

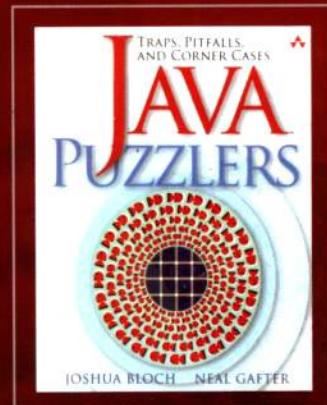
Java Puzzlers

Traps, Pitfalls, and Corner Cases

Java 解惑

[美] Joshua Bloch 著
Neal Gafter
陈昊鹏 译

- *Effective Java* 作者又一力作
- 实例讲授 Java 中令人迷惑和不易掌握的知识点
- 寓教于乐，妙趣横生



人民邮电出版社
POSTS & TELECOM PRESS

TURING

图灵程序设计丛书

Java解惑

Java Puzzlers

Traps, Pitfalls, and Corner Cases

[美] Joshua Bloch 著
Neal Gafter
陈昊鹏 译



人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目 (CIP) 数据

Java 解惑 / (美) 布洛赫, (美) 加夫特著; 陈昊鹏译. —北京: 人民邮电出版社, 2006.1
(图灵程序设计丛书)

ISBN 7-115-14241-6

I. J... II. ①布...②加...③陈... III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2005) 第 152708 号

内 容 提 要

本书特写了95个有关Java或其类库的陷阱和缺陷的谜题, 其中大多数谜题都采用短程序的形式给出, 这些程序的实际行为与表面上大相径庭。在每个谜题之后都给出了详细的解惑方案, 这些解惑方案超越了对程序行为的简单解释, 向读者展示了如何一劳永逸地避免底层的陷阱与缺陷。

本书趣味十足、寓教于乐, 适合于具备Java知识的学习者和有编程经验的Java程序员。

图灵程序设计丛书

Java 解惑

-
- ◆ 著 [美] Joshua Bloch Neal Gafter
 - 译 陈昊鹏
 - 责任编辑 舒 立
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子函件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京顺义振华印刷厂印刷
 - 新华书店总店北京发行所经销
 - ◆ 开本: 800×1000 1/16
 - 印张: 18.75
 - 字数: 418 千字 2006 年 1 月第 1 版
 - 印数: 1~5 000 册 2006 年 1 月北京第 1 次印刷

著作权合同登记号 图字: 01-2005-5230 号

ISBN 7-115-14241-6/TP · 5118

定价: 39.00 元

读者服务热线: (010) 88593802 印装质量热线: (010) 67129223

版 权 声 明

Authorized translation from the English language edition, entitled: *JAVA PUZZLERS:TRAPS, PITFALLS, AND CORNER CASES*, 1st Edition, 032133678X by BLOCH, JOSHUA; GAFTER, NEAL; published by Pearson Education, Inc., publishing as Addison-Wesley Professional.

Copyright © 2005 Pearson Education, 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.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2006.

本书中文简体字版由Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封底贴有Pearson Education (培生教育出版集团) 激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

译者序

Java已经成为越来越多的程序员首选的编程语言，很多程序员都是从C++转到了Java。乍一看，Java与C++的理念和语法都很相似，于是很多程序员都认为Java很容易掌握，但是事实并非如此，像本书中所列举的谜题就不是那么容易解决的。因此，如何真正掌握好Java，尤其是掌握好一些似是而非的知识点，就成为了一个重要的课题。

《Java解惑》这本书以轻松诙谐的语言、简单明了的方式和趣味十足的实例向我们介绍了Java编程语言中许多不易被掌握的知识点，其覆盖面几乎涉及Java编程语言的各个角落。本书不仅指出了造成这些谜题的原因，而且深入探讨了解决这些谜题的方案，有时解决方案甚至不止一种，进而进行总结，归纳出一般的规则和警告。这些规则和警告不仅包括给程序员的部分，还包括给Java语言设计者和API编写者的部分。因而，本书是一本循序渐进、由浅入深和总结归纳的书籍，其阅读价值非常高。

在翻译本书的过程中，我们尽量保持了原书的写作风格，在原文过于简练的极个别地方，适当加入了解释性的语句。由于水平有限，书中难免有不足之处，欢迎广大读者指正。

本书由陈昊鹏翻译，章程、李楠在翻译和校稿的过程中给予了很大的帮助。

陈昊鹏
2005年11月

前　　言

与许多书一样，本书经历了长期的酝酿过程。我们收集Java谜题的时间与我们使用Java这种平台的时间一样长：如果你感兴趣的话，可以告诉你是从1996年中开始至今。在2001年初，我们产生了一个想法：搞一次完全由Java谜题所构成的演讲。我们把这个想法抛给了当时还在Oracle公司的Larry Jacobs，他对这个想法完全买账。

2001年11月在旧金山，我们在Oracle Open World会议上首次作了题为“Java谜题”的演讲。为了增添魅力，我们介绍自己是“Type-it兄弟，Click和Hack”，并且从Tom和Ray Magliozzi主持的Car Talk节目中借用了一大堆的笑话¹。这个演讲被投票选为最佳演讲秀，即使我们不投自己的票结果可能也会如此。由此我们知道找对了路子。

从头到脚穿着蓝领工人那利索的制服，胸前装饰着Java的“咖啡杯”标志，我们在JavaOne 2002上再次利用在Oracle会议上的演讲来鼓吹我们的观点——至少我们的朋友是这么认为的。在接下来的年头里，我们又拿出了另外3个“Java谜题”演讲，并且在数不胜数的会议、公司和大学里宣讲它们，足迹遍及全球许多城市，从奥斯陆到东京。这些演讲几乎得到了普遍的欢迎，几乎没人冲我们扔烂苹果。在*Linux Magazine* 2003年3月刊上，我们发表了一篇完全由Java谜题构成的文章，并且几乎没有收到任何厌恶我们的邮件。本书几乎包含了我们的演讲和文章中的所有谜题，以及许许多多其他的谜题。

尽管本书把注意力放到了Java平台的陷阱和缺陷上，但是我们并不是要以任何方式来诋毁Java。因为热爱Java，我们将近10年的职业生涯都奉献给了它。每一种具有强大能力的平台都会有某些问题，Java与大多数平台相比问题已经少多了。你对问题理解得越透彻，你就越不会受到它们的影响，这正是本书要达到的目的。

本书中的多数谜题都是一些很短的程序，这些程序看起来在“明修栈道”，但实际却“暗渡陈仓”。这就是为什么我们选择视觉幻图来装饰本书的原因，这些幻图看起来是某样事物，但实际上却是另外一件东西。你在努力思考这些程序到底在做什么的时候，去盯着这些幻图好好看看。

毕竟，我们希望本书具有趣味性，真诚地希望你能够尽情享受解惑的乐趣，就像我们尽情享受编写它们的乐趣一样，还希望你能够从中学到很多东西，如我们曾经的那样。

不管怎样，请把你发现的谜题发给我们！如果你有一个你认为应该囊括到本书将来的版本中的谜题，请把它写到一张20美元的账单后面，然后寄给我们，或者发E-mail到puzzlers@javapuzzlers.com。如果采用了你发现的谜题，我们将向你付账。

最后要说的，但不是惟一要说的，就是请不要像我的兄弟那样编写代码。

1. Car Talk是美国NPR电台关于汽车的著名节目，在美国拥有400多万听众。主持人Magliozzi兄弟被称为“Click and Hack, the Tappet Brothers”，名称源自破旧汽车的声音。——编者注

致 谢

我们要感谢Addison-Wesley的整个团队，感谢他们的善举和专业精神。在这个项目的早期，Ann Sellers是我们的编辑。她极富感染力的热情帮助这个项目开了个好头。当Ann离开之后，执行编辑Greg Doench接手了这个项目。Greg是一位令人愉快的编辑和完美的绅士，他眼睛都没眨一下就答应了这个项目的许多要求。Greg的编辑助理是Noreen Regina。我们这本书的项目编辑是Tyrrell Albaugh，营销经理是Stephane Nakib。封面设计是Chuti Prasertsith，加工编辑是Evelyn Pyle。他们在非常紧张的进度安排下都出色地完成了任务。

我们要感谢Google公司管理层的支持。我们的主管Prabha Krishna不断地给我们以信心。我们要感谢Sergey Brin、Larry Page和Eric Schmidt，感谢他们为我们创建了在这个星球上最好的工程环境。

我们要感谢长年累月地给Sun公司发送bug报告的许多Java程序员，特别是那些提交的bug报告被证明为并非真正bug的程序员。这些bug报告也许是谜题资料最丰富的来源：如果正确的行为会误导程序员认为他们发现了一个bug，那么这很可能就是一个陷阱或缺陷。我们以同样的心情感谢Sun公司，感谢她在1996年以非凡的勇气和智慧将完整的Java bug数据库放到网上[Bug]。这个举动是前所未闻的，即便是在今天也很少碰到。

“请把你的谜题发给我们。”我们在每一次演讲结束时都会这么说，而大家的确给我们发来了，从世界各地。特别要感谢Ron Gabor和Mike “madbot” McCloskey，感谢他们所做出的极大贡献。Ron贡献了谜题28、29、30和31，Mike贡献了谜题18、23、40、56和67。我们要感谢Martin Buchholz贡献了谜题81，Armand Dijamco贡献了谜题14，Dominik Gruntz教授（博士）贡献了谜题68和69，Kai Huang贡献了谜题77，Jim Hugunin贡献了谜题45，Tim Huske贡献了谜题41，Peter Kessler贡献了谜题35，Michael Klobe贡献了谜题59，Magnus Lundgren贡献了谜题84，Scott Seligman贡献了谜题22，Peter Stout贡献了谜题39，Michael Tennes贡献了谜题70，Martin Traverso贡献了谜题54。

我们要感谢极富奉献精神的审稿团队，他们在本书还处于粗加工阶段就阅读了每一章：Peter von der Ahé、Pablo Belver、Tracy Bialik、Cindy Bloch、Dan Bloch、Beth Bottos、Joe Bowbeer、Joe Darcy、Bob Evans、Brian Goetz、Tim Halloran、Barry Hayes、Tim Huske、Akiyoshi Kitaoka、Chris Lopez、Mike “madbot” McCloskey、Michael Nielsen、Tim Peierls、Peter Rathmann、Russ Rufer、Steve Schirripa、Yoshiki Shibata、Marshall Spight、Guy Steele、Dean Sutherland、Mark Taylor、Darlene Wallach和Frank Yellin。他们或是发现了缺点，或是给出了改进意见，或是给予了鼓励，或是猛烈地抨击。任何遗留下来的缺点都是我的合著者的失误：）

我们要感谢博客女皇Mary Smaragdis，感谢她在其著名的博客网站上为我们提供了一个舞台〔Mary Blog〕。她亲切地让我们在其读者中测试构成谜题43、53、73、87和94的资料，

我们对解惑方案进行判定，Mary则负责发放奖品。根据记录，获奖者有Tom Hawtin、Tom Hawtin（再次获奖）、Bob “Crazybob” Lee、Chris Nokleberg和来自乌克兰敖德萨的神秘人物AT。在Mary博客网站上的这些讨论对上述谜题贡献良多。

我们要感谢对Java谜题长年累月地投以极大热情的许多支持者。SDForum Java SIG的成员是我们每一个演讲在雏形阶段的试听者。JavaOne的程序委员会为这些演讲提供了一个讲台。Yuka Kamiya和Masayoshi Okutsu使“Java 谜题”演讲在日本取得了成功，在那里他们以在讲台上互相争论的形式将演讲变成了真正的游戏。引人注目的是，获得每一次辩论胜利的都是同一个人：日本无可争辩的Java谜题冠军Yasuhiro Endoh。

我们要感谢James Gosling和其他很多优秀的工程师，他们创造了Java平台，并且在年复一年地不断改进它。像本书这样的书只有对坚如磐石的平台才有意义。没有Java，就没有“Java谜题”。

在Google公司、Sun公司和其他地方的大量同行都参与了改进本书质量的技术讨论。在其他人中，Peter von der Ahé、Dan Bloch和Gilad Bracha贡献了非常有用的见识。我们要特别感谢Doug Lea，他是本书中许多想法的大力宣传者，再次感谢Doug不倦地慷慨大方地贡献自己的时间和知识。

我们要感谢日本京都的Ritsumeikan 大学心理学系的Akiyoshi Kitaoka教授，感谢他允许我们使用他的一些视觉幻图来装饰本书。Kitaoka教授的幻图非常简单，但是却令人震撼。语言已经不能对它们进行描述了，所以你应该自己去看一看。他编写的两本书[Kitaoka02, Kitaoka03]可以在日本买到，包含这两本书内容的英译本不久即将面市[Kitaoka05]。同时，你可以访问他的网站：<http://www.ritsumei.ac.jp/~akitaoka/index-e.html>。你不会失望的。

我们要感谢主持Car Talk节目的Tom和Ray Magliozzi，感谢他们为我们提供了借用的笑话，我们还要感谢他们的法律顾问Dewey、Cheetham和Howe，感谢他们没有起诉我们。

我们要感谢Josh的妻子Cindy，她帮助我们使用FrameMaker排版，并帮我们编写了索引，还帮我们编辑了本书，并且还设计了每一章开始处的装饰条。最后我们要说的，但决不是惟一要说的，就是我们的家庭：Cindy、Tim和Matt Bloch，以及Ricki Lee、Sarah和Hannah Gafter，感谢他们对我们的鼓励和包容。

Joshua Bloch

Neal Gafter

加利福尼亚州圣何塞市

2005年5月

目 录

第1章 绪论	1
第2章 表达式之谜	5
谜题1：奇数性	5
谜题2：找零时刻	7
谜题3：长整除	9
谜题4：初级问题	11
谜题5：十六进制的趣事	13
谜题6：多重转型	15
谜题7：互换内容	17
谜题8：Dos Equis	19
谜题9：半斤	21
谜题10：八两	23
第3章 字符之谜	25
谜题11：最后的笑声	25
谜题12：ABC	27
谜题13：动物庄园	29
谜题14：转义字符的溃败	31
谜题15：令人晕头转向的Hello	33
谜题16：行打印程序	35
谜题17：嗯？	37
谜题18：字符串奶酪	39
谜题19：漂亮的火花（块注释符）	41
谜题20：我的类是什么	43
谜题21：我的类是什么？镜头2	45
谜题22：URL的愚弄	47
谜题23：不劳无获	49
第4章 循环之谜	53
谜题24：尽情享受每一个字节	53
谜题25：无情的增量操作	55

谜题26：在循环中	57
谜题27：变幻莫测的i值	59
谜题28：循环者	61
谜题29：循环者的新娘	63
谜题30：循环者的爱子	65
谜题31：循环者的鬼魂	67
谜题32：循环者的诅咒	69
谜题33：循环者遇到了狼人	71
谜题34：被计数击倒了	73
谜题35：分分钟	75
第5章 异常之谜	77
谜题36：优柔寡断	77
谜题37：极端不可思议	79
谜题38：不受欢迎的宾客	81
谜题39：您好，再见	83
谜题40：不情愿的构造器	85
谜题41：域和流	87
谜题42：异常为循环而抛	89
谜题43：异常地危险	93
谜题44：删除类	97
谜题45：令人疲惫不堪的测验	101
第6章 类之谜	105
谜题46：令人混淆的构造器案例	105
谜题47：啊呀！狸猫变犬子	107
谜题48：我所得到的都是静态的	109
谜题49：比生命更大	111
谜题50：不是你的类型	113
谜题51：要点何在	115
谜题52：总和的玩笑	119
谜题53：做你的事吧	123
谜题54：Null与Void	125
谜题55：特创论	127

第 7 章 库之谜	131
谜题56：大问题	131
谜题57：名字里有什么	133
谜题58：产生它的散列码	137
谜题59：差是什么	139
谜题60：一行以毙之	141
谜题61：日期游戏	143
谜题62：名字游戏	145
谜题63：更多同样的问题	147
谜题64：按余数编组	149
谜题65：疑似排序的惊人传奇	152
第 8 章 更多类之谜	157
谜题66：一件私事	157
谜题67：对字符串上瘾	161
谜题68：灰色的阴影	163
谜题69：黑色的渐隐	165
谜题70：一揽子交易	167
谜题71：进口税	169
谜题72：终极危难	171
谜题73：隐私在公开	173
谜题74：同一性的危机	175
谜题75：头还是尾？	177
名字重用的术语表	180
第 9 章 更多库之谜	183
谜题76：乒乓	183
谜题77：乱锁之妖	185
谜题78：反射的污染	189
谜题79：狗狗的幸福生活	193
谜题80：更深层的反射	195
谜题81：无法识别的字符化	197
谜题82：啤酒爆炸	199
谜题83：诵读困难者的一神论	201
谜题84：戛然而止	203

谜题85：惰性初始化	205
第10章 高级谜题.....	209
谜题86：有害的括号垃圾	209
谜题87：紧张的关系	211
谜题88：原生类型的处理	213
谜题89：泛型迷药	217
谜题90：荒谬痛苦的超类	221
谜题91：序列杀手	224
谜题92：双绞线	229
谜题93：类的战争	231
谜题94：迷失在混乱中	233
谜题95：来份甜点	237
附录A 陷阱和缺陷的目录.....	239
A.1 词汇问题	240
A.2 整数运算	241
A.3 浮点运算	243
A.4 表达式计算	244
A.5 控制流	245
A.6 类初始化	246
A.7 实例的创建与销毁	247
A.8 其他与类和实例相关的主题	248
A.9 名字重用	250
A.10 字符串	251
A.11 I/O	253
A.12 线程	253
A.13 反射	255
A.14 序列化	256
A.15 其他库	257
附录B 书中幻图的注释.....	259
索引	265
参考文献	281

绪 论

本书充满了有关Java编程语言及其核心类库的谜题。任何具有Java应用知识的人都可以理解这些谜题，但是这其中的许多谜题即便是对最富经验的程序员而言，也是一种挑战。因此，如果你不能解决它们，千万不要感到沮丧。根据它们所使用的特性，我们对其进行松散的分组。但是请不要以为解决某个谜题的技巧是与其所在章的题目相关的，我们保留误导你的权力。

多数谜题都利用了那些会导致bug的行为，这些行为要么违反直觉，要么晦涩难懂，它们被认为是陷阱（trap）、缺陷（pitfall）和冷僻案例（corner case）。每一种平台都有这些问题，但是与其他能力相当的平台相比，Java要少得多。本书的目标就是要用这些谜题寓教于乐，让你避免底层的这些陷阱和缺陷。等学习完这些谜题之后，受代码中这类危险折磨的可能性将会大大降低，而在复审或修订代码时发现它们的可能性将会大大提高。

阅读本书时，在旁边放一台计算机。为了最大限度地发挥这些谜题的效用，还需要一个Java开发环境，例如Sun的JDK[JDK-5.0]，它应该支持5.0版本，因为有些谜题依赖于在这个版本中才引入的特性。可以从www.javapuzzlers.com下载这些谜题的源代码。除非你是极能忍辱负重之人，否则我们建议你在解决谜题之前就去下载这些代码，这比你自己录入要轻松得多。

大多数谜题都采用了短程序的形式来表示，这些程序看起来是要做某件事，但是实际上做的却是另外一件事。你的任务就是要指出这些程序要做什么。为了最大限度地发挥这些谜题的效用，我们推荐你采用下面的方式：

（1）研究程序并设法在不使用计算机的情况下预测它的行为。如果你看不出个所以然，那就继续看。

(2) 一旦你认为了解了程序要做什么，那么就运行它。它做了你认为它应该做的事了吗？如果没有，你能对所观察到的行为作出解释吗？

(3) 假设有问题，思考可以怎样改正该程序。

(4) 之后，且只能是之后，阅读解惑方案。

有些谜题要求你编写少量的代码。为了最大限度地发挥这些谜题的效用，我们推荐你尝试着在不使用（至少是暂时不使用）计算机的情况下解决它们，然后在计算机上测试你的解惑方案。如果你的代码不起作用，那么在阅读解惑方案之前，好好琢磨，看看是否可以让它起作用。

与大多数谜题书籍不同，这本书交替介绍谜题和解惑方案。这使读者在阅读本书时不用在谜题和解惑方案之间前后翻找。本书的布局使读者在从谜题转向解惑方案时必须翻页，这样读者就不必担心自己还在努力解决谜题时，一不留神看到了解惑方案，泄漏了天机。

我们鼓励阅读每一个解惑方案，即使你已经成功地解决了谜题。解惑方案所包含的分析，已经远远超过了对程序行为的简单解释，其中还讨论了相关的陷阱和缺陷，并提供了如何避免陷入此类危险的经验教训。与大多数最优实践指南一样，这些教训都不是必须要遵守的规则，但是你应该极少违反它们，即使违反也要有合适的理由。

大多数解惑方案都引用了《Java语言规范第3版》[JLS] (*The Java™ Language Specification, Third Edition*) 的相关章节。这些引用对理解谜题并不是必需的，但是如果还想更深入地钻研谜题底层的语言规则，那么它们就很有用了。与此类似，许多解惑方案都包含了对*Effective Java™ Programming Language Guide* [EJ]—书中相关项的引用。如果你想更深入地钻研最优实践，那么这些引用就很有用了。

有些解惑方案还包含了对语言和API设计决策的讨论，这些设计决策导致了在谜题中所展示的危险。这些“对语言设计者的教训”只是思想的食粮，就像其他任何食粮一样，它们应该被有保留地接受。语言设计决策不能彼此孤立地做出，每一种语言都体现了以细微方式相互作用的成千上万的设计决策。一个设计决策对某一种语言是正确的，但是对另外一种语言可能就是错误的。

在这些谜题中的许多陷阱和缺陷都是可以更正的，它们可以通过静态分析 (static analysis) 被自动探测到，静态分析即在不运行程序的情况下分析程序。有一些很优秀的通过静态分析探测bug的工具可供大家使用，例如Bill Pugh和David Hovemeyer的FindBugs[Hovemeyer04]。某些编译器和IDE工具也可以完成bug探测，例如Jikes和

Eclipse[Jikes, Eclipse]。如果你正在使用的编译器就是其中之一，那么直到你尝试解决某个谜题时才编译该谜题，这一点特别重要：编译器的警告信息可能会泄露解惑方案。

本书的附录是Java平台中的陷阱和缺陷的目录，它提供了一个对谜题中所采用的奇异案例的简明分类法，并且引用了谜题和其他相关的资源。在完成解决谜题的任务之前，不要去看这个附录。先阅读附录会使谜题变得索然无味。应该在完成了所有的谜题之后，才开始浏览这些参考资料。

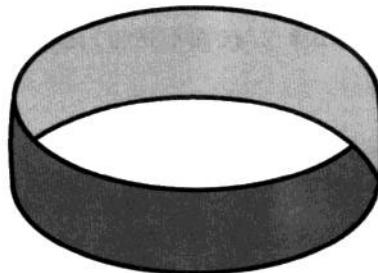
表达式之谜

在本章所描述的谜题都很简单，它们仅仅涉及表达式的计算。但是切记，不能仅仅因为它们简单，就认为它们很容易解决。

谜题1：奇数性

下面方法的目的是确定其惟一的参数是否为奇数。这个方法可行吗？

```
public static boolean isOdd(int i) {  
    return i % 2 == 1;  
}
```



解惑1：奇数性

奇数可定义为被2整除余数为1的整数。表达式`i % 2`计算的是`i`除以2时所产生的余数，因此看起来这个程序应该可行。遗憾的是，它不行；在四分之一的时间里它返回的都是错误的答案。

为什么是四分之一？因为在所有的`int`数值中，有一半是负数，而`isOdd`方法对所有负奇数的判断都会失败。在任何负整数上调用该方法都会返回`false`，无论该整数是偶数还是奇数。

这是Java对取余操作符（%）的定义所产生的后果。该操作符被定义为对于所有的`int`数值`a`和所有的非零`int`数值`b`，都满足下面的恒等式：

$$(a / b) * b + (a \% b) == a$$

换句话说，如果用`b`整除`a`，将商乘以`b`，然后加上余数，那么就得到了最初的值`a` [JLS 15.17.3]。该恒等式具有正确的含义，但是当与Java的截尾整数整除操作符[JLS 15.17.2]相结合时，它就意味着：当取余操作返回一个非零的结果时，它与左操作数具有相同的正负符号。

`isOdd`方法以及它所基于的对术语“奇数”的定义都假设所有的余数都是正数。虽然该假设对某些种类的整除[Boxing]是有意义的，但是Java的取余操作是与舍弃整除结果小数部分的整数整除操作完全匹配的。

当`i`是一个负奇数时，`i % 2`等于`-1`而不是`1`，因此`isOdd`方法将错误地返回`false`。为了防止这种意外，请测试你的方法在为每一个数值型参数传递负数、零和正数值时，其行为是否正确。

这个问题很容易改正。只需将`i % 2`与`0`而不是与`1`比较，并且使用相反的比较含义即可：

```
public static boolean isOdd(int i){
    return i % 2 != 0;
}
```

如果正在一个强调性能的环境中使用`isOdd`方法，那么用位操作符AND（&）替代

取余操作符会显得更好：

```
public static boolean isOdd(int i){  
    return (i & 1) != 0;  
}
```

第二个版本运行起来可能比第一个版本要快得多，这取决于你使用的是什么样的平台和虚拟机，并且不太可能出现运行得更慢的情况。按常规来说，整除和取余操作与其他的算术和逻辑操作相比要慢一些。仓促地优化是不好的，但是在上述情况下，更快的版本与最初的版本一样清晰明白，所以没有任何理由偏爱最初的版本。

总之，无论何时使用了取余操作符，都要考虑操作数和结果的符号。该操作符的行为在其操作数非负时是一目了然的，但是当一个或两个操作数是负数时，它的行为就不那么显而易见了。

谜题2：找零时刻

考虑下面这段话所描述的问题：

Tom在一家汽车配件商店购买一个价值1.10美元的火花塞，但是他钱包中都是两美元一张的钞票。如果他用一张两美元的钞票购买这个火花塞，那么应该找给他多少零钱呢？

下面是一个试图解决上述问题的程序，它会打印什么呢？

```
public class Change{  
    public static void main(String args[]){  
        System.out.println(2.00 - 1.10);  
    }  
}
```

解惑2：找零时刻

你可能会很天真地期望该程序能够打印出0.90，但是它怎能知道你想要打印小数点后两位小数呢？如果对在Double.toString文档[Java-API]中设定的将double类型的值转换为字符串的规则有所了解，你就会知道该程序打印出来的小数，是足以将double类型的值与最靠近它的临近值区分出来的最短小数，它在小数点之前和之后都至少有一位。因此，该程序应该打印0.9才合理。可能是很合理，但是并不正确。运行该程序，就会发现打印的是0.8999999999999999。

问题在于1.1这个数字不能被精确表示为一个double，因此被表示为最接近它的double值。该程序从2中减去的就是这个值。遗憾的是，这个计算的结果并不是最接近0.9的double值。作为结果的double值的最短表示就是你所看到的打印出来的那个可恶的数字。

更一般地说，问题在于并不是所有的小数都可以用二进制浮点数精确表示。如果使用的是JDK 5.0或更新的版本，那么你可能会受其诱惑，通过使用printf工具设置输出精度的方法改正该程序：

```
//Poor solution-still uses binary floating-point!
System.out.printf(" %.2f\n" 2.00-1.10);
```

这条语句打印的是正确结果，但是并不表示它就是对底层问题的通用解决方案：它使用的仍旧是二进制浮点数的double运算。浮点运算在一个范围很广的值域上提供了很好的近似，但是通常不能产生精确的结果。二进制浮点对于货币计算是非常不适合的，因为它不可能将0.1或者10的其他任何次负幂，精确表示为一个有限长度的二进制小数[EJ Item 31]。

解决该问题的一种方式是使用某种整数类型，例如int或long，并且以分为单位来执行计算。如果采纳了此路线，请确保该整数类型大到足以表示程序中将要用到的所有值。对该谜题来说，int就足够了。下面是用int类型、以分为单位表示货币值后重写的println语句。这个版本将打印出正确答案90美分：

```
System.out.println((200 - 110) + "cents");
```

解决该问题的另一种方式是使用执行精确小数运算的BigDecimal。它还可以通过JDBC与SQL DECIMAL类型进行互操作。这里要告诫一点：一定要用BigDecimal

(**String**)构造器，而千万不要用**BigDecimal(double)**。后一个构造器将用它的参数的精确值来创建一个实例。例如new **BigDecimal(.1)**，它将返回一个**BigDecimal**，也即0.10000000000000005551151231257827021181583404541015625。正确使用**BigDecimal**，程序就可以打印出我们所期望的结果0.90：

```
import java.math.BigDecimal;
public class Change {
    public static void main(String args[]){
        System.out.println(new BigDecimal("2.00").
                           subtract(new BigDecimal("1.10")));
    }
}
```

这个版本并不是十分完美，因为Java并没有为**BigDecimal**提供任何语言上的支持。使用**BigDecimal**的计算很有可能比那些使用原生类型的更慢，对某些大量使用小数计算的程序来说，这可能会成为问题，而对大多数程序来说，这显得一点也不重要。

总之，在需要精确答案的地方，要避免使用**float**和**double**；对于货币计算，要使用**int**、**long**或**BigDecimal**。对于语言设计者来说，应该考虑对小数运算提供语言支持。一种方式是提供对操作符重载的有限支持，使得运算符可以对数值引用类型起作用，例如**BigDecimal**。另一种方式是提供原始的小数类型，就像COBOL与PL/I所做的一样。

谜题3：长整除

这个谜题被称为长整除，因为它所涉及的程序是整除两个**long**型数值的。被除数表示一天里的微秒数，而除数表示一天里的毫秒数。这个程序会打印什么呢？

```
public class LongDivision {
    public static void main(String [] args){
        final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY/MILLIS_PER_DAY);
    }
}
```

解惑3：长整除

这个谜题看起来相当直观。每天的毫秒数和每天的微秒数都是常量。为清楚起见，它们都被表示成积的形式。每天的微秒数是 $(24\text{小时}/天 \times 60\text{分钟}/小时} \times 60\text{秒}/分钟} \times 1000\text{毫秒}/秒 \times 1000\text{微秒}/毫秒)$ 。而每天的毫秒数的不同之处只是少了最后一个因子1000。当你用每天的毫秒数来整除每天的微秒数时，除数中所有的因子都被约掉了，只剩下1000，这正是每毫秒包含的微秒数。除数和被除数都是long类型的，long类型大到了可以很容易地保存这两个乘积而不产生溢出。因此，看似程序打印的必定是1000。遗憾的是，它打印的是5。这里到底发生了什么呢？

问题在于常数MICROS_PER_DAY的计算确实溢出了。虽然计算的结果适合放入long中，并且其空间还有富余，但是这个结果并不适合放入int中。这个计算完全是以int运算来执行的，并且只有在运算完成之后，其结果才被提升为long。而此时已经太迟：计算已经溢出，它返回的是一个缩小了200倍的数值。从int提升为long是一种拓宽原生类型转换(widening primitive conversion) [JLS 5.1.2]，它保留了(不正确的)数值。这个值之后被MILLIS_PER_DAY整除，而MILLIS_PER_DAY的计算是正确的，因为它适合int运算。于是整除的结果是5。

那么为什么计算是以int运算来执行的呢？因为所有乘在一起的因子都是int数值。当两个int数值相乘时，你将得到另一个int数值。Java不具有目标确定类型的特性，这是一种语言特性，其含义是指存储结果的变量类型会影响计算所使用的类型。

通过使用long常量来替代int常量作为每一个乘积的第一个因子，我们就可以很容易地改正这个程序。这样做可以强制表达式中所有的后续计算都用long运算来完成。尽管这么做只在MICROS_PER_DAY表达式中是必需的，但是在两个乘积中都这么做是一种很好的方式。相似地，使用long作为乘积的第一个数值也并不总是必需的，但是这么做也是一种很好的形式。两个计算都以long数值开始可以清楚地表明它们都不会溢出。下面的程序将打印我们所期望的1000：

```
public class LongDivision{
    public static void main(String[] args){
        final long MICROS_PER_DAY = 24L * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24L * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY/MILLIS_PER_DAY);
    }
}
```

这个教训很简单：当你在操作很大的数字时，千万要提防溢出——它可是一个缄默杀手。即使用来保存结果的变量已足够大，也并不意味着要产生结果的计算具有正确的类型。当你拿不准时，就使用long运算来执行整个计算。

语言设计者可以从中吸取的教训是：降低缄默溢出产生的可能性也许确实是值得做的事。这可以通过支持不会产生缄默溢出的运算来实现。程序可以抛出一个异常而不是直接溢出，就像Ada所做的那样，或者它们可以在需要的时候自动地切换到一个更大的内部表示上以防止溢出，就像Lisp所做的那样。这两种方式都可能遭受与其相关的性能损失。降低缄默溢出的另一种方式是支持目标确定类型，但是这样会显著增加类型系统的复杂度[Modula-3 1.4.8]。

谜题4：初级问题

前面那个谜题是有点棘手，但它是有关整除的，每个人都知道整除是很麻烦的。下面的程序只涉及加法，它又会打印什么呢？

```
public class Elementary{
    public static void main(String[] args){
        System.out.println(12345+54321);
    }
}
```

解惑4：初级问题

从表面上看，这像是一个简单的谜题——简单到不需要纸和笔你就可以解决它。加号的左操作数的各位是从1到5升序排列的，而右操作数是降序排列的。因此，相应各位的和仍然是常数，程序必定打印66666。对于这样的分析，只有一个问题：当你运行该程序时，它打印的是17777。难道Java对打印这样的非常数字抱有偏见？不管怎么说，这看起来并不像是一个合理的解释。

事物往往有别于它的表象。以这个问题为例，它并没有打印我们想要的输出。请仔细观察`+操作符`的两个操作数。我们是将一个int类型的12345加到了long类型的54321上。请注意左操作数开头的数字1和右操作数结尾的小写字母l之间的细微差异。数字1的水平笔划（称为“臂（arm）”）和垂直笔划（称为“茎（stem）”）之间是一个锐角，而与此相对比的是，小写字母l的臂和茎之间是一个直角。

在大喊“恶心！”之前，你应该注意到这个问题确实已经引起了混乱，还应该注意到这个谜题的题目已经包含了一条暗示：初级问题（Elementary），看到了吗？最后，应该注意到这里确实有一个教训：**在long类型字面常量中，一定要用大写的L，千万不要用小写的l。**这样就可以彻底清除这个谜题所产生混乱的源头。

```
System.out.println(12345+5432L);
```

相类似的，要避免使用单个l字母作为变量名。例如，我们很难通过观察下面的代码段来判断打印的是列表l还是数字1。

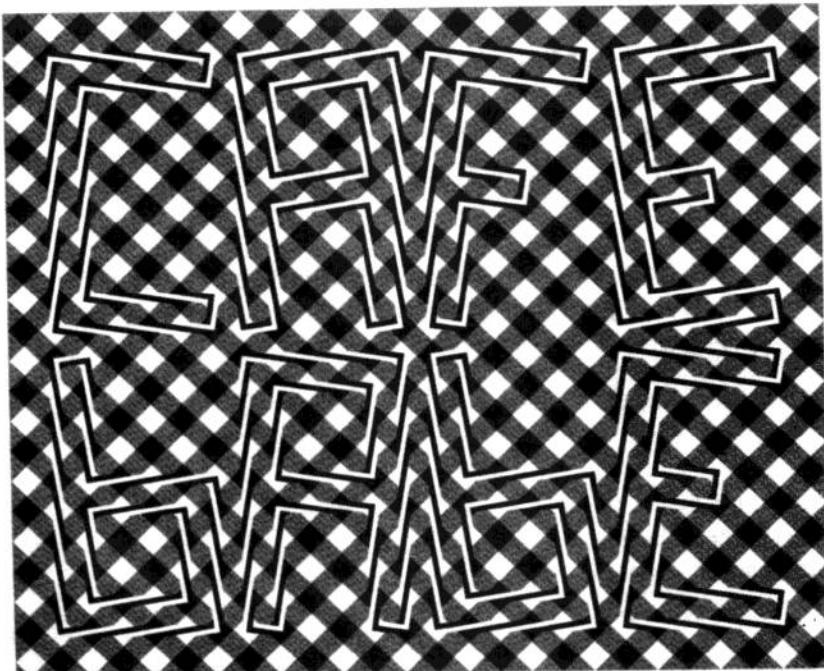
```
//Bad code-uses el(l)as a variable name
List<String> l = new ArrayList<String>();
l.add("Foo");
System.out.println(l);
```

总之，小写字母l和数字1在大多数打字机字体中几乎一样。为避免程序的读者对二者产生混淆，千万不要使用小写的l作为long型字面常量的结尾或是作为变量名。Java从C编程语言中继承良多，包括long型字面常量的语法。也许当初允许用小写的l来编写long型字面常量本身就是一个错误。

谜题5：十六进制的趣事

下面的程序是让两个十六进制字面常量相加，然后打印十六进制的结果。这个程序会打印什么呢？

```
public class JoyOfHex{  
    public static void main(String[] args){  
        System.out.println(  
            Long.toHexString(0x100000000L + 0xcafebabe));  
    }  
}
```



解惑5：十六进制的趣事

看似很明显，该程序应该打印1cafebabe。毕竟，这是十六进制数字100000000₁₆与cafebabe₁₆的和。该程序使用的是long型运算，它支持16位十六进制数，因此运算溢出是不可能的。然而，如果运行该程序，就会发现打印的是cafebabe，并没有任何前导的1。这个输出表示的是正确结果的低32位，但是不知何故，第33位丢失了。似乎程序执行的是int型运算而不是long型运算，或者是忘了加第一个操作数。这里到底发生了什么呢？

十进制字面常量具有一个很好的属性，即所有的十进制字面常量都是正的[JLS 3.10.1]，而十六进制或是八进制字面常量并不具备这个属性。想写一个负的十进制常量，可以使用一元取反操作符（-）连接一个十进制字面常量。以这种方式，你可以用十进制写任何int或long型数值，不管它是正的还是负的，并且负的十进制常数可以很明确地用一个减号符号来标识。但是十六进制和八进制字面常量并不是这样。它们可以具有正的以及负的数值。如果十六进制和八进制字面常量的最高位被置位了，那么它们就是负数。在这个程序中，数字0xcafebabe是int常量，它的最高位被置位了，所以它是一个负数。它等于十进制数值-889275714。

该程序执行的加法是一种混合类型的计算（mixed-type computation）：左操作数是long类型，而右操作数是int类型。为了执行该计算，Java将int类型的数值用拓宽原生类型转换[JLS 5.1.2]提升为long类型，然后对两个long类型数值相加。因为int是有符号的整数类型，所以这个转换执行的是符号扩展：它将负的int类型数值提升为一个在数值上相等的long类型数值。

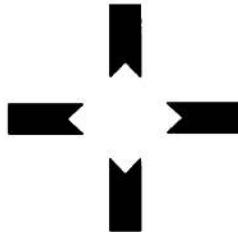
这个加法的右操作数0xcafebabe被提升为long类型的数值0xffffffffffffcafebabeL。之后这个数值加上了左操作数0x100000000L。当视为int类型时，经过符号扩展之后的右操作数的高32位是-1，而左操作数的高32位是1，两个数值相加得到了0，这也解释了为什么在程序输出中丢失了前导1。下面是手写的加法实现。（在加法上方的数字是进位。）

1111111	
0xffffffffffffcafebabeL	
+ 0x00000000100000000L	
<hr/>	
0x00000000cafebabeL	

修正该程序非常简单，只需用一个long十六进制字面常量来表示右操作数即可。这就可以避免具有破坏力的符号扩展，并且程序可以打印我们所期望的结果1cafebabe：

```
public class JoyOfHex{  
    public static void main(String[] args){  
        System.out.println(  
            Long.toHexString(0x100000000L + 0xcafebabeL));  
    }  
}
```

这个谜题给我们的教训是：混合类型的计算可能会产生混淆，尤其需要注意的是十六进制和八进制字面常量无需显式的减号符号就可以表示负的数值。为了避免这种窘境，通常最好是避免混合类型的计算。对于语言的设计者们来说，应该考虑支持无符号的整数类型，从而排除符号扩展的可能性。可能会有这样的争辩：应该禁用负的十六进制和八进制字面常量，但是这可能会打击程序员，他们经常使用十六进制字面常量来表示那些符号没有任何含义的数值。



谜题6：多重转型

转型用于将一个数值从一种类型转换到另一种类型。下面的程序连续使用了三个转型。它会打印什么呢？

```
public class Multicast{  
    public static void main (String[] args){  
        System.out.println((int)(char)(byte) -1);  
    }  
}
```

解惑6：多重转型

无论你怎样分析这个程序，都会感到迷惑。它以int数值-1开始，然后从int转型为byte，之后转型为char，最后转型回int。第一个转型将数值从32位窄化为8位，第二个转型将数值从8位拓宽为16位，最后一个转型又将数值从16位拓宽回32位。这个数值最终回到了起点吗？运行该程序，你就会发现不是。它打印的是65535，但是这是为什么呢？

该程序的行为紧密依赖于转型的符号扩展行为。Java使用了基于2的补码的二进制运算，因此int类型的数值-1的所有32位都是置位的。从int到byte的转型是简明的，它执行了一个窄化原生类型转换（narrowing primitive conversion）[JLS 5.1.3]，直接将除低8位之外的所有位全部砍掉。这样留下的是一个8位都是置位的byte，它仍旧表示-1。

从byte到char的转型稍微麻烦一点，因为byte是有符号类型，而char是无符号类型。在将一个整数类型转换成另一个宽度更宽的整数类型时，通常是可以保持其数值的，但是却不可能用一个char表示一个负的byte数值。因此，我们认为从byte到char的转换不是一个拓宽原生类型转换[JLS 5.1.2]，而是一个拓宽并窄化原生类型的转换（widening and narrowing primitive conversion）[JLS 5.1.4]：byte被转换成int，而这个int又被转换成char。

所有这些听起来有点复杂。幸运的是，有一条简单的规则能够描述从较窄的整型转换成较宽的整型时的符号扩展行为：如果最初的数值类型是有符号的，那么就执行符号扩展；如果它是char，那么不管它将要被转换成什么类型，都执行零扩展。了解这条规则可以很容易地解决这个谜题。

因为byte是有符号的类型，所以在将byte数值-1转换成char时，会发生符号扩展。作为结果的char数值的16位就都被置位了，因此它等于 $2^{16}-1$ ，即65535。从char到int的转型也是一个拓宽原生类型转换，所以这条规则告诉我们，它将执行零扩展而不是符号扩展。作为结果的int数值也就成了65535，这正是程序打印的结果。

尽管这条简单的规则描述了在有符号和无符号整型之间进行拓宽原生类型时的符号扩展行为，最好不要编写依赖于它的程序。如果你正在执行一个转型到char或从char转型的拓宽原生类型转换，并且这个char是仅有的无符号整型，那么最好将你的意图明确地表达出来。

如果你在将一个char数值c转型为一个宽度更宽的类型，并且不希望有符号扩展，那么为清晰表达意图，可以考虑使用一个位掩码，即使它并不是必需的：

```
int i = c & 0xffff;
```

或者，写一句注释来描述转换的行为：

```
int i = c; //Sign extension is not performed
```

如果你在将一个char数值c转型为一个宽度更宽的整型，并且希望有符号扩展，那么就先将char转型为一个short，它与char具有同样的宽度，但是它是有符号的。在给出了这种细微的代码之后，你也应该为它写一句注释：

```
int i = (short) c; //Cast causes sign extension
```

如果你在将一个byte数值b转型为一个char，并且不希望有符号扩展，那么必须使用一个位掩码来限制它。这是一种通用做法，不需要任何注释：

```
char c = (char) (b & 0xff);
```

如果你在将一个byte数值转换为char，并且希望有符号扩展，可以写一条注释：

```
char c=(char)b;//Sign extension is performed
```

这个教训很简单：**如果你通过观察不能确定程序将要做什么，那么它做的就很有可能不是你想要的。**要为明白清晰地表达你的意图而努力。尽管有这么一条简单的规则，描述了涉及有符号和无符号整型拓宽转换的符号扩展行为，但是大多数程序员都不知道它。如果你的程序依赖于它，那么就应该把你的意图表达清楚。

谜题7：互换内容

下面的程序使用了复合的异或赋值操作符，它所展示的技术是一种编程习俗。它会打印什么呢？

```
public class CleverSwap{
    public static void main(String[] args){
        int x = 1984; // (0x7c0)
        int y = 2001; // (0x7d1)
        x^= y^= x^= y;
        System.out.println("x="+x+"; y="+y);
    }
}
```

解惑7：互换内容

就像其名称所暗示的，这个程序应该交换变量x和y的值。如果你运行它，就会发现很悲惨，它失败了，打印的是 $x=0;y=1984$ 。

交换两个变量的最显而易见的方式是使用一个临时变量：

```
int tmp = x;
x = y;
y = tmp;
```

很久以前，当中央处理器只有少数寄存器时，人们发现可以通过利用异或操作符(\wedge)的属性 $(x \wedge y \wedge x) == y$ 来避免使用临时变量：

```
// Swaps variables without a temporary-Don't do this!
x = x ^ y;
y = y ^ x;
x = y ^ x;
```

即使回到那个年代，这项技术也很少被证明是合理的。因为CPU已经拥有很多寄存器，所以这项技术从来没有被证明是合理的。就像大多数“聪明”的代码一样，上面的代码很少被使用。更糟的是，它们把事情复杂化了，因为它们使用了在这个谜题中所展示的惯用法，即在一条单一语句中捆绑使用了三个异或操作。

在C编程语言中曾经使用过这个惯用法，更进一步，在C++中也使用了它，但是它并不保证在二者中都可以正确运行。有一点是肯定的，那就是它在Java中不能正确运行。Java语言规范描述了：操作符的操作数是从左向右求值的 [JLS 15.7]。为了求表达式 $x^=expr$ 的值，在计算 $expr$ 之前提取 x 的值，并且将这两个值的异或结果赋给变量 x [JLS 15.26.2]。在CleverSwap程序中，变量 x 的值被提取了两次，每次在表达式中出现时都提取一次，但是两次提取都发生在所有的赋值操作之前。

下面的代码详细地描述了将互换惯用法分解之后的行为，并且解释了为什么产生的输出：

```
// The actual behavior of x^=y^=x^=y in Java
int tmp1 = x;           // First appearance of x in the expression
int tmp2 = y;           // First appearance of y
int tmp3 = x ^ y;       // Compute x^y
x = tmp3;              // Last assignment:Store x^y in x
```

```

y = tmp2 ^ tmp3 ;      // 2nd assignment: Store original x value in y
x = tmp1 ^ y ;        // First assignment: Store 0 in x

```

在C和C++中，并没有指定表达式的计算顺序。当编译表达式 $x^=expr$ 时，许多C和C++编译器都是在计算 $expr$ 之后才提取 x 的值，这就使得上述的惯用法可以正常运转。尽管可以正常运转，它仍然违背了C/C++有关不能在两个连续的序列点之间重复修改变量的规则[ISO-C]。因此，在C和C++中也没有明确定义这个惯用法的行为。

为了突出其价值，还是可以写出不用临时变量就可以互换两个变量内容的Java表达式的。但是它既丑陋又无用：

```

// Rube Goldberg would approve, but don't ever do this!
y = (x ^= (y ^= x)) ^ y ;

```

这个教训很简单：在单个的表达式中不要对相同的变量赋值两次。表达式如果包含对相同变量的多次赋值，就会引起混乱，并且很少能够执行所希望的操作。即使对多个变量进行赋值也很容易出错。更一般地讲，要避免所谓聪明的编程技巧。它们都容易产生bug，难以维护，并且运行速度经常比它们所替代的简单直观的代码慢。

语言设计者可能会考虑禁止在一个表达式中对相同的变量多次赋值，但是在一般的情况下，强制执行这条禁令会因为别名机制的存在而显得很不灵活。例如，考虑表达式 $x = a[i]++ - a[j]++$ ，它是否递增了相同的变量两次呢？这取决于在计算表达式时 i 和 j 的值，并且编译器通常无法确定这一点。

谜题8：Dos Equis¹

这个谜题将测试你对条件操作符的掌握程度，这个操作符有一个更广为人知的名字：问号冒号操作符。下面的程序将会打印什么呢？

```

public class DosEquis{
    public static void main(String[] args){
        char x = 'X';
        int i = 0;
        System.out.print(true ? x : 0);
        System.out.print(false ? i : x);
    }
}

```

1. Dos Equis是一种在美国销量很大的啤酒品牌，字面意思可以理解为Dos下的等价物，作者借用了这个啤酒品牌来作为这个谜题的题目。——译者注

解惑8：Dos Equis

这个程序由两个变量声明和两个print语句构成。第一个print语句计算条件表达式(`true ? x : 0`)并打印结果，这个结果是char类型变量x的值'x'而第二个print语句计算表达式(`false? i:x`)并打印结果，这个结果依旧是x的'x'，因此这个程序应该打印xx。然而，运行该程序，你就会发现它打印的是x88。这种行为看起来挺怪的。第一个print语句打印的是x，而第二个打印的却是88。它们的不同行为说明了什么呢？

答案就在条件表达式规范[JLS 15.25]的一个阴暗角落里。请注意在这两个表达式中，每一个表达式的第二个和第三个操作数的类型都不相同：`x`是char类型的，而`0`和`i`都是int类型的。就像在谜题5的解答中提到的，混合类型的计算会引起混乱，而这一点在条件表达式中比在其他任何地方都表现得更明显。你可能考虑过，这个程序中两个条件表达式的结果类型是相同的，就像它们的操作数类型是相同的一样，尽管操作数的顺序颠倒了，但是实际情况并非如此。

确定条件表达式结果类型的规则过于冗长和复杂，很难完全记住它们，但是其核心就是以下三点：

- (1) 如果第二个和第三个操作数具有相同的类型，那么它就是条件表达式的类型。换句话说，可以通过绕过混合类型的计算来避免大麻烦。
- (2) 如果一个操作数的类型是`T`，`T`表示byte、short或char，而另一个操作数是一个int类型的常量表达式，它的值可以用类型`T`表示，那么条件表达式的类型就是`T`。
- (3) 否则，将对操作数类型进行二进制数字提升，而条件表达式的类型就是第二个和第三个操作数被提升之后的类型。

对本谜题而言(2)、(3)两点是关键。在程序的两个条件表达式中，一个操作数的类型是char，另一个的类型是int。在两个表达式中，int操作数都是0，它可以被表示成一个char。然而，只有第一个表达式中的int操作数是常量(0)，而第二个表达式中的int操作数是变量(i)。因此，第(2)点被应用到第一个表达式上，它返回的类型是char，而第(3)点被应用到第二个表达式上，其返回的类型是对int和char进行二进制数字提升之后的类型，即int[JLS 5.6.2]。

条件表达式的类型将确定调用哪一个重载的print方法。对第一个表达式来说，将调用`PrintStream.print(char)`，而对第二个表达式来说，将调用`PrintStream.print(int)`。前一个重载方法将变量x的值作为Unicode字符(x)打印，而后一个重载方法将其作为一个十

进制整数(88)打印。至此，谜题解开了。

将final修饰符用于i的声明可以把i转变为一个常量表达式，从而让程序打印xx，但是这仍旧会引起混乱。为了消灭这种混乱，最好是将i的类型从int更改为char，以避免混合类型的计算。

总之，通常最好是在条件表达式中使用类型相同的第二和第三操作数。否则，你和你程序的读者必须彻底理解这些表达式行为的复杂规范。

对语言设计者来说，也许可以设计一个牺牲部分灵活性，但是增加简洁性的条件操作符。例如，要求第二和第三操作数必须为相同的类型，这看起来就很合理。或者，可以定义条件操作符对常量没有任何特殊处理。为了让程序员更加容易接受这些选择，可以提供表示所有原生类型字面常量的语法。这也许确实是一个好主意，因为它增加了语言的一致性和完备性，同时又减少了对转型的需求。

谜题9：半斤

现在该轮到你编写代码了。好消息是，你只需为这个谜题编写两行代码，并为下一个谜题也编写两行代码。这有什么难的呢？给出一个对变量x和i的声明，使得下面这条语句合法：

```
x += i;
```

但是，下面这条不合法：

```
x = x + i;
```

解惑9：半斤

许多程序员会认为该谜题中的第一个表达式 ($x += i$) 只是第二个表达式 ($x = x + i$) 的简写方式。这并不准确。这两个表达式都被称为赋值表达式[JLS 15.26]。第二个表达式使用的是简单赋值操作符 (=)，而第一个表达式使用的是复合赋值操作符。(复合赋值操作符包括 $+=$ 、 $-=$ 、 $*=$ 、 $/=$ 、 $%=$ 、 $<=>$ 、 $>>=$ 、 $\&=$ 、 $\wedge=$ 和 $\mid=$ 。) Java语言规范中讲到，复合赋值 $E1 \ op = E2$ 等价于简单赋值 $E1 = (T)((E1) \ op (E2))$ ，其中 T 是 $E1$ 的类型，除非 $E1$ 只被计算一次[JLS 15.26.2]。

换句话说，复合赋值表达式自动地将所执行计算的结果转型为其左侧变量的类型。如果结果的类型与该变量的类型相同，那么这个转型不会造成任何影响。然而，如果结果的类型比该变量的类型要宽，那么复合赋值操作符将悄悄地执行一个窄化原生类型转换[JLS 5.1.3]。因此，我们有很好的理由去解释为什么尝试着执行等价的简单赋值可能会产生一个编译错误。

为了说明得更具体，并给这个谜题提供一个解决方案，假设我们在该谜题的两个赋值表达式之前有下面这些声明：

```
short x = 0;
int i = 123456;
```

复合赋值编译将不会产生任何错误：

```
x += i; // Contains a hidden cast!
```

你可能期望 x 的值在这条语句执行之后是 123 456，但是并非如此，它的值是 -7 616。 int 类型的数值 123456 对于 $short$ 来说太大了。自动产生的转型悄悄地把 int 数值的高两位截掉了。这也许不是你想要的。

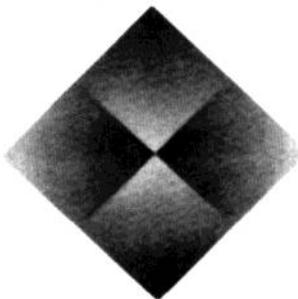
相应的简单赋值是非法的，因为它试图将 int 数值赋值给 $short$ 变量，而这需要一个显式的转型：

```
x = x + i; // Won't compile- "possible loss of precision"
```

这应该是明显的，复合赋值表达式可能是危险的。为了避免这种令人不快的突袭，

请不要将复合赋值操作符作用于**byte**、**short**或**char**类型的变量。在将复合赋值操作符作用于**int**类型的变量上时，要确保表达式右侧不是**long**、**float**或**double**类型。在将复合赋值操作符作用于**float**类型的变量上时，要确保表达式右侧不是**double**类型。这些规则足以防止编译器产生危险的窄化转型。

总之，复合赋值操作符会悄悄地产生一个转型。如果计算结果的类型宽于变量的类型，那么所产生的转型就是危险的窄化转型。这样的转型可能会悄悄地丢掉精度或数量值。对语言设计者来说，也许让复合赋值操作符产生一个不可见的转型本身就是一个错误；对于其中的变量类型比计算结果窄的复合赋值，也许应该让其非法才对。



谜题10：八两

与上面的例子相反，给出变量x和i的声明使如下的语句合法：

```
x = x + i;
```

但是，下面这条语句不合法：

```
x += i;
```

乍一看，这个谜题可能看起来与前面一个谜题相同。但是请放心，它们并不一样。这两个谜题在哪一条语句必是合法的，以及哪一条语句必是不合法的方面，正好相反。

解惑10：八两

就像前面的谜题一样，这个谜题也依赖于有关复合赋值操作符规范的细节。二者的相似之处仅止于此。基于前面的谜题，你可能会认为：复合赋值操作符比简单赋值操作符的限制更少。在一般情况下，这是对的，但是有这么一个领域，在其中简单赋值操作符更宽松。

复合赋值操作符要求两个操作数都是原生类型的，例如int，或包装了的原生类型，例如Integer，但是有一个例外：如果在+=操作符左侧的操作数是String类型的，那么它允许右侧的操作数是任意类型，在这种情况下，该操作符执行的是字符串连接操作[JLS 15.26.2]。简单赋值操作符（=）允许其左侧的是对象引用类型，这就更宽松了：可以使用它们来表示任何你想要表示的内容，只要表达式的右侧与左侧的变量是赋值兼容的即可[JLS 5.2]。

可以利用这一差异来解决该谜题。要想用+=操作符来执行字符串连接操作，就必须将左侧的变量声明为String类型。通过使用直接赋值操作符，字符串连接的结果可以存放到一个Object类型的变量中。

为了说得更具体，并给这个谜题提供一个解决方案，并假设我们在该谜题的两个赋值表达式之前有下面这些声明：

```
Object x = "Buy";
String i = "Effective Java!";
```

简单赋值是合法的，因为x+i是String类型的，而String类型与Object类型兼容：

```
x = x + i;
```

复合赋值是非法的，因为左侧是Object引用类型，而右侧是String类型：

```
x += i;
```

这个谜题对程序员来说几乎算不上什么教训。对语言设计者来说，加法的复合赋值操作符应该在右侧是String类型的情况下，允许左侧是Object类型。这项修改将根除这个谜题所示的违背直觉的行为。

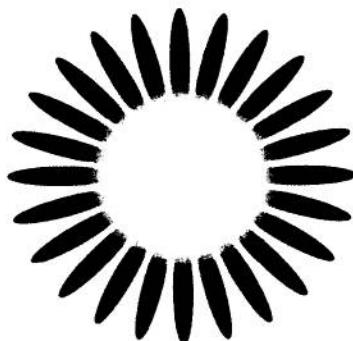
字符之谜

在本章所描述的谜题涉及字符串、字符和其他文本数据。

谜题11：最后的笑声

下面的程序将打印什么？

```
public class LastLaugh{  
    public static void main(String args[]){  
        System.out.print("H"+'a");  
        System.out.print('H'+'a');  
    }  
}
```



解惑11：最后的笑声

如果你和大多数人一样，那么就会认为这个程序将打印HaHa。该程序看起来好像是用两种方式连接了H和a，但是你所见为虚。运行这个程序，就会发现打印的是Ha169。那么，为什么它会产生这样的行为呢？

正如我们所期望的，对第一个System.out.print的调用打印的是Ha：它的参数是表达式 "H" + "a"，显然它执行的是一个字符串连接。而对第二个System.out.print的调用就是另外一回事了。问题在于'H'和'a'是字符型字面常量，因为这两个操作都不是字符串类型的，所以+操作符执行的是加法而不是字符串连接。

编译器在计算常量表达式'H'+'a'时，是通过我们熟知的拓宽原生类型转换[JLS5.1.2, 5.6.2]将两个具有字符型数值的操作数('H'和'a')提升为int数值而实现的。从char到int的拓宽原生类型转换是将16位的char数值零扩展到32位的int。对于'H'，char数值是72，而对于'a'，char数值是97，因此表达式'H'+'a'等价于int常量72+97，或169。

站在语言的立场上，若干个char和字符串的相似之处是虚幻的。语言所关心的是，char是一个无符号16位原生类型整数——仅此而已。对类库来说就不仅如此了，类库包含了许多可以接受char参数的方法，将其作为Unicode字符处理。

那么应该怎样将字符连接在一起呢？可以使用类库。例如，可以使用一个字符串缓冲区：

```
StringBuffer sb = new StringBuffer();
sb.append('H');
sb.append('a');
System.out.println(sb);
```

这么做可以正常运行，但是显得很丑陋。其实有办法避免这种啰嗦的方法。可以通过确保至少有一个操作数为字符串类型，来强制+操作符执行字符串连接操作，而不是加法操作。这种常见的惯用法用一个空字符串("")作为一个连接序列的开始，如下所示：

```
System.out.print ("+"+'H'+ 'a');
```

这种惯用法可以确保子表达式都被转型为字符串。尽管这很有用，但是多少有一点难看，而且它自身可能会引发某些混淆。

能猜到下面的语句将打印什么吗？如果不能确定，就试一下：

```
System.out.println("2 + 2 =" + 2+2);
```

如果使用的是JDK 5.0，还可以使用

```
System.out.printf("%c%c", 'H', 'a');
```

总之，使用字符串连接操作符要格外小心。当且仅当+操作符的操作数中至少有一个是**String**类型时，才会执行字符串连接操作；否则，执行加法。如果要连接的数值没有一个是字符串类型的，那么你可以有几种选择：预置一个空字符串；将第一个数值用**String.valueOf**显式地转换成一个字符串；使用一个字符串缓冲区；或者如果使用的是JDK 5.0，可以用**printf**方法。

这个谜题还包含了一个给语言设计者的教训。即使在Java中只在有限的范围内得到支持，操作符重载仍然会引起混淆。为字符串连接而重载+操作符可能就是一个已铸成的错误。

谜题12：ABC

这个谜题要问的是一个悦耳的问题，下面的程序将打印什么呢？

```
public class Abc{
    public static void main(String[] args){
        String letters = "ABC";
        char[] numbers = {'1', '2', '3'};
        System.out.println(letters + " easy as " + numbers);
    }
}
```

解惑12：ABC

可能大家希望这个程序打印ABC easy as 123。遗憾的是，它没有。如果你运行它，就会发现它打印的是诸如ABC easy as [C@16f0472之类的东西。为什么这个输出会如此丑陋？

尽管char是一个整数类型，但是许多类库都对其进行了特殊处理，因为char数值通常表示的是字符而不是整数。例如，将一个char数值传递给println方法会打印一个Unicode字符而不是它的数字代码。字符数组受到了相同的特殊处理：println的char[]重载版本会打印数组所包含的所有字符，而String.valueOf和StringBuffer.append的char[]重载版本的行为也是类似的。

然而，在这些方法中没有定义字符串连接操作符。该操作符被定义为先对它的两个操作数执行字符串转换，然后连接所产生的两个字符串。对包括数组在内的对象引用的字符串转换定义如下[JLS 15.18.1.1]：

如果引用为null，将其转换成字符串"null"。否则，该转换的执行就像无参数调用该引用对象的toString方法一样；但是如果调用toString方法的结果是null，那么就用字符串"null"来代替。

那么，在一个非空char数组上调用toString方法会产生什么样的行为呢？数组是从Object那里继承的toString方法[JLS 10.7]，规范中描述到：“返回一个字符串，它包含了该对象所属类的名字，'@'符号，以及表示对象散列码的一个无符号十六进制整数”[Java-API]。有关Class.getName的规范描述到：在char[]类型的类对象上调用该方法的结果为字符串"[c]"。将它们连接起来就形成了在我们的程序中打印的那个丑陋的字符串。

有两种方法可以修正这个程序。可以在调用字符串连接操作之前，显式地将一个数组转换成一个字符串：

```
System.out.println(letters + "easy as" +
                    String.valueOf(numbers));
```

或者，可以将System.out.println调用分解为两个调用，以利用println的char[]重载版本：

```
System.out.print(letters + " easy as ");
System.out.println(numbers);
```

注意，这些修正只有在调用了正确的`valueOf`和`println`方法重载版本的情况下，才能正常运行。换句话说，它们严格依赖于数组引用的编译期类型。下面的程序说明了这种依赖性。它像是所描述的第二种修正方式的具体实现，但是其产生的输出却与最初程序产生的输出一样丑陋，因为它调用的是`println`的`Object`重载版本，而不是`char[]`重载版本。

```
// Broken-invokes the wrong overloading of println!
class Abc{
    public static void main(String[] args){
        String letters = "ABC";
        Object numbers = new char[] { '1', '2', '3' };
        System.out.print(letters + " easy as ");
        System.out.println(numbers); // Invokes println (Object)
    }
}
```

总之，`char`数组不是字符串。要想将一个`char`数组转换成一个字符串，就要调用`String.valueOf(char[])`方法。某些类库中的方法对`char`数组提供了类似字符串的支持，通常是提供一个`Object`版本的重载方法和一个`char[]`版本的重载方法，而只有后者才能产生我们想要的行为。

对语言设计者的教训是：`char[]`类型可能应该覆写`toString`方法，使其返回数组中包含的字符。更一般地讲，数组类型可能都应该覆写`toString`方法，使其返回数组内容的字符串表示。

谜题13：动物庄园

George Orwell的*Animal Farm*一书的读者可能还记得老上校的宣言：“所有的动物都是平等的。”下面的Java程序试图测试这项宣言。它将打印什么呢？

```
public class AnimalFarm{
    public static void main(String[] args){
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: "
            + pig == dog);
    }
}
```

解惑13：动物庄园

对该程序进行表面分析后，可能会认为它应该打印Animal are equal: true。毕竟，pig和dog都是final的string类型变量，它们都被初始化为字符序列"length:10"。换句话说，被pig和dog引用的字符串是且永远是彼此相等的。然而，==操作符并不测试两个对象是否相等，它测试的是两个对象引用是否相同。换句话说，它测试这两个对象引用是否正好引用到同一对象上。在本例中，它们并非引用到同一对象上。

你可能知道String类型的编译期常量是内存限定的[JLS 15.28]。换句话说，任何两个String类型的常量表达式，如果指定的是相同的字符序列，那么它们就用同一对象引用表示。如果用常量表达式来初始化pig和dog，那么它们确实会指向相同的对象，但是dog并不是用常量表达式初始化的。既然语言已经对在常量表达式中允许出现的操作作出了限制，而方法调用又不在其中，那么，这个程序就应该打印Animal are equal: false，对吗？

嗯，实际上不对。运行该程序，就会发现它打印的只是false，并没有其他的任何东西。它没有打印Animal are equal:。它怎么会不打印这个字符串字面常量呢？毕竟打印它才是正确的呀！谜题11的解惑方案包含了一条暗示：+操作符，不论是用作加法还是字符串连接操作，它都比 == 操作符的优先级高。因此，println方法的参数是按照下面的方式计算的：

```
System.out.println(("Animals are equal:" + pig) == dog);
```

这个布尔表达式的值当然是false，它正是该程序所打印的输出。有一个能够避免此类窘境的方法：**在使用字符串连接操作符时，总是将不平凡的操作数用括号括起来**。更一般地讲，当你不能确定是否需要括号时，应该选择稳妥的做法，将它们括起来。如果在println语句中像下面这样把比较部分括起来，将产生所期望的输出Animals are equal: false：

```
System.out.println("Animals are equal:"+ (pig == dog));
```

正如可以论证的那样，该程序仍然有问题。**如果可以的话，你的代码应该很少依赖于字符串常量的内存限定机制**。内存限定机制只是设计用来减少虚拟机内存占有量，并不是程序员的一种工具。就像这个谜题所示范的，哪一个表达式会产生字符串常量并非

总是显而易见。更糟的是，如果你的代码依赖于内存限定机制实现操作的正确性，那么你就必须仔细地了解哪些域和参数必须是内存限定的。编译器不会帮你检查这些不变量，因为内存限定的和不限定的字符串使用相同的类型（`String`）来表示。在内存中限定字符串失败而导致的bug是很难以探测的。

在比较对象引用时，应该优先使用**equals**方法而不是`==`操作符，除非需要比较的是对象的标识而不是对象的值。通过把这个教训应用到我们的程序中，我们给出了下面的`println`语句，这才是它应该具有的模样。很明显，在用这种方式修正了该程序之后，它将打印`true`：

```
System.out.println("Animals are equal: "+pig.equals(dog));
```

这个谜题对语言设计者来说有两个教训。字符串连接的优先级不应该和加法一样。这意味着重载 + 操作符来执行字符串连接是有问题的，就像在谜题11中提到的一样。另外，对于不可修改的类型，例如String，其引用的等价性比值的等价性更加让人感到迷惑。也许 == 操作符在被应用于不可修改的类型时应该执行值比较。要实现这一点，一种方法是将 == 操作符作为equals方法的简便写法，并提供一个单独的类似于System.identityHashCode的方法来执行引用标识的比较。

谜题14：转义字符的溃败

下面的程序使用了两个Unicode的转义字符，它们是用其十六进制代码来表示Unicode字符。那么，这个程序会打印什么呢？

解惑14：转义字符的溃败

对该程序的一种很肤浅的分析会认为它应该打印26，因为在由两个双引号 "a\u0022.length() + \u0022b" 标识的字符串之间总共有26个字符。稍微深入一点的分析会认为该程序应该打印16，因为两个Unicode转义字符每一个在源文件中都需要用6个字符来表示，但是它们只表示字符串中的一个字符。因此这个字符串应该比它的外表看起来要短10个字符。运行这个程序，就会发现事情并不是这样。它打印的既不是26也不是16，而是2。

理解这个谜题的关键是要知道：Java对在字符串字面常量中的Unicode转义字符没有提供任何特殊处理。编译器在将程序解析成各种符号之前，先将Unicode转义字符转换成为它们所表示的字符[JLS 3.2]。因此，程序中的第一个Unicode转义字符将作为一个单字符字符串字面常量 ("a") 的结束引号，而第二个Unicode转义字符将作为另一个单字符字符串字面常量 ("b") 的开始引号。程序打印的是表达式 "a".length() + "b".length()，即2。

如果该程序的编写者确实希望得到这种行为，那么下面的语句将更清楚：

```
System.out.println("a".length() + "b".length());
```

因此，更有可能的情况是该编写者希望将两个双引号字符置于字符串字面常量的内部。使用Unicode转义字符是不能实现这一点的，但是可以使用转义字符序列来实现[JLS 3.10.6]。表示一个双引号的转义字符序列是一个反斜杠后面紧跟着一个双引号 (\")。如果将最初程序中的Unicode转义字符用转义字符序列来替换，那么它将打印所期望的16：

```
System.out.println("a\".length() + \"b\".length());
```

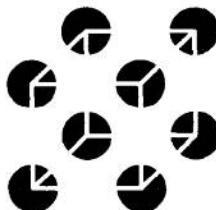
许多字符都有相应的转义字符序列，包括单引号 ('\')、换行 (\n)、制表符 (\t) 和反斜杠 (\\")。可以在字符字面常量和字符串字面常量中使用转义字符序列。实际上，你可以通过使用被称为八进制转义字符的特殊类型转义字符序列，将任何ASCII字符置于一个字符串字面常量或一个字符字面常量中，但是最好是尽可能地使用普通的转义字符序列。普通的转义字符序列和八进制转义字符都比Unicode转义字符要好得多，因为与Unicode转义字符不同，在程序被解析为各种符号之后才处理转义字符序列。

本书中的所有程序都是使用Unicode的ASCII子集写的。ASCII是字符集的最小公共特性集，它只有128个字符，但是Unicode有超过65 000个字符。在只使用ASCII字符的程

序中，一个Unicode转义字符可以用来插入一个Unicode字符。一个Unicode转义字符精确地等价于它所表示的字符。

Unicode转义字符被设计用于需要插入一个不能用源文件字符集表示的字符的情况。它们主要用于将非ASCII字符置于标识符、字符串字面常量、字符字面常量以及注释中。偶尔，Unicode转义字符也被用来在看起来颇为相似的几个字符中明确地标识其中的某一个，从而增加程序的清晰度。

总之，在字符串和字符字面常量中优先选择的是转义字符序列，而不是Unicode转义字符。Unicode转义字符可能会因为它们在编译序列中被过早地处理而引起混乱。不要使用Unicode转义字符来表示ASCII字符。在字符串字面常量和字符字面常量中，应该使用转义字符序列；对于除这些字面常量之外的情况，应该直接将ASCII字符插入到源文件中。



谜题15：令人晕头转向的Hello

下面的程序是对一个老生常谈的例子做了稍许变化之后的版本。它会打印什么呢？

```
/*
 * Generated by the IBM IDL-to-Java compiler, version 1.0
 * from F:\TestRoot\apps\ai\units\include\PolicyHome.idl
 * Wednesday, June 17, 1998 6:44:40 o'clock AM GMT+00:00
 */
public class Test{
    public static void main(String[] args){
        System.out.print("Hell");
        System.out.println("o world");
    }
}
```

解惑15：令人晕头转向的Hello

这个谜题看起来相当简单。该程序包含了两条语句，第一条打印Hello，而第二条在同一行打印world，从而有效地连接两个字符串。因此，你可能期望该程序打印Hello world。但是很可惜，你错了，实际上，它根本无法通过编译。

问题在于注释的第三行，它包含了字符\units。这些字符以反斜杠(\)以及紧跟着的字母u开头，而它(\u)表示的是一个Unicode转义字符的开始。遗憾的是，这些字符后面没有紧跟四个十六进制的数字，因此，这个Unicode转义字符是病构的，而编译器被要求拒绝该程序。**Unicode转义字符必须是良构的，即使出现在注释中也是如此。**

在注释中插入一个良构的Unicode转义字符是合法的，但是几乎没有什么理由去这么做。程序员有时会在JavaDoc注释中使用Unicode转义字符来在文档中生成特殊的字符。

```
// Questionable use of Unicode escape in Javadoc comment
/**
 * This method calls itself recursively, causing a
 * <tt>StackOverflowError</tt> to be thrown.
 * The algorithm is due to Peter von der Ah\u00e9.
 */
```

这项技术表现了Unicode转义字符的不必要的用法。在**Javadoc**注释中，应该使用**HTML实体转义字符**来代替**Unicode转义字符**：

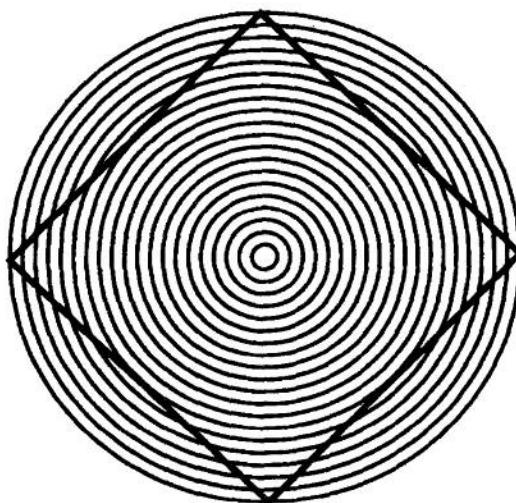
```
/**
 * This method calls itself recursively, causing a
 * <tt>StackOverflowError</tt> to be thrown.
 * The algorithm is due to Peter von der Ah&eacute;.
 */
```

前面的两个注释都应该使得在文档中出现的名字为“Peter der Ahé”，但是后一个注释在源文件中也是可理解的。

可能你会感到很诧异，在这个谜题中，问题出在注释这一信息来源于一个实际的bug报告。该程序是机器生成的，这使得我们很难追踪到问题的源头——IDL-to-Java编译器。为了避免其他程序员也陷入此境地，在没有预先处理Windows文件名，以消除反斜杠的

情况下，工具应该确保不将windows文件名置于所生成的Java源文件的注释中。

总之，要确保字符\u不在一个合法的Unicode转义字符上下文之外出现，即使在注释中也是如此。在机器生成的代码中要特别注意此问题。



谜题16：行打印程序

行分隔符（line separator）是为分隔文本行的字符或字符串而起的名字，并且在不同的平台上它是存在差异的。在Windows平台上，它由CR字符（回车）和紧随其后的LF字符（换行）组成。在UNIX平台上，通常引用单独的LF字符作为换行字符。下面的程序将这个字符传递给println方法。它将打印什么？它的行为是否依赖于平台？

```
public class LinePrinter{
    public static void main(String[] args){
        // Note: \u000A is Unicode representation of linefeed (LF)
        char c = 0x000A;
        System.out.println(c);
    }
}
```

解惑16：行打印程序

这个程序的行为是平台无关的：它在任何平台上都不能通过编译。如果你尝试着去编译它，就会得到类似下面的出错信息：

```
LinePrinter.java:3: ';' expected
    // Note: \u000A is Unicode representation of linefeed (LF)
               ^
1 error
```

如果你和大多数人一样，那么这条信息对界定问题毫无用处。

这个谜题的关键就是程序第三行的注释。与最好的注释一样，这条注释是准确的。遗憾的是，它有一点准确过头了。编译器不仅会在将程序解析成为符号之前把Unicode转义字符转换成它们所表示的字符（谜题14），而且它是在丢弃注释和空格之前做这些事的[JLS 3.2]。

这个程序包含了一个Unicode转义字符（\u000A），它位于程序惟一的注释行中。就像注释所陈述的，这个转义字符表示换行符，编译器将在丢弃注释之前适时地转换它。遗憾的是，这个换行符是开始注释的两个斜杠符之后的第一个行终结符（line terminator），因此它将终止该注释[JLS 3.4]。所以，该转义字符之后的字（is Unicode representation of linefeed (LF)）就不是注释的一部分了，而且它们在语法上也无效。

为了说明得更具体，在转换Unicode转义字符为它所表示的字符之后，该程序如下所示：

```
public class LinePrinter{
    public static void main(String[] args){
        // Note:
        is Unicode representation of linefeed (LF)
        char c = 0x000A;
        System.out.println(c);
    }
}
```



修正该程序最简单的方式就是在注释中移除Unicode转义字符，但是更好的方式是用一个转义字符序列而不是一个十六进制整型字面常量来初始化c，从而消除使用注释的必要：

```
public class LinePrinter{
    public static void main(String[] args){
        char c = '\n';
        System.out.println(c);
    }
}
```

只要这么做了，程序就可以编译并运行，但是这仍然是一个有问题的程序：它是平台相关的，其原因正是本谜题所要提醒的。在某些平台上，例如UNIX，它将打印两个完整的行分隔符；但是在其他平台上，例如Windows，它就不会产生这样的行为。尽管这些输出用肉眼看起来是一样的，但是如果它们被存储到文件中，或是输出到后续的其他处理程序中，就很容易引发问题。

如果想打印两行空行，你应该调用println两次。如果使用的是JDK 5.0，你可以使用带有格式化字符串"%n%n"的printf来代替println。%n的每一次出现都将导致printf打印一个恰当的、与平台相关的行分隔符。

我们希望，上面三个谜题已经使你信服：Unicode转义字符绝对会产生混乱。教训很简单：除非确实是必需的，否则就不要使用**Unicode转义字符**。它们很少是必需的。

谜题17：嗯？

下面的是一个合法的Java程序吗？如果是，它会打印什么呢？

```
\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020\u0020
\u0063\u006c\u0061\u0073\u0073\u0020\u0055\u0067\u006c\u0079
\u007b\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020
\u0020\u0020\u0020\u0020\u0073\u0074\u0061\u0074\u0069\u0063
\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u0069\u006e\u0028
\u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005d\u0020\u0020
\u0020\u0020\u0020\u0020\u0061\u0072\u0067\u0073\u0029\u007b
\u0053\u0079\u0073\u0074\u0065\u006d\u002e\u006f\u0075\u0074
\u002e\u0070\u0072\u0069\u006e\u0074\u006c\u006e\u0028\u0020
\u0022\u0048\u0065\u006c\u006c\u006f\u0020\u0077\u0022\u002b
\u0022\u006f\u0072\u006c\u0064\u0022\u0029\u003b\u007d\u007d
```

解惑17：嗯？

这当然是合法的Java程序！这不是显而易见的吗？它会打印Hello World。噢，可能不是那么明显。事实上，该程序根本让人无法理解。每当你没必要地使用了一个Unicode转义字符时，都会使你的程序缺失一点可理解性，而该程序将这种作法发挥到了极致。如果你很好奇，可以看看下面给出的在Unicode转义字符都被转换为它们所表示的字符之后该程序的样子：

```
public
class Ugly
{public
    static
void main(
String[]
    args){
System.out
.println(
"Hello w"+
"orld");}}
```

下面给出了将其进行格式化整理之后的样子：

```
public class Ugly {
    public static void main(String[] args){
        System.out.println("Hello w"+"orld");
    }
}
```

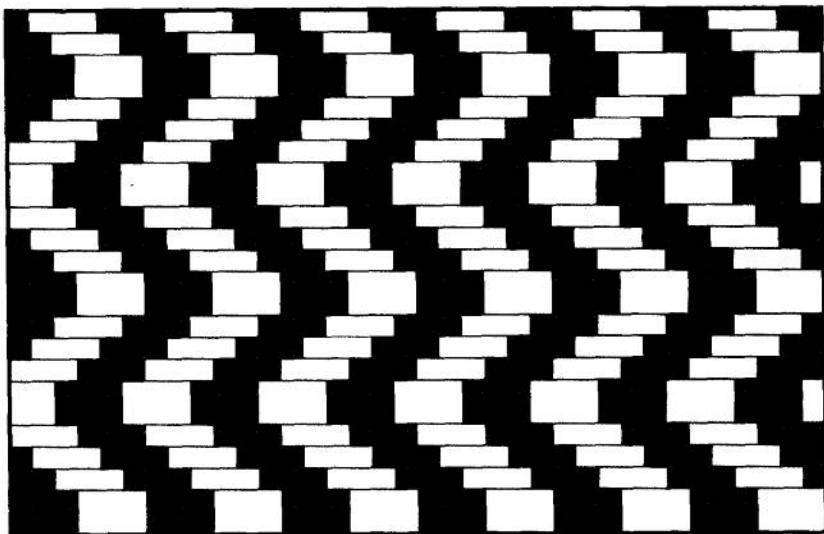
这个谜题的教训是：仅仅是因为你可以不以应有的方式进行表达。或者说，如果你这么做会造成损害，那么就请不要这么做！更严肃地讲，这个谜题是对前面三个教训的补充：只有在你要向程序中插入用其他任何方式都无法表示的字符时，**Unicode**转义字符才是必需的，除此之外的任何情况都应该避免使用它们。Unicode转义字符降低了程序的清晰度，并且增加了产生bug的可能性。

对语言的设计者来说，也许使用Unicode转义字符来表示ASCII字符应该被定义为非法。这样就可以使得在谜题14、15和17（本谜题）中的程序非法，从而消除了大量的混乱。这个限制对程序员并不会造成任何困难。

谜题18：字符串奶酪

下面的程序从一个字节序列创建一个字符串，然后迭代遍历字符串中的字符，并将它们作为数字打印。请描述程序打印的数字序列：

```
public class StringCheese {  
    public static void main(String args[]) {  
        byte bytes[] = new byte[256];  
        for (int i = 0; i < 256; i++)  
            bytes[i] = (byte)i;  
        String str = new String(bytes);  
        for (int i = 0, n = str.length(); i < n; i++)  
            System.out.print((int)str.charAt(i) + " ");  
    }  
}
```



解惑18：字符串奶酪

首先，用从0到255中每一个可能的byte数值初始化byte数组，然后将这些byte数值通过String构造器转换成char数值。最后，将char数值转型为int数值并打印。打印的数值肯定是非负整数，因为char数值是无符号的，因此，你可能期望该程序将按顺序打印0到255的整数。

运行该程序，可能会看到这样的序列。但是再运行一次，可能看到的就不是这个序列了。我们在四台机器上运行它，会看到四个不同的序列，包括前面描述的那个序列。这个程序甚至不能保证正常终止，比打印其他任何特定字符串都更缺乏保证。它的行为完全是不确定的。

这里的罪魁祸首就是String(byte[])构造器。有关它的规范描述道：“在通过解码使用平台缺省字符集的指定byte数组来构造一个新的String时，该新String的长度是字符集的一个函数，因此，它可能不等于byte数组的长度。当给定的所有字节在缺省字符集中并非全部有效时，这个构造器的行为是不确定的” [Java-API]。

到底什么是字符集？从技术角度上讲，它是“被编码的字符集合和字符编码模式的结合物” [Java-API]。换句话说，字符集是一个包，包含了字符、表示字符的数字编码以及在字符编码序列和字节序列之间来回转换的方式。转换模式在字符集之间存在着很大的区别：某些是在字符和字节之间一对一的映射，但是大多数都不是这样。ISO-8859-1是惟一能够让该程序按顺序打印从0到255的整数的缺省字符集，它更为大家所熟知的名字是Latin-1[ISO-8859-1]。

J2SE运行期环境（JRE）的缺省字符集依赖于底层的操作系统和语言。如果想知道你的JRE的缺省字符集，并且使用的是5.0或更新的版本，那么你可以通过调用java.nio.charset.Charset.defaultCharset()来了解。如果使用的是较早版本，那么你可以通过阅读系统属性“file.encoding”来了解。

幸运的是，没有强制要求必须容忍各种稀奇古怪的缺省字符集。在char序列和byte序列之间转换时，可以且通常应该显式地指定字符集。除了接受byte数组之外，还可以接受一个字符集名称的String构造器就是专为此目的而设计的。如果你用下面的构造器替换最初程序中的String构造器，那么不管缺省的字符集是什么，该程序都保证能够按照顺序打印从0到255的整数：

```
String str = new String(bytes, "ISO-8859-1");
```

声明这个构造器会抛出UnsupportedEncodingException异常，因此必须捕获它，或者更适宜的方式是声明main方法抛出它，否则程序不能通过编译。尽管如此，该程序实际上不会抛出异常。Charset的规范要求Java平台的每一种实现都要支持某些种类的字符集，ISO-8859-1就位列其中。

这个谜题的教训是：每当要将一个byte序列转换成一个String时，你都在使用一个字符集，不管是否显式地指定了它。如果想让你的程序行为可预知，那么在每次使用字符集时都明确地指定它。对API的设计者来说，提供这么一个依赖于缺省字符集的String(byte[])构造器可能并不是一个好主意。

谜题19：漂亮的火花（块注释符）

下面的程序用一个方法对字符进行了分类。这个程序会打印什么呢？如果你对String.indexOf(char)方法不太熟悉，那么我告诉你它将返回指定字符在字符串中第一次出现位置的索引，或者在字符串不包含该字符时返回-1：

```
public class Classifier {
    public static void main(String[] args) {
        System.out.println(
            classify('n') + classify('+') + classify('2'));
    }
    static String classify(char ch) {
        if ("0123456789".indexOf(ch) >= 0)
            return "NUMERAL ";
        if ("abcdefghijklmnopqrstuvwxyz".indexOf(ch) >= 0)
            return "LETTER ";
        /* (Operators not supported yet)
           if ("+-*/&|= ".indexof (ch)>= 0)
               return "OPERATOR ";
        */
        return "UNKNOWN";
    }
}
```

解惑19：漂亮的火花（块注释符）

如果你猜想该程序将打印LETTER UNKNOWN NUMBER，就掉进陷阱里面了。这个程序甚至不能通过编译。让我们再看一看相关的部分，这一次用粗体字突出注释部分：

```

if ("abcdefghijklmnopqrstuvwxyz".indexOf(ch) >= 0)
    return "LETTER ";
/* (Operators not supported yet)
if ("+-*/&|= ".indexOf(ch) >= 0)
    return "OPERATOR ";
*/
return "UNKNOWN";
}
}

```

可以看到，注释在包含了字符*/的字符串内部结束，使程序在语法上非法。将程序中的一部分注释出来的尝试之所以失败，是因为在注释中没有特殊处理字符串字面常量。

更一般地讲，注释内部的文本没有以任何方式进行特殊处理[JLS 3.7]。因此，**块注释不能嵌套**。请考虑下面的代码段：

```

/* Add the numbers from 1 to n */
int sum = 0;
for (int i = 1;i<= n; i++)
    sum += i;

```

现在假设要将该代码段注释成一个块注释，我们再次用粗体字突出整个注释：

```

/*
/* Add the numbers from 1 to n */
int sum = 0;
for (int i = 1;i <= n; i++)
    sum += i;
*/

```

可以看到，我们没有将最初的代码段注释掉。好在所产生的代码包含了一个语法错误，因此编译器将会告诉我们代码存在着问题。

你可能偶尔看到过这样的代码段，它被一个布尔表达式为常量false的if语句禁用了：

```
//Code commented out with an if statement-doesn't always work!
if (false) {
    /* Add the numbers from 1 to n */
    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum += i;
}
```

语言规范建议将这种方式作为一种条件编译技术[JLS 14.21]，但是它不适合用来注释代码。除非要被禁用的代码是一个合法的语句序列，否则就不要使用这项技术。

注释掉代码段的最好方式是使用单行的注释序列。大多数IDE工具都可以自动化这个过程：

```
//Code commented out with a sequence of single-line comments
//      /* Add the numbers from 1 to n */
//      int sum = 0;
//      for (int i = 1; i <= n; i++)
//          sum += i;
```

总之，块注释不能可靠地注释掉代码段，应该用单行的注释序列来代替。对语言设计者来说，应该注意可嵌套的块注释并不是一个好主意。他们强制编译器去解析块注释内部的文本，而由此引发的问题比它所解决的问题更多。

谜题20：我的类是什么

设计下面的程序来打印其类文件的名称。如果不熟悉类字面常量，那么我告诉你 `Me.class.getName()` 将返回 `Me` 类的完整名称，即 "`com.javapuzzlers.Me`"。那么，这个程序会打印什么呢？

```
package com.javapuzzlers;
public class Me {
    public static void main(String[] args){
        System.out.println(
            Me.class.getName().replaceAll(".", "/") + ".class");
    }
}
```

解惑20：我的类是什么

该程序看起来会获得它的类名 ("com.javapuzzlers.Me")，然后用"/"替换所有出现的字符串". ."，并在末尾追加字符串". class"。你可能会认为该程序将打印 com/javapuzzlers/Me.class，该程序正是从这个类文件中被加载的。如果你运行这个程序，就会发现它实际上打印的是//////////.class。到底怎么回事？难道我们是斜杠的受害者吗？

问题在于**String.replaceAll**接受了一个正则表达式作为它的第一个参数，而并非接受了一个字符序列字面常量。（正则表达式已经被添加到了Java平台的1.4版本中。）正则表达式". ."可以匹配任何单个的字符，因此，类名中的每一个字符都被替换成了一个斜杠，进而产生了我们看到的输出。

要想只匹配句点符号，在正则表达式中的句点必须在其前面添加一个反斜杠(\)进行转义。因为反斜杠字符在字面含义的字符串中具有特殊的含义——它标识转义字符序列的开始，因此反斜杠自身必须用另一个反斜杠来转义，这样就可以产生一个转义字符序列，它可以在字面含义的字符串中生成一个反斜杠。把这些合在一起，就可以使下面的程序打印所期望的com/javapuzzlers/Me.class：

```
package com.javapuzzlers;

public class Me {
    public static void main(String[] args){
        System.out.println(
            Me.class.getName().replaceAll("\\\\.", "/") + ".class");
    }
}
```

为了解决这类问题，5.0版本提供了新的静态方法**java.util.regex.Pattern.quote**。它接受一个字符串作为参数，并可以添加必需的转义字符，返回一个正则表达式字符串，该字符串将精确匹配输入的字符串。下面是使用该方法之后的程序：

```
package com.javapuzzlers;
import java.util.regex.Pattern;

public class Me {
    public static void main(String[] args){
```

```
System.out.println(Me.class.getName() .  
    replaceAll(Pattern.quote("."),"/") + ".class");  
}  
}
```

该程序的另一个问题是：其正确的行为是与平台相关的。并不是所有的文件系统都使用斜杠符号来分隔层次结构的文件名组成部分。想获取一个正在运行的平台上的有效文件名，你应该使用正确的平台相关的分隔符号来代替斜杠符号。这正是下一个谜题所要做的。

谜题21：我的类是什么？镜头2

下面的程序所要做的事情正是前一个谜题所做的事情，但是它没有假设斜杠符号就是分隔文件名组成部分的符号。相反，该程序使用的是java.io.File.separator，它被指定为一个公共的String域，包含了平台相关的文件名分隔符。这个程序会打印正确的、平台相关的类文件名吗？该程序是从这个类文件中被加载。

```
package com.javapuzzlers;  
import java.io.File;  
  
public class MeToo {  
    public static void main(String[] args){  
        System.out.println(MeToo.class.getName() .  
            replaceAll("\\.", File.separator) + ".class");  
    }  
}
```

解惑21：我的类是什么？镜头2

这个程序根据底层平台的不同会显示两种行为中的一种。如果文件分隔符是斜杠，就像在UNIX上一样，那么该程序将打印com/javapuzzlers/MeToo.class，这是正确的。但是，如果文件分隔符是反斜杠，就像在Windows上一样，那么该程序将打印像下面这样的内容：

```
Exception in thread "main"
StringIndexOutOfBoundsException: String index out of range: 1
    at java.lang.String.charAt(String.java:558)
    at java.util.regex.Matcher.appendReplacement(Mather.java:696)
    at java.util.regex.Matcher.replaceAll(Mather.java:806)
    at java.lang.String.replaceAll(String.java:2000)
    at com.javapuzzlers.MeToo.main(MeToo.java:6)
```

尽管这种行为是平台相关的，但是它并不是我们所期待的。在Windows上出了什么错呢？事实证明，String.replaceAll的第二个参数不是一个普通的字符串，而是一个替代字符串（replacement string），就像在java.util.regex规范中所定义的那样[Java-API]。在替代字符串中出现的反斜杠会把紧随其后的字符进行转义，从而导致其被按字面含义而处理了。当你在Windows上运行该程序时，替代字符串是单独的一个反斜杠，它是无效的。不可否认，抛出的异常应该提供更多有用的信息。

那么应该怎样解决此问题呢？5.0版本提供了不是一个而是两个新的方法来解决它。第一个方法是java.util.regex.Matcher.quoteReplacement，它将字符串转换成相应的替代字符串。下面展示了如何使用这个方法来修正该程序：

```
System.out.println(MeToo.class.getName().replaceAll(
    "\\", Matcher.quoteReplacement(File.separator))
    + ".class");
```

引入到5.0版本中的第二个方法提供了更好的解决方案。该方法就是String.replace(CharSequence, CharSequence)，它做的事情和String.replaceAll相同，但是它将模式和替代物都当作字符串字面常量处理。下面展示了如何使用这个方法来修正该程序：

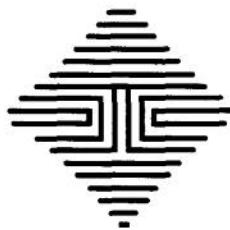
```
System.out.println(MeToo.class.getName() .
    replace (".",File.separator) + ".class");
```

但是如果你使用的是较早版本的Java该怎么办？很遗憾，没有任何捷径能够生成替代字符串。完全不使用正则表达式，而使用String.replace(char,char)也许更容易：

```
System.out.println(MeToo.class.getName() .
    replace('.'. File.separatorChar) + ".class");
```

本谜题和前一个谜题的主要教训是：**在使用不熟悉的类库方法时一定要格外小心。**当你心存疑虑时，就要求助于Javadoc。另外，正则表达式是很棘手的：它所引发的问题倾向于在运行时刻而不是在编译时刻暴露出来。

对API的设计者来说，使用方法具名的模式来区分行为相差极大的方法是很重要的。Java的String类就没有很好地遵从这一原则。对许多程序员来说，记住哪些字符串替代方法使用的是字符串字面常量，以及哪些使用的是正则表达式或替代字符串，并不是一件容易事。



谜题22：URL的愚弄

本谜题利用了Java编程语言中一个鲜为人知的特性。请考虑下面的程序将会做什么？

```
public class BrowserTest {
    public static void main(String[] args) {
        System.out.print("iexplore:");
        http://www.google.com;
        System.out.println(":maximize");
    }
}
```

解惑22：URL的愚弄

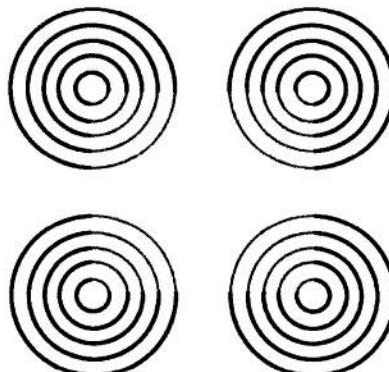
这是一个有点诡异的问题。该程序不会做任何特殊的事情，而是直接打印*iexplore::maximize*。在程序中间出现的URL是一个语句标号（statement label）[JLS 14.7]后面跟着尾注释（end-of-line comment）[JLS 3.7]。在Java中很少需要标号，这多亏了Java没有goto语句。在本谜题中所引用的“Java编程语言中鲜为人知的特性”实际上就是可以在任何语句前面放置标号。这个程序标注了一个表达式语句，它是合法的，但是没什么用处。

它的价值所在，就是提醒你，如果确实希望使用标号，那么这将是一种更合理的格式化程序方式：

```
public class BrowserTest {  
    public static void main(String[] args) {  
        System.out.print("iexplore:";  
  
        http://www.google.com;  
        System.out.println(":maximize");  
    }  
}
```

这就是说，我们没有任何可能的理由去使用与程序没有任何关系的标号和注释。

本谜题的教训是：令人误解的注释和无关的代码会引起混乱。**仔细地写注释，并让它们跟上时代；去除那些已遭废弃的代码。**另外，如果某些东西看起来过于奇怪，以至于不像对的，那么它极有可能就是错的。



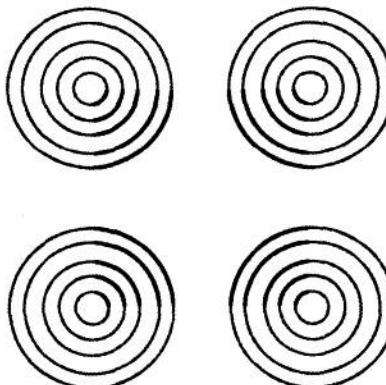
谜题23：不劳无获

下面的程序将打印一个单词，其首字母由一个随机数生成器选择。请描述该程序的行为：

```
import java.util.Random;

public class Rhymes {
    private static Random rnd = new Random();

    public static void main(String[] args) {
        StringBuffer word = null;
        switch(rnd.nextInt(2)) {
            case 1: word = new StringBuffer('P');
            case 2: word = new StringBuffer('G');
            default: word = new StringBuffer('M');
        }
        word.append('a');
        word.append('i');
        word.append('n');
        System.out.println(word);
    }
}
```



解惑23：不劳无获

乍一看，这个程序可能会在一次又一次的运行中，以相等的概率打印Pain, Gain或Main。看起来该程序会根据随机数生成器所选取的值来选择单词的首字母：0选M，1选P，2选G。谜题的题目也许已经提供了线索，它实际上既不会打印Pain，也不会打印Gain。也许更令人吃惊的是，它也不会打印Main，并且它的行为不会在一次又一次的运行中发生变化，它总是打印ain。

三个bug一起引发了这种行为。你完全没有发现它们吗？第一个bug是所选取的随机数使得switch语句只能到达其三种情况中的两种。Random.nextInt(int)的规范描述道：“返回一个伪随机的、均等地分布在从0（包含）到指定的数值（不包含）之间的一个int数值” [Java-API]。这意味着表达式rnd.nextInt(2)可能的取值只有0和1，Switch语句永远无法到达case 2分支，这表示程序永远不会打印Gain。nextInt的参数应该是3而不是2。

这是一个相当常见的问题源，众所周知的“栅栏柱错误（fencepost error）”。这个名字来源于对下面这个问题最常见的但却是错误的答案，如果你要建造一个100英尺长的栅栏，其栅栏柱间隔为10英尺，那么你需要多少根栅栏柱呢？11根或9根都是正确答案，这取决于是否要在栅栏的两端树立栅栏柱，但是10根却是错误的。要当心栅栏柱错误，每当你在处理长度、范围或模数的时候，都应仔细确定其端点是否应该包括在内，并且确保你的代码行为与其相对应。

第二个bug是在不同的情况（case）中没有任何break语句。不论switch表达式为何值，该程序都将执行其相对应的case以及所有后续的case[JLS 14.11]。因此，尽管每一个case都对变量word赋了值，但是总是最后的赋值胜出，覆盖了前面的赋值。最后的赋值将总是最后一种情况（default），即new StringBuffer('M')。这表明该程序将总是打印Main，而从来不打印Pain或Gain。

在switch的各种情况中缺少break语句是非常常见的错误。从5.0版本起，javac提供了-Xlint:fallthrough标志，当你忘记在一个case与下一个case之间添加break语句时，它可以生成警告信息。不要从一个非空的case向下进入另一个case。这是一种拙劣的风格，因为它并不常用，因此会误导读者。十次中有九次它都会包含错误。如果Java不是模仿C建模的，那么它倒是有可能不需要break。对语言设计者的教训是：应该考虑提供一个结构化的switch语句。

最后一个，也是最微妙的一个bug是，表达式new StringBuffer('M')可能没有做你希望它做的事情。你可能对StringBuffer(char)构造器并不熟悉，这很容易解

释：它压根就不存在。StringBuffer有一个无参数的构造器，一个接受一个String作为字符串缓冲区初始内容的构造器，以及一个接受一个int作为缓冲区初始容量的构造器。在本例中，编译器会选择接受int的构造器，通过拓宽原生类型转换把字符数值'M'转换为int数值77[JLS 5.1.2]。换句话说，new StringBuffer ('M') 返回的是一个具有初始容量77的空字符串缓冲区。该程序余下的部分将字符a、i和n添加到了这个空字符串缓冲区中，并打印该字符串缓冲区的内容，它总是ain。

为了避免这类问题，不管在什么时候，都要尽可能使用熟悉的惯用法和API。如果必须使用不熟悉的API，那么请仔细阅读其文档。在本例中，程序应该使用常用的接受一个String的StringBuffer构造器。

下面是该程序改正了这三个bug之后的正确版本，它将以均等的概率打印Pain、Gain和Main：

```
import java.util.Random;

public class Rhymes {
    private static Random rnd = new Random();

    public static void main(String[] args) {
        StringBuffer word = null;
        switch(rnd.nextInt(3)) {
            case 1:
                word = new StringBuffer("P");
                break;
            case 2:
                word = new StringBuffer("G");
                break;
            default:
                word = new StringBuffer("M");
                break;
        }
        word.append('a');
        word.append('i');
        word.append('n');
        System.out.println(word);
    }
}
```

尽管这个程序改正了所有的bug，它还是显得过于冗长。下面是一个更优雅的版本：

```
import java.util.Random;

public class Rhymes {
    private static Random rnd = new Random();
    public static void main(String args[]) {
        System.out.println("PGM".charAt(rnd.nextInt(3)) + "ain");
    }
}
```

下面是一个更好的版本。尽管它稍微长了一点，但是它更加通用。它不依赖于所有可能的输出只是首字母不同的这个事实：

```
import java.util.Random;

public class Rhymes {
    public static void main(String args[]) {
        String a[] = {"Main", "Pain", "Gain"};
        System.out.println(randomElement(a));
    }

    private static Random rnd = new Random();
    private static String randomElement(String[] a){
        return a[rnd.nextInt(a.length)];
    }
}
```

总结一下：首先，要当心栅栏柱错误。其次，牢记在switch语句的每一个case中都放置一条break语句。第三，要使用常用的惯用法和API，并且当你在离开老路子的时候，一定要参考相关的文档。第四，char不是String，而是更像int。最后，要提防各种诡异的谜题。

循环之谜

本章所描述的所有谜题都与循环有关。

谜题24：尽情享受每一个字节

下面的程序循环遍历byte数值，以查找某个特定值。这个程序会打印什么呢？

```
public class BigDelight {  
    public static void main(String[] args) {  
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {  
            if (b == 0x90)  
                System.out.print("Joy!");  
        }  
    }  
}
```

解惑24：尽情享受每一个字节

这个循环在除了Byte.MAX_VALUE之外所有的byte数值中进行迭代，以查找0x90。这个数值适合用byte表示，并且不等于Byte.MAX_VALUE，因此你可能会认为这个循环在该迭代会找到它一次，并将打印Joy!。但是，所见为虚。如果你运行该程序，就会发现它没有打印任何东西。怎么回事？

简单地说，0x90是一个int常量，它超出了byte数值的范围。这与直觉是相悖的，因为0x90是一个两位的十六进制字面常量，每一个十六进制位都占据4个比特的位置，所以整个数值也只占据8个比特，即1个byte。问题在于byte是有符号类型。常量0x90是一个正的最高位被置位的8位int数值。合法的byte数值是从-128到+127，但是int常量0x90等于+144。

一个byte与一个int进行的比较是一个混合类型比较。如果你把byte数值想像为苹果，把int数值想像为桔子，那么该程序就是在拿苹果与桔子比较。请考虑表达式`((byte)0x90 == 0x90)`，尽管外表看起来是成立的，但是它却等于false。为了比较byte数值`(byte)0x90`和int数值`0x90`，Java通过拓宽原生类型转换将byte提升为int[JLS 5.1.2]，然后比较这两个int数值。因为byte是一个有符号类型，所以这个转换执行的是符号扩展，将负的byte数值提升为了在数字上相等的int数值。在本例中，该转换将`(byte)0x90`提升为int数值-112，它不等于int数值`0x90`，即+144。

由于系统总是强制地将一个操作数提升为与另一个操作数相匹配的类型，所以混合类型比较总是容易引起混淆。这种转换是不可视的，而且可能不会产生你所期望的结果。有若干种方法可以避免混合类型比较。我们继续有关水果的比喻，可以选择拿苹果与苹果比较，或者是拿桔子与桔子比较。可以将int转型为byte，之后就可以拿一个byte与另一个byte进行比较了：

```
if (b == (byte)0x90)
    System.out.println("Joy!");
```

或者，可以用一个屏蔽码来消除符号扩展的影响，从而将byte转型为int，之后就可以拿一个int与另一个int进行比较了：

```
if ((b & 0xff) == 0x90)
    System.out.println("Joy!");
```

上面的两个解决方案都可以正常运行，但是避免这类问题的最佳方法还是将常量值移到循环的外面，并在一个常量声明中定义它。下面是我们对此作出的第一个尝试：

```
public class BigDelight{
    private static final byte TARGET = 0x90; //Broken!
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)
            if (b == TARGET)
                System.out.print("Joy!");
    }
}
```

遗憾的是，它根本就通不过编译。常量声明有问题，编译器会告诉你问题所在：0x90对于byte类型来说不是一个有效的数值。如果你像下面这样修正该声明，那么程序将运行得非常好：

```
private static final byte TARGET = (byte)0x90;
```

总之，要避免混合类型比较，因为它们内在地容易引起混乱（谜题5）。为了帮助实现这个目标，请使用声明的常量替代“魔幻数字”。你已经了解了这确实是一个好主意：它说明了常量的含义，集中了常量的定义，并且根除了重复的定义。现在你知道它还可以强制为每一个常量赋予适合其用途的类型，从而消除了产生混合类型比较的一种根源。

对语言设计者的教训是byte数值的符号扩展是产生bug和混乱的一种常见根源。而抵消符号扩展后果所需的屏蔽机制会使得程序混乱无序，从而降低了程序的可读性。因此，byte类型应该是无符号的。还可以考虑为所有的原生类型提供定义字面常量的机制，这可以减少对易于产生错误的类型转换的需求（谜题27）。

谜题25：无情的增量操作

下面的程序对一个变量重复地进行增量操作，然后打印它的值。那么这个值是什么呢？

```
public class Increment {
    public static void main(String[] args) {
        int j = 0;
        for (int i = 0; i < 100; i++)
            j = j++;
        System.out.println(j);
    }
}
```

解惑25：无情的增量操作

乍一看，这个程序可能会打印100。毕竟，它对j做了100次增量操作。可能会令你感到有些震惊，它打印的不是100而是0。所有的增量操作都无影无踪了，为什么？

就像本谜题的题目所暗示的，问题出在执行增量操作的语句上：

```
j = j++;
```

大概该语句的编写者是想让它执行对j的值加1的操作，也就是表达式`j++`所做的操作。遗憾的是，编写者大咧咧地将这个表达式的值又赋回给了j。当`++`操作符被置于一个变量值之后时，其作用就是一个后缀增量操作符（postfix increment operator）[JLS 15.14.2]：表达式`j++`的值等于j在执行增量操作之前的初始值。因此，前面提到的赋值语句首先保存j的值，然后将j设置为其值加1，最后将j复位到它的初始值。换句话说，这个赋值操作等价于下面的语句序列：

```
int tmp = j;
j = j + 1;
j = tmp;
```

程序重复该过程100次之后，j的值还是等于它在循环开始之前的值，即0。

修正该程序非常简单，只需从循环中移除无关的赋值操作，只留下：

```
for (int i = 0; i < 100; i++)
    j++;
```

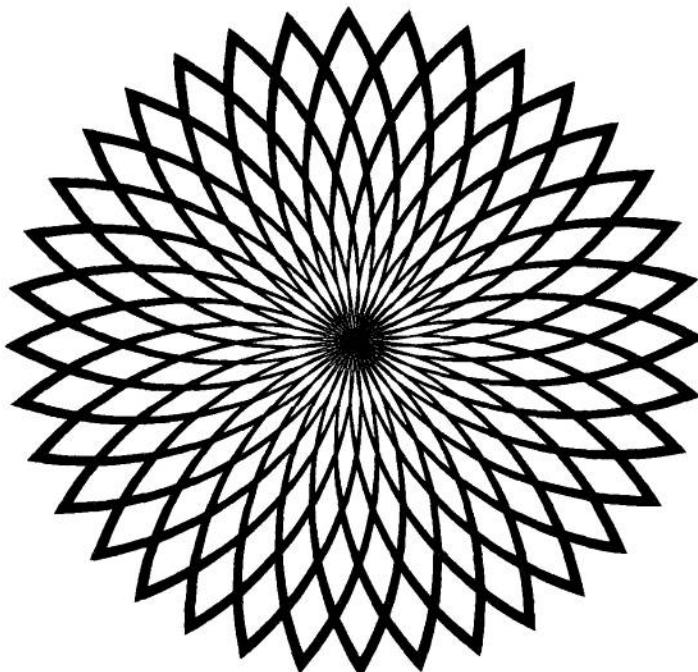
经过这样的修改，程序就可以打印我们所期望的100了。

这与谜题7中的教训相同：不要在单个的表达式中对相同的变量赋值超过一次。对相同的变量进行多次赋值的表达式会产生混淆，并且很少能够产生你希望的行为。

谜题26：在循环中

下面的程序计算了一个循环的迭代次数，并且在循环终止时打印这个计数值。那么，它打印的是什么呢？

```
public class InTheLoop {  
    public static final int END = Integer.MAX_VALUE;  
    public static final int START = END - 100;  
  
    public static void main(String[] args) {  
        int count = 0;  
        for (int i = START; i <= END; i++)  
            count++;  
        System.out.println(count);  
    }  
}
```



解惑26：在循环中

如果没有非常仔细地查看这个程序，你可能会认为它将打印100，因为END比START大100。稍微仔细一点，你可能会发现该程序没有使用典型的循环惯用法。大多数的循环会在循环索引小于终止值时持续运行，而这个循环则是在循环索引小于或等于终止值时持续运行。所以它会打印101，对吗？嗯，根本不对。如果你运行该程序，就会发现它压根就什么都没有打印。更糟的是，它会持续运行直到撤销为止。它没有机会打印count，因为在打印语句之前插入的是一个无限循环。

问题在于这个循环会在循环索引(*i*)小于或等于Integer.MAX_VALUE时持续运行，但是所有的int变量都是小于或等于Integer.MAX_VALUE的。因为它被定义为所有int数值中的最大值。当*i*达到Integer.MAX_VALUE，并且再次执行增量操作时，它就又绕回到了Integer.MIN_VALUE。

如果需要的循环会迭代到int数值的边界附近，最好使用一个long变量作为循环索引。只需将循环索引的类型从int改变为long就可以解决该问题，从而使程序打印我们所期望的101：

```
for (long i = START; i <= END; i++)
```

更一般地讲，这里的教训就是int不能表示所有的整数。无论你在何时使用了一个整数类型，都要意识到其边界条件。如果其数值下溢或是上溢了，会怎么样呢？所以通常最好是使用一个取值范围更大的类型。（整数类型包括byte、char、short、int和long。）

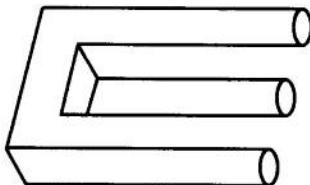
不使用long类型的循环索引变量也可以解决该问题，但是它看起来不那么漂亮：

```
int i = START;
do {
    count++;
} while (i++ != END);
```

如果清晰性和简洁性占据了极其重要的地位，那么在这种情况下使用一个long类型的循环索引几乎总是最佳方案，但是有一个例外：如果你在所有的（或者几乎所有的）int数值上迭代，那么使用int类型的循环索引的速度大约可以提高一倍。下面是将`long`函数作用于所有40亿个int数值上的惯用法：

```
//Apply the function f to all four billion int values
int i = Integer.MIN_VALUE;
do {
    f(i);
}while (i++ != Integer.MAX_VALUE);
```

该谜题对语言设计者的教训与谜题3相同：可能真的值得考虑，应该对那些在产生溢出时不会不抛出异常的算术运算提供支持。同时，可能还值得去考虑，应该对那些在整数值范围之上进行迭代的循环进行特殊设计，就像许多其他语言所做的那样。



谜题27：变幻莫测的i值

与谜题26中的程序一样，下面的程序也包含了一个记录在终止前有多少次迭代的循环。与那个程序不同的是，这个程序使用的是左移操作符（`<<`）。你的任务仍旧是要指出这个程序将打印什么。当你阅读这个程序时，请记住Java使用的是基于2的补码的二进制算术运算，因此-1在任何有符号的整数类型中（`byte`、`short`、`int`或`long`）的表示都是所有的位被置位：

```
public class Shifty {
    public static void main(String[] args) {
        int i = 0;
        while (-1 << i != 0)
            i++;
        System.out.println(i);
    }
}
```

解惑27：变幻莫测的i值

常量-1是所有32位都被置位的int数值(0xffffffff)。左移操作符将0移入到由移位所空出的右边最低位，因此表达式($-1 \ll i$)将最右边的*i*位设置为0，并保持其余的 $32-i$ 位为1。很明显，这个循环将完成32次迭代，因为 $-1 \ll i$ 对任何小于32的*i*来说都不等于0。你可能期望在*i*等于32时终止条件测试返回false，从而使程序打印32，但是它打印的并不是32。实际上，它不会打印任何东西，而是进入了一个无限循环。

问题在于($-1 \ll 32$)等于-1而不是0，因为移位操作符只使用其右操作数的低5位作为移位长度。或者是低6位，如果其左操作数是一个long类型数值[JLS 15.19]。这条规则作用于全部的三个移位操作符：`<<`、`>>`和`>>>`。移位长度总是介于0到31之间，如果左操作数是long类型，则介于0到63之间。这个长度是对32取余的，如果左操作数是long类型的，则对64取余。如果试图对一个int数值移位32位，或者是对一个long数值移位64位，都只能返回这个数值本身。没有任何移位长度可以让一个int数值丢弃其所有的32位，或者是让一个long数值丢弃其所有的64位。

幸运的是，有一个非常容易的方法能够修正该问题。我们不是让-1重复地移位不同的移位长度，而是将前一次移位操作的结果保存起来，并且让它在每一次迭代时都向左再移1位。下面这个版本的程序就可以打印我们所期望的32：

```
public class Shifty {
    public static void main(String[] args) {
        int distance = 0;
        for (int val = -1; val != 0; val <<= 1)
            distance++;
        System.out.println(distance);
    }
}
```

这个修正过的程序说明了一条普遍的原则：如果可能的话，移位长度应该是常量。如果移位长度紧盯着你不放，那么你让其值超过31，或者如果左操作数是long类型的，让其值超过63的可能性就会大大降低。当然，并不总是可以使用常量的移位长度。当必须使用一个非常量的移位长度时，请确保你的程序可以应付这种容易产生问题的情况，或者根本不会碰到这种情况。

前面提到的移位操作符的行为还有另外一个令人震惊的结果。很多程序员都希望具有负移位长度的右移操作符可以起到左移操作符的作用，反之亦然。但是情况并非如此。

右移操作符总是起到右移的作用，而左移操作符也总是起到左移的作用。负的移位长度通过只保留低5位而去除其他位的方式被转换成了正的移位长度——如果左操作数是long类型的，则保留低6位。因此，如果要将一个int数值左移，其移位长度为-1，那么移位的效果是它被左移了31位。

总之，移位长度是对32取余的，或者如果左操作数是long类型的，则对64取余。因此，使用任何移位操作符和移位长度，都不可能将一个数值的所有位全部移走。同时，我们也不可能用右移操作符来执行左移操作，反之亦然。如果可能的话，请使用常量的移位长度，如果移位长度不能设为常量，那么就要千万小心。

语言设计者可能应该考虑将移位长度限制在从0到以位为单位的类型长度的范围内，并且修改移位长度为类型长度时的语义，让其返回0。尽管这可以避免在本谜题中所展示的混乱情况，但是它可能会带来负面的执行结果，因为Java的移位操作符的语义正是许多处理器上的移位指令语义。

谜题28：循环者

下面的谜题以及随后的五个谜题对你来说是扭转了局面。它们不是向你展示某些代码，然后询问你这些代码将做些什么，它们要让你去写代码，但是数量会很少。这些谜题被称为“循环者”。你眼前会展示一个循环，它看起来应该很快就终止，而你的任务就是写一个变量声明，将它正好置于该循环之前时，使得该循环无限循环下去。例如，考虑下面的for循环：

```
for (int i = start; i <= start + 1; i++) {  
}
```

看起来它好像应该只迭代两次，但是通过利用在谜题26中所展示的溢出行为，可以使它无限循环下去。下面的的声明就采用了这项技巧：

```
int start = Integer.MAX_VALUE - 1;
```

现在该轮到你了。什么样的声明能够让下面的循环变成一个无限循环？

```
While (i == i + 1) {  
}
```

解惑28：循环者

仔细查看这个while循环，它真的好像应该立即终止。一个数字永远不会等于它自己加1，对吗？嗯，如果这个数字是无穷大的，又会怎样呢？Java强制要求使用IEEE 754浮点数算术运算[IEEE 754]，它可以让你用一个double或float来表示无穷大。正如我们在学校里学到的，无穷大加1还是无穷大。如果i在循环开始之前被初始化为无穷大，那么终止条件测试($i == i + 1$)就会被计算为true，从而使循环永远不会终止。

可以用任何被计算为无穷大的浮点算术表达式来初始化i，例如：

```
double i = 1.0 / 0.0;
```

不过，最好是能够利用标准类库提供的常量：

```
double i = Double.POSITIVE_INFINITY;
```

事实上，不必将i初始化为无穷大以确保循环永远执行。任何足够大的浮点数都可以实现这一目的，例如：

```
double i = 1.0e40;
```

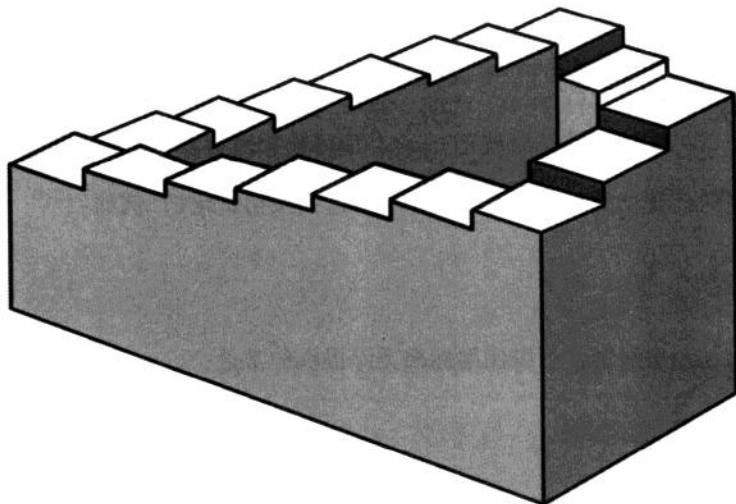
这样做之所以起作用，是因为一个浮点数值越大，它和其后继数值之间的间隔就越大。浮点数的这种分布是用固定数量的有效位来表示它们的必然结果。对一个足够大的浮点数加1不会改变它的值，因为1不足以“填补它与其后继者之间的空隙”。

浮点数操作返回的是最接近其精确数学结果的浮点数值。一旦毗邻的浮点数值之间的距离大于2，那么其中的一个浮点数值加1将不会产生任何效果，因为其结果没有达到两个数值之间的一半。对于float类型，加1不会产生任何效果的最小级数是 2^5 ，即33 554 432；而对于double类型，最小级数是 2^{16} ，大约是 1.8×10^{16} 。

毗邻的浮点数值之间的距离被称为一个ulp，它是最小单位（unit in the last place）的首字母缩写词。在5.0版中，引入了Math.ulp方法来计算float或double数值的ulp。

总之，用一个double或float数值来表示无穷大是可以的。大多数人在第一次听到这句话时，多少都会有一点吃惊，可能是因为无法用任何整数类型来表示无穷大的原

因。第二点，将一个很小的浮点数加到一个很大的浮点数上时，将不会改变大浮点数的值。这过于违背直觉了，因为对实际的数字来说这是不成立的。我们应该记住二进制浮点算术只是对实际算术的一种近似。



谜题29：循环者的新娘

请提供一个对*i*的声明，将下面的循环转变为无限循环：

```
while (i != i) {  
}
```

解惑29：循环者的新娘

这个循环可能比前一个更令人困惑。不管在它前面作何种声明，它看起来确实应该立即终止。一个数字总是等于它自己，对吧？

对，但是IEEE 754浮点算术保留了一个特殊的值用来表示一个不是数字的数量[IEEE-754]。这个值就是NaN（“Not a Number（不是一个数字）”的缩写），对于所有没有良好的数字定义的浮点计算，例如 $0.0/0.0$ ，其值都是它。规范中描述道，**NaN不等于任何浮点数值，包括它自身在内**[JLS 15.21.1]。因此，如果*i*在循环开始之前被初始化为NaN，那么终止条件测试(*i != i*)的计算结果就是true，循环就永远不会终止。很奇怪但却是事实。

可以用任何计算结果为NaN的浮点算术表达式来初始化*i*，例如：

```
double i = 0.0 / 0.0;
```

同样，为了表达清晰，可以使用标准类库提供的常量：

```
double i = Double.NaN;
```

NaN还有其他的惊人之处。任何浮点操作，只要它的一个或多个操作数为NaN，那么其结果为NaN。这条规则是非常合理的，但是它却具有奇怪的结果。例如，下面的程序将打印false：

```
class Test {
    public static void main(String[] args) {
        double i = 0.0 / 0.0;
        System.out.println(i - i == 0);
    }
}
```

这条计算NaN的规则所基于的原理是：一旦一个计算产生了NaN，它就被损坏了，没有任何更进一步的计算可以修复这样的损坏。NaN值有意使受损的计算继续执行下去，直到到达方便处理这种情况的地方为止。

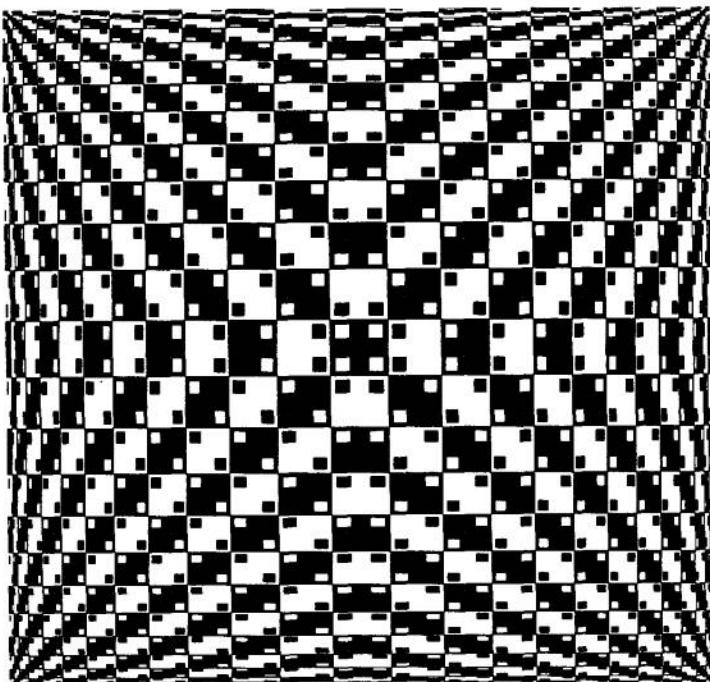
总之，float和double类型都有一个特殊的NaN值，用来表示不是数字的数量。对于涉及NaN值的计算，其规则简单且明智，但是这些规则的结果可能是违背直觉的。

谜题30：循环者的爱子

请提供一个对*i*的声明，将下面的循环转变为一个无限循环：

```
while (i != i + 0) {  
}
```

与前一个谜题不同，你必须在答案中不使用浮点数。换句话说，你不能把*i*声明为double或float类型。



解惑30：循环者的爱子

与前一个谜题一样，这个谜题初看起来是不可能实现的。毕竟，一个数字总是等于它自身加上0，你被禁止使用浮点数，因此不能使用NaN。而在整数类型中没有NaN的等价物。那么，你能给出什么呢？

我们必然可以得出这样的结论，即*i*的类型必须是非数值类型的，并且其中存在着解惑方案。惟一的+操作符有定义的非数值类型就是String。+操作符被重载了：对于String类型，它执行的不是加法而是字符串连接。如果在连接中的某个操作数具有非String的类型，那么这个操作数就会在连接之前转换成字符串[JLS 15.18.1]。

事实上，*i*可以被初始化为任何值，只要它是String类型即可，例如：

```
String i = "Buy seventeen copies of Effective Java!";
```

*int*类型的数值0被转换成String类型的数值"0"并且被追加到感叹号之后，所产生的字符串在用equals方法计算时就不等于最初的字符串了，这样它们在使用==操作符进行计算时，当然就不是相等的。因此，计算布尔表达式(*i* != *i* + 0)得到的值就是true，循环也就永远不会终止了。

总之，操作符重载是很容易令人误解的。在本谜题中的加号看起来是表示一个加法，但是通过为变量*i*选择合适的类型，即String，我们让它执行了字符串连接操作。甚至因为变量被命名为*i*，使得本谜题更加容易令人误解，因为*i*通常被当作整型变量名而保留。对于程序的可读性来说，好的变量名、方法名和类名至少与好的注释同等重要。

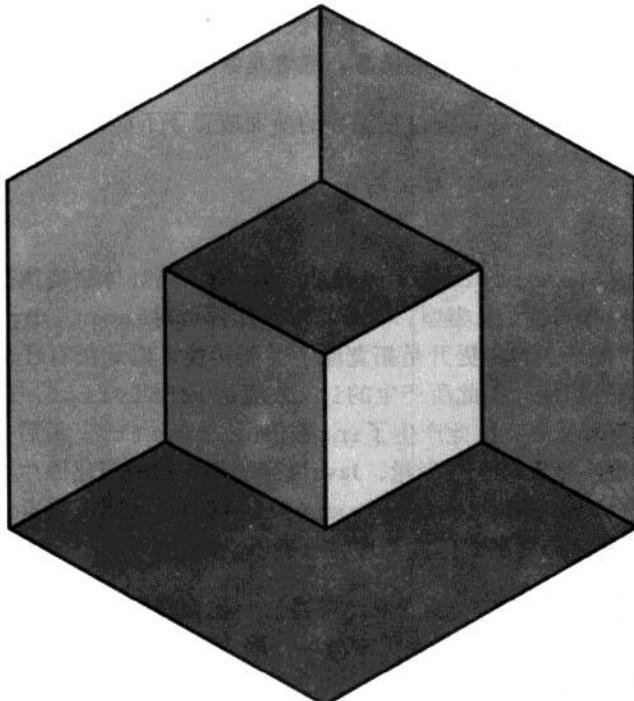
对语言设计者的教训与谜题11和13中的教训相同。操作符重载很容易引起混乱，也许+操作符就不应该被重载来进行字符串连接操作。有充分的理由证明提供一个字符串连接操作符是多么必要，但是它不应该是+。

谜题31：循环者的鬼魂

请提供一个对 i 的声明，将下面的循环转变为一个无限循环：

```
while (i != 0) {  
    i >>>= 1;  
}
```

回想一下，`>>>=`是对应于无符号右移操作符的赋值操作符。0被从左移入到移位操作空出的位上，即使被移位的是负数也是如此。



解惑31：循环者的鬼魂

这个循环比前面三个循环要稍微复杂一点，因为其循环体非空。在其循环体中，*i*的值由它右移一位之后的值所替代。为了使移位合法，*i*必须是一个整数类型（byte、char、short、int或long）。无符号右移操作符把0从左边移入，因此看起来这个循环执行迭代的次数与最大的整数类型所占据的位数相同，即64次。如果在循环的前面放置如下的声明，那么这确实就是将要发生的事情：

```
long i = -1; // -1L has all 64 bits set
```

怎样才能将它转变为一个无限循环呢？解决本谜题的关键在于`>>>=`是一个复合赋值操作符。（复合赋值操作符包括`*=`、`/=`、`%=`、`+=`、`-=`、`<<=`、`>>=`、`>>>=`、`&=`、`^=`和`!=`。）有关复合赋值操作符的一个不幸的事实是，它们可能会自动地执行窄化原生类型转换[JLS 15.26.2]，这种转换把一种数字类型转换成了另一种更缺乏表达能力的数字类型。窄化原生类型转换可能会丢失级数的信息，或者是数值的精度[JLS 5.1.3]。

让我们说明得更具体一些，假设在循环的前面放置了下面的声明：

```
short i = -1;
```

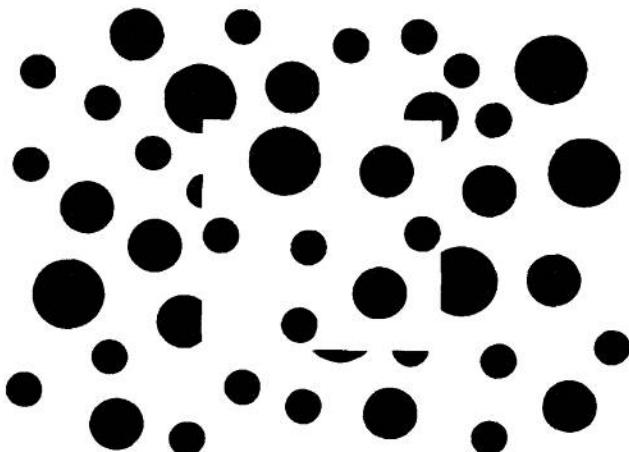
因为*i*的初始值(`(short)0xffff`)是非0的，所以循环体会被执行。在执行移位操作时，第一步是将*i*提升为int类型。所有算术操作都会对short、byte和char类型的操作数执行这样的提升。这种提升是拓宽原生类型转换，因此没有任何信息会丢失。这种提升执行的是符号扩展，因此所产生的int数值是`0xffffffff`。然后，这个数值右移1位，但不使用符号扩展，因此产生了int数值`0x7fffffff`。最后，这个数值被存回*i*中。为了将int数值存入short变量，Java执行的是可怕的窄化原生类型转换，它直接将高16位截掉。这样就只剩下`(short)0xffff`，我们又回到了起点。循环的第二次以及后续的迭代行为都是一样的，因此循环将永远不会终止。

如果将*i*声明为short或byte变量，并且初始化为任何负数，那么这种行为也会发生。如果声明*i*为char，将无法得到无限循环，因为char是无符号的，所以在移位之前的拓宽原生类型转换不会执行符号扩展。

总之，不要在short、byte或char类型的变量之上使用复合赋值操作符。因为这样的表达式执行的是混合类型算术运算，容易造成混乱。更糟的是，它们执行隐式的窄

化类型转换，可能丢失信息，其结果是灾难性的。

对语言设计者的教训是语言不应该自动地执行窄化类型转换。还有一点值得争论的是，Java是否应该禁止在short、byte和char变量上使用复合赋值操作符。



谜题32：循环者的诅咒

提供对i和j的声明，将下面的循环转变为无限循环：

```
while (i <= j && j <= i && i != j) {  
}
```

解惑32：循环者的诅咒

噢，不，不要再给我看起来不可能的循环了！如果 $i \leq j$ 并且 $j \leq i$ ， i 不是肯定等于 j 吗？这一属性对实数肯定有效。事实上，它是如此重要，以至于它有这样的定义：实数上的 \leq 关系是反对称的。Java 的 \leq 操作符在 5.0 版之前是反对称的，但是从 5.0 版之后就不再是了。

直到 5.0 版之前，Java 的数字比较操作符（ $<$ 、 \leq 、 $>$ 和 \geq ）要求它们的两个操作数都是原始数字类型的（`byte`、`char`、`short`、`int`、`long`、`float` 和 `double`）[JLS 15.20.1]。但是在 5.0 版中，规范作出了修改，新规范描述道：每一个操作数的类型必须可以转换成原始数字类型[JLS 15.20.1, 5.1.8]。问题难就难在这里了。

在 5.0 版中，自动包装和自动反包装被添加到了 Java 语言中。如果你对它们并不了解，请查看：<http://java.sun.com/j2se/5.0/docs/guide/language/autoboxing.html> [Boxing]。 \leq 操作符在原始数字类型集上仍然是反对称的，但是现在它还应用在被包装的数字类型上（被包装的数字类型有：`Byte`、`Character`、`Short`、`Integer`、`Long`、`Float` 和 `Double`）。 \leq 操作符在这些类型的操作数上不是反对称的，因为 Java 的判等操作符（`==` 和 `!=`）在作用于对象引用时，执行的是引用 ID 的比较，而不是值的比较。

让我们说明得更具体一些，下面的声明使得表达式($i \leq j \&& j \leq i \&& i != j$)的值为 `true`，从而将这个循环变成了一个无限循环：

```
Integer i = new Integer(0);
Integer j = new Integer(0);
```

前两个子表达式 ($i \leq j$ 和 $j \leq i$) 在 i 和 j 上执行解包转换[JLS 5.1.8]，并且在数字上比较所产生的 `int` 数值。 i 和 j 都表示 0，所以这两个子表达式都被计算为 `true`。第三个子表达式 ($i != j$) 在对象引用 i 和 j 上执行标识比较。因为它们都初始化为新的 `Integer` 实例，两变量引用不同的对象。因此，第三个子表达式同样也被计算为 `true`，循环也就永远地环绕下去了。

你可能会感到奇怪，为什么语言规范没有修改为：当判等操作符作用于被包装的数字类型时，它们执行值比较。答案很简单：兼容性。当一种语言被广泛使用之后，以违反现有规范的方式去改变现有程序的行为是让人无法接受的。下面的程序过去总是保证打印 `false`，因此它必须继续保持。

```
public class ReferenceComparison {
    public static void main(String[] args) {
        System.out.println(new Integer(0) == new Integer(0));
    }
}
```

在两个操作数中只有一个被包装的数字类型，而另一个是原生类型时，判等操作符执行的确实是数值比较。因为这在5.0版之前是非法的，所有在这里没有任何兼容性的问题。更具体一些，下面的程序在1.4版中是非法的，而在5.0版中将打印true：

```
public class ValueComparison {
    public static void main(String[] args) {
        System.out.println(new Integer(0) == 0);
    }
}
```

总之，当两个操作数都是被包装的数字类型时，数值比较操作符和判等操作符的行为存在着根本的差异：数值比较操作符执行的是值比较，而判等操作符执行的是引用标识的比较。

对语言设计者来说，如果判等操作符一直执行的都是数值比较（谜题13），那么生活可能就要简单得多、快乐得多。也许真正的教训应该是：语言设计者应该拥有高质量的水晶球，以预测语言的未来，并且做出相应的设计决策。更严肃地讲，语言设计者应该考虑语言可能会如何演化，并且应该努力去最小化在演化之路上的各种制约影响。

谜题33：循环者遇到了狼人

请提供一个对i的声明，将下面的循环转变为无限循环。这个循环不需要使用任何5.0版的特性：

```
while (i != 0 && i == -i) {  
}
```

解惑33：循环者遇到了狼人

这仍然是一个循环。在布尔表达式($i != 0 \&& i == -i$)中，一元减号操作符作用于 i ，意味着它的类型必须是数字的：一元减号操作符作用于一个非数字类型操作数是非法的。因此，我们要寻找一个非0的数字类型数值，它等于自己的负值。 NaN 不能满足这个属性，因为它不等于任何数值，因此， i 必须表示一个实际的数字。确定没有任何数字满足这样的属性吗？

嗯，没有任何实数具有这种属性，但是没有任何一种Java数字类型能够对实数进行完美建模。浮点数值是用一个符号位、一个被通俗地称为尾数（mantissa）的有效数字以及一个指数来表示的。除了0之外，没有任何浮点数等于其符号位取反之后的值，因此 i 的类型必然是整数的。

有符号的整数类型使用2的补码算术运算：为了取一个数值的负值，要对其每一位取反，然后加1，从而得到结果[JLS 15.15.4]。2的补码算术运算的一个很大优势是，0具有惟一的表示形式。如果要对int数值0取负值，将得到 $0xffffffff+1$ ，它仍然是0。但是，这也有一个相应的缺点。总共存在偶数个int数值——准确地说有 2^{32} 个，其中一个用来表示0，剩下奇数个int数值来表示正整数和负整数，这意味着正的和负的int数值的数量必然不相等。换句话说，这暗示着至少有一个int数值，其负值不能正确地表示为int数值。

事实上，恰恰就有一个这样的int数值，它就是`Integer.MIN_VALUE`，即 -2^{31} 。它的十六进制表示是`0x80000000`。其符号位为1，其余所有的位都是0。如果我们对这个值取负值，将得到`0x7fffffff+1`，也就是`0x80000000`，即`Integer.MIN_VALUE!`换句话说，`Integer.MIN_VALUE`是它自己的负值，`Long.MIN_VALUE`也是一样。对这两个值取负值将会产生溢出，但是Java在整数计算中忽略了溢出。其结果已经阐述清楚了，即使它们并不总是你所期望的。

下面的声明将使得布尔表达式($i != 0 \&& i == -i$)的计算结果为`true`，从而使循环无限循环下去：

```
int i = Integer.MIN_VALUE;
```

下面这个也可以：

```
long i = Long.MIN_VALUE;
```

如果你对取模运算很熟悉，那么有必要指出，也可以用代数方法解决这个谜题。Java的int算术运算是实际的算术运算对 2^{32} 取模，因此本谜题需要一个对这种线性全等的非零解决方案：

$$i \equiv -i \pmod{2^{32}}$$

在恒等式的两边加*i*，可以得到：

$$2i \equiv 0 \pmod{2^{32}}$$

对这种全等的非零解决方案就是 $i = 2^{31}$ 。尽管这个值不能表示成int，但是它和 -2^{31} 是全等的，即与Integer.MIN_VALUE全等。

总之，Java使用2的补码的算术运算，是不对称的。对于每一种有符号的整数类型(int、long、byte和short)，负的数值总是比正的数值多一个，这个多出来的值总是这种类型所能表示的最小数值。对Integer.MIN_VALUE取负值不会改变它的值，Long.MIN_VALUE也是如此。对Short.MIN_VALUE取负值并将所产生的int数值转型回short，返回的同样是最初的值(Short.MIN_VALUE)。对Byte.MIN_VALUE来说，也会产生相似的结果。更一般地讲，千万要当心溢出：就像狼人一样，它是个杀手。

对语言设计者的教训与谜题26中的教训一样。考虑对某种不会悄悄发生溢出的整数算术运算形式提供语言级的支持。

谜题34：被计数击倒了

与谜题26和27中的程序一样，下面的程序有一个单重的循环，它记录迭代的次数，并在循环终止时打印这个数。那么，这个程序会打印什么呢？

```
public class Count {
    public static void main(String[] args) {
        final int START = 2000000000;
        int count = 0;
        for (float f = START; f < START + 50; f++)
            count++;
        System.out.println(count);
    }
}
```

解惑34：被计数击倒了

经过表面的分析也许会认为这个程序将打印50，毕竟，循环变量(*f*)被初始化为2 000 000 000，而终止值比初始值大50，并且这个循环具有传统的“半开”形式：它使用的是<操作符，这使得它包括初始值但是不包括终止值。

然而，这种分析遗漏了关键的一点：循环变量是float类型的，而非int类型的。回想一下谜题28，很明显，增量操作(*f++*)不能正常工作。*f*的初始值接近Integer.MAX_VALUE，因此它需要用31位来精确表示，而float类型只能提供24位的精度。对如此巨大的float数值进行增量操作将不会改变其值。因此，这个程序看起来应该无限地循环下去，因为*f*永远也不可能接近终止值。但是，运行该程序，就会发现它并没有无限循环下去，事实上，它立即就终止了，并打印0。怎么回事呢？

问题在于终止条件测试失败了，其方式与增量操作失败的方式非常相似。这个循环只有在循环索引*f*比(float)(START+50)小的情况下才运行。在将一个int与一个float进行比较时，会自动执行从int到float的提升[JLS 15.20.1]。遗憾的是，这种提升是会导致精度丢失的三种拓宽原生类型转换之一[JLS 5.1.2]（另外两个是从long到float和从long到double）。

*f*的初始值太大了，以至于加上50，然后转型为float时，所产生的数值等于直接将*f*转换成float的数值。换句话说，(float)2000000000==2000000050，因此表达式*f*<START+50在第一次执行循环体之前就是false，所以，循环体也就永远没有机会去运行。

改正这个程序非常简单，只需将循环变量的类型从float改为int即可。这样就避免了所有与浮点数计算有关的不精确性：

```
for (int i = START; i < START + 50; i++)
    count++;
```

如果不使用计算机，如何才能知道2 000 000 050与2 000 000 000有相同的float表示呢？关键是要观察到2 000 000 000有10个因子2：它是一个2乘以9个10，而每个10都是 5×2 。这意味着2 000 000 000的二进制表示是以10个0结尾的。50的二进制表示只需要6位，所以将50加到2 000 000 000上不会对右边6位之外的其他位产生影响。特别是，从右边数过来的第7位和第8位仍旧是0。提升这个31位的int到具有24位精度的float会在第7位和第8位之间四舍五入，从而直接舍弃最右边的7位。而最右边的6位是2 000 000 000与2 000 000 050的不同之处，因此它们的float表示是相同的。

这个谜题寓意很简单：不要使用浮点数作为循环索引，因为它会导致无法预测的行为。如果在循环体内需要一个浮点数，那么请使用int或long循环索引，并将其转换为float或double。在将一个int或long转换成一个float或double时，可能会丢失精度，但是至少它不会影响循环本身。当使用浮点数时，要使用double而不是float，除非你肯定float提供了足够的精度，并且存在强制性的性能需求迫使你使用float。适合使用float而不是double的时刻是非常非常少的。

对语言设计者的教训仍然是，对程序员来说悄悄地丢失精度是非常混乱的。请查看谜题31有关这一点的深入讨论。

谜题35：分分钟

下面的程序在模仿一个简单的时钟。它的循环变量表示一个毫秒计数器，其计数值从0开始直至一小时中包含的毫秒数。循环体以定期的时间间隔对一个分钟计数器执行增量操作。最后，该程序打印分钟计数器。那么它会打印什么呢？

```
public class Clock {  
    public static void main(String[] args) {  
        int minutes = 0;  
        for (int ms = 0; ms < 60*60*1000; ms++)  
            if (ms % 60*1000 == 0)  
                minutes++;  
        System.out.println(minutes);  
    }  
}
```

解惑35：分分钟

在这个程序中的循环是一个标准的惯用for循环。它步进毫秒计数器，从0到一小时中的毫秒数，即3 600 000，包括前者但是不包括后者。循环体看起来是每当毫秒计数器的计数值是60 000（一分钟内所包含毫秒数）的倍数时，对分钟计数器执行增量操作。这在循环的生命周期内总共发生了 $3\ 600\ 000/60\ 000$ 次，即60次，因此可能期望程序打印60，毕竟，这就是一小时所包含的分钟数。但是，该程序的运行却会告诉你另外一番景象：它打印的是60000。为什么它会如此频繁地对minutes执行了增量操作呢？

问题在于那个布尔表达式 $(ms \% 60 * 1000 == 0)$ 。你可能会认为这个表达式等价于 $(ms \% 60000 == 0)$ ，但是它们并不等价。取余和乘法操作符具有相同的优先级[JLS 15.17]，因此表达式 $ms \% 60 * 1000$ 等价于 $(ms \% 60) * 1000$ 。如果 $(ms \% 60)$ 等于0，这个表达式就等于0，因此循环每60次迭代就对minutes执行增量操作。这使得最终的结果相差1 000倍。

修正该程序的最简单方式就是在布尔表达式中插入一对括号，以强制规定计算的正确顺序：

```
if ((ms % (60 * 1000)) == 0)
    minutes++;
```

然而，有一个更好的方法可以修正该程序。用被恰当命名的常量来替代所有的魔幻数字：

```
public class Clock {
    private static final int MS_PER_HOUR = 60 * 60 * 1000;
    private static final int MS_PER_MINUTE = 60 * 1000;
    public static void main(String[] args) {
        int minutes = 0;
        for (int ms = 0; ms < MS_PER_HOUR; ms++)
            if ((ms % MS_PER_MINUTE) == 0)
                minutes++;
        System.out.println(minutes);
    }
}
```

之所以要在最初的程序中展现表达式 $ms \% 60 * 1000$ ，是为了误导你去认为乘法比取余有更高的优先级。然而，编译器是忽略空格的，所以千万不要使用空格来表示分组，要使用括号。空格是靠不住的，而括号是从来不说谎的。

异常之谜

本章所描述的谜题都涉及异常，并且和try-finally语句存在着紧密的联系。忠告你一句：谜题44格外难。

谜题36：优柔寡断

下面这个可怜的小程序并不能干脆地做出决定。它的decision方法将返回true。但是它也返回false。那么，它到底打印的是什么呢？甚至，它是合法的吗？

```
public class Indecisive {
    public static void main(String[] args) {
        System.out.println(decision());
    }

    static boolean decision() {
        try {
            return true;
        } finally {
            return false;
        }
    }
}
```

解惑36：优柔寡断

你可能会认为这个程序是不合法的。毕竟，`decision`方法不能同时返回`true`和`false`。如果你尝试一下，就会发现它编译时没有任何错误，并且它打印的是`false`。为什么呢？

原因就是在在一个`try-finally`语句中，`finally`语句块总是在控制权离开`try`语句块时执行[JLS 14.20.2]。无论`try`语句块是正常结束的，还是意外结束的，情况都是如此。在一条语句或一个语句块抛出了一个异常，或者对某个封闭型语句执行了一个`break`或`continue`，或是像这个程序一样在方法中执行了一个`return`时，将发生意外结束。之所以称为意外结束，是因为它们阻止程序按顺序执行下面的语句。

当`try`语句块和`finally`语句块都意外结束时，在`try`语句块中引发意外结束的原因将被丢弃，而整个`try-finally`语句意外结束的原因将与`finally`语句块意外结束的原因相同。在这个程序中，在`try`语句块中的`return`语句所引发的意外结束将被丢弃，而`try-finally`语句意外结束是由`finally`语句块中的`return`造成的。简单地讲，程序尝试着(`try`)返回(`return`)`true`，但是它最终(`finally`)返回(`return`)的是`false`。

几乎永远都不想丢弃意外结束的原因，因为意外结束的最初原因可能对程序的行为来说会显得重要。对于那些在`try`语句块中执行`break`、`continue`或`return`语句，只是为了使其行为被`finally`语句块否决的程序，要理解其行为是特别困难的。

总之，每一个`finally`语句块都应该正常结束，除非抛出不受检查的异常。千万不要用`return`、`break`、`continue`或`throw`来退出`finally`语句块，并且千万不要允许让受检查的异常传播到`finally`语句块之外。

对于语言设计者，也许应该要求`finally`语句块在未出现不受检查的异常时必须正常结束。朝着这个目标，`try-finally`结构将要求`finally`语句块可以正常结束[JLS 14.21]。`return`、`break`或`continue`语句把控制权传递到`finally`语句块之外应该是被禁止的，任何可以引发将受检查异常传播到`finally`语句块之外的语句也同样应该是被禁止的。

谜题37：极端不可思议

本谜题测试的是你对某些规则的掌握程度，这些规则用于声明从方法中抛出并被catch语句块捕获的异常。下面的三个程序每一个都会打印些什么？不要假设它们都可以通过编译：

```
import java.io.IOException;
public class Arcane1 {
    public static void main(String[] args) {
        try {
            System.out.println("Hello world");
        } catch(IOException e) {
            System.out.println("I've never seen println fail!");
        }
    }
}

public class Arcane2 {
    public static void main(String[] args) {
        try {
            // If you have nothing nice to say, say nothing
        } catch(Exception e) {
            System.out.println("This can't happen");
        }
    }
}

interface Type1 {
    void f() throws CloneNotSupportedException;
}
interface Type2 {
    void f() throws InterruptedException;
}
interface Type3 extends Type1, Type2 {
}
public class Arcane3 implements Type3 {
    public void f() {
        System.out.println("Hello world");
    }
    public static void main(String[] args) {
        Type3 t3 = new Arcane3();
        t3.f();
    }
}
```

解惑37：极端不可思议

第一个程序，Arcane1，展示了受检查异常的一个基本原则。它看起来应该是可以编译的：try子句执行I/O，并且catch子句捕获IOException异常。但是这个程序不能编译，因为println方法没有声明会抛出任何受检查异常，而IOException却正是一个受检查异常。语言规范中描述道：如果一个catch子句要捕获一个类型为E的受检查异常，而其相对应的try子句不能抛出E的某种子类型的异常，那么这就是一个编译期错误[JLS 11.2.3]。

基于同样的理由，第二个程序，Arcane2，看起来应该是不可以编译的，但是它却可以。它之所以可以编译，是因为它惟一的catch子句检查了Exception。尽管JLS在这一点上十分含混不清，但是捕获Exception或Throwable的catch子句是合法的，不管与其相对应的try子句的内容为何。尽管Arcane2是一个合法的程序，但是catch子句的内容永远不会被执行，这个程序什么都不会打印。

第三个程序，Arcane3，看起来它也不能编译。方法f在Type1接口中声明要抛出受检查异常CloneNotSupportedException，并且在Type2接口中声明要抛出受检查异常InterruptedException。Type3接口继承了Type1和Type2，因此，看起来在静态类型为Type3的对象上调用方法f时，有潜在可能会抛出这些异常。一个方法必须要么捕获其方法体可以抛出的所有受检查异常，要么声明它将抛出这些异常。Arcane3的main方法在静态类型为Type3的对象上调用了方法f，但它对CloneNotSupportedException和InterruptedException并没有做这些处理。那么，为什么这个程序可以编译呢？

上述分析的缺陷在于对“Type3.f可以抛出在Type1.f上声明的异常和在Type2.f上声明的异常”所做的假设。这并不正确，因为每一个接口都限制了方法f可以抛出的受检查异常集合。一个方法可以抛出的受检查异常集合是它所适用的所有类型声明要抛出的受检查异常集合的交集，而不是合集。因此，静态类型为Type3的对象上f方法根本就不能抛出任何受检查异常。因此，Arcane3可以毫无错误地通过编译，并且打印Hello world。

总之，第一个程序说明了一项基本要求，即对于捕获受检查异常的catch子句，只有在相应的try子句可以抛出这些异常时才被允许。第二个程序说明了不会应用这项要求的冷僻案例。第三个程序说明了多个继承而来的throws子句的交集，将减少而不是增加方法被允许抛出的异常数量。本谜题所说明的行为一般不会引发难以捉摸的bug，但是你第一次看到它们时，可能会有点吃惊。

谜题38：不受欢迎的宾客

本谜题中的程序所建模的系统，将尝试着从其环境中读取一个用户ID，如果这种尝试失败了，则缺省地认为它是一个来宾用户。该程序的编写者将面对有一个静态域的初始化表达式可能会抛出异常的情况。因为Java不允许静态初始化操作抛出受检查异常，所以初始化必须包装在try-finally语句块中。那么，下面的程序会打印什么呢？

```
public class UnwelcomeGuest {
    public static final long GUEST_USER_ID = -1;

    private static final long USER_ID;
    static {
        try {
            USER_ID = getUserIdFromEnvironment();
        } catch (IdUnavailableException e) {
            USER_ID = GUEST_USER_ID;
            System.out.println("Logging in as guest");
        }
    }

    private static long getUserIdFromEnvironment()
        throws IdUnavailableException {
        throw new IdUnavailableException(); // Simulate an error
    }

    public static void main(String[] args) {
        System.out.println("User ID: " + USER_ID);
    }
}

class IdUnavailableException extends Exception {
    IdUnavailableException(){}
}
```

解惑38：不受欢迎的宾客

该程序看起来很直观。对getUserIdFromEnvironment的调用将抛出一个异常，从而使程序将GUEST_USER_ID(-1L)赋值给USER_ID，并打印Logging in as guest。然后执行main方法，使程序打印User ID: -1。外表再次欺骗了我们，该程序并不能编译。如果你尝试着编译它，将看到和下面内容类似的一条错误信息：

```
UnwelcomeGuest.java:10:  
    variable USER_ID might already have been assigned  
        USER_ID = GUEST_USER_ID;  
        ^
```

问题出在哪里？USER_ID域是一个空final，它是一个在声明中没有进行初始化操作的final域[JLS 4.12.4]。很明显，只有在对USER_ID赋值失败时，才会在try语句块中抛出异常，因此，在catch语句块中赋值是相当安全的。不管怎样执行静态初始化操作语句块，只会对USER_ID赋值一次，这正是空final所要求的。为什么编译器不知道这些呢？

要确定一个程序是否可以不止一次地对一个空final进行赋值是很困难的问题。事实上，这是不可能的。这等价于经典的停机问题，它通常被认为是不可能解决的[Turing 36]。为了能够编写一个Java编译器，语言规范在这一点上采用了保守的方式。在程序中，一个空final域只有在它的确未赋过值的地方才可以被赋值。规范长篇大论，对此术语提供了一个准确但保守的定义[JLS 16]。因为它是保守的，所以编译器必须拒绝某些可以证明是安全的程序。这个谜题就说明了这样的一个程序。

幸运的是，不必为了编写Java程序而去学习那些骇人的用于明确赋值的细节。通常明确赋值规则不会有任何妨碍。如果碰巧编写了一个真的可能会对一个空final赋值超过一次的程序，编译器会帮你指出的。只有在极少的情况下，就像本谜题一样，才会编写一个安全的程序，但是它并不满足规范的形式化要求。编译器的抱怨就好像你编写了一个不安全的程序一样，而且你必须修改你的程序以满足它。

解决这类问题的最好方式就是将这个烦人的域从空final类型改变为普通的final类型，用一个静态域的初始化操作替换静态的初始化语句块。实现这一点的最佳方式是重构静态语句块中的代码为一个助手方法：

```

public class UnwelcomeGuest {
    public static final long GUEST_USER_ID = -1;
    private static final long USER_ID = getUserIdOrGuest();
    private static long getUserIdOrGuest(){
        try {
            return getUserIdFromEnvironment();
        } catch (IdUnavailableException e) {
            System.out.println("Logging in as guest");
            return GUEST_USER_ID;
        }
    }
    ...// The rest of the program is unchanged
}

```

程序的这个版本很显然是正确的，而且比最初的版本更具可读性，因为它为了域值的计算而增加了一个描述性的名字，而最初的版本只有一个匿名的静态初始化操作语句块。这样修改程序，它就可以如所期望地运行了。

总之，大多数程序员都不需要学习明确赋值规则的细节。该规则的行为通常都是正确的。如果必须重构一个程序，以消除由明确赋值规则所引发的错误，那么应该考虑添加一个新方法。这样做除了可以解决明确赋值问题，还可以提高程序的可读性。

谜题39：您好，再见

下面的程序在寻常的Hello world程序中添加了一段不寻常的曲折操作。那么，它将会打印什么呢？

```

public class HelloGoodbye {
    public static void main(String[] args) {
        try {
            System.out.println("Hello world");
            System.exit(0);
        } finally {
            System.out.println("Goodbye world");
        }
    }
}

```

解惑39：您好，再见

这个程序包含两个`println`语句：一个在`try`语句块中，另一个在相应的`finally`语句块中。`try`语句块执行它的`println`语句，并且通过调用`System.exit`来提前结束执行。在此时，你可能希望控制权会转交给`finally`语句块。然而，如果你运行该程序，就会发现它永远不会说再见：它只打印了Hello world。这是否违背了谜题36中所解释的原则呢？

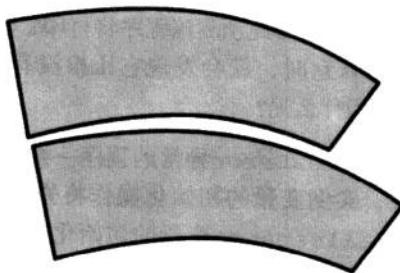
不论`try`语句块的执行是正常地还是意外地结束，`finally`语句块确实都会执行。然而在这个程序中，`try`语句块根本就没有结束其执行过程。**`System.exit`方法将停止当前线程和所有其他当场死亡的线程。**`finally`子句的出现并不能给予线程继续执行的特殊权限。

当`System.exit`被调用时，虚拟机（VM）在关闭前要执行两项清理工作。首先，它执行所有的关闭挂钩操作，这些挂钩已经注册到`Runtime.addShutdownHook`上。这对于释放VM之外的资源很有帮助。**务必要为那些必须在VM退出之前发生的行为关闭挂钩。**下面的程序版本示范了这种技术，它可以如我们所期望地打印出Hello world和Goodbye world：

```
public class HelloGoodbye {
    public static void main(String[] args) {
        System.out.println("Hello world");
        Runtime.getRuntime().addShutdownHook(
            new Thread() {
                public void run() {
                    System.out.println("Goodbye world");
                }
            });
        System.exit(0);
    }
}
```

在`System.exit`被调用时VM执行的第二个清理任务与终结器有关。如果`System.runFinalizersOnExit`或它的魔鬼双胞胎`Runtime.runFinalizersOnExit`被调用了，那么VM将在所有还未终结的对象上调用终结器。这些方法很久以前就已经过时了，而且其原因也很合理。**无论什么原因，永远不要调用`System.runFinalizersOnExit`和`Runtime.runFinalizersOnExit`：**它们属于Java类库中最危险的方法[ThreadStop]。调用这些方法导致的结果是，终结器会在那些其他线程正在并发操作的对象上运行，从而导致不确定的行为或死锁。

总之，`System.exit`将立即停止所有的程序线程，它并不会使`finally`语句块得到调用，但是它在停止VM之前会执行关闭挂钩操作。当VM被关闭时，请使用关闭挂钩来终止外部资源。通过调用`System.halt`可以在不执行关闭挂钩的情况下停止VM，但是很少使用这个方法。



谜题40：不情愿的构造器

尽管在方法声明中看到`throws`子句是很常见的，但是在构造器的声明中看到`throws`子句就很少见了。下面的程序就有一个这样的声明。那么，它将打印什么呢？

```
public class Reluctant {  
    private Reluctant internalInstance = new Reluctant();  
  
    public Reluctant() throws Exception {  
        throw new Exception("I'm not coming out");  
    }  
  
    public static void main(String[] args) {  
        try {  
            Reluctant b = new Reluctant();  
            System.out.println("Surprise!");  
        } catch (Exception ex) {  
            System.out.println("I told you so");  
        }  
    }  
}
```

解惑40：不情愿的构造器

main方法调用了Reluctant构造器，它将抛出一个异常。你可能期望catch子句能够捕获这个异常，并且打印I told you so。凑近仔细看看这个程序就会发现，Reluctant实例还包含第二个内部实例，它的构造器也会抛出一个异常。无论抛出哪一个异常，看起来main中的catch子句都应该捕获它，因此预测该程序将打印I told you so似乎是一个安全的赌注。但是当你尝试着去运行它时，就会发现它压根没有去做这类的事情：它抛出了StackOverflowError异常，为什么呢？

与大多数抛出StackOverflowError异常的程序一样，本程序也包含了一个无限递归。当你调用一个构造器时，**实例变量的初始化操作将先于构造器的程序体而运行**[JLS 12.5]。在本谜题中，internalInstance变量的初始化操作递归调用了构造器，而该构造器通过再次调用Reluctant构造器而初始化该变量自己的internalInstance域，如此无限递归下去。这些递归调用在构造器程序体获得执行机会之前就会抛出StackOverflowError异常。因为StackOverflowError是Error的子类型而不是Exception的子类型，所以main中的catch子句无法捕获它。

对于一个对象包含与它自己类型相同的实例的情况，并不少见。例如，链接列表节点、树节点和图节点都属于这种情况。必须非常小心地初始化这样的包含实例，以避免StackOverflowError异常。

至于本谜题名义上的题目：声明将抛出异常的构造器，需要注意，**构造器必须声明其实例初始化操作会抛出的所有受检查异常**。下面这个举例说明了常见的“服务提供商”模式的程序，将不能编译，因为它违反了这条规则：

```
public class Car {
    private static Class engineClass = ...; // Service provider
    private Engine engine = (Engine)engineClass.newInstance();
    public Car(){ } //Throws two checked exceptions!
}
```

尽管其构造器没有任何程序体，但是它将抛出两个受检查异常，InstantiationException和IllegalAccessException。它们是Class.newInstance抛出的，该方法是在初始化engine域的时候被调用的。修正该程序的最好方式是创建一个私有的、静态的助手方法，它负责计算域的初始值，并恰当地处理异常。在本案中，我们假设选择engineClass所引用的Class对象，保证它是可访问的并且是可实例化的，那么下面的Car版本将可以毫无错误地通过编译：

```
//Fixed - instance initializers don't throw checked exceptions
public class Car {
    private static Class engineClass = ...;
    private Engine engine = newEngine();
    private static Engine newEngine() {
        try {
            return (Engine)engineClass.newInstance();
        } catch (IllegalAccessException e) {
            throw new AssertionError(e);
        } catch (InstantiationException e) {
            throw new AssertionError(e);
        }
    }
    public Car(){}
}
```

总之，实例初始化操作是先于构造器的程序体而运行的。实例初始化操作抛出的任何异常都会传播给构造器。如果初始化操作抛出的是受检查异常，那么构造器必须声明也会抛出这些异常，但是应该避免这样做，因为它会造成混乱。最后，对于我们所设计的类，如果其实例包含同样属于这个类的其他实例，那么对这种无限递归要格外当心。

谜题41：域和流

下面的方法将一个文件拷贝至另一个文件，并且设计为关闭它所创建的每一个流，即使碰到I/O错误也要如此。遗憾的是，它并非总是能够做到这一点。为什么不能呢？如何才能修正它呢？

```
static void copy(String src, String dest) throws IOException {
    InputStream in = null;
    OutputStream out = null;
    try {
        in = new FileInputStream(src);
        out = new FileOutputStream(dest);
        byte[] buf = new byte[1024];
        int n;
        while ((n = in.read(buf)) >=0)
            out.write(buf, 0, n);
    } finally {
        if (in != null) in.close();
        if (out != null) out.close();
    }
}
```

解惑41：域和流

这个程序看起来已经面面俱到了。其流（`in`和`out`）被初始化为`null`，并且新的流一旦被创建，它们马上就被设置为这些流域的新值。对于这些域所引用的流，如果不为空，则`finally`语句块会将其关闭。即便在拷贝操作引发了一个`IOException`的情况下，`finally`语句块也会在方法返回之前执行。出什么错了呢？

问题在`finally`语句块自身中。`close`方法也可能会抛出`IOException`异常。如果这正好发生在`in.close`被调用之时，那么这个异常就会阻止`out.close`被调用，从而使输出流保持在开放状态。

请注意，该程序违反了谜题36的建议：对`close`的调用可能会导致`finally`语句块意外结束。遗憾的是，编译器并不能帮助你发现此问题，因为`close`方法抛出的异常与`read`和`write`抛出的异常类型相同，而其外围方法（`copy`）声明将传播该异常。

解决方式是将每一个`close`都包装在一个嵌套的`try`语句块中。下面的`finally`语句块的版本可以保证在两个流上都会调用`close`：

```

} finally {
    if (in != null) {
        try {
            in.close();
        } catch (IOException ex) {
            // There is nothing we can do if close fails
        }
    }
    if (out != null){
        try {
            out.close();
        } catch (IOException ex) {
            // Again, there is nothing we can do if close fails
        }
    }
}
}

```

从5.0版本开始，可以对代码进行重构，以利用`Closeable`接口：

```

} finally {
    closeIgnoringException(in);
    closeIgnoringException(out);
}
}

```

```

}
private static void closeIgnoringException(Closeable c) {
    if (c != null) {
        try {
            c.close();
        } catch (IOException ex) {
            // There is nothing we can do if close fails
        }
    }
}

```

总之，当你在**finally**语句块中调用**close**方法时，要用一个嵌套的**try-catch**语句来保护它，以防止**IOException**的传播。更一般地讲，对于任何在**finally**语句块中可能抛出的受检查异常都要进行处理，而不是任其传播。这是谜题36中教训的一种特例，而对语言设计者的教训情况也相同。

谜题42：异常为循环而抛

下面的程序循环遍历了一个int类型的数组序列，并且记录了满足某个特定属性的数组个数。那么，该程序会打印什么呢？

```

public class Loop {
    public static void main(String[] args) {
        int[][] tests = { { 6, 5, 4, 3, 2, 1 }, { 1, 2 },
                          { 1, 2, 3 }, { 1, 2, 3, 4 }, { 1 } };
        int successCount = 0;

        try {
            int i = 0;
            while (true) {
                if (thirdElementIsThree(tests[i++]))
                    successCount++;
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            // No more tests to process
        }
        System.out.println(successCount);
    }

    private static boolean thirdElementIsThree(int[] a) {
        return a.length >= 3 & a[2] == 3;
    }
}

```

解惑42：异常为循环而抛

该程序用thirdElementIsThree方法测试了tests数组中的每一个元素。遍历这个数组的循环显然是非传统的循环：它不是在循环变量等于数组长度的时候终止，而是在它试图访问一个并不在数组中的元素时终止。尽管它是非传统的，但是这个循环应该可以工作。如果传递给thirdElementIsThree的参数具有3个或更多的元素，并且其第3个元素等于3，那么该方法将返回true。对于tests中的5个元素来说，有2个将返回true，因此看起来该程序应该打印2。运行它，就会发现打印的是0。肯定是因为哪里出了问题吗？

事实上，这个程序犯了两个错误。第1个错误是该程序使用了一种可怕的循环惯用法，该惯用法依赖的是对数组的访问会抛出异常。这种惯用法不仅难以阅读，而且运行速度非常慢。不要使用异常控制循环；应该只为异常条件而使用异常[EJ Item 39]。为了纠正这个错误，可以将整个try-finally语句块替换为循环遍历数组的标准惯用法：

```
for (int i = 0; i < tests.length; i++)
    if (thirdElementIsThree(tests[i]))
        successCount++;
```

如果使用的是5.0或者是更新的版本，那么可以用for循环结构来代替：

```
for (int[] test : tests)
    if(thirdElementIsThree(test))
        successCount++;
```

就第一个错误的糟糕情况来说，只有它自己还不足以产生我们所观察的行为。然而，修正该错误可以帮助我们找到真正的bug，它更加微妙：

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 2
    at Loop.thirdElementIsThree(Loop.java:19)
    at Loop.main(Loop.java:13)
```

很明显，在thirdElementIsThree方法中有一个bug：它抛出了一个ArrayIndexOutOfBoundsException异常。这个异常事先伪装成了那个可怕的基于异常循环的终止条件。

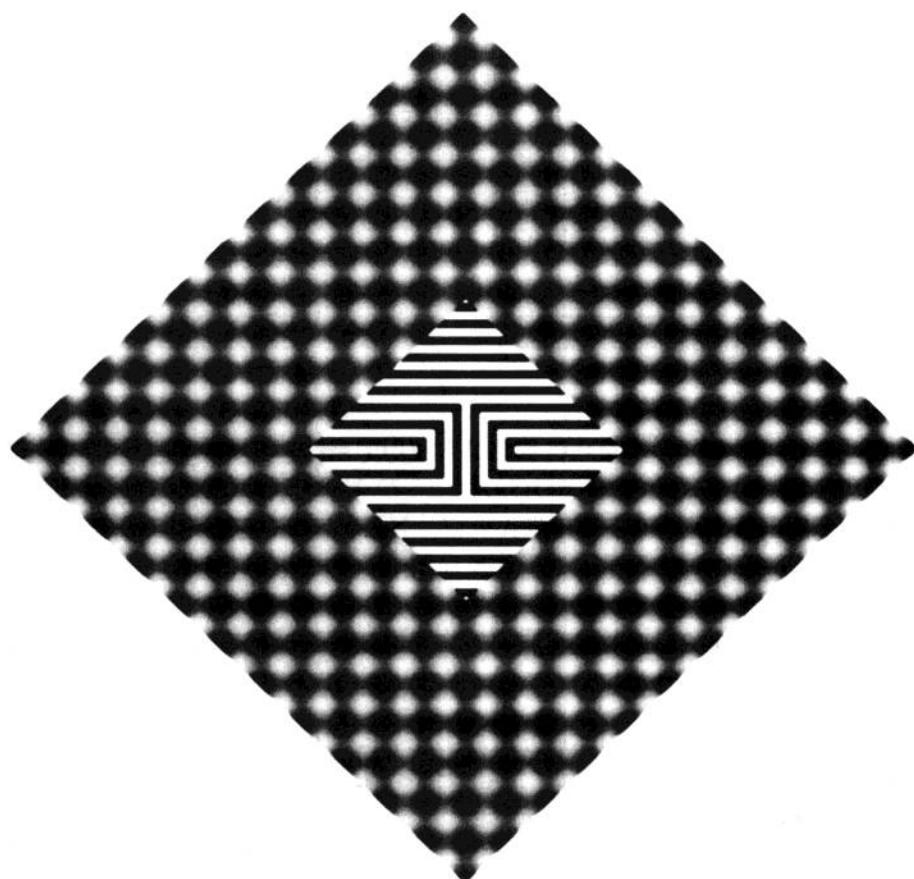
如果传递给thirdElementIsThree的参数具有3个或更多的元素，并且其第3个元素等于3，那么该方法将返回true。问题是在这些条件不满足时它会做什么。如果你仔细观察将会返回值的那个布尔表达式，就会发现它与大多数布尔AND操作有一点不一样。这个表达式是`a.length >= 3 & a[2] == 3`。通常，在这种情况下看到的是`&&`操作符，而这个表达式使用的是`&`操作符。是一个位AND操作符吗？

事实证明`&`操作符有其他的含义。除了常见的作为整型操作数的位AND操作符之外，当用于布尔操作数时，它的功能被重载为逻辑AND操作符[JLS 15.22.2]。这个操作符与更经常使用的条件AND操作符有所不同，`&`操作符总是要计算它的两个操作数，而`&&`操作符在其左边的操作数被计算为`false`时，就不再计算右边的操作数了[JLS 15.23]。因此，thirdElementIsThree方法总是试图访问其数组参数的第3个元素，即使该数组参数的元素不足3个也是如此。修正这个方法只需将`&`操作符替换为`&&`操作符即可。通过这样的修改，这个程序就可以打印所期望的2了：

```
private static boolean thirdElementIsThree(int[] a) {
    return a.length >= 3 && a[2] == 3;
}
```

正像有一个逻辑AND操作符伴随着更经常使用的条件AND操作符一样，还有一个逻辑OR操作符(`l`)也伴随着条件OR操作符(`l|l`)[JLS 15.22.2, 15.24]。`l`操作符总是要计算它的两个操作数，而`l|l`操作符在其左边的操作数被计算为`true`时，就不再计算右边的操作数了。稍不注意，就很容易使用了逻辑操作符而不是条件操作符。遗憾的是，编译器并不能帮助你发现这种错误。有意识地使用逻辑操作符的情形非常少见，少到了我们对所有这样使用的程序都持怀疑态度的地步。如果你真的想使用这样的操作符，为了使你的目的清楚起见，请加上注释。

总之，不要使用那些可怕的运用异常而不是显式终止测试的循环惯用法，因为这种惯用法非常不清晰，而且会掩盖bug。要意识到逻辑AND和OR操作符的存在，并且不要因无意识的误用而受害。对语言设计者来说，这又是一个操作符重载会导致混乱的明证。对于在条件AND和OR操作符之外还要提供逻辑AND和OR操作符这一点，并没有很明显的理由。如果这些操作符确实要得到支持的话，它们应该与其相对应的条件操作符存在着视觉上的明显差异。



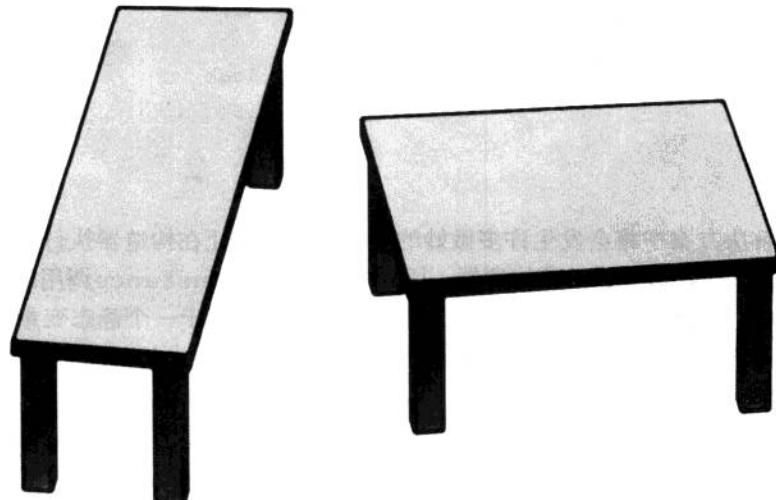
谜题43：异常地危险

在JDK1.2中，`Thread.stop`、`Thread.suspend`以及其他许多线程相关的方法都因为不安全而不推荐使用了[ThreadStop]。下面的方法说明了使用`Thread.stop`可以实现的可怕事情之一。

```
// Don't do this—circumvents exception checking!
public static void sneakyThrow(Throwable t) {
    Thread.currentThread().stop(t); // Deprecated!!
}
```

这个讨厌的小方法所做的事情正是`throw`语句要做的事情，但是它绕过了编译器的所有异常检查操作。你可以在代码的任意一点上抛出任何被检查的或不被检查的异常，而编译器对此连眉头都不会皱一下。

不使用任何不推荐的方法，也可以编写在功能上等价于`sneakyThrow`的方法。事实上，至少有两种方式可以实现这一点，其中一种只能在5.0或更新的版本中运行。你能够编写这样的方法吗？它必须是用Java而不是用JVM字节码编写的，你不能在客户对它编译完之后再去修改它。你的方法不必是完美无瑕的：如果它不能抛出一两个Exception的子类，也是可以接受的。



解惑43：异常地危险

本谜题的一种解决之道是利用Class.newInstance方法中的设计缺陷，该方法通过反射来对一个类进行实例化。引用有关该方法的文档中的话[Java-API]：“请注意，该方法将传播从空的[换句话说，就是无参数的]构造器所抛出的任何异常，包括受检查的异常。使用这个方法可以有效地绕开在其他情况下都会执行的编译期异常检查。”一旦了解了这一点，编写一个sneakyThrow的等价方法就不是太难了。

```
// Don't do this either-circumvents exception checking!
public class Thrower {
    private static Throwable t;

    private Thrower() throws Throwable {
        throw t;
    }

    public static synchronized void sneakyThrow(Throwable t) {
        Thrower.t = t;
        try {
            Thrower.class.newInstance();
        } catch (InstantiationException e) {
            throw new IllegalArgumentException();
        } catch (IllegalAccessException e) {
            throw new IllegalArgumentException();
        } finally {
            Thrower.t = null; // Avoid memory leak
        }
    }
}
```

在这个解决方案中将会发生许多微妙的事情。我们想让在构造器执行期间所抛出的异常不能作为一个参数传递给该构造器，因为Class.newInstance调用的是一个类的无参数构造器。因此，sneakyThrow方法将这个异常藏匿于一个静态变量中。为了使该方法线程安全，它必须被同步，这使得对其的并发调用将顺序地使用静态域t。

要注意的是，t这个域在从finally语句块中出来时是被赋值为空的：这只是因为该方法虽然是劣等的，但这并不意味着它还应该是内存泄漏的。如果这个域不是被赋为空出来的，那么它阻止该异常被垃圾回收。最后，请注意，如果让该方法抛出一个InstantiationException或是一个IllegalAccessException异常，它将以抛

出一个IllegalArgumentException而失败。这是这项技术的一个内在限制。

Class.newInstance的文档继续描述道“Constructor.newInstance方法通过将构造器抛出的任何异常都包装在一个（受检查的）InvocationTargetException异常中而避免了这个问题。”很明显，Class.newInstance应该是做了相同的处理，但是纠正这个缺陷已经为时过晚，因为这么做将引入源代码级别的不兼容性，这将使许多依赖于Class.newInstance的程序崩溃。而反对用这个方法也不切实际，因为它太常用了。当你在使用它时，一定要意识到Class.newInstance可以抛出它没有声明过的受检查异常。

被添加到5.0版本中的“泛型”可以为本谜题提供一个完全不同的解决方案。为了实现最大的兼容性，泛型是通过类型擦除来实现的：泛型信息是在编译期而非运行期检查的[JLS 4.7]。下面的解决方案就利用了这项技术：

```
// Don't do this either - circumvents exception checking!
class TigerThrower<T extends Throwable> {
    public static void sneakyThrow(Throwable t) {
        new TigerThrower<Error>().sneakyThrow2(t);
    }

    private void sneakyThrow2(Throwable t) throws T {
        throw (T) t;
    }
}
```

这个程序在编译时将产生一条警告信息：

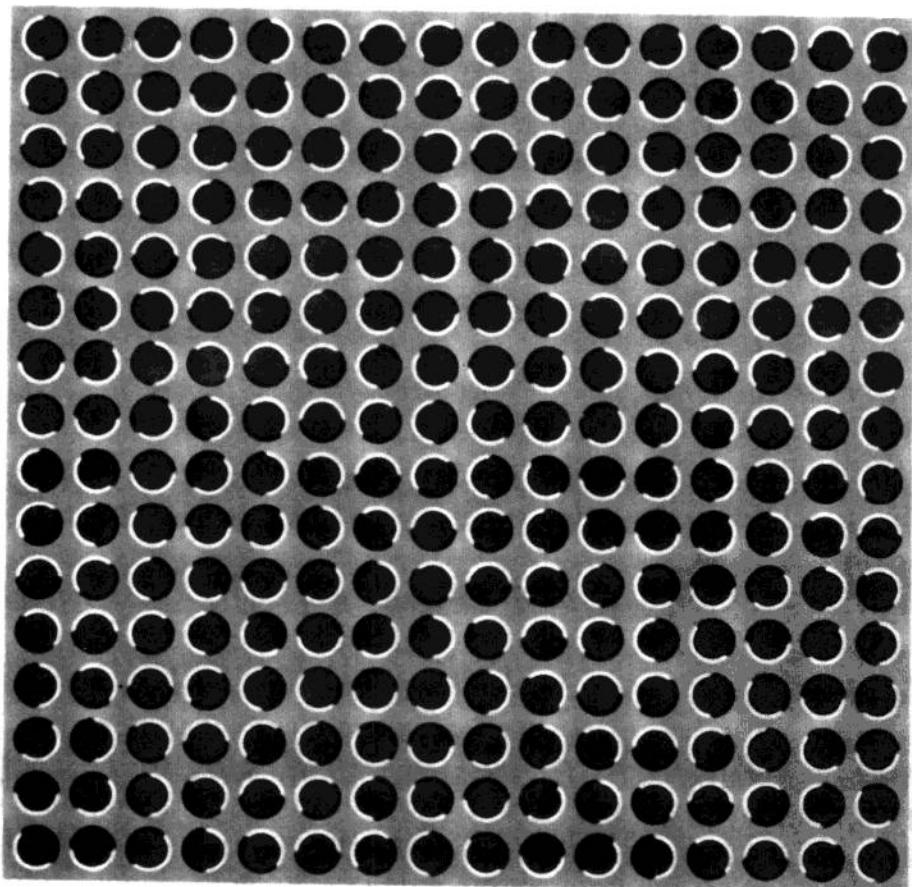
```
TigerThrower.java:7:warning: [unchecked] unchecked cast
found   : java.lang.Throwable, required: T
    throw (T) t;
               ^

```

警告信息是编译器所采用的一种手段，用来告诉你：你可能正在搬起石头砸自己的脚，而且事实也正是如此。“不受检查的转型”警告告诉你这个有问题的转型将不会在运行时刻受到检查。当你获得了一个不受检查的转型警告时，应该修改你的程序以消除它，或者可以确信这个转型不会失败。如果不这么做，那么某个其他的转型可能会在未来不确定的某个时刻失败，而你也就很难跟踪此错误到其源头了。对于本谜题所示的情况，其情况更糟糕：在运行期抛出的异常可能与方法的签名不一致。sneakyThrow2方法正是利用了这一点。

对平台设计者来说，有好几条教训。在设计诸如反射类库之类在语言之外实现的类库时，要保留语言所作的所有承诺。当从头设计一个支持泛型的平台时，要考虑强制要求其在运行期的正确性。Java泛型工具的设计者可没有这么做，因为他们受制于通用类库必须能够与现有客户进行互操作的要求。对于违反方法签名的异常，为了消除其产生的可能性，应该考虑强制在运行期进行异常检查。

总之，Java的异常检查机制并不是虚拟机强制执行的。它只是一个编译期工具，被设计用来帮助我们更加容易地编写正确的程序，但是在运行期可以绕过它。要想减少因为这类问题而被曝光的次数，就不要忽视编译器给出的警告信息。



谜题44：删除类

请考虑下面的两个类：

```
public class Strange1 {
    public static void main(String[] args) {
        try {
            Missing m = new Missing();
        } catch (java.lang.NoClassDefFoundError ex) {
            System.out.println("Got it!");
        }
    }
}

public class Strange2 {
    public static void main(String[] args) {
        Missing m;
        try {
            m = new Missing();
        } catch (java.lang.NoClassDefFoundError ex) {
            System.out.println("Got it!");
        }
    }
}
```

Strange1和Strange2都用到了下面这个类：

```
class Missing {
    Missing() { }
}
```

如果你编译所有这三个类，然后在运行Strange1和Strange2之前删除Missing.class文件，你就会发现这两个程序的行为有所不同。其中一个抛出了一个未被捕获的NoClassDefFoundError异常，而另一个却打印了Got it!到底哪一个程序具有哪一种行为，你又如何去解释这种行为上的差异呢？



解惑44：删除类

程序Strange1只在其try语句块中提及Missing类型，因此你可能会认为它捕获NoClassDefFoundError异常，并打印Got it！另一方面，程序Strange2在try语句块之外声明了一个Missing类型的变量，因此你可能会认为所产生的NoClassDefFoundError异常不会被捕获。如果你试着运行这些程序，就会看到它们的行为正好相反：Strange1抛出了未被捕获的NoClassDefFoundError异常，而Strange2却打印了Got it！怎样才能解释这些奇怪的行为呢？

如果查看Java规范以找出应该抛出NoClassDefFoundError异常的地方，那么你不会得到很多的指导信息。该规范描述道，这个错误可以“在（直接或间接）使用某个类的程序中的任何地方”抛出[JLS 12.2.1]。当VM调用Strange1和Strange2的main方法时，这些程序都间接使用了Missing类，因此，它们都在其权利范围内于这一点上抛出了该错误。

于是，本谜题的答案就是这两个程序可以依据其实现而展示出各自不同的行为。但是这并不能解释为什么在所有我们已知的Java实现上，这些程序的实际行为与所认为的必然行为都正好相反。要查明为什么这样，我们需要研究编译器生成的这些程序的字节码。

比较Strange1和Strange2的字节码，就会发现两者几乎是一样的。除了类名之外，惟一的差异就是catch语句块所捕获的参数ex与VM本地变量之间的映射关系不同。尽管哪一个程序变量被指派给了哪一个VM变量的具体细节会因编译器的不同而有所差异，但是对于和上述程序一样简单的程序来说，这些细节不太可能会差异很大。下面是通过执行javap -c Strange1命令而显示的Strange1.main的字节码：

```

0: new           #2; //class Missing
3: dup
4: invokespecial #3; //Method Missing. "<init>":()V
7: astore_1
8: goto 20
11: astore_1
12: getstatic    #5; // Field System.out:Ljava/io/PrintStream;
15: ldc          #6; // String "Got it!"
17: invokevirtual #7; //Method PrintStream.println: (String;) V
20: return
Exception table:
from to target type
 0  8  11  Class java/lang/NoClassDefFoundError

```

Strange2.main相对应的字节码与其只有一条指令不同：

```
11: astore_2
```

这是一条将catch语句块中的捕获异常存储到捕获参数ex中的指令。在Strange1中，这个参数是存储在VM变量1中的，而在Strange2中，它是存储在VM变量2中的。这就是两个类之间惟一的差异，但是它所造成的程序行为上的差异是多么大呀！

为了运行一个程序，VM要加载和初始化包含main方法的类。在加载和初始化之间，VM必须链接类[JLS 12.3]。链接的第一阶段是校验，校验要确保一个类是良构的，并且遵循语言的语法要求。校验非常关键，它维持着可以区分像Java这样的安全语言与像C或C++这样的不安全语言的各种承诺。

在Strange1和Strange2两个类中，本地变量m碰巧都被存储在VM变量1中。两个版本的main都有一个连接点，从两个不同位置而来的控制流汇聚于此。该连接点就是指令20，即从main返回的指令。在正常结束try语句块的情况下，我们执行到指令8，即goto 20，从而可以到达指令20；而对于在catch语句块中结束的情况，我们将执行指令17，并按顺序执行下去，到达指令20。

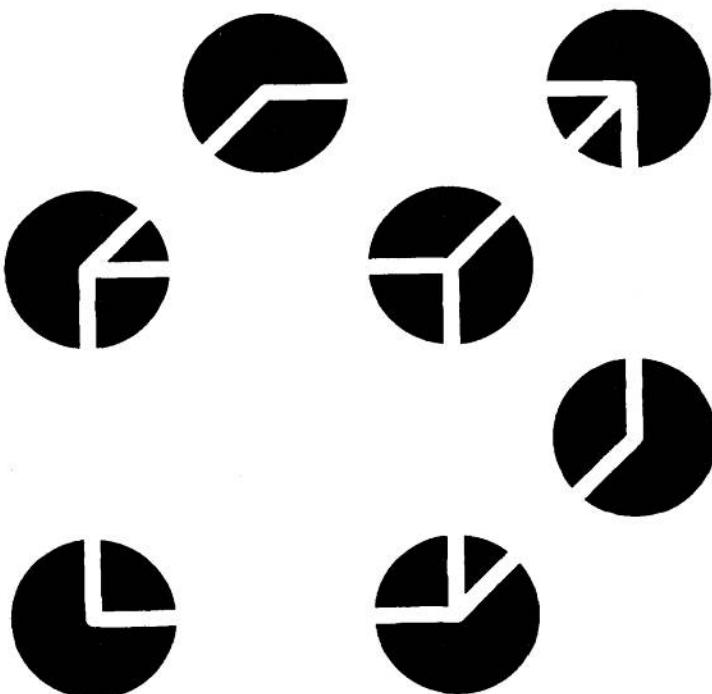
连接点的存在使得在校验Strange1类时产生异常，而在校验Strange2类时并不会产生异常。当校验执行对Strange1.main的流分析（flow analysis）[JLS 12.3.1]时，由于指令20可以通过两条不同的路径到达，因此校验器必须合并变量1中的类型。两种类型是通过计算它们的首个公共超类（first common superclass）[JVMS 4.9.2]而合并的。两个类的首个公共超类是它们所共有的最详细而精确的超类。

在Strange1.main方法中，当从指令8到达指令20时，VM变量1的状态包含了一个Missing类的实例。当从指令17到达时，它包含了一个NoClassDefFoundError类的实例。为了计算首个公共超类，校验器必须加载Missing类以确定其超类。因为Missing.class文件已经被删除了，所以校验器不能加载它，因而抛出了一个NoClassDefFoundError异常。请注意，这个异常是在校验期间、在类被初始化之前，并且早在main方法开始执行之前就抛出的。这就解释了为什么没有打印关于这个未被捕获异常的跟踪栈信息。

要想编写一个能够探测类丢失的程序，请使用反射来引用类而不要使用通常的语言结构[EJ Item35]。下面展示了用这种技巧重写的程序：

```
public class Strange {  
    public static void main(String[] args) throws Exception{  
        try {  
            Object m = Class.forName("Missing").newInstance();  
        } catch (ClassNotFoundException ex) {  
            System.err.println("Got it!");  
        }  
    }  
}
```

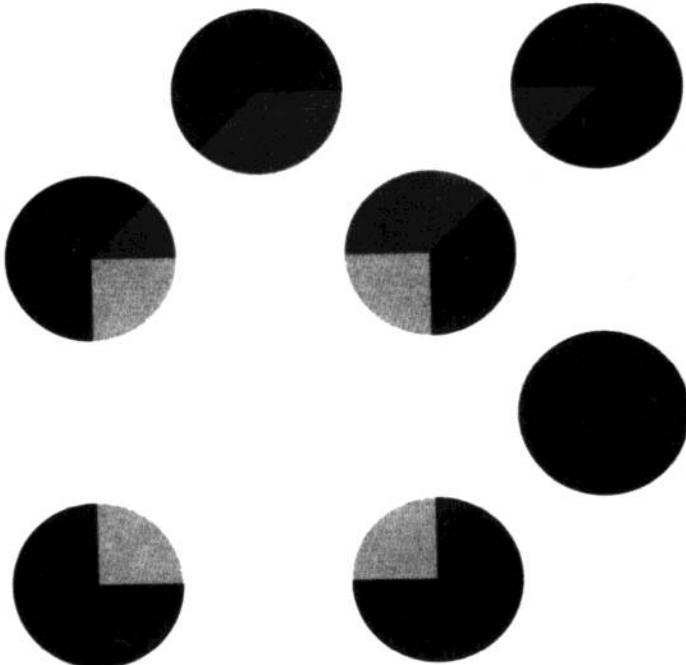
总之，不要对捕获**NoClassDefFoundError**形成依赖。语言规范非常仔细地描述了类初始化是在何时发生的[JLS 12.4.1]，但是类被加载的时机却远不可预测。更一般地讲，捕获**Error**及其子类型几乎是完全不恰当的。这些异常是为那些不能恢复的错误而保留的。



谜题45：令人疲惫不堪的测验

本谜题将测试你对递归的了解程度。下面的程序将做些什么呢？

```
public class Workout {  
    public static void main(String[] args) {  
        workHard();  
        System.out.println("It's nap time.");  
    }  
  
    private static void workHard() {  
        try {  
            workHard();  
        } finally {  
            workHard();  
        }  
    }  
}
```



解惑45：令人疲惫不堪的测验

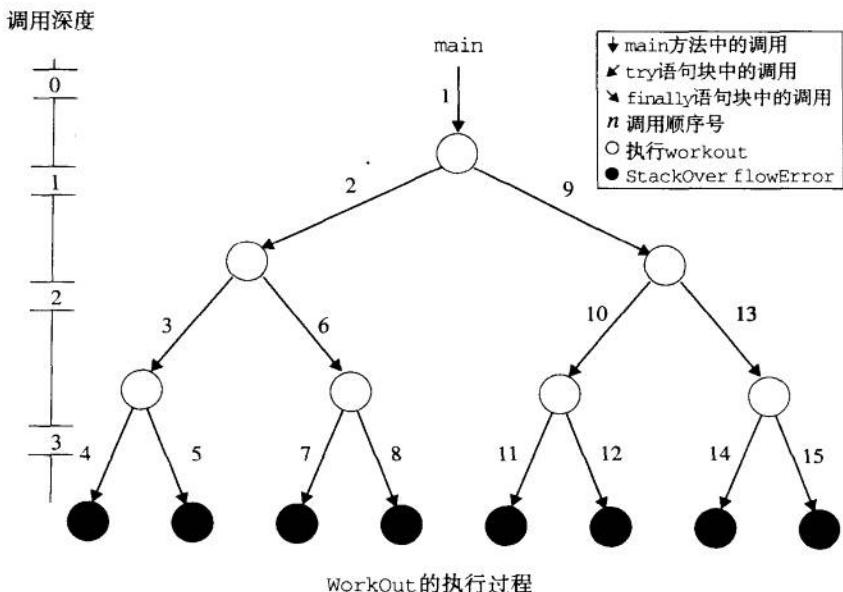
要不是有try-finally语句，该程序的行为将非常明显：workHard方法递归地调用它自身，直到程序抛出StackOverflowError异常，在此刻它以这个未捕获的异常而终止。但是，try-finally语句把事情复杂化了。当它试图抛出StackOverflowError时，程序将会在finally语句块的workHard方法中终止，这样，它就递归调用了自己。这看起来确实像一个无限循环的秘方，但是这个程序真的会无限循环下去吗？如果你运行它，它似乎确实是这么做的，但是确认的惟一方式就是分析它的行为。

Java虚拟机对栈的深度限制到了某个预设的水平。当超过这个水平时，VM就抛出StackOverflowError。为了能够更方便地考虑程序的行为，假设栈的深度为3，这比它实际的深度要小得多。现在跟踪其执行过程。

main方法调用workHard，而它又从其try语句块中递归地调用了自己，然后再一次从其try语句块中调用自己。此时，栈的深度是3。当workHard方法试图从其try语句块中再次调用自己时，该调用立即就会以StackOverflowError而失败。这个错误在最内部的finally语句块中被捕获，在此处栈的深度已经达到3了。在那里，workHard方法试图递归地调用它自己，但是该调用却以StackOverflowError而失败。这个错误将在上一级的finally语句块中被捕获，在此处栈的深度是2。该finally中的调用将与相对应的try语句块具有相同的行为：最终都会产生一个StackOverflowError。这似乎形成了一种模式，而事实也确实如此。

WorkOut的运行过程下图所示。图中，对workHard的调用用箭头表示，workHard的执行用圆圈表示。所有的调用除了第一个之外，都是递归的。会立即产生StackOverflowError异常的调用由灰色圆圈前导的箭头表示，try语句块中的调用以向左边的向下箭头表示，finally语句块中的调用以向右边的向下箭头表示。箭头上的数字描述了调用的顺序。

这张图展示了一个深度为0的调用（即main中的调用），两个深度为1的调用，四个深度为2的调用，和八个深度为3的调用，总共是十五个调用。那八个深度为3的调用每一个都会立即产生StackOverflowError。至少在将栈的深度限制为3的VM上，该程序不会是一个无限循环：它在十五个调用和八个异常之后就会终止。但是对于真实的VM又会怎样呢？它仍然不会是一个无限循环。其调用图与下图相似，只不过要大很多而已。



那么，究竟大到什么程度呢？有一个快速的试验表明许多VM都将栈的深度限制为1 024，因此，调用的数量就是 $1+2+4+8+\dots+2^{1024}=2^{1025}-1$ ，而抛出的异常数量是 2^{1024} 。假设我们的机器可以每秒执行 10^{10} 个调用，并每秒产生 10^{10} 个异常，按照当前的标准，这个假设的数量已经相当高了。在这样的假设条件下，程序将在大约 1.7×10^{29} 年后终止。为了让你对这个时间有直观的概念，告诉你，太阳的生命周期大约是 10^{10} 年，所以可以很确定，我们中没有任何人能够看到这个程序终止的时刻。尽管它不是一个无限循环，但是它也算是一个无限循环吧。

从技术角度讲，调用图是一棵完全二叉树，它的深度就是VM栈深度的上限。WorkOut程序的执行过程等价于先序遍历这棵树。在先序遍历中，程序先访问一个节点，然后递归地访问它的左子树和右子树。对于树中的每一条边，都会产生一个调用，而对于树中的每一个节点，都会抛出一个异常。

本谜题没有很多关于教训方面的东西。它证明了指数算法对于除了最小输入之外的所有情况都是不可行的，它还表明了你甚至可以不费什么劲地编写一个指数算法。

类之谜

在本章所描述的谜题都与类及其实例、方法和域相关。

谜题46：令人混淆的构造器案例

本谜题呈现了两个容易令人混淆的构造器。main方法调用了一个构造器，但是它调用的究竟是哪一个呢？该程序的输出取决于这个问题的答案。那么它会打印什么呢？甚至它是否合法？

```
public class Confusing {
    private Confusing(Object o) {
        System.out.println("Object");
    }

    private Confusing(double[] dArray) {
        System.out.println("double array");
    }

    public static void main(String[] args) {
        new Confusing(null);
    }
}
```

解惑46：令人混淆的构造器案例

传递给构造器的参数是一个空的对象引用，因此，初看起来，该程序好像应该调用参数类型为Object的重载版本，并且将打印Object。另一方面，数组也是引用类型，因此null也可以应用于类型为double[]的重载版本。你由此可能会得出结论：这个调用是模棱两可的，该程序应该不能编译。如果你试着去运行该程序，就会发现这些直观感觉都是不对的：该程序打印的是double array。这种行为可能显得有悖常理，但是有一个很好的理由可以解释它。

Java的重载解析过程是分两阶段运行的。第一阶段选取所有可获得并且可应用的方法或构造器。第二阶段在第一阶段选取的方法或构造器中选取最精确的一个。如果一个方法或构造器可以接受传递给另一个方法或构造器的任何参数，那么我们就说第一个方法比第二个方法缺乏精确性[JLS 15.12.2.5]。

在我们的程序中，两个构造器都是可获得并且可应用的。构造器Confusing(Object)可以接受任何传递给Confusing(double[])的参数，因此Confusing(Object)相对缺乏精确性。（每一个double数组都是一个Object，但是每一个Object并不一定是一个double数组。）因此，最精确的构造器就是Confusing(double[])，这也就解释了为什么程序会产生这样的输出。

如果传递的是一个double[]类型的值，那么这种行为是有意义的；但是如果你传递的是null，这种行为就有违直觉了。理解本谜题的关键在于在测试哪一个方法或构造器最精确时，这些测试没有使用实参：即出现在调用中的参数。这些参数只是被用来确定哪一个重载版本是可应用的。一旦编译器确定了哪些重载版本是可获得且可应用的，它就会选择最精确的一个重载版本，而此时使用的仅仅是形参：即出现在声明中的参数。

要想用一个null参数来调用Confusing(Object)构造器，你需要这样写代码：`new Confusing((Object)null)`。这可以确保只有Confusing(Object)是可应用的。更一般地讲，要想强制要求编译器选择一个精确的重载版本，需要将实参转型为形参所声明的类型。

以这种方式来在多个重载版本中进行选择是相当令人不快的。在你的API中，应该确保不会让客户端走这种极端。理想状态下，你应该避免使用重载：为不同的方法取不同的名称。当然，有时候这无法实现，例如，构造器就没有名称，因而也就无法被赋予不同的名称。然而，你可以通过将构造器设置为私有的并提供公有的静态工厂，以此来缓解这个问题[EJ Item 1]。如果构造器有许多参数，你可以用Builder模式[Gamma95]来减少对重载版本的需求量。

如果你确实进行了重载，那么请确保所有的重载版本所接受的参数类型都互不兼容，这样，任何两个重载版本都不会同时是可应用的。如果做不到这一点，那么就请确保所有可应用的重载版本都具有相同的行为[EJ Item 26]。

总之，重载版本的解析可能会产生混淆。应该尽可能地避免重载，如果你必须进行重载，那么你必须遵守上述方针，以最小化这种混淆。如果一个设计糟糕的API强制你在不同的重载版本之间进行选择，那么请将实参转型为你希望调用的重载版本的形参所具有的类型。

谜题47：啊呀！狸猫变犬子

下面的程序使用了一个Counter类来跟踪每一种家庭宠物叫唤的次数。那么该程序会打印出什么呢？

```
class Counter {
    private static int count;
    public static void increment() { count++; }
    public static int getCount() { return count; }
}

class Dog extends Counter {
    public Dog() { }
    public void woof() { increment(); }
}

class Cat extends Counter {
    public Cat() { }
    public void meow() { increment(); }
}

public class Ruckus {
    public static void main(String[] args) {
        Dog dogs[] = { new Dog(), new Dog() };
        for (int i = 0; i < dogs.length; i++)
            dogs[i].woof();
        Cat cats[] = { new Cat(), new Cat(), new Cat() };
        for (int i = 0; i < cats.length; i++)
            cats[i].meow();
        System.out.print(Dog.getCount() + " woofs and ");
        System.out.println(Cat.getCount() + " meows");
    }
}
```

解惑47：啊呀！狸猫变犬子

我们听到两声狗叫和三声猫叫——肯定是好一阵喧闹，因此，程序应该打印2 woofs and 3 meows，不是吗？不：它打印的是5 woofs and 5 meows。所有这些多出来的吵闹声是从哪里来的？我们做些什么才能够阻止它？

该程序打印出的犬吠声和猫叫声的数量之和是10，它是实际总数的两倍。问题在于Dog 和Cat 都从其共同的超类那里继承了count域，而count又是一个静态域。每一个静态域在声明它的类及其所有子类中共享一份单一的拷贝，因此Dog 和Cat 使用的是相同的count域。每一个对woof或meow的调用都在递增这个域，因此它被递增了5次。该程序分别通过调用Dog.getCount 和Cat.getCount 读取了这个域两次，在每一次读取时，都返回并打印了5。

在设计一个类的时候，如果该类构建于另一个类的行为之上，那么你有两种选择：一种是继承，即一个类扩展另一个类；另一种是组合，即在一个类中包含另一个类的一个实例。选择的依据是，一个类的每一个实例都是另一个类的一个实例，还是都有另一个类的一个实例。在第一种情况应该使用继承，而第二种情况应该使用组合。当你拿不准时，优选组合而不是继承[EJ Item 14]。

一条狗或是一只猫都不是一种计数器，因此使用继承是错误的。Dog 和Cat 不应该扩展Counter，而是应该都包含一个计数器域。每一种宠物都需要有一个计数器，但并非每一只宠物都需要有一个计数器，因此，这些计数器域应该是静态的。我们不必为Counter 类而感到烦恼；一个int 域就足够了。下面是我们重新设计过的程序，它会打印出我们所期望的2 woofs, 3 meows：

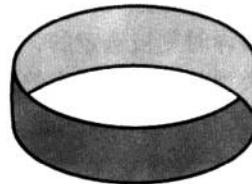
```
class Dog {
    private static int woofCounter;
    public Dog() { }
    public static int woofCount() { return woofCounter; }
    public void woof() { woofCounter++; }
}

class Cat {
    private static int meowCounter;
    public Cat() { }
    public static int meowCount() { return meowCounter; }
    public void meow() { meowCounter++; }
}
```

Ruckus类除了两行语句之外没有其他的变化，这两行语句被修改为使用新的方法名来访问计数器：

```
System.out.print(Dog.woofCount() + " woofs,");
System.out.println(Cat.meowCount() + " meows");
```

总之，静态域由声明它的类及其所有子类所共享。如果你需要让每一个子类都具有某个域的单独拷贝，那么你必须在每一个子类中声明一个单独的静态域。如果每一个实例都需要一个单独的拷贝，那么你可以在基类中声明一个非静态域。还有就是，要优选组合而不是继承，除非导出类真的是某一种基类。



谜题48：我所得到的都是静态的

下面的程序对巴辛吉小鬣狗和其他狗之间的行为差异进行了建模。如果你不知道什么是巴辛吉小鬣狗，那么我告诉你，这是一种产自非洲的小型卷尾狗，它们从来都不叫唤。那么，这个程序将打印出什么呢？

```
class Dog {
    public static void bark() {
        System.out.print("woof ");
    }
}

class Basenji extends Dog {
    public static void bark() { }
}

public class Bark {
    public static void main(String args[]) {
        Dog woof = new Dog();
        Dog nipper = new Basenji();
        woof.bark();
        nipper.bark();
    }
}
```

解惑48：我所得到的都是静态的

随意地看一看，好像该程序应该只打印woof。毕竟，Basenji扩展自Dog，并且它的bark方法定义成什么也不做。`main`方法调用了`bark`方法，第一次是在`Dog`类型的`woofier`上调用，第二次是在`Basenji`类型的`nipper`上调用。巴辛吉小瓢狗并不会叫唤，但是很显然，这一只会。如果你运行该程序，就会发现它打印的是woof woof。这可可怜的小家伙到底出什么问题了？

本谜题的题目已经给出了一个很显著的暗示。问题在于`bark`是一个静态方法，而对静态方法的调用不存在任何动态的分派机制[JLS 15.12.4.4]。当一个程序调用了一个静态方法时，要被调用的方法都是在编译时刻被选定的，而这种选定是基于修饰符的编译期类型而做出的，修饰符的编译期类型就是我们给出的方法调用表达式中圆点左边部分的名字。在本案中，两个方法调用的修饰符分别是变量`woofier`和`nipper`，它们都被声明为`Dog`类型。因为它们具有相同的编译期类型，所以编译器使得它们调用的是相同的方法：`Dog.bark`。这也就解释了为什么程序打印出woof woof。尽管`nipper`的运行期类型是`Basenji`，但是编译器只会考虑其编译器类型。

要改正这个程序，直接从两个`bark`方法定义中删除`static`修饰符即可。这样，`Basenji`中的`bark`方法将覆写而不是隐藏`Dog`中的`bark`方法，而该程序也将会打印出woof，而不是woof woof。通过覆写，你可以获得动态的分派；而通过隐藏，你却得不到这种特性。

当你调用了一个静态方法时，通常都是用一个类而不是表达式来标识它：例如，`Dog.bark`或`Basenji.bark`。当你在阅读一个Java程序时，你会期望类被用作静态方法的修饰符，这些静态方法都是被静态分派的，而表达式被用作实例方法的修饰符，这些实例方法都是被动态分派的。通过耦合类和变量的不同的命名规范，我们可以提供一个很强的可视化线索，用来表明一个给定的方法调用是动态的还是静态的。本谜题的程序使用了一个表达式作为静态方法调用的修饰符，这就误导了我们。千万不要用一个表达式来标识一个静态方法调用。

覆写的使用与上述的混乱局面搅到了一起。`Basenji`中的`bark`方法与`Dog`中的`bark`方法具有相同的方法签名，这正是覆写的惯用方式，预示着要进行动态的分派。然而在本例中，该方法被声明为`static`，而静态方法是不能被覆写的；它们只能被隐藏，而这仅仅是因为你没有表达出应该表达的意思。为了避免这样的混乱，千万不要隐藏静态方法。即便在子类中重用了超类中的静态方法的名称，也不会给你带来任何新的东西，但是却会丧失很多东西。

对语言设计者的教训是：对类和实例方法的调用彼此之间看起来应该具有明显的差异。第一种实现此目标的方式是不允许使用表达式作为静态方法的修饰符；第二种区分静态方法和实例方法调用的方式是使用不同的操作符，就像C++那样；第三种方式是通过完全抛弃静态方法这一概念来解决此问题，就像Smalltalk那样。

总之，要用类名来修饰静态方法的调用，或者当你在静态方法所属的类中去调用它们时，根本不修饰这些方法，但是千万不要用一个表达式去修饰它们。还有就是要避免隐藏静态方法。所有这些原则合起来就可以帮助我们消除那些容易令人误解的覆写，这些覆写对静态方法进行了动态分派。

谜题49：比生命更大

担心你误会本书尽讨论阿猫阿狗的事，我们来说说王者。假如小报是可信的，那么摇滚之王“猫王”就仍然在世。这里说的并不是由他众多的模仿者中的某一位模仿出来的猫王，而是Elvis本人。下面的程序用来估算猫王当前的腰带尺寸，方法是根据在公开演出中所观察到的他的体态发展趋势来进行投射。该程序中使用了`Calendar.getInstance().get(Calendar.YEAR)`这个惯用法，它返回当前的日历年份。那么，该程序会打印出什么呢？

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private final int beltSize;
    private static final int CURRENT_YEAR =
        Calendar.getInstance().get(Calendar.YEAR);

    private Elvis() {
        beltSize = CURRENT_YEAR - 1930;
    }

    public int beltSize() {
        return beltSize;
    }

    public static void main(String[] args) {
        System.out.println("Elvis wears a size " +
                           INSTANCE.beltSize() + " belt.");
    }
}
```

解惑49：比生命更大

第一眼看去，这个程序是在计算当前的年份减去1930的值。如果它是正确的，那么在2006年，该程序将打印出Elvis wears a size 76 belt。如果你尝试着去运行该程序，你就会了解到小报是错误的，这证明你不能相信在报纸上读到的任何东西。该程序将打印出Elvis wears a size -1930 belt。也许猫王已经在反物质的宇宙中定居了。

该程序所遇到的问题是由类初始化顺序中的循环而引起的[JLS 12.4]。让我们来看看其细节。Elvis类的初始化是由虚拟机对其main方法的调用而触发的。首先，其静态域被设置为缺省值[JLS 4.12.5]，其中INSTANCE域被设置为null，CURRENT_YEAR被设置为0。接下来，静态域初始器按照其出现的顺序执行。第一个静态域是INSTANCE，它的值是通过调用Elvis()构造器而计算出来的。

这个构造器会用一个涉及静态域CURRENT_YEAR的表达式来初始化beltSize。通常，读取一个静态域是引起一个类被初始化的事件之一，但是我们已经在初始化Elvis类了。递归的初始化尝试会直接被忽略掉[JLS 12.4.2, 第3步]。因此，CURRENT_YEAR的值仍旧是其缺省值0。这就是为什么Elvis的腰带尺寸变成了-1930的原因。

最后，从构造器返回以完成Elvis类的初始化，假设我们是在2006年运行该程序，那么我们就将静态域CURRENT_YEAR初始化成了2006。遗憾的是，这个域现在所具有的正确值对于影响Elvis.INSTANCE.beltSize的计算来说已经太晚了，beltSize的值已经是-1930了。这正是后续所有对Elvis.INSTANCE.beltSize()的调用将返回的值。

该程序表明，在final类型的静态域被初始化之前，存在着读取其值的可能，而此时该静态域包含的还只是其所属类型的缺省值。这是与直觉相违背的，因为我们通常会将final类型的域看作常量。final类型的域只有在其初始化表达式是常量表达式时才是常量[JLS 15.28]。

由类初始化中的循环所引发的问题是难以诊断的，但是一旦被诊断到，通常是很容易改正的。要想改正一个类初始化循环，需要重新对静态域的初始器进行排序，使得每一个初始器都出现在任何依赖于它的初始器之前。在这个程序中，CURRENT_YEAR的声明属于在INSTANCE声明之前的情况，因为Elvis实例的创建需要CURRENT_YEAR被初始化。一旦CURRENT_YEAR的声明被移走，Elvis就真的比生命更大了。

某些通用的设计模式本质上就是初始化循环的，特别是本谜题所展示的单例模式(Singleton) [Gamma95]和服务提供者框架(Service Provider Framework) [EJ Item 1]。类型安全的枚举模式(Typesafe Enum pattern) [EJ Item 21]也会引起类初始化的循环。

5.0版本添加了对这种使用枚举类型的模式的语言级支持。为了减少问题发生的可能性，对枚举类型的静态初始器做了一些限制[JLS 16.5, 8.9]。

总之，要当心类初始化循环。最简单的循环只涉及一个单一的类，但是它们也可能涉及多个类。类初始化循环也并非总是坏事，但是它们可能会导致在静态域被初始化之前就调用构造器。静态域，甚至是final类型的静态域，可能会在它们被初始化之前，被读走其缺省值。

谜题50：不是你的类型

本谜题要测试你对Java的两个最经典的操作符：`instanceof`和转型的理解程度。下面的三个程序中的每一个都会做些什么呢？

```
public class Type1 {
    public static void main(String[] args) {
        String s = null;
        System.out.println(s instanceof String);
    }
}

public class Type2 {
    public static void main(String[] args) {
        System.out.println(new Type2() instanceof String);
    }
}

public class Type3 {
    public static void main(String args[]) {
        Type3 t3 = (Type3) new Object();
    }
}
```

解惑50：不是你的类型

第一个程序，Type1，展示了`instanceof`操作符应用于一个空对象引用时的行为。尽管`null`对于每一个引用类型来说都是其子类型，但是`instanceof`操作符被定义为在其左操作数为`null`时返回`false`。因此，Type1将打印`false`。这被证明是实践中非常有用的行为。如果`instanceof`告诉你一个对象引用是某个特定类型的实例，那么你就可以将其转型为该类型，并调用该类型的方法，而不用担心会抛出`ClassCastException`或`NullPointerException`异常。

第二个程序，Type2，说明了`instanceof`操作符，在测试一个类的实例，以查看它是否是某个不相关的类的实例时，所表现出来的行为。你可能会期望该程序打印出`false`。毕竟，Type2的实例不是`String`的实例，因此该测试应该失败，对吗？不，`instanceof`测试在编译时刻就失败了，我们只能得到下面这样的出错消息：

```
Type2.java: incompatible types
found : Type2, required: java.lang.String
    System.out.println(new Type2() instanceof String);
^
```

该程序编译失败是因为`instanceof`操作符有这样的要求：如果两个操作数的类型都是类，其中一个必须是另一个的子类型[JLS 15.20.2, 15.16, 5.5]。Type2和`String`彼此都不是对方的子类型，所以`instanceof`测试将导致编译期错误。这个错误有助于让你警惕`instanceof`测试，它们可能并没有去做你希望它们做的事情。

第三个程序，Type3，说明了当要被转型的表达式的静态类型是转型类型的超类时，转型操作符的行为。与`instanceof`操作相同，如果在一个转型操作中的两种类型都是类，那么其中一个必须是另一个的子类型。尽管对我们来说，这个转型很显然会失败，但是类型系统还没有强大到能够洞悉表达式`new Object()`的运行期类型不可能是Type3的一个子类型。因此，该程序将在运行期抛出`ClassCastException`异常。这有一点违背直觉：第二个程序完全具有实际意义，但是却不能编译；而这个程序没有任何实际意义，但是却可以编译。

总之，第一个程序说明了`instanceof`运行期行为的一个很有用的冷僻案例。第二个程序展示了其编译期行为的一个很有用的冷僻案例。第三个程序展示了转型操作符行为的一个冷僻案例，在此案例中，编译器并不能将你从所做的荒唐事中搭救出来，只能靠VM在运行期来帮你绷紧这根弦。

谜题51：要点何在

下面这个程序有两个不可变的值类（value class），值类即其实例表示值的类。第一个类用整数坐标来表示平面上的一个点，第二个类在此基础上添加了一点颜色。主程序将创建和打印第二个类的一个实例。那么，下面的程序将打印出什么呢？

```
class Point {  
    private final int x, y;  
    private final String name; // Cached at construction time  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
        name = makeName();  
    }  
  
    protected String makeName() {  
        return "[" + x + "," + y + "]";  
    }  
  
    public final String toString() {  
        return name;  
    }  
}  
  
public class ColorPoint extends Point {  
    private final String color;  
    ColorPoint(int x, int y, String color) {  
        super(x, y);  
        this.color = color;  
    }  
  
    protected String makeName() {  
        return super.makeName() + ":" + color;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(new ColorPoint(4, 2, "purple"));  
    }  
}
```

解惑51：要点何在

main方法创建并打印了一个ColorPoint实例。println方法调用了该ColorPoint实例的toString方法，这个方法是在Point中定义的。toString方法将直接返回name域的值，这个值是通过调用makeName方法在Point的构造器中初始化的。对于一个Point实例来说，makeName方法将返回[x,y]形式的字符串。对于一个ColorPoint实例来说，makeName方法被覆写为返回[x,y]:color形式的字符串。在本例中，x是4，y是2，color是purple，因此程序将打印[4,2]:purple，对吗？不，如果你运行该程序，就会发现它打印的是[4,2]:null。这个程序出什么问题了呢？

这个程序遇到了实例初始化顺序这一问题。要理解该程序，我们就需要详细跟踪该程序的执行过程。下面是注释过的程序清单，用来引导我们了解其执行顺序：

```

class Point {
    protected final int x, y;
    private final String name;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
        name = makeName();      // (3). Invoke subclass method
    }

    protected String makeName() {
        return "[" + x + "," + y + "]";
    }

    public final String toString() {
        return name;
    }
}

public class ColorPoint extends Point {
    private final String color;

    ColorPoint(int x, int y, String color) {
        super(x, y);           // (2) Chain to Point constructor
        this.color = color;   // (5) Initialize blank final-Too late
    }
}

```

```

protected String makeName() {
    // (4) Executes before subclass constructor body!
    return super.makeName() + ":" + color;
}

public static void main(String[] args) {
    // (1) Invoke subclass constructor
    System.out.println(new ColorPoint(4, 2, "purple"));
}
}

```

在下面的解释中，括号中的数字引用的就是在上述注释过的程序中的注释标号。首先，程序通过调用ColorPoint构造器创建了一个ColorPoint实例（1）。这个构造器以链接调用其超类构造器开始，就像所有构造器所做的那样（2）。超类构造器在构造过程中对该对象的x域赋值为4，对y域赋值为2。然后该超类构造器调用makeName，该方法被子类覆写了（3）。

ColorPoint中的makeName方法（4）是在ColorPoint构造器的程序体之前执行的，这就是问题的核心所在。makeName方法首先调用super.makeName，它将返回我们所期望的[4,2]，然后该方法在此基础上追加字符串“：“和由color域的值转换的字符串。但是此刻color域的值是什么呢？由于它仍处于待初始化状态，所以它的值仍旧是缺省值null。因此，makeName方法返回的是字符串“[4,2]:null”。超类构造器将这个值赋给name域（3），然后将控制流返回给子类的构造器。

这之后子类构造器才将“purple”赋予color域（5），但是此刻已经为时过晚了。color域已经在超类中被用来初始化name域了，并且产生了不正确的值。之后，子类构造器返回，新创建的ColorPoint实例被传递给println方法，它适时地调用了该实例的toString方法，这个方法返回的是该实例的name域的内容，即“[4,2]:null”，这也就成为了程序要打印的东西。

本谜题说明：在一个final类型的实例域被赋值之前，存在着取用其值的可能，而此时它包含的仍旧是其所属类型的缺省值。在某种意义上，本谜题是谜题49在实例方面的相似物，谜题49是在final类型的静态域被赋值之前，取用了它的值。在这两种情况中，谜题都是因初始化的循环而产生的，在谜题49中，是类的初始化；而在本谜题中，是实例初始化。两种情况都存在着产生极大混乱的可能性，但是它们之间有一个重要的差别：循环的类初始化是无法避免的灾难，但是循环的实例初始化是可以且总是应该避免的。

无论何时，只要一个构造器调用了一个已经被其子类覆写了的方法，那么该问题就

会出现，因为以这种方式被调用的方法总是在实例初始化之前执行。要想避免这个问题，就千万不要在构造器中调用可覆盖的方法，直接调用或间接调用都不行[EJ Item 15]。这项禁令应该扩展至实例初始器和伪构造器（pseudoconstructor）`readObject`与`clone`。（这些方法之所以被称为伪构造器，是因为它们可以在不调用构造器的情况下创建对象。）

你可以通过惰性初始化`name`域来修正该问题，即当它第一次被使用时初始化，以此取代积极初始化，即当`Point`实例被创建时初始化。通过这种修改，该程序就可以打印出我们所期望的`[4, 2]:purple`。

```
class Point {
    protected final int x, y;
    private String name; // Lazily initialized

    Point(int x, int y) {
        this.x = x;
        this.y = y;
        // name initialization removed
    }

    protected String makeName() {
        return "[" + x + "," + y + "]";
    }

    // Lazily computes and caches name on first use
    public final synchronized String toString() {
        if (name == null)
            name = makeName();
        return name;
    }
}
```

尽管惰性加载可以修正这个问题，但是对于让一个值类去扩展另一个值类，并且在其上添加一个会对`equals`比较方法产生影响的域来说，这种做法仍旧不是一个好主意。你无法在超类和子类上都提供一个基于值的`equals`方法，而同时又不违反`Object.equals`方法的通用约定，或者不消除在超类和子类之间进行有实际意义的比较操作的可能性 [EJ Item 7]。

循环实例初始化问题对语言设计者来说是问题成堆的地方。`C++`是通过在构造阶段将对象的类型从超类类型改变为子类类型来解决这个问题的。如果采用这种解决方法，

本谜题中最初的程序将打印 [4, 2]。我们发现没有任何一种流行的语言能够令人满意地解决这个问题。也许，值得去考虑，当超类构造器调用子类方法时，通过抛出一个不受检查的异常使循环实例初始化非法。

总之，在任何情况下，你都务必要记住：不要在构造器中调用可覆写的方法。在实例初始化中产生的循环将是致命的。该问题的解决方案就是惰性初始化[EJ Items 13,48]。

谜题52：总和的玩笑

下面的程序在一个类中计算并缓存了一个总和，并且在另一个类中打印了这个总和。那么，这个程序将打印出什么呢？这里给一点提示：你可能已经回忆起来了，在代数中我们曾经学过，从整数1到 n 的总和是 $n(n+1)/2$ 。

```
class Cache {
    static {
        initializeIfNecessary();
    }

    private static int sum;
    public static int getSum() {
        initializeIfNecessary();
        return sum;
    }

    private static boolean initialized = false;
    private static synchronized void initializeIfNecessary() {
        if (!initialized) {
            for (int i = 0; i < 100; i++)
                sum += i;
            initialized = true;
        }
    }
}

public class Client {
    public static void main(String[] args) {
        System.out.println(Cache.getSum());
    }
}
```

解惑52：总和的玩笑

草草地看一遍，你可能会认为这个程序从1加到了100，但实际上它并没有这么做。再稍微仔细地看一看那个循环，它是一个典型的半开循环，因此它将从0循环到99。有了这个印象之后，你可能会认为这个程序打印的是从0到99的整数总和。用前面提示中给出的公式，我们知道这个总和是 $99 \times 100/2$ ，即4 950。但是，这个程序可不这么想，它打印的是9 900，是预期值的整整两倍。是什么导致它如此热情地翻倍计算了这个总和呢？

该程序的编写者显然在确保sum在使用前就已初始化这个问题上，遇到了众多的麻烦。该程序结合了惰性初始化和积极初始化，甚至还用上了同步，以确保缓存在多线程环境下也能工作。看起来这个程序已经把所有的问题都考虑了，但是它仍然不能正常工作。它到底出了什么问题呢？

与谜题49中的程序一样，该程序受到了类初始化顺序问题的影响。为了理解其行为，我们来跟踪其执行过程。在可以调用Client.main之前，VM必须初始化Client类。这项初始化工作异常简单，我们就不多说什么了。Client.main方法调用了Cache.getSum方法，在getSum方法可以执行之前，VM必须初始化Cache类。

回想一下，类初始化是按照静态初始器在源代码中出现的顺序去执行这些初始器的。Cache类有两个静态初始器：在类顶端的一个static语句块，以及静态域initialized的初始化。静态语句块是先出现的，它调用了方法initializeIfNecessary，该方法将测试initialized域。因为该域还没有被赋予任何值，所以它具有缺省的布尔值false。与此类似，sum具有缺省的int值0。因此，initializeIfNecessary方法执行的正是你所期望的行为，将4 950添加到了sum上，并将initialized设置为true。

在静态语句块执行之后，initialized域的静态初始器将其设置回false，从而完成Cache的类初始化。遗憾的是，sum现在包含的是正确的缓存值，但是initialized包含的却是false：Cache类的两个关键状态并未同步。

此后，Client类的main方法调用Cache.getSum方法，它将再次调用initializeIfNecessary方法。因为initialized标志是false，所以initializeIfNecessary方法将进入其循环，该循环将把另一个4 950添加到sum上，从而使其值增加到了9 900。getSum方法返回的就是这个值，而程序打印的也是它。

很明显，该程序的编写者认为Cache类的初始化不会以这种顺序发生。由于不能在惰性初始化和积极初始化之间作出抉择，所以编写者同时运用这二者，结果产生了大麻烦。要么使用积极初始化，要么使用惰性初始化，千万不要同时使用二者。

如果初始化一个域的时间和空间代价比较低，或者该域在程序的每一次执行中都需要用到时，那么使用积极初始化是恰当的。如果其代价比较高，或者该域在某些执行中并不会被用到，那么惰性初始化可能是更好的选择[EJ Item 48]。另外，惰性初始化对于打破类或实例初始化中的循环也可能是必需的（谜题51）。

通过重排静态初始化的顺序，使得initialized域在sum被初始化之后不被复位到false，或者通过移除initialized域的显式静态初始化操作，Cache类就可以得到修复。尽管这样所产生的程序可以工作，但是它们仍旧是混乱的和病构的。Cache类应该被重写为使用积极初始化，这样产生的版本很明显是正确的，而且比最初的版本更加简单。使用这个版本的Cache类，程序就可以打印出我们所期望的4 950：

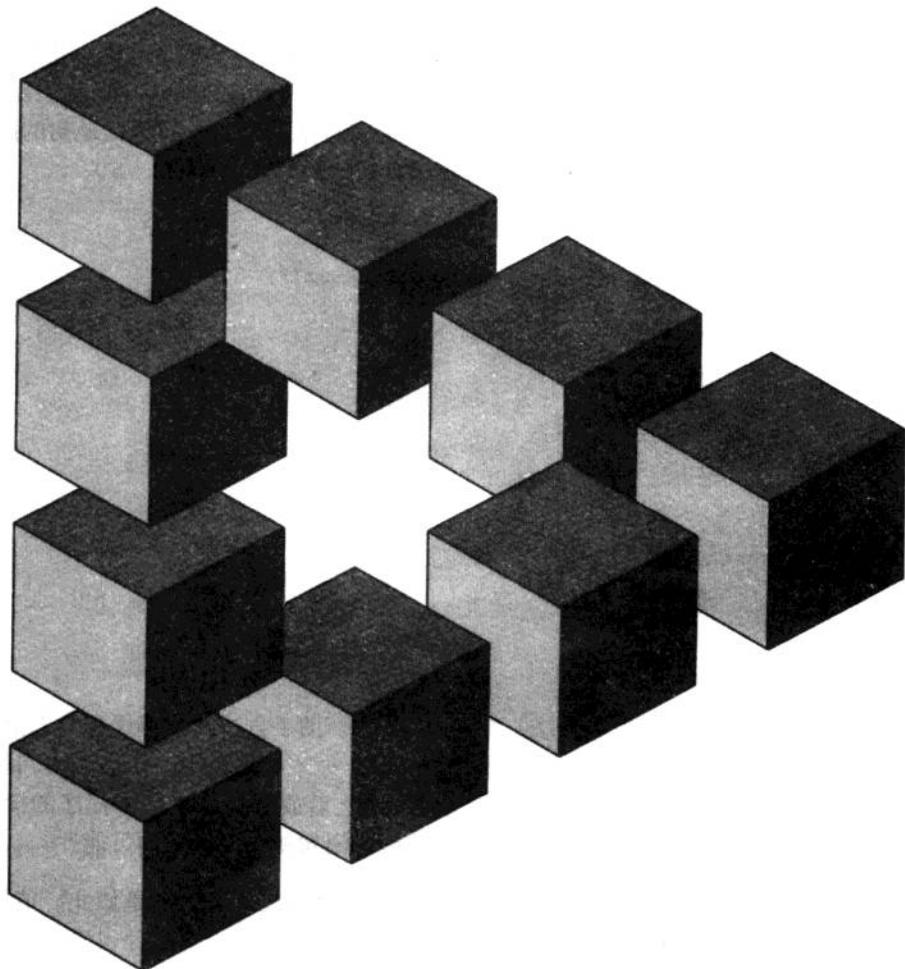
```
class Cache {
    private static final int sum = computeSum();

    private static int computeSum() {
        int result = 0;
        for (int i = 0; i < 100; i++)
            result += i;
        return result;
    }

    public static int getSum() {
        return sum;
    }
}
```

请注意，我们使用了一个助手方法来初始化sum。助手方法通常都优于静态语句块，因为它让你可以对计算命名。只有在极少的情况下，你才必须使用一个静态语句块来初始化一个静态域，此时请将该语句块紧随该域声明之后放置。这提高了程序的清晰度，并且消除了像最初的程序中出现的静态初始化与静态语句块互相竞争的可能性。

总之，请考虑类初始化的顺序，特别是当初始化显得很重要时更是如此。请你执行测试，以确保类初始化序列的简洁。请使用积极初始化，除非你有某种很好的理由要使用惰性初始化，例如性能方面的因素，或者需要打破初始化循环。



谜题53：做你的事吧

现在该轮到你写一些代码了。假设你有一个称为Thing的库类，它惟一的构造器将接受一个int参数：

```
public class Thing {
    public Thing(int i) { ..... }
    .....
}
```

Thing实例没有提供任何可以获取其构造器参数值的途径。因为Thing是一个库类，所以你不具有访问其内部的权限，因此你不能修改它。

假设你想编写一个称为MyThing的子类，其构造器将通过调用SomeOtherClass.func()方法来计算超类构造器的参数。这个方法返回的值被一个个的调用以不可预知的方式所修改。最后，假设你想将这个曾经传递给超类构造器的值存储到子类的一个final实例域中，以供将来使用。那么下面就是你自然会写出的代码：

```
public class MyThing extends Thing {
    private final int arg;

    public MyThing() {
        super(arg = SomeOtherClass.func());
        .....
    }
    .....
}
```

遗憾的是，这个程序是非法的。如果你尝试着去编译它，那么你将得到一条像下面这样的错误消息：

```
MyThing.java:
can't reference arg before supertype constructor has been called
super(arg = SomeOtherClass.func());
^
```

你怎样才能重写MyThing以实现想要的效果呢？MyThing()构造器必须是线程安全的：多个线程可能会并发地调用它。

解惑53：做你的事吧

你可能会尝试在调用Thing构造器之前，将SomeOtherClass.func()调用的结果存储在一个静态域中。这种解决方案是可运行的，但却是笨拙的。为了实现线程安全，你必须同步对被存储值的访问控制，这需要对源代码进行无法想像的修改。这些修改中的某些可以通过使用一个线程本地的静态域（java.util.ThreadLocal）而得以避免，但是还存在着另一个好得多的解决方案。

这个更好的解决方案内在地线程安全和优雅，它涉及对MyThing中第二个私有构造器的运用：

```
public class MyThing extends Thing {
    private final int arg;

    public MyThing() {
        this(SomeOtherClass.func());
    }

    private MyThing(int i) {
        super(i);
        arg = i;
    }
}
```

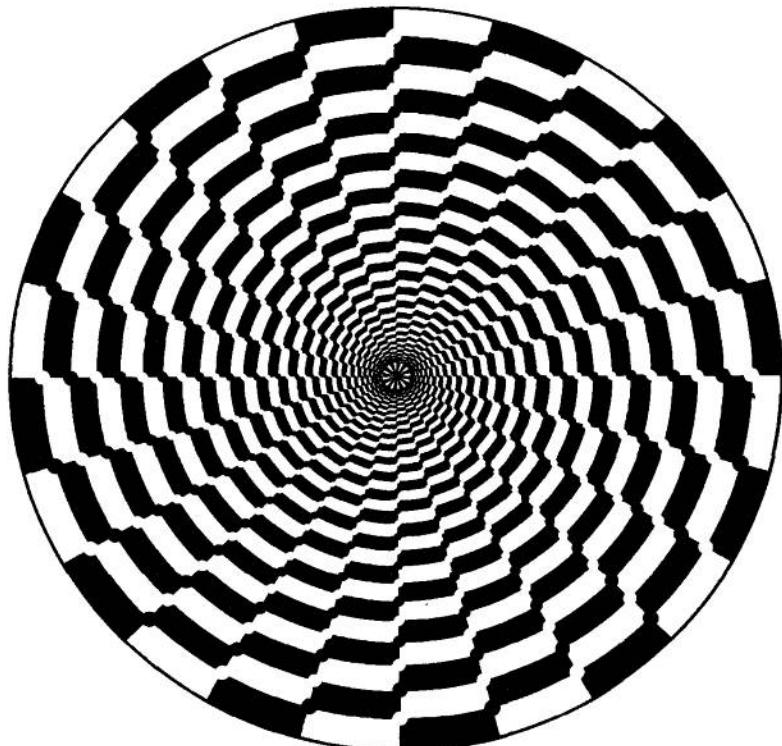
这个解决方案使用了交替构造器调用机制（alternate constructor invocation）[JLS 8.8.7.1]。这个特征允许一个类中的某个构造器链接调用同一个类中的另一个构造器。在本例中，MyThing()链接调用了私有构造器MyThing(int)，它执行了所需的实例初始化。在这个私有构造器中，表达式SomeOtherClass.func()的值已经被捕获到了变量i中，并且它可以在超类构造器返回之后存储到final类型的域param中。

通过本谜题所说明的私有构造器捕获（Private Constructor Capture）惯用法是一种非常有用的模式，你应该把它添加到你的技巧库中。我们已经看到了某些真的是很丑陋的代码，它们本来是可以通过使用本模式而避免如此丑陋的。

谜题54：Null与Void

下面仍然是经典的Hello World程序的另一个变种。那么，这个变种将打印什么呢？

```
public class Null {  
    public static void greet() {  
        System.out.println("Hello world!");  
    }  
  
    public static void main(String[] args) {  
        ((Null) null).greet();  
    }  
}
```



解惑54：Null与Void

这个程序看起来似乎应该抛出`NullPointerException`异常，因为其`main`方法是在常量`null`上调用`greet`方法，而你是不可以在`null`上调用方法的，对吗？嗯，某些时候是可以的。如果你运行该程序，就会发现它打印出了“Hello world！”

理解本谜题的关键是`Null.greet`是一个静态方法。正如你在谜题48中所看到的，在静态方法的调用中，使用表达式作为其限定符并非是一个好主意，而这也正是问题之所在。不仅表达式的值所引用的对象的运行期类型在确定哪一个方法将被调用时并不起任何作用，而且如果对象有标识的话，其标识也不起任何作用。在本例中，没有任何对象，但是这并不会造成任何区别。**静态方法调用的限定表达式是可以计算的，但是它的值将被忽略。**没有任何要求其值为非空的限制。

要想消除该程序中的混乱，你可以用它的类作为限定符来调用`greet`方法：

```
public static void main(String[] args) {  
    Null.greet();  
}
```

然而更好的方式是完全消除限定符：

```
public static void main(String[] args) {  
    greet();  
}
```

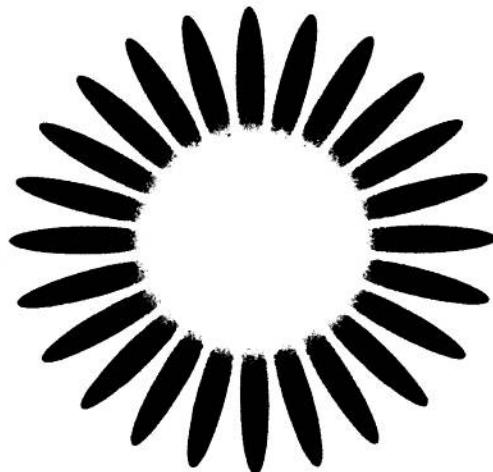
总之，本谜题的教训与谜题48的完全相同：要么用某种类型来限定静态方法调用，要么就根本不要限定它们。对语言设计者来说，应该不允许用表达式来污染静态方法调用的可能性存在，因为它们只会产生混乱。



谜题55：特创论

某些时候，对于一个类来说，跟踪其创建出来的实例个数会非常有用，其典型实现是通过让它的构造器递增一个私有静态域来完成的。在下面的程序中，Creature类示范了这种技巧，而Creator类对其进行操练，将打印出已经创建的Creature实例的数量。那么，这个程序会打印出什么呢？

```
public class Creator {  
    public static void main(String[] args) {  
        for (int i = 0; i < 100; i++)  
            Creature creature = new Creature();  
        System.out.println(Creature.numCreated());  
    }  
}  
  
class Creature {  
    private static long numCreated = 0;  
  
    public Creature() {  
        numCreated++;  
    }  
  
    public static long numCreated() {  
        return numCreated;  
    }  
}
```



解惑55：特创论

这是一个捉弄人的问题。该程序看起来似乎应该打印100，但是它没有打印任何东西，因为它根本就不能编译。如果你尝试着去编译它，你就会发现编译器的诊断信息基本没什么用处。下面就是javac打印的东西：

```
Creator.java:4: not a statement
    Creature creature = new Creature();
               ^
Compiler.java:4: ';' expected
    Creature creature = new Creature();
               ^
               ^
```

一个本地变量声明看起来像是一条语句，但是从技术上说，它不是；它应该是一个本地变量声明语句（local variable declaration statement）[JLS 14.4]。Java语言规范不允许一个本地变量声明语句作为一条语句在for、while或do循环中重复执行[JLS 14.12-14]。一个本地变量声明作为一条语句只能直接出现在一个语句块中。（一个语句块是由一对花括号以及包含在这对花括号中的语句和声明构成的。）

有两种方式可以改正这个问题。最显而易见的方式是将这个声明置于一个语句块中：

```
for (int i = 0; i < 100; i++) {
    Creature creature = new Creature();
}
```

然而，请注意，该程序没有使用本地变量creature。因此，将该声明用一个无任何修饰的构造器调用来替代将更具实际意义，这样可以强调对新创建对象的引用正在被丢弃：

```
for (int i = 0; i < 100; i++)
    new Creature();
```

无论我们做出了上面的哪种修改，该程序都将打印出我们所期望的100。

请注意，用于跟踪Creature实例个数的变量（numCreated）是long类型而不是int类型的。我们很容易想像到，一个程序创建出的某个类的实例可能会多于int数值的最大值，但是它不会多于long数值的最大值。int数值的最大值是 $2^{31}-1$ ，即大约 2.1×10^9 ，而long数值的最大值是 $2^{63}-1$ ，即大约 9.2×10^{18} 。当前，每秒钟创建 10^8 个对

象是可能的，这意味着一个程序在long类型的对象计数器会溢出之前，不得不运行大约三千年。即使是面对硬件速度的提高，long类型的对象计数器也应该足以应付可预见的未来。

还要注意的是，本谜题中的创建计数策略并不是线程安全的。如果多个线程可以并行地创建对象，那么递增计数器的代码和读取计数器的代码都应该被同步：

```
// Thread-safe creation counter
class Creature {
    private static long numCreated;

    public Creature() {
        synchronized (Creature.class) {
            numCreated++;
        }
    }

    public static synchronized long numCreated() {
        return numCreated;
    }
}
```

或者，如果你使用的是5.0或更新的版本，你可以使用一个AtomicLong实例，它在面临并发时可以绕过对同步的需求。

```
// Thread-safe creation counter using AtomicLong;
import java.util.concurrent.atomic.AtomicLong;

class Creature {
    private static AtomicLong numCreated = new AtomicLong();

    public Creature() {
        numCreated.incrementAndGet();
    }

    public static long numCreated() {
        return numCreated.get();
    }
}
```

请注意，把numCreated声明为瞬时的是不足以解决问题的，因为volatile修饰符可以保证其他线程将看到最近赋予该域的值，但是它不能进行原子性的递增操作。

总之，一个本地变量声明不能被用作for、while或do循环中的重复执行语句，它作为一条语句只能出现在一个语句块中。另外，在使用一个变量来对实例的创建进行计数时，要使用long类型而不是int类型的变量，以防止溢出。最后，如果你打算在多线程中创建实例，要么将对实例计数器的访问进行同步，要么使用一个AtomicLong类型的计数器。

库 之 谜

在本章所描述的谜题涉及与基类库相关的话题，例如Object方法、集合、Date和Calendar。

谜题56：大问题

作为一项热身活动，我们来测试一下你对BigInteger的了解程度。下面这个程序将打印出什么呢？

```
import java.math.BigInteger;
public class BigProblem {
    public static void main(String[] args) {
        BigInteger fiveThousand = new BigInteger("5000");
        BigInteger fiftyThousand = new BigInteger("50000");
        BigInteger fiveHundredThousand
            = new BigInteger("500000");
        BigInteger total = BigInteger.ZERO;
        total.add(fiveThousand);
        total.add(fiftyThousand);
        total.add(fiveHundredThousand);
        System.out.println(total);
    }
}
```

解惑56：大问题

你可能会认为这个程序会打印出555000。毕竟，它将total设置为用BigInteger表示的0，然后将5 000、50 000和500 000加到这个变量上。如果你运行该程序，你就会发现它打印的不是555000，而是0。很明显，所有这些加法对total没有产生任何影响。

对此有一个很好理由可以解释：**BigInteger实例是不可变的**。String、BigDecimal以及包装器类型：Integer、Long、Short、Byte、Character、Boolean、Float和Double也是如此。你不能修改它们的值。我们不能修改现有实例的值，对这些类型的操作将返回新的实例。最初，不可变类型看起来可能很不自然，但是它们具有很多胜过与其相对应的可变类型的优势。不可变类型更容易设计、实现和使用；它们出错的可能性更小，并且更加安全[EJ Item 13]。

为了在一个包含对不可变对象引用的变量上执行计算，我们需要将计算的结果赋值给该变量。这样做就会产生下面的程序，它将打印出我们所期望的555000：

```
import java.math.BigInteger;

public class BigProblem {
    public static void main(String[] args) throws Exception {
        BigInteger fiveThousand = new BigInteger("5000");
        BigInteger fiftyThousand = new BigInteger("50000");
        BigInteger fiveHundredThousand
            = new BigInteger("500000");

        BigInteger total = BigInteger.ZERO;
        total = total.add(fiveThousand);
        total = total.add(fiftyThousand);
        total = total.add(fiveHundredThousand);
        System.out.println(total);
    }
}
```

本谜题的教训是：不要被误导，认为不可变类型是可变的。这是在刚入门的Java程序员中很常见的错误。公正地说，Java不可变类型的某些方法名促使我们走上了歧途。像add、subtract和negate之类的名字似乎是在暗示这些方法将修改它们所调用的实例。也许plus、minus和negation才是更好的名字。

对API设计者来说，教训是：在命名不可变类型的方法时，应该优选介词和名词，而不是动词。介词适用于带有参数的方法，而名词适用于不带参数的方法。对语言设计者而言，教训与谜题2相同，那就是应该考虑对操作符重载提供有限的支持，这样算术操作符就可以作用于诸如BigInteger这样的数值型的引用类型。由此，即使是初学者也不会认为计算表达式total + fiveThousand将会对total的值产生任何影响。

谜题57：名字里有什么

下面的程序包含了一个简单的不可变类，它表示一个名字，其main方法将一个名字置于一个集合中，并检查该集合是否确实包含了该名字。那么，这个程序到底会打印出什么呢？

```
import java.util.*;

public class Name {
    private final String first, last;

    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return n.first.equals(first) && n.last.equals(last);
    }

    public static void main(String[] args) {
        Set<Name> s = new HashSet<Name>();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
}
```

解惑57：名字里有什么

一个Name实例由一个姓和一个名构成。两个Name实例在通过`equals`方法进行计算时，如果它们的姓相等且名也相等，则这两个Name实例相等。姓和名是用在String中定义的`equals`方法来比较的，两个字符串如果以相同的顺序包含相同的若干个字符，那么它们就相等。因此，两个Name实例如果表示相同的名字，那么它们就相等。例如，下面的方法调用将返回`true`：

```
new Name("Mickey", "Mouse").equals(new Name("Mickey", "Mouse"))
```

该程序的`main`方法创建了两个Name实例，它们都表示Mickey Mouse。该程序将第一个实例放置到了一个散列集合中，然后检查该集合是否包含第二个实例。这两个Name实例是相等的，因此看起来该程序似乎应该打印`true`。如果你运行它，几乎可以肯定它将打印`false`。那么这个程序出了什么问题呢？

这里的bug在于Name违反了`hashCode`约定。这看起来有点奇怪，因为Name连`hashCode`都没有，但是这确实是问题所在。Name类覆写了`equals`方法，而`hashCode`约定要求相等的对象要具有相同的散列码。为了遵守这项约定，无论何时，只要你覆写了`equals`方法，你就必须同时覆写`hashCode`方法[EJ Item 8]。

因为Name类没有覆写`hashCode`方法，所以它从`Object`那里继承了其`hashCode`实现。这个实现返回的是基于标识的散列码。换句话说，不同的对象几乎总是产生不相等的散列值，即使它们是相等的也是如此。所以说Name没有遵守`hashCode`的约定，因此包含Name元素的散列集合的行为是不确定的。

当程序将第一个Name实例放置到散列集合中时，该集合就会在某个散列位置上放置这个实例对应的项。该集合是基于实例的散列值来选择散列位置的，这个散列值是通过实例的`hashCode`方法计算出来的。当该程序在检查第二个Name实例是否包含在散列集合中时，它基于第二个实例的散列值来选择要搜索的散列位置。因为第二个实例有别于第一个实例，因此它极有可能产生不同的散列值。如果这两个散列值映射到了不同的位置，那么`contains`方法将返回`false`：我们所喜爱的啮齿动物米老鼠就在这个散列集合中，但是该集合却找不到它。

假设两个Name实例映射到了相同的位置，那又会怎样呢？我们所了解的所有`HashSet`实现都进行了一种优化，即每一项在存储元素本身之外，还存储了元素的散列值。在搜索某个元素时，这种实现通过遍历集合中的项，去拿存储在每一项中的散列值

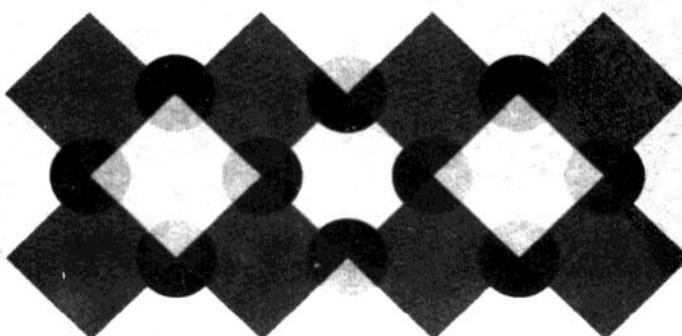
与我们想要查找的元素的散列值进行比较，从而选取适当的散列位置。只有在两个元素的散列值相等的情况下，这种实现才会认为这两个元素相等。这种优化是有实际意义的，因为比较散列码相对于比较元素来说，其代价要小得多。

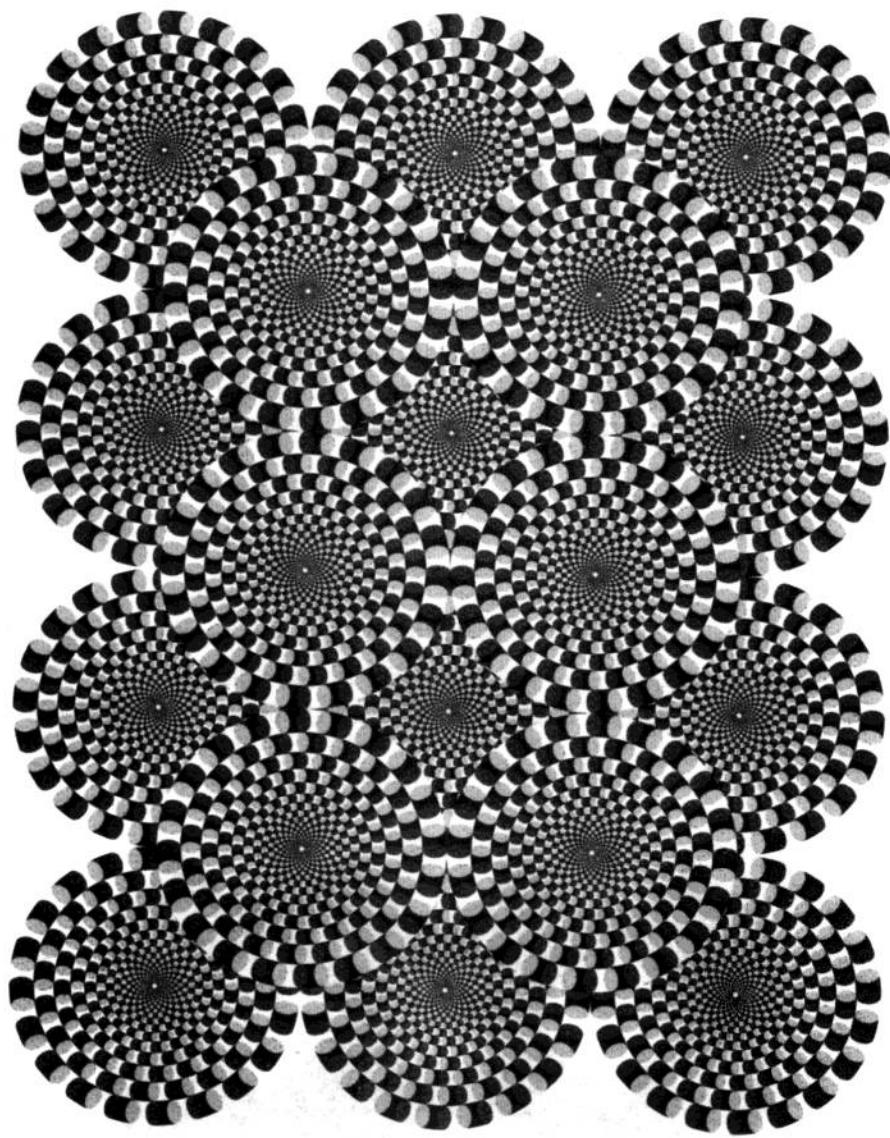
对散列集合来说，这项优化并不足以使其搜索到正确的位置；两个Name实例必须具有相同的散列值才能让散列集合能够将它们识别为是相等的。该程序偶尔也会打印出true，这是因为被连续创建的两个对象偶尔也会具有相同的标识散列码。一个粗略的实验表明，这种偶然性出现的概率大约是两千五百万分之一。这个实验的结果可能会因所使用的Java实现的不同而有所变化，但是在任何我们所知的JRE上，你基本上是不可能看到该程序打印出true的。

要想修正该程序，只需在Name类中添加一个恰当的hashCode方法即可。尽管任何其返回值仅由姓和名来确定的方法都可以满足hashCode的约定，但是高质量的散列函数应该尝试着对不同的名字返回不同的散列值。下面的方法就能够很好地实现这一点[EJ Item 8]。只要我们把该方法添加到了程序中，那么该程序就可以打印出我们所期望的true：

```
public int hashCode() {  
    return 37 * first.hashCode() + last.hashCode();  
}
```

总之，当你覆写equals方法时，一定要记着覆写hashCode方法。更一般地讲，当你在覆写一个方法时，如果它具有一个通用的约定，那么你一定要遵守它。对于大多数在Object中声明的非final的方法，都需要注意这一点[EJ Chapter 3]。不采用这项建议就会导致任意的、不确定的行为。





谜题58：产生它的散列码

本谜题试图从前一个谜题中吸取教训。下面的程序还是由一个Name类和一个main方法构成，这个main方法还是将一个名字放置到一个散列集合中，然后检查该集合是否包含了这个名字。然而，这一次Name类已经覆写了hashCode方法。那么下面的程序将打印出什么呢？

```
import java.util.*;

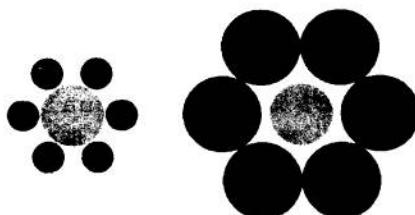
public class Name {
    private final String first, last;

    public Name(String first, String last) {
        this.first = first; this.last = last;
    }

    public boolean equals(Name n) {
        return n.first.equals(first) && n.last.equals(last);
    }

    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }
}

public static void main(String[] args) {
    Set<Name> s = new HashSet<Name>();
    s.add(new Name("Donald", "Duck"));
    System.out.println(
        s.contains(new Name("Donald", "Duck")));
}
```



解惑58：产生它的散列码

与谜题57一样，该程序的main方法创建了两个Name实例，它们表示的是相同的名字。这一次使用的名字是Donald Duck而不是Mickey Mouse，但是它们不应该有很大的区别。main方法同样还是将第一个实例置于一个散列集合中，然后检查该集合中是否包含了第二个实例。这一次hashCode方法明显是正确的，因此看起来该程序应该打印true。但是，表象再次欺骗了我们：它总是打印出false。这一次又是哪里出错了呢？

这个程序的缺陷与谜题57中的缺陷很相似，在谜题57中，Name覆盖了equals方法，但是没有覆盖hashCode方法；而在本谜题中，Name覆盖了hashCode方法，但是没有覆盖equals方法。这并不是说Name没有声明一个equals方法，它确实声明了，但是那是个错误的声明。Name类声明了一个参数类型是Name而不是Object的equals方法。这个类的编写者可能想要覆盖equals方法，但是却错误地重载了它[JLS 8.4.8.1, 8.4.9]。

HashSet类是使用equals(Object)方法来测试元素的相等性的；Name类中声明一个equals(Name)方法对HashSet不造成任何影响。那么Name是从哪里得到了它的equals(Object)方法呢？它是从Object那里继承而来的。这个方法只有在它的参数与其上调用该方法的对象完全相同时才返回true。我们的程序中的main方法将一个Name实例插入到了散列集合中，并且测试另一个实例是否存在于该散列集合中，由此可知该测试一定是返回false的。对我们而言，两个实例都可以代表那令人惊奇的水禽唐老鸭，但是对散列映射表而言，它们只是两个不相等的对象。

修正该程序只需用可以在谜题57中找到的覆盖的equals方法来替换重载的equals方法即可。通过使用这个equals方法，该程序就可以打印出我们所期望的true：

```
public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
    Name n = (Name)o;
    return n.first.equals(first) && n.last.equals(last);
}
```

要让该程序正常工作，你只需增加一个覆盖的equals方法即可。你不必删除那个重载的版本，但是你最好是删掉它。**重载为错误和混乱提供了机会**[EJ Item 26]。如果兼容性要求强制你必须保留一个自身类型的equals方法，那么你应该用自身类型的重载去实现Object的重载，以此来确保它们具有相同的行为：

```
public boolean equals(Object o) {
    return o instanceof Name && equals((Name) o);
}
```

本谜题的教训是：当你想要进行覆写时，千万不要进行重载。为了避免无意识地重载，你应该机械地对你想要覆写的每一个超类方法都拷贝其声明，或者更好的方式是让你的IDE帮你去做这些事。这样做除了保护你免受无意识的重载之害，还保护你免受拼错方法名之害。如果你使用的5.0或者更新的版本，那么对于那些有意地覆写超类方法的方法声明，你可以将@Override注释应用于每一个这样的方法声明上：

```
@Override public Boolean equals(Object o) { ... }
```

在使用这个注释时，除非被注释的方法确实覆写了一个超类方法，否则它将不能编译。对语言设计者来说，值得去考虑在每一个覆写超类方法的方法声明上都添加一个强制性的修饰符。

谜题59：差是什么

下面的程序在计算一个int数组中的元素两两之间的差，将这些差置于一个集合中，然后打印该集合的大小。那么，这个程序将打印出什么呢？

```
import java.util.*;

public class Differences {
    public static void main(String[] args) {
        int vals[] = { 789, 678, 567, 456, 345, 234, 123, 012 };
        Set<Integer> diffs = new HashSet<Integer>();

        for (int i = 0; i < vals.length; i++)
            for (int j = i; j < vals.length; j++)
                diffs.add(vals[i] - vals[j]);
        System.out.println(diffs.size());
    }
}
```

解惑59：差是什么

外层循环迭代数组中的每一个元素，而内层循环从外层循环当前迭代到的元素开始，迭代到数组中的最后一个元素。因此，这个嵌套的循环将遍历数组中每一种可能的两两组合。（元素可以与其自身组成一对。）这个嵌套循环中的每一次迭代都计算了一对元素之间的差（总是正的），并将这个差存储到了集合中，该集合是可以消除重复元素的。因此，本谜题就带来了一个问题，在由`vals`数组中的元素结成的对中，有多少惟一的正的差存在呢？

当你仔细观察程序中的数组时，会发现其构成模式非常明显：连续两个元素之间的差总是111。因此，两个元素之间的差是它们在数组之间的偏移量之差的函数。如果两个元素是相同的，那么它们的差就是0；如果两个元素是相邻的，那么它们的差就是111；如果两个元素被另一个元素隔开了，那么它们的差就是222；以此类推。看起来不同的差的数量与元素间不同的距离的数量是相等的，也就是等于数组的大小，即8。如果你运行该程序，就会发现它打印的是14。怎么回事呢？

上面的分析有一个小的漏洞。要想调查这个缺陷，我们可以通过将`println`语句中的`.size()`这几个字符移除，来打印出集合中的内容。这么做会产生下面的输出：

```
[111, 222, 446, 557, 668, 113, 335, 444, 779, 224, 0, 333, 555, 666]
```

这些数字并非都是111的倍数。在`vals`数组中肯定有两个毗邻的元素的差是113。如果你观察该数组的声明，不可能很清楚地发现原因所在：

```
int vals[] = { 789, 678, 567, 456,
               345, 234, 123, 012 };
```

但是如果你打印数组的内容，你就会看见下面的内容：

```
[789, 678, 567, 456, 345, 234, 123, 10]
```

为什么数组中的最后一个元素是10而不是12呢？因为以0开头的整型字面常量将被解释成为八进制数值[JLS 3.10.1]。这个隐晦的结构是从C编程语言那里遗留下来的东西，C语言产生于20世纪70年代，那时八进制比现在要通用得多。

一旦你知道了`012==10`，就会很清楚为什么该程序打印出了14：有6个不涉及最后

一个元素的惟一的非0差，有7个涉及最后一个元素的非0差，还有0，加在一起正好是14个惟一的差。修正该程序的方法更加明显：将八进制整型字面常量012替换为十进制整型字面常量12。如果你这么做了，该程序将打印出我们所期望的8。

本谜题的教训很简单：千万不要在一个整型字面常量的前面加上一个0；这会使它变成一个八进制字面常量。有意识地使用八进制整型字面常量的情况相当少见，你应该对所有的这种特殊用法增加注释。对语言设计者来说，在决定应该包含什么特性时，应该考虑其限制条件。当有所迟疑时，应该将它剔除在外。

谜题60：一行以毙之

现在该轮到你写一些代码了。下面的谜题每一个都可以用一个方法来解决，这些方法的程序体都只包含一行代码。各就各位，预备，编码！

- A. 编写一个方法，它接受一个包含元素的List，并返回一个新的List，它以相同的顺序包含相同的元素，只不过它把第二次以及后续出现的重复元素都删除了。例如，如果你传递了一个包含"spam", "sausage", "spam", "spam", "bacon", "spam", "tomato"和"spam"的列表，那么你将得到一个包含"spam", "sausage", "bacon", "tomato"的新列表。
- B. 编写一个方法，它接受一个由0个或多个由逗号分隔的标志所组成的字符串，并返回一个表示这些标志的字符串数组，数组中的元素的顺序与这些标志在输入字符串中出现的顺序相同。每一个逗号后面都可能会跟随0个或多个空格字符，这个方法忽略它们。例如，如果你传递的字符串是"fear, surprise, ruthless efficiency, an almost fanatical devotion to the Pope, nice red uniforms"，那么你得到的将是一个包含5个元素的字符串数组，这些元素是"fear", "surprise", "ruthless efficiency", "an almost fanatical devotion to the Pope"和"nice red uniforms"。
- C. 假设你有一个多维数组，出于调试的目的，你想打印它。你不知道这个数组有多少级，以及在数组的每一级中所存储的对象的类型。编写一个方法，它可以向你显示出在每一级上的所有元素。
- D. 编写一个方法，它接受两个int数值，并且如果第一个数值的二进制补码形式比第二个数值的具有更多的位被置位，就返回true。

解惑60：一行以毙之

A. 众所周知，你可以通过把集合中的元素置于一个Set中将集合中的所有重复元素消除。在本谜题中，你还被要求要保持最初的集合中的元素顺序。幸运的是，有一种Set的实现维护其元素插入的顺序，它提供的导人性能接近HashMap。它就是LinkedHashSet，它是在1.4版本的JDK中被添加到Java平台中的。在内部，它是用一个链接列表来处理的，从而被实现为一个散列表。它还有一个映射表版本可供你使用，以定制缓存。一旦你了解了LinkedHashSet，本谜题就很容易解决了。剩下惟一的关键就是你被要求要返回一个List，因此你必须用LinkedHashSet的内容来初始化一个List。把它们放到一块，就形成了下面的解决方案：

```
static <E> List<E> withoutDuplicates(List<E> original) {
    return new ArrayList<E>(new LinkedHashSet<E>(original));
}
```

B. 在将字符串解析成标志时，许多程序员都立刻想到了使用 StringTokenizer。这是最不幸的事情，自1.4版本开始，由于正则表达式被添加到了Java平台中（java.util.regex），StringTokenizer开始变得过时了。如果你试图通过StringTokenizer来解决本谜题，那么你很快就会意识到它不是非常适合。通过使用正则表达式，它就是小菜一碟。为了在一行代码中解决本谜题，我们要使用很方便的方法String.split，它接受一个描述标志分界符的正则表达式作为参数。如果你以前从来没有使用过正则表达式，那么它们看起来会显得有一点神秘，但是它们惊人地强大，值得我们好好学习一下：

```
static String[] parse(String string) {
    return string.split(",\\S*");
}
```

C. 这是一个讲究技巧的问题。你甚至不必去编写一个方法。这个方法在5.0或之后的版本中已经提供了，它就是Arrays.deepToString。如果你传递给它一个对象引用的数组，它将返回一个精密的字符串表示。它可以处理嵌套数组，甚至可以处理循环引用，即一个数组元素直接或间接地引用了其嵌套外层的数组。事实上，5.0版本中的Arrays类提供了一整套的toString、equals和hashCode方法，使你能够打印、比较或散列任何原生类型数组或对象引用数组的内容。

D. 为了在一行代码中解决该谜题，你需要了解在5.0版本中添加到Java平台中的一整套

位操作方法。整数类型的包装器类（Integer、Long、Short、Byte和Char）现在支持通用的位处理操作，包括highestOneBit、lowestOneBit、numberOfLeadingZeros、numberOfTrailingZeros、bitCount、rotateLeft、rotateRight、reverse、signum和reverseBytes。在本例中，你需要的是Integer.bitCount，它返回的是一个int数值中被置位的位数：

```
static boolean hasMoreBitsSet(int i, int j) {  
    return (Integer.bitCount(i) > Integer.bitCount(j));  
}
```

总之，Java平台的每一个主版本都在其类库中隐藏了一些宝藏。本谜题的所有4个部分都依赖于这样的宝藏。每当该平台发布一个新版本时，你都应该研究就一下新特性和提高（new feature and enhancement）页面，这样你就不会漏掉新版本提供的任何惊喜[Features-1.4, Features-5.0]。了解类库中有些什么可以为你节省大量的时间和精力，并且可以提高你的程序的速度和质量。

谜题61：日期游戏

下面的程序演练了Date和Calendar类的某些基本特性，它会打印出什么呢？

```
import java.util.*;  
  
public class DatingGame {  
    public static void main(String[] args) {  
        Calendar cal = Calendar.getInstance();  
        cal.set(1999, 12, 31); // Year, Month, Day  
        System.out.print(cal.get(Calendar.YEAR) + " ");  
  
        Date d = cal.getTime();  
        System.out.println(d.getDay());  
    }  
}
```

解惑61：日期游戏

该程序创建了一个Calendar实例，它应该表示的是1999年的除夕夜，然后该程序打印年份和日。看起来该程序应该打印1999 31，但是它没有；它打印的是2000 1。难道这是致命的Y2K(千年虫)问题吗？

不，事情比我们想像的要糟糕得多：这是致命的Date/Calendar问题。在Java平台首次发布时，它惟一支持的日历计算类就是Date类。这个类在能力方面是受限的，特别是当需要支持国际化时，它就暴露出了一个基本的设计缺陷：Date实例是易变的。在1.1版中，Calendar类被添加到了Java平台中，以矫正Date的缺点，由此大部分的Date方法就都被弃用了。遗憾的是，这么做只能使情况更糟。我们的程序说明Date和Calendar API有许多问题。

该程序的第一个bug就位于方法调用cal.set(1999,12,31)中。当月份以数字来表示时，习惯上我们将第一个月赋值为1。遗憾的是，Date将一月表示为0，而Calendar延续了这个错误。因此，这个方法调用将日历设置到了1999年第13个月的第31天。但是标准的（西历）日历只有12个月，该方法调用肯定应该抛出一个IllegalArgumentException异常，对吗？它是应该这么做，但是它并没有这么做。Calendar类直接将其替换为下一年（在本例中即2000年）的第一个月。这也就解释了为什么我们的程序打印出的第一个数字是2000。

有两种方法可以修正这个问题。你可以将cal.set调用的第二个参数由12改为11，但是这么做容易引起混淆，因为数字11会让读者误以为是11月。更好的方式是使用Calendar专为此目的而定义的常量，即Calendar.DECEMBER。

该程序打印出的第二个数字又是怎么回事呢？cal.set调用很明显是要把日历设置到这个月的第31天，Date实例d表示的是与Calendar相同的时间点，因此它的getDay方法应该返回31，但是程序打印的却是1，这是怎么搞得呢？

为了找出原因，你必须先阅读一下文档，它叙述道**Date.getDay**返回的是**Date**实例所表示的星期日期，而不是月份日期。这个返回值是基于0的，从星期天开始计算，因此程序所打印的1表示2000年12月31日是一个星期一。请注意，相应的Calendar方法，get(Calendar.DAY_OF_WEEK)不知为什么返回的是基于1的星期日期值，而不是像Date的对应方法那样返回基于0的星期日期值。

有两种方法可以修正这个问题。你可以调用Date.getDate这一名字极易让人混淆的方法，它返回的是月份日期。然而，与大多数Date方法一样，它已经被弃用了，因此你最好是将Date彻底抛弃，直接调用Calendar的get(Calendar.DAY_OF_MONTH)方

法。用这两种方法，该程序都可以打印出我们想要的1999 31：

```
public class DatingGame {
    public static void main(String[] args) {
        Calendar cal = Calendar.getInstance();
        cal.set(1999, Calendar.DECEMBER, 31);
        System.out.print(cal.get(Calendar.YEAR) + " ");
        System.out.println(cal.get(Calendar.DAY_OF_MONTH));
    }
}
```

本谜题只是掀开了Calendar和Date缺陷的冰山一角。这些API简直就是雷区。Calendar其他的严重问题包括弱类型（几乎每样事物都是一个int）、过于复杂的状态空间、拙劣的结构、不一致的命名以及不一致的语义等。在使用Calendar或Date的时候一定要当心，千万要记着查阅API文档。

对API设计者来说，其教训是：如果你不能在第一次设计时就使它正确，那么至少应该在第二次设计时应该使它正确，绝对不能留到第三次设计时去处理。如果你对某个API的首次尝试出现了严重问题，那么你的客户可能会原谅你，并且会再给你一次机会。如果你第二次尝试又有问题，你可能会永远坚持这些错误了。

谜题62：名字游戏

下面的程序将两个映射关系放置到了一个映射表中，然后打印它们的大小。那么，它会打印出什么呢？

```
import java.util.*;

public class NameGame {
    public static void main(String args[]) {
        Map<String, String> m =
            new IdentityHashMap<String, String>();
        m.put("Mickey", "Mouse");
        m.put("Mickey", "Mantle");
        System.out.println(m.size());
    }
}
```

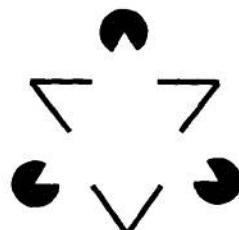
解惑62：名字游戏

对该程序的一种幼稚的分析认为，它应该打印1。该程序虽然将两个映射关系放置到了映射表中，但是它们具有相同的键（Mickey）。这是一个映射表，不是一个多重映射表，所以棒球传奇人物（Mickey Mantle）应该覆盖了啮齿类动画明星（Mickey Mouse），从而只留下一个映射关系在映射表中。

更透彻一些的分析会对这个预测产生质疑。`IdentityHashMap`的文档中叙述道，这个类用一个散列表实现了`Map`接口，它在比较键时，使用的是引用等价性而不是值等价性[Java-API]。换句话说，如果第二次出现的字符串字面常量 "Mickey" 被计算出来是与第一次出现的 "Mickey" 字符串不同的`String`实例的话，那么该程序应该打印2而不是1。如此说来，该程序到底是打印1，还是打印2，抑或是其行为会根据不同的实现而有所变化？

如果你试着运行该程序，你就会发现，尽管我们那个幼稚的分析是有缺陷的，但是该程序正如这种分析所指出的一样，打印出来的是1。这是为什么呢？语言规范保证了字符串是内存限定的，换句话说，相等的字符串常量同时也是相同的[JLS 15.28]。这可以确保在我们的程序中第二次出现的字符串字面常量 "Mickey" 引用到了与第一次相同的`String`实例上，因此尽管我们使用了一个`IdentityHashMap`来代替诸如`HashMap`这样的通用目的的`Map`实现，但是对程序的行为却不会产生任何影响。我们那个幼稚的分析忽略了两个细节，但是这些细节造成的影响却彼此有效地抵消了。

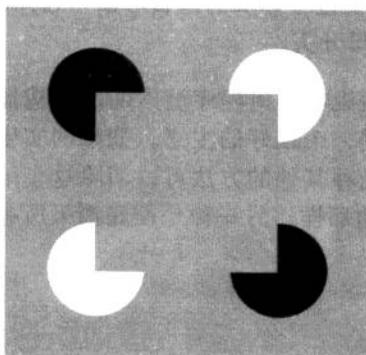
本谜题的一个重要教训是：不要使用`IdentityHashMap`，除非你需要其基于标识的语义；它不是一个通用目的的`Map`实现。这些语义对于实现保持拓扑结构的对象图转换（topology-preserving object graph transformation）非常有用，例如序列化和深层复制。我们得到的次要教训是字符串常量是内存限定的。正如在谜题13中所述，在任何时候，程序都应该尽量不依赖于这种行为去保证它们的操作正确。



谜题63：更多同样的问题

下面的程序除了是面向对象的这一点之外，与前一个非常相似。因为从前一个程序中已经吸取了教训，这个程序使用了一个通用目的的Map实现，即一个HashMap，来替代前一个程序的IdentityHashMap。那么，这个程序会打印出什么呢？

```
import java.util.*;  
  
public class MoreNames {  
    private Map<String, String> m = new HashMap<String, String>();  
  
    public void MoreNames() {  
        m.put("Mickey", "Mouse");  
        m.put("Mickey", "Mantle");  
    }  
  
    public int size() {  
        return m.size();  
    }  
  
    public static void main(String args[]) {  
        MoreNames moreNames = new MoreNames();  
        System.out.println(moreNames.size());  
    }  
}
```



解惑63：更多同样的问题

这个程序看起来很直观，其main方法通过调用无参数的构造器创建了一个MoreNames实例。这个MoreNames实例包含一个私有的Map域（`m`），它被初始化成一个空的HashMap。该无参数的构造器似乎将两个映射关系放置到了映射表`m`中，这两个映射关系都具有相同的键（Mickey）。我们从前一个谜题已知，棒球手（Mickey Mantle）应该覆盖啮齿类动画明星（Mickey Mouse），从而只留下一个映射关系。`main`方法之后在MoreNames实例上调用了size方法，它会调用映射表`m`上的size方法，并返回结果，我们假设其为1。这种分析还剩下一个问题：该程序打印的是0而不是1。这种分析出了什么错呢？

问题在于MoreNames没有任何程序员声明的构造器。它拥有的只是一个返回值为void的实例方法，即MoreNames，编写者可能是想让它作为构造器的。遗憾的是，返回类型（void）的出现将想要的构造器声明变成了一个方法声明，而且该方法永远都不会被调用。因为MoreNames没有任何程序员声明的构造器，所以编译器会帮助（真的是在帮忙吗？）生成一个公共的无参数构造器，它除了初始化它所创建的域实例之外，不做任何事情。就像前面提到的，`m`被初始化成了一个空的HashMap。当在这个HashMap上调用size方法时，它将返回0，这正是该程序打印出来的内容。

修正该程序很简单，只需将void返回类型从MoreNames声明中移除即可，这将使它从一个实例方法声明变成一个构造器声明。通过这种修改，该程序就可以打印出我们所期望的1。

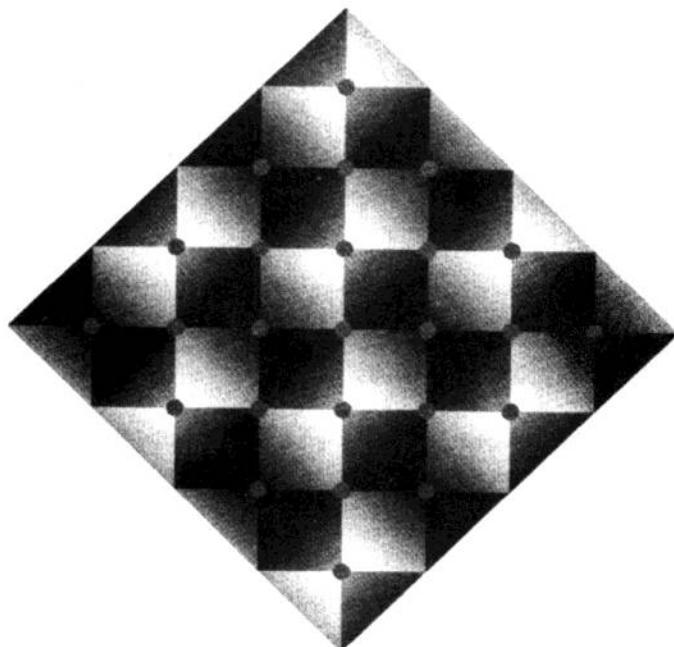
本谜题的教训是：**不要因为偶然地添加了一个返回类型，而将一个构造器声明变成了一个方法声明。**尽管一个方法的名字与声明它的类的名字相同是合法的，但是你千万不要这么做。更一般地讲，**要遵守标准的命名习惯**，它强制要求方法名必须以小写字母开头，而类名应该以大写字母开头。

对语言设计者来说，在没有任何程序员声明的构造器的情况下，自动生成一个缺省的构造器这种做法并非是一个很好的主意。如果确实生成了这样的构造器，也许应该让它们是私有的。有好几种其他的方法可以消除这个陷阱。一种方法是禁止方法名与类名相同，就像C#所做的那样，另一种是彻底消灭所有的构造器，就像Smalltalk所做的那样。

谜题64：按余数编组

下面的程序将生成整数对3取余的柱状图，那么，它将打印出什么呢？

```
public class Mod {  
    public static void main(String[] args) {  
        final int MODULUS = 3;  
        int[] histogram = new int[MODULUS];  
  
        // Iterate over all ints (Idiom from Puzzle 26)  
        int i = Integer.MIN_VALUE;  
        do {  
            histogram[Math.abs(i) % MODULUS]++;  
        } while (i++ != Integer.MAX_VALUE);  
  
        for (int j = 0; j < MODULUS; j++)  
            System.out.println(histogram[j] + " ");  
    }  
}
```



解惑64：按余数编组

该程序首先初始化int数组histogram，其每一个位置都为对3取余的一个结果数值而准备（0、1和2），所有这三个位置都被初始化为0。然后，该程序在所有 2^{32} 个int数值上遍历变量i，使用的是在谜题26中介绍的惯用法。因为整数取余操作（%）在第一个操作数是负数时，可以返回一个负值，就像在谜题1中所描述的那样，所以该程序在计算i被3整除的余数之前，先取i的绝对值。然后用这个余数来递增数组位置的索引。在循环完成之后，该程序将打印histogram数组中的内容，它的元素表示对3取余得到0、1和2的int数值的个数。

该程序所打印的三个数字应该彼此大致相等，它们加起来应该等于 2^{32} 。如果你想知道怎样计算出它们的精确值，且你具备一些数学情结，那么请仔细阅读下面两段话。否则，你可以跳过这两段话。

该程序打印的三个数字不可能精确地相等，因为它们必须加起来等于 2^{32} ，这个数字不能被3除尽。如果你仔细观察2的连续幂级数对3取余的值，就会发现，它们在1和2之间交替变化： 2^0 对3取余是1， 2^1 对3取余是2， 2^2 对3取余是1， 2^3 对3取余是2，以此类推。每一个2的偶次幂对3取余的值都是1，每一个2的奇次幂对3取余的值都是2。因为 2^{32} 对3取余是1，所以该程序所打印的三个数字中有一个将比另外两个大1，但是它是哪一个呢？

该循环依次递增三个数组元素的数值，因此该循环最后递增的那个数值必然是最大的数值，它就是表示Integer.MAX_VALUE或($2^{31}-1$)对3取余的数值。因为 2^{31} 是2的奇次幂，所以它对3取余应该得到2，因此($2^{31}-1$)对3取余将得到1。该程序打印的三个数字中的第二个表示的就是对3取余得到1的int数值的个数，因此，我们期望这个值比第一个和最后一个数值大1。

由此，该程序应该在运行了相当长的时间之后，打印($2^{32}/3$)的较小值($2^{32}/3$)的较大值($2^{32}/3$)的较小值，即1431655765 1431655766 1431655765。但是它真的是这么做的吗？不，它几乎立刻就抛出了下面的异常：

```
Exception in thread "main" ArrayIndexOutOfBoundsException: -2
at Mod.main(Mod.java:9)
```

问题出在哪了呢？

问题在于该程序对Math.abs方法的使用上，它会导致错误的对3取余的数值。考虑一下当i为-2所发生的事情，该程序计算Math.abs(-2)%3的数值，得到2，但是

-2对3取余应该得到1。这可以解释为什么产生了不正确的统计结果，但是还有一个问题留待解决，为什么程序抛出了`ArrayIndexOutOfBoundsException`异常呢？这个异常表明该程序使用了一个负的数组索引，但是这肯定是不可能的：数组索引是通过接受`i`的绝对值并计算这个绝对值被3¹整除时的余数而计算出来的。在计算一个非负的`int`数值整除一个正的`int`数值的余数时，可以保证将产生一个非负的结果[JLS 15.17.3]。我们又要问了，这里又出了什么问题呢？

要回答这个问题，我们必须要去看看`Math.abs`的文档。这个方法的名字有一点带有欺骗性，它几乎总是返回它的参数的绝对值，但是在一种情况下，它做不到这一点。文档中叙述道：“如果其参数等于`Integer.MIN_VALUE`，那么产生的结果与该参数相同，它是一个负数。”通过对这条知识的掌握，就可以很清楚地知道为什么该程序立即抛出了`ArrayIndexOutOfBoundsException`异常。循环索引`i`的初始值是`Integer.MIN_VALUE`，由`Math.abs(Integer.MIN_VALUE) % 3`所产生的数组索引等于`Integer.MIN_VALUE % 3`，即-2²。

为了订正这个程序，我们必须用一个真正的取余操作来替代伪取余计算(`Math.abs(i) % MODULUS`)。如果我们将这个表达式替换为对下面这个方法的调用，那么该程序就可以产生我们期望的输出1431655765 1431655766 1431655765：

```
private static int mod(int i, int modulus) {
    int result = i % modulus;
    return result < 0 ? result + modulus : result;
}
```

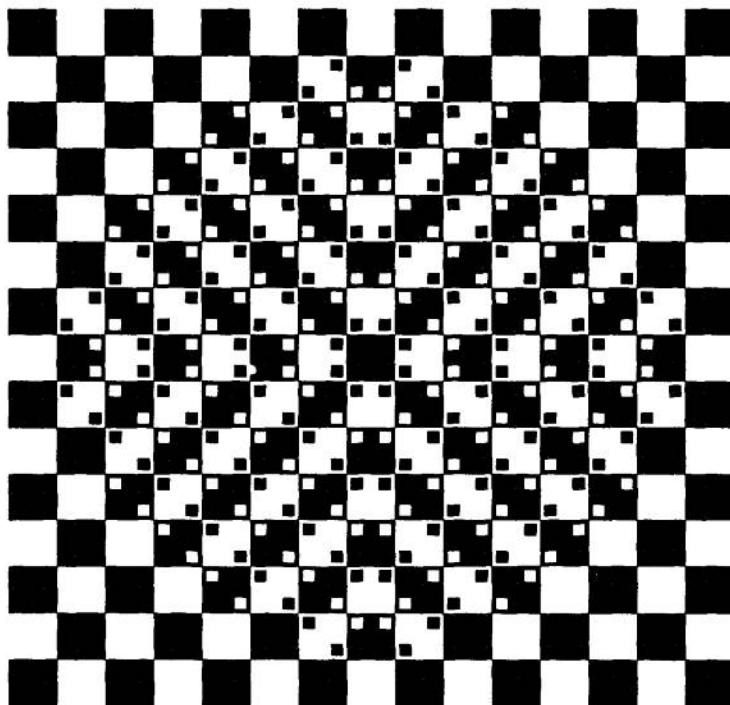
本谜题的教训是：`Math.abs`不能保证一定会返回非负的结果。如果它的参数是`Integer.MIN_VALUE`，或者对于`long`版本的实现传递的是`Long.MIN_VALUE`，那么它将返回它的参数。这个方法在一般情况下是不会这么做的，上述这种行为的根源在于2的补码算数具有不对称性，这在谜题33中已经很详细地讨论过了。简单地讲，没有任何`int`数值可以表示`Integer.MIN_VALUE`的负值，也没有任何`long`数值可以表示`Long.MIN_VALUE`的负值。对类库的设计者来说，也许在将`Integer.MIN_VALUE`和`Long.MIN_VALUE`传递给`Math.abs`时，抛出`IllegalArgumentException`会显得更合理。然而，有人可能会争辩道，该方法的实际行为应该与Java内置的整数算术操作相一致，它们在溢出时并不会抛出异常。

1. 原书中为2，有误。——译者注

2. 原书中为2，缺负号。——译者注

谜题65：疑似排序的惊人传奇

下面的程序使用定制的比较器，对一个由随机挑选的Integer实例组成的数组进行排序，然后打印了一个描述了数组顺序的单词。回忆一下，Comparator接口只有一个方法，即compare，它在第一个参数小于第二个参数时返回一个负数，在两个参数相等时返回0，在第一个参数大于第二个参数时返回一个正数。这个程序是展示5.0版特性的一个范例程序。它使用了自动包装和解包、泛型和枚举类型。那么，它会打印出什么呢？



```
import java.util.*;  
  
public class SuspiciousSort {  
    public static void main(String[] args) {  
        Random rnd = new Random();  
        Integer[] arr = new Integer[100];  
  
        for (int i = 0; i < arr.length; i++)  
            arr[i] = rnd.nextInt();  
  
        Comparator<Integer> cmp = new Comparator<Integer>() {  
            public int compare(Integer i1, Integer i2) {  
                return i2 - i1;  
            }  
        };  
        Arrays.sort(arr, cmp);  
        System.out.println(order(arr));  
    }  
  
    enum Order { ASCENDING, DESCENDING, CONSTANT, UNORDERED };  
  
    static Order order(Integer[] a) {  
        boolean ascending = false;  
        boolean descending = false;  
  
        for (int i = 1; i < a.length; i++) {  
            ascending |=(a[i] > a[i-1]);  
            descending |=(a[i] < a[i-1]);  
        }  
  
        if (ascending && !descending)  
            return Order.ASCENDING;  
        if (descending && !ascending)  
            return Order.DESCENDING;  
        if (!ascending)  
            return Order.CONSTANT; // All elements equal  
        return Order.UNORDERED; // Array is not sorted  
    }  
}
```

解惑65：疑似排序的惊人传奇

该程序的main方法创建了一个Integer实例的数组，并用随机数对其进行了初始化，然后用比较器cmp对该数组进行排序。这个比较器的compare方法将返回它的第二个参数减去第一个参数的值，如果第二个参数表示的是比第一个参数大的数值，其返回值就是正的；如果这两个参数相等，其返回值为0；如果第二个参数表示的是比第一个参数小的数值，其返回值就是负的。这种行为正好与compare方法通常的做法相反，因此，该比较器应该施加降序排列。

在对数组排序之后，main方法将该数组传递给了静态方法order，然后打印由这个方法返回的结果。该方法在数组中所有的元素都表示相同的数值时，返回CONSTANT；在数组中每一对毗邻的元素中第二个元素都大于等于第一个元素时，返回ASCENDING；在数组中每一对毗邻的元素中第二个元素都小于等于第一个元素时，返回DESCENDING；在这些条件都不满足时，返回UNORDERED。尽管理论上说，数组中的100个随机数有可能彼此都相等，但是这种奇特现象发生的概率为 $2^{32 \times 99}$ 分之一，即大约 5×10^{953} 分之一。因此，该程序看起来应该打印DESCENDING。如果你运行该程序，几乎可以肯定你将看到它打印的是UNORDERED。为什么它会产生如此的行为呢？

order方法很直观，它并不会说谎。Arrays.sort方法已经存在许多年了，也不应有问题。现在只有一个地方能够发现bug了：比较器。乍一看，这个比较器似乎不可能出错。毕竟，它使用的是标准的惯用法：用两个数的运算结果的正负号表示这两个数的大小顺序，可以用减法计算二者之差。这个惯用法至少从20世纪70年代早期就一直存在了，它在早期的UNIX里面被广泛地应用。遗憾的是，这种惯用法从来都是有问题的。本谜题也许应该称为“白痴一般的惯用法的案例”。这种惯用法的问题在于定长的整数没有大到可以保存任意两个同等长度的整数之差的程度。当你在做两个int或long数值的减法时，其结果可能会溢出，在这种情况下我们就会得到错误的符号。例如，请考虑下面的程序：



```

public class Overflow {
    public static void main(String[] args) {
        int x = -2000000000;
        int z = 2000000000;
        System.out.println(x - z);
    }
}

```

很明显，`x`比`z`小，但是程序打印的是`294967296`，它是一个正数。既然这种比较的惯用法是有问题的，那么为什么它会被如此广泛地应用呢？因为它在大多数时间里可以正常使用。它只在用来进行比较的两个数字的差大于`Integer.MAX_VALUE`的时候才会出问题。这意味着对于许多应用而言，在实际使用中是不会看到这种错误的。更糟的是，它们被观察到的次数少之又少，以至于这个bug永远都不会被发现和订正。

那么这对于我们的程序的行为意味着什么呢？如果你查阅一下`Comparator`的文档，你就会看到它所实现的排序关系必须是可传递的，换句话说，`(compare(x,y)>0) && (compare(y,z)>0)`蕴含着`compare(x,z)>0`。如果我们取`Overflow`例子中的`x`和`z`，并取`y`为0，那么我们的比较器在这些数值上就违反了可传递性。事实上，在所有随机选取的`int`数值对中，对四分之一的数值对，该比较器都会返回错误的值。用这样的比较器来执行一个搜索或排序，或者用它去排序一个有序的集合，都会产生不确定的行为，就像我们在运行本谜题的程序时所看到的那样。出于数学上的倾向性，`Comparator.compare`方法的一般约定要求比较器要产生一个全序（total order），但是这个比较器在数个计算上都未能做到这一点。

我们可以通过替换遵守上述一般约定的`Comparator`实现来订正我们的程序。因为只是想要反转自然排序的顺序，所以我们甚至可以不必编写自己的比较器。`Collections`类提供了一个可以排序的比较器。如果你用`Arrays.sort(arr, Collections.reverseOrder())`来替代最初的`Arrays.sort`调用，该程序就可以打印出我们所期望的`DESCENDING`。

或者，你可以编写你自己的比较器。下面的代码并不“聪明”，但是它可以工作，从而使该程序可以打印出我们所期望的`DESCENDING`：

```

public int compare(Integer i1, Integer i2) {
    return (i2 < i1 ? -1 : (i2 == i1 ? 0 : 1));
}

```

本谜题有数个教训，最有针对性的是：不要使用基于减法的比较器，除非你能够确保要比较的数值之间的差永远不会大于**Integer.MAX_VALUE**[EJ Item 11]。更一般地讲，要意识到int的溢出，就像谜题3、26和33所讨论的那样。另一个教训是你应该避免“聪明”的代码。应该努力去编写清晰正确的代码，不要对它做任何优化，除非该优化被证明是必需的[EJ Item 37]。

对语言设计者来说，得到的教训与谜题3、26和33相同：也许值得去考虑支持某种形式整数算术运算，它不会在溢出时不抛出异常。还有就是可能应该在语言中提供一个三值的比较器操作符，就像Perl所做的那样（<=>操作符）。

更多类之谜

在本章所描述的谜题涉及继承、覆写和其他形式的名字重用。

谜题66：一件私事

在下面的程序中，子类的一个域具有与超类的一个域相同的名字。那么，这个程序会打印出什么呢？

```
class Base {  
    public String className = "Base";  
}  
  
class Derived extends Base {  
    private String className = "Derived";  
}  
  
public class PrivateMatter {  
    public static void main(String[] args) {  
        System.out.println(new Derived().className);  
    }  
}
```

解惑66：一件私事

对该程序的肤浅分析可能会认为它应该打印Derived，因为这正是存储在每一个Derived实例的className域中的内容。更深入一点的分析会认为Derived类不能编译，因为Derived中的className变量具有比Base中的className变量更具限制性的访问权限。如果你尝试着编译该程序，就会发现这种分析也不正确。该程序确实不能编译，但是错误却出在PrivateMatter中。

如果className是一个实例方法，而不是一个实例域，那么Derived.className()将覆盖Base.className()，而这样的程序是非法的。一个覆盖方法的访问修饰符所提供的访问权限与被覆盖方法的访问修饰符所提供的访问权限相比，至少要一样多[JLS 8.4.8.3]。因为className是一个域，所以Derived.className隐藏（hide）了Base.className，而不是覆盖了它[JLS 8.3]。对一个域来说，当它要隐藏另一个域时，如果隐藏域的访问修饰符提供的访问权限比被隐藏域的少，尽管这么做是不可取的，但是它确实是合法的。事实上，对于隐藏域来说，如果它具有与被隐藏域完全无关的类型，也是合法的：即使Derived.className是GregorianCalendar类型的，Derived类也是合法的。

在我们的程序中的编译错误出现在PrivateMatter类试图访问Derived.className的时候。尽管Base有一个公共域className，但是这个域没有被继承到Derived类中，因为它被Derived.className隐藏了。在Derived类内部，域名className引用的是私有域Derived.className。因为这个域被声明为是private的，所以它对于PrivateMatter来说是不可访问的。因此，编译器产生了类似下面这样的一条错误信息：

```
PrivateMatter.java:11: className has private access in Derived
    System.out.println(new Derived().className);
                           ^
```



请注意，尽管在Derived实例中的公共域Base.className被隐藏了，但是我们还是可以通过将Derived实例转型为Base来访问到它。下面版本的PrivateMatter就可以打印出Base：

```
public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(((Base)new Derived()).className);
    }
}
```

这说明了覆盖与隐藏之间的一个非常大的区别。一旦一个方法在子类中被覆盖，你就不能在子类的实例上调用它了（除了在子类内部，通过使用super关键字的方法）。然而，你可以通过将子类实例转型为某个超类类型来访问被隐藏的域，在这个超类中该域未被隐藏。

如果你想让这个程序打印Derived，也就是说，你想展示覆盖行为，那么你可以用公共方法来替代公共域。在任何情况下，这都是一个好主意，因为它提供了更好的封装[EJ Item 19]。下面的程序版本就使用了这项技术，并且能够打印出我们所期望的Derived：

```
class Base {
    public String getClassName() {
        return "Base";
    }
}

class Derived extends Base {
    public String getClassName() {
        return "Derived";
    }
}

public class PublicMatter {
    public static void main(String[] args) {
        System.out.println(new Derived().getClassName());
    }
}
```

请注意，我们将Derived类中的getClassName方法声明成了public的，尽管在最初的程序中与其相对应的域是私有的。就像前面提到的那样，覆盖方法的访问修饰符

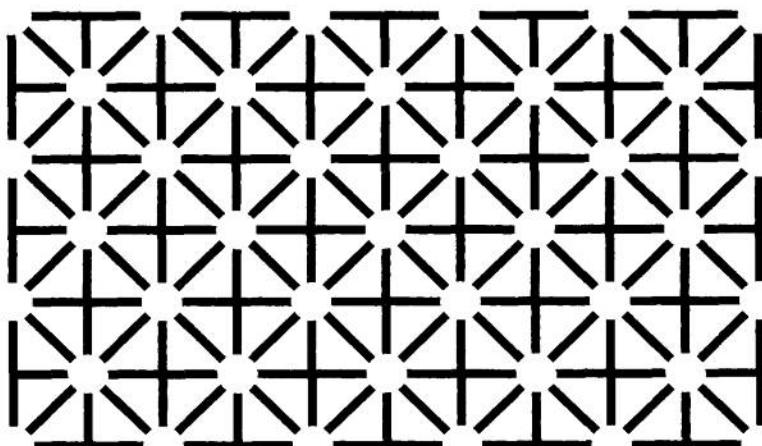
与它要覆写的方法的访问修饰符相比，所具有的限制性不能有任何降低。

本谜题的教训是，隐藏通常都不是一个好主意。Java语言允许你去隐藏变量、嵌套类型，甚至是静态方法（就像在谜题48所展示的那样），但是你不能认为你就应该去隐藏。隐藏的问题在于它将导致读者头脑的混乱。你正在使用一个被隐藏实体，或者是正在使用一个执行了隐藏的实体吗？要避免这类混乱，只需避免隐藏。

如果一个类要隐藏一个域，而用来隐藏该域的域具有的可访问性比被隐藏域更具限制性，就像我们最初的程序那样，那么这就违反了包容性（subsumption）原则，即大家所熟知的Liskov置换原则（Liskov Substitution Principle）[Liskov87]。这项原则叙述道，你能够对基类所做的任何事，都同样能够作用于其子类。包容性是面向对象编程的自然心理模型的一个不可分割的部分。无论何时，只要违反了这项原则，就会对程序的理解造成困难。还有其他数种用另一个域来隐藏某个域的方法也会违反包容性：例如，两个域具有不同的类型；一个域是静态的而另一个域不是；一个域是final的而另一个域不是；一个域是常量而另一个域不是；以及两个域都是常量但是它们具有不同的值。

对于语言设计者而言，应该考虑消除隐藏的可能性：例如，使所有的域都隐含地是私有的。如果这样做显得过于严苛，那么至少应该考虑对隐藏进行限制，以使其遵守包容性原则。

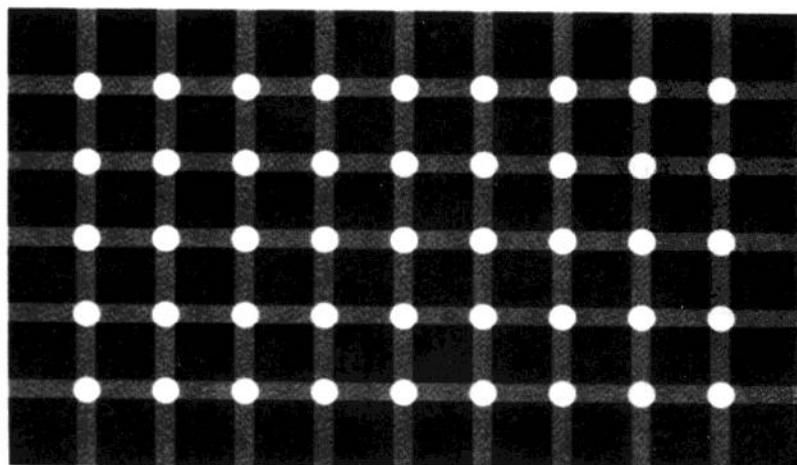
总之，当你在声明一个域、一个静态方法或一个嵌套类型时，如果其名字与基类中相对应的某个可访问的域、方法或类型相同，就会发生隐藏。隐藏是容易产生混乱的：违反包容性的隐藏域在某种意义上是特别有害的。更一般地讲，除了覆写之外，要避免名字重用。



谜题67：对字符串上瘾

一个名字可以被用来引用位于不同包内的多个类。下面的程序就是在探究当你重用了一个平台类的名字时，会发生什么。你认为它会做些什么呢？尽管这个程序属于那种让你通常一看到就会感到尴尬的程序，但是你还是应该继续下去，把门锁上，把百叶窗拉上，然后试试看：

```
public class StrungOut {  
    public static void main(String[] args) {  
        String s = new String("Hello world");  
        System.out.println(s);  
    }  
}  
  
class String {  
    private final java.lang.String s;  
  
    public String(java.lang.String s) {  
        this.s = s;  
    }  
  
    public java.lang.String toString() {  
        return s;  
    }  
}
```



解惑67：对字符串上瘾

如果说这个程序有点让人讨厌的话，它看起来还是相当简单的。在未命名包中的String类就是一个java.lang.String实例的包装器，看起来该程序应该打印Hello world。如果你尝试着运行该程序，会发现不能运行，VM弹出了一个像下面这样的错误消息：

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

但是肯定有main方法：它就白纸黑字地写在那里。为什么VM找不到它呢？

VM不能找到main方法是因为它并不在那里。尽管StrungOut有一个被命名为main的方法，但是它却具有错误的签名。一个main方法必须接受一个单一的字符串数组参数[JVMS 5.2]。VM努力要告诉我们的是StrungOut.main接受的是由我们的String类所构成的数组，它无论如何都与java.lang.String没有任何关系。

如果你确实需要编写自己的字符串类，看在老天爷的份上，千万不要称其为String。要避免重用平台类的名字，并且千万不要重用java.lang中的类名，因为这些名字会被各处的程序自动加载。程序员习惯于看到这些名字以无限定的形式出现，并且会很自然地认为这些名字引用的是我们所熟知的java.lang中的类。如果你重用了这些名字的某一个，那么当这个名字在其自己的包内被使用时，该名字的无限定形式将会引用到新的定义上。

要订正该程序，只需为这个非标准的字符串类挑选一个合理的名字即可。该程序下面的这个版本很明显是正确的，而且它比最初的版本要更易于理解。它将打印出如你所期望的Hello world：

```
public class StrungOut1 {
    public static void main(String[] args) {
        MyString s = new MyString ("Hello world");
        System.out.println(s);
    }
}
class MyString {
    private final String s;
    public MyString(String s) { this.s = s;}
    public String toString() { return s; }
}
```

宽泛地讲，本谜题的教训就是要避免重用类名，尤其是Java平台类的类名。千万不要重用java.lang包内的类名，相同的教训也适用于类库的设计者。Java平台的设计者已经在这个问题上栽过数次了，著名的例子有java.sql.Date，它与java.util.Date和org.omg.CORBA.Object相冲突。与在本章中的许多其他谜题一样，这个教训再次说明了一个原则：除非覆写一般情况下应该避免名字重用。对平台实现者来说，其教训是诊断信息应该清晰地解释失败的原因。VM应该可以很容易地将没有任何具有正确签名的main方法的情况与根本就没有任何main方法的情况区分开。

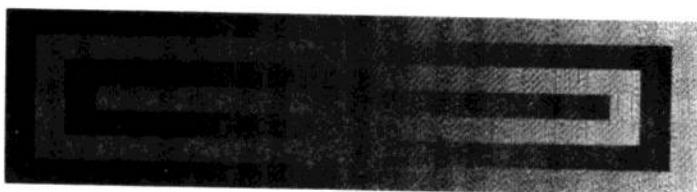
谜题68：灰色的阴影

下面的程序在相同的范围内具有两个名字相同的声明，并且没有任何明显的方式可以在它们二者之间做选择。这个程序会打印Black吗？它会打印White吗？甚至，它是合法的吗？

```
public class ShadesOfGray {
    public static void main(String[] args) {
        System.out.println(X.Y.Z);
    }
}

class X {
    static class Y {
        static String Z = "Black";
    }
    static C Y = new C();
}

class C {
    String Z = "White";
}
```



解惑68：灰色的阴影

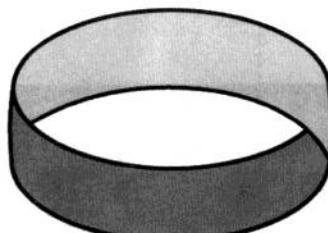
没有任何显而易见的方法可以确定该程序应该打印Black还是White。编译器通常会拒绝模棱两可的程序，而这个程序看起来肯定是模棱两可的。因此，它似乎应该是非法的。如果你试着运行它，就会发现它是合法的，并且会打印出White。你怎样才能事先了解这一切呢？

可以证明，在这样的上下文环境中，有一条规则决定着程序的行为，即当一个变量和一个类型具有相同的名字，并且它们位于相同的作用域时，变量名具有优先权[JLS 6.5.2]。变量名将遮掩（obscure）类型名[JLS 6.3.2]。相似地，变量名和类型名可以遮掩包名。这条规则真的是相当地晦涩，任何依赖于它的程序都极有可能使其读者晕头转向。

幸运的是，遵守标准的Java命名习惯的程序员从来都不会遇上这个问题。类应该以一个大写字母开头，以MixedCase的形式书写；变量应该以一个小写字母开头，以mixedCase的形式书写；而常量应该以一个大写字母开头，以ALL_CAPS的方式书写。单个的大写字母只能用于类型参数，就像在泛型接口Map<K, V>中那样。包名应该以lower.case的方式命名[JLS 6.8]。

为了避免常量名与类名的冲突，在类名中应该将首字母缩拼词当作普通的词处理[EJ Item 38]。例如，一个表示全局惟一标识符的类应该被命名为Uuid，而不是UUID，尽管其首字母缩拼词通常被写为UUID。（Java平台库就违反了这项建议，因为它具有UUID、URL和URI这样的类名。）为了避免变量名与包名的冲突，请不要使用顶层的包名或领域名作为变量的名字，特别是不要将一个变量命名为com、org、net、edu、java或javax。

要想移除ShadesOfGray这个程序中的所有不明确性，只需以遵守命名习惯的方式对其重写即可。很明显，下面的程序将打印Black。作为一种附加的好处，当你大声朗读这个程序时，听起来与最初的那个程序是完全一样的。



```
public class ShadesOfGray {
    public static void main(String[] args) {
        System.out.println(Ex.Why.z);
    }
}

class Ex {
    static class Why {
        static String z = "Black";
    }
    static See y = new See();
}

class See {
    String z = "White";
}
```

总之，应该遵守标准的命名习惯以避免不同的命名空间之间的冲突，还有一个原因就是如果你违反这些习惯，那么你的程序将让人难以辨认。同样，为了避免变量名与通用的顶层包名相冲突，请使用MixedCase风格的类名，即使其名字是首字母缩拼词也应如此。通过遵守这些规则，你就可以确保你的程序永远不会遮掩类名或包名。再次说明一下，这里列举的仍然是你应该在覆写之外的情况下避免名字重用的一个实例。对语言设计者来说，应该考虑去消除遮掩的可能性。C#是通过将域和嵌套类置于相同的命名空间来实现这一点的。

谜题69：黑色的渐隐

假设你不能修改前一个谜题（谜题68）中的X和C这两个类。你能否编写一个类，其main方法将读取X.Y类中的z域的值，然后打印它。注意，不能使用反射。



解惑69：黑色的渐隐

本谜题初看起来是不可能实现的。毕竟，`X.Y`类被具有相同名字的一个域给遮掩了，因此对其命名的尝试将引用到该域上。

事实上，我们是可以引用一个被遮掩的类型的，其技巧就是在某一种特殊的语法上下文环境中使用该名字，在该语法上下文环境中允许出现一个类型但是不允许出现一个变量。在转型表达式的括号中间的部分就是这样一种上下文环境。下面的程序通过使用这种技术解决了这个谜题，并且将打印出我们所期望的`Black`：

```
public class FadeToBlack {
    public static void main(String[] args) {
        System.out.println(((X.Y)null).Z);
    }
}
```

请注意，我们是用一个具有`X.Y`类型的表达式来访问`X.Y`类的`Z`域的。就像我们在谜题48和54中所看到的，用一个表达式而不是类型名来访问一个静态成员是合法的，但却是一种有问题的用法。

不借助这种有问题的用法，而是通过在一个类声明的`extends`子句中使用一个被遮掩的类这种方式，你也可以解决本谜题。因为基类总是一种类型，出现在`extends`子句中的名字从来都不会被解析为变量名。下面的程序就展示了这项技术，它也会打印出`Black`：

```
public class FadeToBlack {
    static class Xy extends X.Y { }

    public static void main(String[] args) {
        System.out.println(Xy.Z);
    }
}
```

如果你使用的是5.0或更新的版本，那么通过在一个类型变量声明的`extends`子句中使用`X.Y`这种方式，你也可以解决本谜题：

```
public class FadeToBlack {
    public static <T extends X.Y> void main(String[] args) {
        System.out.println(T.Z);
    }
}
```

总之，要解决由类型被变量遮掩而引发的问题，需要按照标准的命名习惯来重命名类型和变量，就像在谜题68中所讨论的那样。如果做不到这一点，那么你应该在只允许类型名的上下文环境中使用被遮掩的类型名。幸运的话，你将永远不需要凭借这种对程序的变形来解决问题，因为大多数的类库作者都很明智，他们都避免了必需使用这种变形的有问题的用法。然而，如果你确实发现自己身处这种境地，那么你最好是要知道这个问题需要解决。

谜题70：一揽子交易

下面这个程序设计在不同的包中的两个类的交互，main方法位于hack.TypeIt中。那么，这个程序会打印什么呢？

```
package hack;
import click.CodeTalk;

public class TypeIt {
    private static class ClickIt extends CodeTalk {
        void printMessage() {
            System.out.println("Hack");
        }
    }

    public static void main(String[] args) {
        ClickIt clickit = new ClickIt();
        clickit.doIt();
    }
}

package click;
public class CodeTalk {
    public void doIt() {
        printMessage();
    }

    void printMessage() {
        System.out.println("Click");
    }
}
```

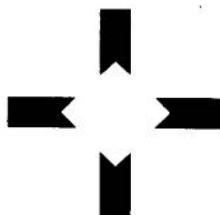
解惑70：一揽子交易

本谜题看起来很直观。`hack.TypeIt`的`main`方法对`TypeIt.ClickIt`类实例化，然后调用其`doIt`方法，该方法是从`CodeTalk`继承而来。接着，该方法调用`printMessage`方法，它在`TypeIt.ClickIt`中被声明为打印Hack。然而，如果你运行该程序，它打印的将是Click。怎么会这样呢？

上面的分析做出了一个不正确的假设，即`hack.TypeIt.ClickIt.printMessage`方法覆写了`click.CodeTalk.printMessage`方法。一个包内私有的方法不能被位于另一个包中的某个方法直接覆写[JLS 8.4.8.1]。在程序中的这两个`printMessage`方法是无关的，它们仅仅是具有相同的名字而已。当程序在`hack`包内调用`printMessage`方法时，运行的是`hack.TypeIt.ClickIt.printMessage`方法。这个方法将打印Click，这也解释了我们所观察到的行为。

如果你想让`hack.TypeIt.ClickIt`中的`printMessage`方法覆写在`click.CodeTalk`中的该方法，那么你必须在`click.CodeTalk`中的该方法声明之前添加`protected`或`public`修饰符。要使该程序能够编译，你还必须在`hack.TypeIt.ClickIt`的覆写声明的前面添加一个修饰符，该修饰符与你在`Click.CodeTalk`¹的`printMessage`方法上放置的修饰符相比，所具备的限制性不能更多[JLS 8.4.8.3]。换句话说，两个`printMessage`方法可以都被声明为是`public`的，也可以都被声明为是`protected`的，或者，超类中的方法被声明为`protected`，而子类中的方法被声明为是`public`的。无论你执行了上述三种修改中的哪一种，该程序都将打印Hack，从而表明确实发生了覆写。

总之，包内私有的方法不能直接被包外声明的方法所覆写。尽管包内私有的访问权限和覆写结合到一起会导致某种混乱，但是Java当前的行为是允许使用包的，以支持比单个的类更大的抽象封装。包内私有的方法是它们所属包的实现细节，在包外重用它们的名字是不会对包内产生任何影响的。



1. 原书写为`hack.TypeIt.ClickIt`，有误。——译者注

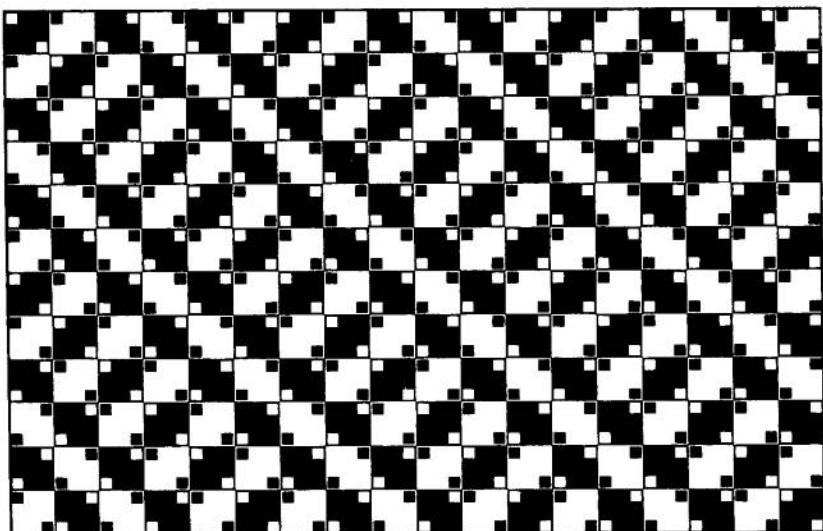
谜题71：进口税

在5.0版中，Java平台引入了大量的可以使操作数组变得更加容易的工具。下面这个谜题使用了变量参数、自动包装、静态导入（请查看[http://java.sun.com/j2se/5.0/docs/guide/language\[Java-5.0\]](http://java.sun.com/j2se/5.0/docs/guide/language[Java-5.0])）以及便捷方法`Arrays.toString`（请查看谜题60）。那么，这个程序会打印什么呢？

```
import static java.util.Arrays.toString;

class ImportDuty {
    public static void main(String[] args) {
        printArgs(1, 2, 3, 4, 5);
    }

    static void printArgs(Object... args) {
        System.out.println(toString(args));
    }
}
```



解惑71：进口税

你可能会期望该程序打印[1,2,3,4,5]，实际上它确实会这么做，只要它能编译。令人沮丧的是，看起来编译器找不到恰当的**toString**方法：

```
ImportDuty.java:9:Object.toString can't be applied to(Object[])
System.out.println(toString(args));
^
```

是不是编译器的理解力太差了？为什么它会尝试着去应用**Object.toString()**呢？它与调用参数列表并不匹配，而**Arrays.toString(Object[])**却可以完全匹配。

编译器在选择在运行期将被调用的方法时，所做的第一件事就是在肯定能找到该方法的范围内挑选[JLS 15.12.1]。编译器将在包含了具有恰当名字的方法的最小闭合范围内进行挑选，在我们的程序中，这个范围就是**ImportDuty**类，它包含了从**Object**继承而来的**toString**方法。在这个范围内没有任何可以应用于**toString(args)**调用的方法，因此编译器必须拒绝该程序。

换句话说，我们想要的**toString**方法没有在调用点所处的范围内。导入的**toString**方法被**ImportDuty**从**Object**那里继承而来的具有相同名字的方法所遮蔽（shadow）了[JLS 6.3.1]。遮蔽与遮掩（谜题68）非常相像，二者的关键区别是一个声明只能遮蔽类型相同的另一个声明：一个类型声明可以遮蔽另一个类型声明，一个变量声明可以遮蔽另一个变量声明，一个方法声明可以遮蔽另一个方法声明。与其形成对照的是，变量声明可以遮掩类型和包声明，而类型声明也可以遮掩包声明。

当一个声明遮蔽了另一个声明时，简单名将引用到遮蔽声明中的实体。在本例中，**toString**引用的是从**Object**继承而来的**toString**方法。简单地说，**本身就属于某个范围的成员在该范围内与静态导入相比具有优先权**。这导致的后果之一就是与**Object**的方法具有相同名字的静态方法不能通过静态导入工具而得到使用。

既然你不能对**Arrays.toString**使用静态导入，那么你就应该用一个普通的导入声明来代替。下面就是**Arrays.toString**被正确使用的方式：

```
import java.util.Arrays;
class ImportDuty {
    static void printArgs(Object... args) {
        System.out.println(Arrays.toString(args));
    }
}
```

如果你特别强烈地想避免显式地限定`Arrays.toString`调用，那么你可以编写你自己的私有静态转发方法：

```
private static String toString(Object[] a) {
    return Arrays.toString(a);
}
```

静态导入工具所专门针对的情况是：程序中会重复地使用另一个类的静态元素，而每一次用到的时候都进行限定又会使程序乱成一锅粥。在这类情况下，静态导入工具可以显著地提高可读性。这比通过实现接口来继承其常量要安全得多，而实现接口这种做法是你从来都不应该采用的 [EJ Item 17]。然而，滥用静态导入工具也会损害可读性，因为这会使得静态成员的类在何处被使用显得非常不清晰。应该有节制地使用静态导入，只有在非常需要的情况下才使用它们。

对API设计者来说，要意识到当某个方法的名字已经出现在某个作用域内时，静态导入工具并不能被有效地作用于该方法上。这意味着静态导入不能用于那些与通用接口中的方法共享方法名的静态方法，而且也从来不能用于那些与`Object`中的方法共享方法名的静态方法。再次说明一下，本谜题所要说明的仍然是你在覆写之外的情况下使用名字重用通常都会产生混乱。我们通过重载、隐藏和遮掩看清楚了这一点，现在我们又通过遮蔽看到了同样的问题。

谜题72：终极危难

本谜题检验当你试图隐藏一个`final`域时将要发生的事情。下面的程序将做些什么呢？

```
class Jeopardy {
    public static final String PRIZE = "$64,000";
}

public class DoubleJeopardy extends Jeopardy {
    public static final String PRIZE = "2 cents";

    public static void main(String[] args) {
        System.out.println(DoubleJeopardy.PRIZE);
    }
}
```

解惑72：终极危难

因为在Jeopardy中的PRIZE域被声明为是public和final的，你可能会认为Java语言将阻止你在子类中重用该域名。毕竟，final类型的方法不能被覆写或隐藏。如果你尝试着运行该程序，就会发现它可以毫无问题地通过编译，并且将打印2 cents。出什么错了呢？

可以证明，final修饰符对方法和域而言，意味着某些完全不同的事情。对于方法，final意味着该方法不能被覆写（对实例方法而言）或者隐藏（对静态方法而言）[JLS 8.4.3.3]。对于域，final意味着该域不能被赋值超过一次[JLS 8.3.1.2]。关键字相同，但是其行为却完全不相关。

在该程序中，final域DoubleJeopardy.PRIZE隐藏了final域Jeopardy.PRIZE，其净损失达到了\$63 999.98。尽管我们可以隐藏一个域，但是通常这都是一个不好的念头。就像我们在谜题66中所讨论的，隐藏域可能会违反包容性，并且会混淆我们对类型与其成员之间的关系所产生的直觉。

如果你想保证在Jeopardy类中的奖金可以保留到子类中，那么你应该用一个final方法来代替final域：

```
class Jeopardy {
    private static final String PRIZE = "$64, 000";

    public static final String prize() {
        return PRIZE;
    }
}
```

对语言设计者来说，其教训是应该避免在不相关的概念之间重用关键字。一个关键字应该只在密切相关的概念之间重用，这样可以帮助程序员构建关于易混淆的语言特性之间的关系的印象。在Java的final关键字这一案例中，重用就导致了混乱。应该注意的是，作为一种有年头的语言来说，在无关的概念之间重用关键字是它的一种自然趋势，这样做可以避免引入新的关键字，而引入新的关键字会对语言的稳定性造成极大的损害。当语言设计者在考虑该怎么做时，总是在两害相权取其轻。

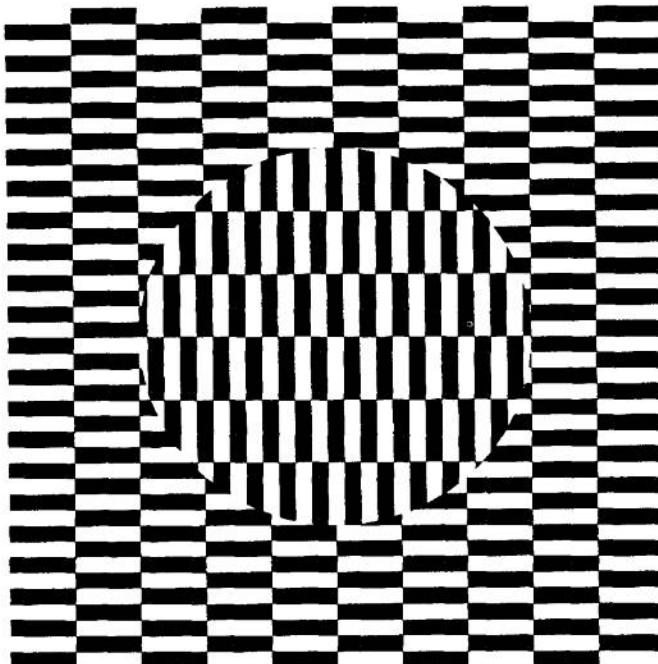
总之，要避免在无关的变量或无关的概念之间重用名字。对无关的概念使用有区别的名字有助于读者和程序员区分这些概念。

谜题73：隐私在公开

私有成员，即私有方法、域和类型这些概念的幕后思想是它们只是实现细节：一个类的实现者可以随意地添加一个新的私有成员，或者修改和移除一个旧的私有成员，而不需要担心对该类的客户造成任何损害。换句话说，私有成员被包含它们的类完全封装了。

遗憾的是，在这种严密的盔甲保护中仍然存在细小的裂缝。例如，序列化就可以打破这种封装。如果使一个类成为可序列化的，并且接受缺省的序列化形式，那么该类的私有实例域将成为其导出API的一部分[EJ Item 54,55]。当客户正在使用现有的被序列化对象时，对私有表示的修改将会导致异常或者是错误的行为。

但是编译期的错误又会怎么样呢？你能否写出一个final的“库”类和“客户”类，这两者都可以毫无问题地通过编译，然后在库类中添加一个私有成员，使得库类仍然能够编译，而客户类却再也不能编译了？



解惑73：隐私在公开

如果你的解惑方案是要对库类添加一个私有构造器，以抑制通过缺省的公共构造器而创建实例的行为，那么你只是一知半解。本谜题要求你添加一个私有成员，严格地讲，构造器不是成员[JLS 6.4.3]。

本谜题有数个解惑方案，其中一个是使用遮蔽：

```
package library;
public final class Api {
    // private static class String {}
    public static String newString() {
        return new String();
    }
}

package client;
import library.Api;
public class Client {
    String s = Api.newString();
}
```

如上编写，该程序就可以毫无问题地通过编译。如果我们不注释掉library.Api中的局部类String的私有声明，那么Api.newString方法就再也不会返回java.lang.String类型了，因此变量Client.s的初始化将不能通过编译：

```
client/Client.java:4: incompatible types
found: library.Api.String, required: java.lang.String
    String s = Api.newString();
                           ^

```

尽管我们所做的文本修改仅仅是添加了一个私有类声明，但是我们间接地修改了一个现有公共方法的返回类型，而这是一个不兼容的API修改，因为我们修改了一个被导出API所使用的名字的含义。

这种解惑方案的数个变种也都可以实现这个目的。被遮蔽类型也可以来自一个外围类而不是来自java.lang。你可以遮蔽一个变量而不是一个类型，而被遮蔽变量可以来自一个static import声明或者是来自一个外围类。

不修改类库的某个被导出成员的类型也可以解决本谜题。下面就是这样的一个解惑方案，它使用的是隐藏而不是遮蔽：

```

package library;
class ApiBase {
    public static final int ANSWER = 42;
}
public final class Api extends ApiBase {
    // private static final int ANSWER = 6 * 9;
}

package client;
import library.Api;
public class Client {
    int answer = Api.ANSWER;
}

```

如上编写，该程序就可以毫无问题地通过编译。如果我们不注释掉library.Api中的私有声明，那么客户类将不能通过编译：

```

client/Client.java:4: ANSWER has private access in library.Api
int answer = Api.ANSWER;
^

```

这个新的私有域Api.ANSWER隐藏了公共域ApiBase.ANSWER，而这个域本来是应该被继承到Api中的。因为新的域被声明为是private的，所以它不能被Client访问。这种解惑方案的数个变种也都可以实现这个目的。你可以用隐藏一个实例域去替代隐藏一个静态域，或者用隐藏一个类型去替代隐藏一个域。

你还可以用遮掩来解决本谜题。所有的解惑方案都是通过重用某个名字来破坏客户类。**重用名字是危险的；应该避免隐藏、遮蔽和遮掩。**是不是对此已经耳熟能详了？很好！

谜题74：同一性的危机

下面的程序是不完整的，它缺乏对Enigma的声明，这个类扩展自java.lang.Object。请为Enigma提供一个声明，它可以使该程序打印false：

```

public class Conundrum {
    public static void main(String[] args) {
        Enigma e = new Enigma();
        System.out.println(e.equals(e));
    }
}

```

噢，还有一件事：你不能覆写equals方法。

解惑74：同一性的危机

乍一看，这似乎不可能实现。因为Object.equals方法将测试对象的同一性，通过Enigma传递给equals方法的对象肯定是与其自身相同的。如果你不能覆写Object.equals方法，那么main方法必然打印true，对吗？

别那么快下结论，伙计。尽管本谜题禁止你覆写Object.equals方法，但是你是可以重载它的，这也就引出了下面的解惑方案：

```
final class Enigma {
    // Don't do this!
    public Boolean equals(Enigma other) {
        return false;
    }
}
```

尽管这个声明能够解决本谜题，但是它的做法确实非常不好。它违反了谜题58的建议：**如果同一个方法的两个重载版本都可以应用于某些参数，那么它们应该具有相同的行为。**在本例中，e.equals(e)和e.equals((Object)e)将返回不同的结果，其潜在的混乱是显而易见的。

然而，有一种解惑方案是不会违反这项建议的：

```
final class Enigma {
    public Enigma() {
        System.out.println(false);
        System.exit(0);
    }
}
```

可能会有些争论，这个解惑方案似乎违背了本谜题的精神：能够产生我们想要的输出的println调用出现在了构造器中，而不是在main方法中。然而，它确实解决了这个谜题，你不得不承认它很伶俐。

这里的教训，可以参阅前面的8个谜题和谜题58。如果你重载了一个方法，那么一定要确保所有的重载版本行为一致。

谜题75：头还是尾？

这个程序的行为在1.4版和5.0版的Java平台上会有些变化。这个程序在这些版本上会分别做些什么呢？（如果你只能访问5.0版本的平台，那么你可以在编译的时候使用`-source 1.4`标记，以此来模拟1.4版的行为。）

```
import java.util.Random;

public class CoinSide {
    private static Random rnd = new Random();

    public static CoinSide flip() {
        return rnd.nextBoolean() ?
            Heads.INSTANCE : Tails.INSTANCE;
    }

    public static void main(String[] args) {
        System.out.println(flip());
    }
}

class Heads extends CoinSide {
    private Heads() { }
    public static final Heads INSTANCE = new Heads();

    public String toString() {
        return "heads";
    }
}

class Tails extends CoinSide {
    private Tails() { }
    public static final Tails INSTANCE = new Tails();

    public String toString() {
        return "tails";
    }
}
```

解惑75：头还是尾？

该程序看起来根本没有使用5.0版的任何新特性，因此很难看出来为什么它们在行为上应该有差异。事实上，该程序在1.4或更早版本的平台上是不能编译的：

```
CoinSide.java:7:
    incompatible types for ?: neither is a subtype of the other
        second operand: Heads
        third operand : Tails
            return rnd.nextBoolean() ?
                    ^

```

条件操作符（?:）的行为在5.0版本之前是非常受限的[JLS2 15.25]。当第二个和第三个操作数是引用类型时，条件操作符要求它们其中的一个必须是另一个的子类型。Heads和Tails彼此都不是对方的子类型，所以这里就产生了一个错误。为了让这段代码能够编译，你可以将其中一个操作数转型为二者的公共超类：

```
return rnd.nextBoolean() ?
    (CoinSide)Heads.INSTANCE : Tails.INSTANCE;
```

在5.0或更新的版本中，Java语言显得更加宽大了，条件操作符在第二个和第三个操作数是引用类型时总是合法的。其结果类型是这两种类型的最小公共超类。公共超类总是存在的，因为Object是每一个对象类型的超类型。在实际使用中，这种变化的主要结果就是条件操作符做正确的事情的情况更多了，而给出编译期错误的情况更少了。对于我们当中语言菜鸟来说，作用于引用类型的条件操作符的结果所具备的编译期类型与在第二个和第三个操作数上调用下面的方法的结果相同：

```
<T> T choose(T a,T b) { }
```

本谜题所展示的问题在1.4和更早的版本中发生得相当频繁，迫使你必须插入只是为了遮掩你的代码的真实目的而进行的转型。这就是说，该谜题本身是人为制造的。在5.0版本之前，使用类型安全的枚举模式来编写CoinSide对程序员来说会显得更自然一些 [EJ Item 21]：

```
import java.util.Random;
public class CoinSide {
```

```
public static final CoinSide HEADS = new CoinSide("heads");
public static final CoinSide TAILS = new CoinSide("tails");

private final String name;
private CoinSide(String name) {
    this.name = name;
}

public String toString() {
    return name;
};

private static Random rnd = new Random();
public static CoinSide flip() {
    return rnd.nextBoolean() ? HEADS : TAILS;
}

public static void main(String[] args) {
    System.out.println(flip());
}
}
```

在5.0或更新的版本中，自然会将CoinSide当作是一个枚举类型来编写：

```
public enum CoinSide {
    HEADS, TAILS;

    public String toString() {
        return name().toLowerCase();
    }
    // flip and main same as in 1.4 implementation above
}
```

本谜题的教训是：**应该升级到最新的Java平台版本上**。较新的版本都包含许多让程序员更轻松的改进，你并不需要费力去学习怎样利用所有的新特性，有些新特性不需要你付出任何努力就可以给你带来实惠。对语言和类库的设计者来说，得到的教训是：不要让程序员去做那些语言或类库本可以帮他们做的事。

名字重用的术语表

本章中的大多数谜题都是基于名字重用的。本节将总结各种不同的名字重用形式。

覆盖

一个实例方法可以覆盖（override）在其超类中可访问到的具有相同签名的所有实例方法[JLS 8.4.8.1]，从而使能了动态分派（dynamic dispatch）；换句话说，VM将基于实例的运行期类型来选择要调用的覆盖方法[JLS 15.12.4.4]。覆盖是面向对象编程技术的基础，并且是唯一没有被普遍劝阻的名字重用形式：

```
class Base {
    public void f() { }
}

class Derived extends Base {
    public void f() { } // overrides Base.f()
}
```

隐藏

一个域、静态方法或成员类型可以分别隐藏（hide）在其超类中可访问到的具有相同名字（对方法而言就是相同的方法签名）的所有域、静态方法或成员类型。隐藏一个成员将阻止其被继承[JLS 8.3, 8.4.8.2, 8.5]：

```
class Base {
    public static void f() { }
}

class Derived extends Base {
    public static void f() { } // hides Base.f()
}
```

重载

在某个类中的方法可以重载（overload）另一个方法，只要它们具有相同的名字和不同的签名。由调用所指定的重载方法是在编译期选定的[JLS 8.4.9, 15.12.2]：

```
class CircuitBreaker {
    public void f(int i) { } // int overloading
```

```
public void f(String s) { } // String overloading
}
```

遮蔽

一个变量、方法或类型可以分别遮蔽（shadow）在一个闭合的文本范围内的具有相同名字的所有变量、方法或类型。如果一个实体被遮蔽了，那么你用它的简单名是无法引用到它的；根据实体的不同，有时你根本就无法引用到它[JLS 6.3.1]：

```
class WhoKnows {
    static String sentence = "I don't know.";

    public static void main(String[] args) {
        String sentence = "I know!";    // shadows static field
        System.out.println(sentence);   // prints local variable
    }
}
```

尽管遮蔽通常是被劝阻的，但是有一种通用的惯用法确实涉及遮蔽。构造器经常将来自其所在类的某个域名重用为一个参数，以传递这个命名域的值。这种惯用法并不是没有风险，但是大多数Java程序员都认为这种风格带来的实惠要超过其风险：

```
class Belt {
    private final int size;
    public Belt(int size) { // Parameter shadows Belt.size
        this.size = size;
    }
}
```

遮掩

一个变量可以遮掩（obscure）具有相同名字的一个类型，只要它们都在同一个范围内：如果这个名字被用于变量与类型都被许可的范围，那么它将引用到变量上。相似地，一个变量或一个类型可以遮掩一个包。遮掩是惟一一种两个名字位于不同的名字空间的名字重用形式，这些名字空间包括：变量、包、方法或类型。如果一个类型或一个包被遮掩了，那么你不能通过其简单名引用到它，除非是在这样一个上下文环境中，即语法只允许在其名字空间中出现一种名字。遵守命名习惯就可以极大地消除产生遮掩的可能性[JLS 6.3.2, 6.5]：

```
public class Obscure {  
    static String System; // Obscures type java.lang.System  
  
    public static void main(String[] args) {  
        // Next line won't compile: System refers to static field  
        System.out.println("hello, obscure world!");  
    }  
}
```

更多库之谜

在本章所描述的谜题讲述了更多的关于库的高级话题。诸如多线程、反射和I/O。

谜题76：乒乓

下面的程序全部是由同步化的静态方法组成的。那么它会打印出什么呢？在你每次运行这段程序的时候，它都能保证会打印出相同的内容吗？

```
public class PingPong{
    public static synchronized void main(String[] a) {
        Thread t = new Thread() {
            public void run(){ pong(); }
        };
        t.run();
        System.out.print( "Ping" );
    }

    static synchronized void pong() {
        System.out.print( "Pong" );
    }
}
```

解惑76：乒乓

在多线程程序中，通常正确的观点是程序每次运行的结果都有可能发生变化，但是上面这段程序总是打印出相同的内容。在一个同步化的静态方法执行之前，它会获取与它的Class对象相关联的一个管程（monitor）锁[JLS 8.4.3.6]。所以在上面的程序中，主线程会在创建第二个线程之前获得与PingPong.class相关联的那个锁。只要主线程占有着这个锁，第二个线程就不可能执行同步化的静态方法。具体地讲，在main方法打印了Ping并且执行结束之后，第二个线程才能执行pong方法。只有当主线程放弃那个锁的时候，第二个线程才被允许获得这个锁并且打印Pong。根据以上的分析，我们似乎可以确信这个程序应该总是打印PingPong。但是这里有一个小问题：当你尝试着运行这个程序的时候，你会发现它总是会打印PongPing。到底发生了什么呢？

正如它看起来那样奇怪，这段程序并不是一个多线程程序。不是一个多线程程序？怎么可能呢？它肯定会生成第二个线程啊。喔，对的，它确实是创建了第二个线程，但是它从未启动这个线程。相反地，主线程会调用那个新的线程实例的run方法，这个run方法会在主线程中同步地运行。由于一个线程可以重复地获得某个相同的锁[JLS 17.1]，所以当run方法调用pong方法的时候，主线程就被允许再次获得与PingPong.class相关联的锁。pong方法打印了Pong并且返回到了run方法，而run方法又返回到main方法。最后，main方法打印了Ping，这就解释了我们看到的输出结果是怎么来的。

要订正这个程序很简单，只需将t.run改写成t.start。这么做之后，这个程序就会如你所愿的总是打印出PingPong了。

这个教训很简单：当你想调用一个线程的start方法时要多加小心，别弄错成调用这个线程的run方法了。遗憾的是，这个错误实在是太普遍了，而且它可能很难被发现。或许这个谜题的教训应该是针对API的设计者的：如果一个线程没有一个公共的run方法，那么程序员就不可能意外地调用到它。Thread类之所以有一个公共的run方法，是因为它实现了Runnable接口，但是这种方式并不是必须的。另外一种可选的设计方案是：使用组合来替代接口继承，让每个Thread实例都封装一个Runnable。正如谜题47中所讨论的，组合通常比继承更可取。这个谜题说明了上述的原则甚至对于接口继承也是适用的。

谜题77：乱锁之妖

下面的这段程序模拟了一个小车间。程序首先启动了一个工人线程，该线程在停止时间到来之前会一直工作（至少是假装在工作），然后程序安排了一个定时器任务（timer task）用来模拟一个恶毒的老板，他会试图阻止停止时间的到来。最后，主线程作为一个善良的老板会告诉工人停止时间到了，并且等待工人停止工作。那么这个程序会打印什么呢？

```
import java.util.*;
public class Worker extends Thread {
    private volatile boolean quittingTime = false;
    public void run() {
        while (!quittingTime)
            pretendToWork();
        System.out.println("Beer is good");
    }
    private void pretendToWork() {
        try {
            Thread.sleep(300); // Sleeping on the job?
        } catch (InterruptedException ex) { }
    }
    // It's quitting time, wait for worker-Called by good boss
    synchronized void quit() throws InterruptedException {
        quittingTime = true;
        join();
    }
    // Rescind quitting time-Called by evil boss
    synchronized void keepWorking() {
        quittingTime = false;
    }
    public static void main(String[] args)
        throws InterruptedException {
        final Worker worker = new Worker();
        worker.start();

        Timer t = new Timer(true); // Daemon thread
        t.schedule(new TimerTask() {
            public void run() { worker.keepWorking(); }
        }, 500);

        Thread.sleep(400);
        worker.quit();
    }
}
```

解惑77：乱锁之妖

想要探究这个程序到底做了什么的最好方法就是手动地模拟一下它的执行过程。下面是一个近似的时间轴，这些时间点的数值是相对于程序的开始时刻进行计算的：

- **300ms**: 工人线程去检查易变的quittingTime域，看看停止时间是否已经到了。这个时候并没有到停止时间，所以工人线程会回去继续“工作”。
- **400ms**: 作为善良的老板的主线程会去调用工人线程的quit方法。主线程会获得工人线程实例上的锁（因为quit是一个同步化的方法），将quittingTime的值设为true，并且调用工人线程上的join方法。这个对join方法的调用并不会马上返回，而是会等待工人线程执行完毕。
- **500ms**: 作为恶毒的老板定时器任务开始执行。它将试图调用工人线程的keepWorking方法，但是这个调用将会被阻塞，因为keepWorking是一个同步化的方法，而主线程当时正在执行工人线程上的另一个同步化方法（quit方法）。
- **600ms**: 工人线程会再次检查停止时间是否已经到来。由于quittingTime域是易变的，那么工人线程肯定会看到新的值true，所以它会打印Beer is good并结束运行。这会让主线程对join方法的调用执行返回，随后主线程也结束了运行。而定时器线程是后台的，所以它也会随之结束运行，整个程序也就结束了。

所以，我们会认为程序将运行不到1s，打印Beer is good，然后正常的结束。但是当你尝试运行这个程序的时候，你会发现它没有打印任何东西，而是一直处于挂起状态（没有结束）。我们的分析哪里出错了呢？

其实，并没有什么可以保证上述几个交叉的事件会按照上面的时间轴发生。无论是Timer类还是Thread.sleep方法，都不能保证具有实时性。这就是说，由于这里计时的粒度太粗，所以上述几个事件很有可能会在时间轴上互有重叠地交替发生。100ms对于计算机来说是一段很长的时间。此外，这个程序被重复地挂起；看起来好像有什么其他的东西在工作着，事实上，确实是有这种东西。

我们的分析存在着一个基本的错误。在500ms时，当作为恶毒老板的定时器任务运行时，根据时间轴的显示，它对keepWorking方法的调用会被阻塞，因为keepWorking是一个同步化的方法并且主线程正在同一个对象上执行着同步化方法quit（在Thread.join中等待着）。这些都是对的，keepWorking确实是一个同步化的方法，并且主线程确实正在同一个对象上执行着同步化的quit方法。即使如此，定时器线程仍然

可以获得这个对象上的锁，并且执行keepWorking方法。这是如何发生的呢？

问题的答案涉及了Thread.join的实现。这部分内容在关于该方法的文档（JDK文档）中是找不到的，至少在迄今为止发布的文档中如此，也包括5.0版。在内部，`Thread.join`方法在表示正在被连接的那个`Thread`实例上调用`Object.wait`方法。这样就在等待期间释放了该对象上的锁。在我们的程序中，这就使得作为恶毒老板的定时器线程能够堂而皇之的将`quittingTime`重新设置成`false`，尽管此时主线程正在执行同步化的`quit`方法。这样的结果是，工人线程永远不会看到停止时间的到来，它会永远运行下去。作为善良的老板的主线程也就永远不会从`join`方法中返回了。

使这个程序产生了预料之外的行为的根本原因就是`WorkerThread`类的编写者使用了实例上的锁来确保`quit`方法和`keepWorking`方法的互斥，但是这种用法与超类（`Thread`）内部对该锁的用法发生了冲突。这里的教训是：除非有关于某个类的详细说明作为保证，否则千万不要假设库中的这个类对它的实例或类上的锁会做（或者不会做）某些事情。对于库的任何调用都可能会产生对`wait`、`notify`、`notifyAll`方法或者某个同步化方法的调用。所有这些，都可能对应用级的代码产生影响。

如果你需要获得某个锁的完全控制权，那么就要确定没有任何其他人能够访问到它。如果你的类扩展了库中的某个类，而这个库中的类可能使用了它的锁，或者如果某些不可信的人可能会获得对你的类的实例的访问权，那么请不要使用与这个类或它的实例自动关联的那些锁。取而代之的，你应该在一个私有的域中创建一个单独的锁对象。在5.0版本发布之前，用于这种锁对象的正确类型只有`Object`或者它的某个普通的子类。从5.0版本开始，`java.util.concurrent.locks`提供了2种可选方案：`ReentrantLock`和`ReentrantReadWriteLock`。相对于`Object`类，这2个类提供了更好的灵活性，但是它们使用起来也要更麻烦一点。它们不能被用在同步化的语句块中，而且必须辅以`try-finally`语句对其进行显式的获取和释放。

订正这个程序最直接的方法是添加一个`Object`类型的私有域作为锁，并且在`quit`和`keepWorking`方法中对这个锁对象进行同步。通过上述修改之后，该程序就会打印出我们所期望的Beer is good。可以看出，该程序能够产生正确行为并不依赖于它必须遵从我们前面分析的时间轴：

```
private final Object lock = new Object();

// It's quitting time, wait for worker - Called by good boss
void quit() throws InterruptedException{
    synchronized (lock){
```

```

        quittingTime = true;
        join();
    }
}

// Rescind quitting time - Called by evil boss
void keepWorking(){
    synchronized(lock){
        quittingTime = false;
    }
}

```

另外一种可以修复这个程序的方法是让Worker类实现Runnable而不是扩展Thread，然后在创建每个工人线程的时候都使用Thread(Runnable)构造器。这样可以将每个Worker实例上的锁与其线程上的锁进行解耦。这是一个规模稍大一些的重构，就把它留给读者作为练习。

正如库类对锁的使用会干扰应用程序一样，应用程序中对锁的使用也会干扰库类。例如，在迄今为止发布的所有版本的JDK（包括5.0版本）中，为了创建一个新的Thread实例，系统都会去获取Thread类上的锁。而执行下面的代码就可以阻止任何新线程的创建：

```

synchronized(Thread.class){
    Thread.sleep(Long.MAX_VALUE);
}

```

总之，永远不要假设库类会（或者不会）对它的锁做某些事情。为了隔离你自己的程序与库类对锁的使用，除了那些专门设计用来被继承的库类之外，请避免继承其他库类 [EJ Item 15]。为了确保你的锁不会遭受外部的干扰，可以将它们设为私有以阻止其他人访问它们。

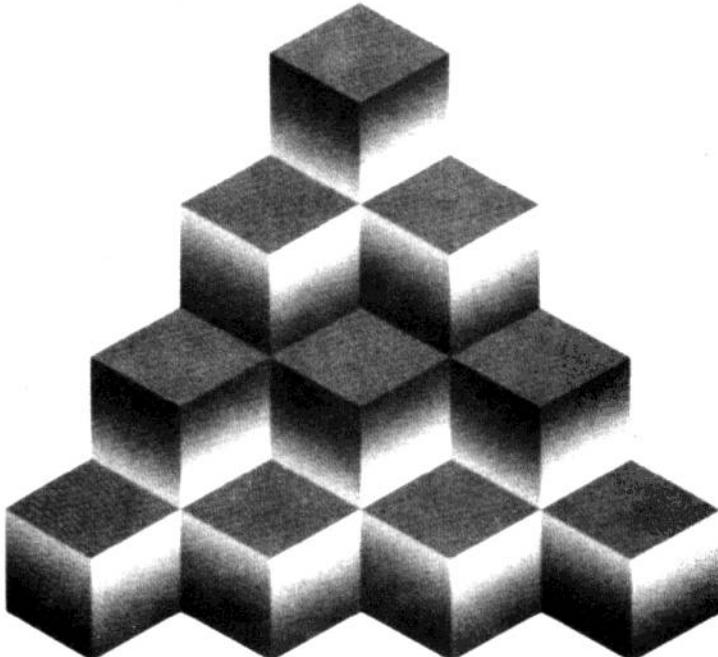
对于语言设计者来说，需要考虑的是为每个对象都关联一个锁是否是合适的。如果你决定这么做了，就需要考虑限制对这些锁的访问。在Java中，锁实际上是对象的公共属性，或许它们变为私有的会更有意义。同时请记住在Java语言中，一个对象实际上就是一个锁：你在对象本身之上进行同步。如果每个对象都有一个锁，而且你可以通过调用一个访问器方法来获得它，这样或许会更有意义。

谜题78：反射的污染

这个谜题举例说明了一个关于反射的简单应用。这个程序会打印出什么呢？

```
import java.util.*;
import java.lang.reflect.*;

public class Reflector {
    public static void main(String[] args) throws Exception {
        Set<String> s = new HashSet<String>();
        s.add("foo");
        Iterator it = s.iterator();
        Method m = it.getClass().getMethod("hasNext");
        System.out.println(m.invoke(it));
    }
}
```



解惑78：反射的污染

这个程序首先创建了一个只包含单个元素的集合，获得了该集合上的迭代器，然后利用反射调用了迭代器的hasNext方法，最后打印出此方法调用的结果。由于该迭代器尚未返回该集合中那个惟一的元素，hasNext方法应该返回true。然而，运行这个程序却得到了截然不同的结果：

```
Exception in thread "main" IllegalAccessException:  
  Class Reflector can not access a member of class HashMap  
  $HashIterator with modifiers "public"  
    at Reflection.ensureMemberAccess(Reflection.java:65)  
    at Method.invoke(Method.java:578)  
    at Reflector.main(Reflector.java:11)
```

这是怎么发生的呢？正如这个异常所显示的，hasNext方法当然是公共的，所以它在任何地方都是可以被访问的。那么为什么这个基于反射的方法调用是非法的呢？

这里的问题并不在于该方法的访问级别，而在于该方法所在的类型的访问级别。这个类型所扮演的角色和一个普通方法调用中的限定类型是相同的[JLS 13.1]。在这个程序中，该方法是从某个类中选择出来的，而这个类型是由从`it.getClass`方法返回的`Class`对象表示的。这是迭代器的动态类型，它恰好是私有的嵌套类`java.util.HashMap.KeyIterator`。出现`IllegalAccessException`异常的原因就是这个类不是公共的，它来自另外一个包：访问位于其他包中的非公共类型的成员是不合法的[JLS 6.6.1]。

无论是一般的访问还是通过反射的访问，上述的禁律都是有效的。下面这段没有使用反射的程序也违反了这条规则。

```
package library;  
public class Api {  
    static class PackagePrivate {}  
    public static PackagePrivate member = new PackagePrivate();  
}  
  
package client;  
import library.Api;  
class Client {  
    public static void main(String[] args) {  
        System.out.println(Api.member.hashCode());  
    }  
}
```

尝试编译这段程序会得到如下的错误：

```
Client.java:5: Object.hashCode() isn't defined in a public
class or interface; can't be accessed from outside package
    System.out.println(Api.member.hashCode());
                                ^

```

这个错误与前面那个由含有反射的程序所产生的运行期错误具有相同的意义。Object类型和hashCode方法都是公共的。问题在于hashCode方法是通过一个限定类型调用的，但用户访问不到这个类型。该方法调用的限定类型是library.Api.PackagePrivate，这是一个位于其他包的非公共类型。

这并不意味着Client就不能调用Api.member的hashCode方法。要做到这一点，只需要使用一个可访问的限定类型即可，在这里可以将Api.member转型成Object。经过这样的修改之后，Client类就可以顺利地编译和运行了：

```
System.out.println(((Object)Api.member).hashCode());
```

实际上，这个问题并不会在普通的非反射的访问中出现，因为API的编写者在他们的公共API中只会使用公共的类型。即使这个问题有可能发生，它也会以编译期错误的形式显现出来，所以比较容易修改。而使用反射的访问就不同了，`object.getClass().getMethod("methodName")`这种惯用法虽然很常见，但是却有问题，它不应该被使用。就像我们在前面的程序中看到的那样，这种用法很容易在运行期产生一个**IllegalAccessException**。

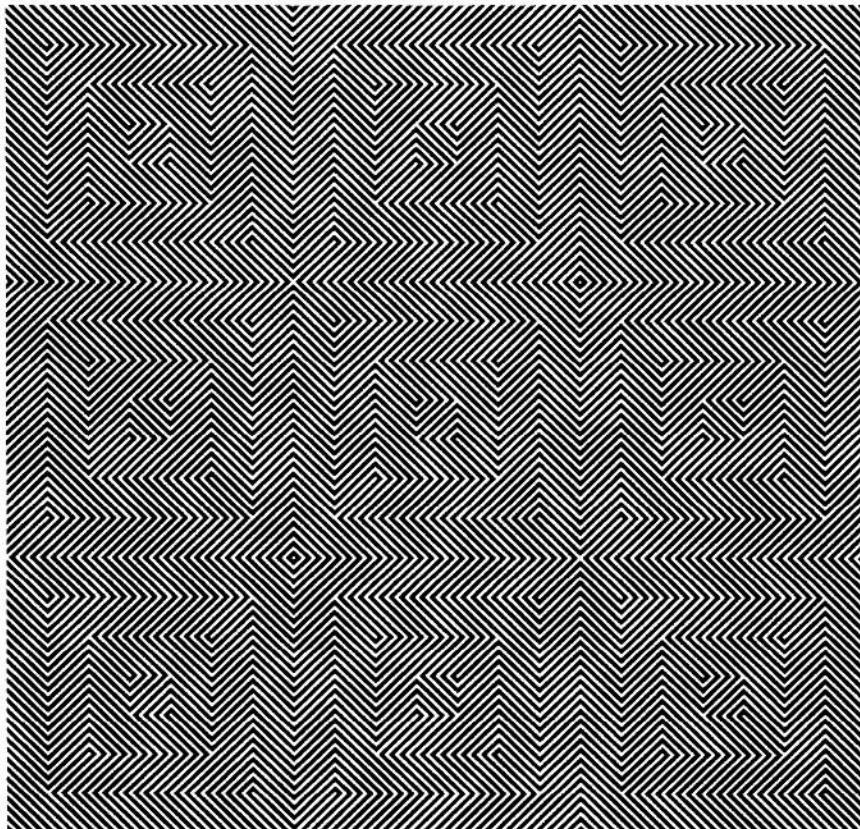
在使用反射访问某个类型时，请使用表示某种可访问类型的class对象。回到我们前面的那个程序，`hasNext`方法是声明在一个公共类型java.util.Iterator中的，所以它的类对象应该被用来进行反射访问。经过这样的修改后，这个Reflector程序就会打印出true：

```
Method m = Iterator.class.getMethod("hasNext");
```

你完全可以避免这一类的问题，你应该只有在实例化时才使用反射，而方法调用都通过使用接口进行[EJ Item 35]。这种使用反射的用法，可以将那些调用方法的类与那些实现这些方法的类隔离开，并且提供了更高程度的类型安全。这种用法在“服务提供者框架”(Service Provider Framework)中很常见。这种模式并不能解决反射访问中的所有问题，但是如果它可以解决你所遇到的问题，请务必使用它。

总之，访问其他包中的非公共类型的成员是不合法的，即使这个成员同时也被声明

为某个公共类型的公共成员也是如此。不论这个成员是否是通过反射被访问的，上述规则都是成立的。这个问题很有可能只在反射访问中才会出现。对于平台的设计者来说，这里的教训与谜题67中的一样，应该让错误症状尽可能清晰地显示出来。对于运行时的异常和编译期的提示都还有些东西需要改进。



谜题79：狗狗的幸福生活

下面的这个类模拟了一个家庭宠物的生活。main方法创建了一个Pet实例，用它来表示一只名叫Fido的狗，然后让它运行。虽然绝大部分的狗都在后院里奔跑，这只狗却是在后台运行。那么，这个程序会打印出什么呢？

```
public class Pet {  
    public final String name;  
    public final String food;  
    public final String sound;  
  
    public Pet(String name, String food, String sound) {  
        this.name = name;  
        this.food = food;  
        this.sound = sound;  
    }  
  
    public void eat() {  
        System.out.println(name + ": Mmmmm, " + food);  
    }  
  
    public void play() {  
        System.out.println(name + ": " + sound + " " + sound);  
    }  
  
    public void sleep() {  
        System.out.println(name + ": Zzzzzz...");  
    }  
  
    public void live() {  
        new Thread() {  
            public void run() {  
                while(true) {  
                    eat();  
                    play();  
                    sleep();  
                }  
            }  
        }.start();  
    }  
  
    public static void main(String[] args) {  
        new Pet("Fido", "beef", "Woof").live();  
    }  
}
```

解惑79：狗狗的幸福生活

main方法创建了一个用来表示Fido的Pet实例，并且调用了它的live方法。然后，live方法创建并且启动了一个线程，该线程反复的调用其外围的Pet实例的eat、play和sleep方法，就这么一直进行下去。这些方法都会打印单独的一行，所以你会想到这个程序会反复地打印以下的三行：

```
Fido: Mmmmm, beef  
Fido: Woof Woof  
Fido: Zzzzzz...
```

但是如果你尝试运行这个程序，你会发现它甚至不能通过编译。而产生的编译错误信息没有什么用处：

```
Pet.java:28: cannot find symbol  
symbol: method sleep()  
        sleep();  
               ^
```

为什么编译器找不到那个符号呢？这个符号确实是白纸黑字地写在那里。与谜题74一样，这个问题源自重载解析过程的细节。编译器会在包含有正确名称的方法的最内层范围内查找需要调用的方法[JLS 15.12.1]。在我们的程序中，对于对sleep方法的调用，这个最内层的范围就是包含有该调用的匿名类，这个类继承了Thread.sleep(long)方法和Thread.sleep(long, int)方法，它们是该范围内仅有的名称为sleep的方法，但是由于它们都带有参数，所以都不适用于这里的调用。由于该方法调用的2个候选方法都不适用，所以编译器就打印出了错误信息。

从Thread那里继承到匿名类中的两个sleep方法遮蔽[JLS 6.3.1]了我们想要调用的sleep方法。正如你在谜题71和谜题73中所看到的那样，你应该避免遮蔽。在这个谜题中的遮蔽是间接地无意识地发生的，这使得它更加“阴险”。

订正这个程序的一个比较显而易见的方法，就是把Pet中的sleep方法的名字改成snooze, doze或者nap。订正该程序的另一个方法，是在方法调用的时候使用受限的this结构来显式地为该类命名。此时的调用就变成了Pet.this.sleep()。

订正该程序的第三个方法，也可以被证明是最好的方法，就是采纳谜题77的建议，使用**Thread(Runnable)**构造器来替代对**Thread**的继承。如果你这么做了，原有的问

题将会消失，因为那个匿名类不会再继承Thread.sleep方法。程序经过少许的修改，就可以产生我们想要的输出了，当然这里的输出可能有点无聊：

```
public void live() {
    new Thread(new Runnable() {
        public void run() {
            while(true) {
                eat();
                play();
                sleep();
            }
        }
    }).start();
}
```

总之，要小心无意间产生的遮蔽，并且要学会识别表明存在这种情况的编译器错误信息。对于编译器的编写者来说，应该尽力去产生那些对程序员来说有意义的错误消息。例如在我们的程序中，编译器应该可以警告程序员，存在着适用于方法调用但却被遮蔽的方法。

谜题80：更深层的反射

下面这个程序通过打印一个由反射创建的对象来产生输出。那么它会打印出什么呢？

```
public class Outer {
    public static void main(String[] args) throws Exception {
        new Outer().greetWorld();
    }

    private void greetWorld()throws Exception {
        System.out.println( Inner.class.newInstance() );
    }

    public class Inner{
        public String toString(){
            return "Hello world";
        }
    }
}
```

解惑80：更深层的反射

这个程序看起来是最普通的Hello World程序的又一个特殊的变体。Outer中的main方法创建了一个Outer实例，并且调用了它的greetWorld方法，该方法以字符串形式打印了通过反射创建的一个新的Inner实例。Inner的toString方法总是返回标准的问候语，所以程序的输出应该与往常一样，是Hello World。如果你尝试运行这个程序，你会发现实际的输出比较长，而且更加令人迷惑：

```
Exception in thread "main" InstantiationException: Outer$Inner
at java.lang.Class.newInstance0(Class.java:335)
at java.lang.Class.newInstance(Class.java:303)
at Outer.greetWorld(Outer.java:7)
at Outer.main(Outer.java:3)
```

为什么会抛出这个异常呢？从5.0版本开始，关于Class.newInstance的文档叙述道：如果那个Class对象“代表了一个抽象类，一个接口，一个数组类，一个原生类型，或者是空；或者这个类没有任何空的[也就是无参数的]构造器；或者实例化由于某些其他原因而失败，那么它就会抛出异常”[JAVA-API]。这里出现的问题满足上面的哪个条件呢？遗憾的是，异常信息没有提供任何提示。

在这些条件中，只有后两个有可能会满足：要么是Outer.Inner没有空的构造器，要么是实例化由于“某些其他原因”而失败了。正如Outer.Inner这种情况，当一个类没有任何显式的构造器时，Java会自动地提供一个不带参数的公共的缺省构造器[JLS 8.8.9]，所以它应该是有一个空构造器的。不过，newInstance方法调用失败的原因还是因为Outer.Inner没有空构造器！

一个非静态的嵌套类的构造器，在编译的时候会将一个隐藏的参数作为它的第一个参数，这个参数表示了它的直接外围实例（immediately enclosing instance）[JLS 13.1]。当你在代码中任何可以让编译器找到合适的外围实例的地方去调用构造器的时候，这个参数就会被隐式地传递进去。但是，上述的过程只适用于普通的构造器调用，也就是不使用反射的情况。当你使用反射调用构造器时，这个隐藏的参数就需要被显式地传递，对于Class.newInstance方法这是不可能做到的。要传递这个隐藏参数的惟一办法就是使用java.lang.reflect.Constructor。当对程序进行了这样的修改之后，它就可以正常的打印出Hello World了：

```

private void greetWorld() throws Exception {
    Constructor c = Inner.class.getConstructor(Outer.class);
    System.out.println(c.newInstance(Outer.this));
}

```

作为其他的选择，你可能观察到，Inner实例并不需要一个外围的Outer实例，所以可以将Inner类型声明为静态的。除非你确实是需要一个外围实例，否则你应该优先使用静态成员类而不是非静态成员类[EJ Item 18]。下面这个简单的修改就可以订正这个程序：

```
public static class Inner {...}
```

Java程序的反射模型和它的语言模型是不同的。反射操作处于虚拟机层次，暴露了很多从Java程序到class文件的翻译细节。这些细节当中的一部分由Java的语言规范来管理，但是其余的部分可能会随着不同的具体实现而有所不同。在Java语言的早期版本中，从Java程序到class文件的映射是很直接的，但是随着一些不能被虚拟机直接支持的高级语言特性的加入，如嵌套类、协变返回类型、泛型和枚举类型，使得这种映射变得越来越复杂了。

考虑到从Java程序到class文件的映射的复杂度，请避免使用反射来实例化内部类。更一般地讲，当我们在用高级语言特性定义的程序元素之上使用反射的时候，一定要小心，从反射的视角观察程序可能不同于从代码的视角去观察它。请避免依赖那些没有被语言规范所管理的翻译细节。对于平台的实现者来说，这里的教训就是要再次重申，请提供清晰准确的诊断信息。

谜题81：无法识别的字符化

下面这个程序看起来是在用一种特殊的方法做一件普通的事。那么，它会打印出什么呢？

```

public class Greeter {
    public static void main(String[] args) {
        String greeting = "Hello World";
        for(int i = 0; i < greeting.length(); i++)
            System.out.write(greeting.charAt(i));
    }
}

```

解惑81：无法识别的字符化

尽管这个程序有点奇怪，但是我们没有理由怀疑它会产生不正确的行为。它将 "Hello World" 写入了 `System.out`，每次写一个字符。你可能会意识到 `write` 方法只会使用其输入参数的低位字节。所以当 "Hello World" 含有任何外来字符的时候，可能会造成一些麻烦，但这里不会：因为 "Hello World" 完全是由 ASCII 字符组成的。无论你是每次打印一个字符，还是一次全部打印，结果都应该是一样的：这个程序应该打印 `Hello World`。然而，如果你运行该程序，就会发现它不会打印任何东西。那句问候语到哪里去了？难道是程序认为它并不令人愉快？

这里的问题在于 `System.out` 是带有缓冲的。`Hello World` 中的字符被写入了 `System.out` 的缓冲区，但是缓冲区从来都没有被刷新。大多数的程序员认为，当有输出产生的时候 `System.out` 和 `System.err` 会自动地进行刷新，这并不完全正确。这两个流都属于 `PrintStream` 类型，在 5.0 版 [Java-API] 中，有关这个类型的文档叙述道：

一个 `PrintStream` 可以被创建为自动刷新的；这意味着当一个字节数组被写入，或者某个 `println` 方法被调用，或者一个换行字符或字节 ('`\n`') 被写入之后，`PrintStream` 类型的 `flush` 方法就会被自动地调用。

`System.out` 和 `System.err` 所引用的流确实是 `PrintStream` 的能够自动刷新的变体，但是上面的文档中并没有提及 `write(int)` 方法。有关 `write(int)` 方法的文档叙述道：将指定的 `byte` 写入流。如果这个 `byte` 是一个换行字符，并且流可以自动刷新，那么 `flush` 方法将被调用 [Java-API]。实际上，`write(int)` 是惟一一个在自动刷新功能开启的情况下不刷新 `PrintStream` 的输出方法。

令人好奇的是，如果这个程序用 `print(char)` 去替代 `write(int)`，它就会刷新 `System.out` 并打印出 `Hello World`。这种行为与 `print(char)` 的文档是矛盾的，因为其文档叙述道 [Java-API]：

打印一个字符：这个字符将根据平台缺省的字符编码方式翻译成一个或多个字节，并且这些字节将完全按照 `write(int)` 方法的方式写出。

类似地，如果程序改用 `print(String)`，它也会对流进行刷新，虽然文档中是禁止这么做的。相应的文档确实应该被修改为描述该方法的实际行为，而修改方法的行为则会破坏稳定性。

修改这个程序最简单的方法就是在循环之后加上一个对 `System.out.flush` 方法的

调用。经过这样的修改之后，程序就会正常地打印出Hello World。当然，更好的办法是重写这个程序，使用我们更熟悉的System.out.println方法在控制台上产生输出。

这个谜题的教训与谜题23一样：**尽可能使用熟悉的惯用法；如果你不得不使用陌生的API，请一定要参考相关的文档。**这里有3条教训给API的设计者们：请让你们的方法的行为能够清晰地反映在方法名上；请清楚而详细地给出这些行为的文档；请正确地实现这些行为。

谜题82：啤酒爆炸

这一章的许多谜题都涉及了多线程，而这个谜题涉及多进程。如果你用一行命令行带上参数slave去运行这个程序，它会打印什么呢？如果你使用的命令行不带任何参数，它又会打印什么呢？

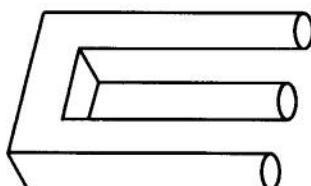
```
public class BeerBlast {  
    static final String COMMAND = "java BeerBlast slave";  
    public static void main(String[] args) throws Exception {  
        if(args.length == 1 && args[0].equals("slave")) {  
            for(int i = 99; i > 0; i--) {  
                System.out.println( i +  
                    " bottles of beer on the wall" );  
                System.out.println(i + " bottles of beer");  
                System.out.println(  
                    "You take one down, pass it around," );  
                System.out.println( (i-1) +  
                    " bottles of beer on the wall" );  
                System.out.println();  
            }  
        }else {  
            // Master  
            Process process = Runtime.getRuntime().exec(COMMAND);  
            int exitValue = process.waitFor();  
            System.out.println("exit value = " + exitValue);  
        }  
    }  
}
```

解惑82：啤酒爆炸

如果你使用参数`slave`来运行该程序，它就会打印出那首激动人心的名为“99 Bottles of Beer on the Wall”的童谣的歌词，这没有什么神秘的。如果你不使用该参数来运行这个程序，它会启动一个`slave`进程来打印这首歌谣，但是你看不到`slave`进程的输出。主进程会等待`slave`进程结束，然后打印出`slave`进程的退出值。根据惯例，0值表示正常结束，所以0就是你可能期望该程序打印的东西。如果你运行了程序，你可能会发现该程序只会悬挂在那，不会打印任何东西，看起来`slave`进程好像永远都在运行着。所以你可能会觉得你应该一直都能听到“99 Bottles of Beer on the Wall”这首童谣，即使是这首歌被唱走调了也是如此，但是这首歌只有99句。而且，计算机是很快的，你假设的情况应该是不存在的，那么这个程序出了什么问题呢？

这个秘密的线索可以在`Process`类的文档中找到，它叙述道：“由于某些本地平台只提供有限大小的缓冲，所以如果不能迅速地读取子进程的输出流，就有可能会导致子进程的阻塞，甚至是死锁” [Java-API]。这恰好就是这里所发生的事情：没有足够的缓冲空间来保存这首冗长的歌谣。为了确保`slave`进程能够结束，父进程必须排空它的输出流，而这个输出流从`master`线程的角度来看是输入流。下面的这个工具方法会在后台线程中完成这项工作：

```
static void drainInBackground(final InputStream is) {
    new Thread(new Runnable() {
        public void run() {
            try {
                while( is.read() >= 0 );
            } catch(IOException e) {
                // return on IOException
            }
        }
    }).start();
}
```



如果我们修改原有的程序，在等待slave进程之前调用这个方法，程序就会打印出0：

```

}else{
    // Master
    Process process = Runtime.getRuntime().exec(COMMAND);
    drainInBackground(process.getInputStream());
    int exitValue = process.waitFor();
    System.out.println(exit Value);
}

```

这里的教训是：为了确保子进程能够结束，你必须排空它的输出流；对于错误流也是一样，而且它可能会更麻烦，因为你无法预测进程什么时候会倾倒一些输出到这个流中。在5.0版本中，加入了一个名为ProcessBuilder的类用于排空这些流。它的redirectErrorStream方法将各个流合并起来，所以你只需要排空这一个流。如果你决定不合并输出流和错误流，你必须并行地排空它们。试图顺序地排空它们会导致子进程被挂起。

多年以来，很多程序员都被这个缺陷所刺痛。这里对于API设计者的教训是，Process类应该避免这个错误，也许应该自动地排空输出流和错误流，除非用户表示要读取它们。更一般地讲，API应该设计得更容易做出正确的事，而很难或不可能做出错误的事。

谜题83：诵读困难者的一神论

从前有一个人，他认为世上只有一只不寻常的狗，所以他写出了如下的类，将它作为一个单件（singleton）[Gamma95]：

```

public class Dog extends Exception {
    public static final Dog INSTANCE = new Dog();
    private Dog() {}
    public String toString() {
        return "Woof";
    }
}

```

结果证明这个人的做法是错误的。你能够在这个类的外部不使用反射来创建出第2个Dog实例吗？

解惑83：诵读困难者的一神论

这个类可能看起来像一个单件，但它并不是。问题在于，Dog扩展了Exception，而Exception实现了java.io.Serializable。这就意味着Dog是可序列化的，并且解序列会创建一个隐藏的构造器。正如下面的这段程序所演示的，如果你序列化了Dog.INSTANCE，然后对得到的字节序列进行解序列，最后你就会得到另外一个Dog。该程序打印的是false，表示新的Dog实例和原来的那个实例是不同的，并且它还打印了Woof，说明新的Dog实例也具有相应功能：

```
import java.io.*;

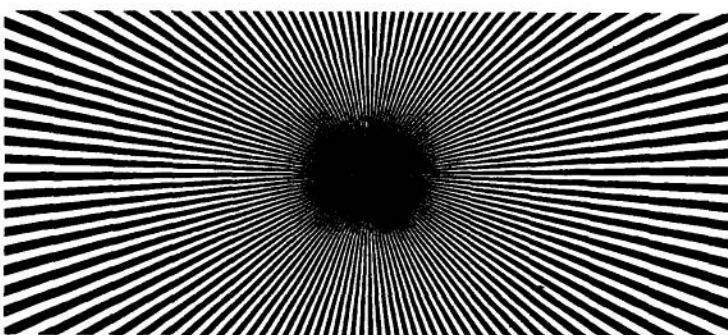
public class CopyDog{ // Not to be confused with copycat
    public static void main(String[] args) {
        Dog newDog = (Dog) deepCopy(Dog.INSTANCE);
        System.out.println(newDog == Dog.INSTANCE);
        System.out.println(newDog);
    }

    // This method is very slow and generally a bad idea!
    static public Object deepCopy(Object obj) {
        try{
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            new ObjectOutputStream(bos).writeObject(obj);
            ByteArrayInputStream bin =
                new ByteArrayInputStream(bos.toByteArray());
            return new ObjectInputStream(bin).readObject();
        } catch(Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

要订正这个问题，可以在Dog中添加一个readResolve方法，它可以将那个隐藏的构造器转变为一个隐藏的静态工厂，以返回原来那个的Dog[EJ Items 2,57]。在Dog中添加了这个方法之后，CopyDog将打印true而不是false，表示那个“副本”实际上就是原来的那个实例：

```
private Object readResolve() {
    // Accept no substitutes!
    return INSTANCE;
}
```

这个谜题的主要教训就是一个实现了**Serializable**的单件类，必须有一个**readResolve**方法，用以返回它的惟一的实例。一个次要的教训就是，有可能由于对一个实现了**Serializable**的类进行了扩展，或者由于实现了一个扩展自**Serializable**的接口，使得我们在无意中实现了**Serializable**。给平台设计者的教训是，隐藏的构造器，例如序列化中产生的那个，会让读者对程序行为产生错觉。



谜题84：戛然而止

在下面这个程序中，一个线程试图中断自己，然后检查中断是否成功。它会打印什么呢？

```
public class SelfInterruption {  
    public static void main(String[] args) {  
        Thread.currentThread().interrupt();  
  
        if(Thread.interrupted()) {  
            System.out.println("Interrupted: " +  
                Thread.interrupted());  
        } else {  
            System.out.println("Not interrupted: " +  
                Thread.interrupted());  
        }  
    }  
}
```

解惑84：戛然而止

虽然一个线程中断自己不是很常见，但这也不是没有听说过的。当一个方法捕捉到了一个InterruptedException异常，而且没有做好处理这个异常的准备时，那么这个方法通常会将该异常重新抛出。但是由于这是一个“受检查的异常”，所以只有在方法声明允许的情况下该方法才能够将异常重新抛出。如果不能重新抛出，该方法可以通过中断当前线程对异常“再构建”(reraise)。这种方式工作得很好，所以这个程序中的线程中断自己应该是没有任何问题的。所以，该程序应该进入if语句的第一个分支，打印出Interrupted:true。如果你运行该程序，你会发现并不是这样。但是它也没有打印Not interrupted:false，它打印的是Interrupted:false。

看起来该程序好像不能确定线程是否被中断了。当然，这种看法是毫无意义的。实际上发生的事情是，Thread.interrupted方法第一次被调用的时候返回了true，并且清除了线程的中断状态，所以在if-then-else语句的then分支中第2次调用该方法的时候，返回的就是false。调用**Thread.interrupted**方法总是会清除当前线程的中断状态。方法的名称没有为这种行为提供任何线索，而对于5.0版本，在相应的文档中一句话的总结同样具有误导性地叙述道：“测试当前的线程是否中断”[Java-API]。所以，可以理解为什么很多程序员都没有意识到Thread.interrupted方法会对线程的中断状态造成影响。

Thread类有两个方法可以查询一个线程的中断状态。另外一个方法是一个名为isInterrupted的实例方法，而它不会清除线程的中断状态。如果使用这个方法重写程序，它就会打印出我们想要的结果true：

```
public class SelfInterruption {
    public static void main(String[] args) {
        Thread.currentThread().interrupt();

        if(Thread.currentThread().isInterrupted()) {
            System.out.println("Interrupted: " +
                Thread.currentThread().isInterrupted());
        }else {
            System.out.println("Not interrupted: " +
                Thread.currentThread().isInterrupted());
        }
    }
}
```

这个谜题的教训是：不要使用**Thread.interrupted**方法，除非你想要清除当前线程的中断状态。如果你只是想查询中断状态，请使用**isInterrupted**方法。这里给API设计者们的教训是方法的名称应该用来描述它们主要功能。根据**Thread.interrupted**方法的行为，它的名称应该是**clearInterruptStatus**，因为相对于它对中断状态的改变，它的返回值是次要的。特别是当一个方法的名称并不完美的时候，文档是否能清楚地描述它的行为就显得非常重要的。

谜题85：惰性初始化

下面这个可怜的小类实在是太懒了，甚至于都不愿意用通常的方法进行初始化，所以它求助于后台线程。这个程序会打印什么呢？每次你运行它的时候都会打印出相同的东西吗？

```
public class Lazy {
    private static boolean initialized = false;

    static {
        Thread t = new Thread(new Runnable() {
            public void run() {
                initialized = true;
            }
        });
        t.start();
        try{
            t.join();
        }catch (InterruptedException e) {
            throw new AssertionError(e);
        }
    }

    public static void main(String[] args) {
        System.out.println(initialized);
    }
}
```

解惑85：惰性初始化

虽然有点奇怪，但是这个程序看起来很直观。静态域`initialized`初始时被设为`false`。然后主线程创建了一个后台线程，该线程的`run`方法将`initialized`的值设为`true`。主线程启动了后台线程之后，就调用了`join`方法等待它的结束。当后台线程完成运行的时候，毫无疑问`initialized`的值已经被设为`true`。当且仅当这个时候，调用了`main`方法的主线程会打印出`initialized`的值。如果是这样的话，程序肯定会打印出`true`吗？如果你运行该程序，你会发现它不会打印任何东西，它只是被挂起了。

为了理解这个程序的行为，我们需要模拟它初始化的细节。当一个线程访问一个类的某个成员的时候，它会去检查这个类是否已经被初始化。在忽略严重错误的情况下，有四种可能的情况[JLS 12.4.2]：

- (1) 这个类尚未被初始化。
- (2) 这个类正在被当前线程初始化：这是对初始化的递归请求。
- (3) 这个类正在被其他线程而不是当前线程初始化。
- (4) 这个类已经被初始化。

当主线程调用`Lazy.main`方法时，它会检查`Lazy`类是否已经被初始化。此时它并没有被初始化（情况1），所以主线程会记录下当前正在进行初始化，并开始对这个类进行初始化。按照我们前面的分析，主线程会将`initialized`的值设为`false`，创建并启动一个后台线程，该线程的`run`方法会将`initialized`设为`true`，然后主线程会等待后台线程执行完毕。此时，有趣的事情开始了。

那个后台线程调用了它的`run`方法。在该线程将`Lazy.initialized`设为`true`之前，它也会去检查`Lazy`类是否已经被初始化。这个时候，这个类正在被另外一个线程进行初始化（情况3）。在这种情况下，当前线程，也就是那个后台线程，会等待`Class`对象直到初始化完成。遗憾的是，那个正在进行初始化工作的线程，也就是主线程，正在等待着后台线程运行结束。因为这两个线程现在正相互等待着，该程序就死锁了。这就是所有的一切，真是遗憾。

有两种方法可以订正这个程序。到目前为止，最好的方法就是不要在类进行初始化的时候启动任何后台线程：有些时候，两个线程并不比一个线程好。更一般地讲，要让类的初始化尽可能地简单。订正这个程序的第二种方法就是让主线程在等待后台线程之前就完成类的初始化：

```
// Bad way to eliminate the deadlock. Complex and error prone
public class Lazy {
    private static boolean initialized = false;
    private static Thread t = new Thread(new Runnable() {
        public void run() {
            initialized = true;
        }
    });
    static {
        t.start();
    }
    public static void main(String[] args) {
        try{
            t.join();
        }catch (InterruptedException e) {
            throw new AssertionError(e);
        }
        System.out.println(initialized);
    }
}
```

虽然这么做确实消除了死锁，但是它却是一个非常不好的想法。主线程需要等待后台线程完成工作，但是其他的线程不需要这么做。一旦主线程完成了对Lazy类的初始化，其他线程就可以使用这个类了。这使得在initialized的值还是false的时候，其他线程就可以观察到它。

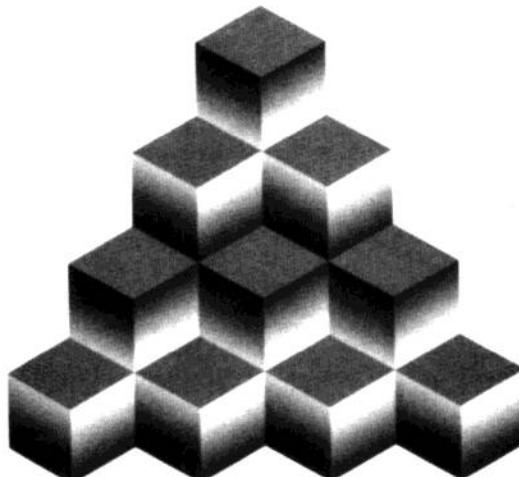
总之，在类的初始化期间等待某个后台线程很可能会造成死锁。要让类初始化的动作序列尽可能地简单。类的自动初始化被公认为是语言设计上的难题，Java的设计者们在这个方面做得很不错。如果你写了一些复杂的类初始化代码，很多种情况下，你这是在搬起石头砸自己的脚。

高级谜题

本章所描述的谜题涉及比较高级的主题，比如嵌套类、泛型、序列化和二进制兼容性。

谜题86：有害的括号垃圾

你能否举出这样一个合法的Java表达式，只要对它的某个子表达式加上括号就可以使其成为不合法的表达式，而添加的括号只是为了注解未加括号时赋值的顺序？



解惑86：有害的括号垃圾

插入一对用来注解现有赋值顺序的括号对程序的合法性似乎是应该没有任何影响的。事实上，绝大多数情况下确实是沒有影响的。但是，在两种情况下，插入一对看上去沒有影响的括号可能会令合法的Java程序变得不合法。这种奇怪的情况是由数值的二进制补码的不对称性引起的，就像在谜题33和谜题64中所讨论的那样。

你可能会联想到，最小的int类型负数其绝对值比最大的int类型正数大1：Integer.MIN_VALUE是 -2^{31} ，即-2 147 483 648，而Integer.MAX_VALUE是 $2^{31}-1$ ，即2 147 483 647。Java不支持负的十进制字面常量；int和long类型的负数常量都是由正数十进制字面常量前加一元负操作符（-）构成。这种构成方式是由一条特殊的语言规则所决定的：在int类型的十进制字面常量中，最大的是2147483648。而从0到2147483647的所有十进制字面常量都可以在任何能够使用int类型字面常量的地方出现，但是字面常量2147483648只能作为一元负操作符的操作数来使用[JLS 3.10.1]。

一旦你知道了这个规则，这个谜题就很容易了。字符-2147483648构成了一个合法的Java表达式，它由一元负操作符加上一个int类型字面常量2147483648组成。通过添加一对括号来注解（很不重要的）赋值顺序，即写成 $-(2147483648)$ ，就会破坏这条规则。信不信由你，下面这个程序肯定会出现一个编译期错误，如果去掉了括号，那么错误也就没有了：

```
public class PoisonParen {  
    int i = -(2147483648);  
}
```

类似地，上述情况也适用于long类型字面常量。下面这个程序也会产生一个编译期错误，并且如果你去掉括号错误也会消失：

```
public class PoisonParen {  
    long j = -(9223372036854775808L);  
}
```

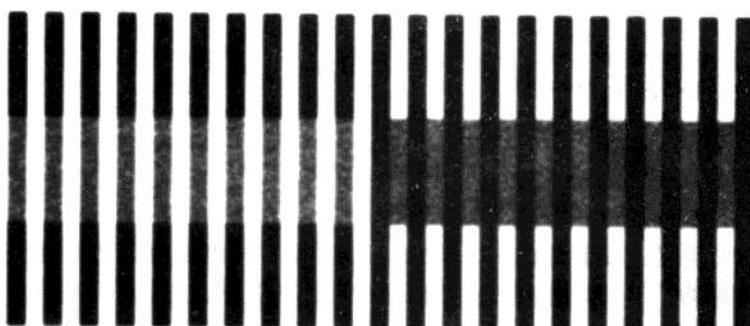
这个谜题没有什么可以当作教训的东西。它只是一种冷僻案例，既纯粹又简单。但是你必须承认，它很有趣。

谜题87：紧张的关系

在数学中，等号（=）定义了一种真实的数之间的等价关系（equivalence relation）。这种等价关系将一个集合分成许多等价类（equivalence class），每个等价类由所有相互相等的值组成。其他的等价关系包括所有三角形集合上的“全等”关系和所有书的集合上的“有相同页数”的关系等。事实上，关系~是一种等价关系，当且仅当它是自反的、传递的和对称的。这些性质定义如下：

- (1) 自反性：对于所有 x , $x\sim x$ 。也就是说，每个值与其自身存在关系~。
- (2) 传递性：如果 $x\sim y$ 并且 $y\sim z$ ，那么 $x\sim z$ 。也就是说，如果第一个值与第二个值存在关系~，并且第二个值与第三个值存在关系~，那么第一个值与第三个值也存在关系~。
- (3) 对称性：如果 $x\sim y$ ，那么 $y\sim x$ 。也就是说，如果第一个值和第二个值存在关系~，那么第二个值与第一个值也存在关系~。

但是这不是一本关于集合论的书；这是一本关于Java的书。在Java中，操作符`==`是否在原生类型数值上定义了一个等价关系呢？如果不是，那它违反了以上三个性质中的哪一个呢？请提供一段程序来演示它是否违反了任意性质。



解惑87：紧张的关系

如果你看了谜题29，便可以知道操作符`==`不是自反的，因为表达式`(Double.NaN == Double.NaN)`值为`false`，表达式`(Float.NaN == Float.NaN)`也是如此。但是操作符`==`是否还违反了对称性和传递性呢？事实上它并不违反对称性：对于所有`x`和`y`的值，`(x==y)`意味着`(y==x)`。传递性则完全是另一回事。

谜题35为操作符`==`作用于原生类型的数值时不符合传递性的原因提供了线索。当比较两个原生类型数值时，操作符`==`首先进行二进制数据类型提升[JLS 5.6.2]。这会导致这两个数值中有一个会进行拓宽原生类型转换。大部分拓宽原生类型转换是不会有问题的，但有三个值得注意的异常情况：将`int`或`long`值转换成`float`值，或将`long`值转换成`double`值时，均会导致精度丢失。这种精度丢失可以证明`==`操作符的不可传递性。

实现这种不可传递性的窍门就是利用上述三种数值比较中的两种去丢失精度，然后就可以得到与事实相反的结果。可以这样构造例子：选择两个较大的但不相同的`long`类型数值赋给`x`和`z`，将一个与前面两个`long`类型数值相近的`double`类型数值赋给`y`。下面的程序就是其代码，它打印的结果是`true true false`，这显然证明了操作符`==`作用于原生类型时具有不可传递性。

```
public class Transitive {
    public static void main(String[] args) throws Exception {
        long x = Long.MAX_VALUE;
        double y = (double) Long.MAX_VALUE;
        long z = Long.MAX_VALUE - 1;

        System.out.print((x == y) + ""); // Imprecise!
        System.out.print((y == z) + ""); // Imprecise!
        System.out.println(x == z); // Precise
    }
}
```

本谜题的教训是：要警惕`float`和`double`类型的拓宽原生类型转换所造成的损失。它们是悄无声息的，但却是致命的。它们会违反你的直觉，并且可以造成非常微妙的错误（见谜题34）。更一般地说，要警惕那些混合类型的运算（谜题5、8、24和31）。本谜题给语言设计者的教训和谜题34一样：悄无声息的精度损失把程序员们搞糊涂了。

谜题88：原生类型的处理

下面的程序由一个单一的类构成，该类表示一对类型相似的对象。它大量使用了5.0版的特性，包括泛型、自动包装、变长参数和for-each循环。关于这些特性的介绍，请查看[http://java.sun.com/j2se/5.0/docs/guide/language\[Java 5.0\]](http://java.sun.com/j2se/5.0/docs/guide/language[Java 5.0])。这个程序的main方法只是执行这个类。那么它会打印什么呢？

```
import java.util.*;

public class Pair<T> {
    private final T first;
    private final T second;

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T first() {
        return first;
    }

    public T second() {
        return second;
    }

    public List<String> stringList() {
        return Arrays.asList(String.valueOf(first),
                           String.valueOf(second));
    }

    public static void main(String[] args) {
        Pair p = new Pair<Object>(23, "skidoo");
        System.out.println(p.first() + " " + p.second());
        for (String s : p.stringList())
            System.out.print(s + " ");
    }
}
```

解惑88：原生类型的处理

这段程序看上去似乎相当简单。它创建了一个对象对，其中第一个元素是一个表示23的Integer对象，第二个元素是一个字符串 "skidoo"，然后这段程序将打印这个对象对的第一个和第二个元素，并用一个空格隔开。最后它循环迭代这些元素的string表示，并且再次打印它们，所以这段程序应该打印23 skidoo两次。然而可惜的是，它根本不能通过编译。更糟的是，编译器的错误消息更是令人困惑：

```
Pair.java:26: incompatible types;
found: Object, required: String
    for (String s : p.stringList())
        ^

```

如果Pair.stringList是声明返回List<Object>的话，那么这个错误消息还是可以理解的，但是它返回的是List<String>。究竟是怎么回事呢？

这个十分奇怪的现象是因为程序使用了原生类型而引起的。一个原生类型就是一个没有任何类型参数的泛型类或泛型接口的名字。例如，List<E> 是一个泛型接口，List<String> 是一个参数化的类型，而List就是一个原生类型。在我们的程序中，惟一用到原生类型的地方就是在main方法中对局部变量p的声明：

```
Pair p = new Pair<Object>(23, "skidoo");
```

一个原生类型很像其对应的参数化类型，但是它的所有实例成员都要被替换掉，而替换物就是这些实例成员被擦掉对应部分之后剩下的东西。具体地说，在一个实例方法声明中出现的每个参数化的类型都要被其对应的原生部分所取代[JLS 4.8]。我们程序中的变量p是属于原生类型Pair的，所以它的所有实例方法都要执行这种擦除。这也包括声明返回List<String>的方法stringList。编译器会将这个方法解释为返回原生类型List。

当List<String>实现了参数化类型Iterable<String>时，List也实现了原生类型Iterable。Iterable<String>有一个iterator方法返回参数化类型Iterator<String>，相应地，Iterable也有一个iterator方法返回原生类型Iterator。当Iterator<String>的next方法返回String时，Iterator的next方法返回Object。因此，循环迭代p.stringList()需要一个Object类型的循环变量，这就解释了编译器的那个奇怪的错误消息的由来。这种现象令人想不通的原因在于参数化类型List<String>虽然是方法stringList的返回类型，但它与Pair的类型参数没有

关系，事实上最后它被擦除了。

你可以尝试通过将循环变量类型从String改成Object这一做法来解决这个问题：

```
// Don't do this; it doesn't really fix the problem!
for (Object s : p.stringList())
    System.out.print(s + " ");
```

这样确实令程序输出了满意的结果，但是它并没有真正解决这个问题。你会失去泛型带来的所有优点，并且如果该循环在s上调用了任何String方法，那么程序甚至不能通过编译。解决这个问题的正确方法是为局部变量p提供一个合适的参数化的声明：

```
Pair<Object> p = new Pair<Object>(23, "skidoo");
```

以下是要点强调：原生类型List和参数化类型List<Object>是不一样的。如果使用了原生类型，编译器不会知道在list允许接受的元素类型上是否有任何限制，它会允许你添加任何类型的元素到list中。这不是类型安全的：如果你添加了一个错误类型的对象，那么在程序接下来的执行中的某个时刻，你会得到一个ClassCastException异常。如果使用了参数化类型List<Object>，编译器便会明白这个list可以包含任何类型的元素，所以你添加任何对象都是安全的。

还有第三种与以上两种类型密切相关的类型：List<?>是一种特殊的参数化类型，被称为通配符类型（wildcard type）。像原生类型List一样，编译器也不会知道它接受哪种类型的元素，但是因为List<?>是一个参数化类型，从语言上来说需要更强的类型检查。为了避免出现ClassCastException异常，编译器不允许你添加除null以外的任何元素到一个类型为List<?>的list中。

原生类型是为兼容5.0版以前的已有代码而设计的，因为它们不能使用泛型。5.0版中的许多核心库类，如collection，已经利用泛型做了改变，但是使用这些类的已有程序的行为仍然与在以前的版本上运行一样。这些原生类型及其成员的行为被设计成可以映射到5.0之前的Java语言上，从而保持了兼容性。

这个Pair程序的真正问题在于编程者没有决定究竟使用哪种Java版本。尽管程序中大部分使用了泛型，而变量p却被声明成原生类型。为了避免被编译错误所迷惑，请避免在打算用5.0或更新的版本来运行的代码中编写原生类型。如果一个已有的库方法返回了一个原生类型，那么请将它的结果存储在一个恰当的参数化类型的变量中。然而，最好的办法还是尽量将该库升级到使用泛型的版本上。虽然Java提供了原生类型和参数化类型间的良好的互用性，但是原生类型的局限性会妨碍泛型的使用。

实际上，这种问题在用`getAnnotation`方法在运行期读取Class的注解的情况下也会发生，该方法是在5.0版中新添到Class类中的。每次调用`getAnnotation`方法时都会涉及两个Class对象：一个是在其上调用该方法的对象，另一个是作为传递参数指出需要哪个类的注解的对象。在一个典型的调用中，前者是通过反射获得的，而后者是一个类名称字面常量，如下例所示：

```
Author a = Class.forName(name).getAnnotation(Author.class);
```

你不必把`getAnnotation`的返回值转型为`Author`。以下两种机制保证了这种做法可以正常工作：(1) `getAnnotation`方法是泛型的。它是通过它的参数类型来确定返回类型的。具体地说，它接受一个`Class<T>`类型的参数，返回一个`T`类型的值。(2) 类名称字面常量提供了泛型信息。例如，`Author.class`的类型是`Class<Author>`。类名称字面常量可以传递运行时和编译期的类型信息。以这种方式使用的类名称字面常量被称作类型符号（type token）[Bracha04]。

与类名称字面常量不同的是，通过反射获得的`Class`对象不能提供完整的泛型类型信息：`Class.forName`的返回类型是通配类型`Class<?>`。在调用`getAnnotation`方法的表达式中，使用的是通配类型而不是原生类型`Class`，这一点很重要。如果你采用了原生类型，返回的注解具有的就是编译期的`Annotation`类型而不是通过类名称字面常量指示的类型了。下面的程序片段错误地使用了原生类型，和本谜题中最初的程序一样不能通过编译，其原因也一样：

```
Class c = Class.forName(name);           // Raw type!
Author a = c.getAnnotation(Author.class); // Type mismatch
```

总之，原生类型的成员被擦掉，是为了模拟泛型被添加到语言中之前的那些类型的行为。如果你将原生类型和参数化类型混合使用，那么便无法获得使用泛型的所有好处，而且有可能产生让你困惑的编译错误。另外，原生类型和以`Object`为类型参数的参数化类型也不相同。最后，如果你想重构现有的代码以利用泛型的优点，那么最好的方法是一次只重构一个API，并且保证新的代码中绝不使用原生类型。

谜题89：泛型迷药

和前一个谜题一样，本谜题也大量使用了泛型。我们从前面的错误中吸取教训，这次不再使用原生类型了。这个程序实现了一个简单的链表数据结构。`main`程序构建了一个包含两个元素的list，然后输出它的内容。那么，这个程序会打印出什么呢？

```
public class LinkedList<E> {
    private Node<E> head = null;

    private class Node<E> {
        E value;
        Node<E> next;

        //Node constructor links the node as a new head
        Node(E value) {
            this.value = value;
            this.next = head;
            head = this;
        }
    }

    public void add(E e) {
        new Node<E>(e);
        //Link node as new head
    }

    public void dump() {
        for (Node<E> n = head; n != null; n = n.next)
            System.out.print(n.value + " ");
    }

    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<String>();
        list.add("world");
        list.add("Hello");
        list.dump();
    }
}
```

解惑89：泛型迷药

又是一个看上去相当简单的程序。新元素被添加到链表的表头，而dump方法也是从表头开始打印list。因此，元素的打印顺序正好和它们被添加到链表中的顺序相反。在本例中，程序先添加了"world"然后添加了"Hello"，所以总体来看它似乎就是一个复杂的Hello World程序。遗憾的是，如果你尝试着编译它，就会发现它不能通过编译。编译器的错误消息是令人完全无法理解的：

```
LinkedList.java:11: incompatible types
found : LinkedList<E>.Node<E>
required: LinkedList<E>.Node<E>
        this.next = head;
               ^
               ^

LinkedList.java:12: incompatible types
found : LinkedList<E>.Node<E>
required: LinkedList<E>.Node<E>
        head = this;
               ^
               ^
```

这看起来就像编译器在抱怨一个类型和它本身不兼容！和之前一样，表象再次欺骗了你。所谓的“found”类型和“required”类型之间根本没有关系。它们看起来一样是因为程序使用了相同的名字却引用了不同的类型。具体来说，这个程序包含两个不同的类型参数声明，都称作E。第一个是LinkedList的类型参数，第二个是内部类LinkedList.Node的类型参数。在内部类中后者遮蔽了前者。我们已经在谜题71、73和79中得到了教训：要避免遮蔽类型参数的名字。这里也同样适用。

除了使用类型参数的简单名之外，没有任何其他的方法可以引用到类型参数，所以错误消息也只能告诉你这两个使用了相同名字的E引用到了不同的类型。如果我们系统地修改Node的类型参数名，比如从E改为T，那么错误消息就会清楚多了。这样做可以使问题更明了，但还不能解决问题。该方法会报出如下的错误消息：

```
LinkedList.java:11: incompatible types
found : LinkedList<E>.Node<E>
required: LinkedList<E>.Node<T>
        this.next = head;
               ^
               ^
```

```
LinkedList.java:12: incompatible types
found : LinkedList<E>.Node<T>
required: LinkedList<E>.Node<E>
    head = this;
    ^

```

编译器试图告诉我们，这个程序太过复杂了。一个泛型类的内部类可以访问到它的**外围类的类型参数**。而编程者的意图很明显，即一个Node的类型参数应该和它外围的LinkedList类的类型参数一样，所以Node完全不需要有自己的类型参数。要订正这个程序，只需要去掉内部类的类型参数即可：

```
// Fixed but could be MUCH better
public class LinkedList<E> {
    private Node head = null;

    private class Node {
        E value;
        Node next;

        //Node constructor links the node as new head
        Node(E value) {
            this.value = value;
            this.next = head;
            head = this;
        }
    }

    public void add(E e) {
        new Node(e);
        //Link node as new head
    }

    public void dump() {
        for (Node n = head; n != null; n = n.next)
            System.out.print(n.value + " ");
    }
}
```

以上是解决问题的最简单的修改方案，但不是最优的。最初的程序所使用的内部类并不是必须的。正如谜题80中提到的，你应该优先使用**静态成员类**而不是**非静态成员类** [EJ Item 18]。LinkedList.Node的一个实例不仅含有value和next域，还有一个隐

藏的域，它包含了对外围的LinkedList实例的引用。虽然外部类的实例在构造阶段会被用来读取和修改head，但是一旦构造完成，它就变成了一个甩不掉的包袱。更糟的是，这样使得构造器中有了修改head的负面影响，从而使程序变得难以读懂。**应该只在类自己的方法中修改该类的实例域。**

因此，一个更好的修改方案是将最初的那个程序中对head的操作移到LinkedList.add方法中，这将会使Node成为一个静态嵌套类而不是真正的内部类。静态嵌套类不能访问它的外围类的类型参数，所以现在Node就必须有自己的类型参数了。修改后的程序既简单清楚又正确无误：

```
class LinkedList<E> {
    private Node<E> head = null;

    private static class Node<T> {
        T value;
        Node<T> next;

        Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
        }
    }

    public void add(E e) {
        head = new Node<E>(e, head);
    }

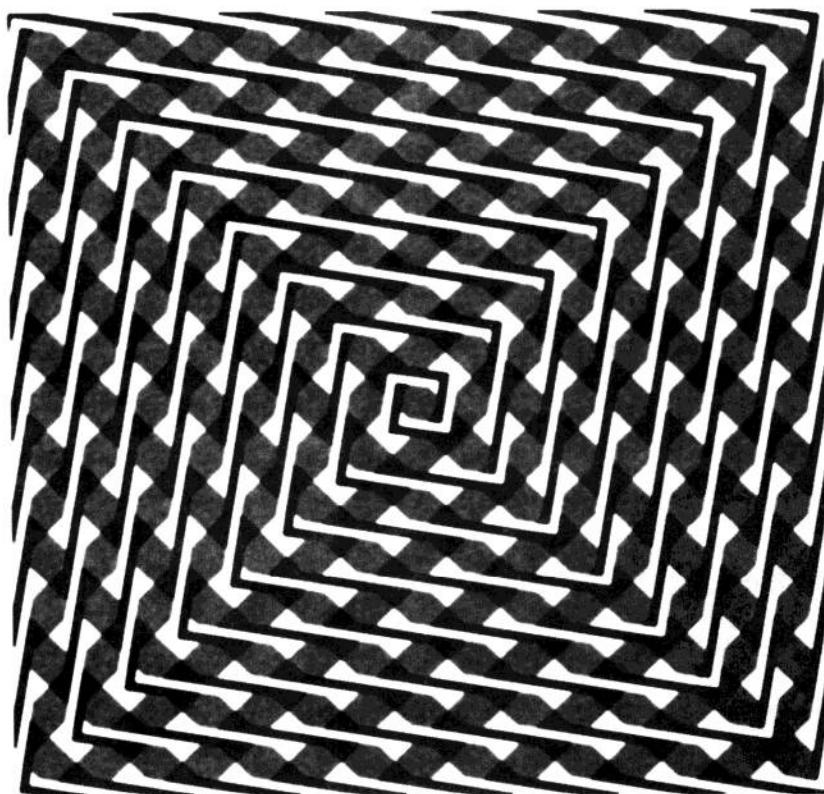
    public void dump() {
        for (Node<E> n = head; n != null; n = n.next)
            System.out.print(n.value + " ");
    }
}
```

总之，泛型类的内部类可以访问其外围类的类型参数，这可能会使得程序混淆难懂。本谜题所阐述的误解对于初学泛型的程序员来说是普遍存在的。在一个泛型类中设置一个内部类并不是必错的，但是很少用到这种情况，而且你应该考虑重构你的代码来避免这种情况。当你在一个泛型类中嵌套另一个泛型类时，最好为它们的类型参数设置不同的名字，即使那个嵌套类是静态的也应如此。对于语言设计者来说，或许应该考虑禁止类型参数的遮蔽机制，同样的，局部变量的遮蔽机制也应该被禁止。这样的规则就可以捕获到本谜题中的错误了。

谜题90：荒谬痛苦的超类

下面的程序实际上不会做任何事情。更糟的是，它连编译也通不过。为什么呢？又怎么来订正它呢？

```
public class Outer {  
    class Inner1 extends Outer {}  
    class Inner2 extends Inner1 {}  
}
```



解惑90：荒谬痛苦的超类

这个程序看上去简单得不可能有错误，但是如果你尝试编译它，就会得到下面这个有用的消息：

```
Outer.java:3: cannot reference this before
    supertype constructor has been called
    class Inner2 extends Inner1 {}
    ^

```

好吧，可能这个消息不那么有用，但是我们还是从此入手。问题在于编译器产生的缺省的Inner2的构造器为它的super调用找不到合适的外围类实例。让我们来看看显式地包含了构造器的该程序：

```
public class Outer {
    public Outer() {}

    class Inner1 extends Outer {
        public Inner1() {
            super(); //invokes Object()constructor
        }
    }

    class Inner2 extends Inner1 {
        public Inner2() {
            super(); //invokes Inner1()constructor
        }
    }
}
```

现在错误消息就会显示更多的信息了：

```
Outer.java:12: cannot reference this before
    supertype constructor has been called
    super(); //invokes Inner1()constructor
    ^

```

因为Inner2的超类本身也是一个内部类，一个晦涩的语言规则登场了。众所周知，要想实例化一个内部类，如类Inner1，需要提供一个外围类的实例给构造器。一般情

况下，它是隐式地传递给构造器的，但是它也可以以expression.super(args)的方式通过超类构造器调用(superclass constructor invocation)显式地传递[JLS 8.8.7]。如果外围类实例是隐式传递的，编译器会自动产生表达式：它使用this来指代最内部的超类是一个成员变量的外围类。这确实有点绕口，但是这就是编译器所做的事情。在本例中，那个超类就是Inner1。因为当前类Inner2间接扩展了Outer类，Inner1便是它的一个继承而来的成员。因此，超类构造器的限定表达式直接就是this。编译器提供外围类实例，将super重写成this.super。解释到这里，编译错误所含的意思可扩展为：

```
Outer.java:12: cannot reference this before
                     supertype constructor has been called
    this.super();
           ^

```

现在问题就清楚了：缺省的Inner2的构造器试图在超类构造器被调用前访问this，这是一个非法的操作[JLS 8.8.7.1]。解决这个问题的铁腕方法是显式地传递合理的外围类实例：

```
public class Outer {
    class Inner1 extends Outer { }

    class Inner2 extends Inner1 {
        public Inner2() {
            Outer.this.super();
        }
    }
}
```

这样可以通过编译，但是它太复杂了。这里有一个更好的解决方案：无论何时你写了一个成员类，都要问问你自己，是否这个成员类真的需要使用外围类实例？如果答案是否定的，那么应该把它设为静态成员类。内部类有时是非常有用的，但是它们很容易增加程序的复杂度，从而使程序难以理解。它们和泛型（谜题89）、反射（谜题80）以及继承（本谜题）都有着复杂的交互方式。在本例中，如果你将Inner1设为静态的便可以解决问题了。如果你将Inner2也设为静态的，你就会真正明白这个程序做了什么：确实是一个相当好的意外收获。

总之，这种一个类既是内部类又是其他类的子类的方式是很不合理的。更一般地讲，扩展一个内部类的方式是很不恰当的；如果必须这样做的话，你也要好好考虑其外围类实例的问题。另外，尽量用静态嵌套类而少用非静态的[EJ Item 18]。大部分成员类可以并且应该被声明为静态的。

谜题91：序列杀手

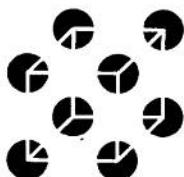
这个程序创建了一个对象并且检查它是否遵从某个类的不变规则。然后该程序序列化这个对象，之后将其反序列化，然后再次检查反序列化得到的副本是否也遵从这个规则。它会遵从这个规则吗？如果不是的话，又是为什么呢？

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

public class SerialKiller {
    public static void main(String[] args) {
        Sub sub = new Sub(666);
        sub.checkInvariant();

        Sub copy = (Sub) deepCopy(sub);
        copy.checkInvariant();
    }

    // Copies its argument via serialization (See Puzzle 83)
    static public Object deepCopy(Object obj) {
        try {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            new ObjectOutputStream(bos).writeObject(obj);
            ByteArrayInputStream bin =
                new ByteArrayInputStream(bos.toByteArray());
            return new ObjectInputStream(bin).readObject();
        } catch(Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```



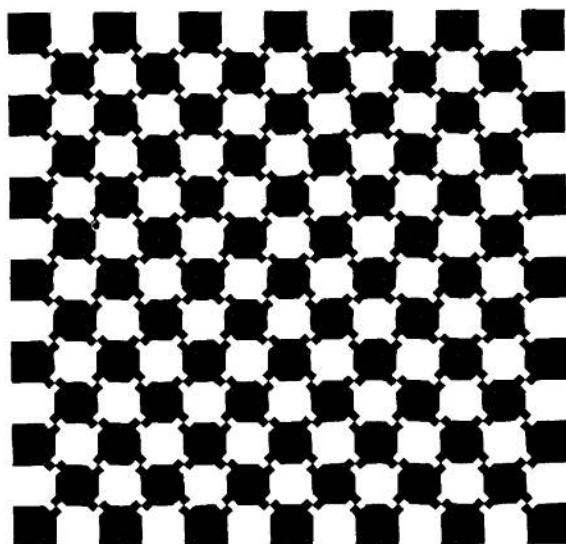
```
class Super implements Serializable {
    final Set<Super> set = new HashSet<Super>();
}

final class Sub extends Super {
    private int id;
    public Sub(int id) {
        this.id = id;
        set.add(this); // Establish invariant
    }

    public void checkInvariant() {
        if (!set.contains(this))
            throw new AssertionError("invariant violated");
    }

    public int hashCode() {
        return id;
    }

    public boolean equals(Object o) {
        return (o instanceof Sub)&& (id == ((Sub)o).id);
    }
}
```



解惑91：序列杀手

程序中除了使用了序列化之外，看起来是很简单的。子类Sub覆写了hashCode和equals方法。这些覆写过的方法符合了相关的一般规约[EJ Item 7.8]。Sub的构造器建立了这个类的不变规则，而在它这么做的时候没有调用可覆写的方法（谜题51）。Super类有一个单独的Set<Super>类型的域，Sub类添加了另外一个int类型的域。Super和Sub都不需要定制的序列化形式。那么什么东西会出错呢？

其实有很多。对于5.0版本，运行该程序会得到如下的“堆轨迹”（stack trace）：

```
Exception in thread "main" AssertionError  
at Sub.checkInvariant(SerialKiller.java:41)  
at SerialKiller.main(SerialKiller.java:10)
```

序列化和反序列化一个Sub实例会产生一个被破坏的副本。为什么呢？阅读程序并不会帮助你找出原因，因为真正引起问题的代码在其他地方。错误是由HashSet的readObject方法引起的。在某些情况下，这个方法会间接地调用某个未初始化对象的被覆写的方法。为了组装正在被反序列化的散列集合，HashSet.readObject调用了HashMap.put方法，而它会去调用每个键的hashCode方法。由于整个对象图正在被反序列化，并没有什么可以保证每个键在它的hashCode方法被调用的时候已经被完全初始化了。实际上，这很少会成为一个问题，但是有时候它会造成绝对的混乱。这个缺陷会在正在被反序列化的对象图的某些循环中出现。

为了更具体一些，让我们看看程序中在反序列化Sub实例的时候发生了什么。首先，序列化系统会反序列化Sub实例中Super的域。惟一的这样的域就是set，它包含了一个对HashSet的引用。在内部，每个HashSet实例包含一个对HashMap的引用，HashMap的键是该散列集合的元素。HashSet类有一个readObject方法，它创建一个空的HashMap，并且使用HashMap的put方法，针对集合中的每个元素在HashMap中插入一个键—值对。put方法会调用键的hashCode方法以确定它所在的单元格。在我们的程序中，散列映射表中惟一的键就是Sub的实例，而它的set域正在被反序列化。这个实例的子类域，即id，尚未被初始化，所以它的值为0，即所有int域的缺省初始值。不幸的是，Sub的hashCode方法将返回这个值，而不是最后保存在这个域中的值666。因为hashCode返回了错误的值，相应的键—值对条目将会放入错误的单元格中。当id域被初始化为666时，一切都太迟了。当Sub实例在HashMap中的时候，改变这个域的值就会破坏这个域，进而破坏HashSet，破坏Sub实例。程序检测到了这个情况，就报告了相应的错误。

这个程序说明，包含了HashMap的readObject方法的序列化系统总体上违背了不能从类的构造器或伪构造器中调用可覆写方法的规则[EJ Item 15]。Super类的（缺省的）readObject方法调用了HashSet的（显式的）readObject方法，该方法进而调用了它内部的HashMap的put方法，put方法又调用了Sub实例的hashCode方法，而该实例正处在创建的过程中。现在我们遇到大麻烦了：Super类中，从Object类继承而来的hashCode方法在Sub中被覆写了，但是这个被覆写的方法在Sub的域被初始化之前就被调用了，而该方法需要依赖于Sub的域。

这个问题和谜题51中的那个本质上几乎是完全相同的。惟一真正的不同是在这个谜题中，readObject伪构造器错误地替代了构造器。HashMap和Hashtable的readObject方法受到的影响是类似的。

对于平台的实现者来说，也许可以通过牺牲一点性能来订正HashSet、HashMap和HashTable中的这个问题。当针对HashSet时，订正的策略可以是重写readObject方法使其在反序列化期间，将集合的元素保存到一个数组中，而不是将它们放入散列集合中。这样，当反序列化了的散列集合的公共方法首次被调用的时候，数组中的元素将在方法执行之前被插入到集合中。

这种方法的代价是它需要在与散列集合的每个公共方法相对应的条目上检查是否要组装散列集合。由于HashSet、HashMap以及HashTable都是性能临界的，所以这个方法看起来是不可取的。更不幸的是，所有的用户都要付出这种代价，甚至当他们不对这些集合进行序列化时也是如此。这就违背了这样一个原则：你绝不应该为你没使用功能而付出代价。

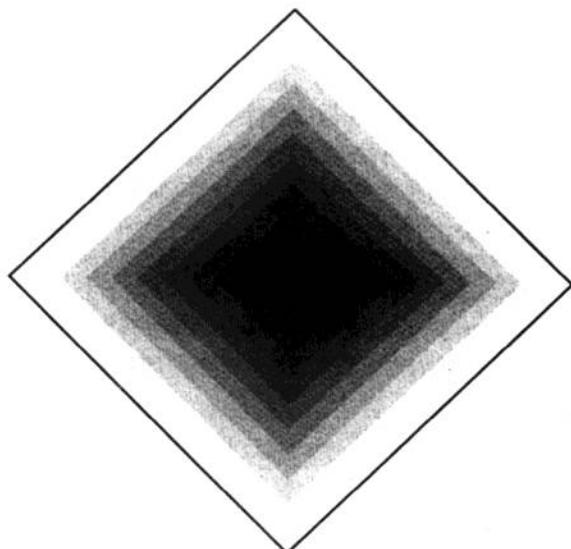
另外一个可能的方法是让HashSet.readObject方法调用ObjectInputStream.registerValidation方法，用以将散列集合的组装延迟到validateObject方法回调时再进行。这个方法看起来更吸引人，因为它仅仅增加了反序列化的开销，但是它会破坏任何在“包含流”（containing stream）的反序列化过程中试图使用HashSet实例的代码。

上述的两个方法是否可行还有待研究。但是现在，我们必须接受这些类的这种行为。幸运的是，有一个工作区（workaround）：如果一个HashSet、Hashtable或HashMap被序列化，那么请确认它们的内容没有直接或间接地引用它们自身。这里的“内容”（content），指的是元素、键和值。

这里也有一个教训送给那些使用可序列化类型的开发者们：在readObject或readResolve方法中，请避免直接或间接地在正在进行反序列化的对象上调用任何方法。如果你必须在某个类C的readObject或readResolve方法中违背这条建议，请确定没有

C的实例会出现在正在被反序列化的对象图的循环内。不幸的是，这不是一个本地的属性，一般说来，你需要考虑整个系统来验证这一点。

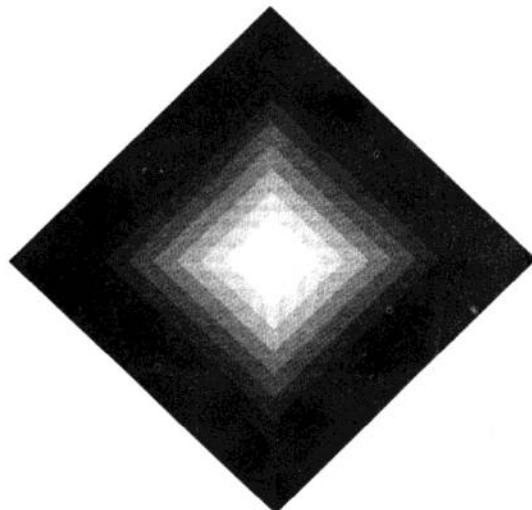
总之，Java的序列化系统是很脆弱的。为了正确而且高效地序列化大量的类，你必须编写readObject或readResolve方法[EJ Item 55~57]。这个谜题说明了，为了避免破坏反序列化的实例，你必须小心翼翼地编写这些方法。HashSet、HashMap和Hashtable的readObject方法很容易产生这种错误。对于平台设计者来说，如果你决定提供序列化系统，请不要提供如此脆弱的东西。健壮的序列化系统是很难设计的。



谜题92：双绞线

下面这个程序使用一个匿名类执行了一个并不自然的动作。它会打印出什么呢？

```
public class Twisted {  
    private final String name;  
    Twisted(String name) {  
        this.name = name;  
    }  
    private String name() {  
        return name;  
    }  
    private void reproduce() {  
        new Twisted("reproduce") {  
            void printName() {  
                System.out.println(name());  
            }  
        }.printName();  
    }  
    public static void main(String[] args) {  
        new Twisted("main").reproduce();  
    }  
}
```



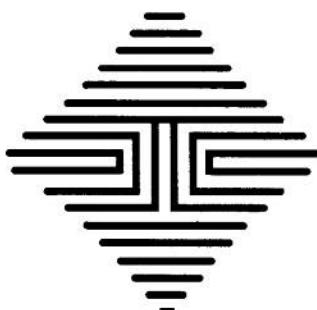
解惑92：双绞线

根据一个肤浅的分析会判断该程序不能通过编译。reproduce方法中的匿名类试图调用Twisted类中的私有方法name。一个类不能调用另一个类的私有方法，能够吗？如果你试图编译这个程序，你会发现它可以成功地通过编译。在顶层的类型（top-level type）中，即本例中的Twisted类，所有的本地的、内部的、嵌套的和匿名的类都可以毫无限制地访问彼此的成员[JLS 6.6.1]。这是一个欢乐的大家庭。

在了解了这些之后，你可能会希望程序打印出reproduce，因为它在new Twisted ("reproduce")实例上调用了printName方法，这个实例将字符串 "reproduce" 传给其超类的构造器使其存储到它的name域中。printName方法调用name方法，name方法返回了name域的内容。但是如果你运行这个程序，你会发现它打印的是main。现在的问题是它为什么会做出这样的事情呢？

这种行为背后的原因是**私有成员不会被继承**[JLS 8.2]。在这个程序中，name方法并没有被继承到reproduce方法中的匿名类中。所以，匿名类中对于printName方法的调用必须关联到外围("main")实例而不是当前("reproduce")实例。这就是含有正确名称的方法的最小外围范围（谜题 71 和 79）。

这个程序违反了谜题90中的建议：在"reproduce"中的匿名类是Twisted类的内部类，也扩展了它。单独这一点就足以使程序难以阅读。再加上调用超类的私有方法的复杂度，这个程序就成了纯粹的冗长废话。这个谜题可以用来强调谜题6中的教训：**如果你不能通过阅读代码来分辨程序会做什么，那么它很可能不会做你想让它做的事。要尽量争取程序的清晰。**



谜题93：类的战争

下面这个谜题测试了你关于二进制兼容性（binary compatibility）的知识：当你改变了某个类所依赖的另外一个类时，第一个类的行为会发生什么改变呢？更特殊的是，假设你编译的是如下的两个类。第一个代表一个客户端，第二个代表一个库类：

```
public class PrintWords {
    public static void main(String[] args) {
        System.out.println(Words.FIRST + " " +
                            Words.SECOND + " " +
                            Words.THIRD);
    }
}

public class Words {
    private Words() {} // Uninstantiable

    public static final String FIRST = "the";
    public static final String SECOND = null;
    public static final String THIRD = "set";
}
```

现在假设你像下面这样改变了那个库类并且重新编译了这个类，但并不重新编译客户端的程序：

```
public class Words {
    private Words() {} // Uninstantiable

    public static final String FIRST = "physics";
    public static final String SECOND = "chemistry";
    public static final String THIRD = "biology";
}
```

此时，客户端的程序会打印出什么呢？



解惑93：类的战争

简单地看看程序，你会觉得它应该打印physics chemistry biology；毕竟Java是在运行时对类进行装载的，所以它总是会访问最新版本的类。但是更深入一点的分析会得出不同的结论。对于常量域的引用会在编译期被转化为它们所表示的常量的值[JLS 13.1]。这样的域从技术上讲，被称作常量变量（constant variable），这可能在修辞上显得有点矛盾。一个常量变量的定义是，一个在编译期被常量表达式初始化的final的原生类型或String类型的变量[JLS 4.12.4]。在知道了这些知识之后，我们有理由认为客户端程序会将初始值Words.FIRST, Words.SECOND及Words.THIRD编译进class文件，然后无论Words类是否被改变，客户端都会打印the null set。

这种分析可能是有道理的，但是却是不对的。如果你运行了程序，你会发现它打印的是the chemistry set。这看起来确实太奇怪了。它为什么会做出这种事情呢？答案可以在编译期常量表达式(compile-time constant expression)[JLS 15.28]的精确定义中找到。它的定义太长了，就不在这里写出来了，但是理解这个程序的行为的关键是null不是一个编译期常量表达式。

由于常量域将会编译进客户端，API的设计者在设计一个常量域之前应该深思熟虑。如果一个域表示的是一个真实的常量，例如π或者一周之内的天数，那么将这个域设为常量域没有任何坏处。但是如果你想让客户端程序感知并适应这个域的变化，那么就不能让这个域成为一个常量。有一个简单的方法可以做到这一点：如果你使用了一个非常量的表达式去初始化一个域，甚至是一个final域，那么这个域就不是一个常量。你可以通过将一个常量表达式传给一个方法使得它变成一个非常量，该方法将直接返回其输入参数。

如果我们使用这种方法来修改Words类，在Words类被重新修改和编译之后，PrintWords类将打印出physics chemistry biology：

```
public class Words {
    private Words() {} // Uninstantiable
    public static final String FIRST = ident("the");
    public static final String SECOND = ident(null);
    public static final String THIRD = ident("set");

    private static String ident(String s) {
        return s;
    }
}
```

在5.0版本中引入的枚举常量，虽然有这样一个名字，但是它们并不是常量变量。你可以在枚举类型中加入枚举常量，对它们重新排序，甚至可以移除没有用的枚举常量，而且并不需要重新编译客户端。

总之，常量变量将会被编译进那些引用它们的类中。一个常量变量就是任何被常量表达式初始化的原生类型或字符串变量。令人惊讶的是，null不是一个常量表达式。

对于语言设计者来说，在一个动态链接的语言中，将常量表达式编译进客户端可能并不是一个好主意。这让很多程序员大吃一惊，并且很容易产生一些难以查出的缺陷：当缺陷被侦测出来的时候，那些定义常量的源代码可能已经不存在了。另外一方面，将常量表达式编译进客户端使得我们可以使用if语句来模拟条件编译[JLS 14.21]。为了正当事目的可以不择手段的做法是需要每个人自己来判断的。

谜题94：迷失在混乱中

下面的shuffle方法声称它将均等地打乱它的输入数组的次序。换句话说，假设其使用的伪随机数发生器是均等的，它将会以均等的概率产生各种排列的数组。它真的兑现了它的诺言吗？如果没有，你将如何订正它呢？

```
import java.util.Random;

public class Shuffle {
    private static Random rnd = new Random();

    public static void shuffle(Object[] a) {
        for(int i = 0; i < a.length; i++)
            swap(a, i, rnd.nextInt(a.length));
    }

    private static void swap(Object[] a, int i, int j) {
        Object tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
}
```

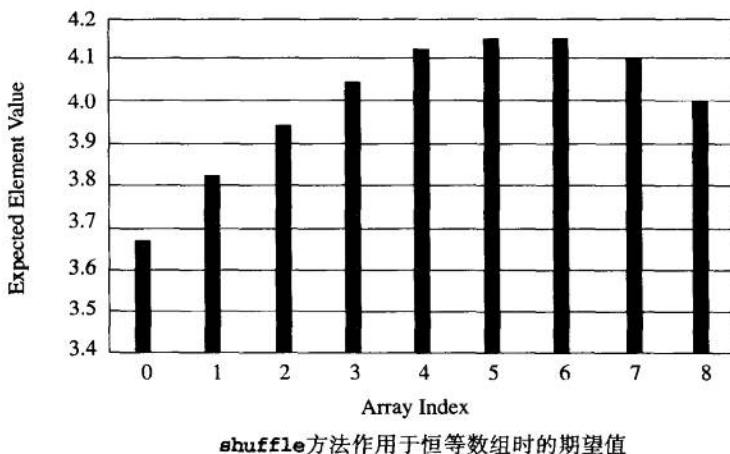
解惑94：迷失在混乱中

看看这个shuffle方法，它并没有什么明显的错误。它遍历了整个数组，将随机抽取的元素互换位置。这会均等地将数组打乱，对吗？不对。“它没有明显的错误”和“它明显没有错误”这两种说法是很不同的。在这里，有很严重的错误，但是它并不明显，除非你专门研究算法。

如果你使用一个长度为 n 的数组作为参数去调用shuffle方法，这个循环体会执行 n 次。在每次执行中，这个方法会选取从0到 $n-1$ 这 n 个整数中的一个。所以，该方法就有 n^n 种不同的执行动作。我们假设随机数发生器是均等的，那么每一种执行动作出现的概率是相等的。每一种执行动作都产生数组的一种排列。但是，这里就有一个小问题：对于一个长度为 n 的数组来说，只有 $n!$ 种不同的排列。（在 n 之后的感叹号表示阶乘操作， n 的阶乘定义为 $n \times (n-1) \times (n-2) \times \dots \times 1$ 。）问题在于，对于任何大于2的 n ， n^n 都无法被 $n!$ 整除，因为 $n!$ 包含了从2到 n 的所有质因子，而 n^n 只包含了 n 所包含的质因子。这就毫无疑问地证明了shuffle方法将会更多地产生某些排列。

为了使这个问题更具体一些，让我们来考虑一个包含了字符串 "a", "b", "c" 的长度为3的数组。此时shuffle方法就有 $3^3=27$ 种执行动作。这些动作出现几率相同，并且都会产生某个排列。数组有 $3!=6$ 种不同的排列：{"a", "b", "c"}, {"a", "c", "b"}, {"b", "a", "c"}, {"b", "c", "a"}, {"c", "a", "b"} 和 {"c", "b", "a"}。由于27不能被6整除，比起其他的排列，某些排列肯定会由更多的执行动作产生，所以shuffle方法并不是均等的。

这里的一个问题就是，上述的证明只是证明了shuffle方法确实存在偏差，而并没有提供任何这种偏差的感性材料。有时候深入了解的最好办法就是动手实验。我们让该方法操作“恒等数组”(identity array，即满足 $a[i]=i$ 的数组a)，然后测试程序将计算每个位置上的元素的期望值。不严格地说，这个期望值，就是在重复运行shuffle方法的时候，你在数组的某个位置上看到的所有数值的平均值。如果shuffle方法是均等的，那么每个位置的元素的期望值应该是相等的($(n-1)/2$)。下图显示了在一个长度为9的数组中各个元素的期望值。请注意这张图特殊的形状：开始的时候比较低，然后增长超过了平均值(4)，然后在最后一个元素下降到平均值。



为什么这张图会有这种形状呢？我们不知道具体的细节，但是我们会有一些直觉上的认识。让我们把注意力集中到数组的第一个元素上。当循环体第一次执行之后，它会有正确的期望值 $(n-1)/2$ 。然而在第二次执行中，有 $1/n$ 的可能性，随机数发生器会返回0且数组第一个元素的值会被设为1或0。也就是说，第二次执行系统地减少了第一个元素的期望值。在第三次执行中，也会有 $1/n$ 的可能性，第一个元素的值会被设为2、1或者0，然后就这么继续下去。在循环的前 $n/2$ 次执行中，第一个元素的期望值是减少的。在后 $n/2$ 次执行中，它的期望值是增加的，但是再也达不到它的平均值了。请注意，数组的最后一个元素肯定会有正确的期望值，因为在方法执行的最后一步，就是在数组的所有元素中为其选择一个值。

好了，我们的shuffle方法是坏掉了。我们怎么修复它呢？使用类库中提供的**shuffle方法**：

```
import java.util.*;
public class Shuffle {
    public static void shuffle(Object[] a) {
        Collections.shuffle(Arrays.asList(a));
    }
}
```

如果库中有可以满足你需要的方法，请务必使用它[EJ Item 30]。一般来说，库提供了高效的解决方案，并且可以让你付出最小的努力。

另外，在你忍受了所有这些数学的东西之后，如果不告诉你如何修复这个坏掉的shuffle方法是不公平的。修复方法是非常简单的。在循环体中，将当前的元素和某个在当前元素与数组末尾元素之间的所有元素中随机选择出来的元素进行互换。不要去碰那些已经进行过值互换的元素。这本质上也就是库中的方法所使用的算法：

```
public static void shuffle(Object[] a) {
    for (int i = 0; i < a.length; i++)
        swap(a, i, i + rnd.nextInt(a.length - i));
}
```

使用归纳法很容易证明这个方法是均等的。对于最基础的情况，我们观察长度为0的数组，它显然是均等的。根据归纳法的步骤，如果你将这个方法用在一个长度 $n > 0$ 的数组上，它会为这个数组的0位置上的元素随机选择一个值。然后，它会遍历数组剩下的元素：在每个位置上，它会在“子数组”中随机选择一个元素，这个子数组从当前位置开始到原数组的末尾截止。对于从位置1到原数组末尾的这个长度为 $n-1$ 的子数组来说，如果将该方法作用在这个子数组上，它实际上也是在做上述的事。这就完成了证明。它同时也提供了shuffle方法的递归形式，它的细节就留给读者作为练习了。

你可能会认为到此为止就是故事的全部内容了，但却还有一部分内容。你设想过这个经过修复的shuffle方法会等概率的产生一个表示52张牌的52个元素的数组的所有排列吗？毕竟我们只是证明了它是均等的。在这里你可能不会很惊讶地发现答案很显然是“不”。这里的问题是，在谜题的开始，我们做出了“使用的伪随机数发生器是均等的”这一假设。但是它不是。

这个随机数发生器，`java.util.Random`，使用的是一个64位的种子，而它产生的随机数完全是由这个种子决定的。52张牌有 $52!$ 种排列，而种子却只有 2^{64} 个。它能够覆盖的排列占所有排列的多少呢？你相信是百分之 2.3×10^{-47} 吗？这只是委婉地表示了“实际上就没怎么覆盖”。如果你使用`java.security.SecureRandom`代替`java.util.Random`，你会得到一个160位的种子，但是它给你带来的东西少得惊人：对于元素个数大于40的数组，这个shuffle方法仍然不能返回它的某些排列（因为 $40! > 2^{160}$ ）。对于一个具有52个元素的数组，你只能获得所有可能的排列的百分之 1.8×10^{-18} 。

这难道意味着你在洗牌的时候不能相信这些伪随机数发生器吗？这要看情况。它们确实只能产生所有可能排列的微不足道的一部分，但是它们没有我们前面所看到的那种系统性的偏差。公平地讲，这些发生器在非正式的场合中已经足够好用了。如果你需要一个尖端的随机数发生器，那就需要到别的地方去寻找了。

总之，像很多算法一样，打乱一个数组是需要慎重对待的。这么做很容易犯错并且

很难发现错误。在其他条件相似的情况下，你应该优先使用类库而不是手写的代码。如果你想学习更多的关于本谜题的论题的内容，请参见[Knuth98 3.4.2]。

谜题95：来份甜点

本章的大多数谜题都是颇具挑战性的。但是这个不是。下面这个程序会打印出什么呢？如果你相信的话，前两个程序被报告为系统的缺陷[Bug 4157460 4763901]：

```
public class ApplePie {
    public static void main(String[] args) {
        int count = 0;
        for(int i = 0; i < 100; i++) {
            count++;
        }
        System.out.println(count);
    }
}

import java.util.*;
public class BananaBread {
    public static void main(String[] args) {
        Integer[] array = { 3, 1, 4, 1, 5, 9 };
        Arrays.sort(array, new Comparator<Integer>() {
            public int compare(Integer i1, Integer i2) {
                return i1 < i2 ? -1 : (i2 > i1 ? 1 : 0);
            }
        });
        System.out.println(Arrays.toString(array));
    }
}

public class ChocolateCake {
    public static void main(String[] args) {
        System.out.println(true?false:true == true?false:true);
    }
}
```

解惑95：来份甜点

如果你受够这些东西了，那么你不需要知道这些愚蠢谜题的详细解释，所以让我们把它们变得又短又甜：

- (1) 这个程序会打印出1。这是由多余的标号造成的。(分号的弊病?)
- (2) 这个程序在我们所知道的所有平台实现上都会打印出[3, 1, 4, 1, 5, 9]。从技术上说，程序的输出是未被定义的。它的比较器承受着“是头我赢，是尾你输”的综合症。
- (3) 这个程序会打印出false。其书写的布局和操作符的优先级并不匹配。加一些括号可以解决问题。

这个谜题的教训，也是整本书的教训，就是：**不要像我的兄弟那样编写代码。**

陷阱和缺陷的目录

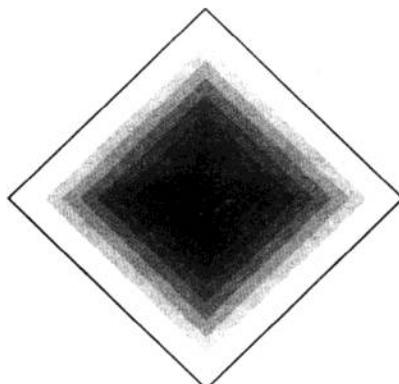
你已经做完这些谜题了吗？如果没有，请回到第1章！直接回到第1章。如果你在做这些谜题之前就阅读本章，那么本书将索然无味。不要说我没有警告过你哟。

本章包含了Java平台中的陷阱和缺陷的简明分类。在目录中的每一项都被分为三部分：

陷阱或缺陷的简短描述

规则：怎样避免陷阱或降低成为受害者的风险。

参考：指向与陷阱相关的附加信息的指针。通常包含一个引用，指向基于该缺陷的谜题。有许多项还具有《Java语言规范》和《Effective Java》参考[JLS, EJ]。



A.1 词汇问题

本节涉及Java程序的词汇结构 (lexical structure)，即构成程序的符号和构成符号的字符。

A.1.1 字母l在许多字体中都与数字1相像

规则：在long类型字面常量中，应该总是用大写的L，千万不要用小写的l。不要用孤零零的一个l作为变量名。

参考：谜题4。

A.1.2 负的十六进制字面常量看起来像是正的

规则：要避免混合类型的计算。在恰当的地方要用long类型的字面常量来代替int类型的字面常量。

参考：谜题5，[JLS 3.10.1]。

A.1.3 八进制字面常量与十进制字面常量相像

规则：要避免八进制字面常量。如果非得使用它们，那么请对所有用到的地方进行注释，以使你的意图清晰。

参考：谜题59，[JLS 3.10.1]。

A.1.4 ASCII字符的Unicode转义字符容易令人迷茫

规则：不要使用ASCII字符的Unicode转义字符。如果可以的话，应该直接使用ASCII字符。在字符串字面常量和字符字面常量中，应优选转义字符列而不是Unicode转义字符。

参考：谜题14、16和17，[JLS 3.2, 3.3]。

A.1.5 反斜杠必须被转义，即使是在注释中

规则：如果你在编写一个可以生成Java源代码的系统，在生成的字符字面常量、字符串字面常量和注释中都要对反斜杠进行转义。Windows文件名最易引发问题。

参考：谜题15和16，[JLS 3.3]。

A.1.6 块注释不要嵌套

规则：使用单行注释来注释掉代码。

参考：谜题19，[JLS 3.7, 14.21]。

A.2 整数运算

本节涉及整数类型上的运算：byte、char、short、int和long。

A.2.1 %操作符的非零结果具有和左操作数相同的正负号

规则：如果你需要一个非负的余数，而%操作符的结果又是负的，那么你应该在结果上加上一个模数。

参考：谜题1和64，[JLS 15.17.3]。

A.2.2 整数运算的悄悄溢出

规则：要用足够大的类型来保存结果，包括中间结果。

参考：谜题3、26、33和65，[JLS 4.2.2]。

A.2.3 int数值之差的正负号不能可靠地指示其大小顺序

规则：不要用基于减法的比较器，除非你可以肯定数值的差永远不会大于Integer.

MAX_VALUE。请注意，这是缺陷2.2的特例。

参考：谜题65，[EJ Item 11]。

A.2.4 复合赋值操作符可能造成悄悄的窄化转型

规则：不要在类型为byte、short或char的变量上使用复合操作符。

参考：谜题9和31，[JLS 15.26.2]。

A.2.5 整数类型不对称：Integer.MIN_VALUE是它自己的负值，long.MIN_VALUE也一样

规则：要保守地编程，如果有必要就用long来代替int。

参考：谜题33和64，[JLS 15.15.4]。

A.2.6 移位操作符只用了其右操作数的低位

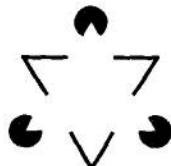
规则：移位的位数应该用常量。如果移位的位数必须用变量，那么一定要检查移位的距离是否在允许的范围内。

参考：谜题27，[JLS 15.19]。

A.2.7 当在整数类型之间转换时，如果源类型是有符号的则执行符号扩展

规则：在操作byte类型的数值时，一定要小心，它是有符号的。要想禁止符号扩展，可以使用位掩码。

参考：谜题6，[JLS 5.1.2]。



A.3 浮点运算

本节涉及浮点类型float和double上的运算。

A.3.1 浮点运算是不精确的

规则：在需要精确结果的地方不要使用浮点，应该使用一个整数类型或BigDecimal。

要避免浮点类型的循环索引。

要避免在浮点变量上使用++和--操作符，因为这些操作对多数的浮点数都不起任何作用。

避免测试浮点值是否相等。

宁愿用double，而不用float。

参考：谜题2、28和34；[JLS 3.4.2.3]，[EJ Item 31]和[IEEE-754]。

A.3.2 NaN不等于任何浮点数值，包括它自身

规则：要避免测试浮点数的相等性。这么做并不总能避免问题，但至少是一个良好的开端。

参考：谜题29，[JLS 15.21.1]和[IEEE-754]。

A.3.3 从int到float、从long到float以及从long到double的转换是有损精度的

规则：要避免整型和浮点型混合的计算。要优选整型算术而不是浮点型算术。

参考：谜题34和87，[JLS 5.1.2]。

A.3.4 BigDecimal(double)构造器返回的是其浮点型参数的精确值

规则：应该总是使用BigDecimal(String)构造器，而永远不要使用BigDecimal(double)。

参考：谜题2。

A.4 表达式计算

本节涉及表达式计算的方方面面，并不专指整数或浮点数运算。

A.4.1 混合类型计算容易令人迷茫

规则：要避免混合类型的计算。

当把?:操作符作用于数字操作数时，第二个和第三个操作数要使用相同的数字类型。宁愿使用不变的变量，不愿使用内置的幻数。

参考：谜题5、8和24。

A.4.2 操作符的操作数是从左到右计算的

规则：要避免在同一个表达式中对相同的变量多次赋值。尤其是在同一个表达式中的多重复合赋值操作符特别令人迷茫。

参考：谜题7、25和42；[JLS 15.7]和[EJ Item 37]。

A.4.3 操作符的优先级并不总是很明显

规则：要使用括号而不是空格来让优先级变得明显。要用具名的常量变量来替换内联的常量表达式。

参考：谜题11、35和95。

A.4.4 操作符==和!=在被包装的原生类型上执行引用比较

规则：要想强制进行值比较，需要在比较之前将一个操作数赋值或转型成恰当的原生类型。

参考：谜题32，[JLS 15.21，5.1.8]。

A.4.5 常量变量在所用的地方是内联的

规则：要避免导出常量域，除非它们表示的是永远都不会变化的真正的常量。可以使用一个恒等函数将一个表达式变成非常量。

参考：谜题93，[JLS 4.12.4, 13.1, 15.28]。

A.4.6 操作符&和|即使在作用于布尔类型的数值时，也要同时计算其两个操作数

规则：要避免将&和|作用于布尔类型的操作数。对有意识的使用要加以注释。

参考：谜题42，[JLS 15.22.2, 15.24]。

A.5 控制流

本节涉及影响控制流的语句和异常。

A.5.1 在switch case语句中缺少break将导致控制流一貫而下

规则：不要从一个非空的case一貫而下到另一个case：要用一个break终止每一个非空的case。对任何有意识的一貫而下要加以注释。

参考：谜题23，[JLS 14.11]。

A.5.2 在Integer.MAX_VALUE上终止以int为索引的循环是困难的

规则：要在Integer.MAX_VALUE处终止一个以int为索引的循环，需要使用long类型的循环索引，或者要非常仔细小心地编写该循环。

参考：谜题26。

A.5.3 finally语句块的意外完成将掩饰挂起的控制传输

规则：要确保每一个finally语句块都正常完成，以防止致命的错误。不要从一个

finally语句块中返回或抛出一个异常。

参考：谜题36和41，[JLS 14.20.2]。

A.5.4 为正常的控制流使用异常将导致bug和很差的性能

规则：应该只为异常情况使用异常，永远不要为正常的控制流使用异常。

参考：谜题42，[EJ Item 39]。

A.6 类初始化

本节涉及类加载和初始化。

A.6.1 类初始化是自顶向下的

规则：要确保静态域以恰当的顺序被初始化。要使用惰性初始化来解决初始化循环问题，该问题可以涉及一个或多个类。

参考：谜题49和52；[JLS 12.4.2]和[EJ Item 48]。

A.6.2 NoClassDefFoundError出现的时机是不可靠的

规则：不要捕获NoClassDefFoundError，而是应该使用反射并捕获ClassNotFoundException。

更一般地讲，不要捕获Error及其子类。

参考：谜题44，[JLS 12.2.1]。

A.7 实例的创建与销毁

本节涉及构造器、伪构造器、实例初始化和终止。

A.7.1 实例初始器在构造器方法体之前执行

规则：如果自身类型的实例域在构造阶段会引发递归，那么要确保该递归能够终止。

参考：谜题40，[JLS 12.5]。

A.7.2 在构造器中调用被覆写的方法会导致该方法在实例初始化之前运行

规则：永远不要在构造器中调用可覆写的方法。

要用惰性初始化来解决初始化循环问题。

参考：谜题51，[EJ Item 15和48]。

A.7.3 引用无效失败会导致内存泄露

规则：要能够使长生命周期的对象中已过时的对象引用变成无效。不能做到这一点就会导致内存泄露，在诸如Java这种垃圾回收语言中，更恰当的说法是无意识的对象保留。

参考：[EJ Item 5]。

A.7.4 添加私有构造器失败会使类不可实例化

规则：如果你想让一个类不可实例化，那么就添加一个私有构造器。

更一般地讲，应该总是至少提供一个构造器，永远都不要依赖于缺省的构造器。

参考：[EJ Item 3]。

A.7.5 终结器是不可预知的、危险的，且速度很慢

规则：要避免使用终结器。

参考：[EJ Item 6]。

A.7.6 被克隆的对象可以共享内部状态

规则：要避免实现Cloneable接口。如果实现了它，那么你应该复制所有不想在该对象及其克隆体之间共享的内部对象。

参考：[EJ Item 10]。

A.8 其他与类和实例相关的主题

本节涉及方法调用、方法设计、类设计和泛型。

A.8.1 在静态方法上没有任何动态分派

规则：永远不要用表达式来限定静态方法调用，应该总是用类型来限定。

参考：谜题48，[JLS 15.12.4.4]。

A.8.2 内部类是令人迷茫的

规则：优先考虑使用静态成员类而不是内部类。

参考：谜题80、89、90和92，[EJ Item 18]。

A.8.3 不能做保护复制就会破坏不变性

规则：在需要的时候，要对输入参数和输出值做保护复制。

参考：[EJ Item 24]。

A.8.4 实现一个接口会影响实现类的API

规则：不要去实现一个可以获取对其静态域的无限定访问权的接口。

不要编写只是由域组成的接口，即所谓的常量接口。

在5.0或更新的版本中，可以使用静态导入作为静态接口这一有害模式的替代物。

参考： [EJ Item 17] 和 [Java-5.0]。

A.8.5 int常量作为枚举值不安全

规则：应该使用enum类型，或者，如果你使用5.0之前的版本，应该实现类型安全的枚举。

参考： [EJ Item 21]。

A.8.6 混合使用原生类型和参数化类型将弱化类型检查机制

规则：应该在你的代码中消除被报告的“不受检查”的警告。

在5.0或更新版本的Java代码里，避免使用原生类型。

参考：谜题88，[JLS 4.8]。

A.8.7 返回null而不是0长度的数组或集合有产生错误的倾向

规则：不要从一个返回数组或集合的方法中返回null。

参考： [EJ Item 27]。



A.9 名字重用

本节涉及在Java中许多可能的名字重用形式。覆写的规则是：除了覆写之外，一律应该避免名字重用。也可以查看第8章中最后的“名字重用术语表”。

A.9.1 想要覆写时很容易就变成了重载

规则：请机械地复制每一个你想要覆写的超类方法的声明。但最好还是让IDE帮忙做。如果你使用的是5.0或更新的版本，请使用@Override注释。

参考：谜题58。

A.9.2 重载解析规则不明显

规则：要避免重载。

如果在你的API中的两个方法都可以应用于某些调用，那么请确保这两个方法在相同的实参上具有相同的行为。

参考：谜题11、46和74；[JLS 15.12.2]和[EJ Item 1和26]。

A.9.3 隐藏实体的程序难以理解

规则：要避免隐藏。

参考：谜题66、72、73和92；[JLS 8.3, 8.4.8.2, 8.5]。

A.9.4 遮蔽实体的程序难以理解

规则：要避免遮蔽。

不要在公共API中重用java.lang.Object中的名字。

不要试图在一个已经被定义的名字上使用静态导入。

参考：谜题71、73、79和89；[JLS 6.3.1, 15.12.1]。

A.9.5 遮掩实体的程序难以理解

规则：要避免遮掩。要遵守命名习惯。

参考：谜题68和69；[JLS 6.3.2, 6.5.2, 6.8]和[EJ Item 38]。

A.9.6 与所在类具有相同名字的方法看似构造器

规则：要遵守命名习惯。

参考：谜题63；[JLS 6.8]和[EJ Item 38]。

A.9.7 重用平台类名的程序难以理解

规则：要避免重用平台类的名字，并且永远不要重用java.lang中的类名。

参考：谜题67。

A.10 字符串

本节涉及字符串。

A.10.1 数组不能覆写Object.toString

规则：对于char数组，应该使用String.valueOf来获取表示指定字符序列的字符串。对于其他类型的数组，应该使用Arrays.toString，或者如果是5.0以前的版本，应该使用Arrays.asList。

参考：谜题12，[JLS 10.7]。

A.10.2 String.replaceAll以正则表达式作为第一个参数

规则：要确保该参数是一个合法的正则表达式，要不然就用String.replace来代替。

参考：谜题20。

A.10.3 String.replaceAll以置換字符串作为第二个参数

规则：要确保该参数是一个合法的置換字符串，要不然就用String.replace来代替。

参考：谜题20。

A.10.4 重复地进行字符串连接可能导致极差的性能

规则：要避免在循环中使用字符串连接。

参考：[EJ Item 33]。

A.10.5 从字节数组到字符数组的转换需要指定字符集

规则：在将一个byte数组转换成一个字符串或一个char数组时，总是要选择一个字符集；如果你没有这么做，就会使用平台缺省的字符集，从而导致不可预知的行为。

参考：谜题18。

A.10.6 char类型值只会默认转换成int，而不是String

规则：要想把一个char转换成一个字符串，应该使用String.valueOf(char)。

参考：谜题11和23，[JLS 5.1.2]。

A.11 I/O

本节涉及java.util.io包。

A.11.1 Stream.close可以抛出IOException异常

规则：要在close上捕获异常，并且一般做法是忽略这些异常。

参考：谜题41。

A.11.2 PrintStream.write(int)不刷新输出流

规则：要避免使用PrintStream.write(int)。如果你使用了它，要在需要时调用flush。

参考：谜题81。

A.11.3 要消费掉一个进程的输出，否则该进程可能挂起

规则：应该总是消费掉你所创建的进程的输出。

参考：谜题82。

A.12 线程

本节涉及多线程编程。多线程编程非常困难！驾驭它的法则就是：要避免低层的多线程编程。而是，要使用更高层的多线程抽象，诸如5.0版中通过java.util.concurrent引入的那些抽象。这是在陷阱15.5中所提建议的一个很重要的特例。

A.12.1 调用Thread.run不能启动一个线程

规则：永远都不要调用Thread.run。

参考：谜题76。

A.12.2 库类可能锁住或通知它们的实例

规则：如果你在扩展一个库类，那么请不要使用实例锁。取而代之的是，应该使用存储在一个私有域中的单独的锁对象。

参考：谜题77，[EJ Item 15]。

A.12.3 Thread.interrupted会清除中断状态

规则：不要使用Thread.interrupted，除非你想清除当前线程的中断状态。

参考：谜题84。

A.12.4 类初始化过程中将持有该类的锁

规则：要避免死锁的风险，永远不要在类初始化过程中等待一个后台线程。

参考：谜题85，[JLS 12.4.2]。

A.12.5 在共享可变状态时同步失败，可能导致不能观察状态的变化

规则：要同步对共享的可变状态的访问。

参考：[EJ Item 48]。

A.12.6 在被同步的语句块中调用外部方法可能导致死锁

规则：永远不要将控制流让给在被同步的方法或语句块中调用的外部方法。

参考：[EJ Item 49]。

A.12.7 在while循环的外部调用wait方法会引发不可预知的行为

规则：永远不要在一个while循环的外部去调用wait。

参考：[EJ Item 50]。

A.12.8 对线程调度器的依赖可能导致不定的且平台依赖的行为

规则：为了编写健壮的、响应迅速的和可移植的多线程程序，应该确保在任何给定的时刻，都只有极少数的线程是可运行的。

参考：[EJ Item 51]。

A.13 反射

本节涉及Java的核心反射API。

A.13.1 反射将检查对实体和实体所属类的访问权限

规则：用反射实例化类，用接口访问实例。

参考：谜题78，[EJ Item 35]。

A.13.2 用反射实例化内部类需要一个额外的参数

规则：不要在内部类上使用反射。

优先考虑使用静态成员类，而不是内部类。

参考：谜题80，[JLS 13.1]和[EJ Item 18]。

A.13.3 Class.newInstance可以抛出未声明的受检查异常

规则：只要存在构造器会抛出受检查异常的任何可能性，那么就应该使用java.lang.reflect.Constructor.newInstance而不是Class.newInstance。

参考：谜题43。

A.14 序列化

本节涉及Java的对象序列化系统。

A.14.1 让一个类可序列化将引入一个公共的伪构造器

规则：在让一个类可序列化之前，一定要三思而后行。

在接受缺省的readObject方法之前，一定要三思而后行。

编写readObject方法时，要采取保护性措施。

参考：[EJ Item 54和56]。

A.14.2 序列化形式是类的公共API的一部分

规则：在设计序列化形式时，应该与设计任何其他的API一样小心仔细。

参考：[EJ Item 54和55]。

A.14.3 使用缺省的序列化形式会在类的公共API中泄漏私有域

规则：应该考虑使用某种定制的序列化形式。

参考：[EJ Item 55]。

A.14.4 使用缺省的序列化形式可能会导致性能低下

规则：应该考虑使用某种定制的序列化形式。

参考：[EJ Item 55]。

A.14.5 维护实例控制的不变规则需要一个readResolve方法

规则：应该总是为单例、自编的类型安全的枚举类型以及其他实例控制的可实例化

的类编写一个readResolve方法。

参考：谜题83，[EJ Item 2和57]。

A.14.6 声明序列版本UID失败会导致脆弱

规则：应该在可序列化的类中声明一个显式的序列版本UID。

参考：[EJ Item 54和55]。

A.14.7 如果readObject或readResolve调用了可覆写的方法，反序列化循环的对象图可能引发崩溃

规则：如果一个HashSet、HashMap或Hashtable将被序列化，那么要确保其内容不会返回来引用到它。

在readObject和readResolve方法中，要避免在当前正在被序列化的对象上调用方法。如果你无法听从这条建议，那么也要确保在对象图中不会存在任何有问题的循环。

参考：谜题91。

A.15 其他库

本节涉及各种Java平台库。

A.15.1 覆写equals方法而不覆写hashCode方法可能会引发不定的行为

规则：当覆写equals方法时，总是要一并覆写hashCode方法。

参考：谜题57，[EJ Item 8]。

A.15.2 Calendar和Date设计得很差劲

规则：在使用Calendar和Date时，要小心仔细。无论何时用到它们，都一定要参

考API文档。

参考：谜题61。

A.15.3 许多类不管其方法名是什么，这些类都是不可变的

规则：不要被误导而认为不可变的类型是可变的。不可变的类型包括String、Integer、Long、Short、Byte、Character、Boolean、Float、Double、BigInteger和BigDecimal。

参考：谜题56。

A.15.4 某些被弃用的方法对程序来说就是毒药

规则：要避免使用已经被弃用的方法，例如Thread.stop、Thread.suspend、Runtime.runFinalizersOnExit和System.runFinalizerOnExit。

参考：谜题39和43，[ThreadStop]。

A.15.5 使用自编的解决方案而不是库容易导致努力白费、bug产生以及极差的性能

规则：要了解并使用库。

参考：谜题60、62和94，[EJ Item 30]。



书中幻图的注释

本附录包含了对出现在全书中的幻图的简短描述，这些描述按种类进行了松散的分组。在每个种类中，大致是按照出现的先后顺序排序的。

多重图形

多重图形（ambiguous figure）是指可以用两种或更多种方式来观看的图画，尽管不是可以在同一时刻以多种方式来观看。多重图形的一种是二维的图画，它们可以表示数种不同的三维图画的一种。《多重立方体》（*ambiguous cube*）（原书67页¹）就可以用三种方式来观看：看作是一个大的立方体在一个角上缺失了一个小立方区域；看作是一个小立方体坐落在一个大立方体内部的一个角上；看作是一个大立方体在一个角上突出了一个小立方体。

另一种多重图形是图形背景幻图（figure-ground illusion），这种图画依赖于你对其图形的感知和对其背景的感知，可以用两种方式来观看。原书158和231页上的图画可以看作是指向外部的黑色箭头和白色的背景，也可以看作是指向内部的白色箭头和黑色的背景。原书97页上的图画可以看作是白色的字母和黑色的背景，也可以看作是黑色的形状和白色的背景。

不可能的图形

不可能的图形（impossible figure）是指不可能在三维世界中存在的一个图形的二维透视图画。原书122页上的立方体的排列就是基于瑞典画家Oscar Reutersvård在1934年创

1. 经过精心排版，本中译本的页码基本与原书的一一对应。——编者注

作的一个不可能的图形而构建的。这幅图画被认为是有史以来创作出来的第一幅不可能的图形。Reutersvärd终身致力于不可能的图形的创作。

《彭罗斯三角形》(*Penrose Triangle*) (原书36和126页) 与Reutersvärd的立方体三角形密切相关,但是它是由物理学家罗杰·彭罗斯在1954年独立创作的。《彭罗斯楼梯》(*Penrose Stairway*) (原书63页) 是由生物学家和精神病学家莱昂内尔·彭罗斯,即罗杰·彭罗斯的父亲在20世纪50年代中期创作的。彭罗斯楼梯构成了M.C.Escher的著名的石版画《上升与下降》(*Ascending and Descending*) (1960) 的基础。

《三棍U形铁》(*Three-Stick Clevis*) (原书59和200页) 还有许多其他的名字,例如小工具(*Widgit*)、*Poiuyt*和不可能的三叉戟(*Impossible Trident*)。它的起源已经无人知晓,但是它至少可以追溯到1964年,那一年D.H.Schuster在《美国心理学学报》(*American Journal of Psychology*) 上写了一篇关于它的文章。这个U形铁还被刊登在《MAD》杂志1965年3月号的封面上,由微笑的Alfred E. Neuman高高地举着。《Ambihelical六边形螺母》(*Ambihelical Hex Nut*) (原书154和249页) 和《不可能的环》(原书109和164页) 是两个起源不明的更不可能的图形。

几何幻图：大小

《Jastrow幻图》(原书85页) 是由心理学家Joseph Jastrow在1891年绘制的。图中的两个阴影区域具有相同的大小和形状,但是大多数人会感觉上面那个要小一些。《Ebbinghaus (或Titchener) 幻图》(原书137页) 是由心理学家Hermann Ebbinghaus在1897年绘制的。图中位于中间的两个圆具有相同的大小,但是大多数人会感觉右面那个要小一些。

《Shepard幻图》(原书93页) 是基于斯坦福的心理学家Roger Shepard在1990年创作的一幅图画而绘制的[Shepard90]。图中的两个桌面具有相同的大小和形状,但是透视信息使它们看起来显得非常不同。Shepard在1981年第一次展示了其效果。

几何幻图：方向

《扭曲弦幻图》(*Twisted Cord illusion*) (原书13和221页),也被称为《Fraser图形》,是由心理学家James Fraser在1908年绘制的。在原书13页的图中的字母“CAFE babe”被设置为直的,而且事实确实如此,感觉到它是斜的只是一种幻觉。在原书221页的图中看起来倾斜的那个正方螺旋线其实是八个同心的正方形,它们被设置为直的,而且事实也确实如此。

《Ehrenstein幻图》(原书35页)是由心理学家Walter Ehrenstein在1925年所绘制的。图中那个正方形的边是直的，但是它们看起来却是向圆心弯曲的。

《咖啡馆外墙幻图》(Café Wall illusion)(原书39页)是由Fraser在1908年首次展示的，并由Richard L. Gregory和Priscilla Heard命名[Gregory79]。由黑白相间的瓷片所构成的线看起来是倾斜的，但是它们确实是完全水平的。该幻图之所以这样命名，是因为在英格兰布里斯托尔的一个小咖啡馆外墙上发现了这样的瓷片图案。

《靠垫幻图》(Cushion illusion)是由视觉科学家和画家Akiyoshi Kitaoka在1998年创作的。这幅图画完全由矩形和正方形构成，它们被设置为直，而且事实也是如此，那些弯曲都只存在于你的脑子中。如果你认为这难以置信，可以用直尺来量一下。《凸包幻图》(Bulge illusion)(原书65页)和《网状旗帜幻图》(Chequered Flag illusion)(原书169页)也是由Kitaoka在1998年创作的。它们都是完全由矩形和正方形构成的，都被设置为直的，而且事实也都是如此，那些凸包和波纹只存在于你的脑子中。所有这三幅幻图都是基于相同的底层效应。

《海龟幻图》(Turtles illusion)是由Kitaoka在2002年创作的。图中垂直的边看起来好像是斜的，但是它们是上下笔直的。其底层效应，被称为《装饰边幻图》(illusion of Fringed Edges)，是由Kitaoka、Pinna和Brelstaff共同绘制的[Kitaoka01]。

主观轮廓线

主观轮廓线(subjective contour)，也被称为幻觉轮廓线(illusory contour)，是实际上并不存在的而被感知到的边线。其经典范例是由完形心理学家Gaetano Kanizsa在1955年创作的《Kanizsa三角》(原书146和242页)。在图中好像有一个白色三角形浮在黑色三角形边框之上，但是构成这个白色三角形的边线并不存在，它们是你的大脑从这些“迷惑人”的图形所暗示的轮廓线中构建出来的。原书147页上的变化形式基于Branka Spehar所创作的一个图形[Spehar00]，而原书15和168页上的变化形式基于Takeo Watanabe和Patrick Cavanagh所创作的一个图形[Watanabe92]。原书100和101页上的三维变化形式基本上基于Bradley和Petry的《主观颈部立方体》(Subjective Necker Cube)，我们将在稍后讨论。原书69页的《Kanizsa圆点窗口》(Kanizsa Dot Window)基于Kanizsa在1979年所绘制的一个图形。

原书27页上的《带阴影的字母》(Shadow Letters)也是一个主观轮廓线幻觉。你的脑子会感知到字母A、B和C，所有这些字母都是以投射出的阴影来表示的。

异常运动幻图

异常运动幻图是指其组成部分看起来是在运动的图画。《MacKay的射线幻图》(*MacKay's Rays illusion*) (原书203页) 是由Donald MacKay在1957年创作的。当你在扫视该图时，围绕着图旋转本书，就会发现该图画在运动。原书47、192和230页上的图画都是基于MacKay在1961年创作的《MacKay的正方形幻图》(*MacKay's Squares illusion*)。当你在观看该图形时，会觉得它在闪烁。这个幻图构成了Reginald Neal的光效应绘画艺术作品Square of Three (1964) 和Square of Two (1965) 的基础。

《Ouchi幻图》(原书173页) 是画家Hajime Ouchi在1973创作的[Ouchi77]。图中的插人物看起来好像与主图形位于不同的平面，并且在不断地摇晃。

《闪烁网格幻图》(*Scintillating Grid illusion*) (原书161页) 是由Elke Lingelbach在1994年创作的[Schrauf97]。在网格线的交汇处，黑色的斑点看起来好像是在白色的底盘上闪烁。

原书96页的图画基于Kitaoka的《波纹幻图》(*Waves illusion*) (2004)。看起来就好像有缓缓的波纹在圆圈之间波动。《旋转蛇幻图》(*Rotating Snakes illusion*) (原书136页) 是由Kitaoka在2003年创作的。如果你还没有看过原作，那么你有必要去访问<http://www.psy.ritsumei.ac.jp/~akitaoka/rotsnakee.html>。其效果真的是令人震惊。波纹和旋转的蛇都基于外围飘逸幻觉 (Peripheral Drift illusion) 的渐变亮度轮廓[Kitaoka03b]。

亮度幻觉

亮度幻觉是指在图像中我们会错误地感知的某部分的亮度 (或流明)。最简单的亮度幻觉是同时反衬幻觉 (Simultaneous Contrast illusion)，在这样的图中，具有相同亮度的区域根据其显示背景的不同将会显得亮一些或暗一些。这种效应在很久以前就已经被人们所了解，原书163和165页上的图像展示了这种效应。在这两个图像中，中间的矩形和从外数的第二个框通体具有相同的灰度，但是在包围它们的边框变得比较暗时，它们看起来就比较亮了。

原书23页上的棋盘图案基本上基于Craik-O'Brien-Cornsweet幻觉，顶部和底部的正方形看起来比左边和右边的正方形更亮一些，但是所有四个正方形都是相同的。在顶部和底部的正方形的较亮的角指向内部，而左边和右边的正方形的较亮的角指向外部。正方形交界部分的边的亮度变换影响了你对正方形相对亮度的感知。

《Logvinenko幻图》(*Logvinenko's illusion*) (原书189页) 是由视觉科学家Alexander Logvinenko创作的[Logvinenko99]。尽管看起来可能有些怪, 立方体的水平面都具有相同的灰度。如果你认为在第一行和第三行的水平面比在第二行和第四行的水平面更亮, 那么请用一个遮盖物遮盖住垂直的面, 然后你就会因其结果大为惊叹了。该幻图的底层效应与Craik-O'Brien-Cornsweet幻觉是密切相关的。

另一个相似的效应在《Todorovic的梯度棋盘幻图》(*Todorovic's Gradient Chessboard illusion*) (原书149页) 中得到了运用。它是由视觉科学家Dejan Todorovic创作的[Todorovic97]。尽管棋盘上的圆盘看起来具有三种不同的灰度, 但是它们实际上是相同的。

《Vasarely幻图》是由光效应绘画艺术家Victor Vasarely创作的, 显示在原书228和229页上。在亮版本中 (原书228页) 好像在对角线上有一个暗十字, 而在暗版本中 (原书228页) 好像在对角线上有一个亮十字, 但是根本就没有任何十字存在。

原书211页上的图画基于White效应[White79]。由白色色条分隔的灰色色条看起来比由黑色色条分隔的灰色色条更亮, 但是所有的灰色色条实际上是一样的。

原书135页上的图画基于Adelson阴霾效应[Adelson 99]。中间看起来像晴天的菱形与两侧看起来像阴霾的菱形具有相同的灰度。

原书127页上的图画基本上基于Kitaoka的《菊花形发光体》(*Light of Chrysanthemum*) (2005), 它利用了Zavagno的炫目效应 (Zavagno's Glare effect) [Zavagno99]。图中花的中心具有不自然的光亮, 在花瓣之间好像有幻影般的迷雾。

复合幻觉

复合幻觉组合了两种或更多种幻觉效应。原书160页上的图画就是基于Ehrenstein图形的, 它是由Walter Ehrenstein在1941年发现的。它组合了主观轮廓线幻觉和亮度幻觉。你脑子里会感知到在网格线交汇处有圆圈, 而且这些圆圈具有不自然的光亮。

《主观颈部立方体》(*Subject Necker Cube*) (原书33和224页) 是由Bradley和Petry创作的[Bradley77]。它组合了主观轮廓线幻觉和多重图形。你可以看到两个立方体, 每次可以看到一个, 但是图中并没有任何立方体, 它只是暗示了构成这些立方体的边。

原书57页上基于视觉科学家Nicholas Wade所发现的幻觉的图画, 组合了主观轮廓线幻觉和亮度幻觉: 你脑子里会感知在圆弧相交处有辐射状的同心圆, 而且这些圆圈比周

围的页更亮。

原书48和49页上的图画基于Marc Albert和Donald Hoffman的《霓虹灯正方形幻图》(*Neon Square illusion*) [Albert99]。这个幻图组合了主观轮廓线幻觉和霓虹灯衍射幻觉(*Neon Spreading effect*) [van Tuij175]。你不但会感知到根本不存在的正方形，而且这些正方形还具有幻影般的色彩。原书48页上的幻觉正方形看起来烟雾缭绕，而原书49页上的正方形看起来色彩深暗。

原书125页上的图画所利用的效果与Kitaoka在靠垫、凸包和网状旗帜中所利用的效果相似。该图画看起来很像一个螺旋线，但是它是由同心圆构成的。如果你向着图画移动，然后再远离它，就会发现它好像在运动。

原书92页上的图画组合了在Kitaoka的《地震》(2001)中所发现的“焦距外”效应与MacKay的正方形。图中的边框就好像浮在没有对准焦距的网格上一样，而该网格看起来还在晃动。

最后，封面上的幻图组合了外围飘逸幻觉、闪烁网格和一个未命名的幻觉。外圈的鱼看起来是在顺时针旋转，而内圈的鱼看起来是在逆时针旋转，在网格线交汇点上的白圆盘看起来在闪烁，而这个插入的网格相对于鱼好像在轻微地移动。

索引

符号

- 一元取反操作符, 14, 72, 120
- 减法操作符, 154
!= 不等操作符, 70
% 取余操作符, 6, 150
%= 取余复合赋值操作符, 22, 68
%n printf的行尾格式, 37
& 与操作符, 7, 91
&& 条件与操作符, 91
&= 与复合赋值操作符, 22, 68
*= 乘法复合赋值操作符, 22, 68
+ 加法操作符, 26, 30, 66
 其优先级, 31
+ 字符串连接操作符, 26, 28, 30, 66
++ 递增操作符, 56
+= 加法复合赋值操作符, 22, 68
/ 整除操作符, 6, 44
/* */ 注释定界符, 42
// 行注释符, 36, 43
/= 整除复合赋值操作符, 22, 68
< 小于关系操作符, 70, 74, 237
<< 左移操作符, 60
<<= 左移复合赋值操作符, 22, 68

<= 小于等于关系操作符, 70
<=> Perl的比较操作符, 156
= 赋值操作符, 22, 24, 211
-= 减法复合赋值操作符, 22, 68
== 等于操作符, 30, 70, 211~212
== 等于操作符, 与equals的比较, 29~31
> 大于关系操作符, 70, 237
>= 大于等于关系操作符, 70
>> 右移操作符, 60
>>= 右移复合赋值操作符, 22, 68
>>> 无符号右移操作符, 60
>>> 无符号右移复合赋值操作符, 22, 67, 68
?: 条件操作符, 19, 237
 其操作数类型, 177~179
@override 注释, 139
\ 转义字符, 44, 46
\'单引号转义字符序列, 32
\\" 转义的反斜杠, 32
\n 换行转义字符序列, 32
\t 制表符转义字符序列, 32
\u Unicode转义字符序列, 35
^ 异或操作符, 18
^= 异或复合赋值操作符, 17, 22, 68
! 或操作符, 91

I = 或复合赋值操作符, 22, 68
II 条件或操作符, 91
~ 按位取补操作符, 211

数字

42, 175

A

abrupt completion, 意外结束, 78
access modifier, 访问权限修饰符
 hiding field and, 以及隐藏域, 158
 overriding method and, 以及覆写方法, 158, 168
accessing nonpublic type, 访问非公共类型, 189~192
acronym, naming convention and, 首字母缩拼词, 以及命名习惯, 164
addition operator, 加法操作符, 30
 precedence of, 其优先级, 31
advanced language feature, reflection and, 高级语言特性以及反射, 197
alternate constructor invocation, 交替构造器调用机制, 124
AND operator, 与操作符
bitwise, 按位与, 7
conditional, 关系与, 91
logical, 逻辑与, 91
annotation, @Override, 注释@Override, 139
annotation, Class, 注释Class, 216
antisymmetric relation, 反对称关系, 70
API change, incompatible, API变化, 不兼容, 174
arithmetic, 算术
 decimal, 小数, 9
 double, 双精度, 8

floating point, 浮点, 8, 62, 63, 64
IEEE 754, 62, 64
int, 10, 73
long, 10
mixed-type, 混合类型, 14, 69
modular, 模数, 73
two's complement binary, 2的二进制补码, 16, 59, 72, 151
arithmetic operation, type conversion in, 算术运算, 其中的类型转换, 68
arithmetic operator, 算术运算符
 overloading, 重载, 9, 133
 performance, 性能, 7
ArrayIndexOutOfBoundsException, 90, 151
Arrays.asList, 235
Arrays.deepToString, 141~143
Arrays.equals, 142
Arrays.hashCode, 142
Arrays.sort, 154
Arrays.toString, 142, 170
assignment compatibility, 赋值兼容性, 24
assignment operator, 赋值操作符
 See compound assignment operator, 查看复合赋值操作符
 See simple assignment operator, 查看简单复制操作符
assignment, definite, 赋值, 限定的, 81~83
autoboxing, 自动包装, 152, 169, 213
operator and, 以及操作符, 69~71

B

backslash character, 反斜杠字符, 44, 46
backward compatibility, 向后兼容性, 173~175
BigDecimal, 9
BigDecimal.signum, 143

- B**
- `BigInteger`, 131~133
 - `BigInteger.bitCount`, 143
 - `BigInteger.signum`, 143
 - binary compatibility**, 二进制兼容性, 231
 - binary floating-point arithmetic**, 二进制浮点算术, 8, 63
 - binary numeric promotion**, 二进制数字提升, 20, 21, 212
 - bit**, 位
 - masks**, 掩码, 17
 - shifts**, 移位, 59~61
 - twiddling**, 反转, 141~143
 - bitwise AND operator**, 按位与操作符, 7
 - blank final**, 空final, 81~83
 - block comment**, 块注释, 41~43
 - blocking**, 阻塞, 200
 - boxed numeric type**, 被包装的数字类型, 70
 - break statement**, break语句, 49~52
 - buffers**, flushing, 缓冲, 刷新, 197~199
 - bugs**, tools for finding, bug查找工具, 3, 50
 - `Byte.MAX_VALUE`, 54
- C**
- Calendar problem**, Calendar问题, 143~145
 - callback**, 回调, `ObjectInputValidation.validateObject`, 227
 - cast operator**, 转型操作符, 114
 - casting**, 转型, 15~17, 113~114
 - hiding and**, 隐藏, 159
 - unchecked**, 不受检查, 95
 - catch clause**, rule for, catch子句, 其规则, 79~81
 - char**
 - arrays**, string representation of, 数组, 其字符串表示, 28
 - concatenation of**, 其连接, 26
 - character literal**, escape sequence and, 字符字面常量, 以及转义字符序列, 33
 - character set**, 字符集, 39~41
 - character vs. integer interpretation**, 字符与整数的释义, 26
 - `Character.reverseBytes`, 143
 - `Charset.defaultCharset`, 40
 - checked exception**, 受检查的异常
 - catch clauses and**, 以及catch子句, 80
 - method declarations and**, 以及方法声明, 41, 80
 - re-raising**, 再构建, 204
 - static initializers and**, 以及静态初始器, 81
 - `Class.forName`, 其返回类型, 216
 - `Class.getAnnotation`, 216
 - `Class.getName`, 28
 - `Class.newInstance`, 86, 94
 - specification for**, 其规范, 196
 - vs.** `Constructor.newInstance`, 与
 - `ClassCastException`, 114, 215
 - class**, 类
 - extending inner**, 扩展内部类, 221~223, 229~230
 - initialization cycle**, 初始化循环, 111~113
 - initialization of**, 类的初始化, 119~121, 205~207
 - deadlock during**, 在类的初始化过程中的死锁, 207
 - inner**, generic and, 内部类, 以及泛型, 217~220
 - library**, constants and, 库, 以及常量, 231~233
 - member**, 成员, 197
 - static vs. nonstatic**, 静态与非静态的, 219
 - missing**, 缺失, 97~100
 - name reuse**, 名字重用, 161~163
 - naming**, 命名, 148, 164
 - nested static**, type parameter and, 嵌套的静态

- 类, 以及类型参数, 220
`Object.toString` and, 以及`Object.toString`, 28
 precedence of member, 成员的优先级, 170
 reflection and, 以及反射, 100
 static vs. inner, 静态内部类与内部类的对比, 223
`ClassNotFoundException`, 97~100
`CloneNotSupportedException`, 80
`Closeable.close`, 88
 code migration to generic, 代码迁移到泛型上, 216
`Collections`, 155
`Collections.reverseOrder`, 155
`Collections.shuffle`, 235
 comment, 注释
 delimiter, 定界符, 32, 42, 43
 nesting, 嵌套, 42
 string literal in, 其中的字符串字面常量, 41~43
 Unicode escape in, 其中的Unicode转义字符, 33~37
`Comparator`, 237~238
 subtraction based, 基于减法的, 152~156
 comparison operator, 比较操作符
 numerical, list of, 数字比较操作符列表, 70
 rule for operand of, 其操作数的规则, 70
 comparison, 比较
 mixed-type, 混合类型的, 53~55, 74, 212
 reference vs. value, 引用比较与数值比较的对比, 70, 71, 146
 compilers, 编译器
 error detection by, 其错误探测, 3, 50
 halting problem and, 以及中止问题, 82
 compile-time constant expression, 编译期常量表达式, 232
 composition, vs. inheritance, 组合与继承, 107~109,
 184
 compound assignment operator, 复合赋值操作符, 22, 67~69
 for addition, 加法的, 22
 avoiding type conversion in, 其间的避免类型转换, 22~23
 casting, 转型, 21~23
 for exclusive OR, 异或的, 17
 list of, 其列表, 68
 operand, 操作数, 23~24
 result type of, 其结果类型, 22
 rules for operand of, 其操作数的规则, 24
 computation, mixed-type, 计算, 混合类型的, 14, 69
 concatenation, 连接
 operator, 操作符, 26, 28, 30, 66
 precedence of, 其优先级, 31
 string字符串, 25~27
 conditional compilation, 条件编译, 43
 conditional expression, 条件表达式, 19~21, 237
 operand types of, 其操作数类型, 177~179
 result types of, 其结果类型, 20
 conditional operator, 条件操作符, 89~91
 constant variable, 常量变量, 232
 constant, 常量
 inheriting from interface, 从接口继承, 171
 library class and, 以及类库, 231~233
 vs. magic number, 与魔幻数字的对比, 55, 76
 naming, 命名, 164
`Constructor.newInstance`, 95
 constructor, 构造器
 exception declaration by, 其异常声明, 86
 hidden, serialization and, 以及隐藏、序列化, 201~203
 of inner class, 内部类的构造器, 195~197
 invoking superclass, 调用超类构造器, 222
 of non-static nested class, 非静态嵌套类的构造

- 器, 196
- order of execution in**, 其中的执行顺序, 86
- overridable method and**, 以及可重载的方法, 227
- signature of**, 其签名, 148
- vs. member**, 与成员的对比, 174
- vs. method**, 与方法的对比, 147~148
- conversion**, 转换
- See **type conversion**, 查看类型转换
- See **unboxing conversion**, 查看未包装类型转换
- covariant return type, reflection and**, 协变式返回类型, 以及反射, 197
- D**
- Date problem**, Date问题, 143~145
- Date, name conflict with Date**, 其命名冲突, 163
- Deadlock**, 死锁, 200, 205~207
- during class initialization**, 在类初始化过程中的死锁, 207
- finalizers and**, 以及初始器, 85
- decimal arithmetic**, 小数运算, 9
- decimal literal**, 小数字面常量, 14
- declaration**, 声明
- obscuring and**, 以及遮掩, 170
- of exception**, 异常声明, 41
- parameterized**, 参数化的声明, 215
- shadowing and**, 以及遮蔽, 170
- default constructor**, 缺省构造器, 148
- definite assignment**, 明确赋值, 81~83
- deprecated method**, 过时的方法
- Runtime.runFinalizersOnExit**, 84
- System.runFinalizerOnExit**, 84
- Thread.stop**, 93
- Thread.suspend**, 93
- deserialization of object graph**, 对象图的反序列化, 224~228
- Double.NaN**, 64, 212
- Double.POSITIVE_INFINITY**, 62
- Double.toString**, 8
- dynamic dispatch**, 动态分派, 180
- static method and**, 以及静态方法, 110
- E**
- Elvis**, 111~113
- encapsulation**, 封装, 159, 173
- end-of-line comment**, 行尾注释, 48
- enum constant**, 枚举常量, 233
- enum type**, 枚举类型, 152, 179
- reflection and**, 以及反射, 197
- equality**, 等价性
- operator**, 操作符, 70, 71
- reference**, 引用, 146
- value**, 值, 146
- equals, vs. ==, equals与==对比**, 29~31
- equivalence class**, 等价类, 211
- Error**, 86
- Catching**, 捕获, 100
- error stream, draining**, 错误流, 排出, 201
- escape sequence**, 转义序列, 32
- in character literal**, 在字符字面常量中的转义序列, 33
- HTML entity**, HTML实体, 34
- octal**, 八进制, 32
- in string literal**, 在字符串字面常量中的转义序列, 33, 44
- Unicode**, 31
- in comment**, 在注释中, 33~35, 35~37
- compilation and**, 以及编译, 32, 36
- vs. escape sequence**, 与转义序列的对比, 31~33, 37
- obfuscation and**, 以及混淆, 37~38

- in string literal, 在字符串字面常量中的转义序列, 32
- Windows file name and, 以及Windows文件名, 34
- vs. Unicode escape, 与Unicode转义字符的对比, 31~33
- e**valuation order, 计算顺序, 75~76
- E**xception
- catching, 捕获, 80
 - serialization and, 以及序列化, 202
- exception checking, bypassing, 异常检查, 旁路, 93~96
- exception, 异常
- See also checked exception, 查看受检查的异常
 - See unchecked exception, 查看不受检查的异常
 - checking, 检查, 96
- `Class.newInstance` and, 以及`Class.newInstance`, 94
- loop control and, 以及循环控制, 89~91
 - verification and, 以及校验, 99
- exclusive OR operator, 异或操作符, 17, 18
- exponent in floating-point representation, 浮点的指数表示, 72
- exponential algorithm, 指数算法, 101~103
- expression statement, 表达式语句, 48
- expression, evaluation order of, 表达式, 以及计算顺序, 18, 19
- extending inner class, 扩展内部类, 221~223, 229~230
- F**
- factorial, 阶乘, 234
 - fairness of shuffling, 洗牌的公正性, 233~237
 - fencepost error, 栅栏柱错误, 49~52
 - field, 域
- hiding, access modifier and, 隐藏, 以及访问修饰符, 158
 - int, initial value of, int, 其整型数值, 226
- I**File.separator, 45
- final
- field, hiding and, 域, 以及隐藏, 171~172
 - method vs. field, 方法与域的对比, 172
- finalizer, deadlock and, 终结器, 以及死锁, 85
- finally clause, finally子句
- exception in, 其中的异常, 88
 - exiting, 退出, 78, 88
- first common superclass, 首个公共超类, 99
- flag for javac, javac的标记
- source, 177
 - Xlint:fallthrough, 50
- `Float.NaN`, 212
- floating point arithmetic, 浮点运算, 8
- infinity in, 其中的无穷大, 61~63
 - NaN, 63~64
- floating-point representation, 浮点表示, 72, 74
- flow analysis, 流分析, 99
- flushing buffer, 刷新缓冲区, 197~199
- for-each loop, for-each循环, 213
- formal parameter, 形参, 106
- G**
- generic interface, 泛型接口, 214
- generic, 泛型, 152, 153
- inner class and, 以及内部类, 217~220
 - reflection and, 以及反射, 197
 - type checking for, 其类型检查, 95
 - vs. raw type, 与原生类型的对比, 213~216
- H**
- half-open loop form, 半开循环形式, 74
- halting problem, 中止问题, 82

- hashCode contract, hashCode约定, 133~135
 hashCode, identity based, hashCode, 基于标识, 134
 HashMap, 146, 148, 226
 serialization and, 以及序列化, 228
 HashSet, serialization and, HashSet, 以及序列化, 228
 HashSet.readObject, 226
 Hashtable, serialization and, Hashtable, 以及序列化, 228
 hexadecimal, 十六进制, 14, 31, 37, 54
 hexadecimal representation of int, int的十六进制表示, 72
 hidden constructor, 被隐藏的构造器, 201~203
 hiding, 隐藏, 110, 111, 157~160, 174
 casting and, 以及转型, 159
 definition of, 其定义, 180
 field, access modifier and, 以及域、访问修饰符, 158
 final field and, 以及final域, 171~172
 vs. overloading, 与重载的对比, 109~111
 HTML entity escape, HTML实体转义, 34
- I**
- identity-based hash-code, 基于标识的散列码, 134
 IdentityHashMap, 145~146
 idiom, private constructor capture, 惯用法, 私有构造器捕获, 123~124
 if statement to comment out code, if语句来注释代码, 43
 IllegalAccessException, 86, 95
 IllegalArgumentException, 94, 144
 immutability, 不可变性, 132
 incompatible API change, 不兼容的API修改, 174
 inference, of return type, 推演, 返回类型的推演, 216
 infinite recursion, 无穷递归, 85~87
 infinity, 无穷性, 61~63
 inheritance, 继承
 of private member, 私有成员的继承, 229~230
 vs. composition, 与组合的对比, 107~109, 184, 194
 initialization, 初始化
 of class, 类的初始化, 119~121, 205~207
 instance, 实例, 115~119
 of instance variable, 实例变量的初始化, 85~87
 of object graph, 对象图的初始化, 226
 inner class, 内部类
 default public constructor of, 其缺省的公共构造器, 196
 extending, 扩展, 221~223, 229~230
 generic and, 以及泛型, 217~220
 vs. static, 与静态内部类的对比, 223
 instance, 实例
 field, updating, 域, 更新, 220
 initialization, 初始化, 115~119
 lock, 锁, 187
 method, qualifier of, 方法, 及其限定符, 110
 variable initialization, 变量初始化, 86
 instanceof, 113~114
 instantiation, reflection and, 实例化, 以及反射, 191
 InstantiationException, 86, 95, 196
 integer division, 整数整除, 5~7, 9~11
 integer literal, 整数字面常量
 octal value of, 八进制数值的整数字面常量, 140
 padding, 填充, 141
 Integer.bitCount, 143
 Integer.highestOneBit, 143
 Integer.lowestOneBit, 143
 Integer.MAX_VALUE, 58, 74, 209~210
 subtraction-based Comparator and, 以及基于减法的Comparator, 155

I
 Integer.MIN_VALUE, 58, 72, 151, 210
 Math.abs and, 以及Math.abs, 149~151
 Integer.numberOfLeadingZeros, 143
 Integer.numberOfTrailingZeros, 143
 Integer.reverse, 143
 Integer.reverseBytes, 143
 Integer.rotateLeft, 143
 Integer.rotateRight, 143
 Integer.signum, 143
 integral type, boundary condition of, 整数类型, 整数类型的边界条件, 58
 interface, 接口
 for method invocation, 方法调用的接口, 191
 inheriting constant from, 从接口继承常量, 171
 interned string, 内存限定的字符串, 30, 146
 InterruptedException, 80, 204
 InvocationTargetException, 95
 IOException, 80, 88
 isInterrupted, 204
 ISO-8859-1, 40

J

java.lang, name reuse and, java.lang, 以及名字重用, 162
 java.lang.reflect.Constructor, vs. Class.-newInstance, java.lang.reflect.Constructor与Class.newInstance的对比, 196
 java.util.concurrent.locks, 187
 java.util.regex, 141~143
 java.util.ThreadLocal, 124
 javac, 50, 177
 join point, 连接点, 99

K

keyword, 关键词

final, 172
 null, 126, 232
 super, 159
 this, 124, 223

L

language, safe vs. unsafe, 语言, 安全与不安全的对比, 99
 Latin-1 character set, Latin-1字符集, 40
 least common supertype, 最小公共超类型, 178
 left shift operator, 左移操作符, 59
 less-than operator, 小于操作符, 74
 library, development of, 库, 库的开发, 231~233
 library class, 库类
 constant in, 其中的常量, 231~233
 locking and, 以及加锁, 187
 line separator, 行分隔符, 35~37
 line terminator, 行终止符, 36
 linkedHashSet, 141~143
 linking, step in, 链接, 其中的步骤, 99
 Liskov Substitution Principle, Liskov置换原则, 160
 List, 142
 Literal, 字面常量, 55
 class, generic type and, 以及类、泛型, 216
 naming of, 其命名, 12
 string, escape sequence and, 字符串, 以及转义序列, 33
 string, in comment, 字符串, 在注释中, 41~43
 Unicode escape in, 其中的Unicode转义字符, 32
 value of, 其值, 14
 local variable declaration statement, 局部变量声明语句, 127~130
 locking, 加锁, 185~188
 of instance vs. thread, 实例加锁与线程加锁的对比, 188

- library class and, 以及库类, 187
lock, reentrant, 锁, 重入, 187
logical operator, 逻辑操作符, 89~91
long arithmetic, long运算, 10
Long.bitCount, 143
Long.highestOneBit, 143
Long.lowestOneBit, 143
Long.MIN_VALUE, 72
 Math.abs and, 以及Math.abs, 151
Long.numberOfTrailingZeros, 143
Long.reverse, 143
Long.reverseBytes, 143
Long.rotateLeft, 143
Long.rotateRight, 143
Long.signum, 143
loop, 循环
 exception for control of, 循环控制的异常, 89~91
 type of indices for, 循环索引的类型, 75
- M**
- magic number**, 魔幻数字, 55, 76
mantissa, in floating-point representation, 尾数, 在浮点表示法中, 72
Map, 146
Matcher.quoteReplacement, 46
Math.abs, 149~151
Math.signum, 143
member, 成员, 174
 class, static vs. nonstatic, 类, 静态与非静态的对比, 197, 219
 precedence, vs. static import, 优先级, 与静态导入的对比, 170
private, inheritance of, 私有, 成员的继承, 229~230
method, 方法
- exception declaration, 异常声明, 41
naming, 命名, 133, 148
overloading, 重载, 21, 29
overridable, constructor and, 可覆写的, 以及构造器, 227
overriding package-private, 覆写, 包私有, 167~168
overriding, access modifier and, 覆写, 以及访问限定符, 158
signature of overriding, 覆写的签名, 139
synchronized, 同步的, 184
unintentional overloading of, 无意的方法重载, 137~139
vs. constructor, 与构造器的对比, 147~148
migrating code to generic, 将代码移植为泛型的, 216
mixed type, 混合类型
 arithmetic on, 其上的算术, 69
 comparison of, 其比较, 53~55, 74, 212
 computation with, 以及计算, 13~15, 69
 conditional expression and, 以及条件表达式, 20
 equality operator and, 以及等价操作符, 71
modular arithmetic, 模算术, 73
monetary calculation, 货币计算, 7~9
multiple assignment, 多重赋值, 19, 55~56
multiplication operator, 乘法操作符, 76
multithreading, 多线程, 184
 locking and, 以及锁, 185~188
 mutual exclusion, 互斥, 187
- N**
- name reuse**, 名字重用, 161~163, 175
naming, 命名
 class, 类, 148, 164
 conflict, within Java, 冲突, Java内的, 163
constant, 常量, 164

- convention, 惯例, 148, 164
 acronym and, 以及首字母缩写, 164
 literal, 字面常量, 12
 method, 方法, 133, 148
 package, 包, 164
 type parameter, 类型参数, 164
 variable, 变量, 11~12, 164
 NaN, 63~64
 narrowing primitive conversion, 窄化基本类型转换, 16, 21~23, 67~69
 nested class, reflection and, 嵌套类, 以及反射, 197
 nesting, 嵌套
 comment, 注释, 42
 try-catch construct, try-catch 结构, 88
 NoClassDefFoundError, 98
 nonstatic member class, 非静态成员类, 197, 219
 NoSuchMethodError, 162
 notify, 187
 notifyAll, 187
 null keyword, null 关键字, 126, 232
 NullPointerException, 114, 126
 numerical comparison operator, 数字型比较操作符, 70, 71
- O**
- Object, name conflict with, 对象, 以及名称冲突, 163
 Object.clone, 118
 Object.equals, 118, 134, 176
 Overloading, 重载, 138, 176
 object.getClass().getMethod(methodName), 191
 Object.hashCode
 contract, 约定, 133~135
 identity-based, 基于标识的, 134
 Object.notify, 187
 Object.notifyAll, 187
 Object.toString, 28, 170
 Object.wait, 187
 ObjectInputStream.readObject, 118, 224~228
 ObjectInputStream.registerValidation, 227
 ObjectInputValidation.validateObject, 227
 object, 对象
 graph, deserialization of, 图, 其反序列化, 226
 referencing during deserialization, 反序列化期间的引用机制, 228
 obscuring, 遮掩, 163~165
 definition of, 其定义, 182
 syntactic context and, 以及语法的上下文, 166
 workaround for, 其工作环境, 165~167
 octal, 八进制的
 escape, 转义, 32
 literal, 字面常量, 14
 value, 数值, 139~141
 operand, 操作数
 of comparison operator, 比较操作符的操作数, 70
 of compound assignment operator, 复合赋值操作符的操作数, 23~24
 of conditional expression, 条件表达式的操作数, 177~179
 of simple assignment operator, 简单赋值操作符的操作数, 24
 operator, 操作符
 See also compound assignment operator, 查看复合赋值操作符
 See also simple assignment operator, 查看简单赋值操作符
 addition, 加法, 30
 arithmetic, precedence of, 算术, 其优先级, 31

- autoboxing and, 以及自动包装, 69~71
 bitwise AND, 按位与, 7
 cast, 转型, 114
 comparison, 比较
 operand of, 其操作数, 70
 rule for, 其规则, 70
 concatenation, 连接, 66
 conditional AND, 条件与, 91
 conditional OR, 条件或, 91
 conditional vs. logical, 条件与逻辑操作符的对比, 91
 equality, 等价, 70, 71
 exclusive OR, 异或, 17, 18
 instanceof, 114
 integer division, 整数整除, 6
 left shift, 左移, 59
 less than, 小于, 74
 logical AND, 逻辑与, 91
 logical OR, 逻辑或, 91
 logical vs. conditional, 逻辑与条件操作符的对比, 89~91
 multiplication, 乘法, 76
 numerical comparison, 数字的比较, 71
 list of, 其列表, 70
 overloading of, 其重载, 9, 27, 65~66, 133
 performance of arithmetic, 运算的性能, 7
 postfix increment, 后缀递增, 56
 precedence of, 其优先级, 29~31, 76
 question-mark colon, 问号冒号
 See conditional expression, 查看条件表达式
 remainder, 余数, 6, 76, 150
 right shift, 右移, 60
 shift, behavior of, 移位, 其行为, 60
 string concatenation, 字符串连接, 26, 28, 30
 precedence of, 其优先级, 31
- three-valued comparator, 三值比较符, 156
 truncating integer division, 截尾的整数整除, 6
 unary negation, 一元取反, 14, 210
 unsigned right shift, 无符号的右移, 60, 67
 OR operator, 或操作符
 Conditional, 条件的, 91
 Logical, 逻辑的, 91
 order of evaluation, 赋值的次序, 75~76
 output stream, 输出流
 draining, 排出, 201
 flushing, 刷新, 197~199
 subprocess and, 以及子进程, 199~201
 overflow, 溢出, 9~11, 58, 71~73
 silent, 悄悄的, 151, 152~156
 overloading, 重载, 175~176
 of constructor 构造器的重载, 106
 definition of, 其定义, 181
 method, 方法, 29
 Object.equals, 138, 176
 PrintStream.print, 21
 PrintWriter.println, 28
 String.valueOf, 28
 StringBuffer.append, 28
 Operator, 操作符, 9, 27, 65~66, 133
 resolution of, 解析, 105~107, 194
 unintentional, 无意的, 137~139
 vs. hiding, 与隐藏的对比, 109~111
 overridable method, constructor and, 以及可覆写的方法, 构造器, 227
 overriding, 覆写, 110
 definition of, 其定义, 180
 method, 方法
 access modifier of, 其访问限定词, 158
 constructor and, 以及构造器, 118
 method signature, 方法签名, 139

- `Object.equals`, 134
`Object.hashCode`, 138
`Object.toString`, 29
`PrintStream.println`, 28
method, constructor and, 方法, 以及构造器, 226
package-private method and, 以及包私有方法, 167~168
- P**
- package-private method, overriding 包私有方法, 覆写, 167~168
package, 包
naming, 命名, 164
precedence of name, 名字的优先级, 164
parameterized declaration, 参数化声明, 215
parameterized type, 参数化类型, 214
parameter, 参数
formal, 形式的, 106
implicit, in inner class constructor, 隐式的, 在内部类的构造器中, 196
type, static nested class and, 类型, 以及静态嵌套类, 220
parenthesis, 括号, 209~210, 237~238
for grouping, 用于分组, 76
order of evaluation and, 以及赋值的次序, 76
string concatenation and, 以及字符串连接, 30
`Pattern.quote`, 44
pattern, typesafe enum, 模式, 类型安全的枚举, 179
platform dependency, 平台依赖性
file name separator, 文件名分隔符, 45
line separator, 换行符, 37
postfix increment operator, 后缀递增操作符, 56
precedence, 优先级
member vs. static import, 成员与静态导入的对比, 170
of operator, 操作符的优先级, 29~31, 76
variable name vs. type, 变量名称与类型的对比, 164
precision, silent loss of, 精度, 其默默的损失, 212
preorder traversal, 先序遍历, 103
`PrintStream`, specification of, `PrintStream`, 其说明, 198
`PrintStream.print`, 21, 198
`PrintStream.printf`, 37
`PrintStream.println`, 28, 37, 80, 84
`PrintStream.write`, 198
`printWriter.printf`, 37
`PrintWriter.println`, 28, 37
private constructor capture idiom, 私有构造器捕获惯用法, 123~124
private member, 私有成员, 173~175
Process, 200, 201
ProcessBuilder, 201
`ProcessBuilder.redirectErrorStream`, 201
programming style, instance field, 编程风格, 实例域, 220
pseudoconstructor, 伪构造器, 118
pseudorandom number generation, systematic bias in, 伪随机数发生器, 其中的系统偏差, 236
punctuation, excessive, 标点过多, 237~238
- Q**
- Qualifier, 限定词, 110
Qualifying, 限定, 111
static member, 静态成员, 166
static method, 静态方法, 125~126
type, 类型
for method invocation, 方法调用的类型, 191
for reflective access, 反射访问的类型, 190
question-mark colon operator, 问号分号操作符

See conditional expression, 查看条件表达式

R

Random, 236

random number generation, 随机数发生器, 233~237

raw type, vs. generic, 原生类型与范型的对比, 213~216

real-time guarantee, 实时保证, 186

recursion, 递归, 101~103

infinite, 无穷的, 85~87

ReentrantLock, 187

ReentrantReadWriteLock, 187

refactoring, 重构

definite assignment and, 以及明确的赋值, 83

inner class and generic, 内部类与泛型, 220

reference equality, 引用相等, 146

reference identity comparison, 引用一致性比较, 70, 71

reflection, 反射, 189~192

advanced language feature and, 以及高级语言特性, 197

class detection with, 以及类型检测, 100

instantiating inner class and, 以及实例化内部类, 197

missing class and, 以及缺失的类, 97~100

reflexivity, 自反性, 211

regular expression, 正则表达式, 43~45, 141~143

remainder operator, 取余操作符, 6, 76, 150

remainder, 余数, 5~7

replacement string, 替换字符串, 45~47

result type, 结果类型

of compound assignment operator, 复合赋值操作符的结果类型, 22

of conditional expression, 条件表达式的结果类型, 20

right shift operator, 右移操作符, 60

unsigned, 无符号的, 60, 67

Runtime.addShutdownHook, 84

Runtime.runFinalizersOnExit, 84

S

safe language, 安全的语言, 99

SecureRandom, 236

Serializable, 202

Serializable.readResolve, 203

Serialization, 序列化, 201~203, 224~228

encapsulation and, 以及封装, 173

Set, 集合, 142

shadowing, 遮蔽, 169~171, 174, 193~195

in constructor, 构造器中的遮蔽, 181

definition of, 其定义, 181

generic and, 以及泛型, 217~220

shift distance, 移位距离, 60

Short.reverseBytes, 143

shuffling, 扰乱, 233~237

shutdown hook, 关闭钩子, 84

sign extension, 有符号扩展, 14, 15, 16, 17, 54

sign-bit, in floating-point representation, 符号位, 在浮点表示法中, 72

significand, in floating-point representation, 有效位, 在浮点表示法中, 72

silent casting, 悄悄的转型, 21~23, 54

silent loss of precision, 悄悄的精度丢失, 212

silent overflow, 悄悄的溢出, 10, 11, 57~59, 73, 151, 152~156

simple assignment operator, 简单赋值操作符, 22
operand of, 其操作数, 24

singleton, 单例, 201~203

slash character, 斜杠字符, 45

-source, 177

- SQL DECIMAL type, SQL DECIMAL类型, 9
StackOverflowError, 86, 102
statement label, 语句标签, 47~48
statement, abrupt completion of, 语句, 其意外结束, 78
static, 静态
 class, 类
 nested, type parameter and, 嵌套, 以及类型参数, 220
 vs. inner, 与内部类的对比, 223
dispatch, 分派, 110, 181
field, 域, 107~109
import, 导入, 169~171
member class, 成员类, 197
 vs. nonstatic, 与非静态的对比, 219
member, accessing, 成员, 访问, 166
method, 方法
 dynamic dispatch and, 以及动态分派, 110
 qualification of, 其限定, 110, 111
 resolution of, 其解析, 109~111, 125~126
nested class, 嵌套类
 type parameter and, 以及类型参数, 220
static analysis tool, 静态分析工具, 3, 50
stream, output, flushing, 流, 输出, 刷新, 197~199
StrictMath.signum, 143
string concatenation operator, 字符串连接操作符, 26, 28, 30
 precedence of, 其优先级, 31
string conversion, 字符串转换, 28
string literal, 字符串字面常量
 in comment, 在注释中, 41~43
 escape sequence in, 其中的转义序列, 33, 44
 Unicode escape and, 以及Unicode转义字符, 32
String.indexOf, 41
String.replace, 46, 47
String.replaceAll, 44, 46
String.valueOf, 27, 28
StringBuffer.append, 28
String, 字符串
 concatenation of, 其连接, 25~27
 conversion of, 其转换, 27~29
 interned, 内存限定, 30, 146
 StringTokenizer, 142
subprocess, stream and, 子进程, 以及流, 199~201
subsumption, 包含, 160
super keyword, super关键字, 159
superclass, 超类
 constructor invocation, 构造器调用, 222
 first common, 首个公共超类, 99
swapping variable value, 交换变量值, 17~19
switch statement, switch语句, 50
symmetric, 对称的, 211
synchronization, 同步, 185~188
synchronized block, reentrant lock and, 块, 以及可重入锁, 187
synchronized method, behavior of, 同步化的方法, 及其行为, 184
System.err, 198
System.exit, 84, 85
System.halt, 85
System.out, 198
System.out.flush, 199
System.out.print, 26
System.out.print(char), 198
System.out.print(String), 198
System.out.println, 80, 84, 199

- System.out.write(int)**, 198
System.runFinalizersOnExit, 84
systematic bias, 系统偏差, 236
- T**
- target typing**, 目标确定类型, 10, 11
this construct, this结构, 194
this keyword, this关键字, 124, 223
Thread(Runnable), 194
Thread.interrupted, 203~205
Thread.isInterrupted, 204
Thread.join, 187
Thread.run, 184
Thread.sleep, 186
Thread.start, 184
Thread.stop, 93
Thread.suspend, 93
Thread, 线程
- interrupted status of**, 其中断状态, 204
 - state change of**, 其状态变化, 203~205
 - Thread.run vs. Thread.start, Thread.run与Thread.start的对比**, 183~184
 - Throwable, catching, Throwable, 捕获**, 80
 - throws clause, constructor declaration and, throws子句, 以及构造器声明**, 85
 - Timer**, 186
 - tool, for detecting bug**, 工具, 用于侦测缺陷, 3, 50
 - topology-preserving object graph transformation, 保留拓扑结构的对象图, 转换**, 146
 - toString**, 28, 170
 - total ordering, Comparators and, 全序, 以及Comparators**, 155
 - transitivity, 传递性**, 211
 - of Comparator, Comparator的传递性**, 155
 - traversal, preorder, 遍历, 前序**, 103
- truncating integer division operator, 截尾整数整除操作符**, 6
- try clause, try子句**
 - checked exception in, 其中受检查的异常**, 87~89
 - nesting, 嵌套**, 88
 - System.exit and, 以及System.exit**, 84
 - try-finally construct, try-finally结构**, 77~78
 - flow of control in, 其中的控制流**, 78
 - interaction with System.exit, 与System.exit的相互作用**, 83~85
 - recursion in, 其中的递归**, 102
 - reentrant lock and, 以及可重入锁**, 187
 - static initializer and, 以及静态初始器**, 81
 - two's-complement binary arithmetic, 2的二进制补码算术**, 16, 59, 151
 - asymmetry in, 其中的不对称性**, 71~73
- type parameter, 类型参数**
 - naming, 命名**, 164, 220
 - static nested class and, 以及静态嵌套类**, 220
- type token, 类型标记**, 216
- type variable declaration, 类型变量声明**, 166
- type, 类型**
 - conversion of, 其转换**, 15~17
 - See also narrowing primitive conversion, 查看窄化原生类型转换**
 - See widening primitive conversion, 查看拓宽原生类型转换**
 - byte to char, byte到char**, 17, 40
 - byte to int, byte到int**, 54
 - byte to String, byte到String**, 41
 - char to int, char 到 int**, 40, 49~52
 - char[] to String, char[] 到 String**, 29
 - double to String, double到String**, 8
 - float to int, float到int**, 73~75

int to float, int到float, 75
 long to double, long到double, 75
 long to float, long到float, 75
 loss of precision in, 其中的精度丢失, 75
 to String, 到String, 27~29, 65~66
 erasure of, 类型的消除, 95, 213~216
 precedence of, 类型的优先级, 164
 raw, vs. generic, 原生类型与泛型的对比, 213~216
 typesafe enum pattern, 类型安全的枚举模式, 179

U

ulp, 61~63
 unary negation operator, 一元取反操作符, 14, 72, 210
 unboxing, 解包装, 70, 152
 unchecked cast, 不受检查的转型, 95
 unchecked exception, 不受检查的异常, 78, 119
 underflow, 下溢, 58
 Unicode character, Unicode字符, 21
 char as, 作为char, 28
 Unicode escape, Unicode转义字符, 31
 in comment, 在注释中的, 33~35, 35~37
 compiler and, 以及编译器, 32, 36
 vs. escape sequence, 与转义序列的对比, 31~33
 Windows file name and, 以及Windows文件名称, 34
 unsafe language, 不安全的语言, 99
 unsigned right shift operator, 无符号右移操作符, 60, 67
 UnsupportedEncodingException, 41

V

value comparison, 值比较, 70, 71
 value equality, 值相等, 146

varargs, 可变参数, 169, 213
 variable, 变量
 multiple assignment of, 其多重赋值, 19, 55~56
 name, precedence of, 其名字和优先级, 164
 naming, 命名, 11~12, 164
 swapping value, 交换值, 17~19
 verification, 校验, 99
 virtual machine, 虚拟机
 exception checking and, 以及异常检查, 96
 exiting, 退出, 84

W

wait, 187
 white space, 空格, ii, vi, 4, 104, 208, 270
 parsing and, 以及解析, 36
 for grouping, 用于分组, 35
 widening primitive conversion, 拓宽原生类型转换, 10, 14, 16, 26, 51, 68
 loss of precision in, 其中的精度丢失, 74, 211~212
 wildcard type, 通配符类型, 215
 Windows file name, Windows文件名称, Unicode escape and, 以及转义字符, 34

X

-xlint:fallthrough, 50

Z

zero-extension, 零扩展, 16

参 考 文 献

- [Adelson99] Adelson, E. H. "Lightness perception and lightness illusions." In M. S. Gazzaniga (Ed.), *The New Cognitive Neurosciences*, MIT Press, 2nd ed., 1999: 339–351. ISBN: 0262071959. Also available as http://web.mit.edu/persci/people/adelson/pub_pdfs/gazzan.pdf
- [Albert99] Albert, Marc K., and Donald D. Hoffman, "The generic-viewpoint assumption and illusory contours." In *Perception*, Vol. 29 (1999): 303–312.
- [Bradley77] Bradley, D. R., and H. M. Petry. "Organizational determinants of subjective contour: the subjective Necker cube." In *American Journal of Psychology*, Vol. 90 (June 1977): 253–262.
- [Boute92] *Autoboxing*. Sun Microsystems. 2004.
<http://java.sun.com/j2se/5.0/docs/guide/language/autoboxing.html>
- [Boxing] Boute, Raymond. "The Euclidean definition of the functions div and mod." In *ACM Transactions on Programming Languages and Systems*, Vol. 14, No. 2 (April 1992): 127–144.

- [Bracha04] Bracha, Gilad. *Generics in the Java Programming Language*. 2004. <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>
- [Bug] *Java Bug Database*. Sun Microsystems, 1994–2005. <http://bugs.sun.com/bugdatabase/index.jsp>
- [Eclipse] *Eclipse Downloads*. The Eclipse Foundation, 2002–2005. <http://www.eclipse.org/downloads/index.php>
- [EJ] Bloch, Joshua. *Effective Java™ Programming Language Guide*. Addison-Wesley, 2001. ISBN: 0201310058.
- [Features-1.4] *J2SE 1.4.2 Summary of New Features and Enhancements*. Sun Microsystems, 2002. <http://java.sun.com/j2se/1.4.2/docs/relnotes/features.html>
- [Features-5.0] *New Features and Enhancements J2SE 5.0*. Sun Microsystems, 2004. <http://java.sun.com/j2se/5.0/docs/relnotes/features.html>
- [Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN: 0201633612.
- [Gregory79] Gregory, Richard L., and Priscilla Heard. “Border locking and the Café Wall illusion.” In *Perception*, Vol. 8 (1979): 365–380.
- [Hovemeyer04] Hovemeyer, David, and William Pugh. *FindBugs—Find Bugs in Java Programs*. 2004–2005. <http://findbugs.sourceforge.net>
- [IEEE-754] *IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers. IEEE 754-1985 (R1990). 1990.
- [ISO-8859-1] ISO/IEC JTC 1/SC 2/WG 3. *ISO 8859, 8-bit Single-Byte Coded Graphic Character Sets—Part 1: Latin Alphabet No. 1*. ISO 8859-1:1987. 1987.

- [ISO-C] *Programming Languages—C*. International Organization for Standardization. ISO/IEC 9899:1999. 1999.
- [Java-5.0] *Java Programming Language Enhancements in JDK 5*. Sun Microsystems, 2004.
<http://java.sun.com/j2se/5.0/docs/guide/language>
- [Java-API] *Java 2 Platform Standard Edition 5.0 API Specification*. Sun Microsystems, 2004. <http://java.sun.com/j2se/5.0/docs/api>
- [JDK-5.0] *Download Java 2 Platform Standard Edition 5.0*. Sun Microsystems, 2004. <http://java.sun.com/j2se/5.0/download.jsp>
- [Jikes] *Jikes Home*. Open Source Technology Group, 1997–2005.
<http://jikes.sourceforge.net>
- [JLS] Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha.
The Java™ Language Specification, Third Edition. Addison-Wesley, 2005. ISBN: 0321246780. Also available as
<http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>
- [JLS2] Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha.
The Java™ Language Specification, Second Edition. Addison-Wesley, 2000. ISBN: 0201310082.
- [JVMS] Lindholm, Tim, and Frank Yellin. *The Java™ Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.
ISBN: 0201432943.
- [Kitaoka01] Kitaoka, A., B. Pinna, and G. Brelstaff. “New variations of spiral illusions.” In *Perception*, Vol. 30 (2001): 637–646.
- [Kitaoka02] Kitaoka, Akiyoshi. *Trick Eyes*, Kanzen, Tokyo, 2002.
ISBN: 4901782118.
- [Kitaoka03] Kitaoka, Akiyoshi. *Trick Eyes 2*, Kanzen, Tokyo, 2003.
ISBN: 4901782169.

- [Kitaoka03b] Kitaoka, Akiyoshi, and Hiroshi Ashida. "Phenomenal characteristics of the peripheral drift illusion." In *Vision (Japan)*, Vol. 15, No. 4 (2003): 261–262. Also available as <http://www.psy.ritsumei.ac.jp/~akitaoka/PDrift.pdf>
- [Kitaoka05] Kitaoka, Akiyoshi. *Trick Eyes*, Barnes and Noble Publishing, October 2005.
- [Knuth98] Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN: 0201896842.
- [Liskov87] Liskov, B. "Data abstraction and hierarchy." In *Addendum to the Proceedings of OOPSLA '87 and SIGPLAN Notices*, Vol. 23, No. 5: 17–34, May 1988.
- [Logvinenko99] Logvinenko, Alexander D. "Lightness induction revisited." In *Perception*, Vol. 28 (1999): 803–816.
- [MaryBlog] Smaragdis, Mary. Weblog: *MaryMaryQuiteContrary*. 2004–2005. <http://blogs.sun.com/roller/page/mary>
- [Modula-3] Nelson, Greg (ed.). *Systems Programming with Modula-3*. Prentice Hall, 1991. ISBN: 0135904641.
- [Ouchi77] Ouchi, Hajime. *Japanese Optical and Geometrical Art*. Dover Publications, 1977. ISBN: 048623553X.
- [Schrauf97] Schrauf, M., B. Lingelbach, and E. R. Wist. "The scintillating grid illusion." In *Vision Research*, Vol. 37 (1997): 1033–1038.
- [Shepard90] Shepard, Roger N., *Mind Sights: Original Visual Illusions, Ambiguities, and Other Anomalies, with a Commentary on the Play of Mind in Perception and Art*. W.H. Freeman and Co, 1990. ISBN: 0716721341.

- [Spehar00] Spehar, B. "Degraded illusory contour formation with non-uniform inducers in Kanizsa configurations: the role of contrast polarity." In *Vision Research*, Vol. 40, No. 19 (September 2000): 2653–2659.
- [ThreadStop] *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?* Sun Microsystems, 1999. <http://java.sun.com/j2se/5.0/docs/guide/misc/threadPrimitiveDeprecation.html>
- [Todorovic97] Todorovic, D. "Lightness and junctions." In *Perception*, Vol. 26 (1997): 379–394.
- [van Tuijl75] van Tuijl, H.F. "A new visual illusion: neonlike color spreading and complementary color induction between subjective contours." In *Acta Psychologica*, Vol. 39 (1975): 441–445.
- [Turing36] Turing, A. "On Computable Numbers, with an Application to the Entscheidungsproblem." In *Proceedings of the London Mathematical Society*, Series 2, Vol. 42, 1936; reprinted in M. David (ed.), *The Undecidable*, Dover Publications, 2004.
- [Watanabe92] Watanabe, T., and P. Cavanagh. "Depth capture and transparency of regions bounded by illusory, chromatic, and texture contours." In *Vision Research*, Vol. 32 (1992): 527–532.
- [White79] White, M. "A new effect of pattern on perceived lightness." In *Perception*, Vol. 8 (1979): 413–416.
- [Zavagno99] Zavagno, D. "Some new luminance-gradient effects." In *Perception*, Vol. 28 (1999): 835–838.

Java Puzzlers
Traps, Pitfalls, and Corner Cases

Java 解惑

“每一种编程语言都有其怪癖的行为。这本生动的书通过趣味十足和发人深省的编程谜题揭示了 Java 编程语言的奇异之处。”

—— Guy Steele, Sun 院士,
《Java 语言规范》的合著者

“我笑了，我叫起来了，我钦佩地举起了双手。”

—— Tim Peierls, Prior Artisans 公司总裁,
JSR 166 专家组成员

你认为自己了解 Java 多少？你是个爱琢磨的代码侦探吗？你是否曾经花费数天时间去追踪一个由 Java 或其类库的陷阱和缺陷而导致的 bug？你喜欢智力测验吗？本书正好适合你！

Bloch 和 Gafter 继承了 *Effective Java* 一书的传统，深入研究了 Java 编程语言及其核心类库的细微之处。本书特写了 95 个噩梦般的谜题，中间穿插着许多有趣的视觉幻象，寓教于乐。任何具备 Java 知识的人都可以理解这些谜题，但甚至是最老练的程序员也会觉得它们具有挑战性。

多数的谜题都是些小程序，其行为诡秘，不可貌相。你能指出它们会做什么吗？根据这些谜题用到的特性以及各题的详细解惑方案，我们把它们松散地分成了几个部分。这些解惑方案超越了对程序行为的简单解释，向你展示了如何一劳永逸地避免底层的陷阱与缺陷。在本书的后部有一个方便的陷阱和缺陷目录，可供以后进行参考。

一旦你解决了这些谜题，那些曾经愚弄过最具经验的 Java 程序员的隐晦且有违直觉的语言行为，将再也无法把你骗过。



Joshua Bloch 是 Google 公司的首席工程师，以及获得过 Jolt 大奖的 *Effective Java* 一书的作者。他之前是 Sun 公司的杰出工程师和 Transarc 公司的资深系统设计师。Bloch 曾是 Java 语言开发团队的核心，领导了大量的 Java 平台特性的设计与实现工作，包括 JDK 5.0 中的语言改进和获奖的 Java 集合框架。他拥有卡耐基-梅隆大学计算机科学的博士学位。



Neal Gafter 是 Google 公司的软件工程师和 Java 专家。他曾是 Sun 公司的资深工程师，领导了 Java 编译器的开发工作，并且实现了 Java 1.4 版至 5.0 版的许多语言特性。Gafter 曾经是 C++ 标准委员会的成员，他在 Sun、Microtec Research 和 Texas Instruments 公司时领导了 C 和 C++ 编译器的开发工作。他拥有罗彻斯特大学计算机科学的博士学位。

本书相关信息请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)88593802

反馈/投稿/推荐信箱：contact@turingbook.com

上架建议 程序设计 /Java

ISBN 7-115-14241-6

9 787115 142412 >



www.PearsonEd.com



ISBN7-115-14241-6/TP·5118

定价：39.00 元

人民邮电出版社网址 www.ptpress.com.cn