Table of Contents

Effective Java 中文版 第2版

（美）**Joshua Bloch** 著

杨春华 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

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个性。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 `equlas()` 、 `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，建议你用心的研读本书。

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章和第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、各种数据类型和用法，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知道原则。

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端传递到客户端呢？GUI组件为什么可以被发现、保存和恢复呢？是因为他们实现了`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。

这是一本分享经验并指引你避免走弯路的经典著作，针对如何编写搞笑、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，也可以发送邮件到YuLimin@163.com与我交流。

在这里，我要感谢在翻译过程中一起讨论和帮助我的朋友们，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刚，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 manufactrures 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木鱼；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，当你是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设计前面两点，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，它会醉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中方法中，有7中方法是笨拙的、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，是大家更容易成功。新版仍然采用大量来自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语气。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于心，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、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公司的经历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。

引言

TBD

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：考虑使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，这样可以具有几大优势。

静态工厂方法与构造器不同的第一大优势在于，它具有名称。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。

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的对象，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 `true` 。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。接口不能有私有方法，因此按照惯例，接口 `Type` 的静态工厂方法放在一个名为 `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Java 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文档。使用这种静态工厂方法是，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。

发新版本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 `RegularEnumSet` 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 Databse 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，实现就按照类名进行注册，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.getConnection`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<Strin
g, 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。z和被称作类型推到（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能够在构造器调用以及方法调用中执行这种类型推到，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，类如果不含有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化方法。
- `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。
- `getType` —— 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。
- `newType` —— 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分标签。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，以此类推，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 so
dium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }

    public NutritionFacts(int servingSize, int servings, int calories, int fat, int so
dium, 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，但还是不得不为他们传递值。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队形，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 sodium       = 0;
        private int carbohydrate = 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 sodium(int val)
            { sodium = val;          return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = 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，就想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`IllegalArgumentsException`。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。

与构造器想必，builder模式的略微优势在于，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，知道每个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。为了创建西乡，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更加冗长，因此它只有在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，并导出共有的静态成员，一遍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创建异常。

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 sigleton 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 prefered approach
public enum Elvis {
    INSTANCE;

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

这种方法在功能上与公有方法相近，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，并由一个 `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(1964, 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：

```
public 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(1964, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }

    public boolean isBabyBoomer() {
        return birthDate.compareTo(boomStart) >= 0 &&
               birthdate.compareTo(boomEnd)   <  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` 方法被频繁的调用，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32000ms，而改进后的版本只需要130ms，大约快乐250倍。除了提高性能之外，代码的含义也更清晰了。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（*lazy 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。由于适配器除了后被对象之外，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，因为她们是由同一个 `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。它们在语义上还是有着微妙的差别，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 = 0;
    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正确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，请不要重用现有对象”。注意，在提倡使用保护性拷贝的时候，因为重用对象要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；而不必要的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.copayOf(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“内存泄露”，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，这种内存泄露会导致磁盘交换（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露呢？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做垃圾回收，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，内存泄露是很隐蔽的（称这类内存泄露为“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露的影响呢？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露问题。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。

内存泄露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` 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`。

内存泄露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。确保回调立即被当做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。

由于内存泄露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++中的析构器（destructor）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，并没有很轻便的方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被终止[JLS， 12.6]。未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2400ns。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结方法，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以后被调用，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，有过有必要的话，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 methods
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各类（`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（非`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发新版本将它增加到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，代价就是为每个将要终被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，该匿名类的唯一作用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
    pricate 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。

第三章 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` 的方法，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端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`** 方法永远也不会被调用。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作映射表（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(x)` 就会一致地返回 `true` ，或者一致的返回 `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（**reflexive**）————第一个要求仅仅说明对象必须等于其自身。很难想象会无意识地违反这一条。加入违背了这一条，然后把该类的实例添加到集合（collection）中，该集合的 `contains` 方法将果断的告诉你，该集合不包含你刚刚添加的实例。

对称性（**symmetric**）————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象。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 - violate symmetry
public final class CaseInsensitiveString {
    private final String s;

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

    // Broken - violate 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这么样。

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象。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)` 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 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，以检验某个整数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 subsititution*）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地很好[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 `CounterPoint` 扩展 `Point` ，而是在 `CounterPoint` 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.
     */
    pulic 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，不值得效仿。

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` 预定。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，应该仔细考虑她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cheng'ta'wei称它为“非空行（Non-nullity）”，意思是指所有的对象都必须不等于 `null`。尽管很难想象什么情况下 `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象。通用约定不允许抛出 `NullPointerException` 异常。许多类的 `equals` 方法都通过显示的 `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 instaceof 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` 方法的参数有事错误的类型，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` 操作符都会指定应该返回 `false` [JLS，15.20.2]。因此，如果把 `null` 传递给 `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步中的类型是个借口，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` 。对于 `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` 域和 `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的多。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，应该罪行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：它是不是对称的、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` 类不应该视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工作，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(707, 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在同一个散列桶中，`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不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苦难。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，并且该域的`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相乘等价于位移运算。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，即用位移和减法来代替乘法，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作散列键（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 `java.lang.Object` 提供了 `toString` 方法的一个实现，但它返回的字符串通常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` 的通用约定之处，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` 的约定进一步之处，“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 `sprint` 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，它可以被用作一种标准的、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人类阅读的数据对象中。例如XML文档。如果你指定了格式，最后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` 和绝大多数苏的基本类型包装类（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的表明你的意图。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);
}
```

如果你绝低挡不指定格式，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,
 */
@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猿，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是该，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，也不值得效仿。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租用，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。拷贝对象的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，这个对象僵尸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 `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。这里提现了一条通则：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（*invariaant*）。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数组中递归的调用 `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通数组，每个散列通都指向“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（它是链表的头结点）。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个好方法，因为针对列表中的每个元素，它都要消耗一端栈空间。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为它们的空白状态（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想必，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者不实现 `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，代表序号或者其他唯一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 `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性，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程可以带一个参数，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 fatory）），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` 接口具有上诉那么多问题，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。由于它具有这么多的缺点，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行比较，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）进行写作。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，比如按字母排序、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 set），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，下面的告诫也同样试用：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` 的权益之计也同样适用于 `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结合接口（ 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("1.0") ，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转型。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 方法中域的比较是有顺序的比较，而不是等同性的比较。比较对象引用域可以是递归地调用 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，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，按照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` ($2^{23}-1$)，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位的整数还没有大道足以表达任意两个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，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：使类的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和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），那些模块机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 acontrol*）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，使该成员编程包级私有的。如果你发现自己经常要做这样的的事情，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“泄露（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编程保护级别是，会大大增强可访问性。受保护的成员时类的导出的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，并且引用不可变对象，当把这个域编程公有的时候，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）。很重要的一点事，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，她便具有非 `final` 域的所有去电。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;
}
```

由于这种类的数据是可以被直接访问的，这些类没有提供封装（ecapsulation）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猿对这种类深恶痛绝，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，无论实在类定义中，还是在使用类的客户端代码中。虽然客户端代码与该类的内

部表示法紧密相连，但是这些代码被先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。它们是不值得效仿的例子，相反，这些类应该被当做反面的警告示例。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_DAT    = 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_DAT)
            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实危害比较小。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，无论这个类是可变的还是不可变的。

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（也成为mutator）。[注1]
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；Goetzo6 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) {
        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.re + im * c.im);
    }

    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种基本的算数运算：加法、减法、乘法和除法。注意这些算数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，对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，即被创建是的状态。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，一个很简单的办法就是，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平台的早起并不好理解，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建（*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困难的工作。

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，还有一种更加灵活的办法也可以做到这一点。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习类。对于处在它的包外部的客户端而言，不可变类实际上是 `final` 的，因为不可能把来自另一个包的类、缺少公有的活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。

静态工厂与构造器想必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” `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.calss)
        return new BigInteger(val.toByteArray());
    return val;
}
```

本条目开头初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，为了提供性能可以有所放松。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。除非有很好的利用要让类成为可变的类，否则就应该是不可变的。不可变的类有许多有点，唯一确定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额的。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编程是非 *`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。不要再构造器或者静态工厂之外再提供共有的初始化方法，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）。与所增加的复杂性想必，“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[Kaha91，Smith63，Thomas94]。

注：

1. 即改变对象属性的方法。——编辑注

第16条：复合优于继承

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）进行跨越包边界的集成，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`。为了调优改程序的性能，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，它记录下视图插入的元素数量，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 `super.addAll` 来调用 `HashSet` 的 `addAll` 实现。然后又一次调用到被 `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，则很有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。这种设计被称作“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 compsition 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 retrainAll(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技术特性的`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`类被称作包装类（*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，为它增加了技术特性。有时候，复合和转发的结果也被错误地成为“委托（*delegation*）”。从技术的角度而言，这不是委托，除非包装对向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]。有些人但系转发方法调用所带来的性能影响，或者包装对向导致的内存占用。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。

只有当子类眞正是超类的子类型（*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，要改正它已经为时太晚了，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是，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，那么就成将会导致脆弱性（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客服自（*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 colletion, if it is present(optional operation). More formally, removes an element e such that `(o==null ? e==nul : o.equals())`, 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 collecting looking for the specified element. If it finds the elements, it removes the element from the collection using the iterators'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==nul : o.equals())`。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行后果。所以，为了设计一个类的文档，以便它能够安全地子类化，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设计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 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 imporve the performance of the `clear` operation on this list and its sublists.
>
> This implementation get a list iterator positioned before `fromIndex` and repeatedly calls `ListIterator.next` follows 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自列表中调用的。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，并一次地重复调用`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，发挥最好的想象，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。如果改程序没有抛出 `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，因为它们把一些是执行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条和第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` 和 `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` 而言，覆盖版本的方法将在子类的 `clone` 方法有机会修正被克隆对象的状态之前被运行。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，就必须使用 `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，对这个类会有一些是实质性的限制。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方法类替代构造器。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。

这条建议可能回引来争议，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到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有调用”。

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。不严格地将，mixin是指这样的类型：类除了实现它的“基本类型（primitive type）”之外，还可以实现这个mixin接口，以表明她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 init);
}
```

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`。将他们称作 `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。

如果涉及得当，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;
        }
    }
}
```

当你考虑实现一个 `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` 数组看做 `Integer` 实例的列表。由于在 `int` 值和 `Integer` 实例之间来回转化需要开销，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力上，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知道原则。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导出的API中。类实现常量接口，这对于这个类的用户来讲没有什么价值。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，就应该使用枚举类型（*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 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，就必须级的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。

幸运的是，面向对象的语言例如Java，就提供了其他更好的方法来定义能够表示多种风格对象的单个数据类型：字类型化（subtyping）。标签类正式类层次的一种简单的仿效。

为了将标签类转变为类层次，首先要为标签类中的每个方法都定义一个包含抽象方法的抽象类，这每个方法的行为都依赖于标签值。在 `Figure` 类中，只有一个这样的方法： `area` 。这个抽象类是类层次的根（root）。如果还有其他的方法其行为不依赖于标签的值，就把这样的方法放在这个类中。同样地，如果所有的方法都用到了某些数据域，就应该把它们放在这个类中。在 `Figure` 类中，不存在这种类型独立的方法或者数据域。

接下来，为每种原始标签类都定义根类的具体子类。在前面的例子中，这样的类型有两个：圆形（circle）和句型（rectangle）。在每个子类中都包含特定于该类型的数据域。在我们的示例中， `radius` 是特定于圆形的， `length` 和 `width` 是特定于矩形的。同时在每个子类中还包括针对根类中每个抽象方法的相应实现。以下是与原始的 `Figure` 类相对应的类层次：

```
// Class hierarchy replacement for a tagged class
abstract class Figure {
    abstract double area();
}

class Circle extends Figure {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    double area() { return Math.PI * (radius * radius); }
}

class Rectangle extends Figure {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width  = width;
    }

    double area() { return length * width; }
}
```

这个类层次纠正了前面提到过的标签类的所有缺点。这段代码简单且清楚，没有包含在原来的版本中所见到的所有样板代码。每个类型的实现都配有自己的类，这些类都没有受到不相关的数据域的拖累。所有的域都是`final`的。编译器确保每个类的构造器都初始化了它的数据域，对于根类中声明的每个抽象方法，都确保有一个实现。这样就杜绝了由于遗漏 switch case 而导致运行时失败的可能性。多个程序员可以独立地扩展层次结构，并且不用访问根类的源代码就能相互操作。每种类型都有一种相关的独立的数据类型，允许程序员指明变量的类型，限制变量，并将参数输入到特殊的类型。

类层次的另一种好处在于，它们可以用来反映类型之间本质上的层次关系，有助于增强灵活性，并进行更好的编译时类型检查。假设上述例子中的标签类也允许表达正方形。类层次可以反映出正方形也是一种特殊的矩形这一事实（假设两者都是不可变的）：

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
}
```

注意，上述层次中的域是被直接访问，而不是通过访问方法。这是为了简洁起见才这么做的，如果层次结构是公有的（见第14条），则不允许这样做。

简而言之，标签类很少有适用的时候。当你想要编写一个包含显式标签域的类时，应该考虑一下，这个标签是否可以被取消，这个类是否可以用类层次来代替。当你遇到一个包含标签域的现有类时，就要考虑将它重构到一个层次结构中去。

第21条：用函数对象表示策略

有些语言支持函数指针（*function pointer*）、代理（*delegate*）、*lambda*表达式（*lambda expression*），或者支持类似的机制，允许程序把“调用特殊函数的能力”存储起来并传递这种的能力。这种机制通常允许函数的调用者通过传入第二个函数，来指定自己的行为。例如，C语言标准库中的`qsort`函数要求用一个指向*comparator*（比较器）函数的指针作为参数，它用这个函数来比较待排序的元素。比较器函数有两个参数，都是指向元素的指针。如果第一个参数所指的元素小于第二个参数所指的元素，则返回一个负整数；如果两个元素相等则返回零；如果第一个参数所指的元素大于第二个参数所指的元素，则返回一个正整数。通过传递不同的比较器函数，就可以获得各种不同的排列顺序。这正是策略（Strategy）模式[Gamma，p.315]的一个例子。比较器函数代表一种为元素排序的策略。

Java没有提供函数指针，但是可以用对象引用实现同样的功能。调用对象上的方法通常是执行该对象（*that object*）上的某项操作。然而，我们也可能定义这样一种对象，它的方法执行其他对象（*other objects*）（这些对象被显式传递给这些方法）上的操作。如果一个类仅仅导出这样的一个方法，它的实例实际上就等同于一个指向该方法的指针。这样的实例被称为函数对象（*function object*）。例如，考虑下面的类：

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

这个类导出一个带两个字符串参数的方法，如果第一个字符串的长度比第二个的短，则返回一个负整数；如果两个字符串的长度相等，则返回零；如果第一个字符串比第二个的长，则放回一个正整数。这个方法是一个比较器，它根据长度来给字符串排序，而不是根据更常用的字典顺序。指向`StringLengthComparator`对象的引用可以被当作是一个指向该比较器的“函数指针（function pointer）”，可以在任意一对字符串上被调用。换句话说，`StringLengthComparator`实例是用于字符串比较操作的具体策略（*concrete strategy*）。

作为典型的具体策略类，`StringLengthComparator`类是无状态的（*stateless*）：它没有域，所以，这个类的所有实例在功能上都是相互等价的。因此，它作为一个 Singleton 是非常合适的，可以节省不必要的对象创建开销（见第3条和第5条）：

```
class StringLengthComparator {
    private StringLengthComparator() { }
    public static final StringLengthComparator
        INSTANCE = new StringLengthComparator();
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

为了把 `StringLengthComparator` 实例传递给方法，需要适当的参数类型。使用 `StringLengthComparator` 并不好，因为客户端将无法传递任何其他的比较策略。相反，我们需要定义一个 `Comparator` 接口，并修改 `StringLengthComparator` 来实现这个接口。换句话说，我们在设计具体的策略类时，还需要定义一个策略接口（*strategy interface*），如下所示：

```
// Strategy interface
public interface Comparator<T> {
    public int compare(T t1, T t2);
}
```

`Comparator` 接口的这个定义碰巧也出现在 `java.util` 包中，但是这并不神奇；你自己也完全可以定义它。`Comparator` 接口是泛型（见第26条）的，因此它适合作为除字符串之外的其他对象的比较器。它的 `compare` 方法的两个参数类型为 `T`（它正常的参数类型），而不是 `String`。只要前面所示的 `StringLengthComparator` 类要这么做，就可以用它实现 `Comparator<String>` 接口：

```
class StringLengthComparator implements Comparator<String> {
    ... // class body is identical to the one shown above
}
```

具体的策略类往往使用匿名类声明（见第22条）。下面的语句根据长度对一个字符串数组进行排序：

```
Arrays.sort(stringArray, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

但是注意，以这种方式使用匿名类时，将会在每次执行调用的时候创建一个新的实例。如果它被重复执行，考虑将函数对象存储到一个私有的静态 `final` 域里，并重用它。这样做的另一种好处是，可以为这个函数对象取一个有意义的域名称。

因为策略接口被用作所有具体策略实例的类型，所以我们并不需要为了导出具体策略，而把具体策略类做成公有的。相反，“宿主类（host class）”还可以导出公有的静态域（或者静态工厂方法），其类型为策略接口，具体的策略类可以是宿主类的私有嵌套类。下面的例子使用静态成员类，而不是匿名类，以便允许具体的策略类实现第二个接口 `Serializable` ：

```
// Exporting a concrete strategy
class Host {
    private static class StrLenCmp
            implements Comparator<String>, Serializable {
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }

    // Returned comparator is serializable
    public static final Comparator<String>
        STRING_LENGTH_COMPARATOR = new StrLenCmp();

    ... // Bulk of class omitted
}
```

`String` 类利用这种模式，通过它的 `CASE_INSENSITIVE_ORDER` 域，导出一个不区分大小写的字符串比较器。

简而言之，函数指针的主要用途就是实现策略（Strategy）模式。为了在Java中实现这种模式，要声明一个接口来表示该策略，并且为每个具体策略声明一个实现了该接口的类。当一个具体策略只被使用一次时，通常使用匿名类来声明和实例化这个具体策略类。当一个具体策略类是设计用来重复使用的时候，它的类通常就要被实现为私有的静态成员类，并通过公有的静态 `final` 域被导出，其类型为该策略接口。

第22条：优先考虑静态成员类

嵌套类（*nested class*）是指被定义在另一个类的内部的类。嵌套类存在的目的应该只是为它的外围类（enclosing class）提供服务。如果嵌套类姜凯可能会用于其他的某个环境中，它就应该是顶层类（top-level class）。嵌套类有四种：静态成员类（*static member class*）、非静态成员类（*nonstatic member class*）、匿名类（*anonymous class*）和局部类（*local class*）。除了第一种之外，其他三种都被称为内部类（*inner class*）。本条目将告诉你什么时候应该使用哪种嵌套类，以及这样做的原因。

静态成员类是最简单的一种嵌套类。最好把他看作是普通的类，只是碰巧被声明在另一个类的内部而已，它可以方位外围类的所有成员，包括那些声明为私有的成员。静态成员类是外围类的一个静态成员，与其他静态成员一样，也遵守同样的可访问性规则。如果它被声明为私有的，它就只能在外围类的内部才可以被访问，等等。

静态成员类的一种常见用法是作为公有的辅助类，仅当与它的外部类一起使用时才有意义。例如，考虑一个枚举，它描述了计算器支持的各种操作（见第30条）。`Operation`枚举应该是`Calculator`类的公有静态成员，然后，`Calculator`类的客户端就可以用诸如`Calculator.Operation.PLUS`和`Calculator.Operation.MINUS`这样的名称来引用这些操作。

从语法上讲，静态成员类和非静态成员类之间唯一的区别是，静态成员类的声明中包含修饰符`static`。尽管它们的语法非常相似，但是这两种嵌套类有很大的不同。非静态成员类的每个实例都隐含着与外围类的一个外围实例（*enclosing instance*）相关联。在非静态成员类的实例方法内部，可以调用外围实例上的方法，或者利用修饰过的`this`构造获得外围实例的引用[JLS，15.8.4]。如果嵌套类的实例可以在它外围类的实例之外独立存在，这个嵌套类就必须是静态成员类：在没有外围实例的情况下，要想创建非静态成员类的实例是不可能的。

当非静态成员类的实例被创建的时候，它和外围实例之前的关联关系也随之被建立起来；而且，这种关联关系以后不能被修改。通常情况下，当在外围类的某个实例方法的内部调用非静态成原来的构造器时，这种关联关系被自动建立起来。使用表达式`enclosingInstance.new MemberClass(args)`来手工建立这种关联关系也是有可能的，但是很少使用。正如你所预料的那样，这种关联关系需要消耗非静态成员类实例的空间，并且增加了构造的时间开销。

非静态成员类的一种常见用法是定义一个Adapterp[Gamma95，p.139]，它允许外部类的实例被看作是另一个不相关的类的实例。例如，`Map`接口的实现往往使用非静态成员类来实现它们的集合视图（*collection view*），这些集合视图是由`Map`的`keySet`、`entrySet`和`values`方法返回的。同样地，诸如`Set`和`List`这种集合接口的实现往往也使用非静态成员类来实现它们的迭代器（iterator）：

```
// Typical use of a nonstatic member class
public class MySet<E> extends AbstractSet<E> {
    ... // bulk of the class omitted

    public Iterator<E> iterator() {
        reteurn new MyIterator();
    }

    private class MyIterator implements Iterator<E> {
        ...
    }
}
```

如果声明成员类不要求访问外围实例，就要始终把 *static* 修饰符放在它的声明中，使它成为静态成员类，而不是非静态成员类。如果省略了 `static` 修饰符，则每个实例都将包含一个额外的指向外围对象的引用。保存这份引用要消耗时间和空间，并且会导致外围实例在符合垃圾回收（见第6条）时却仍然得以保留。如果在没有外围实例的情况下，也需要分配实例，就不能使用非静态成员类，因为非静态成员类的实例必须要有一个外围实例。

私有静态成员类的一种常见用法是用来代替外围类所代表的对象的组件。例如，考虑一个 `Map` 实例，它把键（key）和值（value）关联起来。许多 `Map` 实现的内部都有一个 `Entry` 对象，对应于 `Map` 中每个键-值对。虽然每个 entry 都与一个 `Map` 关联，但是 entry 上的方法（ `getKey` 、 `getValue` 和 `setValue` ）并不需要访问该 `Map` 。因此，使用非静态成员来表示 entry 是很浪费的：私有的静态成员类是最佳的选择。如果不小心漏掉了 entry 声明中的 `static` 修饰符，该 `Map` 仍然可以工作，但是每个 entry 中将会包含一个指向该 `Map` 的引用，这样就浪费了空间和时间。

如果相关的类是导出类的共有的或受保护的成员，毫无疑问，在静态和非静态成员类之间做出正确的选择是非常重要的。在这种情况下，该成员类就是导出的API元素，在后续的发行版本中，如果不违反二进制兼容性，就不能从非静态成员类变为静态成员类。

匿名类不同于Java程序设计语言中的其他任何语法单元。正如你所想象的，匿名类没有名字。它不是外围类的一个成员。它并不与其他的成员一起被声明，而是在使用的同时被声明和实例化。匿名类可以出现在代码中任何允许存在表达式的地方。当且仅当匿名类出现在非静态的环境中时，它才有外围实例。但是即使它们出现在静态的环境中，也不可能拥有任何静态成员。

匿名类的适用性受到诸多的限制。除了在它们被声明的时候之外，是无法将它们实例化的。你不能执行 `instanceof` 而是，或者做任何需要命名类的其他事情。你无法声明一个匿名类来实现多个借口，或者扩展一个类，并同时扩展类和实现接口。匿名类的客户端无法调用任何成员，除了从它的超类型中继承得到之外。由于匿名类出现在表达式中，它们必须保持简短——大约10行或者更少些——否则会影响程序的可读性。

匿名类的一种常见用法是动态地创建函数对象（*function object*，见第21条。例如，第92页中的 `sort` 方法调用，利用匿名的 `Comparator` 实例，根据一组字符串的长度对它们进行裴谐。匿名类的另一种常见用法是创建过程对象（*process object*），比如 `Runnable` 、 `Thread` 或者 `TimeTask` 实例。第三种常见的用法是在静态工厂方法的内部（参见第18条中的 `intArrayAsList` 方法）。

局部类是四种嵌套类中用得最少的类。在任何“可以声明局部变量”的地方，都可以声明局部类，并且局部类也遵守同样的作用域规则。局部类与其他三种嵌套类中的每一种都有一些共同的属性。与成员类一样，局部类有名字，可以被重复地使用。与匿名类一样，只有当局部类实在非静态环境中定义的时候，才有外围实例，它们也不能包含静态成员。与匿名类一样，它们必须非常简短，以便不会影响到可读性。

简而言之，共有四种不同的嵌套类，每一种都有自己的用途。如果一个嵌套类需要在单个方法之外仍然是可见的，或者太长了，不适合于放在方法内部，就应该使用成员类。如果成员类的每个实例都需要一个指向其外围实例的引用，就要把成员类做成非静态的；否则，就做成静态的。假设这个嵌套类属于一个方法的内部，如果你只需要在一个地方创建实例，并且已经有了一个预置的类型可以说明这个类的特征，就要把它做成匿名类；否则，就做成局部类。

第5章 泛型

Java 1.5发行版本中增加了泛型（*Generic*）。在没有泛型之前，从集合中读取到的每一个对象都必须进行转换。如果有人不小心插入了类型错误的对象，在运行时的转换处理就会出错。有了泛型之后，可以告诉编译器每个集合中接受哪些对象类型。编译器自动地为你的插入进行转化，并在编译时告知是否插入了类型错误的对象。这样可以使程序既更加安全，也更加清楚，但是要享有这些优势有一定的难度。本章就是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。有关这部分内容的详情，请参见 Langer 的教程[Langer08]，或者 Naftalin 和 Walder 合著的书[Naftalin07]。

第23条：请不要在新代码中使用原生态类型

先来介绍一些术语。声明中具有一个或者多个类型参数（*type parameter*）的类或者接口，就是泛型（*generic*）类或者接口[JLS，8.1.2，9.1.2]。例如，从Java 1.5发行版本起，`List`接口就只有单个类型参数`E`，表示列表的元素类型。从技术的角度来看，这个接口的名称应该是指现在的`List<E>`（读作“`E`的列表”），但是人们经常把它简称为`List`。泛型类和接口统称为泛型（*generic type*）。

每种泛型定义一组参数化的类型（*parameterized type*），构成格式为：显示类或者接口的名称，接着用尖括号（`< >`）把对应于泛型形式类型参数的实际类型参数列表[JLS，4.4，4.5]括起来。例如，`List<String>`（读作“字符串列表”）是一个参数化的类型，表示元素类型为`String`的列表。（`String`是与形式类型参数`E`相对应的实际类型参数。）

最后一点，每个泛型都定义了一个原生态类型（*raw type*），即不带任何实际类型参数的泛型名称[JLS，4.8]。例如，与`List<E>`相对应的原生态类型是`List`。原生态类型就像从类型声明中删除了所有泛型信息一样。实际上，原生态类型`List`与Java平台没有泛型之前的接口类型`List`完全一样。

在Java 1.5版本发行之前，以下集合声明是值得参考的：

```
// Now a raw collection type - don't do this

/**
 * My stamp collection. Contains only Stamp instances.
 */
private final Collection stamps = ...;
```

如果一不小心将一个 coin 放进了 stamp 集合中，这一错误的插入照样得以编译和运行并且不会出现任何错误提示：

```
// Erroeous insertion of coin into stamp collection
stamps.add(new Coin(...));
```

直到从 stamp 集合中获取 coin 时才会收到错误提示：

```
// Now a raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hashNext(); ) {
    Stamp s = (Stamp) i.next(); // Throws ClassCastException
    ... // Do something with the stamp
}
```

就如本书中经常提到的，出错之后应该尽快发现，最好是编译时就发现。本例中，直到代码运行时才发现错误，已经出错很久了，而且你在代码中所处的位置距离包含错误的这部分代码已经很远了。一旦发现`ClassCastException`，就必须搜索代码，查找将 coin 放进 stamp 集合的方法调用。此时编译器帮不上忙，因为它无法理解这种注释：“Contains only Stamp instances（只包含`Stamp`实例）”。

有了泛型，就可以利用改进后的类型声明来代替集合中的这种注释，告诉编译器之前的注释中所隐含的信息：

```
// Parameterized collection type - typesafe
private final Collection<Stamp> stamps = ... ;
```

通过这条声明，编译器知道`stamps`应该只包含`Stamp`实例，并给予保证，假设整个代码是利用Java 1.5及其之后版本的编译器进行编译的，所有代码在编译过程中都没有发出（或者禁止，请见第24条）任何警告。当`stamps`利用一个参数化的类型进行声明时，错误的插入会产生一条编译时的错误消息，准确地告诉你哪里出错了：

```
Test.java:9: add(Stamp) in Collection<Stamp> cannot be applied
to (Coin)
    stamps.add(new Coin())
                   ^
```

还有一个好处是，从集合中删除元素时不再需要进行手工转换了。编译器会替你插入隐式的转换，并确保它们不会失败（依然假设所有代码都是通过支持泛型的编译器进行编译的，并且没有产生或者禁止任何警告）。无论你是否使用 for-each 循环（见第46条），上述功能都适用：

```
// for-each loop over a parameterized collection - typesafe
for (Stamp s : stamps) { // No cast
    ... // Do something with the stamp
}
```

或者无论是否使用传统的`for`循环也一样：

```
// for loop with parameterized iterator declaration - typesafe
for (Iterator<Stamp> i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = i.next(); // No cast necesary
    ... // Do something with the stamp
}
```

虽然假设不小心将 coin 插入到 stamp 集合中可能显得有点牵强，但这类问题确实真实的。例如，很容易想象有人会不小心将一个`java.util.Date`实例放进一个原本只包含`java.sql.Date`实例的集合中。

如上所述，如果不提供类型参数，使用集合类型和其他泛型也仍然是合法的，但是不应该这么做。如果使用原生态类型，就失掉了泛型在安全性和表述性方面的所有优势。既然不应该使用原生态类型，为什么Java的设计者还要允许使用它们呢？这是为了提供兼容性。因为泛型出现的时候，Java平台即将进入它的第二个10年，已经存在大量没有使用方形的Java代码。人们认为让所有的代码保持合法，并且能够与使用泛型的新代码互用，这一点很重要。它必须合法，才能将参数化类型的实例传递给那些被设计成使用普通类型的方法，反之亦然。这种需求被称作移植兼容性（Migration Compatibility），促成了支持原生态类型的决定。

虽然不应该在新代码中使用像 `List` 这样的原生态类型，使用参数化的类型以允许插入任意对象，如 `List<Object>` ，这还是可以的。原生态类型 `List` 和和参数化的类型 `List<Object>` 之间到底有什么区别呢？不严格地说，前者逃避了泛型检查，后者则明确告知编译器，它能够持有任意类型的对象。虽然你可以将 `List<String>` 传递给 `List` 的参数，但是不能讲它传给类型 `List<Object>` 的参数。泛型有子类型化（subtyping）的规则， `List<String>` 是原生态类型 `List` 的一个子类型，而不是参数化类型 `List<Object>` 的子类型（见第25条）。因此，如果使用像 *`List`* 这样的原生态类型，就会失掉类型安全性，但是如果使用像 *`List<Object>`* 这样的参数化类型，则不会。

为了更具体地进行说明，请参考下面的程序：

```
// Uses raw type (List) - fails at runtime!
public static void main(String[] args) {
    List<String> strings = new ArrayList<String>();
    unsafeAdd(strings, new Integer(42));
    String s = strings.get(0); // Compiler-generated cast
}

private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

这个程序可以进行编译，但是因为它使用了原生态类型 `List` ，你会收到一条警告：

```
Test.java:10: warning: unchecked call to add(E) in raw type list
    list.add(o);
             ^
```

实际上，如果运行这段程序，在程序试图将 `strings.get(0)` 的调用结果转换成一个 `String` 时，会收到一个 `ClassCastException` 异常。这是一个编译器生成的转换，因此一般保证会成功，但是我们在这个例子中忽略了一条编译器警告，就会为此而付出代价。

如果在 `unsafeAdd` 声明中使用参数化类型 `List<Object>` 代替原生态类型 `List` ，并试着重新编译这段程序，会发现它无法再进行编译了。以下是它的错误消息：

```
Test.java:5: unsafeAdd(List<Object>,Object) cannot be applied
to (List<String>,Integer)
    unsafeAdd(strings, new Integer(42));
    ^
```

在不确定或者不在乎集合中的元素类型的情况下，你也许会使用原生态类型。例如，假设想要编写一个方法，它有两个集合（`Set`），并从中返回它们共有的元素的数量。如果你对泛型还不熟悉的话，可以参考一下方式来编写这种方法：

```
// Use of raw type for unknown element type - don't do this!
static int numElementsInCommon(Set s1, Set s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

这个方法倒是可以，但它使用了原生态类型，这是很危险的。从Java 1.5发行版本开始，Java就提供了一种安全的替代方法，称作无限制的通配符类型（*unbounded wildcard type*）。如果要使用泛型，但不确定或者不关心实际的类型参数，就可以使用一个问号代替。例如，泛型`Set<E>`的无限制通配符类型为`Set<?>`（读作“某个类型的集合”）。这是最普通的参数化`Set`类型，可以持有任意集合。下面是`numElementsInCommon`方法使用了无限制通配符类型时的情形：

```
// Unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

在无限制通配符类型`Set<?>`和原生态类型`Set`之间有什么区别呢？这个问号真正起到作用了吗？这一点不需要赘述，但通配符类型是安全的，原生态类型则不安全。由于可以将任何元素放进使用原生态类型的集合中，因此很容易破坏该集合的类型约束条件（如第100页的例子中所示的`unsafeAdd`方法）；但不能将任何元素（除了`null`之外）放到`Collection<?>`中。如果尝试这么做的话，将会产生一条像这样的编译时错误消息：

```
WildCard.java:13: cannot find symbol
symbol  : method add(String)
location: interface Collection<capture#825 of ?>
        c.add("verboten");
          ^
```

这样的错误消息显然还无法令人满意，但是编译器已经尽到了它的职责，防止你破坏集合的类型约束条件。你不仅无法将任何元素（除了 `null` 之外）放进 `Collection<?>` 中，而且根本无法猜测你会得到哪种类型的对象。要是无法接受这些限制，就可以使用泛型方法（generic method，见第27条）或者有限制的通配符类型（bounded wildcard type，见第28条）。

不要在新代码中使用原生态类型，这条规则有两个小小的例外，两者都源于“泛型信息可以在运行时被擦除”（见第25条）这一事实。在类文字（class literal）中必须使用原生态类型。规范不允许使用参数化类型（虽然允许数组类型和基本类型）[JLS，15.8.2]。换句话说，`List.class`，`String[].class` 和 `int.class` 都合法，但是 `List<String.class>` 和 `List<?>.class` 则不合法。

这条规则的第二个例外与 `instanceof` 操作符有关。由于泛型信息可以在运行时被擦除，因此在参数化类型而非无限制通配符类型上使用 `instanceof` 操作符是非法的。用无限制通配符类型代替原生态类型，对 `instanceof` 操作符的行为不会产生任何影响。在这种情况下，尖括号（`<>`）和问号（`?`）就显得多余了。下面是利用泛型来使用 *`instanceof`* 操作符的首选方法：

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) {       // Raw type
    Set<?> m = (Set<?>) o;    // Wildcard type
    ...
}
```

注意，一旦确定这个 `o` 是个 `Set`，就必须将它转换成通配符类型 `Set<?>`，而不是转换成原生态类型 `Set`。这是个受检的（checked）转换，因此不会导致编译时警告。

总之，使用原生态类型会在运行时导致异常，因此不要在新代码中使用。原生态类型只是为了与引入泛型之前的遗留代码进行兼容和互用而提供的。让我们做个快速的回顾：`Set<Object>` 是个参数化类型，表示可以包含任何对象类型的一个集合；`Set<?>` 则是一个通配符类型，表示只能包含某种未知对象类型的一个集合；`Set` 则是个原生态类型，它脱离了泛型系统。前两种是安全的，最后一种不安全。

为便于参考，表 5-1 概括了本条目中所介绍的术语（及本章其他条目中介绍的一些术语）：

表5-1 本章条目中所介绍的术语

术语	示例	所在条目
参数化的类型	`List<String>`	第23条
实际类型参数	`String`	第23条
泛型	`List<E>`	第23，26条
形式类型参数	`E`	第23条
无限制通配符类型	`List<?>`	第23条
原生态类型	`List`	第23条
有限制类型参数	`<E extends Number>`	第26条
递归类型参数	`<T extends Comparable<T>>`	第27条
有限制通配符类型	`List<? extends Number>`	第28条
泛型方法	`static <E> List<E> asList(E[] a)`	第27条
类型令牌	`String.class`	第29条

第24条：消除非受检警告

用泛型编程时，会遇到许多编译器警告：非受检强制转化警告（unchecked cast warnings）、非受检方法调用警告、非受检普通数组创建警告，以及非受检转换警告（unchecked conversion warnings）。当你越来越熟悉泛型之后，遇到的警告也会越来越少，但是不要期待从一开始用泛型编写代码就可以正确地进行编译。

有许多非受检警告很容易消除。例如，假设意外地编写了这样一个声明：

```
Set<Lark> exaltation = new HashSet();
```

编译器会细致地提醒你哪里出错了：

```
Venery.java:4: warning: [unchecked] unchecked conversion
found   : HashSet, required: Set<Lark>
    Set<Lark> exaltation = new HashSet();
                           ^
```

你就可以纠正所显示的错误，消除警告：

```
Set<Lark> exaltation = new HashSet<Lark>();
```

有些警告比较难以消除。本章主要介绍这种警告的示例。当你遇到需要进行一番思考的警告时，要坚持住！要尽可能地消除每一个非受检警告。如果消除了所有警告，就可以确保代码是类型安全的，这是一件很好的事情。这意味着不会在运行时出现 `ClassCastException` 异常，你会更加自信自己的程序可以实现预期的功能。

如果无法消除警告，同时可以证明引起警告的代码是类型安全的，（只有在这种情况下才）可以用一个 `@SuppressWarnings("unchecked")` 注解来禁止这条警告。如果在禁止警告之前没有先证实代码是类型安全的，那就只是给你自己一种错误的安全感而已。代码在编译的时候可能没有出现任何警告，但它在运行时仍然会抛出 `ClassCastException` 异常了但是如果忽略（而不是禁止）明知道是安全的非受检警告，那么当新出现一条真正有问题的警告时，你也不会注意到。新出现的警告就会淹没在所有的错误警告当中。

`SuppressWarnings` 注解可以用在任何粒度的级别中，从单独的局部变量声明到整个类都可以。应该始终在尽可能小的范围中使用 `SuppressWarnings` 注解。它通常是个变量声明，或是非常简短的方法或者 构造器。永远不要再整个类上使用 `SuppressWarnings` 注解，这么做可能会掩盖了重要的警告。

如果你发现自己在长度不止一行的方法或者构造器中使用 `SuppressWarnings` 注解，可以将它移到一个局部变量的声明中。虽然你必须声明一个新的局部变量，不过这么做还是值得的。例如，考虑 `ArrayList` 类当中的 `toArray` 方法：

```
public <T> toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arrayCopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

如果编译 `ArrayList` ，该方法就会产生这条警告：

```
ArrayList.java:305: warning: [unchecked] unchecked cast
found   : Object[], required: T[]
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                               ^
```

将 `SuppressWarnings` 注解放在 `return` 语句中是非法的，因为它不是一个生命[JLS，9.7]。你可以试着将注解放在整个方法上，但是在实践中千万不要这么做，而是应该声明一个局部变量来保存返回值，并注解其声明，像这样：

```
// Adding local variable to reduce scope of @SuppressWarnings
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        // This cast is correct because the array we're creating
        // is of the same type as the one passed in, which is T[].
        @SuppressWarnings("unchecked") T[] result =
            (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arrayCopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

这个方法可以正确地编译，禁止非受检警告的范围也减到了最小。

每当使用 `@SuppressWarnings("unchecked")` 注解时，都要添加一条注释，说明为什么这么做是安全的 。这样可以帮助其他人理解代码，更重要的是，可以尽量减少其他人修改代码后导致计算不安全的概率。如果你觉得这种注释很难编写，就要多加思考。最终你会发现非受检操作时非常不安全的。

总而言之，非受检警告很重要，不要忽略它们。每一条警告都表示可能在运行时抛出`ClassCastException`异常。要尽最大的努力消除这些警告。如果无法消除非受检警告，同时可以证明引起警告的代码是类型安全的，就可以在尽可能小的范围中，用`@SuppressWarnings("unchecked")`注解禁止该警告。要用注释把禁止该警告的原因记录下来。

第25条：列表优先于数组

数组与泛型相比，有两个重要的不同点。首先，数组是协变的（*covariant*）。这个词听起来有点吓人，其实只是表示如果 `Sub` 为 `Super` 的子类型，那么数组类型 `Sub[]` 就是 `Super[]` 的子类型。相反，泛型则是不可变的（*invariant*）：对于任意两个不同的类型 `Type1` 和 `Type2`，`List<Type1>` 既不是 `List<Type2>` 的子类型，也不是 `List<Type2>` 的超类型[JLS，4.10；Naftalin07，2.5]。你可能认为，这意味着泛型是有缺陷的，但实际上可以说数组才是有缺陷的。

下面的代码片段是合法的：

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

但下面这段代码则不合法；

```
// Won't compile!
List<Object> ol = new ArrayList<Object>(); // Incompatible types
ol.add("I don't fit in");
```

这其中无论哪种方法，都不能将 `String` 放进 `Long` 容器中，但是利用数组，你会在运行时发现所犯的错误；利用列表，则可以在编译时发现错误。我们当然希望在编译时发现错误了。

数组与泛型之间的第二大区别在于，数组是具体化的（*reified*）[JLS，4.7]。因此数组会在运行时才知道并检查它们的元素类型约束。如上所述，如果企图将 `String` 保存到 `Long` 数组中，就会得到一个 `ArrayStoreException` 异常。相比之下，泛型则是通过擦除（*erasure*）[JLS，4.6]来实现的。因此泛型只在编译时强化它们的类型信息，并在运行时丢弃（或者擦除）它们的元素类型信息。擦除就是使泛型可以与没有使用泛型的代码随意进行互用（见第23条）。

由于上述这些根本的区别，因此数组和泛型不能很好地混合使用。例如，创建泛型、参数化类型或者类型参数的数组是非法的。这些数组创建表达式没有一个是合法的：`new List<E>[]`、`new List<String>[]` 和 `new E[]`。这些在编译是都会导致一个 generic array creation（泛型数组创建）错误。

为什么创建泛型数组是非法的？因为它不是类型安全的。要是它合法，编译器在其他正确的程序中发生的转换就会在运行时失败，并出现一个 `ClassCastException` 异常。这就违背了泛型系统提供的基本保证。

为了更具体地对此进行说明，考虑以下代码片段：

```
// Why generic array creation is illegal - won't compile
List<String>[] stringLists = new ArrayList<String>[1]; // (1)
List<Integer> intList = Arrays.asList(42);             // (2)
Object[] objects = stringLists;                        // (3)
objects[0] = intList;                                  // (4)
String s = stringLists[0].get(0);                      // (5)
```

我们假设第1行是合法的，它创建了一个泛型数组。第2行创建并初始化了一个包含单个元素的 `List<Integer>` 。第3行将 `List<String>` 数组保存到一个 `Object` 数组变量中，这是合法的，因为数组是协变的。第4行将 `List<Integer>` 保存到 `Object` 数组里唯一的元素中，这是可以的，因为泛型是通过擦除实现的： `List<Integer>` 实例的运行时类型只是 `List` ， `List<String>[]` 实例的运行时类型则是 `List[]` ，因此这种安排不会产生 `ArrayStoreException` 异常。但现在我们有麻烦了。我们将一个 `List<Integer>` 实例保存到了原本声明只包含 `List<String>` 实例的数组中。在第5行中，我们从这个数组里唯一的列表中获取了唯一的元素。编译器自动地将获取到的元素转换成 `String` ，但它是一个 `Integer` ，因此，我们在运行时得到了一个 `ClassCastExcption` 异常。为了防止出现这种情况，（创建泛型数组的）第1行产生了一个编译时错误。

从技术的角度来说，想 `E` 、 `List<E>` 和 `List<String>` 这样的类型应称作不可具体化的（nonreifiable）类型[JLS，4.7]。直观地说，不可具体化的（non-reifiable）类型是指其运行时表示法包含的信息比它的编译时表示法包含的信息更少的类型。唯一可具体化的（reifiable）参数化类型是无限制的通配符类型，如 `List<?>` 和 `Map<?,?>` （见第23条）。虽然不常用，但是创建无限制通配符类型的数组是合法的。

禁止创建泛型数组可能有点讨厌。例如，这表明泛型一般不可能返回它的元素类型数组（部分解决方案请见第29条）。这也意味着在结合使用可变参数（varargs）方法（见第42条）和泛型时会出现令人费解的警告。这是由于每当调用可变参数方法时，就会创建一个数组来存放 varargs 参数。如果这个数组的元素类型时不是可具体化的（reifiable），就会得到一条警告。关于这些警告，除了把它们禁止（见第24条），并且避免在API中混合使用泛型与可变参数之外，别无他法。

当你得到泛型数组创建错误时，最好的解决办法通常是优先使用集合类型 `List<E>` ，而不是数组类型 `E[]` 。这样可能会损失一些性能或者简洁性，但是换回的却是更高的类型安全性和互用性。

例如，假设有一个（ `Collections.synchronizedList` 返回的那种）同步列表和一个函数（它有两个域该列表的元素同类型的参数值，并返回第三个值）。现在假设要编写一个方法 `reduce` ，并使用函数 `apply` 来处理这个列表。假设列表元素类型为整数，并且函数是用来做两个整数求和运算， `reduce` 方法就会返回列表中所有值的总和。如果函数是用来做两个整数求积的运算，该方法就会返回列表中值的乘积。如果列表包含字符串，并且函数连接两个字符串，该方法就会返回一个字符串，它按顺序包含了列表中的所有字符串。除了列表和函

数之外，`reduce`方法还采用初始值进行减法运算，列表为空时会返回这个初始值。（初始值一般为函数的识别元素，加法为0，乘法为1，字符串连接时是""。）以下是没有泛型时的代码？

```
// Reduction without generics, and with concurrency flaw!
static Object reduce(List list, Function f, Object initVal) {
    synchronized(list) {
        Object result = initVal;
        for (Object o : list)
            result = f.apply(result, o);
        return result;
    }
}

interface Function {
    Object apply(Object arg1, Object arg2);
}
```

假设你现在已经读过第67条，它告诉你不要从同步区域中调用“外来的（alien）方法”。因此，在持有锁的时候修改`reduce`方法来复制列表中内容，也可以让你在备份上执行减法。Java 1.5发行版本之前，要这么做一般是利用`List`的`toArray`方法（它在内部锁定列表）：

```
// Reduction without generics or concurrency flaw
static Object reduce(List list, Function f, Object initVal) {
    Object[] snapshot = list.toArray(); // Locks list internally
    Object result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

如果试图通过泛型来完成这一点，就会遇到我们之前讨论过的那种麻烦。以下是`Function`接口的泛型版：

```
interface Function<T> {
    T apply(T arg1, T arg2);
}
```

下面是一种天真的尝试，试图将泛型应用到修改过的`reduce`方法。这是一个泛型方法（generic method，见第27条）。如果你不理解这条声明，也不必担心。对于这个条目来说，应该把注意力集中在方法体上：

```
// Naive generic version of reduction - won't compile!
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    E[] snapshot = list.toArray(); // Locks list
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

如果试着编译这个方法，就会得到下面的错误消息：

```
Reduce.java:12: incompativle types
found   : Object[], required: E[]
        E[] snapshot = list.toArray(); // Locks list
                                ^
```

你会说，这没什么大不了的，我会将 `Object` 数组转换成一个 `E` 数组：

```
E[] snapshot = (E[]) list.toArray();
```

它是消除了那条错误，但是现在得到了一条警告：

```
Reduce.java:12: warning: [unchecked] unchecked cast
found   : Object[], required:L E[]
        E[] snapshot = (E[]) list.toArray(); // Locks list
                                          ^
```

编译器告诉你，它无法在运行时检查转换的安全性，因为它在运行时还不知道 `E` 是什么——记住，元素类型信息会在运行时从泛型中被擦除。这段程序可以运行吗？结果表明，它可以运行，但是不安全。通过微小的修改，就可以让它在没有包含显式转换的行上抛出 `ClassCastException` 异常。 `snapshot` 的编译时类型为 `E[]` ，它可以为 `String[]` 、 `Integer[]` 或者任何其他种类的数组。运行时类型为 `Object[]` ，这是很危险的。不可具体化的类型的数组转换只能在特殊情况下使用（见第26条）。

那么应该做些什么呢？用列表代替数组。下面的 `reduce` 方法编译时就没有任何错误或者警告：

```
// List-based generic reduction
static <E> E reduce(List<E> list, Funciton<E> f, E initVal) {
    List<E> snapshot;
    synchronized(list) {
        snapshot = new ArrayList<E>(list);
    }
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

这个版本的代码比数组版的代码稍微冗长一点，但是可以确定在运行时不会得到`ClassCastException`异常，为此也值了。

总而言之，数组和泛型有着非常不同的类型规则。数组是协变并且可以具体化的；反省是不可变的且可以被擦除的。因此，数组提供了运行时的类型安全，但是没有编译时的类型安全，反之，对于泛型也一样。一般来说，数组和泛型不能很好地混合使用。如果你发现自己将它们混合起来使用，并且得到了编译时的错误或者警告，你的第一反应就应该是用列表代替数组。

第26条：优先考虑泛型

一般来说，将集合声明参数化，以及使用 JDK 所提供的泛型和泛型方法，这些都不太困难。编写自己的泛型会比较困难一些，但是值得花些时间学习如何编写。

考虑第 6 条中这个简单的堆栈实现：

```
// Object-based collection - a prime candidate for generics
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();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

这个类是泛型化（generification）的主要备选对象，换句话说，可以适当地强化这个类类利用泛型。根据实际情况来看，必须转换从堆栈里弹出的对象，以及可能在运行时失败的那些转换。将类泛型化的第一个步骤是给它的声明添加一个或者多个类型参数。在这个例子中有一个类型参数，它表示堆栈的元素类型，这个参数的名称通常为 `E`（见第 44 条）。

下一步是用相应的类型参数替换所有的 `Object` 类型，然后试着编译最终的程序：

```
// Initial attempt to generify Stack = won't compile
public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

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

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

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

    ... // no changes in isEmpty or ensureCapacity
}
```

通常，你将至少得到一个错误或警告，这个类也不例外。幸运的是，这个类只产生一个错误，如下：

```
Stack.java:8: generic array creation
        elements = new E[DEFAULT_INITIAL_CAPACITY];
                   ^
```

如第 25 条中所述，你不能创建不可具体化的（non-reifiable）类型的数组，如 `E` 。每当编写用数组支持的泛型时，都会出现这个问题。解决这个问题有两种方法。第一种，直接绕过创建泛型数组的禁令：创建一个 `Object` 的数组，并将它转换为泛型数组类型。现在错误是消除了，但是编译器会产生一条警告。这种用法是合法的，但（整体上而言）不是类型安全的：

```
Stack.java:8: [unchecked] unchecked cast
found   : Object[], required: E[]
       elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
                        ^
```

编译器不可能证明你的程序是类型安全的，但是你可以证明。你自己必须确保未受检的转换不会危及到程序的类型安全性。相关的数组（即 `elements` 变量）保存在一个私有的域中，永远不会被返回到客户端，或者传递给任何其他方法。这个数组中保存的唯一元素，是传

给 `push` 方法的那些元素，它们的类型为 `E`，因此未受检的转换不会有任何危害。

一旦你证明了未受检的转换是安全的，就要在尽可能小的范围中禁止警告（见第 24 条）。在这种情况下，构造器只包含未受检的数组创建，因此可以在整个构造器中禁止这条警告。通过增加一条注解来完成禁止，`Stack` 能够正确无误地进行编译，你就可以使用它了，无需显式的转化，也无需担心会出现 `ClassCastException` 异常：

```
// The elements array will contain only E instances from push(E).
// This is sufficient to ensure type safety, but the runtime
// type of the array won't be E[]; it will always be Object[]!
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

消除 `Stack` 中泛型数组创建错误的第二种方法是，将 `elements` 域的类型从 `E[]` 改为 `Object[]`。这么做会得到一条不同的错误：

```
Stack.java:19: incompatible types
found   : Object, required: E
        E result = elements[--size];
                            ^
```

通过把从数组中获取到的元素有 `Object` 转换成 `E`，可以将这条错误变成一条警告：

```
Stack.java:19: warning: [unchecked] unchecked cast
found   : Object, required: E
        E result = elements[--size];
                            ^
```

由于 `E` 是一个不可具体化的（non-reifiable）类型，编译器无法在运行时检验转换。你还是可以自己证实未受检的转换是安全的，因此可以禁止该警告。根据第 24 条的建议，我们只要在包含未受检转换的任务上禁止警告，而不是在整个 `pop` 方法上就可以了，如下：

```
// Appropriate suppression of unchecked warning
public E pop() {
    if (size == 0)
        throw new EmptyStackException();

    // push requires elements to be of type E, so cast is correct
    @SuppressWarnings("unchecked") E result =
        (E) elements[--size];

    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

具体选择这两种方法中的哪一种来处理泛型数组创建错误，这要看个人的偏好了。所有其他的东西都一样，但是禁止数组类型的未受检转换比禁止标量类型（scalar type）的更加危险，所以建议采用第二种方案。但是在比 `Stack` 更实际的泛型类中，或许多代码会有多个地方需要从数组中读取元素，因此选择第二种方案需要多次转换成 `E` ，而不是只转换成 `E[]` ，这也是第一种方案之所以更常用的原因[Naftalin07，6.7]。

下面的程序示范了泛型 `Stack` 类的使用。程序以相反的顺序打印出它的命令行参数，并转换成大写字母。如果要从堆栈中弹出的元素上调用 `String` 的 `toUpperCase` 方法，并不需要显式的转换，并且会确保自动生成的转换会成功：

```
// Little program to exercise our generic Stack
public static void mai(String[] args) {
    Stack<String> stack = new Stack<String>();
    for (String arg : args)
        stack.push(args);
    while (!stack.isEmpty())
        System.out.println(stack.pop.toUpperCase());
}
```

看来上述的示例与第 25 条相矛盾了，第 25 条鼓励优先使用列表而非数组。实际上并不可能总是或者总想在泛型中使用列表。Java 并不是生来就支持列表，因此有些泛型如 `ArrayList` 则必须在数组上实现。为了提升性能，其他泛型如 `HashMap` 也在数组上实现。

绝大多数泛型就像我们的 `Stack` 示例一样，因为它们的类型参数没有限制：你可以创建 `Stack<Object>` 、 `Stack<int[]>` 、 `Stack<List<String>>` ，或者任何其他对象引用类型的 `Stack` 。注意不能创建基本类型的 `Stack` ：企图创建 `Stack<int>` 或者 `Stack<double>` 会产生一个编译时错误。这是 Java 泛型系统根本的局限性。你可以通过使用包装类型（boxed primitive type）来避开这条限制（见第 49 条）。

有一些泛型限制了可允许的类型参数值。例如，考虑 `java.util.concurrent.DelayQueue` ，其声明如下：

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>;
```

类型参数列表（`<E extends Delayed>`）要求实际的类型参数 `E` 必须是 `java.util.concurrent.Delayed` 的一个子类型。它允许 `DelayedQueue` 实现及其客户端在 `DelayedQueue` 元素上利用 `Delayed` 方法，无需显式的转换，也没有出现 `ClassCastException` 的风险。类型参数 `E` 被称作有限制的类型参数（bounded type parameter）。注意，子类型关系确定了，每个类型都是它自身的子类型[JLS，4.10]，因此创建 `DelayedQueue<Delayed>` 是合法的。

总而言之，使用泛型比使用需要在客户端代码中进行转换的类型来得更加安全，也更加容易。在设计新类型的时候，要确保它们不需要这种转换就可以使用。这通常意味着要把类做成是泛型的。只要时间允许，就把现有的类型都泛型化。这对于这些类型的新用户来说会变得更加qingso9ng，又不会破坏现有的客户端（见第 23 条）。

第27条：优先考虑泛型方法

就如类可以从泛型中收益一般，方法也一样。静态工具方法尤其适合于泛型化。`Collections` 中的所有“算法”方法（例如 `binarySearch` 和 `sort`）都泛型化了。

编写泛型方法与编写泛型类相类似。例如下面这个方法，它返回两个集合的联合：

```
// Uses raw types - unacceptable! (Item 23)
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

这个方法可以编译，但是有两条警告：

```
Union.java:5: warning: [unchecked] call to
Hashset(Collection<? extends E>) as a member of raw type HashSet
        Set result = new HashSet(s1);
                     ^
Union.java:6: warning: [unchecked] call to
addAll(Collection<? extends E>) as a member of raw type Set
        result.addAll(s2);
                     ^
```

为了修正这些警告，使方法编程是类型安全的，要将方法声明修改为声明一个类型参数，表示这三个集合的元素类型（两个参数和一个返回值），并在方法中使用类型参数。声明类型参数的类型参数列表，处在方法的修饰符及其返回类型之间。在这个示例中，类型参数列表为 `<E>`，返回类型为 `Set<E>`。类型参数的命名管理与泛型方法以及泛型的相同（见第 26 条和第 44 条）：

```
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<E>(s1);
    result.addAll(s2);
    return result;
}
```

至少对于简单的泛型方法而言，就是这么回事了。现在该方法编译时不会产生任何警告，并提供了类型安全性，也更容易使用。以下是一个知性该方法的简单程序。程序中不包含转换，编译时不会有错误或者警告：

```
// Simple program to exercise generic method
public static void main(String[] args) {
    Set<String> guys = new HashSet<String>(
        Arrays.asList("Tom", "Dick", "Harry"));
    Set<String> stooges = new HashSet<String>(
        Arrays.asList("Larry", "Moe", "Curly"));
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

运行这段程序时，会打印出 `[Moe, Harry, Tom, Curly, Larry, Dick]` 。元素的顺序是依赖于实现的。

`union` 方法的局限性在于，三个集合的类型（两个输入参数和一个返回值）必须全部相同。利用有限制的通配符类型（bounded wildcard type），可以使这个方法变得更加灵活（见第28条）。

泛型方法的一个显著特性是，无需明确指定类型参数的值，不像调用泛型构造器的时候是必须指定的。编译器通过检查方法参数的类型来计算类型参数的值。对于上述的程序而言，编译器发现 `union` 的两个参数都是 `Set<String>` 类型，因此知道类型参数 `E` 必须为 `String` 。这个过程称作类型推导(type inference)。

如第1条所述，可以利用泛型方法调用所提供的类型推导，使创建参数化类型实例的过程变得更加轻松。提醒一下：在调用泛型构造器的时候，要明确传递类型参数的值可能有点麻烦。类型参数出现在了变量声明的左右两边，显得有些冗余：

```
// Parameterizedtype instance creation with constructor
Map<String, List<String>> anagrams =
    new HashMap<String, List<String>>();
```

为了消除这种冗余，可以编写一个泛型静态工厂方法（generic static factory method），与想要使用的每个构造器相对应。例如，下面是一个与无参的 `HashMap` 构造器相对应的泛型静态工厂方法：

```
// Generic static factory method
public static <K, V> HashMap<K, V> newHashMap() {
    return new HashMap<K, V>();
}
```

通过这个泛型静态工厂方法，可以用下面这段简洁的代码来取代上面那个重复的声明：

```
// Parameterizedtype instance creation with static factory
Map<String, List<String>> anagrams = newHashMap();
```

在泛型上调用构造器时，如果语言所有的类型推导与调用泛型方法时所做的相同，那就好了。将来的某一天也许可以实现这一点，但截至 Java 1.6 发行版还不行。

相关的模式是泛型单例工厂（generic singleton factory）。有时，会需要创建不可变但又适合于许多不同类型的对象。由于泛型是通过擦除（见第25条）实现的，可以给所有必要的类型参数使用单个对象，但是需要编写一个静态工厂方法，重复地给每个必要的类型参数分发对象啊。这种模式最常用于函数对象（见第21条），如 `Collections.reverseOrder` ，但也适用于像 `Collections.emptySet` 这样的集合。

假设有一个接口，描述了一个方法，该方法接受和返回某个类型 `T` 的值：

```
public interface UnaryFunction<T> {
    T apply(T arg);
}
```

现在假设要提供一个恒等函数（identity function）。如果在每次需要的时候都重新创建一个，这样会很浪费，因为它是无状态的（stateless）。如果泛型被具体化了，每个类型都需要一个恒等函数，但是它们被擦除以后，就只需要一个泛型单例。请看以下示例：

```
// Generic singleton factory pattern
private static UnaryFunction<Object> IDENTITY_FUNCTION =
    new UnaryFunction<Object>() {
        public Object apply(Object arg) { return arg; }
  }

// IDENTITY_FUNCTION is stateless and its type parameter is
// unbounded so it's safe to share one instance across all types.
@SuppressWarnings("unchecked")
public static <T> UnaryFunction<T> identityFunction() {
    return (UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

`IDENTITY_FUNCTION` 转换成（ `UnaryFunction<T>` ）产生了一条未受检转换警告，因为 `UnaryFunction<Object>` 对于每个 `T` 来说并非都是个 `UnaryFunction<T>` 。但是恒等函数很特殊：它返回未被修改的参数，因此我们知道无论 `T` 的值是什么，用它作为 `UnaryFunction<T>` 都是类型安全的。因此，我们可以放心地禁止由这个转换所产生的未受检转换警告。一旦禁止，代码在编译时就不会出现任何错误或者警告。

以下是一个范例程序，利用泛型单例作为 `UnaryFunction<String>` 和 `UnaryFunction<Number>` 。像往常一样，它不包含转换，编译时没有出现错误或者警告：

```
// Sample program to exercise generic singleton
public static void main(String[] args) {
    String[] strings = { "jute", "hemp", "nylon" };
    UnaryFunction<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };
    UnaryFunction<Number> sameNumber = identityFunction();
    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

虽然相对少见，但是通过某个包含该类型参数本身的表达式来限制类型参数是允许的。这就是递归类型限制（**recursive type bound**）。递归类型限制最普遍的用途与 `Comparable` 接口有关，它定义类型的自然顺序：

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

类型参数 `T` 定义的类型，可以与实现 `Comparable<T>` 的类型的元素进行比较。实际上，几乎所有的类型都只能与它们自身的类型的元素相比较。因此，例如 `String` 实现 `Comparable<String>`，`Integer` 实现 `Comparable<Integer>`，等等。

有许多方法都带有一个实现 `Comparable` 接口的元素列表，为了对列表进行排序，并在其中进行搜索，计算它的最小值或者最大值，等等。要完成这其中的任何一项工作，要求列表中的每个元素都能够与列表中的每个其他元素相比较，换句话说，列表的元素可以互相比较（mutually comparable）。下面是如何表达这种约束条件的一个示例：

```
// Using a recursive type bound to express mutual comparability
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

类型限制 `<T extends Comparable<T>>` 可以读作“针对可以与自身进行比较的每个类型 `T`”，这与互比性的概念或多或少有些一致。

下面的方法就带有上述声明。它根据元素的自然顺序计算列表的最大值，编译时没有出现任何错误或者警告：

```
// Returns the maximum value in a list - uses recursive type bound
public static <T extends Comparable<T>> T max(List<T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

递归类型限制可能比这个要复杂得多，但幸运的是，这种情况并不经常发生。如果你理解了这种习惯用法及其通配符变量（见第28条），就能够处理在实践中遇到的许多递归类型限制了。

总而言之，泛型方法就像泛型一样，使用起来比要求客户端转换输入参数并返回值的方法来得更加安全，也更加容易。就像类型一样，你应该确保新方法可以不用转换就能使用，这通常意味着要将它们泛型化。并且就像类型一样，还应该将现有的方法泛型化，使新用户使用起来更加轻松，且不会破坏现有的客户端（见第23条）。

第28条：利用有限通配符来提升API的灵活性

如第25条所述，参数化类型是不可变的（invariant）。换句话说，对于任何两个截然不同的 `Type1` 和 `Type2` 而言，`List<Type1>` 既不是 `List<Type2>` 的子类型，也不是它的超类型。虽然 `List<String>` 不是 `List<Object>` 的子类型，这与直觉相悖，但是实际上很有意义。你可以将任何对象放进一个 `List<Object>` 中，却只能将字符串放进 `List<String>` 中。

有时候，我们需要的灵活性要比不可变类型所能提供的更多。考虑第26条中的堆栈，下面就是它的公共API：

```
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

假设我们想要增加一个方法，让它按顺序将一系列的元素全部放到堆栈中。这是第一次尝试，如下：

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

这个方法编译时正确无误，但并非尽如人意。如果 `Iterable src` 的元素类型与堆栈的完全匹配，就没有问题。但是假如有一个 `Stack<Number>` ，并且调用了 `push(intVal)` ，这里的 `intVal` 就是 `Integer` 类型。这是你可以的，因为 `Integer` 是 `Number` 的一个子类型。因此从逻辑上来说，下面这个方法应该也可以：

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ...;
numberStack.pushAll(integers);
```

但是，如果尝试这么做，就会得到下面的错误消息，因为如前所述，参数化类型是不可变的：

```
StackTest.java:7: pushAll(Iterable<Number>) in Stack<Number>
cannot be applied to (Iterable<Integer>)
        numberStack.pushAll(integers);
                    ^
```

幸运的是，有一种解决办法。Java提供了一种特殊的参数化类型，称作有限制的通配符类型（bounded wildcard type），来处理类似的情况。`pushAll`的输入参数类型不应该为“`E`的`Iterable`接口”，而应该为“`E`的某个子类型的`Iterable`接口”，有一个通配符类型正符合此意：`Iterable<? extends E>`。（使用关键字`extends`有些误导：回忆一下第26条中的说法，确定了子类型（subtype）后，每个类型便都是自身的子类型，即便它没有将自己扩展。）我们修改一下`pushAll`来使用这个类型：

```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

这么修改了以后，不仅`Stack`可以正确无误地编译，没有通过初始的`pushAll`声明进行编译的客户端代码也一样可以。因为`Stack`及其客户端正确无误地进行了编译，你就知道一切都是类型安全的了。

现在假设想要编写一个`pushAll`方法，使之与`popAll`方法相呼应。`popAll`方法从对战中弹出没个元素，并将这些元素添加到指定的集合中。初次尝试编写的`popAll`方法可能像下面这样：

```
// popAll method without weildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

如果目标集合的元素类型与堆栈的完全匹配，这段代码编译时还是会正确无误，运行得很好。但是，也并不意味着尽如人意。假设你有一个`Stack<Number>`和类型`Object`的变量。如果从堆栈中弹出一个元素，并将它保存在该变量中，它的编译和运行都不会出错，那你为何不能也这么做呢？

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ...;
numberStack.popAll(objects);
```

如果试着用上述的`popAll`版本编译这段客户端代码，就会得到一个非常类似于第一次用`pushAll`时所得到的错误：`Collection<Object>`不是`Collection<Number>`的子类型。这一次，通配符类型同样提供了一种解决办法。`popAll`的输入参数类型不应该为“`E`的集合”，而应该为“`E`的某种超类的集合”（这里的超类是确定的，因此`E`是它自身的一个超类型[JLS，4.10]）。仍然有一个通配符正是符合此意：`Collection<? super E>`。让我们修改`popAll`来使用它：

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

做了这个变动之后，`Stack`和客户端代码就都可以正确无误地编译了。

结论很明显。为了获得最大限度的灵活性，要在表示生产者或者消费者的输入参数上使用通配符类型。如果某个输入参数既是生产者，有事消费者，那么通配符类型对你就没有什么好处了：因为你需要的是严格的类型匹配，这是不用任何通配符而得到的。

下面的助记符便于让你记住要使用哪种通配符类型：

PECS 表示 producer-extends，consumer-super。

换句话说，如果参数化类型表示一个`T`生产者，就是用`<? extends T>`；如果它表示一个`T`消费者，就是用`<? super T>`。在我们的`Stack`示例中，`pushAll`的`src`参数产生`E`实例供`Stack`使用，因此`src`相应的类型为`Iterable<? extends E>`；`popAll`的`dst`参数通过`Stack`消费`E`实例，因此`dst`相应的类型为`Collection<? super E>`。PECS这个助记符突出了使用通配符类型的基本原则。Naftalin 和 Wadler 称之为**Get and Put Principle**[Naftalin07， 2.4]。

记住这个助记符，我们下面来看一些之前的条目中提到过的方法声明。第25条中的`reduce`方法就有这条声明：

```
static <E> E reduce(List<E>, list, Function<E> f, E initVal)
```

虽然列表既可以消费也可以产生值，`reduce`方法还是只用它的`list`参数作为`E`生产者（**producer**），因此它的声明就应该使用一个`extends E`的通配符类型。参数`f`表示皆可以消费又可以产生`E`实例的函数，因此通配符类型不适合它。得到的方法声明如下：

```
// Wildcard type for parameter that serves as an E producer
static <E> reduce(List<? extends E> list, Function<E> f,
                  E initVal)
```

这一变化实际上有什么区别吗？事实上，的确有区别。假设你有一个`List<Integer>`，想通过`Function<Number>`把它简化。它不能通过初始声明进行编译，但是一旦添加了有限制的通配符类型，就可以了。

现在让我们看看第27条中的`union`方法。下面是声明：

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

`s1` 和 `s2` 这两个参数都是 `E` 消费者，因此根据 PECS，这个声明应该是：

```
public static <E> Set<E> union(Set<? extends E> s1,
                               Set<? extends E> s2)
```

注意返回类型仍然是 `Set<E>`，不要用通配符类型作为返回类型。除了为用户提供额外的灵活性之外，它还会强制用户在客户端代码中使用通配符类型。

如果使用得当，通配符类型对于类的用户来说几乎是无形的。它们使方法能够接受它们应该接受的参数，并拒绝那些应该拒绝的参数。如果类的用户必须考虑通配符类型，类的*API*或许就会出错。

遗憾的是，类型推导（type inference）规则相当复杂，在语言规范中占了整整16页[JLS，15.12.2.7-8]，而且它们并非总能完成需要它们完成的工作。看看修改过的 `union` 声明，你可能会以为可以像这样编写：

```
Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(integers, doubles);
```

但这么做会得到下面的错误消息：

```
Union.java:14: incompatible types
found : Set<Number & Comparable<? extends Number &
                                        Comparable<?>>>
required: Set<Number>
        Set<Number> numbers = union(integers, doubles);
                                    ^
```

幸运的是，有一种办法可以处理这个错误。如果编译器不能推断你希望它拥有的类型，可以通过一个显式的类型参数（explicit type parameter）来告诉它要使用哪种类型。这种情况不太经常发生，这是好事，因为显式的类型参数不太优雅。增加了这个显式的类型参数之后，程序可以正确无误地进行编译：

```
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

接下来，我们把注意力转向第27条中的 `max` 方法。以下是初始的声明：

```
public static <T extends Comparable<T>> T max(List<T> list)
```

下面是修改过的使用通配符类型的声明：

```
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list)
```

为了从初始声明中得到修改后的版本，要应用PECS转换两次。最直接的是运用到参数 `list` 。它产生 `T` 实例，因此将类型从 `List<T>` 改成 `List<? extends T>` 。更灵活的是运用到类型参数 `T` 。这是我们第一次见到将通配符运用到类型参数。最初 `T` 被指定用来扩展 `Comparable<T>` ，但是 `T` 的 `comparable` 消费 `T` 实例（并产生表示顺序的整值）。因此，参数化类型 `Comparable<T>` 被有限制通配符类型 `Comparable<? super T>` 取代。 `comparable` 始终是消费者，因此使用时始终应该是 *`Comparable<? super T>`* 优先于 *`Comparable<T>`* 。对于 `comparator` 也一样，因此使用时始终应该是 *`Comparable<? super T>`* 优先于 *`Comparable<T>`* 。

修改过的 `max` 声明可能是整本书中最复杂的方法声明了。所增加的复杂代码真的起作用了吗？是的，起作用了。下面是一个简单的列表示例，在初始的声明中不允许这样，修改过的版本则可以：

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

不能将初始方法声明运用给这个列表的原因在于， `java.util.concurrent.ScheduledFuture` 没有实现 `Comparable<ScheduledFuture>` 接口。相反，它是扩展 `Comparable<Delayed>` 接口的 `Delayed` 接口的子接口。换句话说， `ScheduledFuture` 示例并非只能与其他 `ScheduledFuture` 示例相比较；它可以与任何 `Delayed` 实例相比较，这就足以导致初始声明时就会被拒绝。

修改过的 `max` 声明有一个小小的问题：它阻止方法进行编译。下面的方法包含了修改过的声明：

```
// Won't compile - wildcards can require change in method body!
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

以下是它编译时会产生的错误消息：

```
Max.java:7: incompatible types
found   : Iterator<capture#591 of ? extends T>
required: Iterator<T>
        Iterator<T> i = list.iterator();
                                    ^
```

这条错误消息意味着什么，我们又该如何修正这个问题呢？它意味着 `list` 不是一个 `List<T>`，因此它的 `iterator` 方法没有返回 `Iterator<T>`。它返回 `T` 的某个子类型的一个 `iterator`，因此我们用它代替 `iterator` 声明，它使用了一个有限制的通配符类型：

```
Iterator<? extends T> i = list.iterator();
```

这是必须对方法体所做的唯一修改。迭代器的 `next` 方法返回的元素属于 `T` 的某个子类型，因此它们可以被安全地保存在类型 `T` 的一个变量中。

还有一个与通配符有关的话题值得探讨。类型参数和通配符之间具有双重性，许多方法都可以利用其中一个或者另一个进行声明。例如，下面是可能的两种静态方法声明，来交换列表中的两个被索引的项目。第一个使用无限制的类型参数（见第27条），第二个使用无限制的通配符：

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

你更喜欢两种方法中的哪一种呢？为什么？在公共API中，第二种更好一下，因为它更简单。将它传到一个列表中——任何列表——方法就会交换被索引的元素。不用担心类型参数。一般来说，如果类型参数只在方法中出现一次，就可以用通配符取代它。如果是无限制的类型参数，就用无限制的通配符取代它；如果是有限制的类型参数，就用有限制的通配符取代它。

将第二种声明用于 `swap` 方法会有一个问题，它优先使用通配符而非类型参数：下面这个简单的实现都不能编译：

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

试着编译时会产生这条没有什么用处的错误消息：

```
Swap.java:5: set(int,capture#282 of ?) in List<capture#282 of ?>
cannot be applied to (int,Object)
        list.set(i, list.set(j, list.get(i)));
                        ^
```

不能讲元素放回到刚刚从中取出的列表中，这似乎不太对劲。问题在于 `list` 的类型为 `List<?>`，你不能把 `null` 之外的任何值放到 `List<?>` 中。幸运的是，有一种方式可以实现这个方法，无需求助于不安全的转换或者原生态类型（raw type）。这种想法就是编写一个私有的辅助方法类捕捉通配符类型。为了捕捉类型，方法必须是泛型方法，像下面这样：

```
public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}

// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

`swapHelper` 方法知道 `list` 是一个 `List<E>`。因此，它知道从这个列表中取出的任何值均为 `E` 类型，并且知道将 `E` 类型的任何值放进列表都是安全的。`swap` 这个有些费解的实现编译起来却是正确无误的。它允许我们导出 `swap` 这个比较好的基于通配符的声明，同时在内部利用更加复杂的泛型方法。`swap` 方法的客户端不一定要面对更加复杂的 `swapHelper` 声明，但是它们的确从中受益。

总而言之，在API中使用通配符类型虽然比较需要技巧，但是使API变得灵活的多。如果编写的是将被广泛使用的类库，则一定要适当地利用通配符类型。记住基本的原则：producer-extends，consumer-super(PECS)。还要记住所有的 `comparable` 和 `comparator` 都是消费者。

第29条：优先考虑类型安全的异构容器

泛型最常用于集合，如 `Set` 和 `Map` ，以及单元素的容器。
如 `ThreadLocal` 和 `AtomicReference` 。在这些用法中，它都充当被参数化了的容器。这样就限制你每个容器只能有固定数目的类型参数。一般来说，这种情况正是你想要的。一个 `Set` 只有一个类型参数，表示它的元素类型；一个 `Map` 有两个类型参数，表示它的键和值类型；诸如此类。

但是，有时候你会需要更多的灵活性。例如，数据库行可以有任意多的列，如果能以类型安全的方式访问所有列就好了。幸运的是，有一种方法可以很容易地做到这一点。这种想法就是将键（key）进行参数化而不是将容器（container）参数化。然后将参数化的键提交给容器，来插入或者获取值。用泛型系统来确保值的类型与它的键相符。

简单地示范一下这种方法：考虑 `Favorites` 类，它允许客户端从任意数量的其他类中，保存并获取一个“最喜爱”的实例。 `Class` 对象充当参数化键的关键部分。之所以可以这样，是因为类 `Class` 在 Java 1.5 版本中被泛型化了。类的类型从字面上来看不再只是简单的 `Class` ，而是 `Class<T>` 。例如， `String.class` 属于 `Class<String>` 类型， `Integer.class` 属于 `Class<Integer>` 类型。当一个类的字面文字被用在方法中，来传达编译时和运行时的类型信息是，就被称作*type token*[Brancha04]。

`Favorites` 类的API很简单。它看起来就像一个简单的map，除了键（而不是map）被参数化之外。客户端在设置和获取最喜爱的实例时提交 `Class` 对象。下面就是这个API：

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

下面是一个示例程序，检验一下 `Favorites` 类，它保存、获取并打印一个最喜爱的 `String` 、 `Integer` 和 `Class` 实例：

```
// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
    Favorites f = new Favorites();
    f.putFavorite(String.class, "Java");
    f.putFavorite(Integer.class, 0xcafebabe);
    f.putFavorite(Class.class, Favorites.class);
    String favoriteString = f.getFavorite(String.class);
    int favoriteInteger = f.getFavorite(Integer.class);
    Class<?> favoriteClass = f.getFavorite(Class.class);
    System.out.printf("%s %x %s%n", favoriteString,
        favoriteInteger, favoriteClass.getName());
}
```

正如所料，这段程序打印出的是 `Java cafebabe Favorites` 。

`Favorites` 实例是类型安全（typesafe）的：当你向它请求 `String` 的时候，它从来不会返回一个 `Integer` 给你。同时它也是异构的（*heterogeneous*）：不像普通的map，它的所有键都是不同类型的。因此，我们将 `Favorites` 称作类型安全的异构容器（typesafe heterogeneous container）。

`Favorites` 的实现小得出奇。它的完整实现如下：

```
// Typesafe heterogeneous container pattern - implementation
public class Favorites {
    private Map<Class<?>, Object> favorites =
        new HashMap<Class<?>, Object>();

    public <T> void putFavorite(Class<T> type, T instance) {
        if (type == null)
            throw new NullPointerException("Type is null");
        favorites.put(type, instance);
    }

    public <T> getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

这里发生了一些微妙的事情。每个 `Favorites` 实例都得到一个称作 `favorites` 的私有 `Map<Class<?>, Object>` 的支持。你可能认为由于无限通配符类型的关系，将不能把任何东西放进这个 `Map` 中，但事实正好相反。要注意的是通配符类型是嵌套的：它不是属于通配符类型的 `Map` 的类型，而是它的键的类型。由此可见，每个键都可以有一个不同的参数化类型：一个可以是 `Class<String>` ，接下来是 `Class<Integer>` 等等。异构就是从这里来的。

第二件要注意的事情是，`favorites Map` 的值类型只能是 `Object` 。换句话说，`Map` 并不能保证键和值之间的类型关系，即不能保证每个值的类型都与键的类型相同。事实上，`Java` 的类型系统还没有强大到足以表达这一点。但我们知道这是事实，并在获取 `favorite` 的时候利用

了这一点。

putFavorite 方法的实现很简单：它只是把（从指定的 Class 对象到指定的 favorite 实例的）一个映射放到 favorites 中。如前所述，这是放弃了键和值之间的“类型联系”，因此无法知道这个值是键的一个实例。但是没关系，因为 getFavorite 方法能够并且的确重新建立了这种联系。

getFavorite 方法的实现比 putFavorite 的更难一些。它先从 favorites 映射中获得与指定 Class 对象相对应的值。这正是要返回的对象引用，但它的编译时类型是错误的。它的类型只是 Object （ favorites 映射的值类型），我们需要返回一个 T 。因此， getFavorite 方法的实现利用 Class 的 cast 方法，将对象引用动态地转换（dynamically cast）成了 Class 对象所表示的类型。

cast 方法是 Java 的 cast 操作符的动态模拟。它只检验它的参数是否为 Class 对象所表示的类型的实例。如果是，就返回参数；否则就抛出 ClassCastException 异常。我们知道， getFavorite 中的 cast 调用永远也不会抛出 ClassCastException 异常，并假设客户端代码正确无误地进行了编译。也就是说，我们知道 favorites 映射中的值会始终与键的类型相匹配。

假设 cast 方法只返回它的参数，那它能为我们做什么呢？ cast 方法的签名充分利用了 Class 类型被泛型化的这个事实。它的返回类型是 Class 对象的类型参数：

```
public class Class<T> {
    T cast(Object obj);
}
```

这正是 getFavorite 方法所需要的，也正是让我们不必借助于未受检地转换成 T 就能确保 Favorites 类型安全的东西。

Favorites 类有两种局限性值得注意。首先，恶意的客户端可以很轻松地破坏 Favorites 实例的类型安全，只要以它的原生态类型（ray form）使用 Class 对象。但会造成客户端代码在编译时产生未受检的警告。这与一般的集合实现，如 HashSet 和 HashMap 并没有什么区别。也就是说，如果愿意付出一点点代价，就可以拥有运行时的类型安全。确保 Favorites 永远不违背它的类型约束条件的方式是，让 putFavorite 方法检验 instance 是否真的是 type 所表示的类型的实例。我们已经知道这要如何进行了，只要使用一个动态的转换：

```
// Achieving runtime type safty with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

`java.util.Collections` 中有一些集合包装类采用了同样的技巧。它们称作 `checkedSet` 、 `checkedList` 、 `checkedMap` ，诸如此类。除了一个集合（或者映射）之外，它们的静态工厂还采用一个（或者两个） `Class` 对象。静态工厂属于泛型方法，确保 `Class` 对象和集合的编译时类型相匹配。包装类给它们所封装的集合增加了具体化。例如，如果有人视图将 `Coin` 放进你的 `Collection<Stamp>` ，包装类就会在运行时抛出 `ClassCastException` 异常。用这些包装类在混油泛型和遗留代码的应用程序中追溯“谁把错误的类型元素添加到了集合中”很有帮助。

`Favorites` 类的第二种局限性在于它不能用在不可具体化的（non-reifiable）类型中（见第25条）。换句话说，你可以保存最喜爱的 `String` 或者 `String[]` ，但不能保存最喜爱的 `List<String>` 。如果试图保存最喜爱的 `List<String>` ，程序就不能进行编译.原因在于你无法为 `List<String>` 获得一个 `Class` 对象： `List<String>.class` 是个语法错误，这也是件好事。 `List<String>` 和 `List<Integer>` 共用一个 `Class` 对象，即 `List.class` 。如果从“字面（type literal）”上来看， `List<String>.class` 和 `List<Integer>.class` 是合法的，并返回了相同的对象引用，就会破坏 `Favorites` 对象的内部结构。

对于第二种局限性，还没有完全令人满意的解决办法。有一种方法称作 super type token，它在解决这一局限性方面做了很多努力，但是这种方法仍然有它自身的局限性[Gafter07]。

`Favorites` 使用的类型令牌（type token）是无限制的： `getFavorite` 和 `putFavorite` 接受任何 `Class` 对象。有时候，可能需要限制那些可以传给方法的类型。这可以通过有限制的类型令牌（bounded type token）来实现，它只是一个类型令牌，利用有限制类型参数（见第27条）或者有限制通配符（见第28条），来限制可以表示的类型。

注解API（见第35条）广泛利用了有限制的类型令牌。例如，这是一个在运行时读取注解的方法。这个方法来自 `AnnotatedElement` 接口，它通过表示类、方法、域及其他程序元素的反射类型来实现：

```
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```

参数 `annotationType` 是一个表示注解类型的有限制的类型令牌。如果元素有这种类型的注解，该方法就将它返回，如果没有，则返回 `null` 。被注解的元素本质上是个类型安全的异构容器，容器的键属于注解类型。

假设你有一个类型 `Class<?>` 的对象，并且想将它传递给一个需要有限制的类型令牌的方法，例如 `getAnnotation` 。你可以将对象转化成 `Class<? extends Annotation>` ，但是这种转换时非受检的，因此会产生一条编译时警告（见第24条）。幸运的是，类 `Class` 提供了一个安全（且动态）地执行这种转换的实例方法。该方法称作 `asSubclass` ，它将调用它的 `Class` 对象转换成用其他参数表示的类的一个子类。如果转换成功，该方法返回它的参数；如果失败，则抛出 `ClassCastException` 异常。

以下示范了如何利用 `asSubclass` 方法在编译时读取类型未知的注解。这个方法编译时没有出现错误或者警告：

```
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element,
                                String annotationTypeName) {
    Class<?> annotationType = null; // Unbounded type token
    try {
        annotationType = Class.forName(annotationTypeName);
    } catch (Exception ex) {
        throw new IllegalArgumentException(ex);
    }
    return element.getAnnotation(
        annotationType.asSubclass(Annotation.class));
}
```

总而言之，集合API说明了泛型的一般用法，限制你每个容器只能有固定数目的类型参数。你可以通过将类型参数放在键上而不是容器上类避开这一限制。对于这种类型安全的异构容器，可以用 `Class` 对象作为键。以这种方式使用的 `Class` 对象称作类型令牌。你也可以使用定制的键类型。例如，用一个 `DatabaseRow` 类型表示一个数据库行（容器），用泛型 `Column<T>` 作为它的键。

第6章 枚举和注解

Java 1.5 发行版本中增加了两个新的引用类型家族：一种新的类称作枚举类型（enum type），一种新的接口称作注解类型（annotation type）。本章讨论使用这两个新的类型家族的最佳实践。

第30条：用 `enum` 代替 `int` 常量

枚举类型（*enum type*）是指由一组固定的常量组成合法值的类型，例如一年中的季节、太阳系中的行星或者一副牌中的花色。在编程语言还没有引入枚举类型之前，表示枚举类型的常用模式是声明一组具名的 `int` 常量，每个类型成员一个常量：

```
// The int enum pattern - severely deficient!
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

这种方法称作*int*枚举模式（int enum pattern），存在着诸多不足。它在类型安全性和使用方便性方面没有任何帮助。如果你将 apple 传到想要 orange 的方法中，编译器也不会出现警告，还会使用 `==` 操作符将 apple 与 orange 进行对比，甚至更糟糕：

```
// Tasty citrus flavored applesauce!
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

注意每个 apple 常量的名称都以 `APPLE_` 作为前缀，每个 orange 常量则都以 `ORANGE_` 作为前缀。这是因为 Java 没有为 `int` 枚举组提供命名空间。当两个 `int` 枚举组具有相同的明明常量时，前缀可以防止名称发生冲突。

采用 `int` 枚举模式的程序是身份脆弱的。因为 `int` 枚举是编译时常量，被编译到使用它们的客户端中。如果域枚举常量关联的 `int` 发生了变化，客户端就必须重新编译。如果没有重新编译，程序还是可以运行，但是它们的行为就是不确定的。

将 `int` 枚举常量翻译成可打印的字符串，并没有很便利的方法。如果将这种常量打印出来，或者从调试器中将它显示出来，你所见到的就是一个数字，这没有太大的用处。要遍历一个组中的所有 `int` 枚举常量，甚至获得 `int` 枚举组的大小，这些都没有很可靠的方法。

你还可能碰到这种模式的变体，在这种模式中使用的是 `String` 常量，而不是 `int` 常量。这样的变体被称作 `String` 枚举模式，同样也是我们最不期望的。虽然它为这些常量提供了可打印的字符串，但是它会导致性能问题，因为它依赖于字符串的比较操作。更糟糕的是，它会导致初级用户把字符串创两硬编码到客户端代码中，而不是使用适当的域（field）名。如果这样的硬编码字符串常量中包含有书写错误，那么，这样的错误在编译时不会被检测到，但是在运行的时候却会报错。

幸运的是，从 Java 1.5 发行版本开始，就提出了另一种可以替代的解决方案，可以避免 `int` 和 `String` 枚举模式的缺点，并提供许多额外的好处。这就是（JLS， 8.9）。下面以最简单的形式演示了这种模式：

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

表面上看来，这些枚举类型与其他语言中的没有什么两样，例如 C、C++ 和 C#，但是实际上并非如此。Java 的枚举类型是功能十分齐全的类，功能比其他语言中的对等物要强大得多，Java 的枚举本质上是 `int` 值。

> 注：此处原书翻译有误。原文为：
>
> > Java’s enum types are full-fledged classes, far more powerful than their counterparts in these other languages, where enums are essentially int values.
>
> 应翻译为：Java 的枚举类型是功能齐全的类，远比其他语言中的对等物要强大得多，那些语言中的枚举本质上仍然是 int 值。
>
> 感谢 @shuangmulin45 指出

Java 枚举类型背后的基本想法非常简单：它们就是通过公有的静态 `final` 域为每个枚举常量导出实例的类。因为没有可以访问的构造器，枚举类型是真正的 `final` 。因为客户端既不能创建枚举类型的实例，也不能对它进行扩展，因此很可能没有实例，而只有声明过的枚举常量。换句话说，枚举类型是实例受控的。它们是单例（Singleton）的泛型化（见第3条），本质上是单元素的枚举。对于熟悉本书第一版的读者来说，枚举类型为类型安全的枚举（*typesafe enum*）模式[Bloch01，见第21条]提供了语言方面的支持。

枚举提供了编译时的类型安全。如果声明一个参数的类型为 `Apple` ，就可以保证，被传到该参数上的任何非 `null` 的对象引用一定属于三个有效的 `Apple` 值之一。试图传递类型错误的值时，会导致编译时错误，就像试图将某种枚举类型的表达式赋给另一种枚举类型的变量，或者试图利用 `==` 操作符比较不同枚举类型的值一样。

包含同名常量的多个枚举类型可以在一个系统中和平共处，因为每个类型都有自己的命名空间。你可以增加或者重新排列枚举类型中的常量，而无需重新编译它的客户端代码，因为导出常量的域在枚举类型和它的客户端之间提供了一个隔离层：常量值并没有被编译到客户端代码中，而是在 `int` 枚举模式中。最终，可以通过调用 `toString` 方法，将枚举转换成可打印的字符串。

除了完善了 `int` 枚举模式的不足之处，枚举类型还允许添加人的方法和域，并实现任意的接口。它们提供了所有 `Object` 方法（见第3章）的高级实现，实现了 `Comparable` （见第12条）和 `Serializable` 接口（见第11章），并针对枚举类型的可任意改变性设计了序列化方式。

那么我们为什么要将方法或者域添加到枚举类型中呢？首先，你可能是想将数据与它的常量关联起来。例如，一个能够返回水果颜色或者返回水果图片的方法，对于我们的 `Apple` 和 `Orange` 类型来说可能很有好处。你可以利用任何适当的方法来增强枚举类型。枚举类型可以先作为枚举常量的一个简单集合，随着时间的推移再演变成为全功能的抽象。

举个有关枚举类型的好例子，比如太阳系中的8颗行星。每颗行星都有质量和半径，通过这两个属性可以计算出它的表面重力。从而给定物体的质量，就可以计算出一个物体在行星表面上的重量。下面就是这个枚举。每个枚举常量后面括号中的数值就是传递给构造器的参数。在这个例子中，它们就是行星的质量和半径：

```
// Enum type with data and behavior
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS  (4.869e+24, 6.052e6),
    EARTH  (5.975e+24, 6.378e6),
    MARS   (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.566e7),
    NEPTUNE(1.024e+26, 2.477e7);
    private final double mass;           // In kilograms
    private final double radius;         // In meters
    private final double surfaceGravity; // In m / s^2

    // Universal gravitational constant in m^3 / kg s^2
    private static final double G = 6.67300E-11;

    // Constructor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }

    public double mass()           { return mass; }
    public double radius()         { return radius; }
    public double surfaceGravity() { return surfaceGravity; }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity; // F = ma
    }
}
```

编写一个像 `Planet` 这样的枚举类型并不难。为了将数据与枚举常量关联起来，得声明实例域，并编写一个带有数据并将数据保存在域中的构造器。枚举天生就是不可变的，因此所有的域都应该为 `final` 的（见第15条）。它们可以是公有的，但最好将它们做成是私有的，并

提供公有的访问方法（见第14条）。在 `Planet` 这个示例中，构造器还计算和保存表面重力，但这正是一种优化。每当 `surfaceWeight` 方法用到重力时，都会根据质量和半径重新计算，并返回它在该常量所表示的行星上的重量。

虽然 `Planet` 枚举很简单，它的功能却强大的初期。下面是一个简短的程序，根据某个物体在地球上的重量（以任何单位），打印出一张很棒的表格，显示出该物体在所有8颗行星上的重量（用相同的单位）：

```
public class WeightTable {
    public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight / Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Weight on %s is %f%n",
                              p, p.surfaceWeight(mass));
    }
}
```

注意 `Planet` 就像所有的枚举一样，它有一个静态的 `values` 方法，按照声明顺序返回它的值数组。还要注意 `toString` 方法返回每个枚举值的声明名称，使得 `println` 和 `printf` 的打印变得更加容易。如果你还不满足这种字符串表示法，可以通过覆盖 `toString` 方法对它进行修改。下面就是用命令行参数 175 运行这个小小的 `WeightTable` 程序时的结果：

```
Weight on MERCURY is 66.133672
Weight on VENUS is 158.383926
Weight on EARTH is 175.000000
Weight on MARS is 66.430699
Weight on JUPITER is 442.693902
Weight on SATURN is 186.464970
Weight on URANUS is 158.349709
Weight on NEPTUNE is 198.846116
```

如果这是你第一次在实践中见到 Java 的 `printf` 方法，要注意它与C语言的区别，你在这里用的是 `%n` ，在 C 中则用 `\n` 。

与枚举常量关联的有些行为，可能只需要用在定义了枚举的类或者包中。这种行为最好被是现成私有的或者包级私有的方法。于是，每个枚举常量都带有一组隐蔽的行为，这使得包含该枚举的类或者包在遇到这种常量时都可以做出适当的反应。就像其他的类一样，除非迫不得已要将枚举方法导出到它的客户端，否则都应该将它声明为私有的，如有必要，则声明为包级私有的（见第13条）。

如果一个枚举具有普遍适用性，它就应该成为一个顶层类（top-level class）；如果它只是被用在一个特定的顶层类中，它就应该成为该顶层类的一个成员类（见第22条）。例如，`java.math.RoundingMode` 枚举表示十进制小数的舍入模式（rounding mode）。这些舍入

模式用于 `BigDecimal` 类，但是它们提供了一个非常有用的抽象，这种抽象本质上又不属于 `BigDecimal` 类。通过使 `RoundingMode` 成为一个顶层类，库的设计者鼓励任何需要舍入模式的程序员重用这枚举，从而增强API之间的一致性。

`Planet` 示例中所示的方法对于大多数枚举类型来说就足够了，但你有时候会需要更多的方法。每个 `Planet` 常量都关联了不同的数据，但你有时需要将本质上不同的行为（*behavior*）与每个常量关联起来。例如，假设你在编写一个枚举类型，来表示计算器的四大基本操作（即加减乘除），你想要提供一个方法来执行每个常量所表示的算术运算。有一种方法是通过启用枚举的值来实现：

```
// Enum type that switches on its own value - questionable
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    // Do th arithmetic op represented by this constant
    double apply(double x, double y) {
        switch(this) {
          case PLUS:   return x + y;
          case MINUS:  return x - y;
          case TIMES:  return x * y;
          case DIVIDE: return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

这段代码可行，但是不太好看。如果没有 `swithc` 语句，它就不能编译，虽然从技术角度来看代码的结束部分是可以执行到的，但是实际上是不可能执行到这行代码的[JLS，14.2.1]。更糟糕的是，这段代码很脆弱。如果你添加了新的枚举常量，却忘记给 `switch` 添加相应的条件，枚举仍然可以编译，但是当你试图运行新的运算时，就会运行失败。

幸运的是，有一种更好的方法可以将不同的行为与每个枚举常量关联起来：在枚举类型中声明一个抽象的 `apply` 方法，并在特定于常量的类主题（*constant-specific class body*）中，用具体的方法覆盖每个常量的抽象 `apply` 方法。这种方法被称作特定于常量的方法实现（*constant-specific method implementation*）：

```
// Enum type with constant-specific method implementations
public enum Operation {
    PLUS   { double apply(double x, double y){return x + y;} },
    MINUS  { double apply(double x, double y){return x - y;} },
    TIMES  { double apply(double x, double y){return x * y;} },
    DIVIDE { double apply(double x, double y){return x / y;} };

    abstract double apply(double x, double y);
}
```

如果给 `Operation` 的第二种版本添加新的常量，你就不可能会忘记提供 `apply` 方法，因为该方法就紧跟在每个常量声明之后。即使你真的忘记了，编译器也会提醒你，因为没居中的抽象方法必须被它所有的常量中的具体方法所覆盖。

特定于常量的方法实现可以与特定于常量的数据结合起来。例如，下面的 `Operation` 覆盖了 `toString` 来返回通常与该操作关联的符号：

```
// Enum type with constant-specific class bodies and data
public enum Operation {
    PLUS("+") {
        double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    abstract double apply(double x, double y);
}
```

在有些情况下，在枚举中覆盖 `toString` 非常有用。例如，上述的 `toString` 实现使得打印算术表达式变得非常容易，如这段小程序所示：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Opetation.values())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

用 2 和 4 作为命令行参数运行这段程序后，会输出：

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

枚举类型有一个自动产生的 `valueOf(String)` 方法，它将常量的名字转变成为常量本身。如果在枚举类型中覆盖 `toString` ，要考虑编写一个 `fromString` 方法，将定制的字符串表示法变回相应的枚举。下列代码（适当地改变了类型名称）可以为任何枚举完成这一技巧，只要每个常量都有一个独特的字符串表示法：

```
// Implementation a fromString method on an enum type
private static final Map<String, Operation> stringToEnum
    = new HashMap<String, Operation>();
static { // Initialize map from contant name to enum constant
    for (Operation op : values())
        stringToEnum.put(op.toString(), op);
}
// Returns Operation for String, or null if string is invalid
public static Operation fromString(String symbol) {
    return stringToEnum.get(symbol);
}
```

注意，在常量被创建之后， `Operation` 常量从静态代码块中被放入到了 `stringToEnum` 的 `map` 中。试图使每个变量都从自己的构造器将自身放入到 `map` 中，会导致编译时错误。这是好事，因为如果这是合法的，就会抛出 `NullPointerException` 异常。枚举构造器不可以访问枚举的静态域，除了编译时常量之外。这一限制是有必要的，因为构造器运行的时候，这些静态域还没有被初始化。

特定于常量的方法实现有一个美中不足的地方，它们使得在枚举常量中共享代码变得更加困难了。例如，考虑用一个枚举表示薪资包中的工作天数。这个枚举有一个方法，根据给定某工人的基本工资（按小时）以及当天的工作时间，来计算他当天的报酬。在五个工作日中，超过正常八小时的工作时间都会产生加班工资；在双休日中，所有工作都产生加班工资。利用 `switch` 语句，很容易通过将多个 `case` 标签分别应用到两个代码片段中，来完成这一计算。为了简洁起见，这个示例中的代码使用了 `double` ，但是注意 `double` 并不是适合薪资应用程序（见第48条）的数据类型。

```
// Enum that switches on its value to share code - questionable
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;
    private static final int HOURS_PER_SHIFT = 8;
    double pay(double hoursWorked, double payRate) {
        double basePay = hoursWorked * payRate;

        double overtimePay; // Calculate overtime pay
        switch(this) {
          case SATURDAY: case SUNDAY
            overtimePay = hoursWorked * payRate / 2;
          default: // Weekdays
            overtimePay = hoursWorked <= HOURS_PER_SHIFT ?
              0 : (hoursWorked - HOURS_PER_SHIFT) * payRate / 2;
            break;
        }

        return basePay + overtimePay;
    }
}
```

不可否认，这段代码十分简洁，但是从维护的角度来看，它十分危险。假设将一个元素添加到该枚举中，或许是一个表示假期天数的特殊值，但是忘记给 `switch` 语句添加相应的 `case` 。程序依然可以编译，但 `pay` 方法会悄悄地将假期的工资计算成与正常工作日的相同。

为了利用特定于常量的方法实现安全地执行工资计算，你可能必须重复计算每个常量的加班工资，或者将计算移到两个辅助方法中（一个用来计算工作日，一个用来计算双休日），并从每个常量调用相应的辅助方法。这任何一种方法都会产生相当数量的样板代码，结果降低了可读性，并增加了出错的机率。

通过用计算工作日加班工资的具体方法代替 `PayrollDay` 中抽象的 `overtimePay` 方法，可以减少样板代码。这样，就只用双休日必须覆盖该方法了。但是这样也有着与 `switch` 语句一样的不足：如果有增加了一天而没有覆盖 `overtimePay` 方法，就会悄悄地延续工作日的计算。

你真正想要的就是每当添加一个枚举常量时，就强制选择一种加班报酬策略。幸运的是，有一种很好的方法可以实现这一点。这种想法就是将加班工资计算移到一个私有的嵌套枚举中，将这个策略枚举（*strategy enum*）的实例传到 `PayrollDay` 枚举的构造器中。之后 `PayrollDay` 枚举将加班工资计算委托给策略枚举， `PayrollDay` 中就不需要 `switch` 语句或者特定于常量的方法实现了。虽然这种模式没有 `switch` 语句那么简洁，但更加安全，也更加灵活：

```
// The strategy enum pattern
enum PayrollDay {
    MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),
    WEDNESDAY(PayType.WEEKDAY),THURSDAY(PayType.WEEKDAY),
    FRIDAY(PayType.WEEKDAY),
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private pay(double hoursWorked, double payRate) {
        return payType.pay(hoursWorked, payRate);
    }
    // The strategy enum type
    private enum PayType {
        WEEKDAY {
            double overtimePay(double hours, double payRate) {
                return hours <= HOURS_PER_SHIFT ? 0 :
                  (hours - HOURS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            double overtimePay(double hours, double payRate) {
                return hours * payRate / 2;
            }
        };
        private static final int HOURS_PER_SHIFT = 8;

        abstract double overtimePay(double hours, double payRate);

        double pay(double hoursWorked, double payRate) {
            double basePay = hoursWorked * payRate;
            return basePay + overtimePay(hoursworked, payRate);
        }
    }
}
```

如果枚举中的 `switch` 语句不是在没居中实现特定于常量的行为的一种很好的选择，那么它们还有什么用处呢？枚举中的 *`switch`* 语句适合于给外部的枚举类型增加特定于常量的行为。例如，假设 `Operation` 枚举不受你的控制，你希望它有一个实例方法来返回每个运算的反运算。你可以用下列静态方法模拟这种效果：

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
    switch(op) {
      case PLUS:   return Operation.MINUS;
      case MINUS:  return Operation.PLUS;
      case TIMES:  return Operation.DIVIDE;
      case DIVIDE: return Operation.TIMES;
      default: throw new AssertionError("Unkown op: " + op);
    }
}
```

一般来说，枚举会优先使用 `comparable` 而非 `int` 常量。与 `int` 常量相比，枚举有个小小的性能缺点，即装在和初始化枚举时会有空间和时间的成本。除了受资源约束的设备，例如手机和烤面包机之外，在实践中不必太在意这个问题。

那么什么时候应该使用枚举呢？每当需要一组固定常量的时候。当然，这包括“天然你的枚举类型”，例如行星、一周的天数以及棋子的数目等等。但它也包括你在编译时就知道其所有可能值的其他集合，例如菜单的选项、操作代码以及命令行标记等。枚举类型中的常量集并不一定要始终保持不变。专门设计枚举特性是考虑到枚举类型的二进制兼容演变。

总而言之，与 `int` 常量相比，枚举类型的优势是不言而喻的。枚举要易读得多，也更加安全，功能更加强大。许多枚举都不需要显式的构造器或者成员，但许多其他枚举则受益于“每个常量与属性的关联”以及“提供行为受这个属性影响的方法”。只有极少数的枚举受益于将多种行为与单个方法关联。在这种相对少见的情况下，特定于常量的方法要优先于启用自有值的枚举。如果多个枚举常量同时共享相同的行为，则考虑策略枚举。

第31条：用实例域代替序数

许多枚举天生就与一个单独的 `int` 值相关联。所有的枚举都有一个 `ordinal` 方法，它返回每个枚举常量在类型中的数字位置。你可以试着从序数中得到关联的 `int` 值：

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS
public enum Ensemble {
    SOLO,   DUET,   TRIO,  QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET,   DECTET;

    public int numberOfMusicians{ return ordinal() + 1; }
}
```

虽然这个枚举不错，但是维护起来就想一场恶梦。如果常量进行重新排序，`numberOfMusicians` 方法就会遭到破坏。如果要再添加一个与已经用过的 `int` 值关联的枚举常量，就没那么走运了。例如，给双四重奏（*double quartet*）添加一个常量，它就像个八重奏一样，是由8位演奏家组成，但是没有办法做到。

要是没有给所有这些 `int` 值添加常量，也无法给某个 `int` 值添加常量。例如，假设想要添加一个常量表示三四重奏（*triple quartet*），它由12位演奏家组成。对于由11位演奏家组成的合奏曲并没有标准的术语，因此只好给没有用过的 `int` 值（11）添加一个虚拟（dummy）常量。这么做顶多就是不太好看。如果有许多 `int` 值都是从未用过的，可就不切实际了。

幸运的是，有一种很简单的方法可以解决这些问题。永远不要根据枚举的序数导出与它关联的值，而是要将它保存在一个实例域中：

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) { this.numberOfMusicians = size; }
    public int numberOfMusicians{ return numberOfMusicians; }
}
```

`Enum` 规范中谈到 `ordinal` 时这么写到：

> “大多数程序员都不需要这个方法。它是设计成用于像 `EnumSet` 和 `EnumMap` 这种基于枚举的通用数据结构的。”

除非你在编写的是这种数据结构，否则最好完全避免使用 `ordinal` 方法。

第32条：用 `EnumSet` 代替位域

如果一个枚举类型的元素主要用在集合中，一般就是用 `int` 枚举模式（见第30条），将 2 的不同倍数赋予每个常量：

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
    public static final int STYLE_BOLD          = 1 << 0; // 1
    public static final int STYLE_ITALIC        = 1 << 1; // 2
    public static final int STYLE_UNDERLINE     = 1 << 2; // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8

    // parameter is bitwise OR of zero or more STYLE_ constants
    public void applyStyles(int styles) { ... }
}
```

这种表示法让你用 `OR` 位运算将几个常量合并到一个集合中，称作位域（*bit field*）：

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

位域表示法也允许利用位操作，有效地执行像 union（联合）和 intersection（交集）这样的集合操作。但位域有着 `int` 枚举常量的所有缺点，甚至更多。当位域以数字形式打印时，翻译位域比翻译简单的 `int` 枚举常量要困难得多。甚至，要遍历位域表示的所有元素也没有很容易的方法。

有些程序员优先使用枚举而非 `int` 常量，他们在需要传递多组常量集时，仍然倾向于使用位域。其实没有理由这么做，因为还有更好的替代方法。`java.util` 包提供了 `EnumSet` 类来有效地表示从单个枚举类型中提取的多个值的多个集合。这个类实现 `Set` 接口，提供了丰富的功能、类型安全性，以及可以从任何其他 `Set` 实现中得到的互用性。但是在内部具体的实现上，每个 `EnumSet` 内容都表示为位矢量。如果底层的枚举类型有64个或者更少的元素——大多如此——整个 `EnumSet` 就是用单个 `long` 来表示，因此它的性能比得上位域的性能。批处理，如 `removeAll` 和 `retainAll`，都是利用位算法来实现的，就像手工替位域实现得那样。但是可以避免手工操作时容易出现的错误以及不太雅观的代码，因为 `EnumSet` 替你完成了这项艰巨的工作。

下面是一个范例改成用枚举代替位域后的代码，它更加简短、更加清楚，也更加安全：

```
// EnumSet - a modern replacement for bit fields
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) { ... }
}
```

下面是将 `EnumSet` 实例传递给 `applyStyles` 方法的客户端代码。 `EnumSet` 提供了丰富的静态工厂来轻松创建集合，其中一个如这个代码所示：

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

注意 `applyStyles` 方法采用的是 `Set<Styles>` 而非 `EnumSet<Style>` 。虽然看起来好像所有的客户端都可以将 `EnumSet` 传递到这个方法，但是最好还是接受接口类型而非接受实现类型。这是考虑到可能会有特殊的客户端要传递一些其他的 `Set` 实现，并且没有什么明显的缺点。

总而言之，正式因为枚举类型要用在集合（*Set*）中，所以没有利用用位域来表示它。 `EnumSet` 类集位域的简洁和性能优势及第 30 条中所述的枚举类型的所有优点于一身。实际上 `EnumSet` 有个缺点，即截止 Java 1.6 发行版本，它都无法创建不可变的 `EnumSet` ，但是这一点很可能在即将出来的版本中得到修正。同时，可以用 `Collections.unmodifiableSet` 将 `EnumSet` 封装起来，但是简洁性和性能会受到影响。

第33条：用 `EnumMap` 代替序数索引

有时候，你可能会见到利用 `ordinal` 方法（见第31条）来索引数组的代码。例如下面这个过于简化的类，用来表示一种烹饪用的香草：

```
public class Hurb {
    public enum Type { ANNUAL, PERENNIAL, BIENNIAL }

    private final String name,
    private final Type type;

    Herb(String name, Type type) {
        this.name = name;
        this.type = type;
    }

    @Override public String toString() {
        retuan name;
    }
}
```

假设现在有一个香草的数组，表示一座花园中的植物，你想要按照类型（一年生、多年生或者两年生植物）进行组织之后将这些植物列出来。如果要这么做的话，需要构建三个集合，每种类型一个，并且遍历整座花园，将每种香草放到相应的集合中。有些程序员会将这些集合放到一个按照类型的序数进行索引的数组中来实现这一点。

```
// Using ordinal() to index an array - DON'T DO THIS!
Herb[] garden = ...;

Set<Herb>[] herbsByType = // Indexed by Herb.type.ordinal()
    (Set<Herb>[]) new Set[Herb.Type.values().lenght];
for(int i = 0; i < herbsByType.length; i++)
    herbsByType[i] = new HashSet<Herb>();

for(Herb h : garden)
    herbsByTyp[h.type.ordinal()].add(h);

// Print the results
for(int i = 0; i < herbsByType.length; i++) {
    System.out.printf("%s: %s%n",
                      Herb.Type.values()[i], herbsByType[i]) ;
}
```

这种方法的确可行，但是隐藏着许多问题。因为数组不能与泛型（见第25条）兼容，程序需要进行未受检的转换，并且不能进行正确无误的编译。因为数组不知道它的索引代表着什么，你必须手工标注（label）这些索引的输出。但是这种方法最严重的问题在于，当你访问一个按照枚举的序数进行索引的数组时，使用正确的 `int` 值就是你的职责了；`int` 不能提供枚举的类型安全。你如果使用了错误的值，程序就会悄悄地完成错误的工作，或者幸运的话，会抛出 `ArrayIndexOutOfBoundException` 异常。

幸运的是，有一种更好的方法可以达到同样的效果。数组实际上充当着从枚举到值的映射，因此可能还要用到 `Map`。更具体地说，有一种非常快速的 `Map` 实现专门用于枚举键，称作 `java.util.EnumMap`。以下就是用 `java.util.EnumMap` 改写后的程序：

```
// Using an EnumMap to associate data with an enum
Map<Herb.Type, Set<Herb>> herbsByType =
    new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class);
for (Herb.Type t : Herb.Type.values())
    herbsByType.put(t, new HashSet<Herb>());
for (Herb h : garden)
    herbsByType.get(h.type).add(h);
System.out.println(herbsByType);
```

这段程序更简短、更清楚，也更加安全，运行速度方面可以与使用序数的程序相媲美。它没有不安全的转换；不必手工标注这些索引的输出，因为映射键知道如何将自身翻译成可打印字符串的枚举；计算数组索引时也不可能出错。`EnumMap` 在运行速度方面之所以能与通过序数索引的数组相媲美，是因为 `EnumMap` 在内部使用了这种数组。但是它对程序员隐藏了这种实现细节，集 `Map` 的丰富功能和类型安全与数组的快速与一身。注意 `EnumMap` 构造器采用键类型的 `Class` 对象：这是一个有限制的类型令牌（bounded type token），它提供了运行时的泛型信息（见第29条）。

你还可能见到按照序数进行索引（两次）的数组的数组，该序数表示两个枚举值的映射。例如，下面这个程序就是使用这样一个数组将两个阶段映射到一个阶段过渡中（从液体到固体称作凝固，从液体到气体称作沸腾，诸如此类）。

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase { SOLID, LIQUID, GAS;
    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;
        // Rows indexed by src-ordinal, cols by dst-ordinal
        private static final Transition[][] TRANSITIONS = {
          { null,    MELT,     SUBLIME },
          { FREEZE,  null,     BOIL    },
          { DEPOSIT, CONDENSE, null    }
        };

        // Returns the phase transition from one phase to another
        public static Transition from(Phase src, Phase dst) {
            return TRANSITIONS[src.ordinal()][dst.ordinal()];
        }
    }
}
```

这段程序可行，看起来也比较优雅，但是事实并非如此。就想上面那个比较简单的香草花园的示例一样，编译器无法知道序数和数组索引之间的关系。如果在过渡表中出了错，或者在修改 `Phase` 或者 `Phase.Transition` 枚举类型的时候忘记将它更新，程序就会在运行时失败。这种失败的形式可能为 `ArrayIndexOutOfBoundsException` 、 `NullPointerException` 或者（更糟糕的是）没有任何提示的错误行为。这张表的大小是阶段个数的平方，即使非 `null` 项的数量比较少。

同样，利用 `EnumMap` 依然可以做得更好一些。因为每个阶段过渡都是通过一对阶段枚举进行索引的，最好将这种关系表示为一个 `map` ，这个 `map` 的键是一个枚举（其实阶段），值为另一个 `map` ，这第二个 `map` 的键为第二个枚举（目标阶段），它的值为结果（阶段过渡），即形成了 `Map` （其实阶段， `Map` （目标阶段，阶段过渡））这种形式。一个阶段过渡所关联的两个阶段，最好通过“数据与阶段过渡枚举之间的关联”来获取，之后用该阶段过渡枚举来初始化嵌套的 `EnumMap` 。

```
// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

        private final Phase src;
        private final Phase dst;

        Transition(Phase src, Phase dst) {
            this.src = src;
            this.dst = dst;
        }
        // Initialize the phase transition map
        private static final Map<Phase, Map<Phase, Transition>> m =
          new EnumMap<Phase, Map<Phase, Transition>>(Phase.class);
        static {
            for (Phase p : Phase.values())
                m.put(p, new EnumMap<Phase, Transition>(Phase.class));
            for (Transition trans : Transition.value())
                m.get(trans.src).put(trans.dst, trans);
        }

        public static Transition from(Phase src, Phase dst) {
            return m.get(src).get(dst);
        }
    }
}
```

初始化阶段过渡 `map` 的代码看起来可能有点复杂，但是还不算太糟糕。`map` 的类型为 `Map<Phase, Map<Phase, Transition>>`，表示是由键为源 `Phase`（即第一个 `Phase`）、值为另一个 `map` 组成的 `Map`，其中组成值的 `Map` 是由键值对目标 `Phase`（即第二个 `Phase`）、`Transition` 组成的。静态初始化代码块中的第一个循环初始化了外部 `map`，得到了三个空的内容 `map`。代码块中的第二个循环利用每个状态过渡常量提供的起始信息和目标信息初始化了内部 `map`。

现在假设想要给系统添加一个新的阶段：*plasma*（离子）或者电离气体。只有两个过渡与这个阶段关联：电离化，它将气体变成离子；以及消电离化，将离子变成气体。为了更新基于数组的程序，必须给 `Phase` 添加一种新常量，给 `Phase.Transition` 添加两种新常量，用一种新的 16 个元素的版本取代原来 9 个元素的数组的数组。如果给数组添加的元素过多或者过少，或者元素放置不妥当，可就麻烦了：程序可以编译，但是会在运行时失败。为了更新基于 `EnumMap` 的版本，所要做的就是必须将 `PLASMA` 添加到 `Phase` 列表，并将 `IONIZE`（`GAS`，`PLASMA`）和 `DEIONIZE`（`PLASMA`，`GAS`）添加到 `Phase.Transition` 的列

表中。程序会自行处理所有其他的事情，你几乎没有机会出错。从内部来看，`Map`的`Map`被实现成了数组的数组，因此在提升了清楚性、安全性和易维护性的同时，在空间或者时间上还几乎不用任何开销。

总而言之，最好不要用序数来索引数组，而要使用*`EnumMap`*。如果你所表示的这种关系是多维的，就使用`EnumMap<..., EnumMap<...>>`。应用程序的程序员在一般情况下都不使用`Enum.ordinal`，即使要用也很少，因此这是一种特殊情况（见第31条）。

第34条：用接口模拟可伸缩的枚举

就几乎所有方面来看，枚举类型都优越于本书第一版中所述的类型安全枚举模式[Bloch01]。从表面上看，有一个异常与可伸缩性有关，这个异常可能处在原来的模式中，却没有得到语言构造的支持。换句话说，使用这种模式，就有可能让一个枚举类型去扩展另一个枚举类型；利用这种语言特性，则不可能这么做。这绝非偶然。枚举的可伸缩性最后证明基本上都不是什么好点子。扩展类型的元素为基本类型的实例，基本类型的实例却不是扩展类型的元素，这样很是混乱。目前还没有很好的方法来枚举基本类型的所有元素及其扩展。最终，可伸缩性会导致设计和实现的许多方面变得复杂起来。

也就是说，对于可伸缩的枚举类型而言，至少有一种具有说服力的用例，这就是操作吗（*operation code*），也称作*opcode*。操作吗是指这样的枚举类型：它的元素表示在某种机器上的那些操作，例如第30条中的 `Operation` 类型，它表示一个简单的计算器中的某些函数。有时候，要尽可能地让 API 的用户提供它们自己的操作，这样可以有效地扩展 API 所提供的操作集。

幸运的是，有一种很好的方法可以利用枚举类型来实现这种效果。由于枚举类型可以通过给操作码类型和（属于接口的标准实现的）枚举定义接口，来实现任意接口，基本的想法就是利用这一事实。例如，以下是第30条中的 `Operation` 类型的扩展版本：

```
// Emulated extensible enum using an interface
public interface Operation {
    double apply(double x, double y);
}

public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    BasicOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}
```

虽然枚举类型（`BasicOperation`）不是可扩展的，但接口类型（`Operation`）则是可扩展的，它是用来表示 API 中的操作的接口类型。你可以定义另一个枚举类型，它实现这个接口，并用这个新类型的实例代替基本类型。例如，假设你想要定义一个上述操作类型的扩展，由求幂（exponentiation）和求余（remainder）操作组成。你所要做的就是编写一个枚举类型，让它实现`Operation`接口：

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    },

    private final String symbol;
    ExtendedOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}
```

在可以使用基础操作的任何地方，都可以使用新的操作，只要 API 是被写成采用接口类型（`Operation`）而非实现（`BasicOperation`）。注意，在枚举中，不必像在不可扩展的枚举中所做的那样，利用特定于实例的方法实现来声明抽象的 `apply` 方法。这是因为抽象的方法（`apply`）是接口（`Operation`）的一部分。

不仅可以在任何需要“基本枚举”的地方单独传递一个“扩展枚举”的实例，而且除了那些基本类型的元素之外，还可以传递完整的扩展枚举类型，并使用它的元素。例如，通过下面这个测试程序，体验一下上面定义过的所有扩展过的操作：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    doubly y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}
private static <T extends Enum<T> & Operation> void test(
        Class<T> opSet, double x, double y) {
    for (Operation op : opSet.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

注意扩展过的操作类型的类的字面文字（`ExtendedOperation.class`）从 `main` 被传递给了 `test` 方法，来描述被扩展操作的集合。这个的字面文字充当有限制的类型令牌（见第29条）。`opSet` 参数中公认很复杂的声明（`<T extends Enum<T> & Operation>`）确保

了 `Class` 对象既表示枚举又表示 `Operation` 的子类型，这正是遍历元素和执行与每个元素相关联的操作时所需要的。

第二种方法是使用 `Collection<? extends Operation>` ，这是个有限制的通配符类型（*bounded wildcard type*）（见第28条），作为 `opSet` 参数的类型：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    doubly y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}
private static void test(Collection<? extends Operation> opSet,
        double x, double y) {
    for (Operation op : opSet.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

这样得到的代码没有那么复杂，`test` 方法也比较灵活一些：它允许调用者将多个实现类型的操作合并到一起。另一方面，也放弃了在指定操作上使用 `EnumSet` （见第32条）和 `EnumMap` （见第33条）的功能，因此，除非需要灵活地合并多个实现类型的操作，否则可能最好使用有限制的类型令牌。

上面这段程序用命令行参数 `2` 和 `4` 运行时，都会产生这样的输出：

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

用接口模拟可伸缩枚举有个小小的不足，既无法将实现从一个枚举类型集成到另一个枚举类型。在上述 `Operation` 的示例中，保存和获取与某项操作相关联的符号的逻辑代码，可以复制到 `BasicOperation` 和 `ExtendedOperation` 中。在这个例子中是可以的，因为复制的代码非常少。如果共享功能比较多，则可以将它封装在一个辅助类或者静态辅助方法中，来避免代码的复制工作。

总而言之，虽然无法编写可扩展的枚举类型，却可以通过编写几口以及实现该接口的基础枚举类型，对它进行模拟。这样允许客户端编写自己的枚举来实现接口。如果 API 是根据接口编写的，那么在可以使用基础枚举类型的任何地方，也都可以使用这些枚举。

第35条：注解优先于命名模式

Java 1.5 发行版本之前，一般使用命名模式（*naming pattern*）表明有些程序元素需要通过某种工具或者框架进行特殊处理。例如，JUnit 测试框架原本要求它的用户一定要用 `test` 作为测试方法名称的开头[Beck04]。这种方法可行，但是有几个很严重的缺点。首先，文字拼写错误会导致失败，且没有任何提示。例如，假设不小心将一个测试方法命名为 `tsetSafetyOverride` 而不是 `testSafetyOverride` 。Junit 不会出错，但也不会执行测试，造成错误的安全感（即测试方法没有执行，它没有报错的可能，从而给人以测试正确的假象）。

命名模式的第二个缺点是，无法确保它们只用于相应的程序元素上。例如，假设将某个类称作 `testSafetyMechanisms` ，是希望 Junit 会自动地测试它所有的方法，而不管它们叫什么名称。Junit 还是不会出错，但也同样不会执行测试。

命名模式的第三个缺点是，它们没有提供将参数值与程序元素关联起来的好方法。例如，假设想要支持一种测试类别，它只在抛出异常时才会成功。异常类型本质上是测试的一个参数。你可以利用某驻具体的命名模式，将异常类型名称编码到测试方法名称中，但是这样的代码会很不雅观，也很脆弱（见第50条）。编译器不知道要去检验准备命名异常的字符串是否眞正命名成功。如果命名的类不存在，或者不是一个异常，你也要到试着运行测试时才会发现。

注解[JLS，9.7]很好地解决了所有这些问题。假设想要定义一个注解类型来指定简单的测试，它们自动运行，并在抛出异常时失败。一下就是这样的一个注解类型，命名为 `Test` ：

```
// Marker annotation type declaration
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

`Test` 注解类型的声明就是它自身通过 `Retention` 和 `Target` 注解进行了注解。注解类型声明中的这种注解被称作元注解（*meata-annotation*）。 `@Retention(RetentionPolicy.RUNTIME)` 元注解表明， `Test` 注解应该在运行时保留。如果没有保留，测试工具就无法知道 `Test` 注解。 `@Target(ElementType.METHOD)` 元注解表明， `Test` 注解只在方法声明中才是合法的：它不能运用到类声明、域声明或者其他程序元素上。

注意 `Test` 注解声明上方的注释：“Use only on parameterless static methods（只用于无参的静态方法）”。如果编译器能够强制这一限制最好，但是它做不到。编译器可以替你完成多少错误检查，这是有限制的，即使是利用注解。如果将 `Test` 注解放在实例方法的声明中，或者放在带有一个或者多个参数的方法中，测试程序还是可以编译，让测试工具在运行时来处理这个问题。

下面就是现实应用中的 `Test` 注解，称作标记注解（*marker annotation*），因为它没有参数，只是“标注”被注解的元素。如果程序员拼错了 `Test` ，或者将 `Test` 注解应用到程序元素而非方法声明，程序就无法编译：

```
// Program containing marker annotations
public class Sample {
    @Test public static void m1() {} // Test should pass
    public static void m2() {}
    @Test public static void m3() {
        throw new RuntimeException("Boom");
    }
    public static void m4() {}
    @Test public void m5() {} // INVALID USE: nonstatic method
    public static void m6() {}
    @Test public static void m7() {  // Test should fail
        throw newe RuntimeException("Crash");
    }
    public static void m8() {}
}
```

`Sample` 类有8个静态方法，其中4个被注解为测试。这4个中有2个抛出了异常： `m3` 和 `m7` ，另外两个则没有： `m1` 和 `m5` 。但是其中一个没有抛出异常的注解方法： `m5` ，是一个实例方法，因此不属于注解的有效使用。总之， `Sample` 包含4项测试：一项会通过，两项会失败，另一项无效。没有用 `Test` 注解进行标注的4个方法会被测试工具忽略。

`Test` 注解对 `Sample` 类的余熠没有直接的影响。它们只负责提供信息供相关的程序使用。更一般的讲，注解永远也不会改变被注解代码的语义，但是使它可以通过工具进行特殊的处理，例如像这种简单的测试运行类：

```
// Program to process marker annotations
import java.lang.reflect.*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " failed: " + exc);
                } catch (Exception exc) {
                    System.out.println("INVALID @Test: " + m);
                }
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n",
                          passed, tests - passed);
    }
}
```

测试运行工具在命令行上使用完全匹配的类名，并通过调用 `Method.invoke` 反射式地运行类中所有标注了 `Test` 的方法。 `isAnnotationPresent` 方法告知该工具要运行哪些方法。如果测试方法抛出异常，反射机制就会将它封装在 `InvocationTargetException` 中。该工具捕捉到了这个异常，并打印失败报告，包含测试方法抛出的原始异常，这些信息是通过 `getCause` 方法从 `InvocationTargetException` 中提取出来的。

如果尝试通过反射调用测试方法时抛出 `InvocationTargetException` 之外的任何异常，表明编译时没有捕捉到 `Test` 注解的无效用法。这种用法包括实例方法的注解，或者带有一个或者多个参数的方法的注解，或者不可访问的方法的注解。测试运行类中的第二个 `catch` 块捕捉到了这些 `Test` 用法错误，并打印出相应的错误消息。下面就是 `RunTests` 在 `Sample` 上运行时打印的输出：

```
public static void Sample.m3() failed: RuntimeException: Boom
INVALID @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed 3
```

现在我们要针对只在抛出特殊异常时才成功的测试添加支持。为此我们需要一个新的注解类型：

```
// Annotation type with a parameter
import java.lang.annotation.*;
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

这个注解的参数类型是 `Class<? extends Exception>` 。这个通配符类型无疑很绕口。它在英语中的意思是：某个扩展 `Exception` 的类的 `Class` 对象，它允许注解的用户指定任何异常类型。这种用法是有限制的类型令牌（见第29条）的一个示例。下面就是实际应用中的这个注解，注意类名称被用作了注解的参数值：

```
// Program containing annotations with a parameter
public class Sample2 {
    @ExceptionTest(ArithmeticException.class)
    public static void m1() { // Test should pass
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() { // Should fail (wrong exception)
        int[] a = new int[0];
        int i = a[i];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { } // Should fail (no exception)
}
```

现在我们要修改一下测试运行工具来处理新的注解。这其中包括将以下代码添加到 `main` 方法中：

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m)
    } catch (InvocationTargetException wrappedEx) {
        Throwable exc = wrappedEx.getCause();
        Class<? extends Exception> excType =
            m.getAnnotation(ExceptionTest.class).value();
        if (excType.isInstance(exc)) {
            passed++;
        } else {
            System.out.printf(
                "Test %s failed: expected %s, got %s%n",
                m, excType.getName(), exc);
        }
    } catch (Exception exc) {
        System.out.println("INVALID @Test: " + m);
    }
}
```

这段代码类似于用来处理 `Test` 注解的代码，但有一处不同：这段代码提取了注解参数的值，并用它检验该测试抛出的异常是否为正确的类型。没有显示的转换，因此没有出现 `ClassCastException` 的危险。编译过的测试程序确保它的注解参数表示的是有效的异常类型，需要提醒一点：有可能注解参数在编译时是有效的，但是表示特定异常类型的类文件在运行时却不再存在。这种希望很少出现的情况下，测试运行类会抛出 `TypeNotFoundException` 异常。

将上面的异常测试示例再深入一点，想像测试可以抛出任何一种指定异常时都得到通过。注解机制有一种工具，使得支持这种用法变得十分容易。假设我们将 `ExceptionTest` 注解的类型参数改成 `Class` 对象的一个数组：

```
// Annotation type with a parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

注解中数组参数的语法十分灵活。它是进行过优化的单元素数组。使用 `ExceptionTest` 新版的数组参数之后，之前的所有 `ExceptionTest` 注解仍然有效，并产生单元素的数组。为了指定多元素的数组，要用花括号（ `{}` ）将元素包围起来，并用逗号（ `,` ）将它们隔开：

```
// Code containning an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class,
                 NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<String>();

    // The spec permits this method to throw either
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```

修改测试运行工具来处理新的`ExceptionTest`相当简单。下面的代码代替了原来的代码。

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m)
    } catch (InvocationTargetException wrappedEx) {
        Throwable exc = wrappedEx.getCause();
        Class<? extends Exception>[] excTypes =
            m.getAnnotation(ExceptionTest.class).value();
        int oldPassed = passed;
        for (Class<? extends Exception> excType : excTypes) {
            if (excType.isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed) {
            System.out.printf("Test %s failed: %s %n", m, exc);
        }
    }
}
```

本条目中开发的测试框架只是一个试验，但它清楚地示范了注解之于命名模式的优越性。它这还只是揭开了注解功能的冰山一角。如果实在编写一个需要程序员给源文件添加信息的工具，就要定义一组适当的注解类型。既然有了注解，就完全没有理由再使用命名模式了。

也就是说，除了“工具铁匠（toolsmiths——特定的程序猿）”之外，大多数程序员都不必定义注解类性。但是所有的程序员都应该使用 *Java* 平台所提供的预定义的注解类型（见第36和24条）。还要考虑使用 IDE 或者静态分析工具所提供的任何注解。这种注解可以提升由这些工具所提供的诊断信息的质量。但是要注意这些注解还没有标准化，因此如果变换工具或者形成标准，就有很多工作要做了。

第36条：坚持使用 `Override` 注解

随着 Java 1.5 发行版本中增加注解，类库中也增加了集中注解类型[JLS，9.6]。对于传统的程序员而言，这里面最重要的就是 `Override` 注解了。这个注解只能用在方法声明中，它表示被注解的方法声明覆盖了超类型中的一个声明。如果坚持使用这个注解，可以防止一大类的非法错误。考虑下面的程序，这里的类 `Bigram` 表示一个双子母组或者有序的字母对：

```
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public void static main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

主程序反复地将26个双字母添加到集合中，每个双子母都由两个相同的小写字母组成。随后打印出集合的大小。你可能以为程序打印出的大小为 26，因为集合不能包含重复。如果你试着运行程序，会发现打印的不是 26 而是 260。哪里出错了呢？

很显然，`Bigram` 类的创建者原本想要覆盖 `equals` 方法（见第8条），同时还记得覆盖了 `hashCode` 。遗憾的是，不幸的程序员没能覆盖 `equals` ，而是将它重载了(见第41条)。为了覆盖 `Object.equals` ，必须定义一个参数为 `Object` 类型的 `equals` 方法，但是 `Bigram` 的 `equals` 方法的参数并不是 `Object` 类型，因此 `Bigram` 从 `Object` 继承了 `equals` 方法。这个 `equals` 方法测试对象的同一性，就行 `==` 操作符一样。每个 `bigram` 的10个备份中，每一个都与其余的9个不同，因此 `Object.equals` 认为它们不相等，这正解释了程序为什么会打印出 260 的原因。

幸运的是，编译器可以帮助你发现这个错误，但是只有当你告知编译器你想要覆盖 `Object.equals` 时才行。为了做到这一点，要用 `@Override` 标注 `Bigram.equals` ，如下所示：

```
@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

如果插入这个主机，并试着重新编译程序，编译器就会产生一条像这样的错误消息：

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

你会立即意识到哪里错了，拍拍自己的头，恍然大悟，马上用正确的来取代出错的 `equals` 实现（见第8条）：

```
@Override public boolean equals(Bigram b) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

因此，应该在你想要覆盖超类声明的每个方法声明中使用 *`Override`* 注解。这一规则有个小小的例外。如果你在编写一个没有标注为抽象的类，并且确信它覆盖了抽象的方法，在这种情况下，就不必将 `Override` 注解放在该方法上了。在没有声明为抽象的类中，如果没有覆盖抽象的超类方法，编译器就会发出一条错误消息。但是，你可能希望关注类中所有覆盖超类方法的方法，在这种情况下，也可以放心地标注这些方法。

现代的IDE提供了坚持使用 `Override` 注解的另一种理由。这种IDE具有自动检查功能，称作代码检验（*code inspection*）。如果启用相应的代码检验功能，当有一个方法没有 `Override` 注解，却覆盖了超类方法时，IDE就会产生一条警告。如果坚持使用 `Override` 注解，这些警告就会提醒你警惕无意识的覆盖。这些警告补充了编译器的错误消息，提醒你无意识的覆盖失败。IDE和编译器，可以确保你覆盖任何你想要覆盖的方法，无一遗漏。

如果你使用的是 Java 1.6 或者更新的发行版本， `Override` 注解在查找Bug方面还提供了更多的帮助。在 Java 1.6 发行版本中，在覆盖接口以及类的方法声明中使用 `Override` 注解变成是合法的了。在被声明为去实现某接口的具体类中，不必标注出你想要这些方法来覆盖接口方法，因为如果你的类没有实现每一个接口方法，编译器就会产生一条错误消息。当然，你可以选择只包括这些注解，来标明它们是接口方法，但是这并非绝对必要。

但是在抽象类或者接口中，还是值得标注所有你想要的方法，来覆盖超类或者超接口方法，无论是具体的还是抽象的。例如，`Set`接口没有给`Collection`接口添加新方法，因此它应该在它的所有方法声明中包括`Override`注解，以确保它不会意外地给`Collection`接口添加任何新方法。

总而言之，如果在你想要的每个方法声明中使用`Override`注解来覆盖超类声明，编译器就可以替你防止大量的错误，但有一个例外。在具体的类中，不必标注你确信覆盖了抽象方法声明的方法（虽然这么做也没有什么坏处）。

第37条：用标记接口定义类型

标记接口（*marker interface*）是没有包含方法声明的接口，而只是指明（或者“标明”）一个类实现了具有某种属性的接口。例如，考虑 `Serializable` 接口（见第11章）。通过实现这个接口，类表明它的实例可以被写到 `ObjectOutputStream` （或者“被序列化”）。

你可能听说过标记注解（见第35条）使得标记接口过时了。这种断言是不正确的。标记接口有两点胜过标记注解。首先，也是最重要的一点是，标记接口定义的类型是有被标记类的实例实现的；标记注解则没有定义这样的类型。这个类型允许你在编译时捕捉在使用标记注解的情况下要到运行时才能捕捉到的错误。

就 `Serializable` 标记接口而言，如果它的参数没有实现该接口， `ObjectOutputStream.write(Object)` 方法将会失败。令人不解的是， `ObjectOutputStream` API 的创建者在声明 `write` 方法时并没有利用 `Serializable` 接口。该方法的参数类型应该为 `Serializable` 而非 `Object` 。因此，试着在没有实现 `Serializable` 的对象上调用 `ObjectOutputStream.write` ，只会在运行时失败，但也并不一定如此。

标记接口胜过标记注解的另一个有点事，它们可以更加精确地进行锁定。如果注解类型利用 `@Target(ElementType.TYPE)` 声明，它就可以被应用到任何类或者接口。假设有一个标记只适用于特殊接口的实现。如果将它定义成一个标记接口，就可以用它将唯一的接口扩展成它适用的接口。

`Set` 接口可以说就是这种有限制的标记接口（*restricted marker interface*）。它只适用于 `Collection` 子类型，但是它不会添加除了 `Collection` 方法的契约，包括 `add` 、 `equals` 和 `hashCode` 。但是很容易想象只适用于某种特殊接口的子类型的标记接口，它没有改进接口的任何方法的契约。这种标记接口可以描述整个对象的某个约束条件，或者表明实例能够利用其他某个类的方法进行处理（就想 `Serializable` 接口表明实例可以通过 `ObjectOutputStream` 进行处理一样）。

标记注解胜过标记接口的最大优点在于，它可以通过默认的方式添加一个或者多个注解类型元素，给一被使用的注解类型添加更多的信息[JLS，9.6]。随着时间的推移，简单的标记注解类型可以演变成更加丰富的注解类型。这种演变对于标记接口而言则是不可能的，因为它通常不可能在实现接口之后再给它添加方法（见第18条）。

标记注解的另一个优点在于，它们是更大的注解机制的一部分。因此，标记注解在那些支持注解作为编程元素之一的框架中同样具有一致性。

那么什么时候应该使用标记注解，什么时候应该使用标记接口呢？很显然，如果标记是应用到任何程序元素而不是类或者接口，就必须使用注解，因为只有类和接口可以用来实现或者扩展接口。如果标记只应用给类和接口，就要问问自己：我要编写一个还是多个只接受有这种标记的方法呢？如果是这种情况，就应该优先使用标记接口而非注解。这样你就可以用接口作为相关方法的参数类型，它真正可以为你提供编译时进行类型检查的好处。

如果你对第一个问题的答案是否定的，就要再问问自己：我要永远限制这个标记只用于特殊接口的元素吗？如果是，最好将标记定义成该接口的一个子接口。如果这两个问题的答案都是否定的，或许就应该使用标记注解。

总而言之，标记接口和标记注解都各有用处。如果要定义一个任何新方法都不会与之关联的类型，标记接口就是最好的选择。如果想要标记程序元素而非类和接口，考虑到未来可能要给标记添加更多的信息，或者标记要适合于已经广泛使用了注解类型的框架，那么标记注解就是正确的选择。如果你发现自己在编写的是目标位`ElementType.TYPE`的标记注解类型，就要花点时间考虑清楚，它是否真的应该为注解类型，想想标记接口是否会更加合适呢。

从某种意义上说，本条目与第19条中“如果不想定义类型就不要使用接口”的说法相反。本条目最接近的意思是说：如果想要定义类型，一定要使用接口。

第7章 方法

本章要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。本章中大多数内容既适用于构造器，也适用于普通的方法。与第5章一样，本章的焦点也集中在可用性、健壮性和灵活性上。

第38条：检查参数的有效性

绝大多数方法和构造器对于传递给它们的参数值都会有某些限制。例如，索引值必须是非负数，对象引用不能为 `null` ，等等，这些都是很常见的。你应该在文档中清楚地指明所有这些限制，并且在方法体的开头出检查参数，以强制施加这些限制。这是“应该在错误发生之后尽快检测出错误”这一普遍原则的一个具体情形。如果不能做到这一点，检测到错误的可能性就比较小，即使检测到错误了，也比较难以确定错误的根源。

如果传递无效的参数值给方法，这个方法在执行之前先对参数进行了检查，那么它很快就会失败，并且清楚地出现适当的异常（exception）。如果这个方法没有检查它的参数，就有可能发生几种情形。该方法可能在处理过程中失败，并且产生令人费解的异常。更糟糕的是，该方法可以正常返回，但是会悄悄地计算出错误的结果。最糟糕的是，该方法可以正常返回，但是却使得某个对象处于被破坏的状态，将来在某个不确定的时候，在某个不相关的点上会引发错误。

对于公有的方法，要用 Javadoc 的 `@throws` 标签（tag）在文档中说明违反参数值限制时会抛出的异常（见第62条）。这样的异常通常为 `IllegalArgumentException` 、 `IndexOutOfBoundsException` 或 `NullPointerException` （见第60条）。一旦在文档中记录了对于方法参数的限制，并且记录了一旦违反这些限制将要抛出的一样，强加这些限制就是非常简单的事情了。下面是一个典型的例子：

```
/**
 * Returns a BigInteger whose value is (this mod m). This method
 * differs from the remainder method in that it always returns a
 * non-negative BigInteger.
 *
 * @param m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus <= 0: " + m);
    ... // Do the coputation
}
```

对于未被导出的方法（unexported method），作为包的创建者，你可以控制这个方法将在哪些情况下被调用，因此你可以，也应该确保只将有效的参数值传递进来。因此，非公有的方法通常应该使用断言（**assertion**）来检查它们的参数，具体做法如下所示：

```
// Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // Do the computation
}
```

从本质上讲，这些断言是在生成被断言的条件将会为真，无论外围包的客户端如何使用它。不同于一般的有效性检查，断言如果失败，将会抛 `AssertionError`。也不同于一般的有效性检查，如果它们没有起到作用，本质上也不会有成本开销，除非通过 `-ea`（或者 `-enableassertions`）标记（flag）传递给 Java 解释器，来启动它们。关于断言的更多信息，请见 Sun 的教程[Asserts]。

对于有些参数，方法本身没有用到，却被保存起来功以后使用，检验这类参数的有效性尤为重要。例如，考虑第83页中的静态工厂方法，它的参数为一个 `int` 数组，并返回该数组的 `List` 视图。如果这个方法的客户端要传递 `null`，该方法将会抛出一个 `NullPointerException`，因为该方法包含一个显式的条件检查。如果省略了这个条件检查，它就会返回一个指向新建 `List` 实例的引用，一旦客户端企图使用这个引用，立即就会抛出 `NullPointerException`。到那时，要想找到 `List` 实例的来源可能就非常困难了，从而使得调试工作极大地复杂化了。

如前所述，有些参数被方法保存起来供以后使用，构造器正式代表了这种原则的一种特殊情形。检查构造器参数的有效性是非常重要的，这样可以避免构造出来的对象违反了这个类的约束条件。

在方法执行它的计算任务之前，应该先检查它的参数，这一规则也有例外。一个很重要的例外是，在有些情况下，有效性检查工作非常昂贵，或者根本是不切实际的，而且有效性检查已隐含在计算过程中完成。例如，考虑一个为对象列表排序的方法：`Collections.sort(List)`。列表中的所有对象都必须是可以相互比较的。在为列表排序的过程中，列表中的每个对象将与其他某个对象进行比较。如果这些对象不能相互比较，其中的某个比较操作就会抛出 `ClassCastException`，这正是 `sort` 方法所应该做的事情。因此，提前检查列表中的元素是否可以相互比较，这并没有多大意义。然而，请注意，不加选择地使用这种方法将会导致失去失败原子性（failure atomicity）（见第64条）。

有时候，有些计算会隐式地执行必要的有效性检查，但是如果检查不成功，就会抛出错误的异常。换句话说，由于无效的参数值而导致计算过程抛出的异常，与文档中标明这个方法将抛出的异常并不相符。在这种情况下，应该使用第61条中讲述的异常转译（**exception translation**）技术，将计算过程中抛出的异常转换为正确的异常。

不要从本条目的内容中得出这样的结论：对参数的任何限制都是件好事。相反，在设计方法时，应该使它们尽可能地通用，并符合实际的需要。假如方法对于它能接受的所有参数值都能够完成合理的工作，对参数的限制就应该是越少越好。然而，通常情况下，有些限制对于

被实现的抽象来说是固有的。

简而言之，每当编写方法或者构造器的时候，应该考虑它的参数有哪些限制。应该把这些限制写到文档中，并且在这个方法体的开头出，通过显式的检查来实施这些限制。养成这样的习惯是非常重要的。只要有效性检查有一次失败，你为必要的有效性检查所付出的努力便都可以连本带利地得到偿还了。

第39条：必要时进行保护性拷贝

使 Java 使用起来如此舒适的一个因素在于，它是一门安全的语言（**safe language**）。这意味着，它对于缓冲区溢出、数组越界、非法指针以及其他的内存破坏错误都自动免疫，而这些错误却困扰着诸如 C 和 C++ 这样的不安全语言。在一门安全语言中，在设计类的时候，可以确切地知道，无论系统的其他部分发生什么事情，这些类的约束都可以保持为真。对于那些“把所有内存当做一个巨大的数组来看待”的语言来说，这是不可能的。

即使在安全的语言中，如果不采取一点措施，还是无法与其他的类隔离开来。假设类的客户端会尽其所能来破坏这个类的约束条件，因此你必须保护性地设计程序。实际上，只有当有人试图破坏系统的安全性时，才可能发生这种情形；更有可能的是，对你的 API 产生误解的程序员，所导致的各种不可预期的行为，只好由类来处理。无论是哪种情况，编写一些面对客户的不良行为时仍能保持健壮性的类，这是非常值得投入时间去做的事情。

没有对象的帮助时，虽然另一个类不可能修改对象的内部状态，但是对象很容易在无意识的情况下提供这种帮助。例如，考虑下面的类，它声称可以表示一段不可变的时间周期：

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(
                start + " after " + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    ... // Remainder omitted
}
```

乍一看，这个类似乎是不可变的，并且强加了约束条件：周期的起始时间（`start`）不能在结束时间（`end`）之后。然而，因为`Date`类本身是可变的，因此很容易违反这个约束条件：

```
// Attack the internal of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Date();
end.setYear(78); // Modifies internals of p!
```

为了保护`Period`实例的内部信息避免受到这种攻击，对于构造器的每个可变参数进行保护性拷贝（**defensive copy**）是必要的，并且使用备份对象作为`Period`实例的组件，而不使用原始的对象：

```
// Required constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime);
    this.end   = new Date(end.getTime);

    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(start + " after " + end);
}
```

用了新的构造器之后，上述的攻击对于`Period`实例不再有效。注意，保护性拷贝是在检查参数的有效性（见第**38**条）之前进行的，并且有效性检查是针对拷贝之后的对象，而不是针对原始的对象。虽然这样做看起来有点不太自然，却是必要的。这样做可以避免在“危险阶段（window of vulnerability）”期间从另一个线程改变类的参数，这里的危险阶段是指从检查参数开始，知道拷贝参数之间的时间段。（在计算机安全社区中，这被称作**Time-Of-Check/Time-Of-Use**或者**TOCTOU**攻击[Viega01]。）

同时也请注意，我们没有用`Date`的`clone`方法来进行保护性拷贝。因为`Date`是非`final`的，不能保证`clone`方法一定返回类为`java.util.Date`的对象：它有可能返回专门处于恶意的目的而设计的不可信子类的实例。例如，这样的子类可以在每个实例被创建的时候，把指向该实例的引用记录到一个私有的静态列表中，并且允许攻击者访问这个列表。这将使得攻击者可以自由地控制所有的实例。为了阻止这种攻击，对于参数类型可以被不可信任方子类化的参数，请不要使用`clone`方法进行保护性拷贝。

虽然替换构造器就可以成功地避免上述的攻击，但是改变`Period`实例仍然是有可能的，因为它的访问方法提供了对其可变内部成员的访问能力：

```
// Second attack on the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78); // Modifies internals of p!
```

为了防御这第二种攻击，只需修改这两个访问方法，使它返回可变内部域的保护性拷贝即可：

```
// Repaired accessors - make defensive copies of internal fields
public Date start() {
    return new Date(start.getTime());
}

public Date end() {
    return new Date(end.getTime);
}
```

采用了新的构造器和新的访问方法之后，`Period`就真正是不可变的了。不管程序员是多么而已，或者多么不及格，都绝对不会违反“周期的起始时间不能落后于结束时间”这个约束条件。确实如此，因为除了`Period`类自身之外，其他任何类都无法访问`Period`实例中的任何一个可变域。这些域被真正封装在对象的内部。

访问方法与构造器不同，它们在进行保护性拷贝的时候允许使用`clone`方法。之所以如此，是因为我们知道，`Period`内部的`Date`对象的类是`java.util.Date`，而不可能是其他某个潜在的不可信子类。也就是说，基于第11条中所阐述的原因，一般情况下，最好使用构造器或者静态工厂。

参数的保护性拷贝并不仅仅针对不可变类。每当编写方法或者构造器时，如果它要允许客户提供的对象进入到啊内部数据机构中，则有必要考虑一下，客户提供的对象是否有可能是可变的。如果是，就要考虑你的类是否能够容忍对象进入数据结构之后发生的变化。如果答案是否定的，就必须对该对象进行保护性拷贝，并且让拷贝之后的对象而不是原始对象进入到数据结构中。例如，如果你正在考虑使用由客户提供的对象引用作为内部`Set`实例的元素，或者作为内部`Map`实例的键（key），就应该意识到，如果这个对象在插入之后再被修改，`Set`或者`Map`的约束条件就会遭到破坏。

在内部组件被返回给客户端之前，对它们进行保护性拷贝也是同样的道理。不管类是否为不可变的，在把一个指向内部可变组件的引用返回给客户端之前，也应该加倍认真地考虑。解决方案是，应该返回保护性拷贝。记住长度非零的数组总是可变的。因此，在把内部数组返回给客户端之前，应该总要进行保护性拷贝。另一种解决方案是，给客户端返回该数组的不可变试图（immutable view）。这两种方法在第13条中都已经演示过了。

可以肯定地说，上述的真正启示在于，只要有可能，都应该使用不可变的对象作为对象内部的组件，这样就不必再为保护性拷贝（见第15条）操心。在前面的`Period`例子中，值得一提的是，有经验的程序猿通常使用`Date.getTime()`返回的`long`基本类型作为内部的时间表示法，而不是使用`Date`对象引用。他们之所以这样做，主要因为`Date`是可变的。

保护性拷贝可能会带来相关的性能损失，这种说法并不总是正确的。如果类信任它的调用者不会修改内部的组件，可能因为类及其客户端都是同一个包的双方，那么不进行保护性拷贝也是可以的。在这种情况下，类的文档中就必须清楚地说明，调用者决不能修改受到影响的参数或者返回值。

即使跨越包的作用范围，也并不总是适合在将可变参数整合到对象中之前，对它进行保护性拷贝。有一些方法和构造器的调用，要求参数所引用的对象必须有个显式的交接（**handoff**）过程。当客户端调用这样的方法时，它承诺以后不再直接修改该对象。如果方法或者构造器期望接管一个由客户端提供的可变对象，它就必须在文档中明确地指出这一点。

如果类所包含的方法或者构造器的调用需要移交对象的控制权，这个类就无法让自身抵御恶意的客户端。只有当类和它的客户端之间互相的新人，或者破坏类的约束条件不会伤害到除了客户端之外的其他对象时，这种类才是可以接受的。后一种情形的例子是包装类模式（wrapper class pattern）（见第16条）。根据包装类的本质特征，客户端只需在对象被包装之后直接访问它，就可以破坏包装类的约束条件，但是，这么做就往往只会伤害到客户端自己。

简而言之，如果类具有从客户端得到或者返回到客户端的可变组件，类就必须保护性地拷贝这些组件。如果拷贝的成本受到限制，并且类信任它的客户端不会不恰当地修改组件，就可以在文档中指明客户端的职责是不得修改受到影响的组件，以此来代替保护性拷贝。

第40条：谨慎设计方法签名

本条目是若干 API 设计技巧的总结，它们都还不足以单独开设一个条目。综合来说，这些技巧将有助于使你的API更易于学习和使用，并且比较不容易出错。

谨慎地选择方法的名称。方法的名称应该始终遵循标准的命名习惯（见第56条）。首要目标应该是选择易于理解的，并且与同一个包中的其他名称风格一致的名称。第二个目标应该是选择与大众认可的名称（如果存在的话）相一致的名称。如果还有疑问，请参考 Java 类库的 API。尽管 Java 类库中也有大量不一致的地方，考虑到这些 Java 类库的规模和范围，这是不可避免的，但它还是得到了相当程度的认可。

不要过于追求提供便利的方法。每个方法都应该尽其所能。方法太多会使类难以学习、使用、文档化、测试和维护。对于接口而言，这无疑是正确的，方法太多会使接口实现者和接口用户的工作变得复杂起来。对于类和接口所支持的每个动作，都提供一个功能齐全的方法。只有当一项操作被经常用到的时候，才考虑为它提供快捷方式（shorthand）。如果不能确定，还是不提供快捷为好。

避免过长的参数列表。目标是四个参数，或者更少。大多数程序员都无法记住更长的参数列表。如果你编写的很多方法都超过了这个限制，你的 API 就不太便于使用，除非用户不停地参考它的文档。现代的 IDE 会有所帮助，但最好还是使用简短的参数列表。相同类型的长参数序列格外有害。API 的用户不仅无法记住参数的顺序，而且，当他们不小心弄错了参数顺序时，他们的程序依然可以编译和运行，只不过这些程序不会按照作者的意图进行工作。

有三种方法可以缩短过长的参数列表。第一种是把方法分解成多个方法，每个方法只需要这些参数的一个子集。如果不小心，这样做会导致方法过多。但是通过提升它们的正交性（orthogonality），还可以减少（**reduce**）方法的数目。例如，考虑 `java.util.List` 接口。她并没有提供“在子列表（sublist）中查找元素的第一个索引和最后一个索引”的方法，这两个方法都需要三个参数。相反，它提供了 `subList` 方法，这个方法带有两个参数，并返回子列表的一个视图（**view**）。这个方法可以与 `indexOf` 或者 `lastIndexOf` 方法结合起来，获得期望的功能，而这两个方法都分别只有一个参数。而且，`subList` 方法也可以与其他任何“针对 `List` 实例进行操作”的方法结合起来，在子列表上执行任意的计算。这样得到的 API 就有很高的“功能-重量”（power-to-weight）比。

缩短长参数列表的第二种方法是创建辅助类（**helper class**），用来保存参数的分组。这些辅助类一般为静态成员类（见第22条）。如果一个频繁出现的参数序列可以被看作是代表了某个独特的实体，则建议使用这种方法。例如，假设你正在编写一个表示纸牌游戏的类，你会发现，经常要传递一个两参数的序列来表示纸牌的点数和花色。如果增加辅助类来表示一张纸牌，并且把每个参数序列都换成这个辅助类的单个参数，那么这个纸牌类游戏的 API 以及它的内部表示都可能会得到改进。

结合了前两种方法特征的第三种方法是，从对象构建到方法调用都采用 Builder 模式（请见第2条）。如果方法带有多个参数，尤其是当它们中有些是可选的时候，最好定义一个对象来表示所有参数，并允许客户端在这个对象上进行多次“setter”调用，每次调用都设置一个参数，或者设置一个较小的相关的集合。一旦设置了需要的参数，客户端就调用对象的“执行（execute）”方法，它对参数进行最终的有效性检查，并执行实际的计算。

对于参数类型，有优先使用接口而不是类（请见第52条）。只要有适当的接口可用来定义参数，就优先使用这个接口，而不是使用实现该接口的类。例如，没有理由在编写方法是使用 `HashMap` 类来作为输入，相反，应当使用 `Map` 接口作为参数。这使你可以传入一个 `Hashtable` 、 `HashMap` 、 `TreeMap` 、 `TreeMap` 的子映射表（submap），或者任何有待于将来编写的 `Map` 实现。如果使用的是类而不是接口，则限制了客户端只能传入特定的实现，如果碰巧输入的数据是以其他的形式存在，就会导致不必要的、可能非常昂贵的拷贝操作。

对于 *`boolean`* 参数，要优先使用两个元素的枚举类型。它使代码更易于阅读和编写，尤其但你在使用支持自动完成功能的 IDE 的时候。它也使以后更易于添加更多的选项。例如，你可能会有一个 `Thermometer` 类型，它带有一个静态工厂方法，而这个静态工厂方法的签名需要传入这个枚举的值：

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

`Thermometer.newInstance(TemperatureScale.CELSIUS)` 不仅比 `Thermometer.newInstance(true)` 更有用，而且你还可以在未来的发行版本中将 `KELVIN` 添加到 `TemperatureScale` 中，无需非得给 `TemperatureScale` 添加新的静态工厂。你还可以将依赖于温度刻度单位的代码重构到枚举常量的方法中（见第30条）。例如，每个刻度单位都可以有一个方法，它带有一个 `double` 值，并将它规格化成摄氏度。