HZ BOOKS

Sun公司核心技术丛书

Effective Java中文版第2版

Effective Java Second Edition



（美）Joshua Bloch 著
杨春花 俞黎敏 译

“我很希望10年前就拥有这本书。可能有人认为我不需要任何Java方面的书籍，但是我需要这本书。”

Java之父James Gosling

Sun microsystems

机械工业出版社
China Machine Press

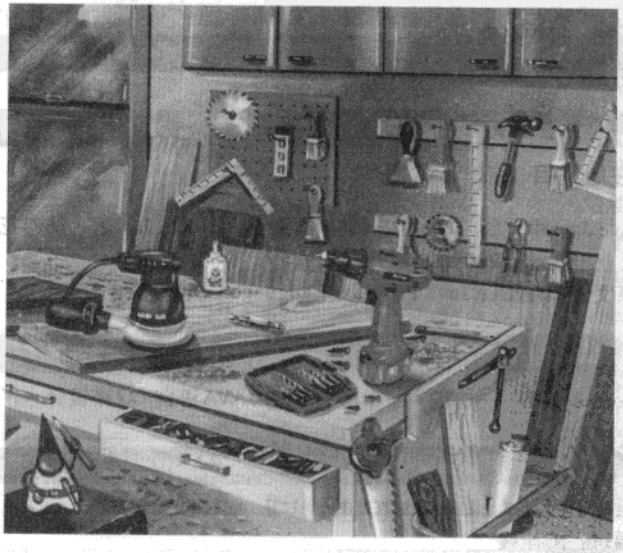
Sun 公司核心技术丛书

TP312Ja
164(2)
12

Effective Java中文版 第2版

Effective Java Second Edition

(美) Joshua Bloch 著
杨春花 俞黎敏 译



机械工业出版社
China Machine Press

本书介绍了在Java编程中78条极具实用价值的经验规则，这些经验规则涵盖了大多数开发人员每天所面临的问题的解决方案。通过对Java平台设计专家所使用的技术的全面描述，揭示了应该做什么，不应该做什么才能产生清晰、健壮和高效的代码。第2版反映了Java 5中最重要的变化，并删去了过时的内容。

本书中的每条规则都以简短、独立的小文章形式出现，并通过示例代码加以进一步说明。本书内容全面，结构清晰，讲解详细。可作为技术人员的参考用书。

Authorized translation from the English language edition entitled Effective Java Second Edition by Joshua Bloch, published by Pearson Education, Inc, publishing as Addison Wesley, Copyright © 2008 by Sun Microsystems, Inc. ISBN 978-0-321-35668-0

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

Chinese simplified language edition published by China Machine Press.

Copyright © 2009 by China Machine Press.

本书中文简体字版由美国Pearson Education培生教育出版集团授权机械工业出版社独家出版。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。

本书封面贴有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。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2008-2445

图书在版编目（CIP）数据

Effective Java中文版　第2版 /（美）布洛克（Bloch, J.）著；杨春花，俞黎敏译. —北京：机械工业出版社，2009.1

书名原文：Effective Java Program Language Guide, 2E

ISBN 978-7-111-25583-3

Ⅰ. E…　Ⅱ①布…　②杨…　③俞…　Ⅲ. JAVA语言—程序设计　Ⅳ. TP312

中国版本图书馆CIP数据核字（2008）第178021号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：陈佳媛

北京牛山世兴印刷厂印刷

2009 年1月第2版第1次印刷

186mm × 240mm · 19印张

标准书号：ISBN 978-7-111-25583-3

定价：52.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

译者序

Java从诞生到日趋完善，经过了不断的发展壮大，目前全世界拥有了成千上万的Java开发人员。如何编写出更清晰、更正确、更健壮且更易于重用的代码，是大家所追求的目标之一。作为经典Jolt获奖作品的新版书，它已经进行了彻底的更新，涵盖了自第1版之后所引入的Java SE 5和Java SE 6的新特性。作者探索了新的设计模式和语言习惯用法，介绍了如何充分利用从泛型到枚举、从注解到自动装箱的各种特性。本书的作者Joshua Bloch曾经是Sun公司的杰出工程师，带领团队设计和实现过无数的Java平台特性，包括JDK 5.0语言增强版和获奖的Java Collections Framework。他也是Jolt奖的获得者，现在担任Google公司的首席Java架构师。他为我们带来了共78条程序员必备的经验法则：针对你每天都会遇到的编程问题提出了最有效、最实用的解决方案。

书中的每一章都包含几个“条目”，以简洁的形式呈现，自成独立的短文，它们提出了具体的建议、对于Java平台精妙之处的独到见解，并提供优秀的代码范例。每个条目的综合描述和解释都阐明了应该怎么做、不应该怎么做，以及为什么。通过贯穿全书透彻的技术剖析与完整的示例代码，仔细研读并加以理解与实践，必定会从中受益匪浅。书中介绍的示例代码清晰易懂，也可以作为日常工作的参考指南。

适合人群

本书不是针对初学者的，读者至少需要熟悉Java程序设计语言。如果你连equals()、toString()、hashCode()都还不了解的话，建议先去看些优秀的Java入门书籍之后再来阅读本书。如果你现在已经在Java开发方面有了一定的经验，而且想更加深入地了解Java编程语言，成为一名更优秀、更高效的Java开发人员，那么，建议你用心地研读本书。

内容形式

本书分为11章共78个条目，涵盖了Java 5.0／6.0的种种技术要点。与第1版相比，本书删除了“C语言结构的替代”一章，增加了Java 5所引入的“泛型”、“枚举和注解”各一章。数量上从57个条目发展到了78个，不仅增加了23个条目，并对原来的所有资料都进行了全面的修改，删去了一些已经过时的条目。但是，各章节没有严格的前后顺序关系，你可以随意选

择感兴趣的章节进行阅读。当然，如果你想马上知道第2版究竟有哪些变化，可以参阅附录中第2版与第1版详细的对照情况。

本书重点讲述了Java 5所引入的全新的泛型、枚举、注解、自动装箱、for-each循环、可变参数、并发机制，还包括对象、类、类库、方法和序列化这些经典主题的全新技术和最佳实践，如何避免Java编程语言中常被误解的细微之处：陷阱和缺陷，并重点关注Java语言本身和最基本的类库：java.lang、java.util，以及一些扩展：java.util.concurrent和java.io等等。

章节简介

第2章阐述何时以及如何创建对象，何时以及如何避免创建对象，如何确保它们能够被适时地销毁，以及如何管理销毁之前必须进行的所有清除动作。

第3章阐述对于所有对象都通用的方法，你会从中获知对equals、hashCode、toString、clone和finalize相当深入的分析，从而避免今后在这些问题上再次犯错。

第4章阐述作为Java程序设计语言的核心以及Java语言的基本抽象单元（类和接口），在使用上的一些指导原则，帮助你更好地利用这些元素，设计出更加有用、健壮和灵活的类和接口。

第5和第6章中分别阐述在Java 1.5发行版本中新增加的泛型（Generic）以及枚举和注解的最佳实践，教你如何最大限度地享有这些优势，又能使整个过程尽可能地简单化。

第7章讨论方法设计的几个方面：如何处理参数和返回值，如何设计方法签名，如何为方法编写文档。从而在可用性、健壮性和灵活性上有进一步的提升。

第8章主要讨论Java语言的具体细节，讨论了局部变量的处理、控制结构、类库的使用、各种数据类型的用法，以及两种不是由语言本身提供的机制（reflection和native method，反射机制和本地方法）的用法。并讨论了优化和命名惯例。

第9章阐述如何充分发挥异常的优点，可以提高程序的可读性、可靠性和可维护性，以及减少使用不当所带来的负面影响。并提供了一些关于有效使用异常的指导原则。

第10章阐述如何帮助你编写出清晰、正确、文档组织良好的并发程序。

第11章阐述序列化方面的技术，并且有一项值得特别提及的特性，就是序列化代理（serialization proxy）模式，它可以帮助你避免对象序列化的许多缺陷。

举个例子，就序列化技术来讲，HTTP会话状态为什么可以被缓存？RMI的异常为什么可以从服务器端传递到客户端呢？GUI组件为什么可以被发送、保存和恢复呢？是因为它们实现了Serializable接口吗？如果超类没有提供一个可访问的无参构造器，它的子类可以被序列化

吗？当一个实例采用默认的序列化形式，并且给某些域标记为transient，那么当实例反序列化回来后，这些标志为transient域的值各是些什么呢？……这些问题如果你现在不能马上回答，或者不能很确定，没有关系，仔细阅读本书，你会对它们有更深入与透彻的理解。

技术范围

虽然本书是讨论更深层次的Java开发技术，讲述的内容深入，涉及面又相当广泛，但是它并没有涉及图形用户界面编程、企业级API以及移动设备方面的技术，不过在各个章节与条目中会不时地讨论到其他相关的类库。

这是一本分享经验与指引你避免走弯路的经典著作，针对如何编写高效、设计优良的程序提出了最实用、最权威的指导方针，是Java开发人员案头上的一本不可或缺的参考书。

本书由我组织进行翻译，第1章到第8章由杨春花负责，我负责前言、附录以及第9章到第11章的翻译，并负责本书所有章节的全面审校。参与翻译和审校的还有：荣浩、邱庆举、万国辉、陆志平、姜法有、王琳、林仪明、凌家亮、李勇、师文丽、刘传飞、王建旭、程旭文、罗兴、翟育明、黄华，在此深表感谢。

虽然我们在翻译过程中竭力追求信、达、雅，但限于自身水平，也许仍有不足，还望各位读者不吝指正。关于本书的翻译和翻译时采用的术语表以及相关的技术讨论大家可以访问我的博客http://blog.csdn.net/YuLimin，也可以发邮件到YuLimin @ 163.com与我交流。

在这里，我要感谢在翻译过程中一起讨论并帮助我的朋友们，他们是：崔毅，郑晖，左轻侯，郭晓刚，满江红开放技术研究组织创始人曹晓钢，Spring中文站创始人杨戈（Yanger），SpringSide创始人肖桦（江南白衣）和来自宝岛台湾的李日贵（jini）、林康司（koji）、林信良（caterpillar），还有责任编辑陈佳媛也为本书出版做了大量工作，在此再次深表感谢。

快乐分享，实践出真知，最后，祝大家能够像我一样在阅读中享受本书带来的乐趣！

Read a bit and take it out, then come back read some more.

俞黎敏

2008年11月

序

如果有一个同事这样对你说，“我的配偶今天晚上在家里制造了一顿不同寻常的晚餐，你愿意来参加吗？”（Spouse of me.this night today manufactures the unusual meal in a home. You will join?）这时候你脑子里可能会浮现起三件事情：第一，满脑子的疑惑；第二，英语肯定不是这位同事的母语；第三，同事是在邀请你参加他的家庭晚宴。

如果你曾经学习过第二种语言，并且尝试过在课堂之外使用这种语言，你就该知道有三件事情是必须掌握的：这门语言的结构如何（语法），如何命名你想谈论的事物（词汇），以及如何以惯用和高效的方式来表达日常的事物（用法）。在课堂上大多只涉及前面两点，当你使出浑身解数想让对方明白你的意思时，常常会发现当地人对你的表述忍俊不禁。

程序设计语言也是如此。你需要理解语言的核心：它是面向算法的，还是面向函数的，或者是面向对象的？你需要知道词汇表：标准类库提供了哪些数据结构、操作和功能（Facility）？你还需要熟悉如何用习惯和高效的方式来构建代码。关于程序设计语言的书籍通常只是涉及前面两点，或者只是蜻蜓点水般地介绍一下用法。也许是因为前面两点比较容易编写。语法和词汇是语言本身固有的特性，但是，用法则反映了使用这门语言的群体的特征。

例如，Java程序设计语言是一门支持单继承的面向对象程序设计语言，在每个方法的内部，它也支持命令式的（面向语句的，Statement-Oriented）编码风格。Java类库提供了对图形显示、网络、分布式计算和安全性的支持。但是，如何把这门语言以最佳的方式运用到实践中呢？

还有一点：程序与口语中的句子以及大多数书籍和杂志都不同，它会随着时间的推移而发生变化。仅仅编写出能够有效地工作并且能够被别人理解的代码往往是不够的，我们还必须把代码组织成易于修改的形式。针对某个任务可能会有10种不同的编码方法，而在这10种方法中，有7种方法是笨拙的、低效的或者是难以理解的。而在剩下的3种编码方法中，哪一种会是最接近该任务的下一年度发行版本的代码呢？

目前有大量的书籍可以供你学习Java程序设计语言的语法，包括《The Java Programming Language》[Arnold05]（作者Arnold、Gosling和Holmes），以及《The Java Language Specification》

[JLS]（作者Gosling、Joy和Bracha）。同样，与Java程序设计语言相关的类库和API的书籍也不少。

本书解决了你的第三种需求：习惯和高效的用法。作者Joshua Bloch在Sun公司多年来一直从事Java语言的扩展、实现和使用的工作；他还大量地阅读了其他人的代码，包括我的代码。他在本书中提出了许多很好的建议，他系统地把这些建议组织起来，旨在告诉读者如何更好地构造代码以便它们能工作得更好，也便于其他人能够理解这些代码，便于将来对代码进行修改和改善的时候不至于那么头疼。甚至，你的程序也会因此而变得更加令人愉悦、更加优美和雅致。

Guy L. Steele Jr.
Burlington, Massachusetts
2001年4月

前 言

自从我于2001年写了本书的第1版之后，Java平台又发生了很多变化，是该出第2版的时候了。Java 5中最为重要的变化是增加了泛型、枚举类型、注解、自动装箱和for-each循环。其次是增加了新的并发类库：java.util.concurrent。我和Gilad Bracha一起，有幸带领团队设计了最新的语言特性。我还有幸参加了设计和开发并发类库的团队，这个团队由Doug Lea领导。

Java平台中另一个大的变化在于广泛采用了现代的IDE（Integrated Development Environment），例如Eclipse、IntelliJ IDEA和NetBeans，以及静态分析工具的IDE，如FindBugs。虽然我还未参与到这部分工作，但已经从中受益匪浅，并且很清楚它们对Java开发体验所带来的影响。

2004年，我离开Sun公司到了Google公司工作，但在过去的4年中，我仍然继续参与Java平台的开发，在Google公司和JCP（Java Community Process）的大力帮助下，继续并发和集合API的开发。我还有幸利用Java平台去开发供Google内部使用的类库。现在我了解了作为一名用户的感受。

我在2001年编写第1版的时候，主要目的是与读者分享我的经验，便于让大家能够避免我所走过的弯路，使大家更容易成功。新版仍然大量采用来自Java平台类库的真实范例。

第1版所带来的反应远远超出了我最大的预期。我在收集所有新的资料以使本书保持最新时，尽可能地保持了资料的真实。毫无疑问，本书的篇幅肯定会增加，从57个条目发展到了78个。我不仅增加了23个条目，并且修改了原来的所有资料，并删去了一些已经过时的条目。在附录中，你可以看到本书中的内容与第1版的内容的对照情况。

在第1版的前言中我说过：Java程序设计语言和它的类库非常有益于代码质量和效率的提高，并且使得用Java进行编码成为一种乐趣。Java 5和6发行版本中的变化是好事，也使得Java平台日趋完善。现在这个平台比2001年的要大得多，也复杂得多，但是一旦掌握了使用

新特性的模式和习惯用法，它们就会使你的程序变得更完美，使你的工作变得更轻松。我希望第2版能够体现出我对Java平台持续的热情，并将这种热情传递给你，帮助你更加高效和愉快地使用Java平台及其新的特性。

Joshua Bloch
San Jose, California
2008年4月

致 谢

我要感谢本书第1版的读者给予本书如此热情的好评，感谢他们将书中的理念铭记在心，感谢他们让我知道该书给他们以及他们的工作带来了怎样积极的影响。我感谢许多教授在教学中采用了本书，感谢许多开发团队应用了本书。

我要感谢Addison-Wesley的整个团队，感谢他们的诚恳、专业、耐心，以及压力之下所体现出来的从容。编辑Greg Doench自始至终保持镇定自若：他是一名优秀的编辑，同时也是一位完美的绅士。产品经理Julie Nahil具备了产品经理应该具备的一切：勤奋、敏捷、训练有素，且待人和气。编审Barbara Wood一丝不苟，富有鉴赏能力。

我有幸再一次得到了所能想到的最佳审核团队的支持，我真诚地感谢他们中的每一位。核心团队负责审核每一个章节，他们包括：Lexi Baugher、Cindy Bloch、Beth Bottos、Joe Bowbeer、Brian Goetz、Tim Halloran、Brian Kernighan、Rob Konigsberg、Tim Peierls、Bill Pugh、Yoshiki Shibata、Peter Stout、Peter Weinberger以及Frank Yellin。其他审核人员包括：Pablo Bellver、Dan Bloch、Dan Bornstein、Kevin Bourrillion、Martin Buchholz、Joe Darcy、Neal Gafter、Laurence Gonsalves、Aaron Greenhouse、Barry Hayes、Peter Jones、Angelika Langer、Doug Lea、Bob Lee、Jeremy Manson、Tom May、Mike McCloskey、Andriy Tereshchenko以及Paul Tyma。这些审核人员再次提出了大量的建议，使本书得到了极大的改善，也让我避免了诸多尴尬。剩下的任何错误都是我自己的责任。

我要特别感谢Doug Lea和Tim Peierls，他们成了书中许多理念的倡导者。Doug和Tim为本书毫不吝惜地奉献了他们的时间和学识。

我要感谢我在Google公司的经理Prabha Krishna，感谢她持续不断的支持和鼓励。

最后，我要感谢我的妻子Cindy Bloch，她鼓励我写作，阅读了初稿中的每个条目，用Framemaker帮我排版，为我编写索引，在我写作的时候一直对我十分宽容。

目　录

译者序

序

前言

致谢

第1章　引言……1

第2章　创建和销毁对象……4

第1条：考虑用静态工厂方法代替构造器……4

第2条：遇到多个构造器参数时要考虑用构建器……9

第3条：用私有构造器或者枚举类型强化Singleton属性……14

第4条：通过私有构造器强化不可实例化的能力……16

第5条：避免创建不必要的对象……17

第6条：消除过期的对象引用……21

第7条：避免使用终结方法……24

第3章　对于所有对象都通用的方法……28

第8条：覆盖equals时请遵守通用约定……28

第9条：覆盖equals时总要覆盖hashCode……39

第10条：始终要覆盖toString……44

第11条：谨慎地覆盖clone……46

第12条：考虑实现Comparable接口……53

第4章　类和接口……58

第13条：使类和成员的可访问性最小化……58

第14条：在公有类中使用访问方法而非公有域……62

第15条：使可变性最小化……64

第16条：复合优先于继承……71

第17条：要么为继承而设计，并提供文档说明，要么就禁止继承……76

第18条：接口优于抽象类……82

第19条：接口只用于定义类型……86

第20条：类层次优于标签类……88

第21条：用函数对象表示策略……91

第22条：优先考虑静态成员类……94

第5章　泛型……97

第23条：请不要在新代码中使用原生态类型……97

第24条：消除非受检警告……103

第25条：列表优先于数组……105

第26条：优先考虑泛型……109

第27条：优先考虑泛型方法……113

第28条：利用有限制通配符来提升API的灵活性……117

第29条：优先考虑类型安全的异构容器……123

第6章　枚举和注解……128

第30条：用enum代替int常量……128

第31条：用实例域代替序数……137

第32条：用EnumSet代替位域……138

第33条：用EnumMap代替序数索引……140

第34条：用接口模拟可伸缩的枚举……144
第35条：注解优先于命名模式……147
第36条：坚持使用Override注解……152
第37条：用标记接口定义类型……154
第7章 方法……156
第38条：检查参数的有效性……156
第39条：必要时进行保护性拷贝……159
第40条：谨慎设计方法签名……163
第41条：慎用重载……165
第42条：慎用可变参数……170
第43条：返回零长度的数组或者集合，而不是null……174
第44条：为所有导出的API元素编写文档注释……176
第8章 通用程序设计……181
第45条：将局部变量的作用域最小化……181
第46条：for-each循环优先于传统的for循环……184
第47条：了解和使用类库……187
第48条：如果需要精确的答案，请避免使用float和double……190
第49条：基本类型优先于装箱基本类型……192
第50条：如果其他类型更适合，则尽量避免使用字符串……195
第51条：当心字符串连接的性能……198
第52条：通过接口引用对象……199
第53条：接口优先于反射机制……201
第54条：谨慎地使用本地方法……204
第55条：谨慎地进行优化……205
第56条：遵守普遍接受的命名惯例……208
第9章 异常……211
第57条：只针对异常的情况才使用异常……211
第58条：对可恢复的情况使用受检异常，对编程错误使用运行时异常……214
第59条：避免不必要地使用受检的异常……216
第60条：优先使用标准的异常……218
第61条：抛出与抽象相对应的异常……220
第62条：每个方法抛出的异常都要有文档……222
第63条：在细节消息中包含能捕获失败的信息……224
第64条：努力使失败保持原子性……226
第65条：不要忽略异常……228
第10章 并发……229
第66条：同步访问共享的可变数据……229
第67条：避免过度同步……234
第68条：executor和task优先于线程……239
第69条：并发工具优先于wait和notify……241
第70条：线程安全性的文档化……246
第71条：慎用延迟初始化……249
第72条：不要依赖于线程调度器……252
第73条：避免使用线程组……254
第11章 序列化……255
第74条：谨慎地实现Serializable接口……255
第75条：考虑使用自定义的序列化形式……260
第76条：保护性地编写readObject方法……266
第77条：对于实例控制，枚举类型优先于readResolve……271
第78条：考虑用序列化代理代替序列化实例……275
附录 第1版与第2版条目对照……278
中英文术语对照……280
参考文献……283

第1章

引　言

本书的目标是帮助读者最有效地使用Java程序设计语言及其基本类库：java.lang、java.util，在某种程度上还包括java.util.concurrent和java.io。本书也会不时地讨论到其他的类库，但是没有涉及图形用户界面编程、企业级API以及移动设备相关的类库。

本书共包含78个条目，每个条目讨论一条规则。这些规则反映了最有经验的优秀程序员在实践中常用的一些有益做法。本书以一种比较自由的方式将这些条目组织成10章，每一章都涉及软件设计的一个主要方面。本书并不一定要按部就班地从头读到尾，因为每个条目都有一定程度的独立性。这些条目相互之间交叉引用，因此你可以很容易地在书中找到自己需要的内容。

Java 5（发行版本1.5）中增加了许多新特性。本书中大多数条目都以一定的方式用到了这些特性。表1-1列出了这些特性所在的主要章节或条目。

表1-1　新增特性所在章节或条目

特　性	所在章节或条目	特　性	所在章节或条目
泛型	第5章	自动装箱	第40、49条
枚举	第30～34条	varargs	第42条
注解	第35～37条	静态导入	第19条
for-each循环	第46条	java.util.concurrent	第68、69条

大多数条目都通过程序示例进行说明。本书一个突出的特点是，包含了许多代码示例，这些例子说明了许多设计模式（Design Pattern）和习惯用法（Idiom）。当需要参考设计模式领域的标准参考书[Gamma 95]时，还为这些设计模式和习惯用法提供了交叉引用。

许多条目都包含有一个或多个应该在实践中避免的程序示例。像这样的例子，有时候也叫做“反模式（Antipattern）”，在注释中清楚地标注为“//Never do this!”。对于每种情况，条目中都解释了为什么此例不好，并提出了另外的解决方法。

本书并不是针对初学者的：本书假设读者已经熟悉Java程序设计语言。如果你还没有做到，请考虑先参阅一本很好的Java入门书籍[Arnold05, Sestoft05]。本书的目标是适用于任何具有实际Java工作经验的程序员，对于高级程序员，也应该能够提供一些发人深思的东西。

本书中大多数规则都源于少数几条基本的原则。清晰性和简洁性最为重要：模块的用户永远也不应该被模块的行为所迷惑（那样就不清晰了）；模块要尽可能小，但又不能太小［本书中使用的术语模块（Module），是指任何可重用的软件组件，从单个方法，到包含多个包的复杂系统，都可以是一个模块］。代码应该被重用，而不是被拷贝。模块之间的依赖性应该尽可能地降到最小。错误应该尽早被检测出来，最好是在编译时刻。

虽然本书中的规则不会百分之百地适用于任何时刻和任何场合，但是，它们确实体现了绝大多数情况下的最佳程序设计实践。你不应该盲目地遵从这些规则，但是，你应该只在偶尔的情况下，有了充分理由之后才去打破这些规则。同大多数学科一样，学习编程艺术首先要学会基本的规则，然后才能知道什么时候可以打破这些规则。

本书大部分内容都不是讨论性能的，而是关心如何编写出清晰、正确、可用、健壮、灵活和可维护的程序来。如果你能够做到这一点的话，那么要想获得所需要的性能往往就相对比较简单了（见第55条）。有些条目确实谈到了性能问题，甚至有的还提供了性能指标。但是，在提及这些指标的时候，也会出现“在我的机器上”这样的话，所以，你最好把这些指标视同近似值。

有必要提及的是，我的机器是一台过时的家用电脑，主机为2.2 GHz双核AMD Opteron 170，2G内存，在Microsoft Windows XP Professional SP2操作系统平台上运行Sun 1.6_05发行版本的Java SE Development Kit（JDK）。这个JDK有两台虚拟机：Java HotSpot Client和Server VM。性能指标是在Server VM上测量的。

讨论Java程序设计语言及其类库特性的时候，有时候必须要指明具体的发行版本。为了简单起见，本书使用了*工程版本号*（engineering version number），而不是正式的发行名称。表1-2列出了发行名称与工程版本号之间的对应关系。

表1-2 Java的工程版本号

正式发行名称	工程版本号
JDK 1.1.x / JRE 1.1.x	1.1
Java 2 Platform, Standard Edition, v 1.2	1.2
Java 2 Platform, Standard Edition, v 1.3	1.3
Java 2 Platform, Standard Edition, v 1.4	1.4
Java 2 Platform, Standard Edition, v 5.0	1.5
Java Platform, Standard Edition 6	1.6

尽管这些例子都很完整，但是它们注重可读性更甚于注重完整性。它们直接使用了java.util和java.io包中的类。为了编译这些示例程序，可能需要在程序中加上一行或者多行这样的import语句：

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

其他代码示例中也有类似被省略的情况。但是，在本书的Web站点：http://java.sun.com/docs/books/effective，提供了每个示例的完整版本，你可以直接编译和运行这些示例。

本书采用的大部分技术术语都与《*The Java Language Specification, Third Edition*》[JLS]⊖相同。有一些术语则值得特别提出来。Java语言支持四种类型：接口（interface）、类（class）、数组（array）和基本类型（primitive）。前三种类型通常被称为引用类型（reference type），类实例和数组是对象（object），而基本类型的值则不是对象。类的成员（member）由它的域（field）、方法（method）、成员类（member class）和成员接口（member interface）组成。方法的签名（signature）由它的名称和所有参数类型组成；签名不包括它的返回类型。

本书也使用了一些与《The Java Language Specification》不同的术语。与《The Java Language Specification》不同的是，本书用术语“继承（inheritance）”作为“子类化（subclassing）”的同义词。本书不再使用“接口继承”这种说法，而是简单地说，一个类实现（implement）了一个接口，或者一个接口扩展（extend）了另一个接口。为了描述“在没有指定访问级别的情况下所使用的访问级别”，本书使用了描述性的术语“包级私有（package-private）”，而不是如[JLS, 6.6.1]中所使用的技术性术语“缺省访问（default access）级别”。

本书也使用了一些在《The Java Language Specification》中没有定义的技术术语。术语“导出的API（exported API）”，或者简单地说API，是指类、接口、构造器（constructor）、成员和序列化形式（serialized form），程序员通过它们可以访问类、接口或者包。（术语API是Application Programming Interface的简写，这里之所以使用API而不用接口（interface），是为了不与Java语言中的interface类型相混淆）。使用API编写程序的程序员被称为该API的用户（user），在类的实现中使用了API的类被称为该API的客户（client）。

类、接口、构造器、成员以及序列化形式被统称为API元素（API element）。导出的API由所有可在定义该API的包之外访问的API元素组成。任何客户端都可以使用这些API元素，而API的创建者则负责支持这些API元素。Javadoc工具类在默认操作模式下也正是为这些元素生成文档，这绝非偶然。不严格地讲，一个包的导出的API是由该包中的每个公有（public）类或者接口中所有公有的或者受保护的（protected）成员和构造器组成。

⊖ 该书影印版《Java语言规范》由机械工业出版社引进出版，书号是：7-111-18839。——编辑注

第2章

创建和销毁对象

本章的主题是创建和销毁对象：何时以及如何创建对象，何时以及如何避免创建对象，如何确保它们能够适时地销毁，以及如何管理对象销毁之前必须进行的各种清理动作。

第1条：考虑用静态工厂方法代替构造器

对于类而言，为了让客户端获取它自身的一个实例，最常用的方法就是提供一个公有的构造器。还有一种方法，也应该在每个程序员的工具箱中占有一席之地。类可以提供一个公有的静态工厂方法（static factory method），它只是一个返回类的实例的静态方法。下面是一个来自Boolean（基本类型boolean的包装类）的简单示例。这个方法将boolean基本类型值转换成了一个Boolean对象引用：

```
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

注意，静态工厂方法与设计模式[Gamma95, p.107]中的工厂方法模式不同。本条目中所指的静态工厂方法并不直接对应于设计模式中的工厂方法。

类可以通过静态工厂方法来提供它的客户端，而不是通过构造器。提供静态工厂方法而不是公有的构造器，这样做具有几大优势。

静态工厂方法与构造器不同的第一大优势在于，它们有名称。如果构造器的参数本身没有确切地描述正被返回的对象，那么具有适当名称的静态工厂会更容易使用，产生的客户端代码也更易于阅读。例如，构造器BigInteger（int, int, Random）返回的BigInteger可能为素数，如果用名为BigInteger.probablePrime的静态工厂方法来表示，显然更为清楚。（1.4的发行版本中最终增加了这个方法。）

一个类只能有一个带有指定签名的构造器。编程人员通常知道如何避开这一限制：通过提

供两个构造器，它们的参数列表只在参数类型的顺序上有所不同。实际上这并不是个好主意。面对这样的API，用户永远也记不住该用哪个构造器，结果常常会调用错误的构造器。并且，读到使用了这些构造器的代码时，如果没有参考类的文档，往往不知所云。

由于静态工厂方法有名称，所以它们不受上述的限制。当一个类需要多个带有相同签名的构造器时，就用静态工厂方法代替构造器，并且慎重地选择名称以便突出它们之间的区别。

静态工厂方法与构造器不同的第二大优势在于，不必在每次调用它们的时候都创建一个新对象。这使得不可变类（见第15条）可以使用预先构建好的实例，或者将构建好的实例缓存起来，进行重复利用，从而避免创建不必要的重复对象。Boolean.valueOf(boolean)方法说明了这项技术：它从来不创建对象。这种方法类似于Flyweight模式[Gamma95，p.195]。如果程序经常请求创建相同的对象，并且创建对象的代价很高，则这项技术可以极大地提升性能。

静态工厂方法能够为重复的调用返回相同对象，这样有助于类总能严格控制在某个时刻哪些实例应该存在。这种类被称作*实例受控的类*（instance-controlled）。编写实例受控的类有几个原因。实例受控使得类可以确保它是一个Singleton（见第3条）或者是不可实例化的（见第4条）。它还使得不可变的类（见第15条）可以确保不会存在两个相等的实例，即当且仅当a==b的时候才有a.equals(b)为ture。如果类保证了这一点，它的客户端就可以使用==操作符来代替equals（Object）方法，这样可以提升性能。枚举（enum）类型（见第30条）保证了这一点。

静态工厂方法与构造器不同的第三大优势在于，它们可以返回原返回类型的任何子类型的对象。这样我们在选择返回对象的类时就有了更大的灵活性。

这种灵活性的一种应用是，API可以返回对象，同时又不会使对象的类变成公有的。以这种方式隐藏实现类会使API变得非常简洁。这项技术适用于基于接口的框架（interface-based framework，见第18条），因为在这种框架中，接口为静态工厂方法提供了自然返回类型。接口不能有静态方法，因此按照惯例，接口Type的静态工厂方法被放在一个名为Types的不可实例化的类（见第4条）中。

例如，Java Collections Framework的集合接口有32个便利实现，分别提供了不可修改的集合、同步集合等等。几乎所有这些实现都通过静态工厂方法在一个不可实例化的类（java.util.Collections）中导出。所有返回对象的类都是非公有的。

现在的Collections Framework API比导出32个独立公有类的那种实现方式要小得多，每种便利实现都对应一个类。这不仅仅是指API数量上的减少，也是概念意义上的减少。用户知道，被返回的对象是由相关的接口精确指定的，所以他们不需要阅读有关的类文档。使用这种静态工厂方法时，甚至要求客户端通过接口来引用被返回的对象，而不是通过它的实现类来引用被返回的对象，这是一种良好的习惯（见第52条）。

公有的静态工厂方法所返回的对象的类不仅可以是非公有的，而且该类还可以随着每次调用而发生变化，这取决于静态工厂方法的参数值。只要是已声明的返回类型的子类型，都是允许的。为了提升软件的可维护性和性能，返回对象的类也可能随着发行版本的不同而不同。

发行版本1.5中引入的类java.util.EnumSet（见第32条）没有公有构造器，只有静态工厂方法。它们返回两种实现类之一，具体则取决于底层枚举类型的大小：如果它的元素有64个或者更少，就像大多数枚举类型一样，静态工厂方法就会返回一个RegalarEumSet实例，用单个long进行支持；如果枚举类型有65个或者更多元素，工厂就返回JumboEnumSet实例，用long数组进行支持。

这两个实现类的存在对于客户端来说是不可见的。如果RegularEnumSet不能再给小的枚举类型提供性能优势，就可能从未来的发行版本中将它删除，不会造成不良的影响。同样地，如果事实证明对性能有好处，也可能在未来的发行版本中添加第三甚至第四个EnumSet实现。客户端永远不知道也不关心他们从工厂方法中得到的对象的类；他们只关心它是EnumSet的某个子类即可。

静态工厂方法返回的对象所属的类，在编写包含该静态工厂方法的类时可以不必存在。这种灵活的静态工厂方法构成了服务提供者框架（Service Provider Framework）的基础，例如JDBC（Java数据库连接，Java Database Connectivity）API。服务提供者框架是指这样一个系统：多个服务提供者实现一个服务，系统为服务提供者的客户端提供多个实现，并把他们从多个实现中解耦出来。

服务提供者框架中有三个重要的组件：服务接口（Service Interface），这是提供者实现的；提供者注册API（Provider Registration API），这是系统用来注册实现，让客户端访问它们的；服务访问API（Service Access API），是客户端用来获取服务的实例的。服务访问API一般允许但是不要求客户端指定某种选择提供者的条件。如果没有这样的规定，API就会返回默认实现的一个实例。服务访问API是“灵活的静态工厂”，它构成了服务提供者框架的基础。

服务提供者框架的第四个组件是可选的：服务提供者接口（Service Provider Interface），这些提供者负责创建其服务实现的实例。如果没有服务提供者接口，实现就按照类名称注册，并通过反射方式进行实例化（见第53条）。对于JDBC来说，Connection就是它的服务接口，DriverManager.registerDriver是提供者注册API，DriverManager.get Connection是服务访问API，Driver就是服务提供者接口。

服务提供者框架模式有着无数种变体。例如，服务访问API可以利用适配器（Adapter）模式[Gamma95，p.139]，返回比提供者需要的更丰富的服务接口。下面是一个简单的实现，包含一个服务提供者接口和一个默认提供者：

```
// Service provider framework sketch

// Service interface
public interface Service {
    ... // Service-specific methods go here
}

// Service provider interface
public interface Provider {
    Service newService();
}

// Noninstantiable class for service registration and access
public class Services {
    private Services() { }  // Prevents instantiation (Item 4)

    // Maps service names to services
    private static final Map<String, Provider> providers =
        new ConcurrentHashMap<String, Provider>();
    public static final String DEFAULT_PROVIDER_NAME = "<def>";

    // Provider registration API
    public static void registerDefaultProvider(Provider p) {
        registerProvider(DEFAULT_PROVIDER_NAME, p);
    }
    public static void registerProvider(String name, Provider p){
        providers.put(name, p);
    }

    // Service access API
    public static Service newInstance() {
        return newInstance(DEFAULT_PROVIDER_NAME);
    }
    public static Service newInstance(String name) {
        Provider p = providers.get(name);
        if (p == null)
            throw new IllegalArgumentException(
                "No provider registered with name: " + name);
        return p.newService();
    }
}
```

静态工厂方法的第四大优势在于，在创建参数化类型实例的时候，它们使代码变得更加简洁。遗憾的是，在调用参数化类的构造器时，即使类型参数很明显，也必须指明。这通常要求你接连两次提供类型参数：

```
Map<String, List<String>> m =
    new HashMap<String, List<String>>();
```

随着类型参数变得越来越长，越来越复杂，这一冗长的说明也很快变得痛苦起来。但是有了静态工厂方法，编译器就可以替你找到类型参数。这被称作类型推导（type inference）。例如，假设HashMap提供了这个静态工厂：

```
public static <K, V> HashMap<K, V> newInstance() {
    return new HashMap<K, V>();
}
```

你就可以用下面这句简洁的代码代替上面这段繁琐的声明：

```
Map<String, List<String>> m = HashMap.newInstance();
```

总有一天，Java将能够在构造器调用以及方法调用中执行这种类型推导，但到发行版本1.6为止暂时还无法这么做。

遗憾的是，到发行版本1.6为止，标准的集合实现如HashMap并没有工厂方法，但是可以把这些方法放在你自己的工具类中。更重要的是，可以把这样的静态工厂放在你自己的参数化的类中。

静态工厂方法的主要缺点在于，类如果不含公有的或者受保护的构造器，就不能被子类化。对于公有的静态工厂所返回的非公有类，也同样如此。例如，要想将Collections Framework中的任何方便的实现类子类化，这是不可能的。但是这样也许会因祸得福，因为它鼓励程序员使用复合（composition），而不是继承（见第16条）。

静态工厂方法的第二个缺点在于，它们与其他的静态方法实际上没有任何区别。在API文档中，它们没有像构造器那样在API文档中明确标识出来，因此，对于提供了静态工厂方法而不是构造器的类来说，要想查明如何实例化一个类，这是非常困难的。Javadoc工具总有一天会注意到静态工厂方法。同时，你通过在类或者接口注释中关注静态工厂，并遵守标准的命名习惯，也可以弥补这一劣势。下面是静态工厂方法的一些惯用名称：

- valueOf——不太严格地讲，该方法返回的实例与它的参数具有相同的值。这样的静态工厂方法实际上是类型转换方法。
- of——valueOf的一种更为简洁的替代，在EnumSet（见第32条）中使用并流行起来。
- getInstance——返回的实例是通过方法的参数来描述的，但是不能够说与参数具有同样的值。对于Singleton来说，该方法没有参数，并返回唯一的实例。
- newInstance——像getInstance一样，但newInstance能够确保返回的每个实例都与所有其他实例不同。
- get*Type*——像getInstance一样，但是在工厂方法处于不同的类中的时候使用。*Type*表示工厂方法所返回的对象类型。
- new*Type*——像newInstance一样，但是在工厂方法处于不同的类中的时候使用。*Type*表示工厂方法所返回的对象类型。

简而言之，静态工厂方法和公有构造器都各有用处，我们需要理解它们各自的长处。静态工厂通常更加合适，因此切忌第一反应就是提供公有的构造器，而不先考虑静态工厂。

第2条：遇到多个构造器参数时要考虑用构建器

静态工厂和构造器有个共同的局限性：它们都不能很好地扩展到大量的可选参数。考虑用一个类表示包装食品外面显示的营养成份标签。这些标签中有几个域是必需的：每份的含量、每罐的含量以及每份的卡路里，还有超过20个可选域：总脂肪量、饱和脂肪量、转化脂肪、胆固醇、钠等等。大多数产品在某几个可选域中都会有非零的值。

对于这样的类，应该用哪种构造器或者静态方法来编写呢？程序员一向习惯采用重叠构造器（telescoping constructor）模式，在这种模式下，你提供第一个只有必要参数的构造器，第二个构造器有一个可选参数，第三个有两个可选参数，依此类推，最后一个构造器包含所有可选参数。下面有个示例，为了简单起见，它只显示四个可选域：

```
// Telescoping constructor pattern - does not scale well!
public class NutritionFacts {
    private final int servingSize;   // (mL)             required
    private final int servings;      // (per container) required
    private final int calories;      //                 optional
    private final int fat;           // (g)             optional
    private final int sodium;        // (mg)            optional
    private final int carbohydrate;  // (g)             optional

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
    public NutritionFacts(int servingSize, int servings,
           int calories, int fat, int sodium, int carbohydrate) {
        this.servingSize  = servingSize;
        this.servings     = servings;
        this.calories     = calories;
        this.fat          = fat;
        this.sodium       = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

当你想要创建实例的时候，就利用参数列表最短的构造器，但该列表中包含了要设置的所有参数：

```
NutritionFacts cocaCola =
    new NutritionFacts(240, 8, 100, 0, 35, 27);
```

这个构造器调用通常需要许多你本不想设置的参数，但还是不得不为它们传递值。在这个例子中，我们给fat传递了一个值为0。如果“仅仅”是这6个参数，看起来还不算太糟，问题是随着参数数目的增加，它很快就失去了控制。

一句话：重叠构造器模式可行，但是当有许多参数的时候，客户端代码会很难编写，并且仍然较难以阅读。如果读者想知道那些值是什么意思，必须很仔细地数着这些参数来探个究竟。一长串类型相同的参数会导致一些微妙的错误。如果客户端不小心颠倒了其中两个参数的顺序，编译器也不会出错，但是程序在运行时会出现错误的行为。

遇到许多构造器参数的时候，还有第二种代替办法，即JavaBeans模式，在这种模式下，调用一个无参构造器来创建对象，然后调用setter方法来设置每个必要的参数，以及每个相关的可选参数：

```
// JavaBeans Pattern - allows inconsistency, mandates mutability
public class NutritionFacts {
    // Parameters initialized to default values (if any)
    private int servingSize  = -1;  // Required; no default value
    private int servings     = -1;  //      "     "     "       "
    private int calories     = 0;
    private int fat          = 0;
    private int sodium       = 0;
    private int carbohydrate = 0;

    public NutritionFacts() { }
    // Setters
    public void setServingSize(int val)  { servingSize = val; }
    public void setServings(int val)      { servings = val; }
    public void setCalories(int val)      { calories = val; }
    public void setFat(int val)           { fat = val; }
    public void setSodium(int val)        { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

这种模式弥补了重叠构造器模式的不足。说得明白一点，就是创建实例很容易，这样产生的代码读起来也很容易：

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

遗憾的是，JavaBeans模式自身有着很严重的缺点。因为构造过程被分到了几个调用中，在构造过程中**JavaBean**可能处于不一致的状态。类无法仅仅通过检验构造器参数的有效性来保证一致性。试图使用处于不一致状态的对象，将会导致失败，这种失败与包含错误的代码大相径庭，因此它调试起来十分困难。与此相关的另一点不足在于，**JavaBeans**模式阻止了把

类做成不可变的可能（见第15条），这就需要程序员付出额外的努力来确保它的线程安全。

当对象的构造完成，并且不允许在解冻之前使用时，通过手工“冻结”对象，可以弥补这些不足，但是这种方式十分笨拙，在实践中很少使用。此外，它甚至会在运行时导致错误，因为编译器无法确保程序员会在使用之前先在对象上调用freeze方法。

幸运的是，还有第三种替代方法，既能保证像重叠构造器模式那样的安全性，也能保证像JavaBeans模式那么好的的可读性。这就是Builder模式[Gamma95，p.97]的一种形式。不直接生成想要的对象，而是让客户端利用所有必要的参数调用构造器（或者静态工厂），得到一个builder对象。然后客户端在builder对象上调用类似于setter的方法，来设置每个相关的可选参数。最后，客户端调用无参的build方法来生成不可变的对象。这个builder是它构建的类的静态成员类（见第22条）。下面就是它的示例：

```
// Builder Pattern
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;

        // Optional parameters - initialized to default values
        private int calories      = 0;
        private int fat           = 0;
        private int carbohydrate  = 0;
        private int sodium        = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings    = servings;
        }

        public Builder calories(int val)
            { calories = val;      return this; }
        public Builder fat(int val)
            { fat = val;           return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = val;  return this; }
        public Builder sodium(int val)
            { sodium = val;        return this; }

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
```

```
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

注意NutritionFacts是不可变的，所有的默认参数值都单独放在一个地方。builder的setter方法返回builder本身，以便可以把调用链接起来。下面就是客户端代码：

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).
  calories(100).sodium(35).carbohydrate(27).build();
```

这样的客户端代码很容易编写，更为重要的是，易于阅读。**builder模式模拟了具名的可选参数**，就像Ada和Python中的一样。

builder像个构造器一样，可以对其参数强加约束条件。build方法可以检验这些约束条件。将参数从builder拷贝到对象中之后，并在对象域而不是builder域（见第39条）中对它们进行检验，这一点很重要。如果违反了任何约束条件，build方法就应该抛出IllegalStateException（见第60条）。异常的详细信息应该显示出违反了哪个约束条件（见第63条）。

对多个参数强加约束条件的另一种方法是，用多个setter方法对某个约束条件必须持有的所有参数进行检查。如果该约束条件没有得到满足，setter方法就会抛出IllegalArgumentException。这有个好处，就是一旦传递了无效的参数，立即就会发现约束条件失败，而不是等着调用build方法。

与构造器相比，builder的微略优势在于，builder可以有多个可变（varargs）参数。构造器就像方法一样，只能有一个可变参数。因为builder利用单独的方法来设置每个参数，你想要多少个可变参数，它们就可以有多少个，直到每个setter方法都有一个可变参数。

Builder模式十分灵活，可以利用单个builder构建多个对象。builder的参数可以在创建对象期间进行调整，也可以随着不同的对象而改变。builder可以自动填充某些域，例如每次创建对象时自动增加序列号。

设置了参数的builder生成了一个很好的抽象工厂（Abstract Factory）[Gamma95，p.87]。换句话说，客户端可以将这样一个builder传给方法，使该方法能够为客户端创建一个或者多个对象。要使用这种用法，需要有个类型来表示builder。如果使用的是发行版本1.5或者更新的版本，只要一个泛型（见第26条）就能满足所有的builder，无论它们在构建哪种类型的对象：

```
// A builder for objects of type T
public interface Builder<T> {
    public T build();
}
```

注意，可以声明NutritionFacts.Builder类来实现Builder<NutritionFacts>。

带有Builder实例的方法通常利用有限制的通配符类型（bounded wildcard type，见第28条）来约束构建器的类型参数。例如，下面就是构建每个节点的方法，它利用一个客户端提供的Builder实例来构建树：

```
Tree buildTree(Builder<? extends Node> nodeBuilder) { ... }
```

Java中传统的抽象工厂实现是Class对象，用newInstance方法充当build方法的一部分。这种用法隐含着许多问题。newInstance方法总是企图调用类的无参构造器，这个构造器甚至可能根本不存在。如果类没有可以访问的无参构造器，你也不会收到编译时错误。相反，客户端代码必须在运行时处理InstantiationException或者IllegalAccessException，这样既不雅观也不方便。newInstance方法还会传播由无参构造器抛出的任何异常，即使newInstance缺乏相应的throws子句。换句话说，**Class.newInstance**破坏了编译时的异常检查。上面讲过的Builder接口弥补了这些不足。

Builder模式的确也有它自身的不足。为了创建对象，必须先创建它的构建器。虽然创建构建器的开销在实践中可能不那么明显，但是在某些十分注重性能的情况下，可能就成问题了。Builder模式还比重叠构造器模式更加冗长，因此它只在有很多参数的时候才使用，比如4个或者更多个参数。但是记住，将来你可能需要添加参数。如果一开始就使用构造器或者静态工厂，等到类需要多个参数时才添加构建器，就会无法控制，那些过时的构造器或者静态工厂显得十分不协调。因此，通常最好一开始就使用构建器。

简而言之，如果类的构造器或者静态工厂中具有多个参数，设计这种类时，**Builder**模式就是种不错的选择，特别是当大多数参数都是可选的时候。与使用传统的重叠构造器模式相比，使用Builder模式的客户端代码将更易于阅读和编写，构建器也比JavaBeans更加安全。

第3条：用私有构造器或者枚举类型强化Singleton属性

Singleton指仅仅被实例化一次的类[Gamma95, p. 127]。Singleton通常被用来代表那些本质上唯一的系统组件，比如窗口管理器或者文件系统。使类成为**Singleton**会使它的客户端测试变得十分困难，因为无法给Singleton替换模拟实现，除非它实现一个充当其类型的接口。

在Java 1.5发行版本之前，实现Singleton有两种方法。这两种方法都要把构造器保持为私有的，并导出公有的静态成员，以便允许客户端能够访问该类的唯一实例。在第一种方法中，公有静态成员是个final域：

```
// Singleton with public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }

    public void leaveTheBuilding() { ... }
}
```

私有构造器仅被调用一次，用来实例化公有的静态final域Elvis.INSTANCE。由于缺少公有的或者受保护的构造器，所以保证了Elvis的全局唯一性：一旦Elvis类被实例化，只会存在一个Elvis实例，不多也不少。客户端的任何行为都不会改变这一点，但要提醒一点：享有特权的客户端可以借助AccessibleObject.setAccessible方法，通过反射机制（见第53条）调用私有构造器。如果需要抵御这种攻击，可以修改构造器，让它在被要求创建第二个实例的时候抛出异常。

在实现Singleton的第二种方法中，公有的成员是个静态工厂方法：

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public static Elvis getInstance() { return INSTANCE; }

    public void leaveTheBuilding() { ... }
}
```

对于静态方法Elvis.getInstance的所有调用，都会返回同一个对象引用，所以，永远不会创建其他的Elvis实例（上述提醒依然适用）。

公有域方法的主要好处在于，组成类的成员的声明很清楚地表明了这个类是一个Singleton：公有的静态域是final的，所以该域将总是包含相同的对象引用。公有域方法在性能上不再有任何优势：现代的JVM（Java虚拟机，Java Virtual Machine）实现几乎都能够将静态工厂方法的调用内联化。

工厂方法的优势之一在于，它提供了灵活性：在不改变其API的前提下，我们可以改变该类是否应该为Singleton的想法。工厂方法返回该类的唯一实例，但是，它可以很容易被修改，比如改成为每个调用该方法的线程返回一个唯一的实例。第二个优势与泛型（见第27条）有关。这些优势之间通常都不相关，public域（public-field）的方法比较简单。

为了使利用这其中一种方法实现的Singleton类变成是可序列化的（Serializable）（见第11章），仅仅在声明中加上“implements Serializable”是不够的。为了维护并保证Singleton，必须声明所有实例域都是瞬时（transient）的，并提供一个readResolve方法（见第77条）。否则，每次反序列化一个序列化的实例时，都会创建一个新的实例，比如说，在我们的例子中，会导致“假冒的Elvis”。为了防止这种情况，要在Elvis类中加入下面这个readResolve方法：

```
// readResolve method to preserve singleton property
private Object readResolve() {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;
}
```

从Java 1.5发行版本起，实现Singleton还有第三种方法。只需编写一个包含单个元素的枚举类型：

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;

    public void leaveTheBuilding() { ... }
}
```

这种方法在功能上与公有域方法相近，但是它更加简洁，无偿地提供了序列化机制，绝对防止多次实例化，即使是在面对复杂的序列化或者反射攻击的时候。虽然这种方法还没有广泛采用，但是单元素的枚举类型已经成为实现**Singleton**的最佳方法。

第4条：通过私有构造器强化不可实例化的能力

有时候，你可能需要编写只包含静态方法和静态域的类。这些类的名声很不好，因为有些人在面向对象的语言中滥用这样的类来编写过程化的程序。尽管如此，它们也确实有它们特有的用处。我们可以利用这种类，以java.lang.Math或者java.util.Arrays的方式，把基本类型的值或者数组类型上的相关方法组织起来。我们也可以通过java.util.Collections的方式，把实现特定接口的对象上的静态方法（包括工厂方法，见第1条）组织起来。最后，还可以利用这种类把final类上的方法组织起来，以取代扩展该类的做法。

这样的工具类（utility class）不希望被实例化，实例对它没有任何意义。然而，在缺少显式构造器的情况下，编译器会自动提供一个公有的、无参的缺省构造器（default constructor）。对于用户而言，这个构造器与其他的构造器没有任何区别。在已发行的API中常常可以看到一些被无意识地实例化的类。

企图通过将类做成抽象类来强制该类不可被实例化，这是行不通的。该类可以被子类化，并且该子类也可以被实例化。这样做甚至会误导用户，以为这种类是专门为了继承而设计的（见第17条）。然而，有一些简单的习惯用法可以确保类不可被实例化。由于只有当类不包含显式的构造器时，编译器才会生成缺省的构造器，因此我们只要让这个类包含私有构造器，它就不能被实例化了：

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    }
    ... // Remainder omitted
}
```

由于显式的构造器是私有的，所以不可以在该类的外部访问它。AssertionError不是必需的，但是它可以避免不小心在类的内部调用构造器。它保证该类在任何情况下都不会被实例化。这种习惯用法有点违背直觉，好像构造器就是专门设计成不能被调用一样。因此，明智的做法就是在代码中增加一条注释，如上所示。

这种习惯用法也有副作用，它使得一个类不能被子类化。所有的构造器都必须显式或隐式地调用超类（superclass）构造器，在这种情形下，子类就没有可访问的超类构造器可调用了。

第5条：避免创建不必要的对象

一般来说，最好能重用对象而不是在每次需要的时候就创建一个相同功能的新对象。重用方式既快速，又流行。如果对象是不可变的（immutable）（见第15条），它就始终可以被重用。

作为一个极端的反面例子，考虑下面的语句：

```
String s = new String("stringette");  // DON'T DO THIS!
```

该语句每次被执行的时候都创建一个新的String实例，但是这些创建对象的动作全都是不必要的。传递给String构造器的参数（“stringette”）本身就是一个String实例，功能方面等同于构造器创建的所有对象。如果这种用法是在一个循环中，或者是在一个被频繁调用的方法中，就会创建出成千上万不必要的String实例。

改进后的版本如下所示：

```
String s = "stringette";
```

这个版本只用了一个String实例，而不是每次执行的时候都创建一个新的实例。而且，它可以保证，对于所有在同一台虚拟机中运行的代码，只要它们包含相同的字符串字面常量，该对象就会被重用[JLS, 3.10.5]。

对于同时提供了静态工厂方法（见第1条）和构造器的不可变类，通常可以使用静态工厂方法而不是构造器，以避免创建不必要的对象。例如，静态工厂方法Boolean.valueOf (String)几乎总是优先于构造器Boolean(String)。构造器在每次被调用的时候都会创建一个新的对象，而静态工厂方法则从来不要求这样做，实际上也不会这样做。

除了重用不可变的对象之外，也可以重用那些已知不会被修改的可变对象。下面是一个比较微妙、也比较常见的反面例子，其中涉及可变的Date对象，它们的值一旦计算出来之后就不再变化。这个类建立了一个模型：其中有一个人，并有一个isBabyBoomer方法，用来检验这个人是否为一个“baby boomer（生育高峰期出生的小孩）”，换句话说，就是检验这个人是否出生于1946年至1964年期间。

```
public class Person {
    private final Date birthDate;

    // Other fields, methods, and constructor omitted
    // DON'T DO THIS!
    public boolean isBabyBoomer() {
        // Unnecessary allocation of expensive object
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
```

```
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
               birthDate.compareTo(boomEnd)   <  0;
    }
}
```

isBabyBoomer每次被调用的时候，都会新建一个Calendar、一个TimeZone和两个Date实例，这是不必要的。下面的版本用一个静态的初始化器（initializer），避免了这种效率低下的情况：

```
class Person {
    private final Date birthDate;
    // Other fields, methods, and constructor omitted

    /**
     * The starting and ending dates of the baby boom.
     */
    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }
    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0 &&
               birthDate.compareTo(BOOM_END)   <  0;
    }
}
```

改进后的Person类只在初始化的时候创建Calendar、TimeZone和Date实例一次，而不是在每次调用isBabyBoomer的时候都创建这些实例。如果isBabyBoomer方法被频繁地调用，这种方法将会显著地提高性能。在我的机器上，每调用一千万次，原来的版本需要32 000ms，而改进后的版本只需130ms，大约快了250倍。除了提高性能之外，代码的含义也更加清晰了。把boomStart和boomEnd从局部变量改为final静态域，这些日期显然是被作为常量对待，从而使得代码更易于理解。但是，这种优化带来的效果并不总是那么明显，因为Calendar实例的创建代价特别昂贵。

如果改进后的Person类被初始化了，它的isBabyBoomer方法却永远不会被调用，那就没有必要初始化BOOM_START和BOOM_END域。通过*延迟初始化*（**lazily initializing**）（见第71条），即把对这些域的初始化延迟到isBabyBoomer方法第一次被调用的时候进行，则有可能消除这些不必要的初始化工作，但是不建议这样做。正如延迟初始化中常见的情况一样，这样做会使方法的实现更加复杂，从而无法将性能显著提高到超过已经达到的水平（见第55条）。

在本条目前面的例子中，所讨论到的对象显然都是能够被重用的，因为它们被初始化之后不会再改变。其他有些情形则并不总是这么明显了。考虑适配器（**adapter**）的情形[Gamma95, p. 139]，有时也叫做视图（**view**）。适配器是指这样一个对象：它把功能委托给一个后备对象（backing object），从而为后备对象提供一个可以替代的接口。由于适配器除了后备对象之外，没有其他的状态信息，所以针对某个给定对象的特定适配器而言，它不需要创建多个适配器实例。

例如，Map接口的keySet方法返回该Map对象的Set视图，其中包含该Map中所有的键（key）。粗看起来，好像每次调用keySet都应该创建一个新的Set实例，但是，对于一个给定的Map对象，实际上每次调用keySet都返回同样的Set实例。虽然被返回的Set实例一般是可改变的，但是所有返回的对象在功能上是等同的：当其中一个返回对象发生变化的时候，所有其他的返回对象也要发生变化，因为它们是由同一个Map实例支撑的。虽然创建keySet视图对象的多个实例并无害处，却也是没有必要的。

在Java 1.5发行版本中，有一种创建多余对象的新方法，称作自动装箱（**autoboxing**），它允许程序员将基本类型和装箱基本类型（Boxed Primitive Type）混用，按需要自动装箱和拆箱。自动装箱使得基本类型和装箱基本类型之间的差别变得模糊起来，但是并没有完全消除。它们在语义上还有着微妙的差别，在性能上也有着比较明显的差别（见第49条）。考虑下面的程序，它计算所有int正值的总和。为此，程序必须使用long算法，因为int不够大，无法容纳所有int正值的总和：

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

这段程序算出的答案是正确的，但是比实际情况要更慢一些，只因为打错了一个字符。变量sum被声明成Long而不是long，意味着程序构造了大约2^{31}个多余的Long实例（大约每次往Long sum中增加long时构造一个实例）。将sum的声明从Long改成long，在我的机器上使运行时间从43秒减少到了6.8秒。结论很明显：要优先使用基本类型而不是装箱基本类型，要当心无意识的自动装箱。

不要错误地认为本条目所介绍的内容暗示着“创建对象的代价非常昂贵，我们应该要尽可能地避免创建对象”。相反，由于小对象的构造器只做很少量的显式工作，所以，小对象的创建和回收动作是非常廉价的，特别是在现代的JVM实现上更是如此。通过创建附加的对象，提升程序的清晰性、简洁性和功能性，这通常是件好事。

反之，通过维护自己的对象池（**object pool**）来避免创建对象并不是一种好的做法，除非池中的对象是非常重量级的。真正正确使用对象池的典型对象示例就是数据库连接池。建立数据库连接的代价是非常昂贵的，因此重用这些对象非常有意义。而且，数据库的许可可能限制你只能使用一定数量的连接。但是，一般而言，维护自己的对象池必定会把代码弄得很乱，同时增加内存占用（footprint），并且还会损害性能。现代的JVM实现具有高度优化的垃圾回收器，其性能很容易就会超过轻量级对象池的性能。

与本条目对应的是第39条中有关“保护性拷贝（**defensive copying**）”的内容。本条目提及“当你应该重用现有对象的时候，请不要创建新的对象”，而第39条则说“当你应该创建新对象的时候，请不要重用现有的对象”。注意，在提倡使用保护性拷贝的时候，因重用对象而付出的代价要远远大于因创建重复对象而付出的代价。必要时如果没能实施保护性拷贝，将会导致潜在的错误和安全漏洞；而不必要地创建对象则只会影响程序的风格和性能。

第6条：消除过期的对象引用

当你从手工管理内存的语言（比如C或C++）转换到具有垃圾回收功能的语言的时候，程序员的工作会变得更加容易，因为当你用完了对象之后，它们会被自动回收。当你第一次经历对象回收功能的时候，会觉得这简直有点不可思议。这很容易给你留下这样的印象，认为自己不再需要考虑内存管理的事情了。其实不然。

考虑下面这个简单的栈实现的例子：

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

这段程序（它的泛型版本请见第26条）中并没有很明显的错误。无论如何测试，它都会成功地通过每一项测试，但是这个程序中隐藏着一个问题。不严格地讲，这段程序有一个“内存泄漏”，随着垃圾回收器活动的增加，或者由于内存占用的不断增加，程序性能的降低会逐渐表现出来。在极端的情况下，这种内存泄漏会导致磁盘交换（Disk Paging），甚至导致程序失败（OutOfMemoryError错误），但是这种失败情形相对比较少见。

那么，程序中哪里发生了内存泄漏呢？如果一个栈先是增长，然后再收缩，那么，从栈中弹出来的对象将不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，它们也不会被回收。这是因为，栈内部维护着对这些对象的过期引用（obsolete reference）。所谓的过期引

用，是指永远也不会再被解除的引用。在本例中，凡是在elements数组的“活动部分（active portion）”之外的任何引用都是过期的。活动部分是指elements中下标小于size的那些元素。

在支持垃圾回收的语言中，内存泄漏是很隐蔽的（称这类内存泄漏为“*无意识的对象保持*（unintentional object retention）”更为恰当）。如果一个对象引用被无意识地保留起来了，那么，垃圾回收机制不仅不会处理这个对象，而且也不会处理被这个对象所引用的所有其他对象。即使只有少量的几个对象引用被无意识地保留下来，也会有许许多多的对象被排除在垃圾回收机制之外，从而对性能造成潜在的重大影响。

这类问题的修复方法很简单：一旦对象引用已经过期，只需清空这些引用即可。对于上述例子中的Stack类而言，只要一个单元被弹出栈，指向它的引用就过期了。pop方法的修订版本如下所示：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

清空过期引用的另一个好处是，如果它们以后又被错误地解除引用，程序就会立即抛出NullPointerException异常，而不是悄悄地错误运行下去。尽快地检测出程序中的错误总是有益的。

当程序员第一次被类似这样的问题困扰的时候，他们往往会过分小心：对于每一个对象引用，一旦程序不再用到它，就把它清空。其实这样做既没必要，也不是我们所期望的，因为这样做会把程序代码弄得很乱。*清空对象引用应该是一种例外，而不是一种规范行为*。消除过期引用最好的方法是让包含该引用的变量结束其生命周期。如果你是在最紧凑的作用域范围内定义每一个变量（见第45条），这种情形就会自然而然地发生。

那么，何时应该清空引用呢？Stack类的哪方面特性使它易于遭受内存泄漏的影响呢？简而言之，问题在于，Stack类*自己管理内存*（manage its own memory）。*存储池*（storage pool）包含了elements数组（对象引用单元，而不是对象本身）的元素。数组活动区域（同前面的定义）中的元素是*已分配的*（allocated），而数组其余部分的元素则是*自由的*（free）。但是垃圾回收器并不知道这一点；对于垃圾回收器而言，elements数组中的所有对象引用都同等有效。只有程序员知道数组的非活动部分是不重要的。程序员可以把这个情况告知垃圾回收器，做法很简单：一旦数组元素变成了非活动部分的一部分，程序员就手工清空这些数组元素。

*一般而言，只要类是自己管理内存，程序员就应该警惕内存泄漏问题。*一旦元素被释放掉，则该元素中包含的任何对象引用都应该被清空。

内存泄漏的另一个常见来源是缓存。一旦你把对象引用放到缓存中，它就很容易被遗忘掉，从而使得它不再有用之后很长一段时间内仍然留在缓存中。对于这个问题，有几种可能的解决方案。如果你正好要实现这样的缓存：只要在缓存之外存在对某个项的键的引用，该项就有意义，那么就可以用WeakHashMap代表缓存；当缓存中的项过期之后，它们就会自动被删除。记住只有当所要的缓存项的生命周期是由该键的外部引用而不是由值决定时，WeakHashMap才有用处。

更为常见的情形则是，“缓存项的生命周期是否有意义”并不是很容易确定，随着时间的推移，其中的项会变得越来越没有价值。在这种情况下，缓存应该时不时地清除掉没用的项。这项清除工作可以由一个后台线程（可能是Timer或者ScheduledThreadPoolExecutor）来完成，或者也可以在给缓存添加新条目的时候顺便进行清理。LinkedHashMap类利用它的removeEldestEntry方法可以很容易地实现后一种方案。对于更加复杂的缓存，必须直接使用java.lang.ref。

内存泄漏的第三个常见来源是监听器和其他回调。如果你实现了一个API，客户端在这个API中注册回调，却没有显式地取消注册，那么除非你采取某些动作，否则它们就会积聚。确保回调立即被当作垃圾回收的最佳方法是只保存它们的弱引用（weak reference），例如，只将它们保存成WeakHashMap中的键。

由于内存泄漏通常不会表现成明显的失败，所以它们可以在一个系统中存在很多年。往往只有通过仔细检查代码，或者借助于Heap剖析工具（Heap Profiler）才能发现内存泄漏问题。因此，如果能够在内存泄漏发生之前就知道如何预测此类问题，并阻止它们发生，那是最好不过的了。

第7条：避免使用终结方法

*终结方法（finalizer）通常是不可预测的，也是很危险的，一般情况下是不必要的。*使用终结方法会导致行为不稳定、降低性能，以及可移植性问题。当然，终结方法也有其可用之处，我们将在本条目的最后再做介绍；但是根据经验，应该避免使用终结方法。

C++的程序员被告知“不要把终结方法当作是C++中析构器（destructors）的对应物”。在C++中，析构器是回收一个对象所占用资源的常规方法，是构造器所必需的对应物。在Java中，当一个对象变得不可到达的时候，垃圾回收器会回收与该对象相关联的存储空间，并不需要程序员做专门的工作。C++的析构器也可以被用来回收其他的非内存资源。而在Java中，一般用try-finally块来完成类似的工作。

终结方法的缺点在于不能保证会被及时地执行[JLS, 12.6]。从一个对象变得不可到达开始，到它的终结方法被执行，所花费的这段时间是任意长的。这意味着，注重时间（time-critical）的任务不应该由终结方法来完成。例如，用终结方法来关闭已经打开的文件，这是严重错误，因为打开文件的描述符是一种很有限的资源。由于JVM会延迟执行终结方法，所以大量的文件会保留在打开状态，当一个程序再不能打开文件的时候，它可能会运行失败。

及时地执行终结方法正是垃圾回收算法的一个主要功能，这种算法在不同的JVM实现中会大相径庭。如果程序依赖于终结方法被执行的时间点，那么这个程序的行为在不同的JVM中运行的表现可能就会截然不同。一个程序在你测试用的JVM平台上运行得非常好，而在你最重要顾客的JVM平台上却根本无法运行，这是完全有可能的。

延迟终结过程并不只是一个理论问题。在很少见的情况下，为类提供终结方法，可能会随意地延迟其实例的回收过程。一位同事最近在调试一个长期运行的GUI应用程序的时候，该应用程序莫名其妙地出现OutOfMemoryError错误而死掉。分析表明，该应用程序死掉的时候，其终结方法队列中有数千个图形对象正在等待被终结和回收。遗憾的是，终结方法线程的优先级比该应用程序的其他线程的要低得多，所以，图形对象的终结速度达不到它们进入队列的速度。Java语言规范并不保证哪个线程将会执行终结方法，所以，除了不使用终结方法之外，并没有很轻便的办法能够避免这样的问题。

Java语言规范不仅不保证终结方法会被及时地执行，而且根本就不保证它们会被执行。当一个程序终止的时候，某些已经无法访问的对象上的终结方法却根本没有被执行，这是完全有可能的。结论是：*不应该依赖终结方法来更新重要的持久状态。*例如，依赖终结方法来释放共享资源（比如数据库）上的永久锁，很容易让整个分布式系统垮掉。

不要被System.gc和System.runFinalization这两个方法所诱惑，它们确实增加了终结方法

被执行的机会，但是它们并不保证终结方法一定会被执行。唯一声称保证终结方法被执行的方法是System.runFinalizersOnExit，以及它臭名昭著的孪生兄弟Runtime.runFinalizersOnExit。这两个方法都有致命的缺陷，已经被废弃了[ThreadStop]。

当你并不确定是否应该避免使用终结方法的时候，这里还有一种值得考虑的情形：如果未被捕获的异常在终结过程中被抛出来，那么这种异常可以被忽略，并且该对象的终结过程也会终止[JLS, 12.6]。未被捕获的异常会使对象处于破坏的状态（a corrupt state），如果另一个线程企图使用这种被破坏的对象，则可能发生任何不确定的行为。正常情况下，未被捕获的异常将会使线程终止，并打印出栈轨迹（Stack Trace），但是，如果异常发生在终结方法之中，则不会如此，甚至连警告都不会打印出来。

还有一点：使用终结方法有一个非常严重的（**Severe**）性能损失。在我的机器上，创建和销毁一个简单对象的时间大约为5.6ns。增加一个终结方法使时间增加到了2 400ns。换句话说，用终结方法创建和销毁对象慢了大约430倍。

那么，如果类的对象中封装的资源（例如文件或者线程）确实需要终止，应该怎么做才能不用编写终结方法呢？只需提供一个显式的终止方法，并要求该类的客户端在每个实例不再有用的时候调用这个方法。值得提及的一个细节是，该实例必须记录下自己是否已经被终止了：显式的终止方法必须在一个私有域中记录下“该对象已经不再有效”。如果这些方法是在对象已经终止之后被调用，其他的方法就必须检查这个域，并抛出IllegalStateException异常。

显式终止方法的典型例子是InputStream、OutputStream和java.sql.Connection上的close方法。另一个例子是java.util.Timer上的cancel方法，它执行必要的状态改变，使得与Timer实例相关联的该线程温和地终止自己。java.awt中的例子还包括Graphics.dispose和Window.dispose。这些方法通常由于性能不好而不被人们关注。一个相关的方法是Image.flush，它会释放所有与Image实例相关联的资源，但是该实例仍然处于可用的状态，如果有必要的话，会重新分配资源。

显式的终止方法通常与**try-finally**结构结合起来使用，以确保及时终止。在finally子句内部调用显式的终止方法，可以保证即使在使用对象的时候有异常抛出，该终止方法也会执行：

```
// try-finally block guarantees execution of termination method
Foo foo = new Foo(...);
try {
    // Do what must be done with foo
    ...
} finally {
    foo.terminate();  // Explicit termination method
}
```

那么终结方法有什么好处呢？它们有两种合法用途。第一种用途是，当对象的所有者忘记调用前面段落中建议的显式终止方法时，终结方法可以充当“安全网（safety net）”。虽然这

样做并不能保证终结方法会被及时地调用，但是在客户端无法通过调用显式的终止方法来正常结束操作的情况下（希望这种情形尽可能地少发生），迟一点释放关键资源总比永远不释放要好。但是如果终结方法发现资源还未被终止，则应该在日志中记录一条警告，因为这表示客户端代码中的一个Bug，应该得到修复。如果你正考虑编写这样的安全网终结方法，就要认真考虑清楚，这种额外的保护是否值得你付出这份额外的代价。

显式终止方法模式的示例中所示的四个类（FileInputStream、FileOutputStream、Timer和Connection），都具有终结方法，当它们的终止方法未能被调用的情况下，这些终结方法充当了安全网。

终结方法的第二种合理用途与对象的本地对等体（**native peer**）有关。本地对等体是一个本地对象（native object），普通对象通过本地方法（native method）委托给一个本地对象。因为本地对等体不是一个普通对象，所以垃圾回收器不会知道它，当它的Java对等体被回收的时候，它不会被回收。在本地对等体并不拥有关键资源的前提下，终结方法正是执行这项任务最合适的工具。如果本地对等体拥有必须被及时终止的资源，那么该类就应该具有一个显式的终止方法，如前所述。终止方法应该完成所有必要的工作以便释放关键的资源。终止方法可以是本地方法，或者它也可以调用本地方法。

值得注意的很重要一点是，“终结方法链（finalizer chaining）”并不会被自动执行。如果类（不是Object）有终结方法，并且子类覆盖了终结方法，子类的终结方法就必须手工调用超类的终结方法。你应该在一个try块中终结子类，并在相应的finally块中调用超类的终结方法。这样做可以保证：即使子类的终结过程抛出异常，超类的终结方法也会得到执行。反之亦然。代码示例如下。注意这个示例使用了Override注解（@Override），这是Java 1.5发行版本将它增加到Java平台中的。你现在可以不管Override注解，或者到第36条查阅一下它们表示什么意思：

```
// Manual finalizer chaining
@Override protected void finalize() throws Throwable {
    try {
        ... // Finalize subclass state
    } finally {
        super.finalize();
    }
}
```

如果子类实现者覆盖了超类的终结方法，但是忘了手工调用超类的终结方法（或者有意选择不调用超类的终结方法），那么超类的终结方法将永远也不会被调用到。要防范这样粗心大意或者恶意的子类是有可能的，代价就是为每个将被终结的对象创建一个附加的对象。不是把终结方法放在要求终结处理的类中，而是把终结方法放在一个匿名的类（见第22条）中，该匿名类的唯一用途就是终结它的外围实例（enclosing instance）。该匿名类的单个实例被称为终结方法守卫者（**finalizer guardian**），外围类的每个实例都会创建这样一个守卫者。外围实例在它的私有实例域中保存着一个对其终结方法守卫者的唯一引用，因此终结方法守卫者

与外围实例可以同时启动终结过程。当守卫者被终结的时候，它执行外围实例所期望的终结行为，就好像它的终结方法是外围对象上的一个方法一样：

```
// Finalizer Guardian idiom
public class Foo {
   // Sole purpose of this object is to finalize outer Foo object
   private final Object finalizerGuardian = new Object() {
      @Override protected void finalize() throws Throwable {
         ... // Finalize outer Foo object
      }
   };
   ...  // Remainder omitted
}
```

注意，公有类Foo并没有终结方法（除了它从Object中继承了一个无关紧要的之外），所以子类的终结方法是否调用super.finalize并不重要。对于每一个带有终结方法的非final公有类，都应该考虑使用这种方法。

总之，除非是作为安全网，或者是为了终止非关键的本地资源，否则请不要使用终结方法。在这些很少见的情况下，既然使用了终结方法，就要记住调用super.finalize。如果用终结方法作为安全网，要记得记录终结方法的非法用法。最后，如果需要把终结方法与公有的非final类关联起来，请考虑使用终结方法守卫者，以确保即使子类的终结方法未能调用super.finalize，该终结方法也会被执行。

第3章

对于所有对象都通用的方法

尽管Object是一个具体类，但是设计它主要是为了扩展。它所有的非final方法（equals、hashCode、toString、clone和finalize）都有明确的通用约定（**general contract**），因为它们被设计成是要被覆盖（override）的。任何一个类，它在覆盖这些方法的时候，都有责任遵守这些通用约定；如果不能做到这一点，其他依赖于这些约定的类（例如HashMap和HashSet）就无法结合该类一起正常运作。

本章将讲述何时以及如何覆盖这些非final的Object方法。本章不再讨论finalize方法，因为第7条已经讨论过这个方法了。而Comparable.compareTo虽然不是Object方法，但是本章也对它进行讨论，因为它具有类似的特征。

第8条：覆盖equals时请遵守通用约定

覆盖equals方法看起来似乎很简单，但是有许多覆盖方式会导致错误，并且后果非常严重。最容易避免这类问题的办法就是不覆盖equals方法，在这种情况下，类的每个实例都只与它自身相等。如果满足了以下任何一个条件，这就正是所期望的结果：

- **类的每个实例本质上都是唯一的。**对于代表活动实体而不是值（value）的类来说确实如此，例如Thread。Object提供的equals实现对于这些类来说正是正确的行为。
- **不关心类是否提供了“逻辑相等（logical equality）”的测试功能。**例如，java.util.Random覆盖了equals，以检查两个Random实例是否产生相同的随机数序列，但是设计者并不认为客户需要或者期望这样的功能。在这样的情况下，从Object继承得到的equals实现已经足够了。
- **超类已经覆盖了equals，从超类继承过来的行为对于子类也是合适的。**例如，大多数的Set实现都从AbstractSet继承equals实现，List实现从AbstractList继承equals实现，Map

实现从AbstractMap继承equals实现。

- **类是私有的或是包级私有的，可以确定它的equals方法永远不会被调用。**在这种情况下，无疑是应该覆盖equals方法的，以防它被意外调用：

```
@Override public boolean equals(Object o) {
    throw new AssertionError(); // Method is never called
}
```

那么，什么时候应该覆盖Object.equals呢？如果类具有自己特有的“逻辑相等”概念（不同于对象等同的概念），而且超类还没有覆盖equals以实现期望的行为，这时我们就需要覆盖equals方法。这通常属于“值类（value class）”的情形。值类仅仅是一个表示值的类，例如Integer或者Date。程序员在利用equals方法来比较值对象的引用时，希望知道它们在逻辑上是否相等，而不是想了解它们是否指向同一个对象。为了满足程序员的要求，不仅必需覆盖equals方法，而且这样做也使得这个类的实例可以被用做映射表（map）的键（key），或者集合（set）的元素，使映射或者集合表现出预期的行为。

有一种“值类”不需要覆盖equals方法，即用实例受控（见第1条）确保“每个值至多只存在一个对象”的类。枚举类型（见第30条）就属于这种类。对于这样的类而言，逻辑相同与对象等同是一回事，因此Object的equals方法等同于逻辑意义上的equals方法。

在覆盖equals方法的时候，你必须要遵守它的通用约定。下面是约定的内容，来自Object的规范[JavaSE6]：

equals方法实现了*等价关系*（**equivalence relation**）：

- **自反性**（reflexive）。对于任何非null的引用值x，x.equals(x)必须返回true。
- **对称性**（symmetric）。对于任何非null的引用值x和y，当且仅当y.equals(x)返回true时，x.equals(y)必须返回true。
- **传递性**（transitive）。对于任何非null的引用值x、y和z，如果x.equals(y)返回true，并且y.equals(z)也返回true，那么x.equals(z)也必须返回true。
- **一致性**（consistent）。对于任何非null的引用值x和y，只要equals的比较操作在对象中所用的信息没有被修改，多次调用x.equals(y)就会一致地返回true，或者一致地返回false。
- 对于任何非null的引用值x，x.equals(null)必须返回false。

除非你对数学特别感兴趣，否则这些规定看起来可能有点让人感到恐惧，但是绝对不要忽

视这些规定！如果你违反了它们，就会发现你的程序将会表现不正常，甚至崩溃，而且很难找到失败的根源。用John Donne的话说，没有哪个类是孤立的。一个类的实例通常会被频繁地传递给另一个类的实例。有许多类，包括所有的集合类（collection class）在内，都依赖于传递给它们的对象是否遵守了equals约定。

现在你已经知道了违反equals约定有多么可怕，现在我们就来更细致地讨论这些约定。值得欣慰的是，这些约定虽然看起来很吓人，实际上并不十分复杂。一旦理解了这些约定，要遵守它们并不困难。现在我们按照顺序逐一查看以下5个要求：

自反性（reflexivity）——第一个要求仅仅说明对象必须等于其自身。很难想像会无意识地违反这一条。假如违背了这一条，然后把该类的实例添加到集合（collection）中，该集合的contains方法将果断地告诉你，该集合不包含你刚刚添加的实例。

对称性（symmetry）——第二个要求是说，任何两个对象对于“它们是否相等”的问题都必须保持一致。与第一个要求不同，若无意中违反这一条，这种情形倒是不难想像。例如，考虑下面的类，它实现了一个区分大小写的字符串。字符串由toString保存，但在比较操作中被忽略。

```
// Broken - violates symmetry!
public final class CaseInsensitiveString {
    private final String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }

    // Broken - violates symmetry!
    @Override public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString) o).s);
        if (o instanceof String)  // One-way interoperability!
            return s.equalsIgnoreCase((String) o);
        return false;
    }
    ...  // Remainder omitted
}
```

在这个类中，equals方法的意图非常好，它企图与普通的字符串（String）对象进行互操作。假设我们有一个不区分大小写的字符串和一个普通的字符串：

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

正如所料，cis.equals(s)返回true。问题在于，虽然CaseInsensitiveString类中的equals方

法知道普通的字符串（String）对象，但是，String类中的equals方法却并不知道不区分大小写的字符串。因此，s.equals(cis)返回false，显然违反了对称性。假设你把不区分大小写的字符串对象放到一个集合中：

```
List<CaseInsensitiveString> list =
    new ArrayList<CaseInsensitiveString>();
list.add(cis);
```

此时list.contains(s)会返回什么结果呢？没人知道。在Sun的当前实现中，它碰巧返回false，但这只是这个特定实现得出的结果而已。在其他的实现中，它有可能返回true，或者抛出一个运行时（runtime）异常。一旦违反了**equals**约定，当其他对象面对你的对象时，你完全不知道这些对象的行为会怎么样。

为了解决这个问题，只需把企图与String互操作的这段代码从equals方法中去掉就可以了。这样做之后，就可以重构该方法，使它变成一条单独的返回语句：

```
@Override public boolean equals(Object o) {
    return o instanceof CaseInsensitiveString &&
        ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

传递性（transitivity）——equals约定的第三个要求是，如果一个对象等于第二个对象，并且第二个对象又等于第三个对象，则第一个对象一定等于第三个对象。同样地，无意识地违反这条规则的情形也不难想像。考虑子类的情形，它将一个新的值组件（value component）添加到了超类中。换句话说，子类增加的信息会影响到equals的比较结果。我们首先以一个简单的不可变的二维整数型Point类作为开始：

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

    @Override public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }

    ... // Remainder omitted
}
```

假设你想要扩展这个类，为一个点添加颜色信息：

```
public class ColorPoint extends Point {
```

```
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    ... // Remainder omitted
}
```

equals方法会怎么样呢？如果完全不提供equals方法，而是直接从Point继承过来，在equals做比较的时候颜色信息就被忽略掉了。虽然这样做不会违反equals约定，但是很明显这是无法接受的。假设你编写了一个equals方法，只有当它的参数是另一个有色点，并且具有同样的位置和颜色时，它才会返回true：

```
// Broken - violates symmetry!
@Override public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

这个方法的问题在于，你在比较普通点和有色点，以及相反的情形时，可能会得到不同的结果。前一种比较忽略了颜色信息，而后一种比较则总是返回false，因为参数的类型不正确。为了直观地说明问题所在，我们创建一个普通点和一个有色点：

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

然后，p.equals(cp)返回true，cp.equals(p)则返回false。你可以做这样的尝试来修正这个问题，让ColorPoint.equals在进行“混合比较”时忽略颜色信息：

```
// Broken - violates transitivity!
@Override public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;

    // If o is a normal Point, do a color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);

    // o is a ColorPoint; do a full comparison
    return super.equals(o) && ((ColorPoint)o).color == color;
}
```

这种方法确实提供了对称性，但是却牺牲了传递性：

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

此时，p1.equals(p2)和p2.equals(p3)都返回true，但是p1.equals(p3)则返回false，很显然违反了传递性。前两种比较不考虑颜色信息（“色盲”），而第三种比较则考虑了颜色信息。

怎么解决呢？事实上，这是面向对象语言中关于等价关系的一个基本问题。我们无法在扩展可实例化的类的同时，既增加新的值组件，同时又保留**equals**约定，除非愿意放弃面向对象的抽象所带来的优势。

你可能听说，在equals方法中用getClass测试代替instanceof测试，可以扩展可实例化的类和增加新的值组件，同时保留equals约定：

```
// Broken - violates Liskov substitution principle (page 40)
@Override public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass())
        return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

这段程序只有当对象具有相同的实现时，才能使对象等同。虽然这样也不算太糟糕，但是结果却是无法接受的。

假设我们要编写一个方法，以检验某个整值点是否处在单位圆中。下面是可以采用的其中一种方法：

```
// Initialize UnitCircle to contain all Points on the unit circle
private static final Set<Point> unitCircle;
static {
    unitCircle = new HashSet<Point>();
    unitCircle.add(new Point( 1,  0));
    unitCircle.add(new Point( 0,  1));
    unitCircle.add(new Point(-1,  0));
    unitCircle.add(new Point( 0, -1));
}

public static boolean onUnitCircle(Point p) {
    return unitCircle.contains(p);
}
```

虽然这可能不是实现这种功能的最快方式，不过它的效果很好。但是假设你通过某种不添加值组件的方式扩展了Point，例如让它的构造器记录创建了多少个实例：

```
public class CounterPoint extends Point {
    private static final AtomicInteger counter =
        new AtomicInteger();

    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }
    public int numberCreated() { return counter.get(); }
}
```

里氏替换原则（*Liskov substitution principle*）认为，一个类型的任何重要属性也将适用于它的子类型，因此为该类型编写的任何方法，在它的子类型上也应该同样运行得很好[Liskov87]。但是假设我们将CounterPoint实例传给了onUnitCircle方法。如果Point类使用了基于getClass的equals方法，无论CounterPoint实例的x和y值是什么，onUnitCircle方法都会返回false。之所以如此，是因为像onUnitCircle方法所用的HashSet这样的集合，利用equals方法检验包含条件，没有任何CounterPoint实例与任何Point对应。但是，如果在Point上使用适当的基于instanceof的equals方法，当遇到CounterPoint时，相同的onUnitCircle方法就会工作得很好。

虽然没有一种令人满意的办法可以既扩展不可实例化的类，又增加值组件，但还是有一种不错的权宜之计（workaround）。根据第16条的建议：复合优先于继承。我们不再让ColorPoint扩展Point，而是在ColorPoint中加入一个私有的Point域，以及一个公有的视图（view）方法（见第5条），此方法返回一个与该有色点处在相同位置的普通Point对象：

```
// Adds a value component without violating the equals contract
public class ColorPoint {
   private final Point point;
   private final Color color;

   public ColorPoint(int x, int y, Color color) {
      if (color == null)
         throw new NullPointerException();
      point = new Point(x, y);
      this.color = color;
   }

   /**
    * Returns the point-view of this color point.
    */
   public Point asPoint() {
      return point;
   }

   @Override public boolean equals(Object o) {
      if (!(o instanceof ColorPoint))
         return false;
      ColorPoint cp = (ColorPoint) o;
      return cp.point.equals(point) && cp.color.equals(color);
   }

   ... // Remainder omitted
}
```

在Java平台类库中，有一些类扩展了可实例化的类，并添加了新的值组件。例如，java.sql.Timestamp对java.util.Date进行了扩展，并增加了nanoseconds域。Timestamp的equals实现确实违反了对称性，如果Timestamp和Date对象被用于同一个集合中，或者以其他方式被混合在一起，则会引起不正确的行为。Timestamp类有一个免责声明，告诫程序员不要混合使用Date和Timestamp对象。只要你不把它们混合在一起，就不会有麻烦，除此之外没有其他的措施可以防止你这么做，而且结果导致的错误将很难调试。Timestamp类的这种行为是个错误，

不值得仿效。

注意，你可以在一个抽象（**abstract**）类的子类中增加新的值组件，而不会违反equals约定。对于根据第20条的建议“用类层次（class hierarchies）代替标签类（tagged class）”而得到的那种类层次结构来说，这一点非常重要。例如，你可能有一个抽象的Shape类，它没有任何值组件，Circle子类添加了一个radius域，Rectangle子类添加了length和width域。只要不可能直接创建超类的实例，前面所述的种种问题就都不会发生。

一致性（consistency）——equals约定的第四个要求是，如果两个对象相等，它们就必须始终保持相等，除非它们中有一个对象（或者两个都）被修改了。换句话说，可变对象在不同的时候可以与不同的对象相等，而不可变对象则不会这样。当你在写一个类的时候，应该仔细考虑它是否应该是不可变的（见第15条）。如果认为它应该是不可变的，就必须保证equals方法满足这样的限制条件：相等的对象永远相等，不相等的对象永远不相等。

无论类是否是不可变的，*都不要使**equals**方法依赖于不可靠的资源*。如果违反了这条禁令，要想满足一致性的要求就十分困难了。例如，java.net.URL的equals方法依赖于对URL中主机IP地址的比较。将一个主机名转变成IP地址可能需要访问网络，随着时间的推移，不确保会产生相同的结果。这样会导致URL的equals方法违反equals约定，在实践中有可能引发一些问题。（遗憾的是，因为兼容性的要求，这一行为无法被改变。）除了极少数的例外情况，equals方法都应该对驻留在内存中的对象执行确定性的计算。

非空性（Non-nullity）——最后一个要求没有名称，我姑且称它为“非空性（Non-nullity）”，意思是指所有的对象都必须不等于null。尽管很难想像什么情况下o.equals（null）调用会意外地返回true，但是意外抛出NullPointerException异常的情形却不难想像。通用约定不允许抛出NullPointerException异常。许多类的equals方法都通过一个显式的null测试来防止这种情况：

```
@Override public boolean equals(Object o) {
    if (o == null)
        return false;
    ...
}
```

这项测试是不必要的。为了测试其参数的等同性，equals方法必须先把参数转换成适当的类型，以便可以调用它的访问方法（accessor），或者访问它的域。在进行转换之前，equals方法必须使用instanceof操作符，检查其参数是否为正确的类型：

```
@Override public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;
    ...
```

```
}
```

如果漏掉了这一步的类型检查，并且传递给equals方法的参数又是错误的类型，那么equals方法将会抛出ClassCastException异常，这就违反了equals的约定。但是，如果instanceof的第一个操作数为null，那么，不管第二个操作数是哪种类型，instanceof操作符都指定应该返回false[JLS，15.20.2]。因此，如果把null传给equals方法，类型检查就会返回false，所以不需要单独的null检查。

结合所有这些要求，得出了以下实现高质量equals方法的诀窍：

1. **使用==操作符检查“参数是否为这个对象的引用”**。如果是，则返回true。这只不过是一种性能优化，如果比较操作有可能很昂贵，就值得这么做。

2. **使用instanceof操作符检查“参数是否为正确的类型”**。如果不是，则返回false。一般说来，所谓“正确的类型”是指equals方法所在的那个类。有些情况下，是指该类所实现的某个接口。如果类实现的接口改进了equals约定，允许在实现了该接口的类之间进行比较，那么就使用接口。集合接口（collection interface）如Set、List、Map和Map.Entry具有这样的特性。

3. **把参数转换成正确的类型**。因为转换之前进行过instanceof测试，所以确保会成功。

4. **对于该类中的每个“关键（significant）”域，检查参数中的域是否与该对象中对应的域相匹配**。如果这些测试全部成功，则返回true；否则返回false。如果第2步中的类型是个接口，就必须通过接口方法访问参数中的域；如果该类型是个类，也许就能够直接访问参数中的域，这要取决于它们的可访问性。

对于既不是float也不是double类型的基本类型域，可以使用==操作符进行比较；对于对象引用域，可以递归地调用equals方法；对于float域，可以使用Float.compare方法；对于double域，则使用Double.compare。对float和double域进行特殊的处理是有必要的，因为存在着Float.NaN、-0.0f以及类似的double常量；详细信息请参考Float.equals的文档。对于数组域，则要把以上这些指导原则应用到每个元素上。如果数组域中的每个元素都很重要，就可以使用发行版本1.5中新增的其中一个Arrays.equals方法。

有些对象引用域包含null可能是合法的，所以，为了避免可能导致NullPointerException异常，则使用下面的习惯用法来比较这样的域：

```
(field == null ? o.field == null : field.equals(o.field))
```

如果field和o.field通常是相同的对象引用，那么下面的做法就会更快一些：

```
(field == o.field || (field != null && field.equals(o.field)))
```

对于有些类，比如前面提到的CaseInsensitiveString类，域的比较要比简单的等同性测试复杂得多。如果是这种情况，可能会希望保存该域的一个“范式（canonical form）”，这样equals方法就可以根据这些范式进行低开销的精确比较，而不是高开销的非精确比较。这种方法对于不可变类（见第15条）是最为合适的；如果对象可能发生变化，就必须使其范式保持最新。

域的比较顺序可能会影响到equals方法的性能。为了获得最佳的性能，应该最先比较最有可能不一致的域，或者是开销最低的域，最理想的情况是两个条件同时满足的域。你不应该去比较那些不属于对象逻辑状态的域，例如用于同步操作的Lock域。也不需要比较冗余域（redundant field），因为这些冗余域可以由“关键域”计算获得，但是这样做有可能提高equals方法的性能。如果冗余域代表了整个对象的综合描述，比较这个域可以节省当比较失败时去比较实际数据所需要的开销。例如，假设有一个Polygon类，并缓存了该区域。如果两个多边形有着不同的区域，就没有必要去比较它们的边和至高点。

5. 当你编写完成了equals方法之后，应该问自己三个问题：它是否是对称的、传递的、一致的？并且不要只是自问，还要编写单元测试来检验这些特性！如果答案是否定的，就要找出原因，再相应地修改equals方法的代码。当然，equals方法也必须满足其他两个特性（自反性和非空性），但是这两种特性通常会自动满足。

根据上面的诀窍构建的equals方法的具体例子，请参看第9条的PhoneNumber. equals。下面是最后的一些告诫：

- **覆盖equals时总要覆盖hashCode**（见第9条）。
- **不要企图让equals方法过于智能。**如果只是简单地测试域中的值是否相等，则不难做到遵守equals约定。如果想过度地去寻求各种等价关系，则很容易陷入麻烦之中。把任何一种别名形式考虑到等价的范围内，往往不会是个好主意。例如，File类不应该试图把指向同一个文件的符号链接（symbolic link）当作相等的对象来看待。所幸File类没有这样做。
- **不要将equals声明中的Object对象替换为其他的类型。**程序员编写出下面这样的equals方法并不鲜见，这会使程序员花上数个小时都搞不清为什么它不能正常工作：

```
public boolean equals(MyClass o) {
    ...
}
```

问题在于，这个方法并没有覆盖Object.equals，因为它的参数应该是Object类型，相反，

它重载（overload）了Object.equals（见第41条）。在原有equals方法的基础上，再提供一个“强类型（strongly typed）”的equals方法，只要这两个方法返回同样的结果（没有强制的理由必须这样做），那么这就是可以接受的。在某些特定的情况下，它也许能够稍微改善性能，但是与增加的复杂性相比，这种做法是不值得的（见第55条）。

@Override注解的用法一致，就如本条目中所示，可以防止犯这种错误（见第36条）。这个equals方法不能编译，错误消息会告诉你到底哪里出了问题：

```
@Override public boolean equals(MyClass o) {
    ...
}
```

第9条：覆盖equals时总要覆盖hashCode

一个很常见的错误根源在于没有覆盖hashCode方法。*在每个覆盖了**equals**方法的类中，也必须覆盖**hashCode**方法*。如果不这样做的话，就会违反Object.hashCode的通用约定，从而导致该类无法结合所有基于散列的集合一起正常运作，这样的集合包括HashMap、HashSet和Hashtable。

下面是约定的内容，摘自Object规范[JavaSE6]：

- 在应用程序的执行期间，只要对象的equals方法的比较操作所用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法都必须始终如一地返回同一个整数。在同一个应用程序的多次执行过程中，每次执行所返回的整数可以不一致。
- 如果两个对象根据equals(Object)方法比较是相等的，那么调用这两个对象中任意一个对象的hashCode方法都必须产生同样的整数结果。
- 如果两个对象根据equals(Object)方法比较是不相等的，那么调用这两个对象中任意一个对象的hashCode方法，则不一定要产生不同的整数结果。但是程序员应该知道，给不相等的对象产生截然不同的整数结果，有可能提高散列表（hash table）的性能。

*因没有覆盖**hashCode**而违反的关键约定是第二条：相等的对象必须具有相等的散列码*（hash code）。根据类的equals方法，两个截然不同的实例在逻辑上有可能是相等的，但是，根据Object类的hashCode方法，它们仅仅是两个没有任何共同之处的对象。因此，对象的hashCode方法返回两个看起来是随机的整数，而不是根据第二个约定所要求的那样，返回两个相等的整数。

例如，考虑下面这个极为简单的PhoneNumber类，它的equals方法是根据第8条中给出的“诀窍”构造出来的：

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    public PhoneNumber(int areaCode, int prefix,
                       int lineNumber) {
        rangeCheck(areaCode,    999, "area code");
        rangeCheck(prefix,      999, "prefix");
        rangeCheck(lineNumber, 9999, "line number");
        this.areaCode  = (short) areaCode;
        this.prefix    = (short) prefix;
        this.lineNumber = (short) lineNumber;
    }
```

```
    private static void rangeCheck(int arg, int max,
                                   String name) {
        if (arg < 0 || arg > max)
            throw new IllegalArgumentException(name +": " + arg);
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNumber == lineNumber
            && pn.prefix  == prefix
            && pn.areaCode  == areaCode;
    }

    // Broken - no hashCode method!

    ... // Remainder omitted
}
```

假设你企图将这个类与HashMap一起使用：

```
Map<PhoneNumber, String> m
    = new HashMap<PhoneNumber, String>();
m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

这时候，你可能期望m.get(new PhoneNumber(408, 867, 5309))会返回“Jenny”，但它实际上返回的是null。注意，这里涉及两个PhoneNumber实例：第一个被用于插入到HashMap中，第二个实例与第一个相等，被用于（试图用于）获取。由于PhoneNumber类没有覆盖hashCode方法，从而导致两个相等的实例具有不相等的散列码，违反了hashCode的约定。因此，put方法把电话号码对象存放在一个散列桶（hash bucket）中，get方法却在另一个散列桶中查找这个电话号码。即使这两个实例正好被放到同一个散列桶中，get方法也必定会返回null，因为HashMap有一项优化，可以将与每个项相关联的散列码缓存起来，如果散列码不匹配，也不必检验对象的等同性。

修正这个问题非常简单，只需为PhoneNumber类提供一个适当的hashCode方法即可。那么，hashCode方法应该是什么样的呢？编写一个合法但并不好用的hashCode方法没有任何价值。例如，下面这个方法总是合法，但是永远都不应该被正式使用：

```
// The worst possible legal hash function - never use!
@Override public int hashCode() { return 42; }
```

上面这个hashCode方法是合法的，因为它确保了相等的对象总是具有同样的散列码。但它也极为恶劣，因为它使得每个对象都具有同样的散列码。因此，每个对象都被映射到同一个散列桶中，使散列表退化为链表（linked list）。它使得本该线性时间运行的程序变成了以平方级时间在运行。对于规模很大的散列表而言，这会关系到散列表能否正常工作。

一个好的散列函数通常倾向于“为不相等的对象产生不相等的散列码”。这正是hashCode约定中第三条的含义。理想情况下，散列函数应该把集合中不相等的实例均匀地分布到所有可能的散列值上。要想完全达到这种理想的情形是非常困难的。幸运的是，相对接近这种理想情形则并不太困难。下面给出一种简单的解决办法：

1. 把某个非零的常数值，比如说17，保存在一个名为result的int类型的变量中。

2. 对于对象中每个关键域f（指equals方法中涉及的每个域），完成以下步骤：

a. 为该域计算int类型的散列码c：

i. 如果该域是boolean类型，则计算(f ? 1 : 0)。

ii. 如果该域是byte、char、short或者int类型，则计算(int)f。

iii. 如果该域是long类型，则计算(int)(f ^ (f >>> 32))。

iv. 如果该域是float类型，则计算Float.floatToIntBits(f)。

v. 如果该域是double类型，则计算Double.doubleToLongBits(f)，然后按照步骤2.a.iii，为得到的long类型值计算散列值。

vi. 如果该域是一个对象引用，并且该类的equals方法通过递归地调用equals的方式来比较这个域，则同样为这个域递归地调用hashCode。如果需要更复杂的比较，则为这个域计算一个“范式（canonical representation）”，然后针对这个范式调用hashCode。如果这个域的值为null，则返回0（或者其他某个常数，但通常是0）。

vii. 如果该域是一个数组，则要把每一个元素当做单独的域来处理。也就是说，递归地应用上述规则，对每个重要的元素计算一个散列码，然后根据步骤2.b中的做法把这些散列值组合起来。如果数组域中的每个元素都很重要，可以利用发行版本1.5中增加的其中一个Arrays.hashCode方法。

b. 按照下面的公式，把步骤2.a中计算得到的散列码c合并到result中：

```
result = 31 * result + c;
```

3. 返回result。

4. 写完了hashCode方法之后，问问自己“相等的实例是否都具有相等的散列码”。要编写单元测试来验证你的推断。如果相等的实例有着不相等的散列码，则要找出原因，并修正错误。

在散列码的计算过程中，可以把冗余域（**redundant field**）排除在外。换句话说，如果一个域的值可以根据参与计算的其他域值计算出来，则可以把这样的域排除在外。必须排除equals比较计算中没有用到的任何域，否则很有可能违反hashCode约定的第二条。

上述步骤1中用到了一个非零的初始值，因此步骤2.a中计算的散列值为0的那些初始域，会影响到散列值。如果步骤1中的初始值为0，则整个散列值将不受这些初始域的影响，因为这些初始域会增加冲突的可能性。值17则是任选的。

步骤2.b中的乘法部分使得散列值依赖于域的顺序，如果一个类包含多个相似的域，这样的乘法运算就会产生一个更好的散列函数。例如，如果String散列函数省略了这个乘法部分，那么只是字母顺序不同的所有字符串都会有相同的散列码。之所以选择31，是因为它是一个奇素数。如果乘数是偶数，并且乘法溢出的话，信息就会丢失，因为与2相乘等价于移位运算。使用素数的好处并不很明显，但是习惯上都使用素数来计算散列结果。31有个很好的特性，即用移位和减法来代替乘法，可以得到更好的性能：31 * i == (i << 5) − i。现代的VM可以自动完成这种优化。

现在我们要把上述的解决办法用到PhoneNumber类中。它有三个关键域，都是short类型：

```
@Override public int hashCode() {
    int result = 17;
    result = 31 * result + areaCode;
    result = 31 * result + prefix;
    result = 31 * result + lineNumber;
    return result;
}
```

因为这个方法返回的结果是一个简单的、确定的计算结果，它的输入只是PhoneNumber实例中的三个关键域，因此相等的PhoneNumber显然都会有相等的散列码。实际上，对于PhoneNumber的hashCode实现而言，上面这个方法是非常合理的，相当于Java平台类库中的实现。它的做法非常简单，也相当快捷，恰当地把不相等的电话号码分散到不同的散列桶中。

如果一个类是不可变的，并且计算散列码的开销也比较大，就应该考虑把散列码缓存在对象内部，而不是每次请求的时候都重新计算散列码。如果你觉得这种类型的大多数对象会被用做散列键（hash keys），就应该在创建实例的时候计算散列码。否则，可以选择“延迟初始化（**lazily initialize**）”散列码，一直到hashCode被第一次调用的时候才初始化（见第71条）。现在尚不清楚我们的PhoneNumber类是否值得这样处理，但可以通过它来说明这种方法该如何实现：

```
// Lazily initialized, cached hashCode
private volatile int hashCode;  // (See Item 71)

@Override public int hashCode() {
    int result = hashCode;
```

```
        if (result == 0) {
            result = 17;
            result = 31 * result + areaCode;
            result = 31 * result + prefix;
            result = 31 * result + lineNumber;
            hashCode = result;
        }
        return result;
    }
```

虽然本条目中前面给出的hashCode实现方法能够获得相当好的散列函数，但是它并不能产生最新的散列函数，截止发行版本1.6，Java平台类库也没有提供这样的散列函数。编写这种散列函数是个研究课题，最好留给数学家和理论方面的计算机科学家来完成。也许Java平台的下一个发行版本将会为它的类提供这种最佳的散列函数，并提供一些实用方法来帮助普通的程序员构造出这样的散列函数。与此同时，本条目中介绍的方法对于绝大多数应用程序而言已经足够了。

不要试图从散列码计算中排除掉一个对象的关键部分来提高性能。虽然这样得到的散列函数运行起来可能更快，但是它的效果不见得会好，可能会导致散列表慢到根本无法使用。特别是在实践中，散列函数可能面临大量的实例，在你选择忽略的区域之中，这些实例仍然区别非常大。如果是这样，散列函数就会把所有这些实例映射到极少数的散列码上，基于散列的集合将会显示出平方级的性能指标。这不仅仅是个理论问题。在Java 1.2发行版本之前实现的String散列函数至多只检查16个字符，从第一个字符开始，在整个字符串中均匀选取。对于像URL这种层次状名字的大型集合，该散列函数正好表现出了这里所提到的病态行为。

Java平台类库中的许多类，比如String、Integer和Date，都可以把它们的hashCode方法返回的确切值规定为该实例值的一个函数。一般来说，这并不是个好主意，因为这样做严格地限制了在将来的版本中改进散列函数的能力。如果没有规定散列函数的细节，那么当你发现了它的内部缺陷时，就可以在后面的发行版本中修正它，确信没有任何客户端会依赖于散列函数返回的确切值。

第10条：始终要覆盖toString

虽然jave.lang.Object提供了toString方法的一个实现，但它返回的字符串通常并不是类的用户所期望看到的。它包含类的名称，以及一个“@”符号，接着是散列码的无符号十六进制表示法，例如“PhoneNumber@163b91”。toString的通用约定指出，被返回的字符串应该是一个“简洁的，但信息丰富，并且易于阅读的表达形式”[JavaSE6]。尽管有人认为“PhoneNumber@163b91”算得上是简洁和易于阅读了，但是与“(707)867-5309”比较起来，它还算不上是信息丰富的。toString的约定进一步指出，“建议所有的子类都覆盖这个方法。”这是一个很好的建议，真的！

虽然遵守toString的约定并不像遵守equals和hashCode的约定（见第8条和第9条）那么重要，但是，*提供好的***toString***实现可以使类用起来更加舒适*。当对象被传递给println、printf、字符串联操作符（+）以及assert或者被调试器打印出来时，toString方法会被自动调用。（Java 1.5发行版本在平台中增加了printf方法，还提供了包括String.format的相关方法，与C语言中的sprintf相似。）

如果为PhoneNumber提供了好的toString方法，那么，要产生有用的诊断消息会非常容易：

```
System.out.println("Failed to connect: " + phoneNumber);
```

不管是否覆盖了toString方法，程序员都将以这种方式来产生诊断消息，但是如果没有覆盖toString方法，产生的消息将难以理解。提供好的toString方法，不仅有益于这个类的实例，同样也有益于那些包含这些实例的引用的对象，特别是集合对象。打印Map时有下面这两条消息：“Jenny=PhoneNumber@163b91”和“Jenny=(408) 867-5309”，你更愿意看到哪一个？

*在实际应用中，***toString***方法应该返回对象中包含的所有值得关注的信息*，譬如上述电话号码例子那样。如果对象太大，或者对象中包含的状态信息难以用字符串来表达，这样做就有点不切实际。在这种情况下，toString应该返回一个摘要信息，例如“Manhattan white pages (1487536 listings)”或者“Thread[main, 5, main]”。理想情况下，字符串应该是自描述的（self-explanatory），（Thread例子不满足这样的要求。）

在实现toString的时候，必须要做出一个很重要的决定：是否在文档中指定返回值的格式。对于值类（value class），比如电话号码类、矩阵类，也建议这么做。指定格式的好处是，它可以被用做一种标准的、明确的、适合人阅读的对象表示法。这种表示法可以用于输入和输出，以及用在永久的适合于人类阅读的数据对象中，例如XML文档。如果你指定了格式，最好再提供一个相匹配的静态工厂或者构造器，以便程序员可以很容易地在对象和它的字符串表示法之间来回转换。Java平台类库中的许多值类都采用了这种做法，包括BigInteger、BigDecimal和绝大多数的基本类型包装类（boxed primitive class）。

指定toString返回值的格式也有不足之处：如果这个类已经被广泛使用，一旦指定格式，就必须始终如一地坚持这种格式。程序员将会编写出相应的代码来解析这种字符串表示法、产生字符串表示法，以及把字符串表示法嵌入到持久的数据中。如果将来的发行版本中改变了这种表示法，就会破坏他们的代码和数据，他们当然会抱怨。如果不指定格式，就可以保留灵活性，便于在将来的发行版本中增加信息，或者改进格式。

*无论你是否决定指定格式，都应该在文档中明确地表明你的意图。*如果你要指定格式，则应该严格地这样去做。例如，下面是第9条中PhoneNumber类的toString方法：

```
/**
 * Returns the string representation of this phone number.
 * The string consists of fourteen characters whose format
 * is "(XXX) YYY-ZZZZ", where XXX is the area code, YYY is
 * the prefix, and ZZZZ is the line number.  (Each of the
 * capital letters represents a single decimal digit.)
 *
 * If any of the three parts of this phone number is too small
 * to fill up its field, the field is padded with leading zeros.
 * For example, if the value of the line number is 123, the last
 * four characters of the string representation will be "0123".
 *
 * Note that there is a single space separating the closing
 * parenthesis after the area code from the first digit of the
 * prefix.
 */
@Override public String toString() {
    return String.format("(%03d) %03d-%04d",
                         areaCode, prefix, lineNumber);
}
```

如果你决定不指定格式，那么文档注释部分也应该有如下所示的指示信息：

```
/**
 * Returns a brief description of this potion. The exact details
 * of the representation are unspecified and subject to change,
 * but the following may be regarded as typical:
 *
 * "[Potion #9: type=love, smell=turpentine, look=india ink]"
 */
@Override public String toString() { ... }
```

对于那些依赖于格式的细节进行编程或者产生永久数据的程序员，在读到这段注释之后，一旦格式被改变，则只能自己承担后果。

*无论是否指定格式，都为*toString*返回值中包含的所有信息，提供一种编程式的访问途径。*例如，PhoneNumber类应该包含针对area code、prefix和line number的访问方法。如果不这么做，就会迫使那些需要这些信息的程序员不得不自己去解析这些字符串。除了降低了程序的性能，使得程序员们去做这些不必要的工作之外，这个解析过程也很容易出错，会导致系统不稳定，如果格式发生变化，还会导致系统崩溃。如果没有提供这些访问方法，即使你已经指明了字符串的格式是可以变化的，这个字符串格式也成了事实上的API。

第11条：谨慎地覆盖clone

Cloneable接口的目的是作为对象的一个**mixin接口**（**mixin interface**）（见第18条），表明这样的对象允许克隆（clone）。遗憾的是，它并没有成功地达到这个目的。其主要的缺陷在于，它缺少一个clone方法，Object的clone方法是受保护的。如果不借助于反射（**reflection**）（见第53条），就不能仅仅因为一个对象实现了Cloneable，就可以调用clone方法。即使是反射调用也可能会失败，因为不能保证该对象一定具有可访问的clone方法。尽管存在这样那样的缺陷，这项设施仍然被广泛地使用着，因此值得我们进一步地了解。本条目将告诉你如何实现一个行为良好的clone方法，并讨论何时适合这样做，同时也简单地讨论了其他的可替换做法。

既然Cloneable并没有包含任何方法，那么它到底有什么作用呢？它决定了Object中受保护的clone方法实现的行为：如果一个类实现了Cloneable，Object的clone方法就返回该对象的逐域拷贝，否则就会抛出CloneNotSupportedException异常。这是接口的一种极端非典型的用法，也不值得仿效。通常情况下，实现接口是为了表明类可以为它的客户做些什么。然而，对于Cloneable接口，它改变了超类中受保护的方法的行为。

如果实现Cloneable接口是要对某个类起到作用，类和它的所有超类都必须遵守一个相当复杂的、不可实施的，并且基本上没有文档说明的协议。由此得到一种语言之外的（**extralinguistic**）机制：无需调用构造器就可以创建对象。

Clone方法的通用约定是非常弱的，下面是来自java.lang.Object规范中的约定内容[JavaSE6]：

创建和返回该对象的一个拷贝。这个“拷贝”的精确含义取决于该对象的类。一般的含义是，对于任何对象x，表达式

```
x.clone() != x
```

将会是true，并且，表达式

```
x.clone().getClass() == x.getClass()
```

将会是true，但这些都不是绝对的要求。虽然通常情况下，表达式

```
x.clone().equals(x)
```

将会是true，但是，这也不是一个绝对的要求。拷贝对象往往会导致创建它的类的一个新实例，但它同时也会要求拷贝内部的数据结构。这个过程中没有调用构造器。

这个约定存在几个问题。“不调用构造器”的规定太强硬了。行为良好的clone方法可以调用构造器来创建对象，构造之后再复制内部数据。如果这个类是final的，clone甚至可能会返回一个由构造器创建的对象。

然而，“x.clone().getClass()通常应该等同于x.getClass()”的规定又太软弱了。在实践中，程序员会假设：如果他们扩展了一个类，并且从子类中调用了super.clone，返回的对象就将是该子类的实例。超类能够提供这种功能的唯一途径是，返回一个通过调用super.clone而得到的对象。如果clone方法返回一个由构造器创建的对象，它就得到有错误的类。因此，*如果你覆盖了非**final**类中的**clone**方法，则应该返回一个通过调用**super.clone**而得到的对象。*如果类的所有超类都遵守这条规则，那么调用super.clone最终会调用Object的clone方法，从而创建出正确类的实例。这种机制大体上类似于自动的构造器调用链，只不过它不是强制要求的。

从1.6发行版本开始，Cloneable接口并没有清楚地指明，一个类在实现这个接口时应该承担哪些责任。*实际上，对于实现了**Cloneable**的类，我们总是期望它也提供一个功能适当的公有的**clone**方法。*通常情况下，除非该类的所有超类都提供了行为良好的clone实现，无论是公有的还是受保护的，否则，都不可能这么做。

假设你希望在一个类中实现Cloneable，并且它的超类都提供行为良好的clone方法。你从super.clone()中得到的对象可能会接近于最终要返回的对象，也可能相差甚远，这要取决于这个类的本质。从每个超类的角度来看，这个对象将是原始对象功能完整的克隆（clone）。在这个类中声明的域（如果有的话）将等同于被克隆对象中相应的域。如果每个域包含一个基本类型的值，或者包含一个指向不可变对象的引用，那么被返回的对象则可能正是你所需要的对象，在这种情况下不需要再做进一步处理。例如，第9条中的PhoneNumber类正是如此。在这种情况下，你所需要做的，除了声明实现了Cloneable之外，就是对Object中受保护的clone方法提供公有的访问途径：

```
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch(CloneNotSupportedException e) {
        throw new AssertionError();  // Can't happen
    }
}
```

注意上述的clone方法返回的是PhoneNumber，而不是返回Object。从Java 1.5发行版本开始，这么做是合法的，也是我们所期待的，因为1.5发行版本中引入了*协变返回类型*（**covariant return type**）作为泛型。换句话说，目前覆盖方法的返回类型可以是被覆盖方法的返回类型的子类了。这样有助于覆盖方法提供更多关于被返回对象的信息，并且在客户端中不必进行转换。由于Object.clone返回Object，PhoneNumber.clone必须在返回super.clone()的结果之前将它转换。这里体现了一条通则：*永远不要让客户去做任何类库能够替客户完成*

的事情。

如果对象中包含的域引用了可变的对象，使用上述这种简单的clone实现可能会导致灾难性的后果。例如，考虑第6条中的Stack类：

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

假设你希望把这个类做成可克隆的（cloneable）。如果它的clone方法仅仅返回super.clone()，这样得到的Stack实例，在其size域中具有正确的值，但是它的elements域将引用与原始Stack实例相同的数组。修改原始的实例会破坏被克隆对象中的约束条件，反之亦然。很快你就会发现，这个程序将产生毫无意义的结果，或者抛出NullPointerException异常。

如果调用Stack类中唯一的构造器，这种情况就永远不会发生。实际上，**clone方法就是另一个构造器；你必须确保它不会伤害到原始的对象，并确保正确地创建被克隆对象中的约束条件（invariant）**。为了使Stack类中的clone方法正常地工作，它必须要拷贝栈的内部信息。最容易的做法是，在elements数组中递归地调用clone：

```
@Override public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

注意，我们不一定要将elements.clone()的结果转换成Object[]。自Java 1.5发行版本起，在数组上调用clone返回的数组，其编译时类型与被克隆数组的类型相同。

还要注意，如果elements域是final的，上述方案就不能正常工作，因为clone方法是被禁止给elements域赋新值的。这是个根本的问题：**clone架构与引用可变对象的final域的正常用法是不相兼容的**，除非在原始对象和克隆对象之间可以安全地共享此可变对象。为了使类成为可克隆的，可能有必要从某些域中去掉final修饰符。

递归地调用clone有时还不够。例如，假设你正在为一个散列表编写clone方法，它的内部数据包含一个散列桶数组，每个散列桶都指向“键－值”对链表的第一个项，如果桶是空的，则为null。出于性能方面的考虑，该类实现了它自己的轻量级单向链表，而没有使用Java内部的java.util.LinkedList。该类如下：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;
    private static class Entry {
        final Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }
    }

    ... // Remainder omitted
}
```

假设你仅仅递归地克隆这个散列桶数组，就像我们对Stack类所做的那样：

```
// Broken - results in shared internal state!
@Override public HashTable clone() {
    try {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

虽然被克隆对象有它自己的散列桶数组，但是，这个数组引用的链表与原始对象是一样的，从而很容易引起克隆对象和原始对象中不确定的行为。为了修正这个问题，必须单独地拷贝并组成每个桶的链表。下面是一种常见的做法：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;
```

```
    private static class Entry {
        final Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }
        // Recursively copy the linked list headed by this Entry
        Entry deepCopy() {
            return new Entry(key, value,
                    next == null ? null : next.deepCopy());
        }
    }

    @Override public HashTable clone() {
        try {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];
            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy();
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
    ... // Remainder omitted
}
```

私有类HashTable.Entry被加强了，它支持一个“深度拷贝（deep copy）”方法。HashTable上的clone方法分配了一个大小适中的、新的buckets数组，并且遍历原始的buckets数组，对每一个非空散列桶进行深度拷贝。Entry类中的深度拷贝方法递归地调用它自身，以便拷贝整个链表（它是链表的头节点）。虽然这种方法很灵活，如果散列桶不是很长的话，也会工作得很好，但是，这样克隆一个链表并不是一种好办法，因为针对列表中的每个元素，它都要消耗一段栈空间。如果链表比较长，这很容易导致栈溢出。为了避免发生这种情况，你可以在deepCopy中用迭代（iteration）代替递归（recursion）：

```
// Iteratively copy the linked list headed by this Entry
Entry deepCopy() {
   Entry result = new Entry(key, value, next);

   for (Entry p = result; p.next != null; p = p.next)
      p.next = new Entry(p.next.key, p.next.value, p.next.next);

   return result;
}
```

克隆复杂对象的最后一种办法是，先调用super.clone，然后把结果对象中的所有域都设置成它们的空白状态（virgin state），然后调用高层（higher-level）的方法来重新产生对象的状态。在我们的HashTable例子中，buckets域将被初始化为一个新的散列桶数组，然后，对于正在被克隆的散列表中的每一个键－值映射，都调用put（key, value）方法（上面没有给出其代

码)。这种做法往往会产生一个简单、合理且相当优美的clone方法，但是它运行起来通常没有“直接操作对象及其克隆对象的内部状态的clone方法”快。

如同构造器一样，clone方法不应该在构造的过程中，调用新对象中任何非final的方法(见第17条)。如果clone调用了一个被覆盖的方法，那么在该方法所在的子类有机会修正它在克隆对象中的状态之前，该方法就会先被执行，这样很有可能会导致克隆对象和原始对象之间的不一致。因此，上一段落中讨论到的put(key, value)方法应该要么是final的，要么是私有的（如果是私有的，它应该算是非final公有方法的“辅助方法[helper method]”)。

Object的clone方法被声明为可抛出CloneNotSupportedException异常，但是，覆盖版本的clone方法可能会忽略这个声明。公有的clone方法应该省略这个声明，因为不会抛出受检异常(checked exception）的方法与会抛出异常的方法相比，使用起来更加轻松（见第59条)。如果专门为了继承而设计的类[见第17条] 覆盖了clone方法，覆盖版本的clone方法就应该模拟Object.clone的行为：它应该被声明为protected、抛出CloneNotSupportedException异常，并且该类不应该实现Cloneable接口。这样做可以使子类具有实现或不实现Cloneable接口的自由，就仿佛它们直接扩展了Object一样。

还有一点值得注意。如果你决定用线程安全的类实现Cloneable接口，要记得它的clone方法必须得到很好的同步，就像任何其他方法一样（见第66条)。Object的clone方法没有同步，因此即使很满意，可能也必须编写同步的clone方法来调用super.clone()。

简而言之，所有实现了Cloneable接口的类都应该用一个公有的方法覆盖clone。此公有方法首先调用super.clone，然后修正任何需要修正的域。一般情况下，这意味着要拷贝任何包含内部“深层结构”的可变对象，并用指向新对象的引用代替原来指向这些对象的引用。虽然，这些内部拷贝操作往往可以通过递归地调用clone来完成，但这通常并不是最佳方法。如果该类只包含基本类型的域，或者指向不可变对象的引用，那么多半的情况是没有域需要修正。这条规则也有例外，譬如，代表序列号或其他唯一ID值的域，或者代表对象的创建时间的域，不管这些域是基本类型还是不可变的，它们也都需要被修正。

真的有必要这么复杂吗？很少有这种必要。如果你扩展一个实现了Cloneable接口的类，那么你除了实现一个行为良好的clone方法外，没有别的选择。否则，最好提供某些其他的途径来代替对象拷贝，或者干脆不提供这样的功能。例如，对于不可变类，支持对象拷贝并没有太大的意义，因为被拷贝的对象与原始对象并没有实质的不同。

另一个实现对象拷贝的好办法是提供一个拷贝构造器（**copy constructor**）或拷贝工厂（**copy factory**)。拷贝构造器只是一个构造器，它唯一的参数类型是包含该构造器的类，例如：

```
public Yum(Yum yum);
```

拷贝工厂是类似于拷贝构造器的静态工厂：

```
public static Yum newInstance(Yum yum);
```

拷贝构造器的做法，及其静态工厂方法的变形，都比Cloneable/clone方法具有更多的优势：它们不依赖于某一种很有风险的、语言之外的对象创建机制；它们不要求遵守尚未制定好文档的规范；它们不会与final域的正常使用发生冲突；它们不会抛出不必要的受检异常（checked exception）；它们不需要进行类型转换。虽然你不可能把拷贝构造器或者静态工厂放到接口中，但是由于Cloneable接口缺少一个公有的clone方法，所以它也没有提供一个接口该有的功能。因此，使用拷贝构造器或者拷贝工厂来代替clone方法时，并没有放弃接口的功能特性。

更进一步，拷贝构造器或者拷贝工厂可以带一个参数，参数类型是通过该类实现的接口。例如，按照惯例，所有通用集合实现都提供了一个拷贝构造器，它的参数类型为Collection或者Map。基于接口的拷贝构造器和拷贝工厂（更准确的叫法应该是“转换构造器（conversion constructor）”和转换工厂（conversion factory）），允许客户选择拷贝的实现类型，而不是强迫客户接受原始的实现类型。例如，假设你有一个HashSet，并且希望把它拷贝成一个TreeSet。clone方法无法提供这样的功能，但是用转换构造器很容易实现：new TreeSet(s)。

既然Cloneable具有上述那么多问题，可以肯定地说，其他的接口都不应该扩展（extend）这个接口，为了继承而设计的类（见第17条）也不应该实现（implement）这个接口。由于它具有这么多缺点，有些专家级的程序员干脆从来不去覆盖clone方法，也从来不去调用它，除非拷贝数组。你必须清楚一点，对于一个专门为了继承而设计的类，如果你未能提供行为良好的受保护的（protected）clone方法，它的子类就不可能实现Cloneable接口。

第12条：考虑实现Comparable接口

与本章中讨论的其他方法不同，compareTo方法并没有在Object中声明。相反，它是Comparable接口中唯一的方法。compareTo方法不但允许进行简单的等同性比较，而且允许执行顺序比较，除此之外，它与Object的equals方法具有相似的特征，它还是个泛型。类实现了Comparable接口，就表明它的实例具有内在的排序关系（natural ordering）。为实现Comparable接口的对象数组进行排序就这么简单：

```
Arrays.sort(a);
```

对存储在集合中的Comparable对象进行搜索、计算极限值以及自动维护也同样简单。例如，下面的程序依赖于String实现了Comparable接口，它去掉了命令行参数列表中的重复参数，并按字母顺序打印出来：

```
public class WordList {
    public static void main(String[] args) {
        Set<String> s = new TreeSet<String>();
        Collections.addAll(s, args);
        System.out.println(s);
    }
}
```

一旦类实现了Comparable接口，它就可以跟许多泛型算法（generic algorithm）以及依赖于该接口的集合实现（collection implementation）进行协作。你付出很小的努力就可以获得非常强大的功能。事实上，Java平台类库中的所有值类（value classes）都实现了Comparable接口。如果你正在编写一个值类，它具有非常明显的内在排序关系，比如按字母顺序、按数值顺序或者按年代顺序，那你就应该坚决考虑实现这个接口：

```
public interface Comparable<T> {
    int compareTo(T t);
}
```

compareTo方法的通用约定与equals方法的相似：

将这个对象与指定的对象进行比较。当该对象小于、等于或大于指定对象的时候，分别返回一个负整数、零或者正整数。如果由于指定对象的类型而无法与该对象进行比较，则抛出ClassCastException异常。

在下面的说明中，符号**sgn**（*表达式*）表示数学中的**signum**函数，它根据表达式（expression）的值为负值、零和正值，分别返回－1、0或1。

- 实现者必须确保所有的x和y都满足sgn(x.compareTo(y)) == -sgn (y.compareTo(x))。(这也暗示着，当且仅当y.compareTo(x)抛出异常时，x.compareTo(y)才必须抛出异常。)
- 实现者还必须确保这个比较关系是可传递的：(x.compareTo(y) > 0 && y.compareTo(z) > 0) 暗示着x.compareTo(z) > 0。
- 最后，实现者必须确保x.compareTo(y) == 0暗示着所有的z都满足sgn(x. compareTo(z)) == sgn(y.compareTo(z))。
- 强烈建议(x.compareTo(y) == 0) == (x.equals(y))，但这并非绝对必要。一般说来，任何实现了Comparable接口的类，若违反了这个条件，都应该明确予以说明。推荐使用这样的说法："注意：该类具有内在的排序功能，但是与equals不一致。"

千万不要被上述约定中的数学关系所迷惑。如同equals约定（见第8条）一样，compareTo约定并没有它看起来的那么复杂。在类的内部，任何合理的顺序关系都可以满足compareTo约定。与equals不同的是，在跨越不同类的时候，compareTo可以不做比较：如果两个被比较的对象引用不同类的对象，compareTo可以抛出ClassCastException异常。通常，这正是compareTo在这种情况下应该做的事情，如果类设置了正确的参数，这也正是它所要做的事情。虽然以上约定并没有把跨类之间的比较排除在外，但是从Java 1.6发行版本开始，Java平台类库中就没有哪个类有支持这种特性了。

就好像违反了hashCode约定的类会破坏其他依赖于散列做法的类一样，违反compareTo约定的类也会破坏其他依赖于比较关系的类。依赖于比较关系的类包括有序集合类TreeSet和TreeMap，以及工具类Collections和Arrays，它们内部包含有搜索和排序算法。

现在我们来回顾一下compareTo约定中的条款。第一条指出，如果颠倒了两个对象引用之间的比较方向，就会发生下面的情况：如果第一个对象小于第二个对象，则第二个对象一定大于第一个对象；如果第一个对象等于第二个对象，则第二个对象一定等于第一个对象；如果第一个对象大于第二个对象，则第二个对象一定小于第一个对象。第二条指出，如果一个对象大于第二个对象，并且第二个对象又大于第三个对象，那么第一个对象一定大于第三个对象。最后一条指出，在比较时被认为相等的所有对象，它们跟别的对象做比较时一定会产生同样的结果。

这三个条款的一个直接结果是，由compareTo方法施加的等同性测试（equality test），也一定遵守相同于equals约定所施加的限制条件：自反性、对称性和传递性。因此，下面的告诫也同样适用：无法在用新的值组件扩展可实例化的类时，同时保持compareTo约定，除非愿意放弃面向对象的抽象优势（见第8条）。针对equals的权宜之计也同样适用于compareTo方法。如果你想为一个实现了Comparable接口的类增加值组件，请不要扩展这个类；而是要编写一

个不相关的类，其中包含第一个类的一个实例。然后提供一个“视图（view）”方法返回这个实例。这样既可以让你自由地在第二个类上实现compareTo方法，同时也允许它的客户端在必要的时候，把第二个类的实例视同第一个类的实例。

compareTo约定的最后一段是一个强烈的建议，而不是真正的规则，只是说明了compareTo方法施加的等同性测试，在通常情况下应该返回与equals方法同样的结果。如果遵守了这一条，那么由compareTo方法所施加的顺序关系就被认为“与**equals一致**（**consistent with equals**）”。如果违反了这条规则，顺序关系就被认为“与**equals不一致**（**inconsistent with equals**）”。如果一个类的compareTo方法施加了一个与equals方法不一致的顺序关系，它仍然能够正常工作，但是，如果一个有序集合（sorted collection）包含了该类的元素，这个集合就可能无法遵守相应集合接口（Collection、Set或Map）的通用约定。这是因为，对于这些接口的通用约定是按照equals方法来定义的，但是有序集合使用了由compareTo方法而不是equals方法所施加的等同性测试。尽管出现这种情况不会造成灾难性的后果，但是应该有所了解。

例如，考虑BigDecimal类，它的compareTo方法与equals不一致。如果你创建了一个HashSet实例，并且添加new BigDecimal（"1.0"）和new BigDecimal（"1.00"），这个集合就将包含两个元素，因为新增到集合中的两个BigDecimal实例，通过equals方法来比较时是不相等的。然而，如果你使用TreeSet而不是HashSet来执行同样的过程，集合中将只包含一个元素，因为这两个BigDecimal实例在通过compareTo方法进行比较时是相等的。（详情请参阅BigDecimal的文档。）

编写compareTo方法与编写equals方法非常相似，但也存在几处重大的差别。因为Comparable接口是参数化的，而且comparable方法是静态的类型，因此不必进行类型检查，也不必对它的参数进行类型转。如果参数的类型不合适，这个调用甚至无法编译。如果参数为null，这个调用应该抛出NullPointerException异常，并且一旦该方法试图访问它的成员时就应该抛出。

CompareTo方法中域的比较是顺序的比较，而不是等同性的比较。比较对象引用域可以是通过递归地调用compareTo方法来实现。如果一个域并没有实现Comparable接口，或者你需要使用一个非标准的排序关系，就可以使用一个显式的Comparator来代替。或者编写自己的Comparator，或者使用已有的Comparator，譬如针对第8条中CaseInsensitiveString类的这个compareTo方法使用一个已有的Comparator：

```
public final class CaseInsensitiveString
        implements Comparable<CaseInsensitiveString> {
    public int compareTo(CaseInsensitiveString cis) {
        return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    }
    ... // Remainder omitted
}
```

注意CaseInsensitiveString类实现了Comparable<CaseInsensitiveString>接口。由此可见，CaseInsensitiveString引用只能与其他的Comparable<CaseInsensitiveString>引用进行比较。在声明类去实现Comparable接口时，这是常用的模式。还要注意compareTo方法的参数是CaseInsensitiveString，而不是Object。这是上述的类声明所要求的。

比较整数型基本类型的域，可以使用关系操作符 < 和 > 。例如，浮点域用Double.compare或者Float.compare，而不用关系操作符，当应用到浮点值时，它们没有遵守compareTo的通用约定。对于数组域，则要把这些指导原则应用到每个元素上。

如果一个类有多个关键域，那么，按什么样的顺序来比较这些域是非常关键的。你必须从最关键的域开始，逐步进行到所有的重要域。如果某个域的比较产生了非零的结果（零代表相等），则整个比较操作结束，并返回该结果。如果最关键的域是相等的，则进一步比较次最关键的域，以此类推。如果所有的域都是相等的，则对象就是相等的，并返回零。下面通过第9条中的PhoneNumber类的compareTo方法来说明这种方法：

```
public int compareTo(PhoneNumber pn) {
    // Compare area codes
    if (areaCode < pn.areaCode)
        return -1;
    if (areaCode > pn.areaCode)
        return  1;
    // Area codes are equal, compare prefixes
    if (prefix < pn.prefix)
        return -1;
    if (prefix > pn.prefix)
        return  1;

    // Area codes and prefixes are equal, compare line numbers
    if (lineNumber < pn.lineNumber)
        return -1;
    if (lineNumber > pn.lineNumber)
        return  1;

    return 0;  // All fields are equal
}
```

虽然这个方法可行，但它还可以进行改进。回想一下，compareTo方法的约定并没有指定返回值的大小（magnitude），而只是指定了返回值的符号。你可以利用这一点来简化代码，或许还能提高它的运行速度：

```
public int compareTo(PhoneNumber pn) {
    // Compare area codes
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff != 0)
        return areaCodeDiff;

    // Area codes are equal, compare prefixes
    int prefixDiff = prefix - pn.prefix;
    if (prefixDiff != 0)
        return prefixDiff;
```

```
        // Area codes and prefixes are equal, compare line numbers
        return lineNumber - pn.lineNumber;
    }
```

这项技巧在这里能够工作得很好，但是用起来要非常小心。除非你确信相关的域不会为负值，或者更一般的情况：最小和最大的可能域值之差小于或等于INTEGER.MAX_VALUE (2^{31}-1)，否则就不要使用这种方法。这项技巧有时不能正常工作的原因在于，一个有符号的32位的整数还没有大到足以表达任意两个32位整数的差。如果i是一个很大的正整数（int类型），而j是一个很大的负整数（int类型），那么（i-j）将会溢出，并返回一个负值。这样就使得compareTo方法将对某些参数返回错误的结果，违反了compareTo约定的第一条和第二条。这不是一个纯粹的理论问题：它已经在实际的系统中导致了失败。这些失败可能非常难以调试，因为这样的compareTo方法对于大多数的输入值都能正常工作。

第4章

类和接口

类和接口是Java程序设计语言的核心，它们也是Java语言的基本抽象单元。Java语言提供了许多强大的基本元素，供程序员用来设计类和接口。本章阐述的一些指导原则，可以帮助你更好地利用这些元素，设计出更加有用、健壮和灵活的类和接口。

第13条：使类和成员的可访问性最小化

要区别设计良好的模块与设计不好的模块，最重要的因素在于，这个模块对于外部的其他模块而言，是否隐藏其内部数据和其他实现细节。设计良好的模块会隐藏所有的实现细节，把它的API与它的实现清晰地隔离开来。然后，模块之间只通过它们的API进行通信，一个模块不需要知道其他模块的内部工作情况。这个概念被称为*信息隐藏*（**information hiding**）或*封装*（**encapsulation**），是软件设计的基本原则之一[Parnas72]。

信息隐藏之所以非常重要有许多原因，其中大多数理由都源于这样一个事实：它可以有效地解除组成系统的各模块之间的耦合关系，使得这些模块可以独立地开发、测试、优化、使用、理解和修改。这样可以加快系统开发的速度，因为这些模块可以并行开发。它也减轻了维护的负担，因为程序员可以更快地理解这些模块，并且在调试它们的时候可以不影响其他的模块。虽然信息隐藏本身无论是对内还是对外，都不会带来更好的性能，但是它可以有效地调节性能：一旦完成一个系统，并通过剖析确定了哪些模块影响了系统的性能（见第55条），那些模块就可以被进一步优化，而不会影响到其他模块的正确性。信息隐藏提高了软件的可重用性，因为模块之间并不紧密相连，除了开发这些模块所使用的环境之外，它们在其他的环境中往往也很有用。最后，信息隐藏也降低了构建大型系统的风险，因为即使整个系统不可用，但是这些独立的模块却有可能是可用的。

Java程序设计语言提供了许多机制（facility）来协助信息隐藏。*访问控制*（**access control**）机制[JLS, 6.6]决定了类、接口和成员的*可访问性*（**accessibility**）。实体的可访问性

是由该实体声明所在的位置，以及该实体声明中所出现的访问修饰符（private、protected和public）共同决定的。正确地使用这些修饰符对于实现信息隐藏是非常关键的。

第一规则很简单：*尽可能地使每个类或者成员不被外界访问*。换句话说，应该使用与你正在编写的软件的对应功能相一致的、尽可能最小的访问级别。

对于顶层的（非嵌套的）类和接口，只有两种可能的访问级别：*包级私有的*（**package-private**）和*公有的*（**public**）。如果你用public修饰符声明了顶层类或者接口，那它就是公有的；否则，它将是包级私有的。如果类或者接口能够被做成包级私有的，它就应该被做成包级私有。通过把类或者接口做成包级私有，它实际上成了这个包的实现的一部分，而不是该包导出的API的一部分，在以后的发行版本中，可以对它进行修改、替换，或者删除，而无需担心会影响到现有的客户端程序。如果你把它做成公有的，你就有责任永远支持它，以保持它们的兼容性。

如果一个包级私有的顶层类（或者接口）只是在某一个类的内部被用到，就应该考虑使它成为唯一使用它的那个类的私有嵌套类（见第22条）。这样可以将它的可访问范围从包中的所有类缩小到了使用它的那个类。然而，降低不必要公有类的可访问性，比降低包级私有的顶层类的更重要得多：因为公有类是包的API的一部分，而包级私有的顶层类则已经是这个包的实现的一部分。

对于成员（域、方法、嵌套类和嵌套接口）有四种可能的访问级别，下面按照可访问性的递增顺序罗列出来：

- **私有的**（private）——只有在声明该成员的顶层类内部才可以访问这个成员。
- **包级私有的**（package-private）——声明该成员的包内部的任何类都可以访问这个成员。从技术上讲，它被称为“*缺省*（**default**）访问级别”，如果没有为成员指定访问修饰符，就采用这个访问级别。
- **受保护的**（protected）——声明该成员的类的子类可以访问这个成员（但有一些限制[JLS, 6.6.2]），并且，声明该成员的包内部的任何类也可以访问这个成员。
- **公有的**（public）——在任何地方都可以访问该成员。

当你仔细地设计了类的公有API之后，可能觉得应该把所有其他的成员都变成私有的。其实，只有当同一个包内的另一个类真正需要访问一个成员的时候，你才应该删除private修饰符，使该成员变成包级私有的。如果你发现自己经常要做这样的事情，就应该重新检查你的系统设计，看看是否另一种分解方案所得到的类，与其他类之间的耦合度会更小。也就是说，私有成员和包级私有成员都是一个类的实现中的一部分，一般不会影响它的导出的API。然而，

如果这个类实现了Serializable接口（见第74和75条），这些域就有可能会被“泄漏（leak）”到导出的API中。

对于公有类的成员，当访问级别从包级私有变成保护级别时，会大大增强可访问性。受保护的成员是类的导出的API的一部分，必须永远得到支持。导出的类的受保护成员也代表了该类对于某个实现细节的公开承诺（见第17条）。受保护的成员应该尽量少用。

有一条规则限制了降低方法的可访问性的能力。如果方法覆盖了超类中的一个方法，子类中的访问级别就不允许低于超类中的访问级别[JLS, 8.4.8.3]。这样可以确保任何可使用超类的实例的地方也都可以使用子类的实例。如果你违反了这条规则，那么当你试图编译该子类的时候，编译器就会产生一条错误消息。这条规则有种特殊的情形：如果一个类实现了一个接口，那么接口中所有的类方法在这个类中也都必须被声明为公有的。之所以如此，是因为接口中的所有方法都隐含着公有访问级别[JLS，9.1.5]。

为了便于测试，你可以试着使类、接口或者成员变得更容易访问。这么做在一定程度上来说是好的。为了测试而将一个公有类的私有成员变成包级私有的，这还可以接受，但是要将访问级别提高到超过它，这就无法接受了。换句话说，不能为了测试，而将类、接口或者成员变成包的导出的API的一部分。幸运的是，也没有必要这么做，因为可以让测试作为被测试的包的一部分来运行，从而能够访问它的包级私有的元素。

实例域决不能是公有的（见第14条）。如果域是非final的，或者是一个指向可变对象的final引用，那么一旦使这个域成为公有的，就放弃了对存储在这个域中的值进行限制的能力；这意味着，你也放弃了强制这个域不可变的能力。同时，当这个域被修改的时候，你也失去了对它采取任何行动的能力。因此，*包含公有可变域的类并不是线程安全的*。即使域是final的，并且引用不可变的对象，当把这个域变成公有的时候，也就放弃了“切换到一种新的内部数据表示法”的灵活性。

同样的建议也适用于静态域，只是有一种例外情况。假设常量构成了类提供的整个抽象中的一部分，可以通过公有的静态final域来暴露这些常量。按惯例，这种域的名称由大写字母组成，单词之间用下划线隔开（见第56条）。很重要的一点是，这些域要么包含基本类型的值，要么包含指向不可变对象的引用（见第15条）。如果final域包含可变对象的引用，它便具有非final域的所有缺点。虽然引用本身不能被修改，但是它所引用的对象却可以被修改——这会导致灾难性的后果。

*注意，长度非零的数组总是可变的，所以，类具有公有的静态final数组域，或者返回这种域的访问方法，这几乎总是错误的。*如果类具有这样的域或者访问方法，客户端将能够修改数组中的内容。这是安全漏洞的一个常见根源：

```
// Potential security hole!
public static final Thing[] VALUES = { ... };
```

要注意，许多IDE会产生返回指向私有数组域的引用的访问方法，这样就会产生这个问题。修正这个问题有两种方法。可以使公有数组变成私有的，并增加一个公有的不可变列表：

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
   Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

另一种方法是，可以使数组变成私有的，并添加一个公有方法，它返回私有数组的一个备份：

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final Thing[] values() {
    return PRIVATE_VALUES.clone();
}
```

要在这两种方法之间做出选择，得考虑客户端可能怎么处理这个结果。哪种返回类型会更加方便？哪种会得到更好的性能？

总而言之，你应该始终尽可能地降低可访问性。你在仔细地设计了一个最小的公有API之后，应该防止把任何散乱的类、接口和成员变成API的一部分。除了公有静态final域的特殊情形之外，公有类都不应该包含公有域。并且要确保公有静态final域所引用的对象都是不可变的。

第14条：在公有类中使用访问方法而非公有域

有时候，可能会编写一些退化类（degenerate classes），没有什么作用，只是用来集中实例域：

```
// Degenerate classes like this should not be public!
class Point {
    public double x;
    public double y;
}
```

由于这种类的数据域是可以被直接访问的，这些类没有提供封装（encapsulation）的功能（见第13条）。如果不改变API，就无法改变它的数据表示法，也无法强加任何约束条件；当域被访问的时候，无法采取任何辅助的行动。坚持面向对象程序设计的程序员对这种类深恶痛绝，认为应该用包含私有域和公有访问方法（getter）的类代替。对于可变的类来说，应该用包含私有域和公有设值方法（setter）的类代替：

```
// Encapsulation of data by accessor methods and mutators
class Point {
    private double x;
    private double y;

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

    public double getX() { return x; }
    public double getY() { return y; }

    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
}
```

毫无疑问，说到公有类的时候，坚持面向对象程序设计思想的看法是正确的：*如果类可以在它所在的包的外部进行访问，就提供访问方法*，以保留将来改变该类的内部表示法的灵活性。如果公有类暴露了它的数据域，要想在将来改变其内部表示法是不可能的，因为公有类的客户端代码已经遍布各处了。

然而，如果类是包级私有的，或者是私有的嵌套类，直接暴露它的数据域并没有本质的错误——假设这些数据域确实描述了该类所提供的抽象。这种方法比访问方法的做法更不会产生视觉混乱，无论是在类定义中，还是在使用该类的客户端代码中。虽然客户端代码与该类的内部表示法紧密相连，但是这些代码被限定在包含该类的包中。如有必要，不改变包之外的任何代码而只改变内部数据表示法也是可以的。在私有嵌套类的情况下，改变的作用范围被进一步限制在外围类中。

Java平台类库中有几个类违反了“公有类不应该直接暴露数据域”的告诫。显著的例子包括java.awt包中的Point和Dimension类。它们是不值得仿效的例子，相反，这些类应该被当作反面的警告示例。正如第55条中所讲述的，决定暴露Dimension类的内部数据造成了严重的性能问题，而且，这个问题至今依然存在。

让公有类直接暴露域虽然从来都不是种好办法，但是如果域是不可变的，这种做法的危害就比较小一些。如果不改变类的API，就无法改变这种类的表示法，当域被读取的时候，你也无法采取辅助的行动，但是可以强加约束条件。例如，这个类确保了每个实例都表示一个有效的时间：

```
// Public class with exposed immutable fields - questionable
public final class Time {
    private static final int HOURS_PER_DAY    = 24;
    private static final int MINUTES_PER_HOUR = 60;

    public final int hour;
    public final int minute;

    public Time(int hour, int minute) {
        if (hour < 0 || hour >= HOURS_PER_DAY)
            throw new IllegalArgumentException("Hour: " + hour);
        if (minute < 0 || minute >= MINUTES_PER_HOUR)
            throw new IllegalArgumentException("Min: " + minute);
        this.hour = hour;
        this.minute = minute;
    }
    ... // Remainder omitted
}
```

总之，公有类永远都不应该暴露可变的域。虽然还是有问题，但是让公有类暴露不可变的域其危害比较小。但是，有时候会需要用包级私有的或者私有的嵌套类来暴露域，无论这个类是可变还是不可变的。

第15条：使可变性最小化

不可变类只是其实例不能被修改的类。每个实例中包含的所有信息都必须在创建该实例的时候就提供，并在对象的整个生命周期（lifetime）内固定不变。Java平台类库中包含许多不可变的类，其中有String、基本类型的包装类、BigInteger和BigDecimal。存在不可变的类有许多理由：不可变的类比可变类更加易于设计、实现和使用。它们不容易出错，且更加安全。

为了使类成为不可变，要遵循下面五条规则：

1. **不要提供任何会修改对象状态的方法**（也称为mutator）。[⊖]

2. **保证类不会被扩展**。这样可以防止粗心或者恶意的子类假装对象的状态已经改变，从而破坏该类的不可变行为。为了防止子类化，一般做法是使这个类成为final的，但是后面我们还会讨论到其他的做法。

3. **使所有的域都是final的**。通过系统的强制方式，这可以清楚地表明你的意图。而且，如果一个指向新创建实例的引用在缺乏同步机制的情况下，从一个线程被传递到另一个线程，就必需确保正确的行为，正如内存模型（**memory model**）中所述[JLS，17.5；Goetz06 16]。

4. **使所有的域都成为私有的**。这样可以防止客户端获得访问被域引用的可变对象的权限，并防止客户端直接修改这些对象。虽然从技术上讲，允许不可变的类具有公有的final域，只要这些域包含基本类型的值或者指向不可变对象的引用，但是不建议这样做，因为这样会使得在以后的版本中无法再改变内部的表示法（见第13条）。

5. **确保对于任何可变组件的互斥访问**。如果类具有指向可变对象的域，则必须确保该类的客户端无法获得指向这些对象的引用。并且，永远不要用客户端提供的对象引用来初始化这样的域，也不要从任何访问方法（accessor）中返回该对象引用。在构造器、访问方法和readObject方法（见第76条）中请使用保护性拷贝（**defensive copy**）技术（见第39条）。

前面条目中的许多例子都是不可变的，其中一个例子是第9条中的PhoneNumber，它针对每个属性都有访问方法（accessor），但是没有对应的设值方法（mutator）。下面是个稍微复杂一点的例子：

```
public final class Complex {
    private final double re;
    private final double im;

    public Complex(double re, double im) {
```

⊖ 即改变对象属性的方法。——编辑注。

```
        this.re = re;
        this.im = im;
    }

    // Accessors with no corresponding mutators
    public double realPart()      { return re; }
    public double imaginaryPart() { return im; }

    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }

    public Complex subtract(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex multiply(Complex c) {
        return new Complex(re * c.re - im * c.im,
                           re * c.im + im * c.re);
    }

    public Complex divide(Complex c) {
        double tmp = c.re * c.re + c.im * c.im;
        return new Complex((re * c.re + im * c.im) / tmp,
                           (im * c.re - re * c.im) / tmp);
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Complex))
            return false;
        Complex c = (Complex) o;

        // See page 43 to find out why we use compare instead of ==
        return Double.compare(re, c.re) == 0 &&
               Double.compare(im, c.im) == 0;
    }
    @Override public int hashCode() {
        int result = 17 + hashDouble(re);
        result = 31 * result + hashDouble(im);
        return result;
    }

    private int hashDouble(double val) {
        long longBits = Double.doubleToLongBits(re);
        return (int) (longBits ^ (longBits >>> 32));
    }

    @Override public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}
```

这个类表示一个复数（**complex number**，具有实部和虚部）。除了标准的Object方法之外，它还提供了针对实部和虚部的访问方法，以及4种基本的算术运算：加法、减法、乘法和除法。注意这些算术运算是如何创建并返回新的Complex实例，而不是修改这个实例。大多数重要的不可变类都使用了这种模式。它被称为*函数的*（**functional**）做法，因为这些方法返回了一个函数的结果，这些函数对操作数进行运算但并不修改它。与之相对应的更常见的是*过程的*

（**procedural**）或者命令式的（**imperative**）做法，使用这些方式时，将一个过程作用在它们的操作数上，会导致它的状态发生改变。

如果你对函数方式的做法还不太熟悉，可能会觉得它显得不太自然，但是它带来了不可变性，具有许多优点。不可变对象比较简单。不可变对象可以只有一种状态，即被创建时的状态。如果你能够确保所有的构造器都建立了这个类的约束关系，就可以确保这些约束关系在整个生命周期内永远不再发生变化，你和使用这个类的程序员都无需再做额外的工作来维护这些约束关系。另一方面，可变的对象可以有任意复杂的状态空间。如果文档中没有对mutator方法所执行的状态转换提供精确的描述，要可靠地使用一个可变类是非常困难的，甚至是不可能的。

不可变对象本质上是线程安全的，它们不要求同步。当多个线程并发访问这样的对象时，它们不会遭到破坏。这无疑是获得线程安全最容易的办法。实际上，没有任何线程会注意到其他线程对于不可变对象的影响。所以，不可变对象可以被自由地共享。不可变类应该充分利用这种优势，鼓励客户端尽可能地重用现有的实例。要做到这一点，一个很简便的办法就是，对于频繁用到的值，为它们提供公有的静态final常量。例如，Complex类有可能会提供下面的常量：

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE  = new Complex(1, 0);
public static final Complex I    = new Complex(0, 1);
```

这种方法可以被进一步扩展。不可变的类可以提供一些静态工厂（见第1条），它们把频繁被请求的实例缓存起来，从而当现有实例可以符合请求的时候，就不必创建新的实例。所有基本类型的包装类和BigInteger都有这样的静态工厂。使用这样的静态工厂也使得客户端之间可以共享现有的实例，而不用创建新的实例，从而降低内存占用和垃圾回收的成本。在设计新的类时，选择用静态工厂代替公有的构造器可以让你以后有添加缓存的灵活性，而不必影响客户端。

“不可变对象可以被自由地共享”导致的结果是，永远也不需要进行保护性拷贝（见第39条）。实际上，你根本无需做任何拷贝，因为这些拷贝始终等于原始的对象。因此，你不需要，也不应该为不可变的类提供clone方法或者拷贝构造器（**copy constructor**，见第11条）。这一点在Java平台的早期并不好理解，所以String类仍然具有拷贝构造器，但是应该尽量少用它（见第5条）。

不仅可以共享不可变对象，甚至也可以共享它们的内部信息。例如，BigInteger类内部使用了符号数值表示法。符号用一个int类型的值来表示，数值则用一个int数组表示。negate方法产生一个新的BigInteger，其中数值是一样的，符号则是相反的。它并不需要拷贝数组；新

建的BigInteger也指向原始实例中的同一个内部数组。

不可变对象为其他对象提供了大量的构件（**building blocks**），无论是可变的还是不可变的对象。如果知道一个复杂对象内部的组件对象不会改变，要维护它的不变性约束是比较容易的。这条原则的一种特例在于，不可变对象构成了大量的映射键（map key）和集合元素（set element）；一旦不可变对象进入到映射（map）或者集合（set）中，尽管这破坏了映射或者集合的不变性约束，但是也不用担心它们的值会发生变化。

不可变类真正唯一的缺点是，对于每个不同的值都需要一个单独的对象。创建这种对象的代价可能很高，特别是对于大型对象的情形。例如，假设你有一个上百万位的BigInteger，想要改变它的低位：

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

flipBit方法创建了一个新的BigInteger实例，也有上百万位长，它与原来的对象只差一位不同。这项操作所消耗的时间和空间与BigInteger的成正比。我们拿它与java.util.BitSet进行比较。与BigInteger类似，BitSet代表一个任意长度的位序列，但是与BigInteger不同的是，BitSet是可变的。BitSet类提供了一个方法，允许在固定时间（constant time）内改变此“百万位”实例中单个位的状态。

如果你执行一个多步骤的操作，并且每个步骤都会产生一个新的对象，除了最后的结果之外其他的对象最终都会被丢弃，此时性能问题就会显露出来。处理这种问题有两种办法。第一种办法，先猜测一下会经常用到哪些多步骤的操作，然后将它们作为基本类型提供。如果某个多步骤操作已经作为基本类型提供，不可变的类就可以不必在每个步骤单独创建一个对象。不可变的类在内部可以更加灵活。例如，BigInteger有一个包级私有的可变“配套类（companing class）”，它的用途是加速诸如“模指数（modular exponentiation）”这样的多步骤操作。由于前面提到的诸多原因，使用可变的配套类比使用BigInteger要困难得多，但幸运的是，你并不需要这样做。因为BigInteger的实现者已经替你完成了所有的困难工作。

如果能够精确地预测出客户端将要在不可变的类上执行哪些复杂的多阶段操作，这种包级私有的可变配套类的方法就可以工作得很好。如果无法预测，最好的办法是提供一个公有的可变配套类。在Java平台类库中，这种方法的主要例子是String类，它的可变配套类是StringBuilder（和基本上已经废弃的StringBuffer）。可以这样认为，在特定的环境下，相对于BigInteger而言，BitSet同样扮演了可变配套类的角色。

现在你已经知道了如何构建不可变的类，并且了解了不可变性的优点和缺点，现在我们来讨论其他的一些设计方案。前面提到过，为了确保不可变性，类绝对不允许自身被子类化。除了“使类成为final的”这种方法之外，还有另外一种更加灵活的办法可以做到这一点。让

不可变的类变成final的另一种办法就是，让类的所有构造器都变成私有的或者包级私有的，并添加公有的静态工厂（**static factory**）来代替公有的构造器（见第1条）。

为了具体说明这种方法，下面以Complex为例，看看如何使用这种方法：

```
// Immutable class with static factories instead of constructors
public class Complex {
    private final double re;
    private final double im;

    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }

    ... // Remainder unchanged
}
```

虽然这种方法并不常用，但它经常是最好的替代方法。它最灵活，因为它允许使用多个包级私有的实现类。对于处在它的包外部的客户端而言，不可变的类实际上是final的，因为不可能把来自另一个包的类、缺少公有的或受保护的构造器的类进行扩展。除了允许多个实现类的灵活性之外，这种方法还使得有可能通过改善静态工厂的对象缓存能力，在后续的发行版本中改进该类的性能。

静态工厂与构造器相比具有许多其他的优势，正如在第1条中所讨论的。例如，假设你希望提供一种“基于极坐标创建复数”的方式。如果使用构造器来实现这样的功能，可能会使得这个类很零乱，因为这样的构造器与已用的构造器Complex(double, double)具有相同的签名。通过静态工厂，这很容易做到。只需添加第二个静态工厂，并且工厂的名字清楚地表明了它的功能即可：

```
public static Complex valueOfPolar(double r, double theta) {
    return new Complex(r * Math.cos(theta),
                       r * Math.sin(theta));
}
```

当BigInteger和BigDecimal刚被编写出来的时候，对于“不可变的类必须为final的”还没有得到广泛地理解，所以它们的所有方法都有可能会被覆盖。遗憾的是，为了保持向后兼容，这个问题一直无法得以修正。如果你在编写一个类，它的安全性依赖于（来自不可信客户端的）BigInteger或者BigDecimal参数的不可变性，就必须进行检查，以确定这个参数是否为“真正的”的BigInteger或者BigDecimal，而不是不可信任子类的实例。如果是后者的话，就必须在假设它可能是可变的前提下对它进行保护性拷贝（见第39条）：

```
public static BigInteger safeInstance(BigInteger val) {
```

```
        if (val.getClass() != BigInteger.class)
            return new BigInteger(val.toByteArray());
        return val;
    }
```

本条目开头处关于不可变类的诸多规则指出，没有方法会修改对象，并且它的所有域都必须是final的。实际上，这些规则比真正的要求更强硬了一点，为了提高性能可以有所放松。事实上应该是这样：没有一个方法能够对对象的状态产生外部可见（**externally visible**）的改变。然而，许多不可变的类拥有一个或者多个非final的域，它们在第一次被请求执行这些计算的时候，把一些开销昂贵的计算结果缓存在这些域中。如果将来再次请求同样的计算，就直接返回这些缓存的值，从而节约了重新计算所需要的开销。这种技巧可以很好地工作，因为对象是不可变的，它的不可变性保证了这些计算如果被再次执行，就会产生同样的结果。

例如，PhoneNumber类的hashCode方法（见第9条）在第一次被调用的时候，计算出散列码，然后把它缓存起来，以备将来被再次调用时使用。这种方法是*延迟初始化*（**lazy initialization**）（见第71条）的一个例子，String类也用到了。

有关序列化功能的一条告诫有必要在这里提出来。如果你选择让自己的不可变类实现Serializable接口，并且它包含一个或者多个指向可变对象的域，就必须提供一个显式的readObject或者readResolve方法，或者使用ObjectOutputStream.writeUnshared和ObjectInputStream.readUnshared方法，即使默认的序列化形式是可以接受的，也是如此。否则攻击者可能从不可变的类创建可变的实例。这个话题的详细内容请参见第76条。

总之，*坚决不要为每个get方法编写一个相应的set方法。除非有很好的理由要让类成为可变的类，否则就应该是不可变的。*不可变的类有许多优点，唯一缺点是在特定的情况下存在潜在的性能问题。你应该总是使一些小的值对象，比如PhoneNumber和Complex，成为不可变的（在Java平台类库中，有几个类如java.util.Date和java.awt.Point，它们本应该是不可变的，但实际上却不是）。你也应该认真考虑把一些较大的值对象做成不可变的，例如String和BigInteger。只有当你确认有必要实现令人满意的性能时（见第55条），才应该为不可变的类提供公有的可变配套类。

*对于有些类而言，其不可变性是不切实际的。如果类不能被做成是不可变的，仍然应该尽可能地限制它的可变性。*降低对象可以存在的状态数，可以更容易地分析该对象的行为，同时降低出错的可能性。因此，*除非有令人信服的理由要使域变成是非**final**的，否则要使每个域都是**final**的。*

构造器应该创建完全初始化的对象，并建立起所有的约束关系。不要在构造器或者静态工厂之外再提供公有的初始化方法，除非有令人信服的理由必须这么做。同样地，也不应该提供“重新初始化”方法（它使得对象可以被重用，就好像这个对象是由另一不同的初始状态

构造出来的一样）。与所增加的复杂性相比，“重新初始化”方法通常并没有带来太多的性能优势。

可以通过TimerTask类来说明这些原则。它是可变的，但是它的状态空间被有意地设计得非常小。你可以创建一个实例，对它进行调度使它执行起来，也可以随意地取消它。一旦一个定时器任务（timer task）已经完成，或者已经被取消，就不可能再对它重新调度。

最后值得注意的一点与本条目中的Complex类有关。这个例子只是被用来演示不可变性的，它不是一个工业强度（即产品级）的复数实现。它对复数乘法和除法使用标准的计算公式，会进行不正确的舍入，并对复数NaN和无穷大没有提供很好的语义[Kahan91, Smith62, Thomas94]。

第16条：复合优先于继承

继承（inheritance）是实现代码重用的有力手段，但它并非永远是完成这项工作的最佳工具。使用不当会导致软件变得很脆弱。在包的内部使用继承是非常安全的，在那里，子类和超类的实现都处在同一个程序员的控制之下。对于专门为了继承而设计、并且具有很好的文档说明的类来说（见第17条），使用继承也是非常安全的。然而，对普通的具体类（concrete class）进行跨越包边界的继承，则是非常危险的。提示一下，本书使用“继承”一词，含义是*实现继承*（**implementation inheritance**，当一个类扩展另一个类的时候）。本条目中讨论的问题并不适用于*接口继承*（**interface inheritance**，当一个类实现一个接口的时候，或者当一个接口扩展另一个接口的时候）。

与方法调用不同的是，继承打破了*封装性*[**Snyder86**]。换句话说，子类依赖于其超类中特定功能的实现细节。超类的实现有可能会随着发行版本的不同而有所变化，如果真的发生了变化，子类可能会遭到破坏，即使它的代码完全没有改变。因而，子类必须要跟着其超类的更新而演变，除非超类是专门为了扩展而设计的，并且具有很好的文档说明。

为了说明得更加具体一点，我们假设有一个程序使用了HashSet。为了调优该程序的性能，需要查询HashSet，看一看自从它被创建以来曾经添加了多少个元素（不要与它当前的元素混淆起来，元素数目会随着元素的删除而递减）。为了提供这种功能，我们得编写一个HashSet变量，它记录下试图插入的元素数量，并针对该计数值导出一个访问方法。HashSet类包含两个可以增加元素的方法：add和addAll，因此这两个方法都要被覆盖：

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet<E> extends HashSet<E> {
    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

这个类看起来非常合理，但是它并不能正常工作。假设我们创建了一个实例，并利用addAll方法添加了三个元素：

```
InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>();
s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
```

这时候，我们期望getAddCount方法将会返回3，但是它实际上返回的是6。哪里出错了呢？在HashSet的内部，addAll方法是基于它的add方法来实现的，即使HashSet的文档中并没有说明这样的实现细节，这也是合理的。InstrumentedHashSet中的addAll方法首先给addCount增加3，然后利用supper.addAll来调用HashSet的addAll实现。然后又依次调用到被InstrumentedHashSet覆盖了的add方法，每个元素调用一次。这三次调用又分别给addCount加了1，所以，总共增加了6：通过addAll方法增加的每个元素都被计算了两次。

我们只要去掉被覆盖的addAll方法，就可以“修正”这个子类。虽然这样得到的类可以正常工作，但是，它的功能正确性则需要依赖于这样的事实：HashSet的addAll方法是在它的add方法上实现的。这种“自用性（self-use）”是实现细节，不是承诺，不能保证在Java平台的所有实现中都保持不变，不能保证随着发行版本的不同而不发生变化。因此，这样得到的InstrumentedHashSet类将是非常脆弱的。

稍微好一点的做法是，覆盖addAll方法来遍历指定的集合，为每个元素调用一次add方法。这样做可以保证得到正确的结果，不管HashSet的addAll方法是否是在add方法的基础上实现，因为HashSet的addAll实现将不会再被调用到。然而，这项技术并没有解决所有的问题，它相当于重新实现了超类的方法，这些超类的方法可能是自用的（self-use），也可能不是自用的，这种方法很困难，也非常耗时，并且容易出错。此外，这样做并不总是可行的，因为无法访问对于子类来说的私有域，所以有些方法就无法实现。

导致子类脆弱的一个相关的原因是，它们的超类在后续的发行版本中可以获得新的方法。假设一个程序的安全性依赖于这样的事实：所有被插入到某个集合中的元素都满足某个先决条件。下面的做法就可以确保这一点：对集合进行子类化，并覆盖所有能够添加元素的方法，以便确保在加入每个元素之前它是满足这个先决条件的。如果在后续的发行版本中，超类中没有增加能插入元素的新方法，这种做法就可以正常工作。然而，一旦超类增加了这样的新方法，则很可能仅仅由于调用了这个未被子类覆盖的新方法，而将“非法的”元素添加到子类的实例中。这不是个纯粹的理论问题。在把Hashtable和Vector加入到Collections Framework中的时候，就修正了几个这类性质的安全漏洞。

上面这两个问题都来源于覆盖（overriding）动作。如果在扩展一个类的时候，仅仅是增加新的方法，而不覆盖现有的方法，你可能会认为这是安全的。虽然这种扩展方式比较安全

一些，但是也并非完全没有风险。如果超类在后续的发行版本中获得了一个新的方法，并且不幸的是，你给子类提供了一个签名相同但返回类型不同的方法，那么这样的子类将无法通过编译[JLS, 8.4.6.3]。如果给子类提供的方法带有与新的超类方法完全相同的签名和返回类型，实际上就覆盖了超类中的方法，因此又回到上述的两个问题上去了。此外，你的方法是否能够遵守新的超类方法的约定，这也是很值得怀疑的，因为当你在编写子类方法的时候，这个约定根本没有面世。

幸运的是，有一种办法可以避免前面提到的所有问题。不用扩展现有的类，而是在新的类中增加一个私有域，它引用现有类的一个实例。这种设计被称做“复合（**composition**）”，因为现有的类变成了新类的一个组件。新类中的每个实例方法都可以调用被包含的现有类实例中对应的方法，并返回它的结果。这被称为转发（**forwarding**），新类中的方法被称为转发方法（**forwarding method**）。这样得到的类将会非常稳固，它不依赖于现有类的实现细节。即使现有的类添加了新的方法，也不会影响新的类。为了进行更具体的说明，请看下面的例子，它用复合/转发的方法来代替InstrumentedHashSet类。注意这个实现分为两部分：类本身和可重用的转发类（**forwarding class**），包含了所有的转发方法，没有其他方法。

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}

// Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }

    public void clear()               { s.clear();            }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()          { return s.isEmpty();   }
    public int size()                 { return s.size();      }
    public Iterator<E> iterator()     { return s.iterator();  }
    public boolean add(E e)           { return s.add(e);      }
    public boolean remove(Object o)   { return s.remove(o);   }
    public boolean containsAll(Collection<?> c)
                                      { return s.containsAll(c); }
    public boolean addAll(Collection<? extends E> c)
```

```
                                { return s.addAll(c);       }
    public boolean removeAll(Collection<?> c)
                                { return s.removeAll(c);    }
    public boolean retainAll(Collection<?> c)
                                { return s.retainAll(c);    }
    public Object[] toArray()          { return s.toArray();  }
    public <T> T[] toArray(T[] a)      { return s.toArray(a); }
    @Override public boolean equals(Object o)
                                { return s.equals(o);   }
    @Override public int hashCode()    { return s.hashCode(); }
    @Override public String toString() { return s.toString(); }
}
```

Set接口的存在使得InstrumentedSet类的设计成为可能，因为Set接口保存了HashSet类的功能特性。除了获得健壮性之外，这种设计也带来了格外的灵活性。InstrumentedSet类实现了Set接口，并且拥有单个构造器，它的参数也是Set类型。从本质上讲，这个类把一个Set转变成了另一个Set，同时增加了计数的功能。前面提到的基于继承的方法只适用于单个具体的类，并且对于超类中所支持的每个构造器都要求有一个单独的构造器，与此不同的是，这里的包装类（wrapper class）可以被用来包装任何Set实现，并且可以结合任何先前存在的构造器一起工作。例如：

```
Set<Date> s = new InstrumentedSet<Date>(new TreeSet<Date>(cmp));
Set<E> s2 = new InstrumentedSet<E>(new HashSet<E>(capacity));
```

InstrumentedSet类甚至也可以用来临时替换一个原本没有计数特性的Set实例：

```
static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<Dog>(dogs);
    ... // Within this method use iDogs instead of dogs
}
```

因为每一个InstrumentedSet实例都把另一个Set实例包装起来了，所以InstrumentedSet类被称做*包装类*（**wrapper class**）。这也正是Decorator模式[Gamma95, p.175]，因为InstrumentedSet类对一个集合进行了修饰，为它增加了计数特性。有时候，复合和转发的结合也被错误地称为“*委托*（**delegation**）”。从技术的角度而言，这不是委托，除非包装对象把自身传递给被包装的对象[Gamma95, p. 20]。

包装类几乎没有什么缺点。需要注意的一点是，包装类不适合用在*回调框架*（**callback framework**）中；在回调框架中，对象把自身的引用传递给其他的对象，用于后续的调用（“回调”）。因为被包装起来的对象并不知道它外面的包装对象，所以它传递一个指向自身的引用（this），回调时避开了外面的包装对象。这被称为SELF问题[Lieberman86]。有些人担心转发方法调用所带来的性能影响，或者包装对象导致的内存占用。在实践中，这两者都不会造成很大的影响。编写转发方法倒是有点琐碎，但是只需要给每个接口编写一次构造器，转发类则可以通过包含接口的包替你提供。

只有当子类真正是超类的子类型（**subtype**）时，才适合用继承。换句话说，对于两个类A和B，只有当两者之间确实存在“is-a”关系的时候，类B才应该扩展类A。如果你打算让类B扩展类A，就应该问问自己：每个B确实也是A吗？如果你不能够确定这个问题的答案是肯定的，那么B就不应该扩展A。如果答案是否定的，通常情况下，B应该包含A的一个私有实例，并且暴露一个较小的、较简单的API：A本质上不是B的一部分，只是它的实现细节而已。

在Java平台类库中，有许多明显违反这条原则的地方。例如，栈（stack）并不是向量（vector），所以Stack不应该扩展Vector。同样地，属性列表也不是散列表，所以Properties不应该扩展Hashtable。在这两种情况下，复合模式才是恰当的。

如果在适合于使用复合的地方使用了继承，则会不必要地暴露实现细节。这样得到的API会把你限制在原始的实现上，永远限定了类的性能。更为严重的是，由于暴露了内部的细节，客户端就有可能直接访问这些内部细节。这样至少会导致语义上的混淆。例如，如果p指向Properties实例，那么p.getProperty(key) 就有可能产生与p.get(key)不同的结果：前者考虑了默认的属性表，而后者是继承自Hashtable的，它则没有考虑默认属性列表。最严重的是，客户有可能直接修改超类，从而破坏子类的约束条件。在Properties的情形中，设计者的目标是，只允许字符串作为键（key）和值（value），但是直接访问底层的Hashtable就可以违反这种约束条件。一旦违反了约束条件，就不可能再使用Properties API的其他部分（load和store）了。等到发现这个问题时，要改正它已经太晚了，因为客户端依赖于使用非字符串的键和值了。

在决定使用继承而不是复合之前，还应该问自己最后一组问题。对于你正试图扩展的类，它的API中有没有缺陷呢？如果有，你是否愿意把那些缺陷传播到类的API中？继承机制会把超类API中的所有缺陷传播到子类中，而复合则允许设计新的API来隐藏这些缺陷。

简而言之，继承的功能非常强大，但是也存在诸多问题，因为它违背了封装原则。只有当子类和超类之间确实存在子类型关系时，使用继承才是恰当的。即便如此，如果子类和超类处在不同的包中，并且超类并不是为了继承而设计的，那么继承将会导致脆弱性（fragility）。为了避免这种脆弱性，可以用复合和转发机制来代替继承，尤其是当存在适当的接口可以实现包装类的时候。包装类不仅比子类更加健壮，而且功能也更加强大。

第17条：要么为继承而设计，并提供文档说明，要么就禁止继承

第16条提醒我们，对于不是为了继承而设计、并且没有文档说明的“外来”类进行子类化是多么危险。那么对于专门为了继承而设计并且具有良好文档说明的类而言，这又意味着什么呢？

首先，该类的文档必须精确地描述覆盖每个方法所带来的影响。换句话说，*该类必须有文档说明它可覆盖*（**overridable**）*的方法的自用性*（**self-use**）。对于每个公有的或受保护的方法或者构造器，它的文档必须指明该方法或者构造器调用了哪些可覆盖的方法，是以什么顺序调用的，每个调用的结果又是如何影响后续的处理过程的（所谓可覆盖（overridable）的方法，是指非final的，公有的或受保护的）。更一般地，类必须在文档中说明，在哪些情况下它会调用可覆盖的方法。例如，后台的线程或者静态的初始化器（initializer）可能会调用这样的方法。

按惯例，如果方法调用到了可覆盖的方法，在它的文档注释的末尾应该包含关于这些调用的描述信息。这段描述信息要以这样的句子开头：“This implementation. （该实现……）”。这样的句子不应该被认为是在表明该行为可能会随着版本的变迁而改变。它意味着这段描述关注该方法的内部工作情况。下面是个示例，摘自java.util.AbstractCollection的规范：

```
public boolean remove(Object o)
```

> Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element e such that (o==null ? e==null: o.equals(e)), if the collection contains one or more such elements. Returns true if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).
>
> This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method. Note that this implementation throws an `UnsupportedOperationException` if the iterator returned by this collection's `iterator` method does not implement the `remove` method.

（如果这个集合中存在指定的元素，就从中删除该指定元素中的单个实例（这是项可选的操作）。更一般地，如果集合中包含一个或者多个这样的元素e，就从中删除这种元素，以便(o==null ? e==null: o.equals(e))。如果集合中包含指定的元素，就返回true（如果调用最终改变了集合，也一样）。

该实现遍历整个集合来查找指定的元素。如果它找到该元素，将会利用迭代器的remove方法将之从集合中删除。注意，如果由该集合的iterator方法返回的迭代器没有实现remove方法，该实现就会抛出UnsupportedOperationException。)

该文档清楚地说明了，覆盖iterator方法将会影响remove方法的行为。而且，它确切地描述了iterator方法返回的Iterator的行为将会怎样影响remove方法的行为。与此相反的是，在第16条的情形中，程序员在子类化HashSet的时候，并无法说明覆盖add方法是否会影响addAll方法的行为。

关于程序文档有句格言：好的API文档应该描述一个给定的方法做了什么工作，而不是描述它是如何做到的。那么，上面这种做法是否违背了这句格言呢？是的，它确实违背了！这正是继承破坏了封装性所带来的不幸后果。所以，为了设计一个类的文档，以便它能够被安全地子类化，你必须描述清楚那些有可能未定义的实现细节。

为了继承而进行的设计不仅仅涉及自用模式的文档设计。为了使程序员能够编写出更加有效的子类，而无需承受不必要的痛苦，类必须通过某种形式提供适当的钩子（**hook**），以便能够进入到它的内部工作流程中，这种形式可以是精心选择的受保护的（**protected**）方法，也可以是受保护的域，后者比较少见。例如，考虑java.util.AbstractList中的removeRange方法：

```
protected void removeRange(int fromIndex, int toIndex)
```

Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the `ArrayList` by (`toIndex-fromIndex`) elements. (`If toIndex==fromIndex`, this operation has no effect.)

This method is called by the `clear` operation on this list and its sublists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the `clear` operation on this list and its sublists.

This implementation gets a list iterator positioned before `fromIndex` and repeatedly calls `ListIterator.next` followed by `ListIterator.remove`, until the entire range has been removed. Note: If `ListIterator.remove` requires linear time, this implementation requires quadratic time.

Parameters:

`fromIndex` index of first element to be removed.

`toIndex` index after last element to be removed.

（从列表中删除所有索引处于fromIndex（含）和toIndex（不含）之间的元素。将所有符合条件的元素移到左边（减小索引）。这一调用将从ArrayList中删除（toIndex-fromIndex）之间的元素。（如果toIndex == fromIndex，这项操作就无效。）

这个方法是通过clear操作在这个列表及其子列表中调用的。覆盖这个方法来利用列表实现的内部信息，可以充分地改善这个列表及其子列表中的clear操作的性能。

这项实现获得了一个处在fromIndex之前的列表迭代器，并依次地重复调用ListIterator.remove和ListIterator.next，直到整个范围都被移除为止。注意：如果ListIterator.remove需要线性的时间，该实现就需要平方级的时间。

参数：

`fromIndex` 要移除的第一个元素的索引

`toIndex` 要移除的最后一个元素之后的索引）

这个方法对于List实现的最终用户并没有意义。提供该方法的唯一目的在于，使子类更易于提供针对子列表（sublist）的快速clear方法。如果没有removeRange方法，当在子列表（sublist）上调用clear方法时，子类将不得不用平方级的时间（quadratic performance）来完成它的工作。否则，就得重新编写整个subList机制——这可不是件容易的事情！

因此，当你为了继承而设计类的时候，如何决定应该暴露哪些受保护的方法或者域呢？遗憾的是，并没有神奇的法则可供你使用。你所能做到的最佳途径就是努力思考，发挥最好的想像，然后编写一些子类进行测试。你应该尽可能少地暴露受保护的成员，因为每个方法或者域都代表了一项关于实现细节的承诺。另一方面，你又不能暴露得太少，因为漏掉的受保护方法可能会导致这个类无法被真正用于继承。

*对于为了继承而设计的类，唯一的测试方法就是编写子类。*如果遗漏了关键的受保护成员，尝试编写子类就会使遗漏所带来的痛苦变得更加明显。相反，如果编写了多个子类，并且无一使用受保护的成员，或许就应该把它做成私有的。经验表明，3个子类通常就足以测试一个可扩展的类。除了超类的创建者之外，都要编写一个或者多个这种子类。

在为了继承而设计有可能被广泛使用的类时，必须要意识到，对于文档中所说明的自用模式（self-use pattern），以及对于其受保护方法和域中所隐含的实现策略，你实际上已经做出了永久的承诺。这些承诺使得你在后续的版本中提高这个类的性能或者增加新功能都变得非常困难，甚至不可能。因此，*必须在发布类之前先编写子类对类进行测试。*

还要注意，因继承而需要的特殊文档会打乱正常的文档信息，普通的文档被设计用来让程序员可以创建该类的实例，并调用类中的方法。在编写本书之时，几乎还没有适当的工具或

者注释规范，能够把“普通的API文档”与“专门针对实现子类的程序员的信息”分开来。

为了允许继承，类还必须遵守其他一些约束。构造器决不能调用可被覆盖的方法，无论是直接调用还是间接调用。如果违反了这条规则，很有可能导致程序失败。超类的构造器在子类的构造器之前运行，所以，子类中覆盖版本的方法将会在子类的构造器运行之前就先被调用。如果该覆盖版本的方法依赖于子类构造器所执行的任何初始化工作，该方法将不会如预期般地执行。为了更加直观地说明这一点，下面举个例子，其中有个类违反了这条规则：

```
public class Super {
    // Broken - constructor invokes an overridable method
    public Super() {
        overrideMe();
    }
    public void overrideMe() {
    }
}
```

下面的子类覆盖了方法overrideMe，Super唯一的构造器就错误地调用了这个方法：

```
public final class Sub extends Super {
    private final Date date; // Blank final, set by constructor

    Sub() {
        date = new Date();
    }

    // Overriding method invoked by superclass constructor
    @Override public void overrideMe() {
        System.out.println(date);
    }

    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

你可能会期待这个程序会打印出日期两次，但是它第一次打印出的是null，因为overrideMe方法被Super构造器调用的时候，构造器Sub还没有机会初始化date域。注意，这个程序观察到的final域处于两种不同的状态。还要注意，如果overrideMe已经调用了date中的任何方法，当Super构造器调用overrideMe的时候，调用就会抛出NullPointerException异常。如果该程序没有抛出NullPointerException异常，唯一的原因就在于println方法对于处理null参数有着特殊的规定。

在为了继承而设计类的时候，Cloneable和Serializable接口出现了特殊的困难。如果类是为了继承而被设计的，无论实现这其中的哪个接口通常都不是个好主意，因为它们把一些实质性的负担转嫁到了扩展这个类的程序员的身上。然而，你还是可以采取一些特殊的手段，使得子类实现这些接口，无需强迫子类的程序员去承受这些负担。第11条和74条中讲述了这些特殊的手段。

如果你决定在一个为了继承而设计的类中实现Cloneable或者Serializable接口，就应该意识到，因为clone和readObject方法在行为上非常类似于构造器，所以类似的限制规则也是适用的：*无论是**clone**还是**readObject**，都不可以调用可覆盖的方法，不管是以直接还是间接的方式*。对于readObject方法，覆盖版本的方法将在子类的状态被反序列化（deserialized）之前先被运行；而对于clone方法，覆盖版本的方法则是在子类的clone方法有机会修正被克隆对象的状态之前先被运行。无论哪种情形，都不可避免地将导致程序失败。在clone方法的情形中，这种失败可能会同时损害到原始的对象以及被克隆的对象本身。例如，如果覆盖版本的方法假设它正在修改对象深层结构的克隆对象的备份，就会发生这种情况，但是该备份还没有完成。

最后，如果你决定在一个为了继承而设计的类中实现Serializable，并且该类有一个readResolve或者writeReplace方法，就必须使readResolve或者writeReplace成为受保护的方法，而不是私有的方法。如果这些方法是私有的，那么子类将会不声不响地忽略掉这两个方法。这正是“为了允许继承，而把实现细节变成一个类的API的一部分”的另一种情形。

到现在为止，应该很明显：*为了继承而设计类，对这个类会有一些实质性的限制*。这并不是很轻松就可以承诺的决定。在某些情况下，这样的决定很明显是正确的，比如抽象类，包括接口的*骨架实现*（**skeletal implementation**）（见第18条）。但是，在另外一些情况下，这样的决定却很明显是错误的，比如不可变的类（见第15条）。

但是，对于普通的具体类应该怎么办呢？它们既不是final的，也不是为了子类化而设计和编写文档的，所以这种状况很危险。每次对这种类进行修改，从这个类扩展得到的客户类就有可能遭到破坏。这不仅仅是个理论问题。对于一个并非为了继承而设计的非final具体类，在修改了它的内部实现之后，接收到与子类化相关的错误报告也并不少见。

这个问题的最佳解决方案是，对于那些并非为了安全地进行子类化而设计和编写文档的类，要禁止子类化。有两种办法可以禁止子类化。比较容易的办法是把这个类声明为final的。另一种办法是把所有的构造器都变成私有的，或者包级私有的，并增加一些公有的静态工厂来替代构造器。后一种办法在第15条中讨论过，它为内部使用子类提供了灵活性。这两种办法都是可以接受的。

这条建议可能会引来争议，因为许多程序员已经习惯于对普通的具体类进行子类化，以便增加新的功能设施，比如仪表功能（instrumentation，如计数显示等）、通知机制或者同步功能，或者为了限制原有类中的功能。如果类实现了某个能够反映其本质的接口，比如Set、List或者Map，就不应该为了禁止子类化而感到后悔。第16条中介绍的*包装类*（**wrapper class**）模式提供了另一种更好的办法，让继承机制实现更多的功能。

如果具体的类没有实现标准的接口，那么禁止继承可能会给有些程序员带来不便。如果你

认为必须允许从这样的类继承，一种合理的办法是确保这个类永远不会调用它的任何可覆盖的方法，并在文档中说明这一点。换句话说，完全消除这个类中可覆盖方法的自用特性。这样做之后，就可以创建“能够安全地进行子类化”的类。覆盖方法将永远也不会影响其他任何方法的行为。

你可以机械地消除类中可覆盖方法的自用特性，而不改变它的行为。将每个可覆盖方法的代码体移到一个私有的“辅助方法（helper method）”中，并且让每个可覆盖的方法调用它的私有辅助方法。然后，用“直接调用可覆盖方法的私有辅助方法”来代替“可覆盖方法的每个自用调用”。

第18条：接口优于抽象类

Java程序设计语言提供了两种机制，可以用来定义允许多个实现的类型：接口和抽象类。这两种机制之间最明显的区别在于，抽象类允许包含某些方法的实现，但是接口则不允许。一个更为重要的区别在于，为了实现由抽象类定义的类型，类必须成为抽象类的一个子类。任何一个类，只要它定义了所有必要的方法，并且遵守通用约定，它就被允许实现一个接口，而不管这个类是处于类层次（class hierarchy）的哪个位置。因为Java只允许单继承，所以，抽象类作为类型定义受到了极大的限制。

现有的类可以很容易被更新，以实现新的接口。如果这些方法尚不存在，你所需要做的就只是增加必要的方法，然后在类的声明中增加一个implements子句。例如，当Comparable接口被引入到Java平台中时，会更新许多现有的类，以实现Comparable接口。一般来说，无法更新现有的类来扩展新的抽象类。如果你希望让两个类扩展同一个抽象类，就必须把抽象类放到类型层次（type hierarchy）的高处，以便这两个类的一个祖先成为它的子类。遗憾的是，这样做会间接地伤害到类层次，迫使这个公共祖先的所有后代类都扩展这个新的抽象类，无论它对于这些后代类是否合适。

接口是定义mixin（混合类型）的理想选择。不严格地讲，mixin是指这样的类型：类除了实现它的“基本类型（primary type）”之外，还可以实现这个mixin类型，以表明它提供了某些可供选择的行为。例如，Comparable是一个mixin接口，它允许类表明它的实例可以与其他的可相互比较的对象进行排序。这样的接口之所以被称为mixin，是因为它允许任选的功能可被混合到类型的主要功能中。抽象类不能被用于定义mixin，同样也是因为它们不能被更新到现有的类中：类不可能有一个以上的父类，类层次结构中也没有适当的地方来插入mixin。

接口允许我们构造非层次结构的类型框架。类型层次对于组织某些事物是非常合适的，但是其他有些事物并不能被整齐地组织成一个严格的层次结构。例如，假设我们有一个接口代表一个singer（歌唱家），另一个接口代表一个songwriter（作曲家）：

```
public interface Singer {
    AudioClip sing(Song s);
}
public interface Songwriter {
    Song compose(boolean hit);
}
```

在现实生活中，有些歌唱家本身也是作曲家。因为我们使用了接口而不是抽象类来定义这些类型，所以对于单个类而言，它同时实现Singer和Songwriter是完全允许的。实际上，我们可以定义第三个接口，它同时扩展了Singer和Songwriter，并添加了一些适合于这种组合的新方法：

```
public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

你并不总是需要这种灵活性，但是一旦你这样做了，接口可就成了救世主，能帮助你解决大问题。另外一种做法是编写一个臃肿（bloated）的类层次，对于每一种要被支持的属性组合，都包含一个单独的类。如果在整个类型系统中有n个属性，那么就必须支持2^n种可能的组合。这种现象被称为“组合爆炸（combinatorial explosion）”。类层次臃肿会导致类也臃肿，这些类包含许多方法，并且这些方法只是在参数的类型上有所不同而已，因为类层次中没有任何类型体现了公共的行为特征。

通过第16条中介绍的包装类（**wrapper class**）模式，接口使得安全地增强类的功能成为可能。如果使用抽象类来定义类型，那么程序员除了使用继承的手段来增加功能，没有其他的选择。这样得到的类与包装类相比，功能更差，也更加脆弱。

虽然接口不允许包含方法的实现，但是，使用接口来定义类型并不妨碍你为程序员提供实现上的帮助。通过对你导出的每个重要接口都提供一个抽象的骨架实现（**skeletal implementation**）类，把接口和抽象类的优点结合起来。接口的作用仍然是定义类型，但是骨架实现类接管了所有与接口实现相关的工作。

按照惯例，骨架实现被称为AbstractInterface，这里的Interface是指所实现的接口的名字。例如，Collections Framework为每个重要的集合接口都提供了一个骨架实现，包括AbstractCollection、AbstractSet、AbstractList和AbstractMap。将它们称作SkeletalCollection、SkeletalSet、SkeletalList和SkeletalMap也是有道理的，但是现在Abstract的用法已经根深蒂固。

如果设计得当，骨架实现可以使程序员很容易提供他们自己的接口实现。例如，下面是一个静态工厂方法，它包含一个完整的、功能全面的List实现：

```
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(final int[] a) {
    if (a == null)
        throw new NullPointerException();

    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i];  // Autoboxing (Item 5)
        }

        @Override public Integer set(int i, Integer val) {
            int oldVal = a[i];
            a[i] = val;     // Auto-unboxing
            return oldVal;  // Autoboxing
        }

        public int size() {
```

```
            return a.length;
        }
    };
}
```

当你考虑一个List实现应该为你完成哪些工作的时候，可以看出，这个例子充分演示了骨架实现的强大功能。顺便提一下，这个例子是个Adapter[Gamma95, p.139]，它允许将int数组看作Integer实例的列表。由于在int值和Integer实例之间来回转换需要开销，它的性能不会很好。注意，这个例子中只提供一个静态工厂，并且这个类还是个不可被访问的匿名类（**anonymous class**）（见第22条），它被隐藏在静态工厂的内部。

骨架实现的美妙之处在于，它们为抽象类提供了实现上的帮助，但又不强加“抽象类被用作类型定义时”所特有的严格限制。对于接口的大多数实现来讲，扩展骨架实现类是个很显然的选择，但并不是必需的。如果预置的类无法扩展骨架实现类，这个类始终可以手工实现这个接口。此外，骨架实现类仍然能够有助于接口的实现。实现了这个接口的类可以把对于接口方法的调用，转发到一个内部私有类的实例上，这个内部私有类扩展了骨架实现类。这种方法被称作模拟多重继承（**simulated multiple inheritance**），它与第16条中讨论的包装类模式密切相关。这项技术具有多重继承的绝大多数优点，同时又避免了相应的缺陷。

编写骨架实现类相对比较简单，只是有点单调乏味。首先，必须认真研究接口，并确定哪些方法是最为基本的（primitive），其他的方法则可以根据它们来实现。这些基本方法将成为骨架实现类中的抽象方法。然后，必须为接口中所有其他的方法提供具体的实现。例如，下面是Map.Entry接口的骨架实现类：

```
// Skeletal Implementation
public abstract class AbstractMapEntry<K,V>
        implements Map.Entry<K,V> {
    // Primitive operations
    public abstract K getKey();
    public abstract V getValue();

    // Entries in modifiable maps must override this method
    public V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    // Implements the general contract of Map.Entry.equals
    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (! (o instanceof Map.Entry))
            return false;
        Map.Entry<?,?> arg = (Map.Entry) o;
        return equals(getKey(),   arg.getKey()) &&
               equals(getValue(), arg.getValue());
    }
    private static boolean equals(Object o1, Object o2) {
        return o1 == null ? o2 == null : o1.equals(o2);
    }
```

```
        // Implements the general contract of Map.Entry.hashCode
        @Override public int hashCode() {
            return hashCode(getKey()) ^ hashCode(getValue());
        }
        private static int hashCode(Object obj) {
            return obj == null ? 0 : obj.hashCode();
        }
    }
```

因为骨架实现类是为了继承的目的而设计的，所以应该遵从第17条中介绍的所有关于设计和文档的指导原则。为了简短起见，上面例子中的文档注释部分被省略掉了，但是对于骨架实现类而言，好的文档绝对是非常必要的。

骨架实现上有个小小的不同，就是简单实现（**simple implementation**），AbstractMap.SimpleEntry就是个例子。简单实现就像个骨架实现，这是因为它实现了接口，并且是为了继承而设计的，但是区别在于它不是抽象的：它是最简单的可能的有效实现。你可以原封不动地使用，也可以看情况将它子类化。

使用抽象类来定义允许多个实现的类型，与使用接口相比有一个明显的优势：抽象类的演变比接口的演变要容易得多。如果在后续的发行版本中，你希望在抽象类中增加新的方法，始终可以增加具体方法，它包含合理的默认实现。然后，该抽象类的所有现有实现都将提供这个新的方法。对于接口，这样做是行不通的。

一般来说，要想在公有接口中增加方法，而不破坏实现这个接口的所有现有的类，这是不可能的。之前实现该接口的类将会漏掉新增加的方法，并且无法再通过编译。在为接口增加新方法的同时，也为骨架实现类增加同样的新方法，这样可以在一定程度上减小由此带来的破坏，但是，这样做并没有真正解决问题。所有不从骨架实现类继承的接口实现仍然会遭到破坏。

因此，设计公有的接口要非常谨慎。接口一旦被公开发行，并且已被广泛实现，再想改变这个接口几乎是不可能的。你必须在初次设计的时候就保证接口是正确的。如果接口包含微小的瑕疵，它将会一直影响你以及接口的用户。如果接口具有严重的缺陷，它可以导致API彻底失败。在发行新接口的时候，最好的做法是，在接口被“冻结”之前，尽可能让更多的程序员用尽可能多的方式来实现这个新接口。这样有助于在依然可以改正缺陷的时候就发现它们。

简而言之，接口通常是定义允许多个实现的类型的最佳途径。这条规则有个例外，即当演变的容易性比灵活性和功能更为重要的时候。在这种情况下，应该使用抽象类来定义类型，但前提是必须理解并且可以接受这些局限性。如果你导出了一个重要的接口，就应该坚决考虑同时提供骨架实现类。最后，应该尽可能谨慎地设计所有的公有接口，并通过编写多个实现来对它们进行全面的测试。

第19条：接口只用于定义类型

当类实现接口时，接口就充当可以引用这个类的实例的类型（**type**）。因此，类实现了接口，就表明客户端可以对这个类的实例实施某些动作。为了任何其他目的而定义接口是不恰当的。

有一种接口被称为常量接口（**constant interface**），它不满足上面的条件。这种接口没有包含任何方法，它只包含静态的final域，每个域都导出一个常量。使用这些常量的类实现这个接口，以避免用类名来修饰常量名。下面是一个例子：

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER   = 6.02214199e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

常量接口模式是对接口的不良使用。类在内部使用某些常量，这纯粹是实现细节。实现常量接口，会导致把这样的实现细节泄露到该类的导出API中。类实现常量接口，这对于这个类的用户来讲并没有什么价值。实际上，这样做反而会使他们更加糊涂。更糟糕的是，它代表了一种承诺：如果在将来的发行版本中，这个类被修改了，它不再需要使用这些常量了，它依然必须实现这个接口，以确保二进制兼容性。如果非final类实现了常量接口，它的所有子类的命名空间也会被接口中的常量所“污染”。

在Java平台类库中有几个常量接口，例如java.io.ObjectStreamConstants。这些接口应该被认为是反面的典型，不值得效仿。

如果要导出常量，可以有几种合理的选择方案。如果这些常量与某个现有的类或者接口紧密相关，就应该把这些常量添加到这个类或者接口中。例如，在Java平台类库中所有的数值包装类，如Integer和Double，都导出了MIN_VALUE和MAX_VALUE常量。如果这些常量最好被看作枚举类型的成员，就应该用枚举类型（**enum type**）（见第30条）来导出这些常量。否则，应该使用不可实例化的工具类（**utility class**）（见第4条）来导出这些常量。下面的例子是前面的PhysicalConstants例子的工具类翻版：

```
// Constant utility class
package com.effectivejava.science;

public class PhysicalConstants {
  private PhysicalConstants() { }  // Prevents instantiation
```

```
    public static final double AVOGADROS_NUMBER   = 6.02214199e23;
    public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    public static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

工具类通常要求客户端要用类名来修饰这些常量名，例如PhysicalConstants. AVOGADROS_NUMBER。如果大量利用工具类导出的常量，可以通过利用静态导入（**static import**）机制，避免用类名来修饰常量名，不过，静态导入机制是在Java发行版本1.5中才引入的：

```
// Use of static import to avoid qualifying constants
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
    double atoms(double mols) {
        return AVOGADROS_NUMBER * mols;
    }
    ...
    // Many more uses of PhysicalConstants justify static import
}
```

简而言之，接口应该只被用来定义类型，它们不应该被用来导出常量。

第20条：类层次优于标签类

有时候，可能会遇到带有两种甚至更多种风格的实例的类，并包含表示实例风格的标签（tag）域。例如，考虑下面这个类，它能够表示圆形或者矩形：

```
// Tagged class - vastly inferior to a class hierarchy!
class Figure {
    enum Shape { RECTANGLE, CIRCLE };

    // Tag field - the shape of this figure
    final Shape shape;

    // These fields are used only if shape is RECTANGLE
    double length;
    double width;

    // This field is used only if shape is CIRCLE
    double radius;

    // Constructor for circle
    Figure(double radius) {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }

    // Constructor for rectangle
    Figure(double length, double width) {
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }

    double area() {
        switch(shape) {
          case RECTANGLE:
            return length * width;
          case CIRCLE:
            return Math.PI * (radius * radius);
          default:
            throw new AssertionError();
        }
    }
}
```

这种标签类（**tagged class**）有着许多缺点。它们中充斥着样板代码，包括枚举声明、标签域以及条件语句。由于多个实现乱七八糟地挤在了单个类中，破坏了可读性。内存占用也增加了，因为实例承担着属于其他风格的不相关的域。域不能做成是final的，除非构造器初始化了不相关的域，产生更多的样板代码。构造器必须不借助编译器，来设置标签域，并初始化正确的数据域：如果初始化了错误的域，程序就会在运行时失败。无法给标签类添加风格，除非可以修改它的源文件。如果一定要添加风格，就必须记得给每个条件语句都添加一个条件，否则类就会在运行时失败。最后，实例的数据类型没有提供任何关于其风格的线索。一句话，标签类过于冗长、容易出错，并且效率低下。