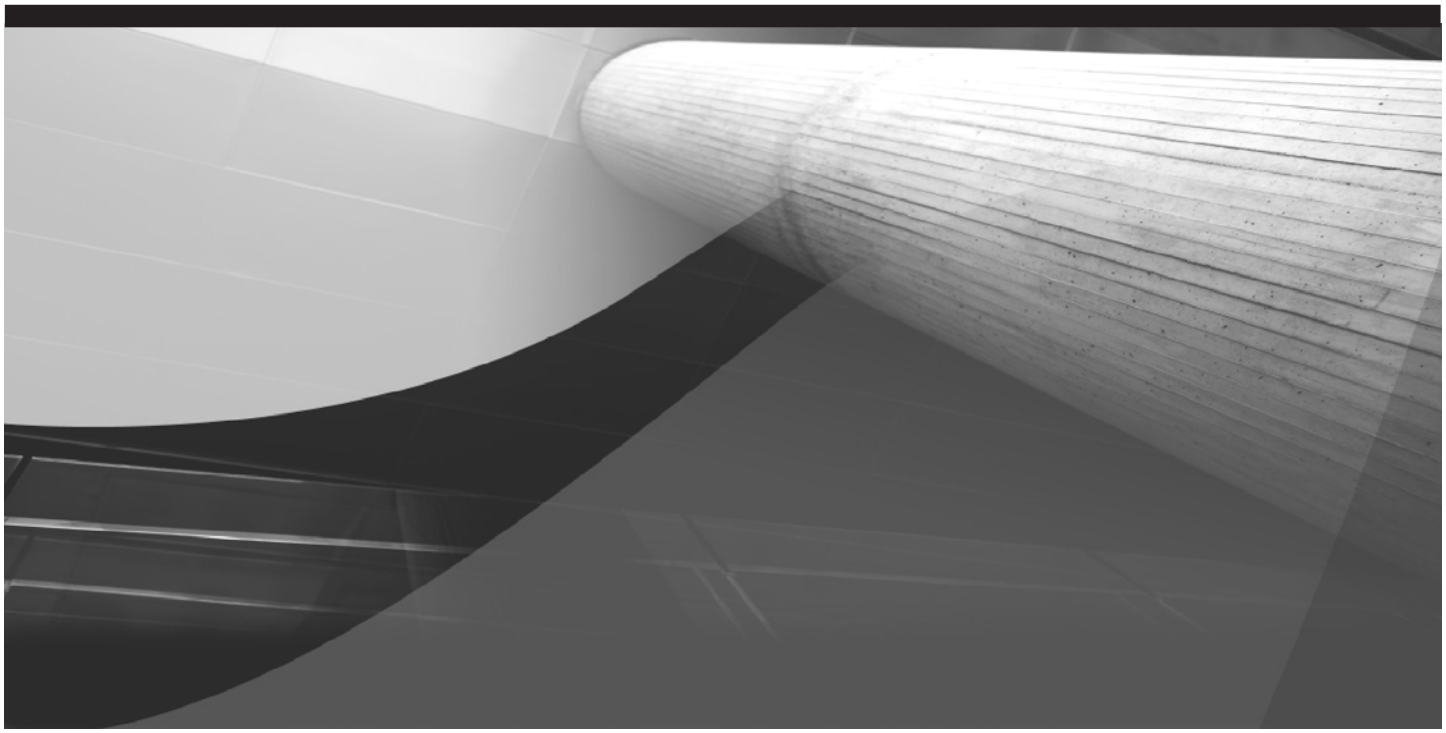
Java SE平台支持JVM上的动态类型化程序语言实现；为此，JVM新增了一套叫做invokedynamic的指令。尽管会在第21章详细讨论内省和反射API，但是我们不会介绍这套指令的使用，因为与本书的上下文没什么关联。并发API新增了轻量级的分支/合并(Fork/Join)框架，这将在第19章进行全面介绍。在客户端，Java SE 7为Swing添加了下一代的跨平台外观，名为Nimbus外观。这种外观在本书的多个应用程序中将会用到。书中的好几个例子都使用了这个框架。除了这些以外，还有一些与本书上下文无关的增强。例如，XML栈被升级以支持最新版本的XML处理、绑定与Web服务API。MBean API增添了更多的管理功能。在安全性和加密API中，新增了标准的椭圆曲线加密(Elliptic Curve Cryptographic，ECC)算法的可移植实现。国际化API被增强以支持Unicode 6.0版本。Locale类得到升级，区域设置处理得到更新，从而将格式化区域设置从用户界面语言区域设置中分离。我们鼓励读者访问openjdk站点(http://openjdk.java.net/projects/jdk7/features/)，查看Java SE 7平台新增的其他改动。

1.6 小结

Java自从1996年公布于众以来，就一直风靡于世界范围内的开发人员中——大部分缘于Java的跨平台特性。如今，Java已经演变成可供创建、部署及运行各种不同类型应用程序的成熟平台。为了寻求可移植性，Java使用了虚拟机(VM)的概念。Java源文件会编译成由伪CPU指令集构成的字节码。伪CPU会通过运行进程在内存中进行仿真，编译后的字节码会运行在这个仿真器中。这种做法使得Java可执行文件可以运行在任何具有JVM(Java虚拟机)的平台之上。Java值得注意的主要特性有：体积小、简单易学、面向对象、兼具编译与解释特性、平台无关、鲁棒且安全、支持多线程以及动态特性。对于每一种特性，我们在本章均进行了深入介绍。

Java在过去17里历经了若干版本。这些修订版本由Java工具包的版本号标记。JDK 1.0发布于1996年，接下去的主要发布版本——JDK 1.1仅在1年后就发布了。在之后的若干年里，Java演变成若干不同的平台，包括Java SE(Java平台标准版)、Java EE(Java平台企业版)、Java ME(Java平台微型版)以及Java Card技术。在本章，我们介绍了所有这些主要发布版本的重要特性以及相比上一版本增添的特性。

在简短地介绍完激动人心的Java平台后，让我们开始学习吧。



第2章

数　组

第1章介绍了Java技术的概况以及Java在各个不同阶段的发展历程。本章将介绍如何进行编程。大部分编程书籍都使用Hello World示例开头，本书也不例外。不过由于本书没有太多篇幅用于讲述Java基本语法，因此我和McGraw-Hill出版社商议，决定将相关资料放在网上。资料中的语法参考部分共分3章，分别介绍概念、运算符和控制流语句。由于这3章内容涵盖了Java 7最新增添的部分，因此建议读者阅读一下。另外，如果你熟悉其他编程语言或是了解一些Java知识，可以考虑快速阅读并只关注Java 7中的改动。

由于已将3章关于基本语法的内容放在了网上，因此现在我们将学习一个重要的概念——数组。数组用于帮助创建和访问具有相同数据类型的若干元素。在本章，你将学习如

何在程序中创建和使用数组、如何创建一维和多维数组。

具体学习内容如下：

- 声明数组
- 访问/修改数组元素
- 在运行时初始化数组
- 使用数组字面量初始化数组
- 多维数组
- 非矩形数组
- 计算数组大小
- 复制数组
- 理解数组的类表示形式

2.1 数组

当想要操作相同数据类型的变量集合，或是想将这些变量放在一处时，可以创建数组。数组本质上是相同数据类型的元素的集合。其中，元素是数组的组成部分。数组中的每个元素可以使用唯一的索引值进行访问，这些索引也叫做下标。例如，整型数组中包含的每个元素都是int类型，浮点型数组中包含的每个元素都是float类型。由于数组元素使用单一变量名与下标进行访问，因此不需要在程序代码中为存储和访问相同数据类型的多个变量创建多个唯一的变量名。图2-1展示了典型的整型数组。

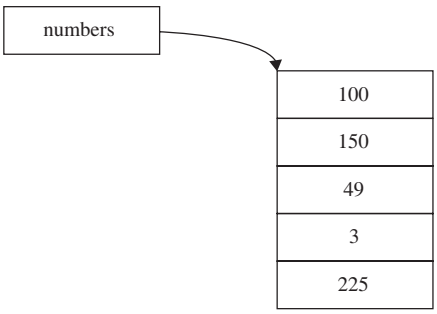


图2-1 包含5个整型变量的数组

图2-1中的数组存储了5个整型变量，其中的每个变量都将使用单一名称numbers以及数组唯一索引进行访问。另外，每个变量拥有的数值与其他元素拥有的数值间相互独立，且每个元素都可以被单独访问和修改。在接下来的内容中，你将学习如何定义数组以及访问数组元素。

2.1.1 声明数组

声明数组的一般语法如下：

```
type arrayName[ ];
```

此外，Java还提供了另一种声明数组的语法，如下所示：

```
type[ ] arrayName;
```

注意

虽然Java支持两种类型的声明，但一般不使用第一种形式。在声明中，方括号表示声明的变量为数组类型，而type表示数组将要包含的元素类型，因此在方括号后跟上指定类型更有意义一些。所以，后者是声明的首选形式，而本书采用的便是第二种形式。

语法中的方括号也称为索引运算符，因为它们指定了数组中元素的索引。type指明数组将要存储的元素类型，arrayName指明程序代码中数组元素寻址的名称。与其他许多语言类似，声明中不允许指定数组大小。

声明整型数组的代码如下：

```
int[] numbers;
```

上述声明中的数组名称为numbers。因此，数组中的每个元素都可以使用该名称连同对应的索引值进行访问。

声明浮点型数组的代码如下：

```
float[] floatNumbers;
```

数组名称为floatNumbers，数组中的每个元素都保存一个浮点数。

注意

这些声明简单地创建了数组类型的变量。它们并没有创建实际的数组，并且没有分配内存空间用于存储数组元素。

2.1.2 创建数组

既然已经了解了如何声明数组变量，下一个任务将是为数组元素分配空间。为了分配空间，你会用到new关键字。例如，可以使用如下代码创建大小为10个元素的数组：

```
int[] numbers;
numbers = new int[10];
```

第一条语句声明了一个数组类型的变量，叫做numbers，数组元素的类型为int。第二条语句为保存10个整数分配了连续内存，并将首个元素的内存地址赋给变量numbers。数组初始化器会为numbers数组的所有元素提供初始化值。也可以在代码中的一些地方为这些变量赋予不同的值，我们将在后面进行解释。

可以使用如下代码创建包含20个浮点数的数组：

```
float[] floatNumbers;
floatNumbers = new float[20];
```

第一条语句声明了一个叫做floatNumbers的数组类型变量。其中，数组的每个元素均为float类型。第二条语句分配了保存20个float类型元素的内存空间并将首个元素的内存地址赋值给数组变量。

2.1.3 访问和修改数组元素

在了解完如何创建数组之后，我们的下一个任务是访问数组元素。你可以使用索引来访问数组元素。数组的每个元素都拥有唯一的索引值。数组元素可使用数组名并跟上方括号内的索引值进行访问。访问数组元素的一般形式如下：

```
arrayName[indexValue];
```

例如，考虑如下整型数组的声明：

```
int[] numbers = new int[5];
```

请注意声明和分配是如何在相同的程序语句中完成的。

在该声明中，numbers数组共有5个元素。数组索引以0开始，因此数组的首个元素可以使用如下语法访问：

```
numbers[0]
```

如图2-2所示。

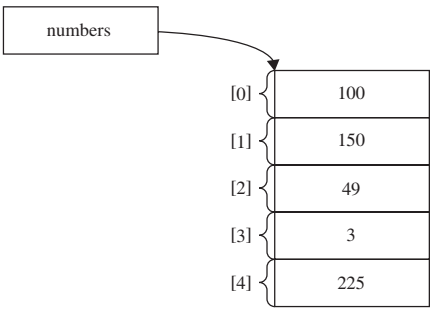


图2-2 使用下标访问数组元素

后续元素将使用语法numbers[1]、numbers[2]等访问。最后那个元素使用语法numbers[4]访问。请注意使用索引值5是非法的，因为numbers[5]会试图访问第6个元素，而这超出了给定数组的边界。

提示

试图访问数组边界之外的元素会引发ArrayIndexOutOfBounds异常。关于该异常，我们将在第8章进行解释。

提示

不管在什么时候遇到新的类，比如ArrayIndexOutOfBounds，我们都强烈建议你打开javadocs(http://docs.oracle.com/javase/7/docs/api/)以了解更多关于新类的知识。

数组元素总是存储在一块连续的内存中。图2-3显示了使用如下语句创建的包含5个元素的整型数组的内存分配情况：

```
int[] numbers;
numbers = new int[5];
```

由于int占用4个字节的内存，因此你会注意到图2-3中数组的每个元素占据了4个字节。因此，总的内存占用空间是20字节。通常，在访问或修改数组元素时，你不用担心关于内存分配的问题。

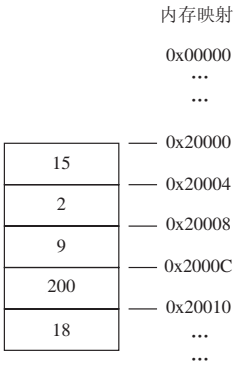


图2-3 显示整型数组的内存映射

可以使用如下语法修改数组元素的值：

```
arrayName[ index_value] = data;
```

例如，在前面声明的numbers数组中，可以使用如下语句将处于索引3位置的元素的值设置为100：

```
numbers[3] = 100;
```

请注意，索引为3的元素是指数组中的第4个元素。

2.2 初始化数组

此时此刻，你已经学习了如何声明数组、为数组元素分配空间以及访问数组元素。现在，你将学习如何在使用数组元素前初始化整个数组到某个目标状态。

警告

虽然Java为数组元素提供了默认初始化。但是当数组元素为对象时，默认初始化结果会导致对象引用设置为null。如果元素没有初始化为合适的对象引用，就会导致运行时异常。

初始化数组元素有两种方法。数组元素既可以在运行时通过赋值完成，如前面所示；也可以通过数组字面量进行初始化。

2.2.1 在运行时初始化

与前面讨论的一样，数组元素可以在运行时使用索引值进行初始化。考虑如下整型数组的声明：

```
int[] numbers = new int[10];
```

通过使用如下代码段，可以将每一个元素初始化为目标值：

```
numbers[0] = 10;
numbers[1] = 5;
numbers[2] = 145;
...
numbers[9] = 24;
```

如果决定将所有元素初始化为同一个值，可以使用Syntax Reference 3中讨论的一种循环结构来完成。例如，下面的for循环将初始化先前数组的所有元素为0：

```
for (int i = 0; i < 10; i++) {
    numbers[i] = 0;
}
```

2.2.2 使用数组字面量初始化

数组字面量提供了一种更短、可读性更好的方法来初始化数组。例如，考虑如下程序语句：

```
int[] numbers = {15, 2, 9, 200, 18};
```

该语句声明了包含5个元素的整型数组。编译器会根据花括号内指定的初始化器的数量决定数组大小。当JVM在内存中加载代码时，会使用花括号内指定的数值对数组分配的内存位置进行初始化。

这类初始化有时也被称为集初始化(aggregate initialization)，并且是一种非常安全的用于初始化数组的方法。如果使用前面提到的运行时初始化，那么代码会比较容易出错，因为可能会在修改元素时无意中为索引指定错误的值。使用数组字面量可以对初始化代码进行集中，并且添加和修改元素不会出错。例如，只要在初始化器列表中编写新的初始化器就可以添加新的元素；而只要从列表中删除初始化器就可以删除相应的元素。使用该方法不会造成数组越界错误，因而使用更为安全。

还有很重要的一点需要知道，Java虚拟机架构不支持任何形式的高效率的数组初始化。实际上，数组字面量是在程序运行时而非程序编译时被创建和初始化。例如，考虑先前数组字面量的声明：

```
int[] numbers = {15, 2, 9, 200, 18};
```

上述代码会被编译成如下等价代码：

```
int[] numbers = new int[5];
numbers[0] = 15;
numbers[1] = 2;
numbers[2] = 9;
numbers[3] = 200;
numbers[4] = 18;
```

因此，数组字面量只是前面讨论的运行时初始化代码的一种快速写法。

警告

如果想要在Java程序中包含大量数据，请不要使用数组字面量，因为Java编译器会创建大量的Java字节码用于初始化数组。更好的方法是在外部文件中存储数据，然后在运行时从中读取内容，并使用这些值初始化数组元素。

注意

考虑如下声明：

```
long[] times = {System.nanoTime(), 0};
```

times数组中下标为0的元素将在程序运行时，而非代码编译时保存当前时间的长整数值。

警告

同C++一样，Java不允许在方括号内指定数组大小，而是根据表达式右侧(Right-Hand Side，RHS)值的数量决定数组大小。指定数组大小是非法的。因此，即使指定与右侧值数量相同的大小，程序语句也仍然无法编译。

上述声明的内存表示见图2-4。

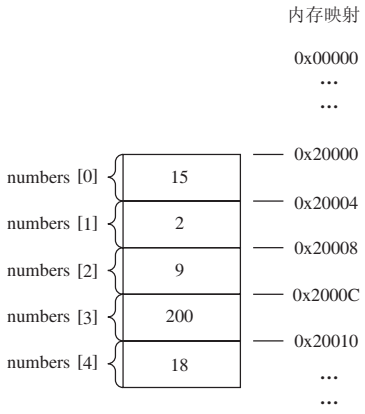


图2-4 显示整型数组的内存映射

现在让我们用一个简单的程序看看如何使用数组。程序清单2-1声明和使用了一个存储学生数学考试分数的整型数组。程序提示用户输入每一个学生的分数。输入数据存储在一个整型数组中。在接收完所有学生的成绩后，程序会通过访问存储在数组中的分数计算班级平均分。

程序清单2-1 显示数组用法的程序

```
import java.io.*;

public class TestScoreAverage {

    public static void main(String[] args) {
        final int NUMBER_OF_STUDENTS = 5;
        int[] marks = new int[NUMBER_OF_STUDENTS];
        try {
            BufferedReader reader =
                    new BufferedReader(new InputStreamReader(System.in));
            for (int i = 0; i < NUMBER_OF_STUDENTS; i++) {
                System.out.print("Enter marks for student #" + (i + 1) + ": ");
                String str = reader.readLine();
                marks[i] = Integer.parseInt(str);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        int total = 0;
        for (int i = 0; i < NUMBER_OF_STUDENTS; i++) {
            total += marks[i];
```

```
            }
            System.out.println("Average Marks " + (float) total / NUMBER_OF_STUDENTS);
        }
    }
```

在main方法中，我们首先创建一个常数用于定义班级学生的数目：

```
final int NUMBER_OF_STUDENTS = 5;
```

提示

使用final关键字在程序中创建常量。使用final声明的变量只能被初始化一次。变量在初始化后，它们的值在程序生命周期中无法被修改。

对于以下情况，创建常量总是好的做法：如果班级的学生人数发生变化，只需要修改一条语句即可，而程序中的剩余部分将不会受到影响(或至少需要做的改动最少)。

接下来声明并创建了名为marks的整型数组，这个数组的大小等于NUMBER_OF_STUDENTS，如下所示：

```
int[] marks = new int[NUMBER_OF_STUDENTS];
```

现在，我们通过提示用户在终端输入每个学生的分数来进行读取。为了从终端读取输入，在System.in对象上创建BufferedReader对象，如下所示：

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

关于使用BufferedReader和InputStreamReader类接收来自用户输入的部分，Syntax Reference 3中已经简要地介绍过。

程序现在通过循环读取分数：

```
for (int i = 0; i < NUMBER_OF_STUDENTS; i++) {
```

对于每一个学生，我们使用System.out.print语句提示用户：

```
System.out.print("Enter marks for student #" + (i + 1) + ": ");
```

用户输入通过调用reader对象的readLine方法来读取：

```
String str = reader.readLine();
```

输入分数为String格式。我们需要在将分数赋值给数组元素前将字符串转换为整数。为了将字符串转换为整数，我们使用Integer类的parseInt方法：

```
marks[i] = Integer.parseInt(str);
```

由于readLine与parseInt方法都会生成运行时错误，因此我们将它们放置在了try-catch块中。异常处理器通过调用Exception对象的printStackTrace方法来打印堆栈踪迹：

```
e.printStackTrace();
```

在读取完用户输入的分数并将它们复制到数组元素之后，使用for循环计算分数总和：

```
int total = 0;
for (int i = 0; i < NUMBER_OF_STUDENTS; i++) {
    total += marks[i];
}
```

最后，我们使用如下语句将班级平均分打印到用户控制台：

```
System.out.println("Average Marks " + (float) total / NUMBER_OF_STUDENTS);
```

请注意，在计算平均分时，我们将total变量转换成了float类型。如果我们不这么做，编译器会对两个操作数执行整数除法，从而导致结果不精确。

这个简单程序展示了一维数组的声明、数组元素的初始化以及如何在程序代码中访问数组元素。

现在，我们已经了解了一些关于数组的知识。下面让我们稍作休息，暂时讨论一会儿控制流。和你在Syntax Reference 3中学习到的一样，Java还提供了另一种概念用于遍历数组元素，即for-each循环。这个概念在J2SE 5.0中引入，稍后我们将对其进行讨论。

2.3 for-each循环

for-each循环可以让你在不使用元素索引值的情况下遍历整个数组。for-each循环的一般形式如下：

```
for (type variableName : collection) {
    loopBody
}
```

注意

for-each循环在J2SE 5.0中引入。对于这种循环结构，官方还使用了其他一些名称，如“加强型for”循环、“For-Each”循环和“foreach”语句。

variableName指定变量类型及名称，collection代表数组名称。对于for-each循环的每一次遍历，loopBody执行一次。遍历过程将持续，直到数组的最后一个元素被处理完为止。借助这种循环结构，你可以使用如下代码遍历前面声明的marks数组：

```
for (int m : marks) {
    System.out.println (m);
}
```

for-each循环的每一次遍历会打印marks数组的一个元素值。在每一次遍历后，数组索引会自动增长以获取下一个元素。for-each循环会持续下去，直到获取最后一个元素为止。for-each循环对于遍历数组的所有元素非常有用。特别是可以让你在不使用迭代器或索引变量的情况下遍历整个集合和数组(第16章将会谈到集合)。虽然for-each语句非常强大，但是并没有对集合遍历进行优化。

提示

使用for-each可以轻松地将数组的所有元素输出到用户控制台。不过，使用Arrays类的toString方法也有异曲同工之妙。我们稍后将在本章对其进行介绍。

for-each虽然很强大，但是也有一些固有的限制。可以用于访问数组元素，但却不能对数组元素进行修改。不适用于并行遍历多个集合的循环中，例如比较两个数组中的元素。只能用于单个元素访问，而不能用于比较数组中的后续元素。for-each结构是顺向的迭代器。如果只想访问数组中的一些元素，可能需要使用传统的for循环。

在结束对for-each循环的讨论之前，你可能想知道为什么Java设计人员没有像C#(.NET语言)那样选择引入新的foreach关键字。之所以不引入foreach关键字，是因为要确保向后兼容J2SE 5.0之前的代码，而那些代码可能会将该关键字作为标识符使用。其他语言中for-each循环的一般语法如下：

```
foreach (element in collection)
```

使用上述语法会导致代码与前面版本不兼容，因为这里的in是关键字(例如Java中有System.in)。

回到我们关于数组的讨论中，目前为止我们已经讨论了一维数组。现在我们将讨论多维数组。

2.4 多维数组

多维数组，顾名思义，是指包含的数组维数多于一维。在Java中，可以创建二维、三维和n维数组(这里n为任意自然数)。维数可能为任意大的数值，这取决于编译器施加的限制。JDK编译器将上限设置为255，由于该数值很大，因此开发人员不需要为之担心。

让我们首先探讨一下二维数组。

2.4.1 二维数组

二维数组可以被看做一张包含行与列的表格，其中表格的每一个单元格代表一个数组元素。

声明二维数组的一般形式如下：

```
type arrayName[ ] [ ] ;
```

或

```
type[ ] [ ] arrayName;
```

提示

虽然这两种语法均有效，但是第二种更受青睐，其中方括号紧接着置于数据类型之后。

type指定每一个数组元素保存的数据类型。arrayName为数组名称。两对方括号表明当前数组变量被声明为二维数组。

假如你要编写程序，将获得的班级中每个学生的分数存储到不同的对象。我们假定每个学生选修5门科目，班级中共有50名学生，你可能需要创建二维数组来存储数据。可以使用下列代码创建下面这样的数组：

```
final int NUMBER_OF_SUBJECTS = 5;
final int NUMBER_OF_STUDENTS = 50;
```

```
int[][] marks;
marks = new int[NUMBER_OF_SUBJECTS][NUMBER_OF_STUDENTS];
```

数组的第一维(即行)指定科目ID，第二维(即列)指定学生ID。你可以通过改变先前声明中行与列的声明顺序，轻松地交换行与列，如下所示：

```
marks = new int[NUMBER_OF_STUDENTS][NUMBER_OF_SUBJECTS];
```

在这种情况下，第一维指定学生ID，第二维指定科目ID。图2-5使用表格形式展示了第二种声明的内存布局。

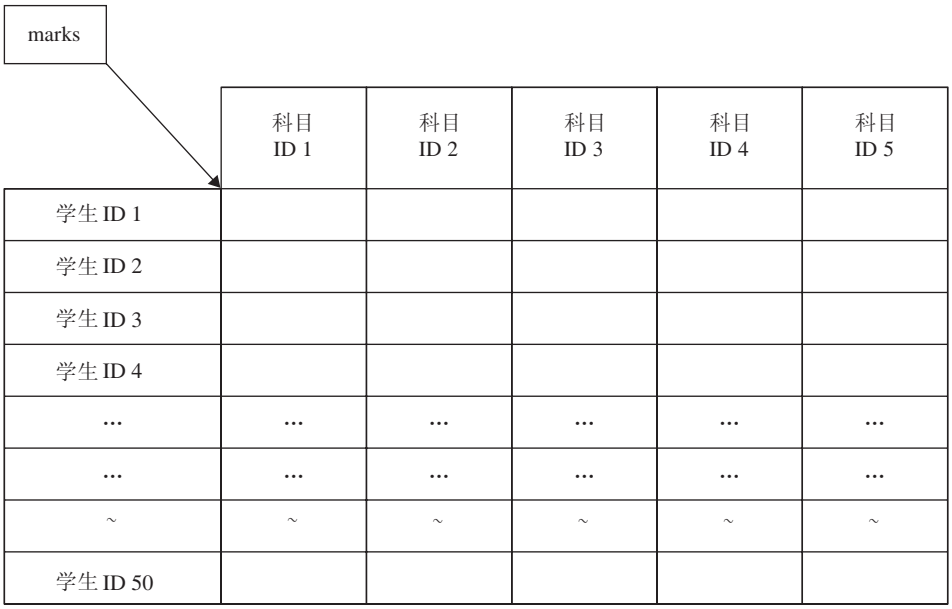


图2-5 显示二维数组的内存映射

图2-5展示的内存映射显示了学生ID和科目ID，它们均从1开始(一般来说，一名学生和一门科目的ID不会被分配为0)。当存储学生分数到该数组中时，需要将两个下标都调整1以遵照Java对数组索引的要求。例如，可以使用如下语句对ID为6的学生在ID为4的科目上取得的分数进行赋值：

```
marks[5][3] = 78;
```

请注意，行索引为5表示表格中的第6行，因为索引总是从0开始。类似的，列索引为3代表表格中的第4列。

再如语法marks[0][0]，代表ID为1的学生在ID为1的指定科目上取得的分数(同样，请记住在我们的标记中，学生ID与科目ID以1开始)。类似的，marks[49][4]将表示班级中最后一名学生(学生ID为50)在最后一门科目(科目ID为5)上取得的分数。

2.4.2 初始化二维数组

就像通过运行时初始化或数组字面量对一维数组进行初始化一样，也可以使用这两种方法对二维数组进行初始化。下面讨论一下这两种方法。

1. 在运行时初始化

为了初始化二维数组中的元素，可以使用前面讨论的语法来访问数组元素，然后使用赋值语句进行初始化：

```
arrayName[row] [col] = data;
```

例如，考虑如下整型二维数组的声明：

```
int[][] marks;
marks = new int[5][50];
```

数组的单个元素可以使用如下程序语句初始化：

```
marks[0][5] = 78;
marks[2][10] = 56;
```

第一条语句将地址为第1行、第6列的数组元素的值初始化为78，第二条语句将地址为第2行、第11列的数组元素的值初始化为56。

可以使用嵌套for循环将二维数组的每个元素初始化为某个特定值。可以使用下面的嵌套for循环，将数组中的每个元素值初始化为0：

```
final int MAX_ROWS = 5, MAX_COLS = 50;
for (int row = 0; row < MAX_ROWS; row++) {
    for (int col = 0; col < MAX_COLS; col++) {
        marks[row][col] = 0;
    }
}
```

2. 使用数组字面量初始化

可以在数组声明过程中，在赋值运算符右侧的花括号里定义单个数组元素的值。初始化二维数组元素的一般形式如下：

```
int[][] subjectMarks = {
    {1, 98},
    {2, 58},
    {3, 78},
    {4, 89}
};
```

请注意上面使用嵌套花括号分离不同数据声明行的写法。Java编译器会为前面的声明生成等价于如下代码的字节码：

```
subjectMarks = new int[][];
subjectMarks[0][0] = 1;
subjectMarks[0][1] = 98;
subjectMarks[1][0] = 2;
...
```

就像前面说的，数组字面量只不过是逐个初始化数组元素的一种速写方式。

警告

与其他语言(比如C++)不同的是，Java并不允许忽略用于标记二维数组中每一行的内部花括号。因此，如下声明在Java中是无效的：

```
int[][] subMarks = {1, 98, 2, 58, 3, 78, 4, 89};
```

这是因为Java不允许在声明语句中指定数组维数。

接下来，我们将借助程序示例来讨论二维数组的用法。我们会开发程序以存储和显示班级中所有学生在所有科目上取得的分数。根据前面的解释，我们需要创建二维数组才能达到上述目的。

程序清单2-2展示了如何声明、初始化和访问二维数组。

程序清单2-2 展示使用二维数组的程序

```
public class MultiDimArrayApp {

    public static void main(String[] args) {
        final int MAX_STUDENTS = 50, MAX_SUBJECTS = 3;
        int[][] marks = new int[MAX_STUDENTS][MAX_SUBJECTS];
        // Adding data to the array
        for (int id = 0; id < MAX_STUDENTS; id++) {
            for (int subject = 0; subject < MAX_SUBJECTS; subject++) {
                marks[id][subject] = (int) (Math.random() * 100);
            }
        }
        // Printing Array
        System.out.print("Student\t");
        for (int subject = 0; subject < MAX_SUBJECTS; subject++) {
            System.out.print("\t" + "Subject " + subject + "\t");
        }
        System.out.println();
        for (int id = 0; id < MAX_STUDENTS; id++) {
            System.out.print("Student " + (id + 1) + '\t');
            for (int subject = 0; subject < MAX_SUBJECTS; subject++) {
                System.out.print("\t" + marks[id][subject] + "\t");
            }
            System.out.println();
        }
    }
}
```

程序首先创建了两个常数用于定义数组大小：

```
final int MAX_STUDENTS = 50, MAX_SUBJECTS = 3;
```

真实情况下，两个常数的数值都会比上面的数值大一些，这里主要是为了简化问题。其次，我们使用这些常数声明了一个整型二维数组：

```
int[][] marks = new int[MAX_STUDENTS][MAX_SUBJECTS];
```

请注意第一维用于跟踪学生，其中每一行将对应唯一的学生ID；而第二维用于跟踪科目，其中每一列将对应唯一的科目ID。

现在，我们将向数组中添加一些数据。与其在程序中让用户输入每一名学生在每一门科目上的分数，不如在程序中采用编程的方式在数组中输入值。我们首先使用循环遍历所有学生的ID，如下所示：

```
for (int id = 0; id < MAX_STUDENTS; id++) {
```

对于每一个学生ID，我们将通过建立内部for循环遍历所有的科目ID：

```
for (int subject = 0; subject < MAX_SUBJECTS; subject++) {
    marks[id][subject] = (int) (Math.random() * 100);
}
```

通过为数组元素赋值范围在0到99之间的随机数来对数组进行初始化。Math类的random方法会生成范围在0到1.0(不包括1.0)的double值。我们将该值乘上100，接着将结果类型转换为int，最后将之赋值给marks数组元素。

请注意，我们使用语法marks[i][j]访问数组的第[i, j]个元素。嵌套for循环保证我们访问到数组的每一行和每一列，并对数组中的每一个元素进行初始化。一旦整个数组都被填充值，就将数组元素的值打印到用户控制台。

之后，我们再一次使用嵌套for循环遍历所有的行与列。下列语句用于访问单个数组元素并将它们输出到控制台：

```
System.out.print("\t" + marks[id][subject] + "\t");
```

我们在每个单元值之后输出制表符，用以对列进行分隔。程序会在适当的位置打印制表符以格式化输出。以下是部分输出结果：

```
Student         Subject 0     Subject 1    Subject 2
Student 1           23            41           17
Student 2           72            44           46
Student 3           65            65           56
Student 4           12            11           76
Student 5           49            53           36
```

2.4.3 使用for-each结构进行循环

可以简单地使用前面提到的for-each结构循环二维数组中的所有元素。例如，可以使用下面的代码段遍历前面示例中声明的marks数组中的元素：

```
int i = 0;
for (int[] student : marks) {
    System.out.print("Student " + i++ + '\t');
    for (int value : student) {
        System.out.print("\t" + value + "\t");
    }
    System.out.println();
}
```

外层for循环遍历数组中的每一行，内层for循环访问数组中的每一列，value变量保存数组元

素的值。因此，所有数组元素都以表格样式输出到控制台。请注意为了输出学生ID，我们使用了另一个变量i。这是因为外层for循环现在指代了一个整型数组，而不是前面例子中的整型ID。

2.5 n维数组

到目前为止，我们已经讨论了一维数组和二维数组。相同的概念也可以扩展用于表示和访问n维数组。例如，可以使用下列语法声明三维数组：

```
type[] [] [] arrayName= new type[size] [size] [size] ;
```

例如，下面是三维数组的声明示例：

```
int[][][] matrix = new int[5][15][10];
```

上述声明创建了一个三维数组，其中第一维大小为5，第二维大小为15，第三维大小为10。数组的每个元素将存储一个整数值，数组的元素总数为5 × 15 × 10(即750)。为整个数组分配的内存为750乘上int数据类型的大小。由于int数据类型的大小为4，因此字节分配为3000。

可以使用语法matrix[i][j][k]访问第[i, j, k]个元素。与前面提到的一样，数组元素既可以在运行时初始化，也可以使用数组字面量初始化。同样的概念可以进一步扩展到创建和访问多于3维的数组中。

2.6 非矩形数组

到目前为止，你见到的都是矩形数组的声明与使用。Java还允许创建非矩形数组。非矩形数组是这样一种数组：数组行拥有的列数可能不同。图2-6展示了非矩形数组的内存布局。

如图2-6所示，数组A是拥有4行的非矩形数组。当对数组内存进行分配时，A会指代4个单元的持续内存分配，其中每个单元保存一个指向一维数组的引用。在图2-6中，第1个单元保存一个拥有5个元素的数组引用；第2个单元保存一个拥有2个元素的数组引用；第3个单元保存一个拥有3个元素的数组引用；第4个单元保存一个拥有4个元素的数组引用。

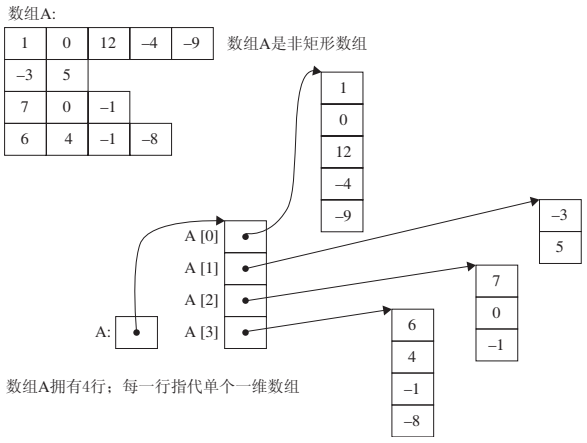


图2-6 非矩形数组的内存布局

1. 在运行时初始化

考虑如下二维数组——第一维大小为5，那么该数组将包含5行。现在可以为每一行定义不同的列大小。例如，第一行可以包含4列；第二行可以包含3列，等等。如下便是这样的声明：

```
int[][] jaggedArray = new int[5][];
jaggedArray[0] = new int[4];
jaggedArray[1] = new int[3];
jaggedArray[2] = new int[5];
jaggedArray[3] = new int[2];
jaggedArray[4] = new int[4];
```

在第一条语句中，声明的jaggedArray是二维数组。在声明过程中，将数组的第一维初始化为5，表示数组将包含5行。这里并没有指定数组的第二维大小。

接下来，初始化jaggedArray的第一行。每行都包含特定数量的int类型的单元。每行中单元的数量为数组中列的长度。初始化该值为4，也就是第1行(ID为0的行)将包含4列。对于第2行(行ID为1)，声明列大小为3，因此第2行将包含3列。类似的，对于第3行，设置列大小为5。第4行列大小为2，第5行列大小为4。

与前面例子中一样，可以使用语法jaggedArray[i][j]访问数组元素。需要确保i与j的值处于数组声明的范围以内。例如jaggedArray[0][3]是有效的，而jaggedArray[1][3]是无效的，因为第2行仅包含3列。

这类非矩形数组也叫做“交错”或“参差”数组，有时还被称作“数组的数组”。请注意，与矩形数组一样(数组每行拥有相同数量的列)，非矩形数组中的每个元素必须具有相同的数据类型。

2. 使用数组字面量初始化

同使用数组字面量初始化一维数组和其他矩形数组一样，也可以使用数组字面量初始化非矩形数组。考虑如下声明：

```
int[][] myArray = {
    {3, 4, 5},
    {1, 2}
};
```

这里，myArray是整型二维数组。数组的第1行由3个元素组成，而第2行仅由2个元素组成。因此，这是非矩形数组。在访问该数组的元素时，需要注意确保对于每一行，两个下标均在数组范围以内。

警告

以下内容需要了解面向对象编程知识，其中会涉及一些关于类方面的高级概念。如果你是这方面的初学者，请略过本章剩余部分，并在学完第3章之后再进行阅读。

2.7 几样好东西

当在Java中声明数组时，数组会被看做定义了有用的属性与方法的内部类。借助这些属

性与方法，可以执行确定数组长度、复制数组内容等其他一些有用的操作。现在，你将学习如何执行这些操作。

2.7.1 确定数组长度

内部数组类的length字段指定了数组的长度。程序清单2-3展示了如何使用该字段。

程序清单2-3 确定数组的长度

```
public class ArrayLengthApp {

    public static void main(String[] args) {
        final int SIZE = 5;
        int[] integerArray = new int[SIZE];
        float[] floatArray = {5.0f, 3.0f, 2.0f, 1.5f};
        String[] weekDays = {"Sunday", "Monday", "Tuesday",
            "Wednesday", "Thursday", "Friday", "Saturday"};
        int[][] jaggedArray = {
            {5, 4},
            {10, 15, 12, 15, 18},
            {6, 9, 10},
            {12, 5, 8, 11}
        };
        System.out.println("integerArray length: " + integerArray.length);
        System.out.println("floatArray length: " + floatArray.length);
        System.out.println("Number of days in a week: " + weekDays.length);
        System.out.println("Length of jaggedArray: " + jaggedArray.length);
        int row = 0;
        for (int[] memberRow : jaggedArray) {
            System.out.println("\tArray length for row "
                    + ++row + ": " + memberRow.length);
        }
    }
}
```

在main方法中，程序声明了若干数组。integerArray是包含5个具有默认初始值的元素的一维数组。floatArray是包含4个初始化元素的浮点型数组。weekDays是String对象数组，初始值为每周的星期几。jaggedArray是初始化的非矩形数组。为了确定每个数组的长度，我们使用语法arrayName.length。程序输出如下：

```
integerArray length: 5
floatArray length: 4
Number of days in a week: 7
Length of jaggedArray: 4
Array length for row 1: 2
Array length for row 2: 5
Array length for row 3: 3
Array length for row 4: 4
```

注意，integerArray数组的元素未被显式初始化，它的长度打印为5；floatArray使用字面

量进行初始化，它的长度为4；weekDays是Strings对象数组，同样使用字面量进行初始化，它的长度打印为7；表达式jaggedArray.length返回二维数组中的行数。每行都被看做数组；因此，为了确定每行的长度，我们使用前面讨论过的for-each循环遍历所有行：

```
for (int[] memberRow : jaggedArray) {
```

注意jaggedArray中的每个元素是如何被看做整型数组的。每行的长度可以使用memberRow.length获得。

2.7.2 复制数组

想要复制数组，可以调用数组对象的clone方法，如程序清单2-4所示。

程序清单2-4 复制数组

```
import java.util.Arrays;

public class ArrayCopyApp {

    public static void main(String[] args) {
        float[] floatArray = {5.0f, 3.0f, 2.0f, 1.5f};
        float[] floatArrayCopy = floatArray.clone();
        System.out.println(Arrays.toString(floatArray) + " - Original");
        System.out.println(Arrays.toString(floatArrayCopy) + " - Copy");
        System.out.println();
        System.out.println("Modifying the second element of the original array");
        floatArray[1] = 20;
        System.out.println(Arrays.toString(floatArray)
                + " - Original after modification");
        System.out.println(Arrays.toString(floatArrayCopy) + " - Copy");
        System.out.println();
        System.out.println("Modifying the third element of the copy array");
        floatArrayCopy[2] = 30;
        System.out.println(Arrays.toString(floatArray) + " - Original");
        System.out.println(Arrays.toString(floatArrayCopy)
                + " - Copy array after modification");
    }
}
```

在main方法中，我们声明了名为floatArray的浮点型数组，并将数组元素初始化为某些值。为了复制该数组，我们使用了如下语句：

```
float[] floatArrayCopy = floatArray.clone();
```

clone方法复制floatArray的所有元素到名为floatArrayCopy的新数组中。为了确认新数组已复制完成，我们将尝试修改两个数组中的每个元素，并将两者各打印一次。为了打印数组内容，我们使用了表达式Arrays.toString(floatArray)，其中的Arrays是Java自Java 2之后提供的类。

Arrays类的toString方法接受一个类型为数组的参数，并将其中的内容转换为字符串。另外一种打印所有元素的方法是使用传统循环遍历数组的所有元素，在每一次迭代循环中进行打印。上述程序首先修改原数组中位于索引位置1的元素，然后修改复制数组中位于索引位置2的元素。每一次修改后均会打印这两个数组。程序输出如下：

```
[5.0, 3.0, 2.0, 1.5] - Original
[5.0, 3.0, 2.0, 1.5] - Copy

Modifying the second element of the original array
[5.0, 20.0, 2.0, 1.5] - Original after modification
[5.0, 3.0, 2.0, 1.5] - Copy

Modifying the third element of the copy array
[5.0, 20.0, 2.0, 1.5] - Original
[5.0, 3.0, 30.0, 1.5] - Copy array after modification
```

注意

clone方法执行的是浅复制而非深复制。浅复制仅仅复制对象的“表层”部分。实际对象不仅包含“表层”，还包含引用指向的所有对象以及由那些对象指向的对象，等等。我们把复制整个对象网叫做深复制。

2.7.3 找出数组的类表示

如前所述，到目前为止我们展示的资料都算是高级知识，需要读者具有面向对象编程的知识。本节展示的资料依然非常高级。我们将这些内容放在这里是为了满足那些想知道数组的类表示形式的高级用户。程序清单2-5回答了该问题。

程序清单2-5 找出数组的类表示

```
public class ArrayClassNameApp {

    public static void main(String[] args) {
        final int SIZE = 5;
        int[] integerArray = new int[SIZE];
        float[] floatArray = {5.0f, 3.0f, 2.0f, 1.5f};
        String[] weekDays = {"Sunday", "Monday", "Tuesday",
            "Wednesday", "Thursday", "Friday", "Saturday"};
        int[][] jaggedArray = {
            {5, 4},
            {10, 15, 12, 15, 18},
            {6, 9, 10},
            {12, 5, 8, 11}
        };
        Class cls = integerArray.getClass();
        System.out.println("The class name of integerArray: " + cls.getName());
        cls = floatArray.getClass();
        System.out.println("The class name of floatArray: " + cls.getName());
```

```
            cls = weekDays.getClass();
            System.out.println("The class name of weekDays: " + cls.getName());
            cls = jaggedArray.getClass();
            System.out.println("The class name of jaggedArray: " + cls.getName());
            System.out.println();
            cls = cls.getSuperclass();
            System.out.println("The super class of an array object: "
                + cls.getName());
        }
    }
```

与前面的程序清单2-3一样，main方法声明了4种不同数据类型的数组。为了获取数组对象的类表示，我们使用了表达式arrayName.getClass()。getClass方法是Object类的方法成员，可以返回给定对象的类表示。Java定义了名为Class的类，用于提供(现有的)类表示(请参考第21章以了解更多关于Class类的介绍)。我们使用如下语句获得Class对象：

```
Class cls = integerArray.getClass();
```

现在，为了打印获取到的类的名称，我们调用Class类对象cls的getName方法。下列语句将类名打印至控制台：

```
System.out.println("The class name of integerArray: " + cls.getName());
```

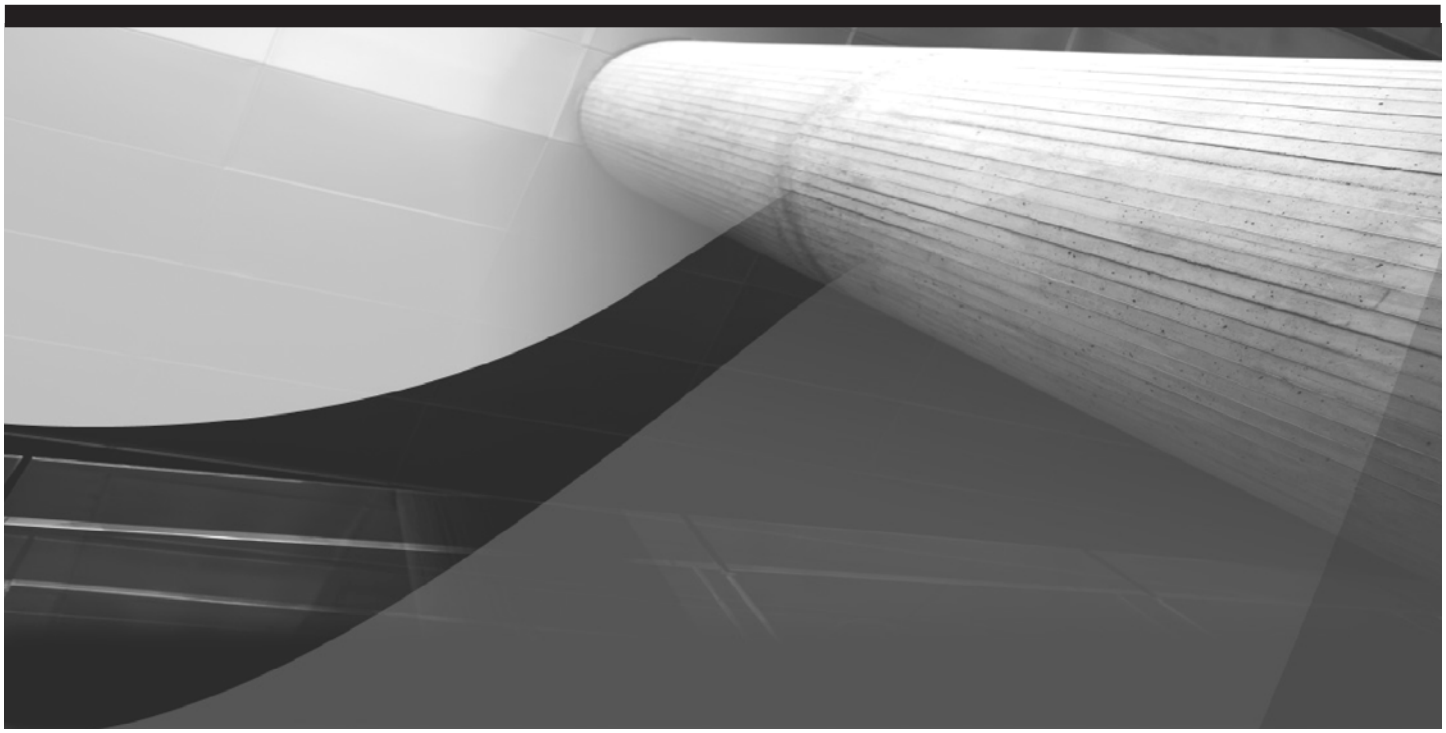
现在，让我们看看程序输出：

```
The class name of integerArray: [I
The class name of floatArray: [F
The class name of weekDays: [Ljava.lang.String;
The class name of jaggedArray: [[I
The super class of an array object: java.lang.Object
```

一维整型数组的类名为[I，二维整型数组的类名为[[I，浮点型数组的类名为[F，String类型数组的类名为[Ljava.lang.String。虽然Java语言规范没有在任何地方提到过数组类名的组成方式，不过我们可以观察到：数组维数由左方括号的数量指定，而元素类型在为原始类型时表现为单个字母。如果使用本地类作为数组元素，那么类的完整限定名称将跟随在方括号之后。对于Object类型的数组，类名为 [Ljava.lang.Object；对于 ArrayClassNameApp类型的数组，类名为[LArrayClassNameApp。

2.8 小结

本章我们讨论了Java中的数组。当想要对一堆具有相同数据类型的条目执行共同操作时，可以使用数组。数组可以让你使用单个变量名访问大量的变量集合。数组元素使用数组名跟上方括号中的数组索引来访问。数组可以包含单维或多维。Java可以让你创建矩形和非矩形数组。在介绍完数组创建和操作的基本知识之后，我们还给出了一些高级知识。我们谈及Java中数组的内部表示，确定数组长度的技巧，复制数组以及获取数组的类表示。



第 3 章

类

使用面向对象编程背后的主要动机是代码重用。Java是一门面向对象编程语言，因此允许重用代码。典型的Java应用程序会包含许多对象，对象是类的实例，而程序有可能包含许多个类。在本章，你将学习什么是类，什么是类的实例，以及如何在Java程序中使用类。在前面几章中，我们已经看到了一些Java程序，不过这些程序只用到了一个类，因此在前面几章中并没有真正利用到面向对象。本章将介绍面向对象编程背后的概念与特性，并更深入地讨论类。此外，本章还会涵盖类的一些特性以及类在Java编程中的用法。

具体学习内容如下：

- 创建类的模板

- 如何声明属性及方法
- 如何定义类成员的可见性
- 如何定义类的构造函数
- 源程序的布局
- package语句的作用
- 如何导入外部类

3.1 面向对象的概念

在自然界中，我们会看到许许多多的对象——鸟类、动物、植物等。每类对象都拥有某些独有的特征。例如，鸟类可以飞翔，动物有4条腿，植物没法移动(除非你帮它从一个位置移到另一个位置)。翅膀可以看做鸟类独有的特性。在面向对象编程中，我们把这种独有的特性称为对象的特性。对象在本质上是真实世界中的实体，拥有自己独有的行为。例如，鸟可以借助翅膀飞翔，那么飞翔便可以描述为对象执行的操作。我们将这类操作叫做对象的方法。面向对象编程中的一些概念正是源于自然界。

在面向对象编程中，我们谈论的对象正是和自然界中各种各样的对象一样。假如我们需要编写程序用于计算公司员工每月的工资，那么可以直接把“员工”看做这类系统中的对象。每一位员工对象都将拥有唯一的ID、姓名和性别，它们是员工对象中公开声明的特征。员工对象还会包含一些附加的特征，如基本工资、绩效工资、社保、假旅费津贴等。当然，我们不想让公司的每一个人都能访问这些数据，因为这些数据都是敏感数据，需要避免其他员工能够查看。另外，在定义功能(比如计算每位员工的月收入)时，我们会考虑将这类功能作为函数实现，用于操作考虑范围内特定员工的数据。

工资系统中每个员工对象持有的数据与工资系统中其他员工对象持有的数据类似。另外，每一个员工对象展现的功能与工资系统中所有其他员工对象展现的功能完全一致。自然而然， 我们会想到创建模板作为员工对象的基础。在文字处理中，我们会为不同的目的创建模板，如简历、发货单、备忘录等。求职者会利用简历模板创建简历并投递给未来的雇主。而每一位求职者都可以使用相同的模板编写简历。在面向对象编程中，我们创建模板用来定义或表征具有共同行为的对象。我们把面向对象编程中的模板称作“类”。类是面向对象编程的真正核心，是整套系统构建的基础。到本章结束时，你将会了解到类的各部分特性。

那么首先，让我们看看面向对象编程的重要特性。

3.1.1 面向对象编程的特性

面向对象编程的三大主要特性是封装、继承和多态。下面我们会详细探讨每个特性。

1. 封装

如前所述，在数据和操作数据的方法(方法是函数或过程的别名，也叫做操作)之间建立联系是很好的做法。其中，数据应当对外隐藏，这意味着数据对当前上下文(更确切地说，是当前对象)之外的代码不可见。我们把信息隐藏的过程以及将数据和方法结合成单一逻辑单元的做法叫做“封装”。我们把上述过程称为将数据以及操作数据的方法封装到名为类的单个单元中。

类由数据(更确切的称呼是属性)和方法构成。它们都叫做类的成员。类的属性应被视为类实例的“私有”部分；只有类的方法才能访问这些属性。在定义类时，应当为这些属性设定访问可见性。它们也许需要对当前类定义的外部代码可见。稍后，我们会在本章深入探讨类时进行更多介绍。目前，你只需要知道我们将数据和相关的方法封装到了叫做类的逻辑单元中。

2. 继承

从前面内容中，我们了解到类包含属性以及操作属性的方法。类的作用就像是模板，可以基于类创建不同的对象。虽然每个对象都拥有自己特定的数据，但是类类型相同的所有对象拥有相同的特征。例如，在我们创建Employee类时，每个Employee对象都将包含相同的数据属性，如ID、姓名、基本工资、社保、绩效工资、假旅费津贴等。赋给这些属性的值在不同的员工间不一样。每个Employee对象展现的功能相同，这些功能由Employee类中的方法定义。一段时间以后，我们可能想要在软件中表示经理。为此，我们只需要创建继承了Employee类特征的Manager类即可。毕竟，经理也是员工。

在自然界中，我们可以观察到，孩子会继承父母的一些特征。这样的家族类有许多，如鸟类、动物、哺乳动物等。每个家族会包含几个对象。同一指定家族中的所有对象共享相同的特征(或者在面向对象编程上下文中，叫做共同功能)。家族中的孩子们从他们的父母继承特征。孩子也可以展现除从父母那里继承之外的特征(功能)。

在软件工程中，当我们开发软件时，会搜寻已有的家族类，就和我们在自然界中观察到的一样。例如，为了在软件应用程序中表示不同类型的车，可以设计名为Vehicle(车辆)的父类。在Vehicle类中定义所有汽车的共同功能。之后可以基于这个父类(Vehicle)进一步定义类，如Car(汽车)、Truck(卡车)等。每一种车都在从Vehicle类继承过来的功能中增加了独有功能。这些添加的功能对于定义的类是独一无二的。这意味着Car和Truck将会为Vehicle添加一些不同于同一家族中其他类的功能。例如，Car可能会将载客量作为描述特征，而Truck可能将最大装载量作为自己添加的特征。汽车还可以进一步分为跑车、SUV等。我们可以为这些分类定义类。SUV类将继承自Car类，而Car类则反过来继承自Vehicle类。类层次结构见图3-1。

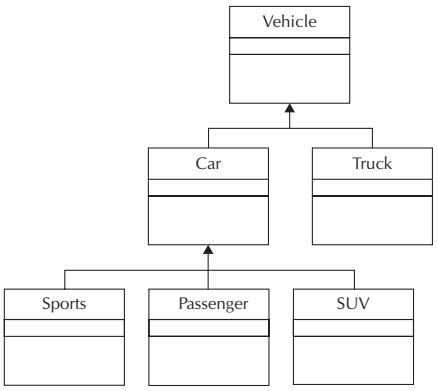


图3-1 类层次结构

继承的概念能够在保持已有代码的基础上，允许我们对已有类的功能进行扩展。

3. 多态

在我们的汽车分类中，Car对象在类层次结构中对应着不同的类。这些Car对象会表现一些共同功能。例如，“drive”方法可以定义在适用于所有Car对象的类中。因为这是共同功能，从而我们应当将之定义在父类中。子类也可能定义“drive”方法，并可能修改继承而来的“drive”方法。两个方法都可以使用同样的名字(即drive)。虽然开卡车、开跑车、开SUV是不一样的，但是我们可以说，对于所有车辆“我们都是在开车”。因此，即使不同对象中

的实现不一样，它们的功能名称也依然相同。此特性在面向对象术语中被称为“多态”。多态(polymorphism)这个词语起源于希腊词汇polymorph，意思是相同对象拥有不同的表象。多态是面向对象语言的重要特性之一，我们将在第4章中深入讨论。

3.1.2 面向对象编程的好处

既然我们已经了解了面向对象编程的三大主要特性，下面让我们看看面向对象编程如何能够帮助我们创建结构良好的程序，使得代码一旦开发完毕就可被轻易重用。程序如果能够借助少量工作轻易地实现扩展，就可以使我们重用已有代码(先前已经被测试过)，继而减少软件的维护成本。

3.2 类

类是模板，其中封装了数据以及操作数据的方法，比如前面描述的Employee类和Automobile类。类是一种用户自定义的数据类型，可以内嵌方法以操作内部封装的数据。对象可以基于该模板创建，在Java中，new关键字用于创建对象(我们将这一过程称作类的实例化)。可以从单个类创建多个对象，其中所有的这些对象拥有相同的类型，即预定义的数据成员集合。这些数据成员保存的值因对象的不同而不同。尽管每个对象与其他相同类型的对象较为相似，但是它们都有着自己的独有标识符。例如，两辆具有相同类型和颜色的汽车会拥有不同的车辆识别码以用作它们独有的标识符。

3.2.1 定义类

定义类的一般形式如下：

```
Modifiers可选 class ClassName{
    classbody可选
}
```

注意

这里采用的是一种伪标记法。Java语言规范(JLS)在JLS标记法中定义了完整的语法。下面的部分类定义采用JLS格式来表示：

```
ClassModifiers可选 class Identifier ClassBody
```

JLS非常复杂，如果你想了解更多信息，可以从http://java.sun.com/docs/books/jls/下载Java语言规范。

定义类需要使用class关键字。ClassName是合法标识符，用于为类定义唯一的名称。该名称应当在整个应用程序范围内保证唯一。但是，在两个独立应用程序或Java包中，可以为类使用相同的名称。

注意

每个类都拥有类似packagename.ClassName的完全限定名称。在不同的packagename中，ClassName可以重复。关于Java包的这部分内容，我们将在第5章详细讲解。

class关键字之前的修饰符用于定义类的可见性。目前请暂时不用担心修饰符方面的问题，我们稍后将在本章及后续章节中进行详细介绍。

如下是典型的类声明：

```
class MyClass {

    // attributes
    // constructors
    // methods
}
```

注意

一个类的声明中可能会包含另一个类的声明，我们把被包含的类称为外层声明类的成员类。

类定义包括在一对花括号内。在类的定义主体中，可以定义0个或多个属性、0个或多个构造函数、0个或多个方法。类的属性也就是指类的数据成员，它们也被叫做“字段”。字段提供了类及其对象的状态。例如，在前面谈到的Employee类中，ID即为Employee类的字段。构造函数是类的一种特殊方法，在实例化新对象时会用到。类也可以包含其他方法，用于定义类及其对象的行为。

注意

Java语言规范中使用“字段”指代“属性”。因此，今后若无特殊说明，术语“字段”即表示类的属性。

让我们把注意力转回到定义类的字段、构造函数及方法上。首先，我们会以仅包含字段而没有方法的简单类为例。而后，我们会在这一类模板的基础上添加更多功能。

3.2.2 定义Point类

下列代码段声明了Point类：

```
class Point {

    int x;
    int y;
}
```

该类使用关键字class并紧跟类名Point定义而成。由于修饰符在类定义时是可选的，因此并未包含在上述定义中。修饰符用于定义类的可见性，我们稍后将讨论修饰符的一些可选值。类的主体包含在花括号中。Point类的主体仅包含了字段部分。

注意

C++中的类定义需要以分号结尾，而在Java中，不用强制使用分号结尾。

我们的Point类声明中包含了两个类型为int的字段——x与y。这个简单的Point类声明并没有包含任何方法。如前所述，方法定义了类的功能。Point类目前并没有展示任何功能。

从上面的例子可以看出，可以创建不包含任何方法而仅包含字段的类。同样，也可以创

建主体为空的类——没有字段、没有构造函数且没有方法。不过，创建这种类型的类通常没有意义，除非是为了名称考虑，比如提供给其他类进行继承。

注意

对于顶层的类，比如前面提到过的Vehicle类或Employee类，类的主体可能为空。而对于从这些顶层类继承而来的类，将在它们的定义中增加字段和方法。

3.2.3 使用类

类定义的作用就像是一套模板，可以在应用程序中借助类定义创建对象。例如，通过使用Point的类定义，可以创建一些Point对象。每个Point对象的特征体现在两个不同的字段x和y中，它们用于指定点的x坐标与y坐标。每个Point对象都拥有自己的数据成员x与y。这些成员也叫做对象的实例变量(instance variable)，因为它们隶属于类的某个特定实例。

我们把从类定义创建对象的过程称为类的实例化。创建对象时，类即被实例化。下列声明用于创建对象：

```
Point p = new Point();
```

我们使用new关键字实例化类。在new关键字之后，我们指定类名并紧跟一对括号。其中的括号对表明这是方法调用——调用构造函数。我们稍后会在本章中介绍类的构造函数。类一旦被实例化，类的对象就会被分配内存以保存数据。指向内存分配的引用必须被复制和保存在某处，从而使得创建的对象可以在后面的程序代码中被访问到。在前面的声明中，我们把内存引用复制到了赋值运算符左边声明的变量p中。变量p的类型为Point，表明p保存的是指向Point类型对象的引用。

前面的语句在执行时会在运行时创建Point类型的对象。Point对象包含两个字段——x与y。对象将通过变量p进行引用。x与y这两个字段的默认数值为0。字段的默认赋值取决于字段自身的类型。

3.2.4 访问/修改字段

访问前面例子中的字段很简单。可以使用p.x语法访问点p的x坐标，使用p.y语法访问点p的y坐标。访问字段的一般形式是objectReference.fieldName。在我们的例子中，objectReference是p，而fieldName是y或x。

3.2.5 类的示例程序

我们现在将编写Java程序，定义Point类、对Point类进行实例化并在应用程序中进行使用。程序清单3-1给出了声明和使用Point类的完整程序。

程序清单3-1 介绍如何使用类的示例程序

```
class Point {

    int x;
    int y;
}
```

```
class TestPoint {

    public static void main(String[] args) {
        System.out.println("Creating a Point object ... ");
        Point p = new Point();
        System.out.println("Initializing data members ...");
        p.x = 4;
        p.y = 5;
        System.out.println("Printing object");
        System.out.println("Point p(" + p.x + ", " + p.y + ")");
    }
}
```

程序的输出结果如下：

```
C:\360\ch03>java TestPoint
Creating a Point object ...
Initializing data members ...
Printing object
Point p(4, 5)
```

注意

你需要在命令行中指定TestPoint以运行代码，因为main方法定义在TestPoint中。

在程序清单3-1中，Point类的定义与我们之前讨论的一致。为了测试Point类，需要编写另外一个类。上述程序定义了TestPoint类，用于测试Point类。TestPoint类声明了main方法，main方法是程序开始的执行点。在main方法主体中，我们使用下列语句创建Point类的实例：

```
Point p = new Point();
```

我们使用下列语句访问创建对象的x与y字段：

```
p.x = 4;
p.y = 5;
```

上述语句对数据成员(也就是字段)的值进行了设置。如前所述，这些成员也叫做实例变量，因为它们属于类的某个特定实例。上例中的p即为实例。如果我们创建另外一个实例(比方说p2)，那么p2也会拥有自己的x与y字段。这里，我们使用objectReference.fieldName来访问字段。两条语句分别对两个字段进行赋值。为了验证赋值结果，我们使用下列两行代码打印对象内容：

```
System.out.println("Printing object");
System.out.println("Point p (" + p.x + ", " + p.y + ")");
```

这里，我们使用了与前面相同的语法p.x与p.y，用于检索实例变量。

3.2.6 声明方法

在上面段落中，声明的Point类仅包含字段。现在，我们将向Point类添加方法，以帮助你了解函数声明及其调用语法。向类定义添加方法的主要目的是为类提供一些功能。在我们

将要为Point类添加的功能中，包含计算点到原点的距离。因此，我们会添加名为getDistance的方法，用于返回点到原点的距离。

修改后的程序见程序清单3-2。

程序清单3-2 展示在类中如何声明方法的程序

```
import java.util.*;

class Point {

    int x;
    int y;

    double getDistance() {
        return(Math.sqrt(x * x + y * y));
    }
}

class TestPoint {

    public static void main(String[] args) {
        System.out.println("Creating a Point object ... ");
        Point p1 = new Point();
        System.out.println("Initializing object ...");
        p1.x = 3;
        p1.y = 4;
        double distance = p1.getDistance();
        StringBuilder sb = new StringBuilder();
        Formatter formatter = new Formatter(sb, Locale.US);
        formatter.format("Distance of Point p1(" + p1.x + "," + p1.y
                + ") from origin is %.02f", distance);
        System.out.println(sb);
        System.out.println();
        sb.delete(0, sb.length());
        System.out.println("Creating another Point object ... ");
        Point p2 = new Point();
        System.out.println("Initializing object ...");
        p2.x = 8;
        p2.y = 9;
        distance = p2.getDistance();
        formatter.format("Distance of Point p2(" + p2.x + ","
            + p2.y + ") from origin is %.02f", distance);
        System.out.println(sb);
    }
}
```

当我们编译并运行程序时，可以看到下列输出：

```
C:\360\ch03>java TestPoint
Creating a Point object ...
```

```
Initializing object ...
Distance of Point p1(3,4) from origin is 5.00

Creating another Point object ...
Initializing object ...
Distance of Point p2(8,9) from origin is 12.04
```

现在，我们为Point类定义添加了一个方法。getDistance方法计算当前点到原点的距离，并返回double值给调用方：

```
doublegetDistance() {
    return(Math.sqrt(x * x + y * y));
}
```

距离采用内置的Math类的sqrt方法计算而得。你将会在本书中学习到许多这样的内置类。

在TestPoint类的main方法中，我们首先创建Point类的实例p1，而后使用前面示例中的语法对数据成员进行初始化。接下来，我们调用getDistance方法计算当前点到原点的距离：

```
double distance = p1.getDistance();
```

我们使用objectReference.methodName语法来调用方法。在我们的例子中，objectReference是指向Point对象的引用，也就是p1，而methodName是getDistance。为了输出对象内容以及结果距离，我们使用了StringBuilder类(之前在Syntax Reference 2中用到过)：

```
StringBuildersb = new StringBuilder();
Formatter formatter = new Formatter(sb, Locale.US);
formatter.format("Distance of Point p(" + p1.x + ","
        + p1.y + ") from origin is %.02f", distance);
System.out.println(sb);
```

注意

你还可以使用前面章节介绍的System.out对象的printf方法来代替这里的Formatter类。

在构建完字符串之后，我们将结果打印到控制台。为了演示在应用程序中可以创建多个Point对象，我们创建了Point类的另一个实例p2。程序对p2的字段进行了初始化，并通过调用getDistance方法计算p2到原点的距离，将结果和对象内容输出至控制台。如你所见，Point类的两个对象可以展现同样的功能(getDistance)。因此，方法可以为所有具有相同类类型的对象定义通用的功能。

注意

StringBuilder类的delete方法用于清除对象内容，其中的第一个参数指定开始位置，第二个参数指定要清除字符的数量。

警告

C++允许在类声明的外部定义方法体，但在Java中，你必须在类声明的内部定义方法体。

3.2.7 对象的内存表示

Point对象包含两个类型为int的字段——x与y。存储这些字段的空间在对象内存空间内分配。对象占据的空间往往大于其中字段占据的空间，因为对象还保存了一些隐藏的信息用于表明类型。Point对象的内存表示见图3-2。

可以通过声明一些类型为Point类的变量，并将Point类的实例赋值给它们中的每一个来创建多个Point对象。下面的代码段创建了3个Point对象：

```
Point p1 = new Point();
Point p2 = new Point();
Point p3 = new Point();
```

对于每一个变量声明，都会被分配一份独立的内存块，如图3-3所示。

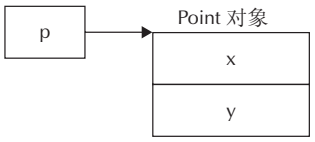


图3-2 单个对象的内存分配

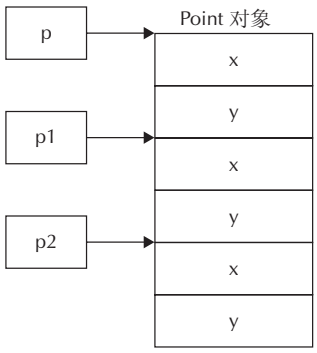


图3-3 多个对象的内存分配

注意

每一个Point变量声明都会接受自己的实例变量x与y的副本。

警告

虽然对于图3-3中的3个对象来说，内存布局被连续分配，但事实并非总是如此。

3.3 信息隐藏

在我们的日常生活中，大部分人都常常设法对其他人隐藏信息。在许多情况下，信息隐藏是我们生活中至关重要的一部分。例如，当你去杂货店购物时，你会让商店售货员从你的兜里取钱吗？钱包(可看做面向对象中的对象)对外隐藏了金钱(金钱是对象的信息或属性/字段)。钱包可以提供称作“拿出X美元”的方法，当这个方法被外部对象执行时，结果会将X美元给予调用方而无须公开钱包里有多少钱。这在面向对象编程里叫做信息隐藏。现在让我们通过一个实际的例子来介绍这些概念如何工作。程序清单3-3包含了Wallet类以及从Wallet对象中取钱的Person类。

程序清单3-3 展示信息隐藏概念的程序

```
class Wallet {

    private float money;

    public void setMoney(float money) {
        this.money = money;
    }
    public boolean pullOutMoney(float amount) {
        if (money >= amount) {
            money -= amount;
            return true;
```

```
        }
        return false;
    }
}

class Person {

    public static void main(String[] args) {
        Wallet wallet = new Wallet();
        System.out.println("Putting $500 in the wallet\n");
        wallet.setMoney(500);
        System.out.println("Pulling out $100 ...");
        boolean isMoneyInWallet = wallet.pullOutMoney(100);
        if (isMoneyInWallet) {
            System.out.println("Got it!");
        } else {
            System.out.println("Nope, not enough money");
        }
        System.out.println("\nPulling out $300 ...");
        isMoneyInWallet = wallet.pullOutMoney(300);
        if (isMoneyInWallet) {
            System.out.println("Got it!");
        } else {
            System.out.println("Nope, not enough money");
        }
        System.out.println("\nPulling out $200 ...");
        isMoneyInWallet = wallet.pullOutMoney(200);
        if (isMoneyInWallet) {
            System.out.println("Got it!");
        } else {
            System.out.println("Nope, not enough money");
        }
    }
}
```

在编译并运行程序时，我们会看到如下输出：

```
C:\360\ch03>java Person
Putting $500 in the wallet
Pulling out $100 ...
Got it!
Pulling out $300 ...
Got it!
Pulling out $200 ...
Nope, not enough money
```

就像你期望的那样，Wallet类声明了名为money的字段。自然地，该字段的类型被设置为float以让钱包可以保留一些零钱：

```
class Wallet {

    private float money;
```

请注意字段声明前的private关键字，之前我们提到过这个访问修饰符。访问修饰符控制声明中附加部分的可见性。在这个例子中，private访问修饰符附加到字段声明上。还可以将访问修饰符附加到类和访问声明上，其中一个例子就是我们到目前为止一直在使用的main方法。main方法有一个public访问修饰符附加其上。在Syntax Reference 1中第一次使用main方法时，我们讨论过将main方法声明为public，进而使得main方法可以被JVM调用——或者更确切地说，由外部对象调用。main方法被声明为public是强制性的，但类定义中的其他方法则不需要一定为public，例如前面Point类中的getDistance方法就不是public(注意，如果不指定访问修饰符，可见性默认为包私有，比public可见性的范围要低)。我们还可以将访问修饰符应用到类声明中。随着你阅读剩下的章节部分，你会学习到更多关于private与public的相关知识。

回到Wallet类的定义，类中定义了名为setMoney的方法，如下所示：

```
public void setMoney(float money) {
    this.money = money;
}
```

该方法接受一个float参数，并且没有返回值给调用方。方法被声明为public，这使得方法可以被外部对象调用，很快你就会看到这点(显然，在你希望买东西时，需要用钱包带上一些钱)。

在方法内部，复制money的值(作为参数接收)到具有相同名称的实例变量money中。由于方法参数与实例变量使用相同名称，我们需要在代码中进行区分。而这项工作可借助this关键字完成。this关键字是对当前对象的引用，因而语法为“this.”。点之后跟上对象引用，可以指代对象的一些变量或方法。this.money指代当前对象的money字段。我们设置money字段的值为传入的方法参数。setMoney方法帮助在我们的钱包中放入一些钱。附带提一下，我们把这类方法叫做setter或存值器(mutator)方法，因为它们设置字段的值。setter方法按照惯例会以set打头，后面跟上操作的字段名称，其中字段的首字母大写。与setter方法类似，getter方法以get打头，并返回字段值给调用方。我们会在接下来的程序中用getter方法。

现在，让我们看看在Wallet类中定义的下一个方法。pullOutMoney方法以float值作为参数，并返回布尔值给调用方：

```
public boolean pullOutMoney(float amt) {
```

在方法实现中，我们检查需要的额度是否低于钱包中的当前余额。如果低于，就从余额中减掉需要的额度并返回true给调用方：

```
if(money >= amount) {
   money -= amount;
   return true;
}
```

如果资金不足，就返回false给调用方。

注意pullOutMoney方法声明包含了public访问修饰符，表示该方法可以被外部对象访问。

现在让我们把注意力放到使用Wallet的Person类。在Person类的main方法中，我们首先创建Wallet类的一个对象：

```
Wallet wallet = new Wallet();
```

通过调用setMoney方法向钱包中放钱：

```
wallet.setMoney(500);
```

该语句将wallet对象的money变量设置为500。setMoney方法由于被声明为public，因此可以被Person类中的代码调用。如果先前使用的是private修饰符，那么编译器在编译Person类时会报告“非法访问”错误。

现在，让我们试着从钱包中取出一些钱。调用wallet对象的public方法pullOutMoney。由于该方法像setMoney一样被定义为public，因此可以被外部类的代码调用：

```
boolean isMoneyInWallet = wallet.pullOutMoney(100);
```

我们根据返回值向用户控制台打印了对应的消息。要注意的是，pullOutMoney方法并没有显示当前余额(因而实现了信息隐藏)。接下来，我们进行第2次和第3次取钱。第2次取了300美元，而第3次取了200美元。注意这些取钱操作都没有显示钱包中的余额。事实上，在最后一次取钱失败时，仍然没有显示还有多少钱可以被取出——这是一种有效的信息隐藏实现。

现在，如果想要知道钱包中还有多少余额，该怎么办呢？如果是这样的话，可以提供public可见性的getter方法，用来读取字段的值。getter方法的声明见如下代码段：

```
public float getMoney() {
    return money;
}
```

注意

getter方法也叫做访问器(accessor)方法，因为它们可以用于访问属性值。

getMoney方法使用标准的getter方法约定——get之后跟上属性名，并且属性名的首字母大写。该方法不接受任何参数，并返回float值给调用方。该实现简单地返回了money属性的当前值。该方法被声明为public以便能够被外部对象调用。 现在，在代码中，我们可以使用wallet.getMoney()语法读取当前余额。

提示

根据面向对象编程的准则以及良好的设计实践，类中定义的所有字段都应当被定义为private，你应当提供必需的public getter/setter方法以访问它们。

现在的问题是，为什么我们需要应用访问修饰符到声明中？到目前为止，你已经看到了两种访问修饰符——private与public。private访问修饰符使得对应的字段、方法或类声明真正地对所属类私有。在我们的例子中，money变量被定义为private，从而只对Wallet类内部的代码可见。因此，使用前面提到的点语法(wallet.money)将会导致编译错误。访问money字段的唯一方法是使用操作该属性的public getter/setter方法。设置money字段私有还可以帮助保护money字段免受外部代码的意外修改，因为外部代码无法使用objectReference.fieldName访问money字段。

3.4 封装

在前面内容中，你看到了信息隐藏的重要性。另一个与面向对象编程相关的重要概念是封装。每一个类在自己的字段中都会包含一些数据。通过定义它们为private可以隐藏这些字段，并定义getter/setter方法来对它们进行访问。在为字段赋值时，赋值前我们会对值进行验证吗？通过提供setter方法，我们获得了提供验证的机会。作为例子，考虑如下声明的Date类：

```
class Date {

    publicint day;
    publicint month;
    publicint year;
}
```

该方法声明了三个公有字段。

虽然这与面向对象编程的最佳实践相悖，但是将这些字段声明为public主要是为了说明public声明带来的相关问题。

如果d是Date类的对象，就可以使用如下语句设置d的month值：

```
d.month = 13;
```

虽然上述代码语法正确并且执行没有错误，但这显然是赋值错误，可能是由于简单的打字错误造成。为了防止这类不太可能避免的错误，请确保将字段设置为private，并定义setter方法用于对字段进行赋值，另外还可以在setter中(在赋值前)提供验证。因此，我们可以按照如下方法为month字段编写setter方法：

```
public void setMonth(int month) {
    if (month >= 1 && month <= 12) {
        this.month = month;
    } else {
        // print an error message to the user
    }
}
```

该实现检查我们试图赋予的值是否在公历月份范围内，这可以确保非法值不会被赋值给month字段。类似的检查也可以施加在day与year字段的setter方法中。

将数据与操作数据的方法一起放在类定义的过程叫做封装。如你所见，封装getter/setter方法可以帮助创建鲁棒的对象。在谈论封装时，我们指的不仅仅是getter/setter方法，类中的其他方法在逻辑上也可以归为此类。例如，让我们看看printDate方法，该方法使用某种预定义的日期格式打印date属性的值。在Date类中声明和定义printDate方法很有道理，而这正是封装的意义所在。可以在类定义中封装操作类属性以及对于类定义合乎逻辑的方法。例如，Date类可以提供一些方法，用于判断当前日期在经过一段日期之后是否处于闰年中，等等。这些方法都应当定义在属于Date类的方法中。

以下是其他一些可以考虑使用封装的场景：如果不在类定义中封装相应的方法，就需要在应用程序中的多个地方提供错误检查代码。例如，如果不在Date类的setter方法中提供验

证，就需要在应用程序中的多个地方重复验证代码。另外，如果验证代码发生改变，就需要做大量工作来确保修改后的代码出现在每一处设置字段值的地方。

注意

作为面向对象系统的设计人员，你需要为所有的类定义找出和提供相应数据与方法的封装。

3.5 声明构造函数

构造函数是一类特殊的方法，会在对象创建过程中由运行时执行。在前面内容中，可以看到如下调用类构造函数的语句：

```
Point p = new Point();
```

当你使用这里显示的声明实例化类时，对象创建后的字段会被赋予一些默认值。尽管如此，你仍然希望创建的对象具有初始状态，并不同于编译器设置的默认状态。例如，你想要创建一个Point对象，其初始值x与y分别设置为4和5。这可以借助构造函数来完成。可以使用下列方法创建构造函数：

```
public Point() {
    x = 4;
    y = 5;
}
```

该方法的名称与类名相同，这是定义构造函数时的一条规则。另外要注意的是，该方法没有声明返回类型，甚至连void都不是。这是定义构造函数时的第2条规则——构造函数没有返回类型。稍后将总结构造函数的声明规则。

现在，让我们看看构造函数的其他特性。例子中的构造函数没有包含任何参数。一般来说，构造函数可以接受0到多个参数。如果构造函数不指定参数，就称之为无参构造函数。有的时候，我们将之称为默认构造函数或不带参数的构造函数。

在构造函数体中，我们放入了两条赋值语句，用于初始化x与y字段为目标值。可以为它们赋予任何选择的值。当创建对象时，对象将会拥有那些值。

一旦在类定义中编写类似这样的构造函数，只要类被实例化，构造函数就会被调用。程序清单3-4展示了这一点。

程序清单3-4 展示带有构造函数的类的程序

```
class Point {

    private int x;
    private int y;

    public Point() {
        x = 10;
        y = 10;
    }
```

```
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}

class CustomConstructorApp {

    public static void main(String[] args) {
        System.out.println("Creating a Point object ... ");
        Point p = new Point();
        System.out.println("\nPrinting Point object");
        System.out.println("Point p (" + p.getX() + ", " + p.getY() + ")");
    }
}
```

同前面的例子一样，Point类声明了两个int类型的字段x与y。Point类同样声明了构造函数，如下所示：

```
public Point() {
    x = 10;
    y = 10;
}
```

构造函数将这两个字段的初始值均设为10，它是一个无参构造函数。Point类同样定义了两个getter方法用于访问这两个字段的值。注意，这里并没有定义setter方法，因为我们的程序除了在构造函数中需要为字段设置默认值外，并不需要对字段的值进行设置。

在CustomConstructorApp类的main方法中，我们实例化了Point类，如下所示：

```
Point p = new Point();
```

语句执行会调用类定义中的构造函数。构造函数设置x与y字段的值为10。可以使用下列语句输出对象的内容以进行验证：

```
System.out.println("Point p (" + p.getX() + ", " + p.getY() + ")");
```

请注意获取x与y值的getter方法。运行程序可得到如下输出：

```
Creating a Point object ...
Printing Point object
Point p (10, 10)
```

在构造完对象后，还可以调用对象的setter方法(我们已在类定义中提供)以改变数据成员的值。通过提供无参构造函数，可以确保任何新创建对象的数据成员的值在构造函数中被设置为默认值。

现在，如果想要每次创建对象时为数据成员设置不同的值，该怎么办？要做到这一点，需要将值作为参数传递给构造函数。因此，需要定义带有参数的构造函数。下列代码段展示了这种构造函数：

```
public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
```

将两个参数指定的值赋值给this指代的当前对象的两个实例变量。下面的语句显示了如何在程序代码中调用该构造函数：

```
Point p = new Point(4, 5);
```

当调用构造函数时，为参数设置值。实例变量现在使用这些值进行初始化。可以使用前面例子中的方法输出对象的内容以进行验证。

有时，我们或许只想要初始化实例变量中的一个，而让另一个拥有默认值。在这种情况下，可以声明Point类的另一个构造函数，如下所示：

```
public Point(int x) {
    this.x = x;
    y = 10;
}
```

该构造函数只接受一个参数，并将参数值赋值给x字段。另一个实例变量为y，将它的值设置为10。如果没有为变量y赋值的话，那么y的值将为编译器提供的默认值0。现在可以使用下列语句调用构造函数：

```
Point p = new Point(5);
```

该语句创建了Point对象(5, 10)，其中的内容可以通过打印对象的内容来进行验证。

注意

不能为相同的类编写两个具有相同参数数量及参数类型的构造函数，因为这样一来，平台将无法对它们进行区分，从而引发编译时错误。

类可在定义中声明多个构造函数。既然如此，那么如何才能在类被实例化时知道哪个构造函数被调用了呢？编译器会根据参数数量及参数类型决定调用哪个构造函数。请看下面的代码段：

```
Point p1 = new Point();
Point p2 = new Point(15);
Point p3 = new Point(5, 10);
```

第1条语句调用无参构造函数并创建对象p1(10, 10)，第2条语句创建对象p2(15, 10)，而第3条语句创建对象p3(5,10)。因此，编译器已根据传入的参数数量决定了调用哪个构造函数。

3.5.1 默认构造函数

在前面，你学会了如何编写自己的构造函数。而在此前，我们未曾编写过任何构造函数。那么类是否提供了默认的构造函数呢？答案是肯定的。

如果你不编写构造函数，Java编译器会提供无参构造函数，我们将之称为默认构造函数。默认构造函数没有实现，也就是说不包含任何代码。

警告

如果你提供了自己的构造函数，那么不管它在类定义中是否有参数，编译器都不会再生成默认构造函数。

注意

与C++不同的是，Java并不提供析构函数。对于由那些不再用到的对象占用的资源，Java依赖垃圾收集器进行清理。

关于构造函数的进一步细节以及它们如何被调用的内容，请参见第5章。

3.5.2 构造函数的定义规则

创建构造函数的规则可以总结如下：

- 构造函数的名称必须与类名相同。
- 构造函数没有返回值，返回类型也并非void。
- 构造函数可以包含0到多个参数。
- 如果你提供了自己的构造函数，那么不管它在类定义中是否有参数，编译器都不会再提供默认构造函数。

3.6 源文件布局

到目前为止，你已经从示例中学习了关于如何编写Java程序的足够知识。现在，让我们讨论一下Java源文件的完整结构。当编写Java源程序时，需要遵循一种特定的结构或模板。程序清单3-5展示了这种模板。

程序清单3-5 Java源程序的布局

```
/**
 * NewClass.java
*/
package javaapplication;
import classes;
public class NewClass {
```

```
    public NewClass() {
    }
}
```

import语句在编译期间可以让外部类的声明对当前Java源程序可见。

本书前面曾提到，可以使用任何文本编辑器编写Java源程序。整个程序可能由多个源文件组成。每个Java源文件的名称必须和它所定义的public类名相同。每个Java源文件仅可以包含一个public类声明。文件扩展名必须为.java。由于文件名大小写敏感，因此前面的源代码必须存储在名为NewClass.java的文件中。虽然源文件可以包含多于一份的类声明，但是只有一份这样的声明可以为public。

源文件包含3个主要部分——package、import和class的定义，另外还有注释部分，你可以在源代码中的任何部分嵌入注释。程序清单3-5的顶部展示了多行注释，显示了代码必须被保存成的文件名。由于class声明之前已经讨论过，因此接下来我们将注意力放到package与import部分。

3.6.1 package语句

package语句可以让你将相关的类逻辑性地分组到单个独立单元中。在我们早期的程序中，使用过import语句，如下所示：

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

上述语句中的java.io与java.util都是包。所有与I/O相关的类都分组放在java.io包中。类似地，所有的实用类都分组放在java.util包中。Java开发工具包定义了许多这样的包，这些包将各种各样的类按照它们的功能分组到不同的逻辑单元中。当在应用程序中开发一些类，或当创建一些应用程序时，也会想将这些类安排到不同的逻辑单元中。要完成该工作，可以在应用程序中创建包。为了创建包，需要使用package语句。package语句的基本语法如下：

```
package packagename;
```

包名可以包含0到多个使用点分隔的子包名。下面是一些合法的包声明：

```
package mypackage;
package mypackage.reports;
package mypackage.reports.accounts.salary;
```

第一条语句声明了顶层包mypackage。第二条语句在mypackage内声明了名为reports的子包。第三条语句声明了一个包层次，其中mypackage位于顶层，reports包位于mypackage内，而accounts包位于reports内，最后的salary包位于accounts内。

当声明包时，必须遵循一些特定规则：

- 包必须早于任何其他语句在源文件的开头进行声明。

注意

注释可以放在包语句之前，因为注释不被看做程序语句。

- 源文件只允许包含一条包声明。
- 包名必须为层次结构并由点分隔。
- 如果没有声明包，编译器会创建默认包，而所有不包含包声明的类都会被放入默认包中。

第一条规则是说，包声明必须为源文件的顶层语句。在源程序的其他地方编写package声明语句会导致编译时错误。

第二条规则是说，不能在单个源文件中拥有多于一条的包声明。注意源文件可能包含多条类声明。尽管如此，源文件仅仅能包含一条package声明。所以，定义在源文件中的类将会被分组到源代码顶层声明的包中。

第三条规则是说，如果决定使用子包，那么整个名称必须按照层次化进行组织，并且子包名称必须由点分隔。可以想象层次包名为目录结构，其中包含主目录，然后主目录包含子目录，直到完整包名结束。包中的类将会被放进整个目录结构的最后一个子目录中。源程序的目录布局包含package语句，我们将在后面进行介绍。

最后一条规则是说，如果源文件没有包含package声明，那么所有定义在源文件中的类都将会被分组到默认的包中。用于默认包的目录就是当前目录本身。因此，所有定义在源文件中的类在编译时必须被放入当前目录。

注意

package语句包含的意义不仅仅是对类进行逻辑分组，另外还控制定义在源程序中的类、字段以及方法的可见性。我们会在第5章介绍继承与可见性时讨论这些可见性规则。

3.6.2 import语句

紧接在package声明之后的是import声明。使用import语句可以告诉编译器在编译时去哪里找到源程序需要的外部类。import语句的完整语法如下：

```
import packagename;
```

或者

```
import packagename.* ;
```

以下是import语句的一些示例：

```
import mypackage.MyClass;
import mypackage.reports.accounts.salary.EmployeeClass;
import java.io.BufferedWriter;
import java.awt.*;
```

第一条语句导入mypackage包中的MyClass类定义。第二条语句导入mypackage.reports.accounts.salary包中的EmployeeClass类定义。第三条语句导入JDK中java.io包提供的BufferedWriter类。最后一条语句导入所有属于java.awt包的类。请注意最后一条语句中的星号(*)表示所有类都会被包含。

正如语法所示，可以导入包中的单个类或所有类。可以指定类名来导入单个类，而通过指定星号*可以导入所有类。

注意

import语句为编译器指定了寻找指定类的路径。事实上，import语句与C或C++中的#include语句不同，import语句并不会加载代码。因此，带有星号*的import语句并不会影响应用程序运行时的性能。

3.7 目录布局和包

你已经学会了如何创建Java包以及如何导入这些包中定义的类。现在，让我们看看包的层次结构。如前所述，包遵循了一种层次化的结构。当编译源程序时，编译器根据包中的定义生成文件夹结构，并在最里层的子文件夹中为程序创建.class文件。为了指示编译器生成文件夹层次，需要在编译器命令行中指定-d选项。例如，如果源程序叫做Employee.java，其中包含如下package语句：

```
package mypackage.reports.accounts.salary;
```

当编译源程序时，请使用如下命令行：

```
C:\360\ch03>javac -d . Employee.java
```

-d选项会告诉编译器在由命令行后-d选项指定的目录下创建文件夹层次结构。在这个例子中，起始目录是点(.)，表示这就是当前工作目录。因此，当编译器编译程序时，会在当前工作目录下创建如图3-4所示的文件夹层次，并在salary子文件夹下创建.class文件。

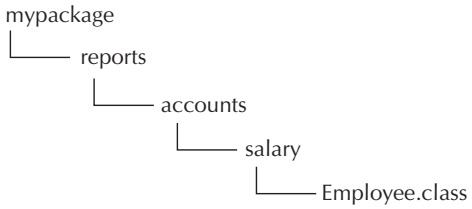


图3-4 编译代码的文件夹层次

如果编译时不指定-d选项，那么编译器会在当前目录中生成Employee.class文件。你将需要自己创建文件夹结构，并将.class文件复制到salary文件夹中。可以使用如下命令运行.class文件(假定其中包含main方法)：

```
C:\360\ch03>java mypackage.reports.accounts.salary.Employee
```

请注意，需要在运行.class文件时使用完整的文件夹层次。这是因为Employee.class的完全限定名称是packagename.ClassName。如果在运行Employee类时不指定包名，那么你会在运行时碰到“无法找到类”的错误，如下所示：

```
C:\360\ch03>java Employee          // generates runtime error
```

如果定位到salary文件夹，并试图在salary子文件夹中运行Employee类，那么在运行时会生成错误，如下所示：

```
C:\360\ch03\MyPackage\Reports\Accounts\Salary>java Employee     //error
```

警告

如果在Java源文件中使用package语句访问创建的.class文件，就必须使用完整限定类名。

提示

当引用定义在包中的某个类时，类的完整限定路径必须对当前文件夹可见。假定当前目录定义在CLASSPATH环境变量中，可以使用操作系统提供的工具修改CLASSPATH环境变量。另外一种方法是：可以在运行应用程序时，在命令行指定CLASSPATH。

3.8 小结

Java是一门面向对象编程语言。Java程序由类组成，类是用于对象创建的模板。运行中的Java程序包含若干对象，可以通过实例化类来创建对象。类可能包含数据成员以及操作数据的方法。可以定义不含有任何实现的类，在这种情况下，类没有任何数据与方法声明。类的每个成员可以在声明时指定访问修饰符，用于定义成员在整个应用程序中的可见性。在本章，你使用过的访问修饰符有public与private。我们推荐类的所有数据成员都使用private修饰符定义。使用private修饰符声明的数据成员不能被类定义之外的外部代码使用点语法访问。为了能够从外部代码访问私有变量，需要定义getter和setter方法，这些方法也分别叫做访问器和存值器方法。我们把这种做法称为信息隐藏。为可变字段定义setter方法还可以帮助在赋值变量前对数据输出进行验证。

可以使用new关键字实例化类。实例化会调用类的构造函数。构造函数是定义在类中的一种特殊方法，用于初始化创建对象的状态。编译器会提供空主体的默认构造函数。也可以编写自己的构造函数。构造函数可以带有参数，从而为新创建的对象提供初始值。类可以包含多个拥有不同参数的构造函数。构造函数的声明必须遵循一系列预定义的规则。编译器会根据参数的数量及类型决定调用哪个构造函数。

Java源程序遵循预定义的布局，包含package、import与class定义段。package语句帮助对逻辑相关的类进行分组以控制它们在整个应用程序中的可见性。包可能会为类的逻辑组织创建层次结构。当源程序声明package语句时，编译器会按照package声明生成文件夹层次。必须在编译器命令提示符中使用import语句，在源程序中导入外部类，import语句必须指定导入类的完全限定名称。

在下一章，你将会学习面向对象编程的另一重要特性——继承。