Broadview®
www.broadview.com.cn

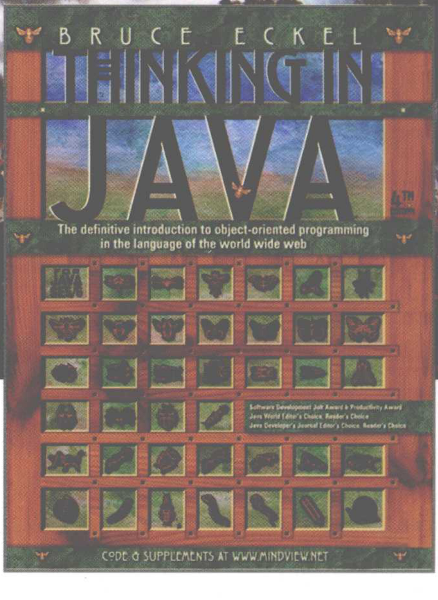
PEARSON

传世经典书丛
Eternal Classics

Java 编程思想 (第4版)
(评注版)

[美] Bruce Eckel 著

刘中兵 评注


BRUCE ECKEL
THINKING IN JAVA
The definitive introduction to object-oriented programming in the language of the world wide web
CODE & SUPPLEMENTS AT WWW.MINDVIEW.NET

Thinking in Java (Fourth Edition)

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Java编程思想

（第4版）（评注版）

Thinking in Java(Fourth Edition)

［美］Bruce Eckel 著

刘中兵 评注

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书作者拥有多年教学经验，对 C、C++以及 Java 语言都有独到、深入的见解，书中以通俗易懂且小而直接的示例阐释了一个个晦涩抽象的概念，是一本当之无愧的经典之作。本评注版讲解了 Java 设计、语法和库的各个方面，包括 Java 的运算符、控制逻辑、构造、回收、重用、接口、内部类、存储、异常、字符串、类型、泛型、数组、容器、I/O、注释、并发等内容。

对于国外技术图书，选择翻译版还是影印版，常常让人陷入两难的境地。本评注版力邀国内资深专家执笔，在英文原著基础上增加中文点评与注释，旨在融合二者之长，既保留经典的原创文字与味道，又以先行者的学研心得与实践感悟，对读者阅读与学习加以点拨、指明捷径。

经过评注的版本，更值得反复阅读与体会。希望这本书能够帮助您跨越 Java 的重重险阻，领略高处才有的壮美风光，做一个成功而快乐的 Java 程序员。

Authorized Adaptation from the English language edition, entitled *Think in Java, Fourth Edition*, 9780131872486 by Bruce Eckel, published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2006 by Bruce Eckel , President, MindView, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

ENGLISH language adaptation edition published by PEARSON EDUCATION ASIA LTD. And PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2011. ENGLISH language adaptation edition is manufactured in the People's Republic of China, and is authorized for sale only in People's Republic of China excluding Hong Kong and Macau.

本书影印改编版由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社出版。专有出版权受法律保护。

本书影印改编版在中国大陆地区生产，仅限于在中国大陆地区销售。

本书影印改编版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号：图字：01-2011-3089

图书在版编目（CIP）数据

Java 编程思想 = Thinking in Java, Fourth Edition：第 4 版：评注版 /（美）埃克尔（Eckel, B.）著；刘中兵评注. —4 版. —北京：电子工业出版社，2011.6

（传世经典书丛）

ISBN 978-7-121-13521-7

Ⅰ. ①J…　Ⅱ. ①埃…　②刘…　Ⅲ. ①JAVA 语言—程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2011）第 084509 号

策划编辑：张春雨
责任编辑：李云静
印　　刷：北京天宇星印刷厂
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：47.25　字数：1738 千字
印　　次：2011 年 6 月第 1 次印刷
定　　价：108.00 元

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 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

悦读上品 得乎益友

孔子云:“取乎其上,得乎其中;取乎其中,得乎其下;取乎其下,则无所得矣”。

对于读书求知而言,这句古训教我们去读好书,最好是好书中的上品——经典书。其中,科技人员要读的技术书,因为直接关乎客观是非与生产效率,阅读选材本更应慎重。然而,随着技术图书品种的日益丰富,发现经典书越来越难,尤其对于涉世尚浅的新读者,更为不易,而他们又往往是最需要阅读、提升的重要群体。

所谓经典书,或说上品,是指选材精良、内容精练、讲述生动、外延丰盈、表现手法体贴入微的读品,它们会成为读者的知识和经验库中的重要组成部分,并且拥有从不断重读中汲取养分的空间。因此,选择阅读上品的问题便成了有效阅读的首要问题。当然,这不只是效率问题,上品促成的既是对某一种技术、思想的真正理解和掌握,同时又是一种感悟或享受,是一种愉悦。

与技术本身类似,经典 IT 技术书多来自国外。深厚的积累、良好的写作氛围,使一批大师为全球技术学习者留下了璀璨的智慧瑰宝。就在那个年代即将远去之时,无须回眸,也能感受到这一部部厚重而深邃的经典著作,在造福无数读者后从未蒙尘的熠熠光辉。而这些凝结众多当今国内技术中坚美妙记忆与绝佳体验的技术图书,虽然尚在国外图书市场上大放异彩,却已逐渐淡出国人的视线。最为遗憾的是,迟迟未有可以填补空缺的新书问世。而无可替代,不正是经典书被奉为圭臬的原因?

为了不让国内读者,尤其是即将步入技术生涯的新一代读者,就此错失这些滋养过先行者们的好书,以出版 IT 精品图书,满足技术人群需求为己任的我们,愿意承担这一使命。本次机遇惠顾了我们,让我们有机会携手权威的 Pearson 公司,精心推出“传世经典书丛”。

在我们眼中,“传世经典”的价值首先在于——既适合喜爱科技图书的读者,也符合专家们挑剔的标准。幸运的是,我们的确找到了这些堪称上品的佳作。丛书带给我们的幸运颇多,细数一下吧。

得以引荐大师著作

有恐思虑不周,我们大量参考了国外权威机构和网站的评选结果,并得到了 Pearson 的专业支持,又进

一步对符合标准之图书的国内外口碑与销售情况进行细致分析，也听取了国内技术专家的宝贵建议，才有幸选出对国内读者最富有技术养分的大师上品。

向深邃的技术内涵致敬

中外技术环境存在差异，很多享誉国外的好书未必适用于国内读者；且技术与应用瞬息万变，很容易让人心生迷惘或疲于奔命。本丛书的图书遴选，注重打好思考方法与技术理念的根基，旨在帮助读者修炼内功，提升境界，将技术真正融入个人知识体系，从而可以一通百通，从容面对随时涌现的技术变化。

翻译与评注的双项选择

引进优秀外版著作，将其翻译为中文供国内读者阅读，较为有效与常见。但另有一些外语水平较高、喜好阅读原版的读者，苦于对技术理解不足，不能充分体会原文表述的精妙，需要有人指导与点拨。而一批本土技术精英经过长期经典熏陶及实践锤炼，已足以胜任这一工作。有鉴于此，本丛书在翻译版的同时推出融合英文原著与中文点评、注释的评注版，供不同志趣的读者自由选择。

承蒙国内一流译（注）者的扶持

优秀的英文原著最终转化为真正的上品，尚需跨越翻译鸿沟，外版图书的翻译质量一直屡遭国内读者诟病。评注版的增值与含金量，同样依赖于评注者的高卓才具。好在，本丛书得到了久经考验的权威译（注）者的认可和支持，首肯我们选用其佳作，或亲自参与评注工作。正是他们的参与保证了经典的品质，既再次为我们的选材把关，更提供了一流的中文表述。

期望带给读者良好的阅读体验

一本好书带给人的愉悦不止于知识收获，良好的阅读感受同样不可缺少，且对学业不无助益。为让读者收获与上品相称的体验，我们在图书装帧设计与选材用料上同样不敢轻率，惟愿送到读者手中的除了珠玑章句，还有舒适与熨帖的视觉感受。

所有参与丛书出版的人员，尽管能力有限，却无不心怀严谨之心与完美愿望。如果读者朋友能从潜心阅读这些上品中偶有获益，不啻为对我们工作的最佳褒奖。若有阅读感悟，敬请拨冗告知，以鼓励我们继续在这一道路上贡献绵薄之力。如有不周之处，也请不吝指教。

电子工业出版社博文视点

试读结束：需要全本请在线购买： www.ertongbook.com

评注者序

与 *Thinking in Java* 的渊源

学习 Java 的道路好比西天取经，Java 就是要取得的真经，我们都是去往 Java 之路的圣徒。

学习 Java 不仅要经历九九八十一难，更要理解 Java 这本圣经的精髓。大家都觉得 Java 之路不好走，因为它不仅仅是 Java，还有无数的技术，如 JSP、Servlet、JavaBean、J2EE、Struts、Spring、Hibernate、数据库 SQL、SOA 分布式、缓存、JVM、Eclipse、Tomcat、HTTP、JSON 等等，它们组成了取经道路上的九九八十一难。众多技术，围绕的精髓是 Java。

在学习 Java 的道路上，我也经历过九九八十一难。我通过编写《Java 高手真经》对 Java 道路上的层层技术做了一个自认为完整的总结，也是对自己步入 Java 世界以来的种种所见所闻所感的记录和升华。但此时，对 Java 我有种被掏空的感觉。因为，我觉得自己对 Java 的精髓理解得还是不够。

偶然的机会接到博文视点的邀约，为这本《Java 编程思想》（*Thinking in Java*）的英文版本做点评。尽管我写过很多技术类的书，但对于这种圣经式的经典著作，借用时下最流行的一句话，内心有一点“忐忑”☺。因为，对经典著作做的点评，必须是点睛之笔，对原文要能够画龙点睛。想到这里，内心就更加忐忑了！为此，我抱着学习交流的心态重读了这本书。

刚学习 Java 时，往往会经资深人士推荐说：“读这本《Java 编程思想》才是学真正的 Java，国人所著太浅！”于是怀着崇拜之情，读之，傻眼，1000 多页、大量本质的分析，再加上蹩脚的英语，读之皮毛矣！从此也就束之高阁。

何时应该读这本书

李彦宏关于互联网的发展曾说过：“时机很重要，进去太早会饿死，太晚就没有机会了。”实际上这句话在这里也适用，太早读这本书会难以理解，太晚你会后悔为什么才看到这本书。这是因为，正如书名所言，它讲解的是 Java 编程的内在思想，只有拥有了一定 Java 编程基础和开发经验后，读它才是对 Java 理解的升华！

因此，这本书最大的作用不是让你“知其然”，而是让你“知其所以然”！

如何读这本书

本书原英文版涵盖了 Java 设计、语法和库的各个方面，包括 Java 的运算符、控制逻辑、构造、回收、重用、多态、接口、内部类、存储、异常、字符串、类型、泛型、数组、容器、I/O、注释、并发、图形 GUI。原英文版是按美国人的习惯散列排列的，本书为了更加一目了然，更加符合国人的思

维方式，将这些章节分为五个大的部分，每一部分都涵盖了深入分析的独家内容。

第 1 部分　基本语法：运算符、控制逻辑

运算符和控制逻辑是任何一门语言的最基础语法，Java 参考了 C++的优点，扬长避短，因此它的运算符也基于 C++，但在某些方面进行了简化和改进。熟悉 C++将会对本部分轻车熟路。Java 中的控制语句共提供了如下几个关键字：if/else/while/do-while/for/return/break/continue/switch。虽然有 goto，但是不要使用，它会使代码混乱。在 Java 中已经摒弃了 goto 语句。

第 2 部分　面向对象：对象的创建与销毁、访问控制、重用、接口、内部类、异常

本书是关于类的初始化和销毁讲解最为完美的一本书，细致、全面、易懂。访问控制专门开辟了一章，对于类和包的访问控制关系讲解小巧到位。要让每个人编写的程序能够为他人所用，首先是类的包结构清晰，其次是代码中的函数包装清晰。

第 3 部分　数据存储：字符串、数组、存储、容器

java.util 包中提供了纷繁复杂的容器类，包括集合类 Collection（包括 Queue 先进先出队列、List 允许重复的有序列表、Set 不允许重复的集合）和映射类 Map（键值对）。本书深度分析了基于这些接口的各种实现类的使用、工作原理。“Containers in Depth”一章是讲解 Java 容器类最深入的部分，它会告诉你何时应该使用哪些容器、哪些是最有用的、为什么使用以及怎么使用。

第 4 部分　核心功能：输入输出、并发

I/O 提供了 Java 与外部系统进行通信的基础库，包括控制台、文件、网络。这些是 I/O 需要涉及的方方面面。输入输出的方式包括顺序读取、随机读取、缓存、二进制、字符、行读取、字节读取等等，因此 I/O 也提供了各种读写的包装类。本书通过大量实例来演示这些类的使用方法和应用场景。

多线程是与单线程比较而言的，本书从基本的 Thread、Runnable 的使用讲起，深入且全面讲解了线程的休眠、优先级、Daemon、线程的异常处理、资源共享、线程的控制、与其他线程的协作、死锁等，并讲解了 Java 中使用的一些线程工具类。

第 5 部分　高级特性：类型、泛型、枚举、注释

书中从 Class 类、cast、静态分析、动态分析、动态代理、Mock 等各方面讲解类的 RTTI 解析，使用 Generic 实现对 Java 的扩展。JDK 5.0 通过名为注释（Annotation）的新功能，将一个更通用的元数据工具合并到核心 Java 语言中。对于开发者来说，不仅可以使用 JDK 内置的注释，还可以使用第三方提供的强大注释功能，比如单元测试、EJB、Hibernate 等，还可以自定义注释。

我的建议

由于 AWT 和 Swing 目前在实际应用中比例不高，并且擅长界面编程的中文图书比比皆是，因此就删除了图形化编程 GUI 一章。另外，本书开头的两章，即“Introduction to Objects”和“Everything Is an Object”没有收入本书，

但读者可从网上下载加了评注的这些内容，网址是 http://www.broadview.com.cn/13521。

还请读者留意以下两点：

（1）由于本评注版经重新编排，章节顺序与原著不尽相同，加之部分章节有所节略，因此书中涉及引用章节之处与图书现状略有出入。由此给读者带来的不便，还请谅解。

（2）各章练习序号后的小括号中的文字，代表本练习的难度等级（用 1~10 表示）。

最后，在您阅读本书时，我有两点建议：

- 对经典的分析、代码及时做批注，多做对比和总结。
- 上机编写运行书中的经典实例，尝试不同的运行结果，分析代码的用意。
- 对书中的实例举一反三、反复推敲，分析更多的实际应用场景。

只有用心去读、去体会、去实践，才能够真正理解 Java 编程中的思想。

本评注版由刘中兵、陈景春、周同、刘中敏、陈万珍、刘剑、李瑞霞、李建海、李金刚、刘中丽执笔，撰写相关中文评注。行文若有不妥之处，敬请广大读者提出宝贵意见和建议。

刘中兵

2011 年 4 月 20 日于北京

目　录

第1部分　基本语法

Operators（新增批注30条）……1

Simpler print statements……1

Using Java operators……2

Precedence……2

Assignment……3

Mathematical operators……4

Unary minus and plus operators……6

Auto increment and decrement……6

Relational operators……7

Testing object equivalence……7

Logical operators……9

Literals……10

Exponential notation……11

Bitwise operators……12

Shift operators……13

Ternary **if-else** operator……16

String operator + and +=……17

Common pitfalls when using operators……18

Casting operators……18

Truncation and rounding……19

Promotion……20

Java has no "sizeof"……20

Summary……20

Controlling Execution（新增批注21条）……21

true and **false**……21

if-else……21

Iteration……22

do-while……23

for……23

The comma operator……24

Foreach syntax……25

return……27

break and **continue**……27

The infamous "goto"……29

switch……32

Summary……34

第2部分　面向对象

Initialization & Cleanup（新增批注55条）……35

Guaranteed initialization with the constructor……35

Method overloading……37

Distinguishing overloaded methods……39

Overloading with primitives……39

Overloading on return values……42

Default constructors……43

The **this** keyword……44

Calling constructors from constructors……46

The meaning of **static**……47

Cleanup: finalization and garbage collection……47

What is **finalize()** for?……48

You must perform cleanup……49

The termination condition……50

How a garbage collector works……51

Member initialization……54

Specifying initialization……55

Constructor initialization……56

Order of initialization……56

static data initialization……57

Explicit **static** initialization……59

Non-**static** instance initialization……61

Array initialization……62

Variable argument lists……65

Enumerated types……70

Summary……72

Access Control（新增批注21条）……73

package: the library unit……74

Code organization……75

Creating unique package names……76

A custom tool library……79

Java access specifiers……80

Package access……80

public: interface access ··· 81
private: you can't touch that! ··· 82
protected: inheritance access ··· 83
Interface and implementation ··· 85
Class access ··· 86
Summary ··· 87

Reusing Classes（新增批注35条） ··· 89
Composition syntax ··· 89
Inheritance syntax ··· 92
Initializing the base class ··· 94
Delegation ··· 96
Combining composition and inheritance ··· 97
Guaranteeing proper cleanup ··· 99
Name hiding ··· 101
Choosing composition vs. inheritance ··· 103
protected ··· 104
Upcasting ··· 105
Why "upcasting"? ··· 106
Composition vs. inheritance revisited ··· 106
The **final** keyword ··· 107
final data ··· 107
final methods ··· 110
final classes ··· 112
final caution ··· 113
Initialization and class loading ··· 113
Initialization with inheritance ··· 114
Summary ··· 115

Interfaces（新增批注16条） ··· 117
Abstract classes and methods ··· 117
Interfaces ··· 120
"Multiple inheritance" in Java ··· 123
Extending an interface with inheritance ··· 125
Name collisions when combining interfaces ··· 127
Fields in interfaces ··· 127
Initializing fields in interfaces ··· 128
Interfaces and factories ··· 129
Summary ··· 130

Inner Classes（新增批注32条） ··· 131
Creating inner classes ··· 131
The link to the outer class ··· 133
Using **.this** and **.new** ··· 134
Inner classes and upcasting ··· 135
Anonymous inner classes ··· 137
Factory Method revisited ··· 140
Nested classes ··· 142
Classes inside interfaces ··· 143
Reaching outward from a multiply nested class ··· 145
Why inner classes? ··· 145
Closures & callbacks ··· 148
Inheriting from inner classes ··· 150
Can inner classes be overridden? ··· 150
Local inner classes ··· 152
Inner-class identifiers ··· 153
Summary ··· 154

Error Handling with Exceptions（新增批注52条） ··· 155
Basic exceptions ··· 155
Exception arguments ··· 156
Catching an exception ··· 157
The **try** block ··· 157
Exception handlers . ··· 157
Creating your own exceptions ··· 159
Exceptions and logging ··· 161
The exception specification ··· 164
Catching any exception ··· 164
The stack trace ··· 166
Rethrowing an exception ··· 167
Exception chaining ··· 169
Standard Java exceptions ··· 172
Special case: **RuntimeException** ··· 172
Performing cleanup with **finally** ··· 174
What's finally for? ··· 175
Using **finally** during **return** ··· 177
Pitfall: the lost exception ··· 178
Exception restrictions ··· 180
Constructors ··· 182
Exception matching ··· 187
Alternative approaches ··· 188
Passing exceptions to the console ··· 189
Summary ··· 189

第 3 部分　数据存储

Strings（新增批注53条） ··· 191
Immutable **Strings** ··· 191

Overloading '+' vs. **StringBuilder** 192
Unintended recursion 195
Operations on **Strings** 196
Formatting output 199
  printf() 199
  System.out.format() 199
  The **Formatter** class 200
  Format specifiers 200
  Formatter conversions 202
  String.format() 204
Regular expressions 205
  Basics 206
  Creating regular expressions 208
  Quantifiers 210
  Pattern and **Matcher** 211
  split() 218
  Replace operations 218
  reset() 220
  Regular expressions and Java I/O 221
Scanning input 222
  Scanner delimiters 224
  Scanning with regular expressions 225
StringTokenizer 225
Summary 226

Arrays（新增批注36条） 227

Why arrays are special 227
Arrays are first-class objects 228
Returning an array 231
Multidimensional arrays 232
Arrays and generics 235
Creating test data 237
  Arrays.fill() 238
  Data **Generators** 239
Arrays utilities 243
  Copying an array 243
  Comparing arrays 244
  Array element comparisons 245
  Sorting an array 248
  Searching a sorted array 249
Summary 251

Holding Your Objects（新增批注35条） 253

Generics and type-safe containers 254
Basic concepts 256
Adding groups of elements 258
Printing containers 259
List 261
Iterator 263
  ListIterator 266
LinkedList 267
Stack 268
Set 270
Map 273
Queue 276
  PriorityQueue 277
Collection vs. **Iterator** 279
Foreach and iterators 281
  The *Adapter Method* idiom 283
Summary 286

Containers in Depth（新增批注102条） 289

Full container taxonomy 289
Filling containers 290
  A **Generator** solution 291
  Map generators 292
Collection functionality 294
Optional operations 297
  Unsupported operations 298
List functionality 300
Sets and storage order 302
  SortedSet 306
Queues 307
  Priority queues 308
  Deques 309
Understanding **Maps** 310
  Performance 311
  SortedMap 314
  LinkedHashMap 315
Hashing and hash codes 316
  Understanding **hashCode()** 319
  Hashing for speed 321
  Overriding **hashCode()** 324
Choosing an implementation 329
  A performance test framework 330
  Choosing between **Lists** 333
  Microbenchmarking dangers 338
  Choosing between **Sets** 339
  Choosing between **Maps** 341
Utilities 344
  Sorting and searching **Lists** 347
  Making a **Collection** or **Map** unmodifiable 349
  Synchronizing a **Collection** or **Map** 350
Holding references 351
  The **WeakHashMap** 353
Java 1.0/1.1 containers 355
  Vector & **Enumeration** 355
  Hashtable 356
  Stack 356
  BitSet 357
Summary 359

试读结束：需要全本请在线购买： www.ertongbook.com

第4部分　核心功能

I/O（新增批注89条）…… 361
- The **File** class …… 361
  - A directory lister …… 361
  - Directory utilities …… 364
  - Checking for and creating directories …… 369
- Input and output …… 370
  - Types of **InputStream** …… 371
  - Types of **OutputStream** …… 372
- Adding attributes and useful interfaces …… 373
  - Reading from an **InputStream** with **FilterInputStream** …… 374
  - Writing to an **OutputStream** with **FilterOutputStream** …… 375
- **Readers & Writers** …… 376
  - Sources and sinks of data …… 377
  - Modifying stream behavior …… 377
  - Unchanged classes …… 378
- Off by itself: **RandomAccessFile** …… 379
- Typical uses of I/O streams …… 379
  - Buffered input file …… 379
  - Input from memory …… 380
  - Formatted memory input …… 381
  - Basic file output …… 382
  - Storing and recovering data …… 383
  - Reading and writing random-access files …… 385
  - Piped streams …… 386
- File reading & writing utilities …… 386
  - Reading binary files …… 389
- Standard I/O …… 389
  - Reading from standard input …… 389
  - Changing **System.out** to a **PrintWriter** …… 390
  - Redirecting standard I/O …… 391
- Process control …… 391
- New I/O …… 393
  - Converting data …… 396
  - Fetching primitives …… 398
  - View buffers …… 399
  - Data manipulation with buffers …… 403
  - Buffer details …… 404
  - Memory-mapped files …… 406
  - File locking …… 409
- Compression …… 411
  - Simple compression with GZIP …… 412
  - Multifile storage with Zip …… 413
  - Java ARchives (JARs) …… 415
- Object serialization …… 416
  - Finding the class …… 419
  - Controlling serialization …… 420
  - Using persistence …… 427
- XML …… 432
- Summary …… 434

Concurrency（新增批注117条）…… 435
- Basic threading …… 435
  - Defining tasks …… 435
  - The **Thread** class …… 436
  - Using **Executors** …… 438
  - Producing return values from tasks …… 440
  - Sleeping …… 442
  - Priority …… 443
  - Yielding …… 444
  - Daemon threads …… 445
  - Coding variations …… 446
  - Joining a thread …… 450
  - Catching exceptions …… 451
- Sharing resources …… 454
  - Resolving shared resource contention …… 454
  - Atomicity and volatility …… 457
  - Atomic classes …… 460
  - Critical sections …… 462
  - Synchronizing on other objects …… 462
  - Thread local storage …… 463
- Terminating tasks …… 464
  - The ornamental garden …… 465
  - Terminating when blocked …… 467
  - Interruption …… 469
- Cooperation between tasks …… 475
  - **wait()** and **notifyAll()** …… 475
  - **notify()** vs. **notifyAll()** …… 479
  - Producers and consumers …… 482
  - Producer-consumers and queues …… 486
  - Using pipes for I/O between tasks …… 491
- Deadlock …… 492
- New library components …… 497
  - **CountDownLatch** …… 497
  - **CyclicBarrier** …… 499
  - **DelayQueue** …… 501
  - **PriorityBlockingQueue** …… 503
  - The greenhouse controller with **ScheduledExecutor** …… 505
  - **Semaphore** …… 508
  - **Exchanger** …… 511
- Simulation …… 513
  - Bank teller simulation …… 513
  - The restaurant simulation …… 517
  - Distributing work …… 521
- Performance tuning …… 526
  - Comparing mutex technologies …… 526
  - Lock-free containers …… 532
  - **ReadWriteLocks** …… 533
- Active objects …… 535
- Summary …… 537

第 5 部分　高级特性

Type Information（新增批注59条）............ 539

The need for RTTI 539

The **Class** object 541

Class literals 545

Generic class references 547

New cast syntax 549

Checking before a cast 550

Using class literals 555

A dynamic **instanceof** 557

Counting recursively 558

Registered factories 559

instanceof vs. **Class** equivalence 562

Reflection: runtime class information 563

A class method extractor 564

Dynamic proxies 566

Null Objects 570

Mock Objects & Stubs 575

Interfaces and type information 576

Summary 580

Generics（新增批注126条）............ 583

Comparison with C++ 584

Simple generics 584

A tuple library 586

A stack class 588

RandomList 589

Generic interfaces 590

Generic methods 593

Leveraging type argument inference 594

Varargs and generic methods 596

A generic method to use with **Generators** 596

A general-purpose **Generator** 597

Simplifying tuple use 598

A **Set** utility 600

Anonymous inner classes 603

Building complex models 604

The mystery of erasure 606

The C++ approach 607

Migration compatibility 609

The problem with erasure 611

The action at the boundaries 612

Compensating for erasure 615

Creating instances of types 616

Arrays of generics 618

Bounds 622

Wildcards 625

How smart is the compiler? 628

Contravariance 629

Unbounded wildcards 632

Capture conversion 636

Issues 637

No primitives as type parameters 637

Implementing parameterized interfaces 639

Casting and warnings 640

Overloading 641

Base class hijacks an interface 642

Self-bounded types 642

Curiously-recurring generics 643

Self-bounding 644

Argument covariance 646

Dynamic type safety 649

Exceptions 650

Mixins 651

Mixins in C++ 651

Mixing with interfaces 653

Using the Decorator pattern 654

Mixins with dynamic proxies 655

Latent typing 657

Compensating for the lack of latent typing 660

Reflection 661

Applying a method to a sequence 662

When you don't happen to have the right interface 664

Simulating latent typing with adapters 665

Using function objects as strategies 668

Summary: Is casting really so bad? 672

Enumerated Types（新增批注55条）............ 675

Basic **enum** features 675

Using **static** imports with **enums** 676

Adding methods to an **enum** 677

Overriding **enum** methods 678

enums in **switch** statements 678

The mystery of **values()** 679

Implements, not inherits 681

Random selection 682

Using interfaces for organization 683

Using **EnumSet** instead of flags 686

Using **EnumMap** 688

Constant-specific methods ········ 689
Chain of Responsibility with **enums** ········ 692
State machines with **enums** ········ 695
Multiple dispatching ········ 700
Dispatching with **enums** ········ 702
Using constant-specific methods ·· 704
Dispatching with **EnumMaps** ······ 705
Using a 2-D array ········ 706
Summary ········ 707

Annotations
（新增批注51条） ········ 709

Basic syntax ········ 710
Defining annotations ········ 710
Meta-annotations ········ 712
Writing annotation processors ····712
Annotation elements ········ 713
Default value constraints ········ 713
Generating external files ········ 714
Annotations don't support inheritance ········ 717
Implementing the processor ········ 717
Using **apt** to process annotations ········ 719
Using the *Visitor* pattern with **apt** ········ 723
Annotation-based unit testing ··· 726
Using **@Unit** with generics ········ 733
No "suites" necessary ········ 735
Implementing **@Unit** ········ 735
Removing test code ········ 741
Summary ········ 742

Operators

▶Java参考了C++的优点，扬长避短，因此它的运算符也基于C++，但在某些方面进行了简化和改进。熟悉C++将会对本章轻车熟路。

At the lowest level, data in Java is manipulated using operators.

Because Java was inherited from C++, most of these operators will be familiar to C and C++ programmers. Java has also added some improvements and simplifications.

If you're familiar with C or C++ syntax, you can skim through this chapter and the next, looking for places where Java is different from those languages. However, if you find yourself floundering a bit in these two chapters, make sure you go through the multimedia seminar *Thinking in C*, freely downloadable from *www.MindView.net*. It contains audio lectures, slides, exercises, and solutions specifically designed to bring you up to speed with the fundamentals necessary to learn Java.

第1部分 基本语法

Simpler print statements

In the previous chapter, you were introduced to the Java print statement:

```
System.out.println("Rather a lot to type");
```

You may observe that this is not only a lot to type (and thus many redundant tendon hits), but also rather noisy to read. Most languages before and after Java have taken a much simpler approach to such a commonly used statement.

The *Access Control* chapter introduces the concept of the *static import* that was added to Java SE5, and creates a tiny library to simplify writing print statements. However, you don't need to know those details in order to begin using that library. We can rewrite the program from the last chapter using this new library:

```
//: operators/HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
  public static void main(String[] args) {
    print("Hello, it's: ");
    print(new Date());
  }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:05 MDT 2005
*///:~
```

▶import static关键字是直接导入某个类的所有变量和方法到本类，这样就可以直接在当前类中引用导入类的变量和方法，例如被导入的print()方法。

The results are much cleaner. Notice the insertion of the **static** keyword in the second **import** statement.

In order to use this library, you must download this book's code package from *www.MindView.net* or one of its mirrors. Unzip the code tree and add the root directory of that code tree to your computer's CLASSPATH environment variable. (You'll eventually get a full introduction to the classpath, but you might as well get used to struggling with it early. Alas, it is one of the more common battles you will have with Java.)

Although the use of **net.mindview.util.Print** nicely simplifies most code, it is not justifiable everywhere. If there are only a small number of print statements in a program, I forego the **import** and write out the full **System.out.println( )**.

Exercise 1: (1) Write a program that uses the "short" and normal form of print statement.

▶最简单的就是算术运算符：加、减、乘、除、等于、相等、不等于。

Using Java operators

An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary method calls, but the effect is the same. Addition and unary plus (+), subtraction and unary minus (-), multiplication (*), division (/), and assignment (=) all work much the same in any programming language.

All operators produce a value from their operands. In addition, some operators change the value of an operand. This is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but you should keep in mind that the value produced is available for your use, just as in operators without side effects.

Almost all operators work only with primitives. The exceptions are '=', '==' and '!=', which work with all objects (and are a point of confusion for objects). In addition, the **String** class supports '+' and '+='.

▶加、减、乘、除的优先级与算术中相同。

Precedence

Operator precedence defines how an expression evaluates when several operators are present. Java has specific rules that determine the order of evaluation. The easiest one to remember is that multiplication and division happen before addition and subtraction. Programmers often forget the other precedence rules, so you should use parentheses to make the order of evaluation explicit. For example, look at statements **(1)** and **(2)**:

```
//: operators/Precedence.java

public class Precedence {
  public static void main(String[] args) {
    int x = 1, y = 2, z = 3;
    int a = x + y - 2/2 + z;           // (1)
    int b = x + (y - 2)/(2 + z);       // (2)
    System.out.println("a = " + a + " b = " + b);
  }
} /* Output:
a = 5 b = 1
*///:~
```

These statements look roughly the same, but from the output you can see that they have very different meanings which depend on the use of parentheses.

Notice that the **System.out.println( )** statement involves the '+' operator. In this context, '+' means "string concatenation" and, if necessary, "string conversion." When the compiler sees a **String** followed by a '+' followed by a non-**String**, it attempts to convert the non-**String** into a **String**. As you can see from the output, it successfully converts from **int** into **String** for **a** and **b**.

Assignment

▶赋值操作符=。

Assignment is performed with the operator =. It means "Take the value of the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*)." An rvalue is any constant, variable, or expression that produces a value, but an lvalue must be a distinct, named variable. (That is, there must be a physical space to store the value.) For instance, you can assign a constant value to a variable:

```
a = 4;
```

but you cannot assign anything to a constant value—it cannot be an lvalue. (You can't say **4 = a;**.)

Assignment of primitives is quite straightforward. Since the primitive holds the actual value and not a reference to an object, when you assign primitives, you copy the contents from one place to another. For example, if you say **a = b** for primitives, then the contents of **b** are copied into **a**. If you then go on to modify **a**, **b** is naturally unaffected by this modification. As a programmer, this is what you can expect for most situations.

When you assign objects, however, things change. Whenever you manipulate an object, what you're manipulating is the reference, so when you assign "from one object to another," you're actually copying a reference from one place to another. This means that if you say **c = d** for objects, you end up with both **c** and **d** pointing to the object that, originally, only **d** pointed to. Here's an example that demonstrates this behavior:

▶Java数据类型分为原始数据类型和对象数据类型，原始类型如int、boolean、double，对象类型即为Java类的实例。如果将一个原始类型赋值给另一个变量，则会复制一份，各自修改值互不干涉；而如果将对象类型赋值给另一个变量，则只是赋给它对象的引用（也就是指针），修改该对象的值两者都会修改。

```
//: operators/Assignment.java
// Assignment with objects is a bit tricky.
import static net.mindview.util.Print.*;

class Tank {
  int level;
}

public class Assignment {
  public static void main(String[] args) {
    Tank t1 = new Tank();
    Tank t2 = new Tank();
    t1.level = 9;
    t2.level = 47;
    print("1: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1 = t2;
    print("2: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1.level = 27;
    print("3: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
  }
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*///:~
```

The **Tank** class is simple, and two instances (**t1** and **t2**) are created within **main( )**. The **level** field within each **Tank** is given a different value, and then **t2** is assigned to **t1**, and **t1** is changed. In many programming languages

试读结束：需要全本请在线购买： www.ertongbook.com